Supporting Multi-device Interaction in the Wild by Exposing Application State

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorglegt von

Diplom-Informatiker Jonathan Diehl

aus Kiel, Deutschland

Berichter: Prof. Dr. Jan Borchers
Prof. James Eagan PhD

Tag der mündlichen Prüfung: 19. November 2013

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Contents

List of Figures

List of Tables

Preface

Abstract

We are at the verge of living in a world where computing has become ubiquitous. However, ubiquitous computing has not developed as expected, where computing devices are embedded in the things that surround us making them smart. Instead, computing capabilities are accessed ubiquitously through a manifold of small interactive devices that people carry with them at all times and use and combine opportunistically. In consequence, the need to interact with multiple devices arises in unexpected ways, or as called in this thesis "in the wild".

The main goal of this thesis is to raise awareness of the unique properties of multi-device interaction in the wild and the misalignment between these properties and current efforts in academia and industry. To this end, the thesis classifies possible types of multi-device interaction as simultaneous or sequential use towards a common or distinct tasks. To support these types of interaction in the wild, systems must enable the opportunistic rearrangement of devices where transitions are robust and can be performed in ad-hoc situations.

The second part of the thesis explores how application state can serve as a conceptual model for users and designers to enable multi-device interaction in the wild. The concept supplies users with a first-class interactive object representing the state of applications, similar to how the file represents the state of information, which can be manipulated with tools that are separated from the task. It is this separation that allows application state to be used in unexpected situations, making it a good fit for multi-device interaction in the wild.

The final part of the thesis elaborates on how the concept of application state can be integrated into current interactive systems. A simple programming interface was developed that separates state extraction from state sharing: The task applications provide the functionality needed to extract and restore their state into a standardized container, which is then managed and shared through designated state management tools. After describing the state exchange system architecture, the thesis explores how to support legacy applications in implementing state extraction and restoration up to complete automation. There is, however, a trade-off between automating state extraction and providing a semantically meaningful state that can be shared between different applications of the same type to transition tasks between device classes.

Acknowledgements

I want to thank all the people who were somehow involved in the creation process of this thesis, whether it be directly through concentrated discussions or indirectly by supporting and pushing me to make the most out of this thesis. This work would not have been possible without you! In particular, I want to thank the following people who had a grave impact on my work: My advisor Jan Borchers has always supported me in my long journey towards this thesis and contributed greatly by always leading me in the right direction with the right questions asked at the right time. My secondary advisor James Eagan has patiently lent me his expertise numerous times and greatly contributed to the main arguments and structure of the thesis. Joel Brandt has been a great source of inspiration and motivation and eagerly shared his extensive knowledge of scientific and technical practice with me. Ying Zhang, Mario Fraikin, Sören Busch, Stefan Plücken, and Ahsan Nazir contributed to this thesis with the dedicated work on their own theses. Thorsten Karrer, Malte Weiß, and Jan-Peter Krämer have been of great assistance shaping the main contributions of the thesis with their numerous insights. Vera Klautke always stood behind me and pushed me to go further than I thought I could. Finally, I want to thank my parents and my family who have always been there for me. Thank you!

Conventions

The thesis is written in American English. Technical terms or jargon that appear for the first time are in *italics*. Some of the material covered in this thesis has been previously published by me and students that I worked with – these passages always include a reference to the related work.

Chapter 1

Introduction

Users have an increasing number of interactive devices at their disposal

People have access to an increasing number of personal interactive devices during their daily routine. 81% of all German households owned a computer in 2012 (Source: Statistisches Bundesamt [2012]), and 61% of all German employees used a computer at work in 2011 (Source: BITKOM [2011]). At the same time, portable interactive devices such as smartphones and tablet computers are rapidly gaining popularity. 36% of all Germans owned a smartphone in 2012, corresponding to an increase of 8% over 2011 (Source: comScore [2012]). Additionally, 13% of all Germans have purchased a tablet computer since its first wide-spread commercial release in 2010 (Source: BITKOM [2012]). Consequently, a large part of the German population already has regular access to at least two interactive devices, and more are becoming multi-device users every year.

Several independent studies show that early adopters make active use of this device diversity in their daily routine

Several studies have established that early adopters who have regular access to multiple devices employ this device diversity for their everyday tasks. Oulasvirta and Sumari [2007] interviewed and observed mobile office workers after a company-wide introduction of a new smartphone about their usage and management of multiple devices in their daily routine. Dearman and Pierce [2008] interviewed knowledge workers who own on average more than six interactive devices about their strategies to manage these devices for their everyday tasks. Karlson et al. [2009] recorded the times when information workers use their personal computers and their smartphones to determine typical usage patterns and strategies of these two devices. Figure 1.1 illustrates this multi-device usage by example of the desk of a knowledge worker containing several personal computers that are used in combination to work on complex tasks. All of these studies confirm that early adopters make active use of a large diversity of devices to work on the tasks that arise during their daily routine. In particular, users fre-

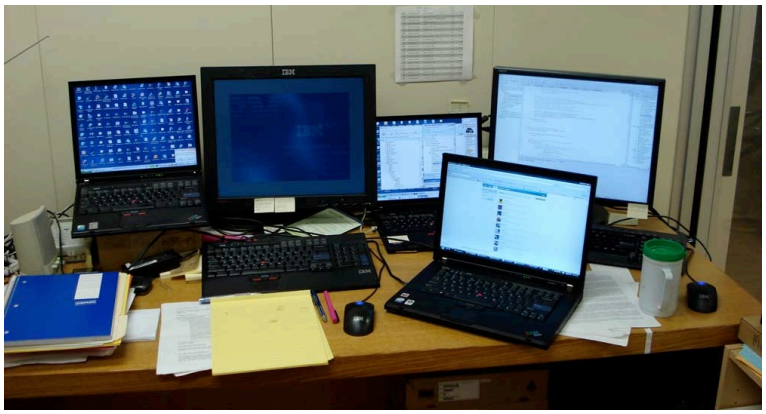

Figure 1.1: One of the test participants' workspace with several laptop computers that are all included in the user's work flow. Picture taken form Dearman and Pierce [2008].

quently switch between devices to adjust to changes of the situation and combine multiple devices to multitask or to work on complex tasks.

Switching between devices allows users to optimize their device usage according to the given situation

All of the above studies reported users frequently switching between devices either to address a different task or to continue the ongoing task on a different device. The most important reason to switch devices that was mentioned by participants was the ability to adjust the device usage to the situation. In this context, the appropriateness of a device for a situation is defined by the suitability of its input and output modalities, the time and effort needed to set up the device, the portability of the device, its social unobtrusiveness, and the personal preference of the user. For example, many participants prefer to check and answer emails on a laptop computer over a smarthpone because of the better input and output capabilities. When in a meeting or in transit, however, the smartphone is preferred because it can be accessed quicker and interacting with it is less disturbing to others. Additionally, users often have to switch between devices because a device lacks the functionality or data to perform a task. For example, many companies have strict security policies that require users to employ specific computers for security-related tasks such as accessing data on the company's file servers. In consequence, switching between devices can improve a user's efficiency and satisfaction and enable work that would otherwise be impossible.

Combining devices improves the users' ability to perform complex tasks or to work on multiple tasks at once

In addition to switching between devices, users also experience benefits from using multiple devices simultaneously. Users can improve their ability to multitask by designating tasks to devices. Typically, users assign secondary tasks to auxiliary devices, allowing them to perform these tasks

without interfering with their primary task on the primary device. For example, many users have a web browser and email client open on a separate device while working on other tasks on their primary device. Additionally, users can divide tasks among multiple devices by assigning each device a role within the task. This division allows users to address complex tasks where any single device cannot provide all the functionality needed for the task. For example, one user reported to use one computer to write source code and another computer to test the source code, allowing the primary computer to stay up-to-date while the secondary computer maintains a consistent state. Either way, combining multiple devices allows users to perform their everyday tasks more efficiently.

Multi-device interaction in the wild describes the evolving behavior of employing multiple devices to work on everyday tasks

As the studies show it is beneficial for users to employ multiple devices in their daily routine. By switching between devices users can adapt and optimize their device usage according to the situation. By combining devices users can improve their ability to multitask and work on complex tasks. Thus, users benefit from being able to use multiple devices sequentially and simultaneously to work on a common or distinct tasks. This benefit increases if these multi-device interactions can be performed seamlessly and spontaneously to allow users to quickly adjust to sudden changes of the situation. In the context of this thesis this evolving behavior is called multi-device interaction in the wild, which is introduced in more detail in chapter 2.

1.1 Problem Statement

The main problem that is addressed in this thesis can be summarized as following:

Problem Statement

> There are no interaction and no system solutions that provide adequate support for multi-device interaction in the wild and consequently hinder users from making optimal use of the available device diversity in their daily routine.

As observed in the studies described above users increasingly have access to multiple devices for their everyday tasks and benefit from using these devices in a coordinated fashion. However, the design of current devices does not consider multi-device relationships and consequently places the burden of coordinating these devices on the user.

Figure 1.2: Cloud sharing services synchronize local copies of digital content by propagating changes between all connected devices "through the cloud".

At the same time, research efforts to improve multi-device interaction are largely focused on collaborative settings, and the solutions cannot be directly transfered to the ad-hoc situation of everyday use. To effectively support multi-device interaction in the wild, interactive devices must allow users to seamlessly and spontaneously change the arrangement of devices and tasks to adapt to sudden changes of the situation.

Cloud services have enabled ubiquitous access to digital content from all of the user's devices

One of the major challenges that was identified in all of the above-mentioned studies was the inability of current devices to make digital content ubiquitously accessible from all of the users devices. Since the studies were conducted, this shortcoming has been somewhat mitigated with the introduction of cloud sharing services. Cloud sharing services synchronize digital content across multiple devices by storing the content locally on each device and synchronizing the local copies through an Internet repository "in the cloud" (see Figure 1.2). There are two different kinds of cloud sharing services: File-based cloud services replicate a part of the user's file system across multiple devices, allowing users to synchronize digital content independent of the editing application. Examples of these services are Dropbox[1], Google Drive[2], and Own Cloud[3]. Application-based cloud services are integrated into special purpose applications and replicate the digital content of these applications across all instances of the application running on various devices. Examples of application-based cloud services are IMAP Email services, Evernote[4] (synchronize notes), or any

[1] http://dropbox.com
[2] http://drive.google.com
[3] http://owncloud.org
[4] http://evernote.com

Mac or iOS application using iCloud[5]. Marshall and Tang [2012] interviewed early adopters of cloud sharing services and confirmed that these services are frequently used and that their users are generally satisfied with the services.

Ubiquitous content access is not sufficient to support multi-device interaction in the wild

Ubiquitous access to digital content enables users to switch devices without losing access to the relevant task data. However, it does not make the switch seamless and provides only limited support for the coordination of multiple devices towards a common task. When switching devices, users must manually recreate the task on the new device by opening and configuring the required task applications. This process can be laborious because users need to find and open the related content in the cloud storage and configure the user interface to match the previous state. Similarly, cloud sharing services can be used to coordinate the usage of multiple devices by synchronizing the digital content underlying a task. However, since the task applications remain unaware of other devices participating in the same task working on synchronized content often yields conflicts that typically must be resolved by the user. As a consequence, ubiquitous information access makes it possible to use multiple devices in the everyday routine, but it does not make the experience seamless or coordinated.

Synchronizing task state through the cloud improves the seamlessness of transitions at the cost of user control

Cloud services can be used to synchronize the interaction state of multiple devices. For example, Amazon Kindle[6] synchronizes the user's books and the furthest page read in these books across multiple devices. This synchronization allows users to switch between devices and continue reading where they left off with little overhead. Similarly, some applications store aspects of the interaction state, like the current cursor position, alongside the edited file. When synchronizing these files across devices, this interaction state is preserved and can result in a more seamless transition between devices. However, these solutions enforce a certain multi-device behavior because the interaction state is always synchronized across all devices. This enforcement can lead to unwanted behavior due to the manifold of different user strategies that exist. For example, always synchronizing all books on multiple devices to the furthest page read is annoying for users who share some of these devices with others, especially if two users are reading the same book at the same time. Similarly, when sharing com-

[5] http://icloud.com

[6] http://amazon.com/kindle

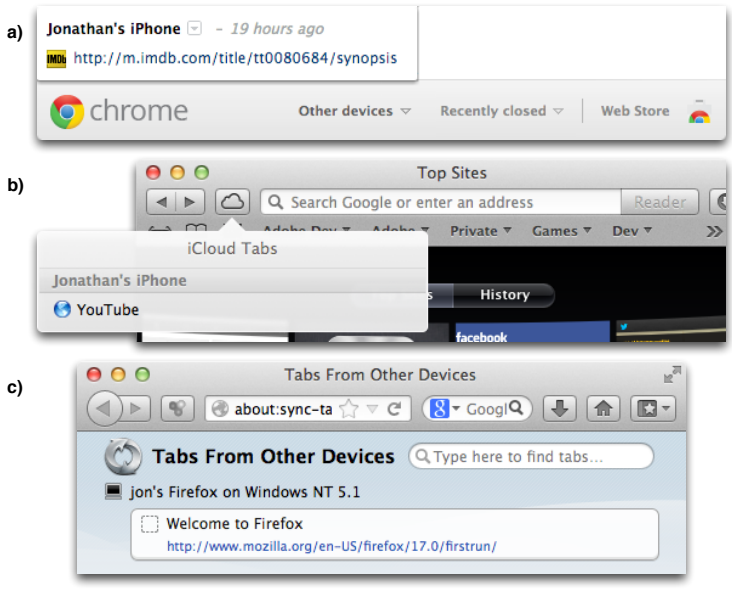

Figure 1.3: Application-level solutions for multi-device interaction suffer from low interaction consistency. For example, tabs from other devices are accessed differently in each web browser: (a) In Google Chrome the tabs are accessed from the bottom of the start screen; (b) in Apple Safari a tool bar button is used; (c) in Mozilla Firefox the "History" menu is used.

plex documents, collaborators might miss some content of the document because the cursor position is not as expected at the beginning of the document. While synchronizing additional task state across devices can be beneficial in some cases, it decreases user control over how information and tasks are distributed across their devices.

Multi-device interactions that are embedded in applications suffer from low interaction consistency and a lack of cross-application compatibility

Other solutions operate at the application level to explicitly offer multi-device capabilities to the user. For example, many web browsers offer to synchronize the settings and the browsing history across all of the users devices. Users can then inspect and open the browsing tabs that are currently open on all connected devices. Using this mechanisms, users can almost seamlessly transition a web browsing task from one device to another by selecting the desired tab from a list of open tabs on the target device. However, these application-level solutions come with several drawbacks for the user. First, users cannot rely on all applications to support multi-device interaction. This means that whenever they are confronted with an application that does not support multi-device interaction, they will be forced to use another application or abstain from using multiple devices while working with that application. Second, the multi-device interactions provided by applications are not consistent across different applications and devices. In above example, open tabs on remote devices are accessed differently in all major web browsers that support

this feature as illustrated in Figure 1.3. Finally, multi-device interactions are constrained to a set of specific applications, typically from the same vendor. None of the web browsers from the previous example allow accessing the open tabs from one of the other web browsers. This restriction prevents users from selecting the optimal set of applications on their devices and can lead to some devices being excluded from multi-device interaction due to a lack of software support.

Most research efforts in the area of multi-device interaction focus on collaborative settings and augmented environments, not the daily routine

There is a large body of HCI literature concerned with multi-device interaction, which is discussed in more detailed in chapter 3. However, most research projects in this area focus on collaborative settings in augmented environments called Active Spaces or Multi-displays Environments. These augmented environments provide a well-defined physical space where multi-device interaction is supported. When the users leave this space, multi-device interactions are typically no longer available. Due to this restriction the interaction concepts and system architectures developed in this area are very hard to transfer to the everyday situation where the need to conduct multi-device interaction can arise anywhere. To support multi-device interaction in the wild, a solution must be able to adapt to a frequently changing setup of devices and infrastructure and allow users to manage multiple device in an ad-hoc way.

1.2 Approach

The file enables users to interact with content independent of task applications, including the ability to share content between devices

As argued above sharing content across multiple devices works relatively well compared to sharing tasks across devices. The major technical difference between content and tasks in this context is that content can be made persistent in the form of files, while tasks cannot. Users can generally store any content that they work on in files independent of the applications used to edit the content. These files can then be organized and shared using a variety of file manager applications. In consequence, the file has become a first-class interactive object that allows users to manage their content independent of specific task applications. In computing, a first-class object is an entity that can be constructed at runtime and passed between different computing methods. Applying this concept to interactive objects, i.e., objects that can be manipulated by the user, yields the

Definition: *First-class interactive object*

following definition: A first-class interactive object is an entity that users can construct and pass between different processes including those running on different devices at runtime.

The thesis explores turning the state of applications into first-class interactive objects

The approach pursued in this thesis is to turn the state of applications into first-class interactive objects and explore how these objects enable multi-device interaction in the wild. Turning application state into first-class interactive objects allows users to store the state of individual applications at runtime and transfer the state between different devices. The stored application state includes everything that is needed to reconstruct the application at a later point in time, including the complete state of the user interface and any content that is being accessed. Based on this conceptual model designers can create multi-device interaction techniques that operate on application state independent of any task applications. Similar to file management, diverse applications can be created to manage application state, including all forms of sharing that exist for files today. Users can then chose the most appropriate of these state management applications for any given situation and task to transition and synchronize application state between devices.

Application state enables support for multi-device interaction in the wild

Making application state accessible as a first-class interactive object enables the development of interaction techniques that address the challenges of multi-device interaction in the wild. Tasks can be migrated between devices by transferring the state of all applications involved in the task between the devices. This task migration via application state can be done as long as the two devices can communicate with one another and the transferred applications run independent of the original devices, allowing the opportunistic rearrangement of tasks and devices that is core to multi-device interaction in the wild. At the same time, task execution can be coordinated across multiple devices by synchronizing the state of the involved applications across these devices. The synchronized state provides task applications with a communication channel that creates awareness of other participating devices and can be used to assign and coordinate different roles among these devices. The concept of application state and how it can be applied to the setting of multi-device interaction in the wild is described in detail in chapter 4.

The second part of the thesis explores how application state can be exposed in current interactive systems

In addition to exploring the benefit of application state as a conceptual model for users and designers, the thesis also explores how application state can be exposed in modern operating systems to allow the tight integration of multi-device interactions into these systems. To this end, a system architecture that integrates support for application state into common operating systems is described in chapter 5. The proposed system architecture enables the separate development of state extraction capabilities in task applications and multi-device interaction techniques, which communicate with one another through state objects. Chapter 6 finally explores how application developers can be supported in integrating support for state extraction into their applications. Using advanced features available in modern operating systems, this integration can be fully automated. However, this automation comes at the cost of reducing the semantic structure of the extracted state. In summary, automated methods to extract state are promising for integrating support for state exchange into legacy applications, but cannot replace the extraction mechanism of a skilled application developer.

1.3 Thesis Statement

The main contributions of this thesis can be summarized in the following three statements:

We need new interaction and system support for multi-device interaction in the wild

S1 The thesis describes the properties of multi-device interaction in the wild to raise awareness of the mismatch of current trends in the industry and academia in contrast to this evolving behavior.

Application state is an interaction concept that addresses multi-device interaction in the wild

S2 The thesis introduces application state as a conceptual model where the state of ongoing tasks is made available as first-class interaction objects that can be transferred between devices to enable multi-device interaction in the wild.

Application state can be integrated into current interactive systems

S3 The thesis describes a system architecture that integrates the concept of application state into common operating systems and explores different approaches to support this integration.

These statements leads to three research challenges: identify challenges of multi-device interaction in the wild, describe and analyze the interaction concept, describe and discuss the integration

These statements describe three individual research challenges that must be addressed: First, the behavior of multi-device interaction in the wild must be identified and the underlying challenges for interaction solutions must be developed. Second, the concept of application state must be described with an argument that explains how it addresses the identified challenges of multi-device interaction in the wild. Finally, the integration of the application state concept into existing interactive systems must be described and analyzed. These challenges are reflected in the following set of research questions, which will be addressed in this thesis:

RQ1 What is multi-device interaction in the wild? What are the unique challenges of providing support for multi-device interaction in the wild? How do current approaches fail to address these challenges?

RQ2 How can application state be exposed to the user as a first-class interactive object? What operations allow users to migrate and coordinate tasks across multiple devices via application state? How does this concept support multi-device interaction in the wild?

RQ3 How can the concept of application state be integrated into common interactive systems? What aspects of modern operating systems can be exploited to simplify the integration of application state?

RQ1 sets the frame of this thesis by describing how users employ multiple devices for their everyday tasks

By answering the first research question the desirable user behavior of using multiple device for everyday tasks is described and differentiated from multi-device interaction as addressed in the literature. Chapter 2 describes a classification of multi-device interaction in the wild, which is used to derive the unique challenges of this behavior. Chapter 3 then summarizes related work from the literature that addresses multi-device interaction in general and analyzes the differences between the investigated scenarios and multi-device interaction in the wild.

RQ2 addresses the interaction solution proposed in this thesis and how it solves the challenges of multi-device interaction in the wild

The answer to the second research question represents the main contribution of this thesis: A novel interaction concept that addresses the challenges of multi-device interaction in the wild is introduced and evaluated. The introduction of the interaction concept is done in abstract terms to guarantee platform independence. The concept is then evaluated by presenting the results of a workshop where

students designed new ways of interacting with multiple devices based on application state. Furthermore, four prototype systems are described that apply the concept of application state to different usage scenarios. This research question is addressed in chapter 4.

RQ3 investigates how the concept of application state can be integrated into current interactive systems and how this integration can be simplified for the application developers

The last research question pushes the focus of this thesis a bit further into the technical domain by investigating how the concept of application state can be enabled in current interactive systems. To this end, chapter 5 describes a system architecture that separates state extraction and restoration from the actual interaction techniques based on application state. The chapter describes how this system architecture is integrated into a typical interactive systems and how it fulfills all of the requirements to enable support for multi-device interaction in the wild. Chapter 6 then analyzes how certain system functions can be exploited to automate the process of integrating state extraction and restoration into legacy applications. This automation is demonstrated with a prototype, and the trade-off between automatism and semantic structure is discussed.

1.4 Thesis Overview

Chapter 2: Multi-device Interaction in the Wild

The next chapter introduces multi-device interaction in the wild as the opportunistic combination of multiple devices to address tasks in the everyday routine. The behavior is developed based on the findings of several studies from the literature which are summarized. Finally, the novel challenges for the design of interactive systems aim at supporting this evolving behavior are discussed.

Chapter 3: Related Work

The third chapter describes previous solutions for multi-device interaction that are related to the work presented in this thesis. The chapter describes existing interaction techniques and system architectures for multi-device interaction and analyzes their application for multi-device interaction in the wild. Based on this analysis, the need for a new approach that is tailored to the unique challenges of multi-device interaction in the wild is developed.

Chapter 4: Interacting with State

The forth chapter introduces the interaction concept of exposing application state as a first-class interactive object and discusses how it addresses the challenges of multi-device interaction in the wild. The interaction concept is evaluated in two ways: First, the results of a workshop to

uncover new interaction techniques based on application state show that application state can be used effectively as a design tool. Second, the design and evaluation of several prototype implementations that apply the concept of application state in diverse situations demonstrate the applicability of the conceptual model in realistic situations.

Chapter 5: The State Exchange Architecture

The fifth chapter explains how exposing application state can be integrated into common interactive systems. The chapter develops the requirements to implement the state operations from the previous chapter and describes a system architecture that extends interactive systems to implement these requirements. A prototype implementation of this system architecture demonstrates that common interactive systems can be extended with the necessary functionality to make application state a first-class interactive object. Finally, the validity of the system architecture is analyzed by verifying the fulfillment of the initial requirements and discussing the technical limitations of the approach.

Chapter 6: Integrating State Exchange into Legacy Systems

The sixth chapter explores how to expose the state of legacy applications to allow them to participate in multi-device interaction. In some cases, the state of a legacy application can be exposed automatically based on the knowledge of the application that is inherent in current operating systems. The chapter describes what frameworks can be used to extract this knowledge and how the knowledge can be used to expose the application's state. Additionally, legacy applications can be supported by giving third-party developers the means to implement custom support to expose the application's state. Both approach are demonstrated with a prototype on Mac OS X and the limitations and challenges of the approaches are discussed.

Chapter 7: Conclusion

The final chapter concludes the thesis by discussing the potential impact of exposing state on our daily usage of interactive devices. Finally, the chapter describes several opportunities for future work to improve support for multi-device interaction in the wild.

Chapter 2

Multi-device Interaction in the Wild

Previous multi-device interaction research has focused on collaborative environments

In the past, most of the HCI research concerning multi-device interaction has focused on collaborative meeting-room scenarios. In this context, multi-device interaction takes place in a multi display environment (MDE), which is a dedicated physical space to support synchronous, co-located collaboration. MDEs typically provide multiple shared interactive devices, such as large wall screens, tabletop displays, or other embedded devices. In addition, most MDEs allow users to integrate their own devices into the room's infrastructure. All devices in the room are then interconnected to form a coherent virtual workspace that matches the physical workspace.

A new form a multi-device interaction is evolving: users employ multiple devices in their daily routine

With the wide-spread adoption of mobile interactive devices, a new form of multi-device interaction is evolving. Users are no longer confined to augmented environments to interact with multiple devices. Instead, they do so opportunistically with the devices available to them in any given situation. For example, Oulasvirta and Sumari [2007] have found that mobile workers frequently switch between a smartphone and a desktop computer to adjust to situational changes: When mobility and quick access to basic functionality is essential, the smartphone is preferred. When efficient input and output capabilities are important, the desktop computer is used.

Multi-device interaction is not well-supported by everyday interactive devices

Despite this incipient adoption of multi-device usage users face many challenges when they are combining devices in their daily routine. For example, transitioning tasks between devices requires users to recreate the task manually on the new device, even if the underlying data is synchronized: When transitioning devices while writing an email the user must save the current text as a draft, open the email client on the second machine, find and open the text in

the drafts folder, and browse to the appropriate position in the text. A better solution requires a task to migrate between devices seamlessly, i.e., while preserving all work state. However, solutions from the literature that enable this kind of seamless multi-device interaction do not transfer well to the everyday situation because they are typically embedded in a specific environment.

Chapter outline

This chapter first describes several studies from the literature that report how users with access to multiple devices use this device diversity in their daily routine. Based on these studies the evolving user behavior of multi-device interaction in the wild is developed. The next chapter then analyzes existing multi-device interaction techniques and system architectures in light of this new behavior.

2.1 Understanding Multi-device Interaction in the Wild

Several studies have been published in the literature that examine multi-device behavior of early adopters in everyday situations. Oulasvirta and Sumari [2007] found that mobile information workers switch frequently between devices. Dearman and Pierce [2008] discovered that knowledge workers own many diverse devices and assign different roles to them. Karlson et al. [2009] uncovered typical usage patterns of smartphone and computer usage including many transitions between the two devices. Karlson et al. [2010] listed barriers to mobile task flow and strategies to continue suspended tasks including transitioning the task to a different device.

2.1.1 Mobile Kits and Laptop Trays: Managing Multiple Devices in Mobile Information Work

Study goals: understand how mobile information workers use and manage multiple devices

Oulasvirta and Sumari [2007] conducted a field study of multi-device usage among mobile information workers with two goals: First, they wanted to investigate how workers use multiple devices and what benefits they see in having multiple devices. Second, they wanted to understand how workers handle their multi-device management, i.e., the activities needed to prepare and maintain multiple devices for productive work.

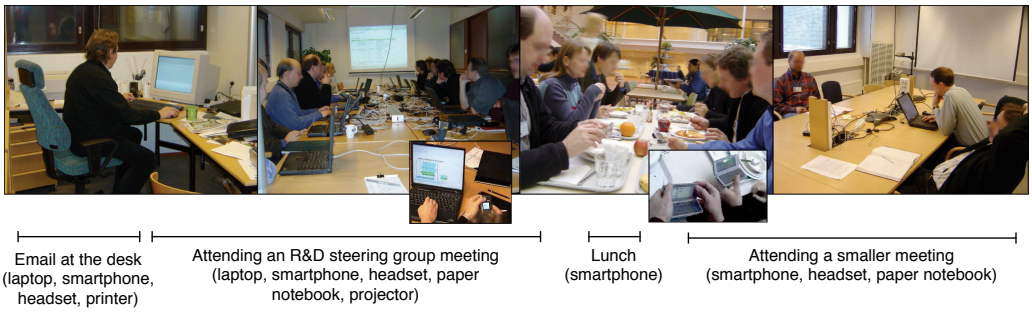

Figure 2.1: Typical device usage during the daily routine of a mobile information worker at Nokia. Picture taken from Oulasvirta and Sumari [2007].

Study setup: eleven mobile workers from Nokia were interviewed about their multi-device usage and two of them were observed closely during a regular work day

The study took place in 2005 at Nokia in Finland after a company-wide introduction of a business smartphone. Eleven participants holding various positions at Nokia (executive, customer-facing employee, subject matter expert) were recruited for the study. All of the participants regularly employed multiple devices for their work, one of which being the newly introduced smartphone. Data was collected by interviewing the participants about their multi-device usage. In addition, two participants were chosen based on the interview results for a close observation of one work day (seven hours).

Mobile information workers switch frequently between devices for various reasons

The findings of the study confirm that users often switch between devices during their work. Device switching happens in various situations (when attending a meeting, going to lunch, sitting at a workstation, in transit) and with various devices (laptop, mobile phone, smartphone, headset, projector, paper). Figure 2.1 demonstrates this behavior with the log of one of the closely observed study participant's daily routine. The participants expressed various social, personal, and work-related reasons for switching devices:

- Depending on the task some devices are preferred over others because of more suitable input or output modalities (larger screen, keyboard).
- Devices that take only little time and effort to set up and thus become ready for use quicker are preferred for simple tasks.
- For some tasks no single device has all the functionality needed to accomplish the task.

- Some devices serve as fall-back devices for important tasks.
- Devices might be preferred or avoided for personal reasons (a tendency to forget a device).
- In social situations a device might be preferred because interacting with it is acceptable, while interacting with another device is not (mobile phone vs. laptop computer in a meeting).
- Organizational practices impose restrictions on which devices can be used for a given task.

Using multiple devices imposes extra work on users in the form of multi-device management

When dealing with multiple devices, users must decide which device to use for each task and prepare it for the task. User strategies that cope with this process revolve around three central aspects: reduce the effort of managing multiple device; have the right data and functionality available at the right time; align these efforts with their context of use.

There is a lack of commercial and academic support for the multi-device usage observed in the study

Oulasvirta and Sumari [2007] conclude that current systems and efforts in HCI are misaligned with the behavior observed in their study. Even though today's systems provide very limited support for multi-device interaction, workers still manage and use multiple devices for their daily work. Future efforts should focus on improving support for multi-device interaction by reducing the overhead of having multiple devices (data synchronization and device setup), reducing the difficulty of planning future device usage, propagating information usage across devices, and sharing resources across devices.

2.1.2 It's on my other Computer!: Computing with Multiple Devices

Study goal: understand how users employ multiple devices

Dearman and Pierce [2008] studied how users make use of multiple devices for their personal and work activities. They were especially interested in activities that span multiple devices and how users cope with the challenges of managing information and activities across these devices.

Study setup: 27 researchers were interviewed about what devices they own and how they use them

The study was conducted in 2007 with 27 participants from IBM Research and Stanford university. The participants were interviewed in their regular work environment using semi-structured interviews. First, each participant was asked to list all devices where the participant is the primary

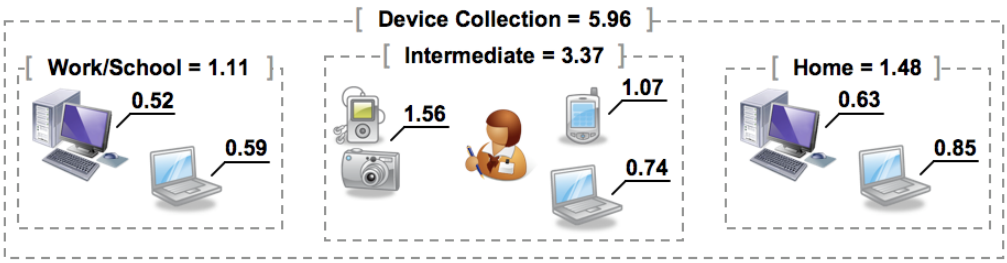

Figure 2.2: On average, participants reported using different devices at work, at home, and when transitioning between places. Picture taken from Dearman and Pierce [2008].

user. Then the configuration of these devices and their interrelationships when working on tasks were inquired.

Users employ on average almost six devices for various reasons

Based on the results Dearman and Pierce [2008] could establish that users employ multiple devices to work on their everyday tasks. On average, each user was the primary users of 5.96 devices consisting of one computer at work, one or two at home, and three to four intermediate devices like smartphones, digital cameras, and laptop computers (see Figure 2.2). The participants reported the following reasons for having this device diversity:

- Different devices have different form factors that make them more or less suitable depending on the situation.
- Portable devices allow users to freely choose the setting where to perform a task.
- Different devices have different levels of efficiency (set-up time, task completion time) such that the optimal device for a given task varies with the situation.
- Some users prefer to separate work from personal activities by employing different computers.
- Some tasks are bound to specific devices for technical reasons.
- Some devices are highly specialized in a single task (digital camera), making them very efficient at that task.
- When transitioning to a new device, the old device is sometimes kept around because it is not trivial to migrate all data to the new device.

When devices are combined, they take on different roles within a workflow

The study participants reported that they often spread out tasks across multiple devices and assign a designated role to each device. These roles can be highly interconnected such that the users interact with several devices on the same data to accomplish the task. Other roles are dedicated to perform secondary tasks that are related to the main task but do not require a tight connection of the devices.

Multi-device interaction is hindered by information dispersion

The main inhibitor for effectively combining multiple devices for a task that was reported by the study participants is the difficulty of keeping information in synchronization across these devices. Even though users had multiple strategies to cope with this challenge, none of them succeeded to the users' full content. Specifically, users complained that interaction histories cannot be shared between devices. Automatic synchronization services were not employed by the participants because they do not trust these services.

Information and activities are tied to devices which are unaware of their roles

Dearman and Pierce [2008] conclude that the design of current devices exacerbates multi-device usage in two ways: First, devices tie information and activities to the device making it hard to transition between devices during an ongoing activity. Second, devices are unaware of a multi-device setup and their role in this setup, leaving all coordination activities between these devices to the user.

2.1.3 Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker

Study goals: understand how users perform tasks that span smartphones and other devices

Karlson et al. [2009] studied how information workers use their smartphones and desktop computers in a typical day. To this end, they extracted patterns of smartphone and desktop usage from a record of all activities performed on either device.

Study setup: logging study of 16 information workers with 4 follow-up interviews

The study was performed in 2008 with 16 information workers from a technology company. All participants installed special software on their smartphones and desktops, which logged the current activity (window titles and applications) and user activity (keyboard and mouse). After gathering log data for 5-30 days four of the participants who demonstrated varying usage patterns were interviewed in their usual work environment. During these interviews the participants were questioned about the ac-

tivities that occurred in the logs with a special focus on interleaved and concurrent smartphone-desktop usage.

Email was the most employed activity on both devices

The analysis of the logs revealed that the most used application on both the desktop and the smartphone was the corporate email client. On the desktop other frequent activities were file browsing and web browsing. On the smartphone email was done substantially more than any other activities.

Usage patterns vary among users with several device transitions happening during the day

The participants demonstrated a wide range of different behavior when interacting with their desktops and smartphones. These differences range from using the smartphone almost exclusively to using the desktop almost exclusively. At the same time, several common patterns were identified that reflect frequent device transitions during the day. For most users the smartphone was the first device used in the morning and the last device used at night, and the desktop was used primarily during the day. Additionally, many users switched from the desktop to the smartphone before lunch and back to the desktop afterwards. In the interviews the following reasons for preferring the smartphone over a desktop were uncovered:

- Smartphones are always with the user and provide continuous access to important activities (email).
- Smartphones are immediately ready for use.
- Smartphones can serve as an ultra-portable substitute for desktops.

Transitioning between desktop and smartphone is hindered by a lack of information access

When switching between smartphone and desktop, users did not express the desire to continue a previously begun activity on the new device. Instead, users desired ubiquitous access to their data from all devices to improve their experience with multiple devices.

2.1.4 Mobile Taskflow in Context: A Screenshot Study of Smartphone Usage

Study goal: understand barriers of mobile task flow and user strategies that address these barriers

In a subsequent study Karlson et al. [2010] focused on mobile users with two primary goals: First, they investigated what kind of barriers users encounter that prevent the completion of a mobile task. Second, they inquired user strategies to follow up with these tasks at a later time and whether users were content with their strategies.

Study setup: 24 participants took and annotated screen shots of task disruptions on their smartphones

The study took place in 2009 with 24 participants who were recruited from a software company. Half of the users were iPhone users, and the other half were Windows Mobile Pocket PC users. They were asked to capture all disruptions they encountered on their smartphones during a typical day by making a screen shot. At the end of the day all users were asked to annotate these screen shots and explain the cause of the disruption, how they followed up on the disrupted task, and how frustrating the overall experience was.

Many different types of barriers can impede mobile task flow

The study uncovered the following barriers to mobile task flow:

- The smartphone lacks functionality needed for the task.
- The smartphone's screen is not sufficient for the task.
- The task could not be completed due to network problems.
- The task was too complex to finish on the smartphone.
- The task requires too much time or work for too little benefit.
- Environmental factors prevented the task completion.
- The smartphone's input capabilities were insufficient.

Some of these barriers like missing functionality and network problems are likely an effect of the relative infancy of smartphone technology. Other barriers, however, are caused by aspects that are inherent in mobility such as environmental factors, device properties, and task complexity.

The experience of following-up a task on a different device suffers if there is no cross-device task migration support

The most common method to follow-up on a disrupted task was to finish the task on a desktop computer. For email activities, following up was not a major source of frustration because users can access their email data from all devices allowing a more or less seamless transition between the devices. Other activities that lack this kind of cross-device support show higher frustration ratings when transitioning between devices to finish a disrupted task. Karlson et al. [2010] conclude that users transition tasks between devices for strategic reasons and that these transitions should be supported by more activities on the smartphone and the desktop.

2.1.5 Summary

Oulasvirta and Sumari [2007]: users frequently switch between devices for various reasons employing various strategies

Oulasvirta and Sumari [2007] were the first to establish that users frequently switch devices during a typical day. They uncovered several user benefits that motivate the use of multiple devices and various user strategies for selecting the most appropriate of all available devices. In their conclusion, they argue for better synchronization of information between devices including device setup (available data and applications) and task information (email flags).

Dearman and Pierce [2008]: users have multiple devices for various reasons, devices are assigned roles when used in concert

Dearman and Pierce [2008] confirmed that users employ several devices during their everyday routine and extended the list of user benefits for having multiple devices. In addition, they observed users designating specific roles to specific devices to combine them for a complex task. To support this behavior, they suggested making devices aware of their roles and interrelationships with other devices and untying information and activities from devices.

Karlson et al. [2009]: users employ different patterns for smartphone and desktop usage

Karlson et al. [2009] observed several distinct patterns of smartphone and desktop usage, ranging from almost exclusive use of either device to frequent switching between the devices. When switching between devices, ongoing tasks are rarely continued, which is likely caused by the high overhead of task migration in current devices.

Karlson et al. [2010]: there are many barriers to mobile task flow, which often cause users to finish their tasks on a different device

Karlson et al. [2010] examined typical barriers that prevent users from completing a task on the smartphone, which are often inherent in the mobility of the smartphone and thus not affected by new technology. The most common strategy to follow up on a disrupted task was to finish the task later on a desktop. However, this transition between smartphone and desktop was a high source of frustration, as many activities do not support a seamless migration between devices.

Cloud sharing services have enabled seamless data synchronization across devices

All of the above studies emphasize the need for better data synchronization across devices. By now, advances in cloud synchronization have addressed this need to some extent. Marshall and Tang [2012] verified that early adopters frequently use these services and are very satisfied with them. However, ubiquitous data access is not sufficient to support all aspects of multi-device interaction. In particular, it does not allow users to seamlessly switch between devices during an ongoing task, and it does not make devices aware of their roles when used in combination with other devices.

Users employ multiple devices sequentially

All studies found that users switch frequently between devices, or in other words, they employ multiple devices sequentially. These switches can occur between tasks (the new device is used for a new task) or during ongoing tasks (the new device is used for the same task). Especially in the latter case, switching devices can cause a large overhead for the user to recreate the work state of the task on the new device.

Users employ multiple devices simultaneously

At the same time, the studies have found that users also use a device, while another device is still operating, or in other words, they employ multiple devices simultaneously. Again, users can employ these devices simultaneously for distinct tasks (each device is used for a separate task) or for the same task (all devices operate on the same task). In both cases today's systems are ignorant of their roles in this multi-device setup: For distinct tasks devices running in the background are unaware of their status and the primary task, such that they demand attention at inappropriate times. For a common task the coordination of devices is typically left to the users, creating a significant task management overhead for them.

User behavior is evolving from employing a single device to employing multiple devices for their daily routine

The studies show an evolution of user behavior in their everyday routine from employing a single interactive device to employing multiple devices. In the scope of this thesis, this new behavior is called multi-device interaction in the wild. The remainder of this chapter describes this behavior in detail and develops guidelines for the evaluation and design of interactive systems that support this behavior.

2.2 Multi-device Interaction in the Wild

Users employ multiple devices in their everyday routine by combining them opportunistically to work on distinct or common tasks

Throughout the day users strive to use the most appropriate of the available devices depending on the task, situation, and personal preference to accomplish their goals. The choice of devices can be any combination of the available devices, which may be used to each work on an individual task or to work on a common task together. When interacting with the chosen devices, users may do so sequentially (move attention from one device to another) or simultaneously (split attention between devices). This setup is constantly challenged by the changes of device availability, the task, and the general situation, resulting in frequent changes of the employed device combinations.

The most appropriate device is selected by assessing the properties of all available devices in the context of the current situation and task and the cost of transitioning against each other. The most distinguishing properties of a device are rooted in the class of the device. The following is a list of device classes that are typically encountered in a modern workplace and their properties. See Table 2.1 for an overview of these properties.

The most appropriate device(s) are selected depending on the situation and task

- The desktop computer provides one or more large screens, a full-size keyboard, and a mouse for precise and efficient input and output. It can be used to work collaboratively on a task in small groups. Using a desktop computer during on ongoing conversation is typically not well-received.

Desktop computer: most efficient input

- The laptop computer provides smaller and less efficient input and output modalities than the desktop computer. In return, the laptop computer is portable and can be carried with the user but not used efficiently while being carried. It can be used in collaborative settings like a desktop computer but not as efficiently. Like the desktop, the laptop computer interrupts social interactions.

Laptop computer: portable with good input

- The tablet provides a medium-sized screen with touch input, which is sufficient for many tasks but not suitable for complex tasks such as extended text entry. It is highly portable and well-suited to be used in mobile situations. It can be used collaboratively in small groups by passing it around or interacting with it from multiple sides. Due to its smaller size and collaborative features, it is somewhat acceptable to use during social interactions.

Tablet: usable while mobile

- The modern smartphone provides a small screen with touch input. The small screen is sufficient for simple tasks but lacks screen space and efficient input for moderate to complex tasks. It is the most portable device, designed to be always carried with the user and used while mobile. Because of its small size, it is not suitable for collaboration. At the same time, it does not raise much attention and thus is quite acceptable to be used during ongoing social interactions.

Smartphone: always carried with the user

- The wall screen or projector provides a very large but typically low-resolution screen. Even if the screen is augmented with touch capabilities, interacting with

Wall screen / projector: largest output, most appropriate for collaboration

Device Class	Output	Input	Mobility	Collab.	Social accept.
Desktop computer	$+$	$++$	$--$	0	$--$
Laptop computer	0	$+$	0	$-$	$--$
Tablet	0	0	$+$	$+$	0
Smartphone	$--$	$-$	$++$	$--$	$+$
Wall screen	$++$	$--$	$--$	$++$	$--$
Tabletop screen	$++$	$-$	$--$	$++$	$--$

Table 2.1: The device classes that today's users are confronted with offer various properties, making them more or less appropriate depending on the situation.

a large screen is strenuous, inefficient, and imprecise. Instead, it is designed for collaborative use, especially to visualize large amounts of data. Due to its large size, interacting with a large screen is very disrupting to an ongoing social interaction.

Tabletop screen: large output with better input than the wall screen

- The interactive tabletop screen provides a large horizontal screen with touch capabilities. It is less strenuous to interact with a large horizontal surface than a vertical surface, but table input is nevertheless inefficient and imprecise compared to other devices. Similar to wall screens, tabletop screens are designed for collaborative use and typically very distractive during ongoing social interactions.

Devices can be shared among multiple users, which imposes additional challenges for the design of these devices

Some of these devices are shared among multiple users. Allowing shared devices to seamlessly integrate themselves into multi-device interaction in the wild just like personal devices introduces additional challenges for the design of these devices: First, user data and applications should be accessible from the shared device without imposing a security risk on the user. Second, users should be able to pick up and interact with a shared device without a lengthy setup process. Similarly, a shared device should reset itself when it is left behind. It is important to note that all devices may be used as shared devices, independent of the device class. Therefore, it is generally not sufficient for the design of a device to focus only on single user operation.

The cost of transitioning includes the effort to set up a device, access data and applications, and configure the device's appearance

The cost of transitioning is determined by the effort it takes to set up the new device to start, continue, or participate in the task. This effort includes the effort needed to set up the device itself, access the relevant data and applications on the device, and configure the device's appearance to match the current state of the task. By reducing the cost of transition, users are encouraged to switch devices more often

resulting in a better match of task, situation, and employed devices.

Devices are used sequentially and simultaneously

The chosen devices can be used sequentially or simultaneously. During sequential operation the user stops interacting with a device and starts interacting with another device. This transition can be immediate or delayed (occur after a pause where other interactive devices may be used). After the transition the user is no longer aware of the original device. During simultaneous operation the user sets up multiple devices to operate in parallel. The user does not necessarily interact with the devices in parallel but instead switches attention between the devices. The difference to sequential operation is that the user has intentionally set up multiple devices and is always aware of these devices and their interplay.

Devices are used to work on a common or distinct tasks

At the same time, the chosen devices can be used to work on a common task or to work on multiple distinct tasks. When working on a common task, all participating devices must be coordinated to work on the task in concert. This typically requires that task activities are synchronized across all devices and the effect of changes from one device is propagated to all other devices. On the other hand, when working on distinct tasks, the use of each device is separated from all other devices. Actions performed on separated devices should have no impact on each other. Nevertheless, separated devices can still benefit from being aware of the overall setup. For instance, devices can suppress potentially distracting messages while users are focused on a different task that is executed in parallel.

The multi-device interaction matrix classifies multi-device interaction in the wild as the sequential or simultaneous use of multiple devices towards a common or distinct tasks

This device usage can be expressed as a 2×2 matrix (see Figure 2.3): The first dimension of the matrix expresses the timing of device usage, which is either sequential or simultaneous. The second dimension of the matrix expresses the task affinity of the device usage, which can be to perform distinct or common tasks on the devices. The following list describes the four possible modes of multi-device interaction expressed by this matrix:

Sequential use, distinct tasks

- Devices are used sequentially to perform distinct tasks. In this mode the user stops working on a task on one device and starts working on another task on a new device. In consequence, the user transitions between devices and tasks at the same time and after the transition the original device is no longer in use. An

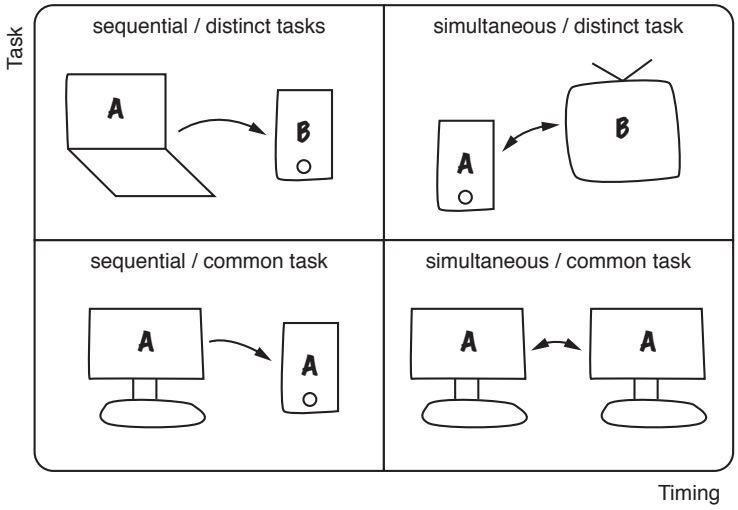

Figure 2.3: The multi-device interaction matrix describes the different ways of employing multiple devices in the wild: Users interact with multiple devices sequentially or simultaneously while working on a distinct task on each device or combining devices to work on a common task.

example of this mode of operation is to finish a task (write a report) on the desktop and subsequently start a new task (read a document) on the tablet. The device change is primarily motivated by the properties of the new device and the cost of starting (or restoring) the new task on the new device.

Sequential use, common task

- Devices are used sequentially to perform a common task. In this mode the user stops interacting with one device and starts interacting with another device without changing the task. Thus, the user transitions between devices during an ongoing task. An example of this mode of operation is to interrupt a task (write an email) on the desktop and resume the interrupted task on the smartphone. Reasons for this device change are often changes of the current situation. For example, the need to reach a different location may result in a switch from a non-portable to a portable device.

Simultaneous use, distinct tasks

- Devices are used simultaneously to perform distinct tasks. In this mode the user interacts with multiple devices at the same time, each of which is set up to work on a distinct task. Each device is dedicated to a distinct task allowing the user to remain aware of all tasks and frequently switch between them by switching devices. An example of this mode of operation is to watch television while simultaneously browsing the web with a smartphone.

- Devices are used simultaneously to perform a common task. In this mode the user sets up multiple devices to work on the same task in concert. Typically, different devices are assigned specific roles within the task and thus represent different aspects of the task. During task execution, the user can switch between these different aspects of the task by switching between the associated devices. An example of this mode of operation is to designate one computer to writing source code and another to testing source code. Once set up this way, the user can switch between the writing and testing activities by switching between the different computers. At the same time, changes to the source code from the computer used for writing must be reflected on the computer used for testing, illustrating the tight relationship between the devices.

Simultaneous use, common task

2.3 Challenges

As above studies have shown, current systems do not support multi-device interaction in the wild very well. To improve support for multi-device interaction in the wild, the following challenges must be addressed:

- All of above work modes must be supported: Users must be able to sequentially or simultaneously use multiple devices towards a common or distinct tasks.

Support all work modes

- Users must be able to switch between these work modes opportunistically as the situation changes. They must be able to frequently reconfigure the arrangement of tasks and devices. Additionally, it is important that the overhead of changing this arrangement is kept as low as possible to encourage optimal use of the available device diversity.

Support opportunistic rearrangement of devices and tasks

- Due to the volatile nature of device availability, it is important that all transitions between tasks and devices are robust, i.e., the task can be continued on a device after a transition even if the original device becomes unavailable. Solutions that rely on an active network link between devices that goes beyond the moment of task transition are inappropriate for multi-device interaction in the wild. Similarly, solutions that require a constant connection to a fixed in-

Transitions must be robust

frastructure including Internet servers are unacceptable, as users must sustain their ability to work on tasks under a complete loss of connectivity.

Support ad-hoc situations (rely on the infrastructure that the devices provide)

- Solutions for multi-device interactions can make use of sophisticated environments to enable powerful multi-device interactions. However, it is important that these solutions also consider ad-hoc situations where no such infrastructure is available and only the capabilities included in the devices can be used. The basic work modes must be supported independent of any external infrastructure other than basic connectivity.

Before introducing application state as a conceptual model that addresses these challenges, the next chapter will describe the various approaches to multi-device interaction that have been pursued in the past and discuss how they influenced the design of the conceptual model presented in this thesis.

Chapter 3

Related Work

In the literature, there are two main perspectives concerned with multi-device interaction: interaction support and system support. Most of this research has evolved from the domain of multi-display environments, where a physical space like a meeting room is augmented with multiple interactive displays to facilitate group collaboration. Even though this situation is very different from the situation considered in this thesis where multi-device interaction occurs in the wild, many of the insights from both perspectives can be readily transferred.

Research in multi-device interaction has focused on interaction support and system support

Multi-device interaction techniques enable users to span and coordinate their activities across multiple devices. Applied to the domain of multi-device interaction in the wild, these techniques enable the bottom half of the multi-device interaction matrix: sequential and simultaneous use of multiple devices towards a common task. Thus, it is important to understand the diversity of existing multi-device interaction techniques and observe this diversity when designing systems for multi-device interaction in the wild. Section 3.1 will take a closer look at multi-device interaction techniques.

Multi-device interaction techniques are the drivers of multi-device interaction in the wild

Many different systems have been proposed in the literature that support multi-device interaction. Even though none of these systems provides a complete solution for multi-device interaction in the wild, they solve many of its issues. Thus, it is important to understand the diversity of existing approaches and identify the aspects that can be transferred to multi-device interaction in the wild. Section 3.2 will analyze existing system approaches to this end.

Existing system approaches that support multi-device interaction provide many useful insights for multi-device interaction in the wild

3.1 Interaction support

Numerous ways of supporting multi-device interaction have been proposed in the literature. These solutions range from complete interaction frameworks that define conceptual models for user interaction to concrete interaction techniques.

3.1.1 Multi-Device Direct Manipulation

Direct manipulation

A direct manipulation interface, as introduced by Shneiderman [1983], allows users to directly manipulate digital objects similar to how physical objects are manipulated. In particular, direct object manipulations are incremental, reversible, and provide continuous feedback. This way, users can evaluate their actions while performing them and compensate for mistakes before finishing the action.

Pick-and-Drop

Pick-and-drop is a multi-device direct manipulation technique

Rekimoto [1997] applied the concepts of direct manipulation to multi-device interaction and designed an interaction technique called *Pick-and-drop*. He observed that users struggle with exchanging information between computers that are physically close to each other because of a lack of simple interaction techniques for multiple-devices interaction that make use of the spatial arrangement of the devices.

Pick-and-drop has evolved from Drag-and-drop

The Pick-and-drop technique has evolved from applying the *Drag-and-drop* technique to a pen input device. Drag-and-drop is a direct manipulation technique that allows users to move information by "holding" it with the mouse. The user grabs an object by clicking on it and holding the mouse button. Afterwards, the object can be moved by dragging the mouse cursor to the desired destination, where the mouse button is released. Applying this interaction technique directly to a pen is problematic because users find it difficult to drag a pen over a distance with constant contact to the surface.

Pick-and-drop allows users to "pick up" and object by tapping it and "drop" it by tapping the destination

The Pick-and-drop technique was designed to overcome this problem. Instead of dragging the pen over the surface, the user "picks up" an object by tapping the pen on the object and drops it by tapping the pen somewhere else. While

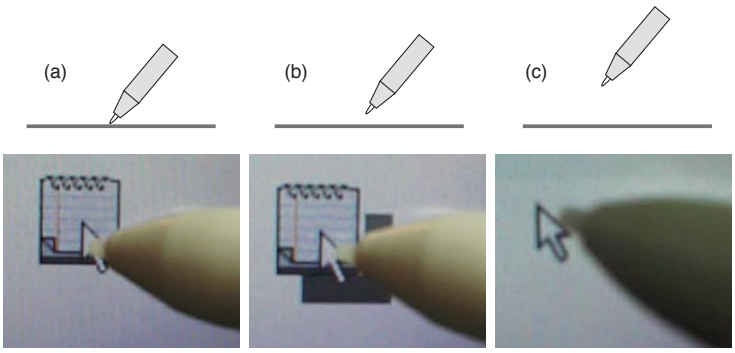

Figure 3.1: Pick-and-drop applies Drag-and-drop to a pen: Tapping the pen on an objects picks up the object (a). The object is associated with the pen and appears under it while hovering (b). When the pen is removed from the display, the object disappears (c). The object is dropped by tapping the pen on the destination. Picture taken from Rekimoto [1997].

moving the pen, the attached object appears close to the pen as long as the pen remains close to a display. If the pen is moved away from the screen, the object disappears but remains associated with the pen. Upon dropping an object, the object is moved to the location where the pen touched the screen. Figure 3.1 illustrates the basic interaction with Pick-and-drop.

Pick-and-drop can span multiple devices

Since pens are not tethered to devices they can be used on multiple devices. Thus, Pick-and-drop can be used as a multi-device interaction technique: Users can pick up an object from one device and drop it on a different device. To maintain the mapping between peens and picked-up objects, each pen on the network receives a unique identifier. The association between objects and pens is then managed by a central service on the network. Once a Pick-and-drop action is complete, this service initiates the object transfer from the source to the target device.

Pick-and-drop applications

Similar to Drag-and-drop, Pick-and-drop is a generic interaction technique that can be applied to various situations. Rekimoto [1997] demonstrates this versatility with several example applications (see also Figure 3.2):

Information exchange between devices

- Pick-and-drop can support information exchange between devices. Users can tap on an information object on any device to pick it up and transfer the object to another device by dropping it there. This way, users can transfer data from a colleague's tablet device to their own when they meet physically. Alternatively, users can exchange information by placing

it on a shared screen, where they or others can pick it up later.

Combining large display and tablet devices

- The unique pen identifiers used for Pick-and-drop can be used to augment large displays with personal tablets. For example, when sketching on a large display, the tablet can show colors and clip-art graphics that can be selected by tapping on them with the same pen used for drawing.

Anonymous displays

- Pick-and-drop can also serve to transfer data between a desktop computer and multiple tablet computers, which are reachable from the desktop. These tablets then act as "temporary work buffers", which users can freely use to temporarily store work objects for later reuse.

Information exchange between interactive and static objects

- Finally, Pick-and-drop can be used to transfer objects between a static object and a digital device. For example, users can pick up an object from a piece of paper by tapping on a printed icon representing the object. This object can then be dropped on an interactive device by dropping the object on the device. To this end, physical artifacts (icons) can be detected through a ceiling-mounted camera and printed markers.

Pick-and-drop uses physical mappings for multi-device interaction

The underlying design philosophy of Pick-and-drop is that current workspaces are a fusion of physical and virtual spaces. Despite this fusion, traditional information exchange methods take place entirely in the virtual space based on symbolic representations of the physical devices. Pick-and-drop, on the other hand, includes the physical space in multi-device interaction by allowing users to identify physical devices by their physical representation.

Passage

Passage associates digital information with physical objects

Konomi et al. [1999] introduce a similar interaction technique called *Passage*, which makes use of real-world objects to transport digital objects. Users connect digital information with a real-world object, called passenger, by placing the passenger on a *Bridge*. As soon as the Bridge identifies the passenger, the user can drag information onto the screen area reserved for the Bridge to associate the information with the passenger. Finally, the physical object can be carried to a different location and placed on another Bridge, where the linked information is recovered and shown.

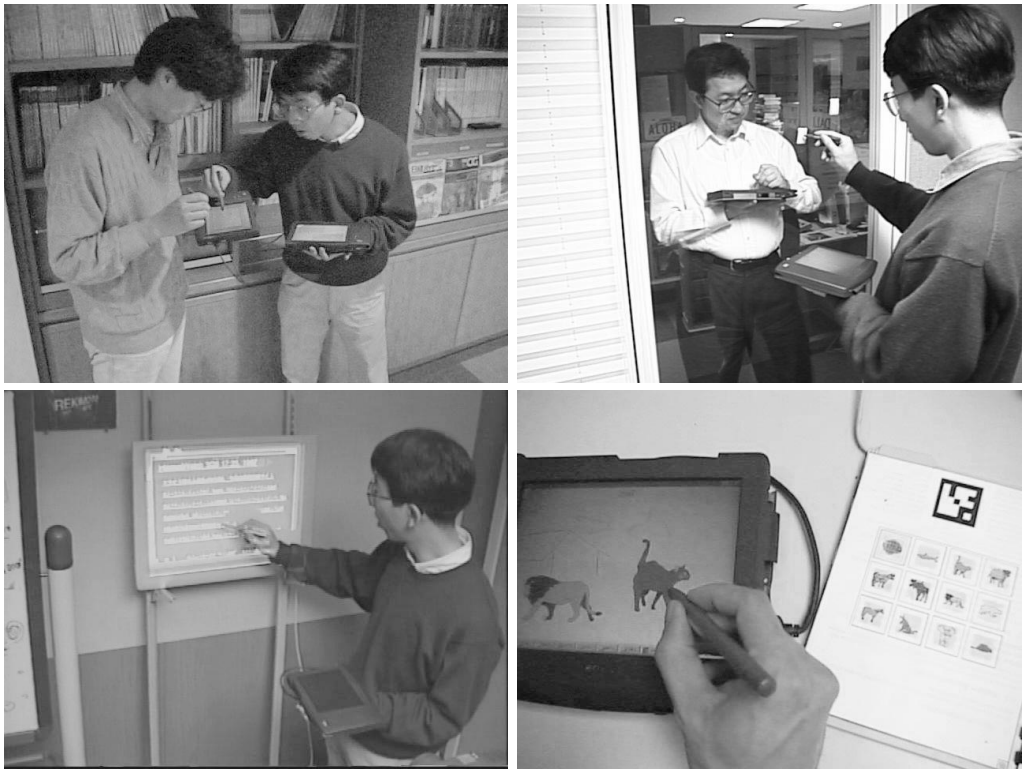

Figure 3.2: Pick-and-drop has various application areas: Users can exchange information between personal tablets (top left) or through a shared display (top right). Interacting with a wall screen can be augmented with a tablet display (bottom left). Objects can be picked up from a printed representation of the object (bottom right). Pictures taken from Rekimoto [1997].

To identify physical objects as passengers, the authors suggest the following two methods:

Identify a passenger by weight

- The weight of a physical object as measured by an electronic scale can serve as an identifier for the object. This allows users to employ any physical object as a passenger. However, if two objects have the same weight they are treated by the system as a single object, which will likely lead to confusion. Additionally, if the weight of an object changes, the link to the data will get lost. Thus, identification by weight is very useful to quickly transport data from one device to another but should not be used to store data over a longer time.

Identify a passenger through an electronic tag

- Alternatively, an electronic tag can be used to identify a physical object. Electronic tags can be sensed over a short distance and contain a unique code, which can

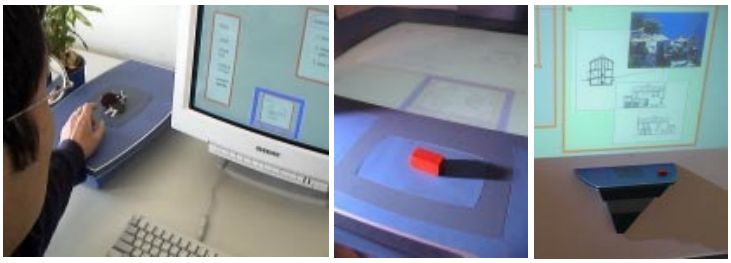

Figure 3.3: Passage links digital objects to physical passengers, which can be transported between bridges, e.g., connected to desktop computers (left), tabletop displays (middle), or wall screens (right). Pictures taken from Konomi et al. [1999] and Streitz et al. [1999].

be used to identify the object. However, the tag must be embedded in the physical object, which limits the number of objects that can serve as passengers. Since the identifier transmitted by an electronic tag is guaranteed to be unique, these objects can be used to store data reliably for a long time.

Passage and the i-LAND environment

Passage was integrated into the *i-LAND* interactive environment presented by Streitz et al. [1999]. The i-LAND environment is an interactive environment that combines a virtual work space with a physical work space by complementing either space with artifacts from the other: The arrangement of the physical work space is integrated into the virtual work space and the information from the virtual work space is visualized in the physical work space. In this context, the passage technique serves as the main information exchange mechanism for bringing information into the room, transporting information among the various components of the room, and extracting information from the room. To this end, the individual components are each equipped with a bridge, allowing information to be added to and removed from the components through a passenger as shown in Figure 3.3.

PaperWindows

PaperWindows allows users to manage digital content on sheets of paper through natural gestures

PaperWindows by Holman et al. [2005] is a set of direct manipulation interaction techniques that augment paper displays with interactive capabilities to manage the arrangement of digital content on multiple displays. These interaction techniques make use of the natural properties of the paper to drive the operations underlying a classic window manager. By mixing paper and digital information

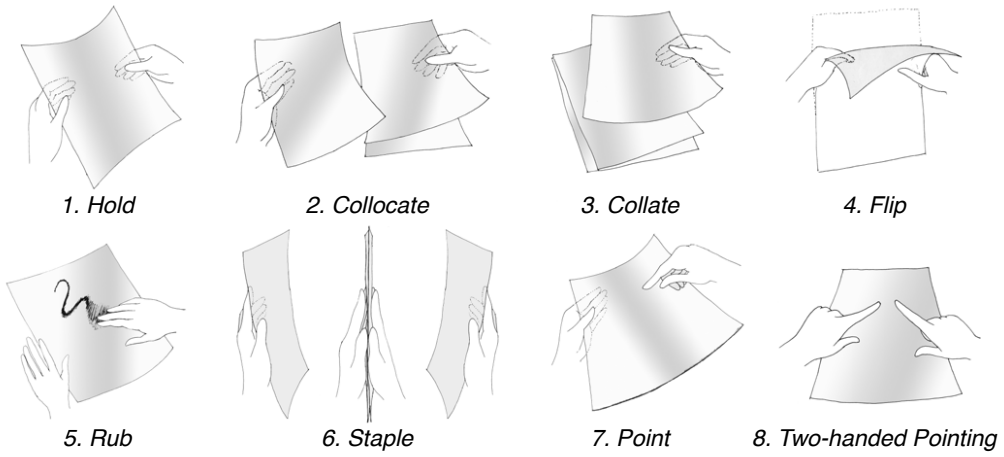

Figure 3.4: Paper Windows turns regular paper into interactive displays and allows users to interact with digital content through gestures performed on the actual paper: (1) Holding paper marks the content as active, (2, 3) collocating and collating paper organizes the content, (4) flipping paper navigates longer content, (5) rubbing paper transfers content between different sheets, (6) stapling paper links the underlying content, and (7, 8) pointing or two-handed pointing interacts with the content on the paper. Picture taken from Holman et al. [2005].

users benefit from the advantages of both worlds: They can handle information as paper and distribute it in the environment. At the same time, they can easily replicate and archive digital copies of the information.

PaperWindows defines several interaction techniques for managing applications across multiple sheets of paper

PaperWindows defines the following interactions on paper (see also Figure 3.4):

1. *Hold*: Holding a sheet of paper marks the content on the paper as active.

2. *Collocate*: Paper Windows can be arranged and distributed in the environment just like regular paper to organize the content or reflect relationships between different information.

3. *Collate*: Collating paper into stacks groups the information on the paper together.

4. *Flip*: Flipping a sheet of paper is used to navigate a multi-page document on a single sheet of paper.

5. *Rub*: Rubbing on a sheet of paper that is placed on another sheet of paper or an external device transfers the information from the topmost paper to the paper or device below.

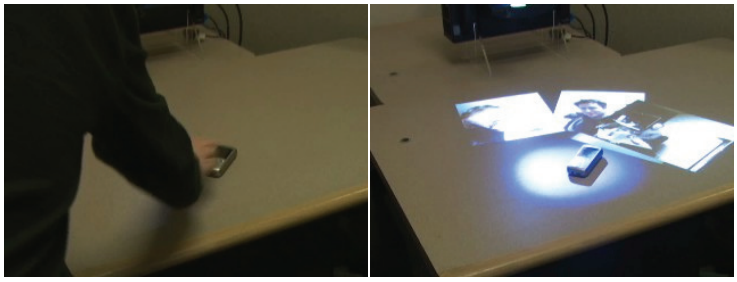

Figure 3.5: BlueTable detects and identifies mobile phones that are placed on an interactive tabletop display. Once identified, the mobile phone can exchange data with the table. Pictures taken from Wilson and Sarin [2007].

6. *Staple*: Two sheets of paper can be linked to the same document by pressing them against each other in a stapling gesture.
7. *Point*: Users can interact with the content on a sheet of paper by pointing and tapping the content on the paper.
8. *Two-handed Pointing*: Pointing is not restricted to a single target but can be performed with multiple hands to allow disjoint selection.

PaperWindows adapts common functions of window managers to interacting with paper

Through these interactions, the most important functions of a typical window manager become available on a multi-screen system based on paper screens. Using these interactions, users can migrate information between paper and devices and organize information that spans multiple sheets of paper.

BlueTable

BlueTable automatically connects physically close interactive devices

Wilson and Sarin [2007] investigated the use of a mobile phone as a physical medium to connect to other devices and exchange information. The *BlueTable* system can detect and identify mobile phones placed on top of a tabletop display. This connection can be used to exchange information between the phone and the tabletop display. For example, users can share their photos by simply placing their phones on the tabletop display. The BlueTable system then detects the phones, extracts the photos, and displays them around the phone on the table. The photos can be distributed further by dragging them between different mobile devices also placed on the tabletop display.

BlueTable uses a camera to detect and identify mobile phones

The system visually detects a mobile phone through a camera that is mounted above the table. As soon as a new phone is detected, the system initiates a connection to all

(a) (b) (c)

Figure 3.6: PhoneTouch allows users to drop information on a surface by touching it with a phone (a), manipulate the information on the surface (b), and finally pick it up by touching it again with the phone (c). Pictures taken from Schmidt et al. [2010].

available BlueTooth devices and initiates a visual signal on these devices. By matching this visual signal with the initial phone detection, the system can associate the device on the table with a BlueTooth connection and exchange data with the device.

PhoneTouch

PhoneTouch allows users to use a mobile phone as a stylus

PhoneTouch by Schmidt et al. [2010] is an extension of above technique that allows the mobile device to be used as an input device in addition to a transport medium. With PhoneTouch user can use a phone to directly interact with a surface by treating the phone as a stylus. For example, a user can share photos by selecting the photos to be shared on the phone and tapping the phone on an interactive table. The photos are then displayed on the table and can be manipulated through the interactive capabilities provided by the table. In addition, others can transfer photos to their phones by tapping the photos they like.

PhoneTouch uses event correlation in time to associate devices and touches

When a phone touches the tabletop display, the touch area can be distinguished from the touch area of regular finger touches. At the same time, the phone can detect the time of the touch by detecting peaks in its device acceleration. By matching the touch events on the phone and the table a connection between the phone touching the table and the touch point can be created.

Figure 3.7: Deep Shot allows users to migrate a task between a mobile phone and a desktop computer by taking a picture of the desktop computer's screen with the mobile phone's camera. The task can be migrated to the mobile phone using deep shooting as depicted in the figure, or it can be migrated to the desktop computer using deep posting. Picture taken from Chang and Li [2011].

Deep Shot

Deep Shot makes use of a camera picture taken from a mobile device to identify the source or target of a migration

Deep Shot by Chang and Li [2011] uses the camera of a mobile phone to identify the source or the target for multi-device interaction. It allows users to migrate a task from a desktop computer to a mobile device or vice versa by taking a picture of the desktop computer's screen with the mobile device. *Deep shooting* migrates the task that is represented by the window in the picture from the personal desktop to the mobile device (see Figure 3.7). *Deep posting* conversely migrates the active task from the mobile device to the photographed location on the desktop computer.

Deep Shot matches the pictures taken from the mobile devices with screen shots of all desktop computers to identify the photographed computer and screen area

Upon taking a picture Deep Shot matches the picture with the screen content of all connected desktop computers and thus identifies the target computer and screen area. For deep shooting, the targeted window and consequently the application is identified based on above data and queried for its state. This state is then transferred to the mobile device and restored in an appropriate application that restores the previously extracted state. Deep posting is done by extracting the state from the active application on the mobile device and restoring it on the desktop computer.

State is encoded as unified resource identifiers, which can be accompanied by a file

Deep Shot uses unified resource identifiers (URIs) to store the state of applications for migration. The data stored in

a URI includes a unique identifier for the application and versatile data in the form of a path (list of keywords) and named parameters. In addition, a file can be attached to the URI, which is also included in the state. For web applications, a URI is a natural representation of their state because URIs are already used to represent the navigation state of the web application. Other applications can use custom schemes and the named parameters to encode their state.

A framework helps integrate support for Deep Shot into applications

Chang and Li [2011] developed a framework for JavaScript and Java to ease the integration of Deep Shot into custom applications. The framework defines a callback for Deep Shot events and a method to trigger deep posting. The callback is called when a device on the network is attempting to send a state to the application, which should be restored. The posting method is called with the state as its parameter to allow users to take a picture of the target device and migrate the state to that device.

3.1.2 Remote Pointing

Remote pointing extends the user's reach to remote devices

Remote pointing interaction techniques extend the user's reach beyond the currently used device. They are typically designed to be used in multi-display environments, where users often need to interact with screens that are not in reach. Instead of physically moving to the remote screen, users can extend the range of their pointing device to include the remote device.

Radar View

Radar views visualize the work areas of all participants in an overview of the entire workspace

Gutwin et al. [1996] propose an interaction technique called *Radar view*, which displays a miniature overview of the entire workspace. The user's and all collaborator's current work areas are highlighted with colored rectangles in this overview. The overview can also be used for navigation by dragging the rectangle representing the user's work area to a different location. Additionally, users can point to remote targets or interact with remote objects by interacting with the miniature representation of the target in the overview.

Radar views can be used to visualize collaborative activities

In collaborative settings Radar views can be augmented with portraits or other personalized information to link the various work areas in the overview with the collaborators

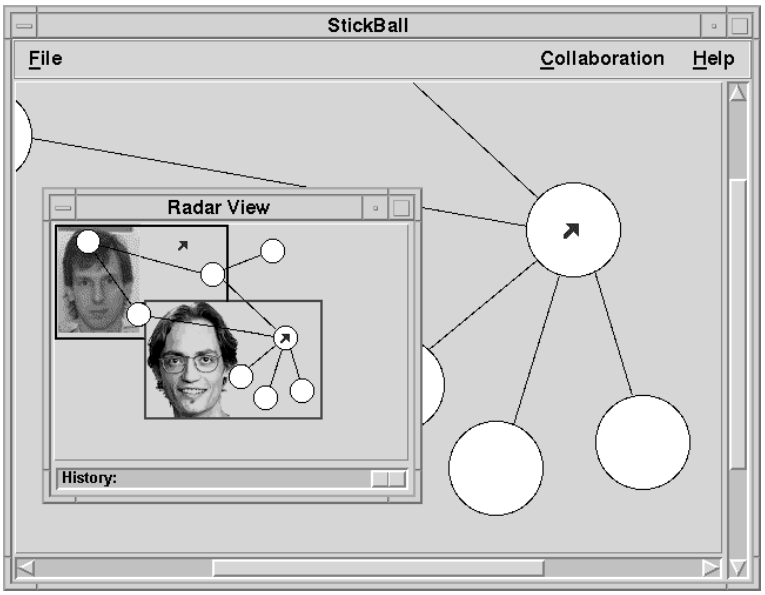

Figure 3.8: The Radar view displays an overview of the entire workspace with the active work areas of all participants highlighted. Picture taken from Gutwin et al. [1996].

(see Figure 3.8). The user's awareness of the collaborators' activities can be increased further by showing the individual pointer positions of all participants.

Radar views can be applied to multi-display environments by arranging the individual device spaces in a virtual workspace

In a multi-display environment Radar views can be used to access remote displays by arranging the visible areas of all displays in a virtual environment, which is then shown in the overview of the Radar view. This way, users can access and interact with all displays from a personal device by either dragging their own view port to the miniature representation of the display or by directly interacting with the remote device through the miniaturized version in the overview.

Hyperdragging

Hyperdragging allows users to drag objects beyond the boundaries of an interactive display onto the surrounding surfaces

Hyperdragging by Rekimoto and Saitoh [1999] allows users to exchange information between multiple devices by moving it through the physical space between the devices. Dragging objects beyond the boundaries of a device is enabled by augmenting the surrounding surfaces with projected screens. Users can then drag objects from an interactive device to the surrounding *Augmented surface* using the mouse. The dragged objects can be left on the surface or dragged further to another device located adjacent to the same Augmented surface. Figure 3.9 shows two users em-

Figure 3.9: Hyperdragging allows users to drag objects from an interactive device through the surface surrounding the device to other devices. Picture taken from Rekimoto and Saitoh [1999].

ploying Hyperdragging to collaborate on and exchange information via an Augmented surface on a desk.

Hyperdragging is used to navigate the spatially continuous workspace created by Augmented surfaces

Hyperdragging was designed to navigate a spatially continuous workspace that is spanned across multiple devices by creating Augmented surfaces between the devices. Through this continuous workspace, multiple interactive devices are connected and can exchange information with one another and the augmented workspace. To this end, users use hyperdragging to access all parts of the augmented workspace and transfer information to the environment and other devices through Drag-and-drop.

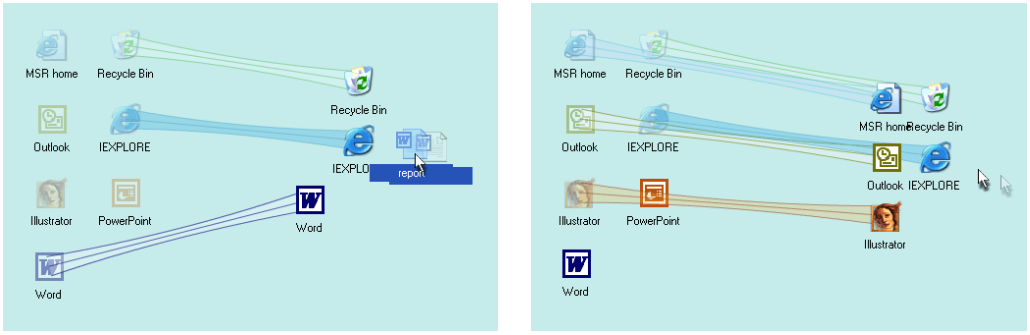

Figure 3.10: Drag-and-pop (left): By dragging an object towards other, compatible objects, these objects are replicated close to the cursor and can be targeted by releasing the mouse button on the replication. Drag-and-pick (right): By dragging the mouse cursor without selecting an object, all objects in the direction of the mouse motion are replicated close to cursor and can be actuated by releasing the mouse button on the desired object. Pictures taken from Baudisch et al. [2003].

Hyperdragging was extended with several design concepts to facilitate information sharing collaboration

To facilitate collaboration and information sharing, hyperdragging was extended with several design concepts. The anchored cursor visualizes the connection between a remote cursor and the interactive device from which it is controlled with a line from the device to the cursor. Users can use this visual cue to identify their own cursor and distinguish it from the collaborators' cursors. Augmented surfaces can be used to hold and share information. Objects placed on a shared surface can be accessed and manipulated by all collaborators. Finally, digital objects can be associated with physical objects that are placed on the Augmented surface, similar to how the passage technique associates digital objects with physical artifacts. The linked objects can then be transferred to a remote location by carrying the physical object to the desired location.

Drag-and-Pop and Drag-and-Pick

Drag-and-drop can be extended to allow remote interaction

Baudisch et al. [2003] present two interaction techniques for transferring objects on large displays to locations that are beyond the user's reach. Both techniques are extensions of the commonly used Drag-and-drop, where objects are moved by dragging them with a mouse or, in this case, an interactive pen. Upon starting a dragging motion the appropriate remote objects in the direction of the motion are brought close to the cursor. *Drag-and-pop* uses this technique to allow users to drag objects to unreachable loca-

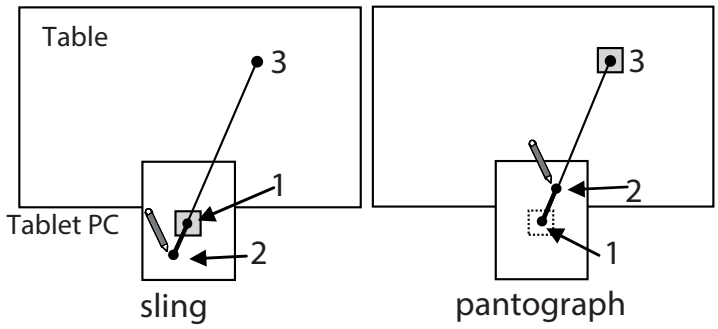

Figure 3.11: Slingshot and Pantograph allow users to move objects to remote locations by dragging a pen from the object towards (Pantograph) or away from (Slingshot) the remote location. The pen movement is then amplified to allow reaching remote objects. Picture taken from Nacenta et al. [2005].

tions. *Drag-and-pick* allows users to actuate remote objects. Figure 3.10 illustrates these interaction techniques.

Drag-and-pop and Drag-and-pick can be used as multi-device interaction techniques

Drag-and-pop and Drag-and-pick can be utilized to access remote objects or transfer objects to remote devices. To this end, the objects that are brought close to the cursor also include objects from remote devices that are located in the direction of the dragging motion. By dragging the touch pen on the tablet towards icons on the computer screen, these icons are replicated on the tablet display and can be actuated by releasing the pen on them.

Pantograph and Slingshot

Pantograph and Slingshot amplify pen movement to reach distant targets

Pantograph and *Slingshot* by Hascoet [2003] are similar interaction techniques that translate short pen dragging into long object movement. Pantograph follows the metaphor of throwing an object: the pen is dragged from the object towards the destination. Slingshot, on the other hand, follows the metaphor of a physical Slingshot: the pen is dragged from the object away from the destination. Both techniques are initiated by tapping the pen on the target object that should be moved. The distance of the object movement is then defined by a multiple of the distance of the pen movement. Additionally, users can control the direction of the object movement by moving the pen sideways. To aid aiming, the system gives constant feedback about the projected destination of the object. Figure 3.11 illustrates the two interaction techniques.

Pantograph and Slingshot support multi-device interaction

The Pantograph and Slingshot interaction techniques support multi-device interaction similar to how Drag-and-pop and Drag-and-pick allow users to bridge multiple devices.

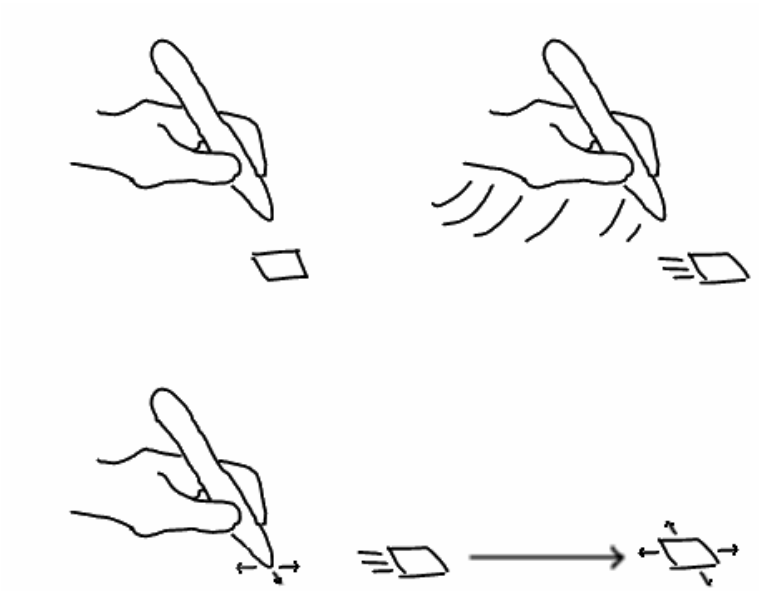

Figure 3.12: Superflick allows users to "flick" a digital object to a remote location by applying inertia to it. To enhance precision, users can move the object after the flick by moving the pen. Picture taken from Reetz et al. [2006].

By moving the pen towards (Pantograph) or away from (Slingshot) another interactive device, the destination can jump onto that device allowing users to transfer objects between devices. Multiple devices can be distinguished by direction and distance.

Superflick

Superflick allows users to "flick" objects to their destination

Superflick by Reetz et al. [2006] is an interaction technique that allows users to roughly "flick" objects to remote locations and subsequently position the object precisely. A flick is initiated by holding the pen on the object and dragging and releasing the pen in a quick motion towards the desired destination. The object continues to move towards the destination with decreasing speed, just like a physical object would continue to move because of inertia. The user can adjust the final position of the object after the flick by holding and moving the pen on the now empty surface next to the user. The Superflick interaction technique is illustrated in Figure 3.12.

Objects can be "flicked" to remote devices

Superflick can be used to transfer objects to remote devices by flicking the object in the direction of the device. When the object reaches the boundary of a device with some inertia left, it jumps to the next remote device in the direction of movement and continues its trajectory there. Once the

object stops, the user can position it precisely on the remote device by dragging the pen on the local device.

TractorBeam

TractorBeam augments a pen's touch interaction with remote pointing

TractorBeam by Parker et al. [2005] is a hybrid point-touch interaction technique that allows users to switch between touch and pointing to interact with objects. TractorBeam is controlled with a pen, which is used as usual to interact with objects in reach. Additionally, when lifting the pen above a certain threshold, users can point at remote objects to interact with them. Thus, the user can decide whether to use the pen as a touch device or as a pointing device.

TractorBeam supports multi-device interaction

TractorBeam supports multi-device interaction by allowing users to point at objects located on remote devices. Even though the initial design of TractorBeam only considers target acquisition, it can be easily augmented with physical buttons on the pen to enable moving or actuating remote objects. Additionally, since users can point at reachable targets as well as remote targets, they can use TractorBeam to move unreachable objects into reach. After moving an object this way, it can be manipulated via direct touch with the pen. Later, the object can be moved back to its original position using TractorBeam.

PointRight

PointRight allows users to redirect their input to other devices by moving the mouse cursor beyond the boundaries of the local screen towards another device

PointRight by Johanson et al. [2002b] is a peer-to-peer input device redirection system that allows users to dynamically control remote devices with the keyboard and mouse of their personal device. The available devices are arranged in a virtual space, where each screen is connected to up to four neighboring screen via the sides of the device. The virtual arrangement of the devices should be chosen to match the physical layout of the displays. Users can then transfer their input from one device to a neighboring device by moving the mouse pointer beyond the boundary of the screen in the direction of the target device. Upon reaching the screen boundary, the cursor immediately jumps to the next device and all input is redirected to that device.

PointRight transfers the established interaction technique of operating multiple displays on a single computer to a multi-display environment

The PointRight interaction technique reflects how today's personal computers give users access to multiple connected

displays: Moving the mouse cursor beyond the boundary of the primary display in the direction of the secondary display transfers the mouse cursor to the second display. PointRight uses this well-established interaction technique to connect the multiple devices of the *iRoom* (see section 3.2.1) to form a coherent system that can be operated from a single mouse and keyboard.

Perspective Cursor

The Perspective cursor allows users navigate a single mouse cursor across multiple displays under consideration of the user's perspective

Similar to PointRight, the *Perspective cursor* by Nacenta et al. [2006] enables multi-device interaction by allowing users to traverse multiple devices with a single cursor. However, the Perspective cursor addresses the misalignment that occurs when users control this cursor on a remote device that is viewed from an angle. Instead of keeping the mapping between mouse movement and cursor movement constant, the Perspective cursor adjusts this mapping according to the perspective of the user. Through this adjustment, the cursor movement appears consistent from the user's perspective.

Perspective correction is achieved by simulating the work environment in a 3D space

To enable the adjustment of the cursor movement to the user's perspective, the position of the user and all employed devices are tracked in a 3D space. In this space, the cursor is represented by a vector that originates from the position of the user. Moving the mouse controls the direction of this vector: horizontal movement changes the horizontal angle and vertical movement changes the vertical angle of the vector. The actual cursor position is located at the intersection of the vector and one of the devices' screens.

The Perspective cursor remains static upon user movement, adjusts its size to the user perspective, and is visualized on all devices

The design of the Perspective Cursor includes several features that improve its overall visibility. To avoid accidental cursor movement, the cursor is fixed to its current position when the user moves. In other words, the origin of the vector is adjusted, while the point of interception is fixed. Furthermore, the size of the cursor is kept constant in the perspective of the user by adjusting the cursor size based on the distance between the device and the user. Finally, the off-screen cursor position and distance is visualized on all devices that do not show the actual cursor by drawing the part of the circle with the cursor position at its center that intersects the edges of the screen similar to the Halo technique by Baudisch and Rosenholtz [2003]. This technique

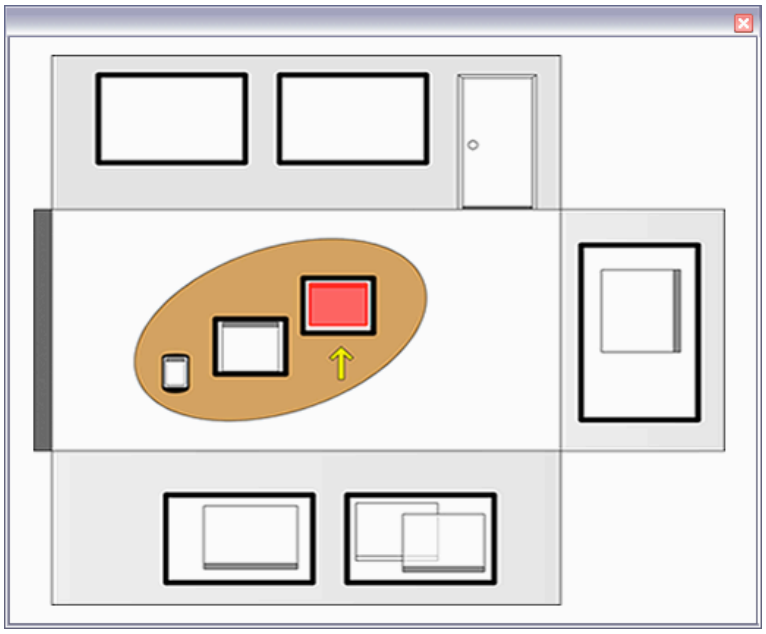

Figure 3.13: ARIS uses an iconic map to represent the available interactive devices and their arrangement in space. The walls of the room and any devices hanging on the walls are projected onto the floor. The map also includes non-interactive but characteristic objects such as tables and doors to assist user orientation. Picture taken from Biehl and Bailey [2004].

helps users locate the cursor by allowing them to estimate the cursor position when it is located in between devices.

ARIS

ARIS uses an iconic map that enables users to relocate running applications within an active space

The *Application Relocator for Interactive Spaces* (*ARIS*) by Biehl and Bailey [2004] is an application manager for interactive spaces. It allows users to relocate running applications to devices within the interactive space by selecting the target device from an iconic map as shown in Figure 3.13. To relocate an application, the user opens the iconic map from a special button on the application's main window. Then, the user selects the target of the relocation by clicking the icon representing the target device. The position of the application on the remote device's screen can be adjusted after the transfer by dragging the mouse on the icon of the remote device.

The iconic map visualizes the available devices and their spatial arrangement

The iconic map used by ARIS shows all available devices in the space with an iconic representation. The device icons are arranged the same way as the physical devices, allowing users to apply their knowledge of the physical space to navigate the virtual space. To provide additional cues, characteristic artifacts of the physical environment such as tables and doors are also included even though they are not interactive.

ARIS allows input redirection by hovering the mouse cursor over the device to be controlled

In addition to relocating application, ARIS also allows redirecting input to remote devices. Input redirection is enabled by opening the iconic map and hovering the mouse cursor over the target device for a short time. All input is then redirected to the indicated device. To stop input redirection, the user opens the iconic map on the currently controlled device and hovers briefly over the own device. Input redirection is automatically initiated after an application relocation.

3.1.3 Synchronous Gestures

Definition of Synchronous gestures

Synchronous gestures are "patterns of activity, contributed by multiple users (or one user with multiple devices), which take on a new meaning when they occur together in time, or in a specific sequence in time" [Hinckley, 2003, p.149]. In other words, interaction gestures obtain a new meaning when they are conducted in parallel or sequentially on different devices or by different users. For example, shaking one device can have a different meaning than shaking two devices at the same time.

Bumping

Bumping uses a physical collision of two devices to initiate a multi-device operation between the devices

Bumping by Hinckley [2003] is an interaction technique based on physically colliding or "bumping" two objects against each other. For example, portable devices such as tablets or smartphones can be bumped against each other or against a fixed surface such as a tabletop display or a wall screen to initiate an operation that affects both devices. Such a collision can be detecting with an accelerometer by watching for time-synchronous acceleration peaks in opposing directions on both affected devices. Figure 3.14 illustrates the interaction technique by example of bumping two tablet computers against each other.

Bumping can be used to address versatile tasks: extend a display, mutually exchange information, or transfer content

Hinckley [2003] suggests several examples that use Bumping to enable multi-device interactions. *Dynamic display tiling* allows users to combine multiple tablet computers to form a large display. To this end, the master device is placed on a horizontal surface and additional devices are bumped against the master device to extend its display. *Mutual sharing* allows devices to exchange information with one another. The devices are simply bumped

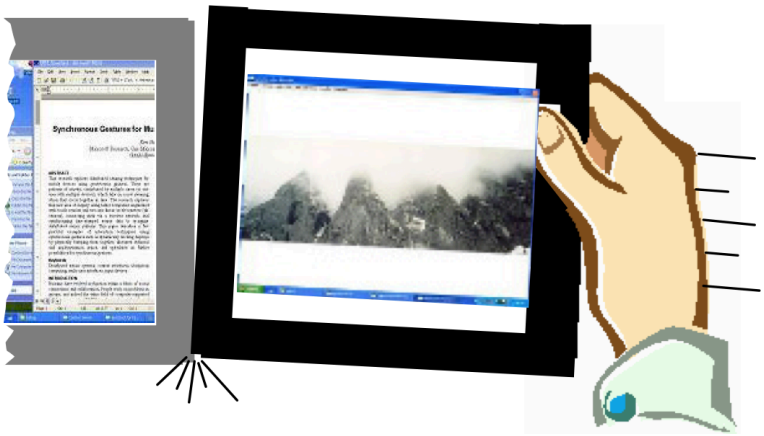

Figure 3.14: Bumping allows users to extend the display of a device or exchange information between multiple devices by bumping them against each other. Picture taken from Hinckley [2003].

against each other to trigger the information exchange. *One-way sharing* allows a device to transfer information to another device. This is done by tilting the device that the information should be sent from while performing the bump.

SyncTap

SyncTap enables multi-device interaction by synchronously pressing a key on multiple devices

SyncTap by Rekimoto et al. [2003] enables users to initiate network connections by synchronously tapping a designated key on two devices. These network connections can be used to exchange content between the devices or to perform automated tasks depending on the devices. For example, coupling a digital camera with a desktop computer this way transfers all images from the camera to the desktop. Figure 3.15 illustrates how SyncTap can be deployed on various device combinations.

3.1.4 Proxemic Interaction

Proxemic interactions augment interaction techniques with knowledge about the proxemics of people and objects

Ballendat et al. [2010] describe *Proxemic interactions* as interaction techniques based on a shared knowledge of surrounding people and objects that includes their position, identity, movement, and orientation. The sensed objects include fixed and portable interactive devices as well as common household objects like furniture, books, kitchenware, and newspapers. The knowledge of the proxemic relationships between people and objects can be exploited to design implicit and explicit interaction techniques that can anticipate user intentions to some extend and adapt their functionality accordingly.

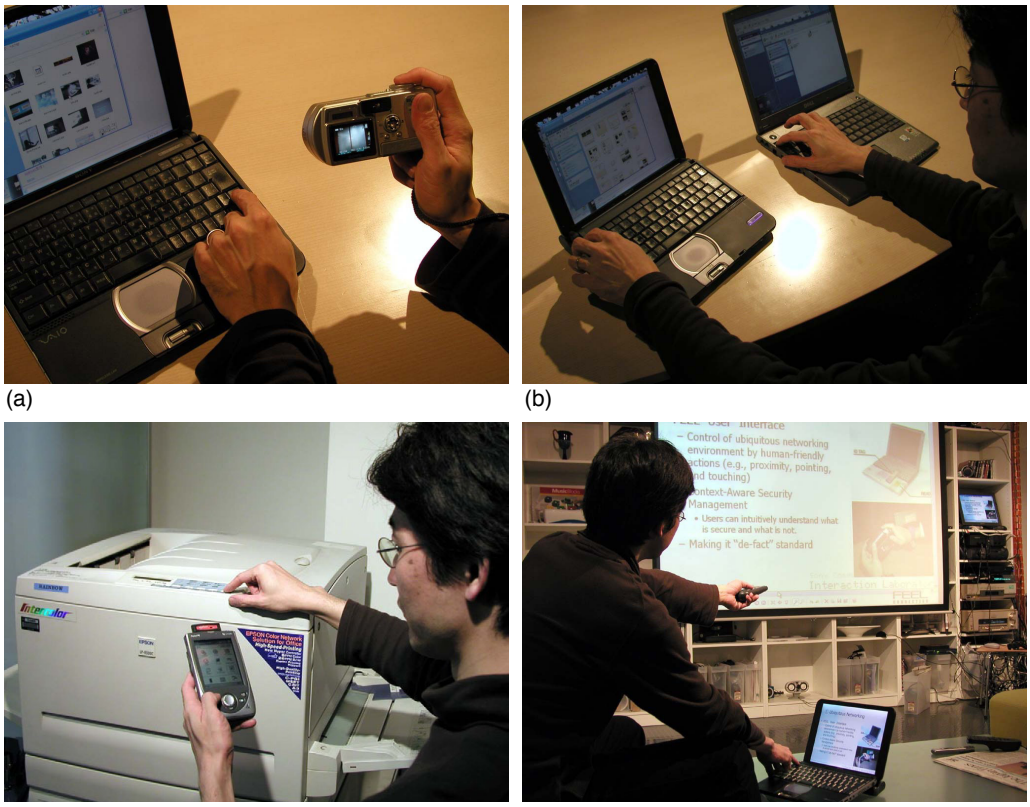

Figure 3.15: SyncTap allows users to initiate network connections between digital devices by synchronously tapping a key on both devices. SyncTap can be used in various situations: (a) Transfer images from a digital camera to a PC, (b) connect two PCs with each other, (c) print a document from a PDA, or (d) display a presentation on a wall screen. Pictures taken from Rekimoto et al. [2003].

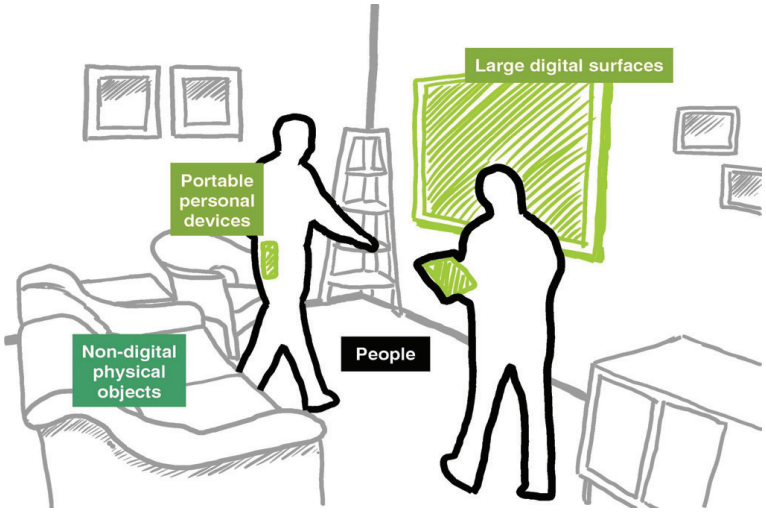

Figure 3.16: Proxemic interactions augment interactive devices with knowledge about surrounding people and objects. Picture taken from Ballendat et al. [2010].

Proxemics can be used to augment interactive systems with implicit actions that anticipate user needs

Interactive systems can be improved by adapting their behavior implicitly to the proxemics of devices, users, and their directed attention. For example, a media player can adapt its information density to the distance between the media player and the user, revealing more content and background information about the media as the user comes closer. However, when the user starts doing something else like reading a newspaper, the media player can stop offering content to that user.

The transitions initiated by these implicit actions can be continuous or discrete

The transitions between these different modes of operations caused by implicit actions based on proxemics can be either continuous or discrete: A continuous transition scales the content change uniformly to the movement of the user. A discrete transition, on the other hand, switches between different visualizations for different distance zones. Such a discrete switch is especially beneficial for switching between awareness and direct attention. When a user enters the room, the media player senses awareness and offers content to the user. However, if the user does not direct attention to the media player, it goes back to sleep after a short while.

Proxemics, especially between different physical objects, can also be used to trigger explicit interactions

Explicit interactions can be designed based on the proxemic relationships between different physical objects. For example, a portable object can be used to trigger an interaction on a static device, e.g., by pointing at the device. By sensing the proxemic relationships between the user, the portable object, and the direction of the object, this pointing gesture can be turned into an explicit interaction. Similarly, the device-to-device proxemics of two objects can be used to exchange information between these devices. For example, when bringing a camera close to a media player, the media player can display the photos taken by the camera.

The Proximity toolkit provides developers with tools to sense, monitor, and debug Proxemic interactions

A major challenge of enabling Proxemic interactions is to determine proxemic relationships from raw sensor data. To this end, the *Proximity toolkit* by Marquardt et al. [2011] provides designers with fine-grained proxemic information about people and objects in a room-sized environment. The proxemic information is sensed via customizable tracking plug-ins on a server and is made accessible through an event-driven programming interface for all clients in the room. This programming interface allows tracking the orientation, location, identity, and motion of individual objects and people, as well as the distance between all measured entities. All of this information can be monitored visually using the included monitoring tool. Finally, a specific situation can be simulated by recording and playing back desirable proxemic sequences.

Proxemic knowledge can improve many of the previously discussed multi-device interaction techniques

A shared knowledge of the proxemic relationships of people and objects can be very beneficial for many of the previously discussed interaction techniques. Techniques that associate digital content with physical objects can use proxemics to determine where and how to display the content held by the object. The virtual workspaces used by Radar view and PointRight can be configured automatically to match the spatial arrangement of their physical counterparts. Directed techniques such as Drag-and-pop, Drag-and-pick, Pantograph, Slingshot, and Superflick can use the proxemic relationships to determine the appropriate target device for their interaction. Synchronous gestures can give good estimations to resolve conflicts when multiple users trigger the same gesture at the same time if the proxemic relationships between devices and users is known. In consequence, Proxemic interactions provide many opportunities for application in the area of multi-device interaction.

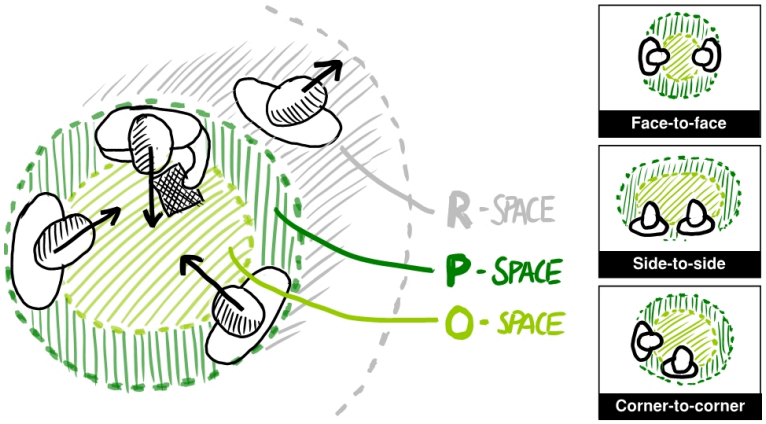

Figure 3.17: An f-formation is a group of people standing together while being engaged in a joint activity. It defines three distinct areas for the group: The O-space is the area of collaboration, the P-space is reserved for the collaborators, and the R-space is excluded from the collaboration. Picture taken from Marquardt et al. [2012b].

Group Together

Group Together senses sociological concepts of people in groups to facilitate interaction

Group Together by Marquardt et al. [2012b] explores the application of *F-formations* and *Micro-mobility* for Proxemic interactions. F-formations are two or more persons collaborating by standing in a group. The spatial arrangement of collaborators defines three distinct areas: The inner space (*O-space*) is reserved for group activities. The ring space (*P-space*) is where the collaborators stand. The surrounding region (*R-space*) is excluded from the collaboration. Figure 3.17 illustrates the three spaces of the f-formation. Micro-mobility describes how people share information on devices by tilting them towards each other to ease group visibility. These sociological constructs can be exploited to design systems that facilitate group interaction.

Group Together introduces four interaction techniques that leverage common group behavior

Four interaction techniques were designed based on common behavior that groups show when collaborating in F-formations. *Tilt-to-preview* allows users to share content with collaborators standing in a group by tapping on the content to be shared and tilting the device within the inner area of the F-formation. The temporary copy of the indicated content then appears on all devices within the F-formation. *Face-to-mirror* enables users to mirror all content on their device onto all other devices by tilting the device to a vertical position. *Portals* are tinted indicators that appear at the edge of a table when it is slightly tilted towards another tablet. Users can use these Portals to transfer content to other tablets by dragging the content across the indicator. *Cross-device pinch-to-zoom* extends the display of a tablet with other surrounding tablets opportunistically.

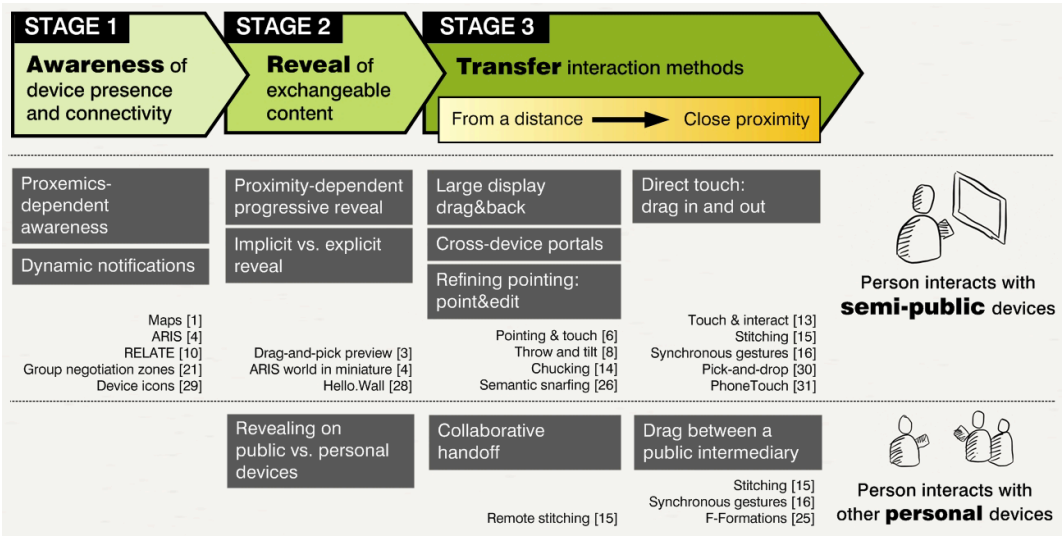

Figure 3.18: Gradual Engagement traverses three stages: (1) Users are made aware of available devices and how they can be connected; (2) Content that can be exchanged between these devices is revealed; (3) Content is transferred between the devices in a way that is tailored to the participating devices and possible user tasks. For each stage, several interaction techniques for engaging shared and personal devices are listed. Picture taken from Marquardt et al. [2012a].

When zooming an object, it is shown on other tablets in the same F-formation that are close by and held similarly.

Gradual Engagement

Gradual Engagement is a design pattern to gradually transition between awareness and actuation of information exchange based on proxemics

Gradual Engagement by Marquardt et al. [2012a] is a design pattern for exchanging information between devices based on Proxemic interactions. The pattern describes how to gradually engage the user to convey the capabilities of two devices to exchange information based on their proximity. As users approach a typically fixed device like a wall screen with a portable device, the fixed device reveals more and more information about what information can be exchanged between the two. For example, when approaching a media center's screen with a camera, it first displays a camera icon indicating that information exchange is possible. As the user moves the camera closer to the media center, photos appear around the camera icon.

Gradual Engagement traverses three stages: (1) devices convey awareness, (2) devices reveal exchangeable content, (3) devices exchange content

The Gradual Engagement pattern suggests that conveyed information exchange capabilities flow across three stages: In the first stage, the device convey to the user that it is possible to exchange information between them creating awareness. In the second stage, the devices reveal the con-

tent that can be exchanged. In the final stage, the devices offer interaction methods for transferring the content. The device transitions between these stages from the first to the third gradually as the distance between the device and the trigger decreases.

3.1.5 Taxonomy of Multi-device Interaction

Under the supervision of the author Zhang [2012] created a taxonomy of multi-device interaction techniques. The taxonomy is an extension of the taxonomy presented by Nacenta et al. [2009], which organizes multi-device interaction techniques in three dimensions: the referential domain, the relationship between input space and display configuration, and the control paradigm. Zhang [2012] developed seven more classification dimensions to reflect more subtle aspects of multi-device interaction in the taxonomy.

Taxonomy Dimensions

Input method: input modalities of an interaction technique

Input method describes the input modalities used to perform the interaction technique. The possible options are organized in five categories: (1) includes commonly used pointing devices, such as the mouse or a stylus pen, as well touch-based systems; (2) includes devices based on physical buttons, such as keyboards or other special purpose interfaces; (3) includes techniques that employ physical tokens to represent digital information, such as RFID tags or other objects that can be uniquely identified; (4) includes vision-based techniques that use mounted or environmental cameras to observe the execution of the interaction technique; (5) includes techniques that employ other kinds of sensors to track objects or people, such as a 3D depth camera. The choice of input method describes the technical requirements underlying a system, which mostly influence how easily the system can be installed in different locations.

Positional mapping: spatial mapping between input and output

Positional mapping describes the mapping between input position, movement, and system action. The positional mapping can be absolute, relative, or rate-based. When absolute mapping is used, the physical position detected by the input device is directly mapped to a virtual position (e.g., touch-screen). For relative mapping, the absolute physical position is ignored and instead changes of

the physical position are mapped to changes of the virtual position (e.g., mouse). Rate-based systems map the absolute physical position or manipulation of the input device to changes of the virtual position (e.g., joystick). Absolute mappings are arguably more natural to use than other mappings, however, relative and rate-based systems can be more precise and also easier to use if indirect control mechanisms with a high power of working area are employed (see below).

Replace-ability of input devices: the extent to what an input device can be replaced

Replace-ability of input devices describes to what extent an input device can be replaced at runtime. Four different types of replace-ability are distinguished: (1) The input device cannot be replaced. (2) The system can be used with a variety of input devices of a specific kind, but the replacement device must be configured prior to its use. (3) Any input device of a specific kind can be used without prior configuration. (4) Any object can be used as input without prior configuration, including non-interactive objects. The higher the replace-ability is, the easier it is for the user to appropriate the system by using diverse input devices.

Power of working area: distance between the user's hand and the furthest reachable virtual position

Power of working area refers to the distance between the user's hand and the furthest possible virtual destination that can be reached. Three different levels of power are distinguished: (1) Within hand's reach (high power) refers to systems that allow users to control distant devices without moving their arm significantly (e.g., wireless mouse and keyboard). (2) Within arm's reach refers to systems that require the user to use up to the full length of the arm to reach or point to remote devices (e.g., remote control). (3) Beyond arm's reach refers (low power) to systems where the user needs to physically move to a distant device to manipulate it (e.g., direct touch or stylus-based systems). While designing for high power of working area is desirable to reduce the effort needed to interact with remote devices, low power systems benefit from a more natural spatial mapping between the physical layout and the virtual space.

Referential environment: reference method to identify target devices

Referential environment describes the reference method used to indicate the target of a multi-device operation. The reference methods can be coarsely divided into two categories: spatial and non-spatial methods. Non-spatial methods organize the possible target devices in a way that does not reflect the spatial arrangement of the physical devices. A discrete referential environment is such a non-spatial method that uses some discrete list or hierarchy of names to give

access to all possible targets of the environment. A virtual space that arranges devices spatially but does not reflect the spatial arrangement of the physical devices is also a non-spatial referential environment. Spatial methods, on the other hand, reflect the physical arrangement of devices in their organization. A virtual environment that resembles the arrangement of the physical devices in its virtual counterpart is considered a spatial method. Finally, the physical environment itself can also be used as a spatial referential environment. Spatial methods can be easier and more natural to use because they make use of the existing physical arrangement of devices to identify a target and thus eliminate the need to learn a different mapping. Non-spatial methods, on the other hand, can facilitate different properties of the available devices to give access to them by need (e.g., list all objects that can print), and methods with high power can benefit from a hierarchical organization to reduce the space needed.

Input model type: mapping between devices in the physical and virtual space

Input model type describes the mapping between the physical and the virtual arrangement of devices. It only applies if a spatial referential environment is used to identify target systems. The input model can be planar, perspective, or literal. In a literal model, the physical environment itself is used as the virtual space and no mapping is performed. In the planar and the perspective model, the 3-dimensional environment must be mapped to a 2-dimensional plane to allow the visualization of the virtual space on current display technology. In the planar model, this mapping ignores the user's position and typically uses a simplified representation of the arrangement of devices and key objects in the environment. In the perspective model, the mapping is based on the user's perspective and attempts to present the virtual environment in a way that closely resembles the user's current view of the physical space. While the perspective model is a more natural representation of the space, it requires knowledge about the user's position, which can be challenge in a multi-user environment, and it reduces the user's ability to learn the spatial arrangement of the virtual space as it changes with the position of the user.

Feed-forward: ability to see and influence the target of an operation during its execution

Feed-forward describes the extend to which a system allows a user to manipulate the final destination of multi-device operations during their execution. An open-loop system can visualize the final destination of an operation during

	Input Method					Positional Mapping			Replace-ability of input device				Power of Working Area		
	Mouse/ Pen/Finger	Token-based	Tracking s./Motion sensing/ Sensor-based	Button-based	Vision -based	Relative	Absolute	Rate-based	Dedicated	Alternative	Compatible	Use anything as Input	Within Hand's Reach	Within Arm's Reach	Beyond Arm's reach
Radar View	✓						✓				✓			✓	
Hyperdragging	✓					✓					✓		✓		
Passage		✓										✓			✓
Perspective Cursor	✓					✓					✓		✓		
Drag-and-Pop/Pick	✓					✓					✓		✓		
Superflick	✓							✓			✓		✓		
Pantograph and Slingshot	✓						✓				✓		✓		
Deepshot					✓						✓		✓		
IM-based Techniques	✓					✓					✓		✓		
TractorBeam	✓		✓	✓			✓		✓					✓	
Pick-and-Drop	✓	✓					✓			✓					✓
SyncTab				✓						✓			✓		
Synchronous Gesture			✓				✓		✓				✓		

Figure 3.19: Classification of 13 multi-device interaction techniques according to the Input method, Positional mapping, Replace-ability of input devices, and Power of working area. Picture taken from Zhang [2012].

the planning phase of the operation but provides no control over the destination once the operation was executed. A closed-loop system, on the other hand, enables the user to adjust the final destination of an operation at all times during its execution. While closed-loop systems give users finer-grained control over multi-device operations, they often require the user to pay attention during the entire execution phase and cannot display the target and consequently the impact of the operation before execution.

Feedback: sensory channels used to notify the user about the impact of operations

Feedback describes how the system informs the user about operations and their impact. This information can be provided in the form of visual, audible, or haptic feedback. Visual feedback is the most common type of feedback and should always be included. Audible and haptic feedback, however, are valuable extensions to visual feedback that do not require the user to pay direct attention to the system to be perceived. On the downside, audible and haptic feedback can be disruptive.

	Referential Environment				Input Model Types			Feed-forward		Feedback		
	Spatial		Non-spatial		Planar	Perspective	Literal	Open-loop	Closed-loop	Visual	Audio	Haptic
	Coupled virtual and physical space	Physical space	Discrete/ No topology	Virtual space								
Radar View	✓				✓				✓	✓		
Hyperdragging	✓				✓				✓	✓		
Passage		✓					✓		✓	✓		
Perspective Cursor	✓					✓			✓	✓		
Drag-and-Pop/Pick	✓				✓				✓	✓		
Superflick	✓				✓				✓	✓		
Pantograph and Slingshot	✓				✓				✓	✓		
Deepshot			✓						✓	✓		
IM-based Techniques			✓						✓	✓		
TractorBeam	✓					✓			✓	✓		
Pick-and-Drop		✓					✓		✓	✓		
SyncTab		✓					✓		✓	✓		
Synchronous Gesture		✓					✓		✓	✓		

Figure 3.20: Classification of 13 multi-device interaction techniques according to Referential environment, Input model type, Feed-forward, and Feedback. Picture taken from Zhang [2012].

Parallelism: ability to support multiple users and operations in parallel

Parallelism describes the ability of the system to be used in parallel. Parallel use can be performed by a single or multiple users, resulting in four distinct cases to be considered: Single-user, single-operation restricts the interaction with the system to a single operation at a time. Single-user, multi-operation allows a single user to perform multiple operations in parallel, while restricting the use of the system to a single user at a time. Conversely, multi-user, single-operation allows multiple users to interact with the system in parallel but restricts each user to perform a single operation at a time. Multi-user, multi-operation is the most flexible solution that allows multiple users to each perform multiple operations in parallel. Supporting increased parallelism can pose a significant challenge for the technical design of a system as each parallel operation and user must be identified as such and performed independent of any other operations that are currently active.

	Multiple Users		Single User		Identification				
	Parallel Operation	Serial Operation	Parallel Operation	Serial Operation	Distributed User ID	Centralized User ID	No User ID/ Device ID	Visual labeled	System identified
Radar View	✓			✓	✓			✓	
Hyperdragging	✓			✓			✓	✓	
Passage	✓		✓				✓	✓	
Perspective Cursor	✓			✓			✓	✓	
Drag-and-Pop/Pick				✓					
Superflick	✓			✓	✓			✓	
Pantograph and Slingshot	✓				✓			✓	
Deepshot				✓					
IM-based Techniques	✓		✓			✓		✓	
TractorBeam	✓			✓	✓			✓	
Pick-and-Drop	✓			✓			✓	✓	
SyncTab	✓			✓	✓				✓
Synchronous Gesture		✓		✓					

Figure 3.21: Classification of 13 multi-device interaction techniques according to Parallelism and Identification. Picture taken from Zhang [2012].

Identification: method to manage the ownership of operation for multiple users

Identification describes how the system manages the ownership of operations if multiple users perform operations in parallel. Users can be identified in two distinct ways: A distributed user identifier allows users to specify their own identify, which is then distributed as needed in the environment. A centralized user identifier, on the other hand, is specified beforehand in the system and each participating device must authenticate with the identity server prior to using the system. Additionally, the system can ignore user identities and instead only identify devices as owners of operations. The choice of identification method largely depends on the tasks and the integration of the multi-device environment into other systems that might prescribe certain identification methods.

Classification of multi-device interaction techniques

13 interaction techniques were classified using the presented taxonomy

Figures 3.19, 3.20, and 3.21 show how 13 interaction techniques are classified in the taxonomy presented in this

section. All of these techniques, except for "IM-based Techniques", are described in the text above. IM-based techniques are interaction techniques that use an instant-messaging protocol such as Jabber[1] to communicate and transfer data between different devices.

The classification can be used to identify appropriate interaction techniques for specific multi-device scenarios

Since the appropriateness of the characteristics of the different dimensions often depends on the usage scenario, it is impossible to deduce an abstract rating of the "good"-ness of these interaction techniques from the classification. However, the classification can be used to identify appropriate interaction techniques for a given multi-device scenario. If, for example, it is important to use a perspective input model type that is within-hands reach, then the Perspective cursor can be quickly identified as an appropriate interaction technique.

3.2 System Support

3.2.1 Ubiquitous Computing and Roomware

Ubiquitous computing aims at making the computer disappear, allowing us to focus our attention on the tasks and goals instead of the interaction

Most of the research in multi-device interaction was motivated at least in part by Mark Weiser's vision of *Ubiquitous computing*. In his seminal article, Weiser [1991] juxtaposes computer technology with writing technology: Writing has become almost invisible in our world, allowing us to use it for versatile tasks without paying attention to the writing technology itself. Computing, on the other hand, remains to be very visible, drawing our attention to the technology instead of allowing us to focus on the tasks and goals that we are using it for. Ubiquitous computing aims at resolving these shortcomings by removing the computer from our direct attention and instead embedding it in the environment, where it assist us with our tasks and goals when needed but without demanding our attention while doing so.

Three device classes make up the interaction space of Ubiquitous computing: tabs, pads, and boards

Weiser [1991] identified three fundamental sizes for ubiquitous computers: *Tabs* are small, location-aware interactive devices similar to today's smartphones that people carry around with them as personal devices. *Pads* are scrapbooks the size of a piece of paper similar to today's tablet computers that anyone can pick up and use when needed and discard afterwards. *Boards* are large displays similar to to-

[1] `http://jabber.org`

day's wall screens and table-top displays mainly used for sharing information with others.

Ubiquitous computing requires a seamless transition between multiple devices

Three major technologies play an essential role in the realization of this vision of coordinated device usage: (1) It must be possible to create cheap and low-powered devices in great quantities; (2) There must be extensive and high-speed wireless network coverage for these devices; (3) Software systems must support a seamless experience that spans multiple devices. While the first two technologies are becoming more and more reality today, the third demand remains an unsolved challenge.

Roomware connects multiple devices in a room to form a coherent space

Roomware systems address the third challenge of ubiquitous computing by exploring how software can span an entire room by combining the available devices to form a coherent workspace called interactive space. Many technical solutions for roomware systems have been developed, which differ mostly in the technical aspects of the infrastructure. Two influential systems are discussed in the next two sections: the iRoom, and i-LAND.

Interactive Workspaces Project

The iRoom is an interactive space that was designed according to several guiding principles

The Interactive Workspaces project (or *iRoom*) by Johanson et al. [2002a] investigates the design and use of multi-display environments (MDEs) that combine multiple interactive devices, including portable devices brought by the users, into a coherent system. The iRoom consists of several wall-sized displays, a tabletop display, and the devices that users bring with them into the room. The guiding principles of the iRoom project are:

The iRoom was used actively by the group

- *Practice what we preach*: The iRoom was used actively by the group to conduct meetings.

Focus on co-located cooperative work

- *Emphasize co-location*: The iRoom focuses on co-located collaboration in a shared physical space.

Provide simple means to adjust the environment to the user

- *Reliance on social conventions*: Instead of reacting to users intelligently, the iRoom makes it easy for users to adjust the room to their needs while using it.

Design for general use

- *Wide applicability*: The design of the iRoom strives for the room to be as flexible as possible, allowing it to be used in many different situations.

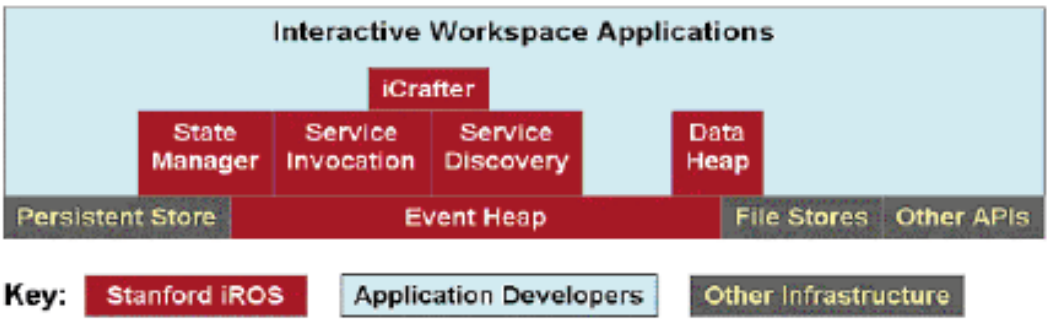

Figure 3.22: iROS consists of three subsystems: The Event Heap facilitates dynamic application coordination through decoupled event propagation via a tuple space. The Data Heap acts as a central data storage for distributed applications. The iCrafter provides advertisement and remote invocation capabilities for distributed services and supports on-the-fly user interface generation for these services. Picture taken from Johanson et al. [2002a].

- *Keep it simple*: The UI design and software development interfaces should be as simple as possible.

Keep the UI and the system design simple

The iRoom was designed to support three basic task characteristics: *Moving data* between devices and applications. *Moving control* between devices such that any input device can be used to control any application running in the space. *Dynamic application coordination* allows diverse task applications to be coordinated in manifold ways. These tasks are executed in a dynamic and heterogeneous environment (a changing variety of diverse devices is used), where users interact with multiple devices and applications in parallel.

The iRoom was designed to support moving data and control across all devices and a dynamic coordination of diverse applications

The middleware infrastructure underlying the iRoom is called the *Interactive Room Operating System* (*iROS*). It ties together the devices included in the interactive workspace, which all have their own low-level operating system. The design of iROS follows three general design principles: iROS applications are decoupled from one another allowing the dynamic rearrangement of devices and applications. Any devices that fail during operation of the room can be reset by simply restarting the device. Web technologies are used to render user interfaces where possible. The following three subsystems make up iROS (see Figure 3.22 for an overview):

The iRoom is driven by the middleware infrastructure iROS, which consists of three major subsystems

- The *Event Heap* is a coordination infrastructure for the applications running in the iRoom. It is based on a shared *tuple space*, where ordered collections of values called *tuples* can be posted to and read from a shared

The Event Heap coordinates applications through shared events

space. Tuples are queried by providing a tuple template, which defines the fields of the tuple that the returned tuples must match. This read operation can be instructed to "consume" the tuple after retrieval, i.e., remove it from the shared space. A detailed description of the Event Heap is given by Johanson and Fox [2002].

The Data Heap allows applications to store and exchange data

- The *Data Heap* provides a centralized data storage for the iRoom. Any application can request to store data in the Data Heap, which can subsequently be retrieved independent of the device. The stored data is organized with a custom set of attributes, which can be used to query and retrieve the data later on.

iCrafter provides an interface to available room services and allows the dynamic combination of these services

- The *iCrafter system* is a framework for services in ubiquitous computing environments. It facilitates service discovery and invocation, which is done via the Event Heap. Additionally, it allows users to select one or more services and service usage patterns to generate an adapted user interface. This generated user interface gives access to the service functionalities or combines services through a predefined pattern, e.g., move data from a data provider to a data consumer. For a detailed description of the iCrafter system, please refer to Ponnekanti et al. [2001].

Interactive Landscape for Creativity and Innovation

i-LAND is an interactive space with integrated information and architectural spaces and support for dynamic, flexible, and mobile group cooperation

The interactive Landscape for Creativity and Innovation (*i-LAND*) by Streitz et al. [1999] is an interactive workspace for dynamic teams. It consists of a wall-sized display, a tabletop display, and several mobile chairs with built-in tablet-sized displays. The environment was designed to integrate information and architectural spaces by augmenting furniture and other architectural artifacts with computing technology. Furthermore, i-LAND was designed to support a high degree of dynamics, flexibility, and mobility in group cooperation.

The *BEACH* application model by Tandler [2004] defines the high-level structure of ubiquitous computing applications in i-LAND. It consists of three design dimensions:

Separating basic concerns

- *Separating basic concerns* organizes applications into five models that pursue well-defined purposes: The *application model* and *data model* specify what tasks can

be performed on what types of data. The *environment model* encapsulates the available context information of the physical environment. The *user interface model* defines the user interface to access the tasks from the application model based on the provided context information from the environment model. The *interaction model* relies on all other models to specify the interaction styles of the environment.

Coupling and sharing

- *Couping and sharing* denotes the distribution of applications across multiple devices by operating on shared data and coupling the state of applications. In the BEACH model, this notion is applied to the separation of concerns into five models: Each model can be shared among other devices and applications in the room: *Sharing the data model* enables multiple applications to work on the same data. *Sharing the application model* enables the tight coupling of applications and their user interfaces through their state. *Sharing the user interface model* enables distributed user interfaces that are spread out across multiple devices. *Sharing the environment model* enables multiple devices to share a common understanding of the environment. *Sharing the interaction model* enables interaction styles that span multiple devices, such as Passage (see section 3.1.1).

Conceptual levels of abstraction

- The implementation of each of above models can be considered at four different *levels of abstraction*: The *core level* encapsulates the low-level handling of the platform-dependent functionality. The *model level* defines application-, domain-, and platform-independent abstractions of low-level tasks. The *generic level* provides reusable functionality in the form of application-independent but domain-dependent components. The *task level* implements tailored support for the tasks of the application.

Figure 3.23 illustrates how the three dimensions of the BEACH application model can be utilized to structure applications for ubiquitous environments: First, the application is split up into components according to the models of the *separation of concerns* dimension at a given *level of abstraction*. Then, for each component the appropriate *degree of coupling and sharing* is determined, depending on the level of collaboration that is desired.

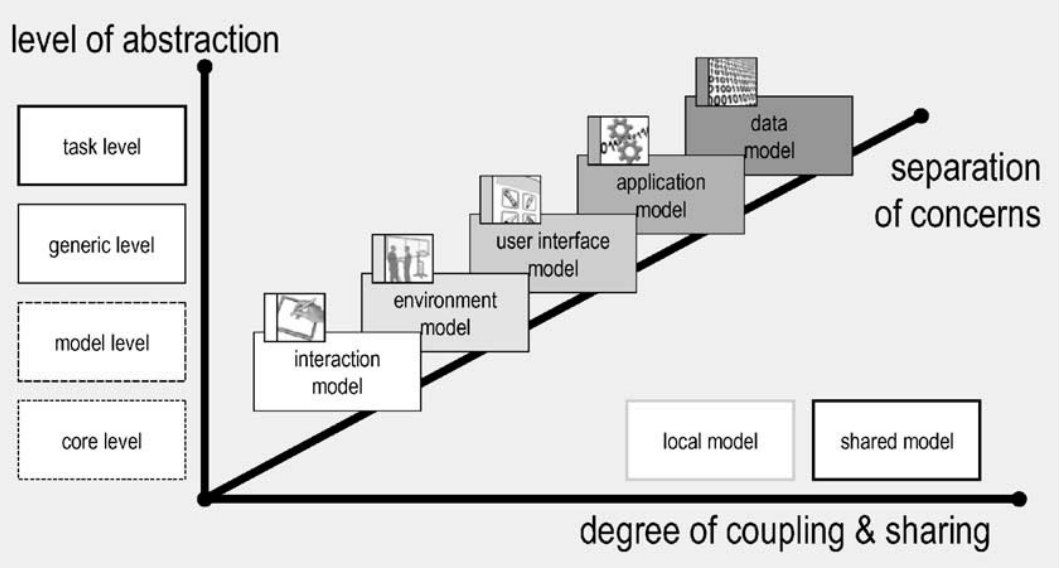

Figure 3.23: The design dimensions of the BEACH application model: Applications are separated into five models representing different concerns; For each model, the appropriate degree of coupling and sharing must be determined; Support for these models can be implemented at four levels of abstraction. Picture taken from Tandler [2004]

Roomware systems are tailored to a specific environment and cannot be used outside this environment

Roomware solutions are designed to augment a defined workspace with computing technology to improve collaboration and other work activities. Even though these solutions allow multiple devices to be used in a coordinated fashion, they cannot be readily applied to multi-device interaction in the wild. The main restriction of Roomware solutions is that they are typically bound to a specific environment with services that are tailored to this environment. These services drive the multi-device coordination and they become unavailable once the user leaves the environment. Consequently, Roomware solutions cannot address the opportunistic nature of multi-device interaction in the wild.

3.2.2 Instrumental Interaction

Instrumental interaction by Beaudouin-Lafon [2000] is an interaction model for graphical user interfaces. In this context an interaction model is defined as:

Definition: *Interaction Model*

> An interaction model is a set of principles, rules and properties that guide the design of an interface. It describes how to combine interaction techniques in a meaningful and consistent way

> and defines the "look and feel" of the interaction from the user's perspective. Properties of the interaction model can be used to evaluate specific interaction designs. [Beaudouin-Lafon, 2000, p446]

In instrumental interaction users manipulate domain objects with instruments

Instrumental interaction is an interaction model that extends and generalizes the principles of direct manipulation, based on how we naturally use tools, called *instruments*, to manipulate objects of interest, called *domain objects*. Domain objects are the data that applications operate upon. They are the primary focus of the user and form the basis and purpose of the interaction. Interaction instruments are mediators between users and domain objects. They decompose interaction into two layers: (1) the physical action of the user on the instrument, and (2) the command sent to the domain object and its response (feedback). An Interaction instrument is the reification of one or more commands. Instruments can themselves be domain objects, which are operated upon using meta-instruments.

The instrumental interaction model can be used to describe and compare existing interaction techniques, and facilitate the process of generating new interaction techniques. To this end, Beaudouin-Lafon [2000] suggested the following three metrics to describe and compare interaction techniques:

Degree of indirection: spatial and temporal offset between instrument and domain object

- The *Degree of indirection* measures the 2-dimensional offset (spatial and temporal) between an instrument and a domain object. The spatial offset is the distance between the logical part of the instrument and the domain object it operates upon. The temporal offset is the time difference between actuation of an instrument and the response of the object, or its feedback. In general, the smaller the degree of indirection is, the more direct and responsive a user interface is and consequently the more desirable it is.

Degree of integration: ratio between the degrees of freedom of the instrument and the input device

- The *Degree of integration* measures the ratio between the degrees of freedom of the instrument and the input device. A degree of one is typically most desirable as it corresponds to a natural mapping between input and output. If the ratio is above one, more degrees of freedom are controlled with an input device than the device provides, which often results in a complicated mapping. If the ratio is below one, the input device is

not used towards its full potential as some of its degrees of the freedom are ignored.

Degree of compatibility: similarity of physical actions and their response

- The *Degree of compatibility* measures the similarity between physical actions on the instrument and the response of the domain object. Direct manipulation actions have a high degree of compatibility as the actions have a direct and corresponding response of the object that is manipulated. Indirect manipulation actions, such as specifying changes via commands and values, have a low degree of compatibility as the actions that lead to the specification of the command are unrelated to the manipulation of the object.

Ubiquitous Instrumental Interaction

Ubiquitous instrumental interaction makes domain objects and instruments into freely exchangeable among people and devices

Klokmose and Beaudouin-Lafon [2009] extended the instrumental interaction model to explicitly consider multi-device interaction. This extension, called *Ubiquitous instrumental interaction*, turns instruments into explicit constructs that can be exchanged between users and devices. With this extension the authors pursued two major goals: The first goal is to enable distributed interfaces with fluid interaction across stationary and mobile devices. The second goal is to allow the dynamic configuration of those interfaces according to the available devices and user needs.

The VIGO architecture implements the concepts of ubiquitous instrumental ineraction

Based on this extension of the instrumental interaction model, Klokmose and Beaudouin-Lafon [2009] developed the *VIGO architecture* (Views, Instruments, Governors, Objects). In this architecture objects are passive data containers that expose their state through well-defined properties, which are synchronized across all devices in the environment. Views are device-dependent translations of these objects into modalities that are perceivable by the users but do not contain any form of interaction. Instead, instruments embody the possible interactions with objects and act as mediators between users and objects. At the same time, instruments are not coupled with specific objects but instead provide general functionality that can be dynamically attached to any existing objects and distributed among the devices of the environment. Finally, governors implement the application logic of objects by moderating and reacting to changes to the object properties from instruments. They are separate from objects and instruments to facilitate reuse of governors among different objects.

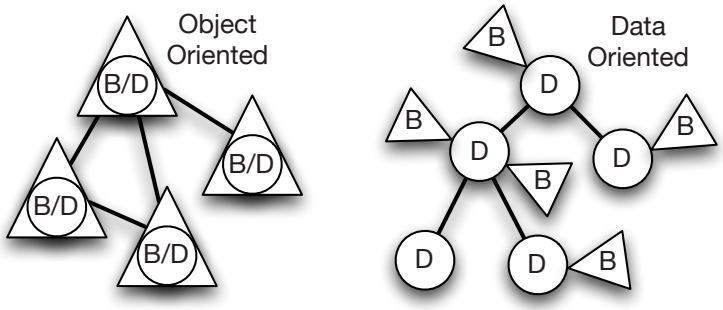

Figure 3.24: In object-orientated programming, data (D) and behavior (B) is merged into Objects. In data-oriented programming, data and behavior are separated and loosely coupled, allowing flexible coupling of data and behavior at runtime. Picture taken from Gjerlufsen et al. [2011].

Shared Substance

Shared Substance is a programming framework for distributed applications based on data-oriented programming

Shared Substance by Gjerlufsen et al. [2011] is a programming framework to support the development of interactive applications that can span multiple devices. It is based on a flexible notion of sharing that is achieved by decoupling data from behavior and sharing both data and behavior across distributed devices. These sharing mechanisms are implemented close to the programming language in the form of a *data-oriented programming* framework called Substance. Through data-oriented programming, sharing and distribution of data and behavior across multiple devices is enabled at the lowest possible level, making it as transparent as possible for developers.

Data-oriented programming separates data from behavior

Data-orientation is a programming model that pursues the fundamental separation of data from behavior. Figure 3.24 shows a comparison between object-oriented programming and data-oriented programming. In data-oriented programming, data is stored as properties of *nodes* and behavior is stored in *facets*. Nodes are organized in a tree such that each node can be uniquely identified by its path. Facets can be dynamically added to and removed from nodes at runtime to have the behavior associated to or dissociated from the data. As such, facets resemble aspects in aspect-oriented programming (see Kiczales et al. [1997]). However, aspects are typically applied globally to all instances of a class, while facets are applied only to specific data instances.

Substance is a reference implementation of data-oriented programming in Python

Substance is a reference implementation of the data-oriented programming model in the programming language Python. Nodes are simple data structures that may contain only values, representing the properties of the data.

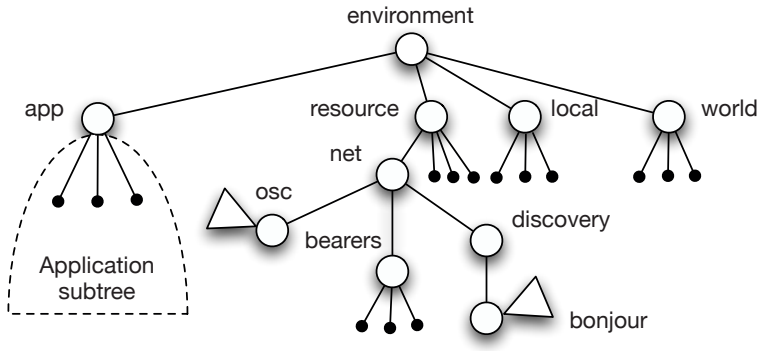

Figure 3.25: A Shared Substance environment contains four specific subtrees at the root level (circles are nodes and triangles are facets): App contains the application logic, resource contains references to local resources, local contains the local configuration, and world contains references to all discovered environments. Picture taken from Gjerlufsen et al. [2011].

Facets are Python objects that may contain handlers, publishers, and errors. Handlers are the equivalent of object-oriented methods, which can be called to trigger specific behavior. Publishers are used to emit events, which other facets can listen to and respond. Since facets are simple objects, they can inherit functionality from any public class, including graphics toolkits such as Qt[2].

Shared Substance is a distributed application model that organizes applications in environments and facilitates communication via shares

Shared Substance employs Substance to implement a distributed application model. A distributed application comprises several *environments*, which are remotely discoverable Shared Substance processes running on different devices. These environments communicate with one another and other applications through *shares*, which are publicly available subtrees.

Shared Substance environments follow a standard organization

Every Shared Substance environment contains four specific subtrees at the root level (see also Figure 3.25) The *app* subtree contains the application logic. The *resource* subtree contains representations of all locally available physical resources. The *local* subtree contains the local configuration of the environment. The *world* subtree contains references to the other discovered environments. Developers build Shared Substance applications by creating nodes, facets, and instruments inside the app subtree and connecting them with local and shared resources.

Shares can be accessed from remote environments in two ways: mounting and replication

Since everything in Substance is represented by trees, everything including nodes, facets, and resources can be shared. Shares are automatically discovered and appear in

[2] `http://qt-project.org`

the world subtree of every connected environment. Developers can access shares in two different ways: *mounting* and *replication*. Mounting a share gives an environment direct access to the remote data and functionality, similar to remote procedure calls. Replicating a share creates a synchronized copy of the shared subtree inside the environment. Providing two different sharing methods allows developers to select the most appropriate method for their tasks.

Instrumental interaction can drive multi-device interaction in the wild but is not compatible with today's application and system architectures.

At its core, Instrumental interaction suggests splitting interactive applications into instruments and domain objects. If this split is done, domain objects and instruments can be combined independent of one another. Using the approaches described in this section, both domain objects and instruments can then be made accessible across multiple devices. Thus, the principles of Instrumental interaction can lead to support for multi-device interaction in the wild. However, current applications and system cannot be split up into domain objects and instruments, which makes approaches based on Instrumental interaction incompatible with today's technology.

3.2.3 Runtime Program Migration and Distribution

The approaches in this section employ program migration at runtime to migrate or distribute an application across multiple devices. A program is migrated between devices by transferring the program and its current state to the target machine, where its operation is resumed.

Virtual Machines

Definition: *Virtual Machine*

A *virtual machine* is a software implementation of a machine. Originally, a virtual machine was defined as an "efficient, isolated duplicate of the real machine" [Popek and Goldberg, 1974, p. 413]. Today, this connection between physical and virtual machines is deprecated, and there are many virtual machines that have no direct correspondence to any physical hardware. Instead, the efficient execution of isolated processes has become the main application area for virtual machines.

System virtual machines execute multiple processes via an operating system; Process virtual machines execute a single process

There are two types of virtual machines: A *system virtual machine* implements a complete system platform, allowing the installation of an operating system (OS) that is then used to execute applications. The main benefit of employing system virtual machines is that multiple OS environments can be executed on a single physical machine, which is often used to create multiple virtual servers on a single physical server to increase redundancy and optimize resource usage. A *process virtual machine* or *language virtual machine* implements only the aspects of a machine to execute a single process or program without the need to install an operating system. Process virtual machines are mainly used to execute programs written in a high-level programming language like Java. To this end, the machine code created by the compiler is adjusted for and executed in the virtual machine as opposed to the physical machine, which can be used to make the program hardware-independent. Smith and Nair [2005] describe different architectural approaches to implement these two types of virtual machines.

Virtual machines can be migrated between different physical hosts at runtime

Liu et al. [2009] describe how system virtual machines can be migrated between different hosts a runtime. First, the system is *checkpointed*, i.e., the system state including the memory and the virtual disks of the virtual machine are stored persistently. This state is then transferred to the new host and restored in a compatible virtual machine. To minimize system downtime during migration, they suggest transferring the system state iteratively while the original system is still running. This process can also be used to migrate process virtual machines, as demonstrated by Suezawa [2000].

Virtual machines migration supports legacy applications but cannot not be adapted to the target device or distributed across multiple devices

Migrating virtual machines is probably the most generic way of migrating tasks between devices. There are no restrictions on what applications are supported other than the existence of a virtual machine for the hosting platform. Additionally, the applications do not need to be changed in any way. However, it is also the most inflexible approach to migrate tasks. The applications are restored at exactly the same state on the target device because essentially the memory area containing the application program is simply replicated on the target device. This inflexibility makes it unclear how to support distribution or to adapt the user interface of an application to the target device.

Distributed Objects

Distributed objects are objects residing in different processes that can communicate transparently with one another

In object-oriented software, *distributed objects* are program objects that can transparently communicate with one another even though they are not located within the same process or physical machine. The communication between remote objects relies *remote method invocation* (RMI), which gives the developer the illusion of calling a method on a local object while in fact the invocation occurs on the remote object. An RMI is initiated by calling a method on a *proxy object*, which forwards the invocation to the remote process, where it is executed.

Distributed objects are implemented in various programming languages, including a standard for cross-language distribution

Distributed objects can be implemented on top of object-oriented programming languages to assist the development of distributed systems. For example, Wollrath et al. [1996] describe an implementation of distributed objects in Java that tightly integrates into the Java object model. Integrating distributed objects into the language makes the distribution transparent but cannot be used to communicate between applications written in heterogeneous programming languages. To this end, the Common Object Request Broker Architecture (*CORBA*) by Vinoski [1997] defines a standard for language-independent distributed objects.

An application's user interface can be distributed by turning UI objects into distributed objects

Distributed objects can be used to drive toolkits that enable the development of distributed applications. For example, Melchior et al. [2009] present such a toolkit that turns all UI objects (widgets) contained in the toolkit into distributed objects. This way, the user interface of applications created with this toolkit can be distributed at the widget level, without the need to implement this distribution explicitly in the application. Applications created with the toolkit follow a three-layer architecture: The top layer is the application layer, which contains the custom application logic. The middle layer provides the migration and adaptation features in the form of an extension of the Tcl/TK[3] graphical user interface toolkit that augments standard widgets with distribution functionality. The lower layer implements distributed objects and additional support for live migration through the Mozart/OZ programming system (see van Roy [2004]).

[3] `http://tcl.tk`

Distributed objects assist the development of distributed applications but do not include mechanisms to configure or change this distribution at runtime

Distributed objects enable developers to partition applications into parts that can be distributed transparently across multiple devices. Toolkits, such as the one described above, can then make the distribution of the user interface and other well-defined objects transparent for the application developer. However, to benefit from these toolkits, the application must be developed with the toolkit, making a wide-spread adoption unlikely.

Automatic Application Partitioning

J-Orchestra augments compiled Java programs with distribution capabilities

Tilevich and Smaragdakis [2009] describe a system for *automatic application partitioning* of Java applications called *J-Orchestra*. Automatic application partitioning is the process of augmenting existing programs with distribution capabilities without changing the source code of the program or the application runtime. Instead, the distribution mechanisms are integrated into the program by changing the program binaries. This approach enables the transformation of legacy applications without access to source code into distributed applications.

J-Orchestra partitions a program into parts that can be distributed while preserving their ability to communicate with one another

The automatic partitioning process employed by J-Orchestra can be summarized as following: The tool queries the user for the parts of the program's code that should be augmented with distribution capabilities and then divides the program such that the indicated parts can run on distributed machines. In the case of Java applications, these parts are identified by selecting classes that are then augmented to become distributed. Any communication between the separated parts of the program is automatically turned into remote communication. To this end, any method calls between Java classes that are located in different parts of the program are replaced with remote method calls via proxy objects. Direct access to remote object properties is replaced by getter/setter method calls. In addition to program objects, all UI framework objects are augmented with distribution capabilities.

J-Orchestra allows users to turn legacy applications into distributed applications but does not support UI adaptation

Using J-Orchestra users can turn legacy applications into distributed applications and configure the distribution at runtime. However, it is unclear how the distributed user interface can be adapted to different device form factors or modalities as the distributed objects cannot be easily replaced by adapted objects. Nevertheless, "hacking" the runtime of applications is a promising approach to extend

legacy applications with new behavior, which is also pursued in chapter 6 to integrate state extraction into legacy applications.

Software Agents

Software agents are programs that can migrate between systems but do not include a user interface

Software agents are non-interactive programs that can be migrated at runtime between different systems. To migrate an agent from one system to another, the agent's program code is transferred to the new system, where it is executed by the local *agent server*. The state of an agent is stored in a *suitcase* and transmitted alongside the agent. In addition to the suitcase, the agent also receives a *briefing*, describing the new system where it is launched.

Migratory applications are interactive agents that can be migrated like Software agents

Bharat and Cardelli [1995] present a programming model called *Migratory applications* that allows interactive applications to be migrated between devices based on Software agent migration. To support interactive applications, the Software agent paradigm is extended to support the migration of user interfaces. The resulting procedure uses the following five steps to perform a program migration:

1. Verify with the agent server that the migration can occur.
2. Extract the state of the user interface into migratable objects.
3. Destroy the user interface and remove any links to the underlying application.
4. Remove any links to the local runtime (e.g., event handlers).
5. Add all migratable objects to the suitcase and execute the migration.

At the destination, the suitcase is extracted and the user interface is reconstructed by the agent server based on the information stored in the suitcase. During this reconstruction the links to the application and the local runtime are restored. In summary, the interactive application is migrated by migrating application logic separately from the user interface.

Visual Oblique is a distributed scripting language that includes mechanism to design user interfaces that can be migrated

The suggested procedure assumes that the application logic of the interactive application is developed as a Software agent, which can already be migrated. To this end, Bharat and Cardelli [1995] used *Visual Oblique*, an extension to the distributed scripting language *Oblique* (see Cardelli [1995]) that adds mechanism to create user interfaces for Software agents. Visual Oblique was created to facilitate the creation of interactive distributed applications as demonstrated by Bharat and Brown [1994]. It includes an editor that is used to visually design the user interface and generate custom Oblique code that creates the specified user interface at runtime. Additionally, Visual Oblique implements the mechanisms needed to inspect, store, and reconstruct a user interface for migration.

Approaches based on Software agents enable application migration but fundamentally challenge how applications are built

The presented approach enables the development of applications that can be migrated between multiple devices. Supported applications, however, must be developed as Software agents which is fundamentally different from how interactive applications are developed today. Thus, a wide-spread adoption of this technique appears unlikely.

Recombinant Computing

Recombinant computing strives at improving interoperability through three premises

Edwards et al. [2002] describe *Recombinant computing* as an approach to support interoperability among heterogeneous devices that requires minimal a priori knowledge of one another. The approach is based on the following three fundamental premises:

Agree on a small, fixed, and generic set of interfaces

1. Devices should agree on a small set of interfaces that are known by all parties and generic enough to serve multiple purposes.

Rely on mobile code to extend these interfaces

2. Mobile code, i.e., small programs that can be migrated between devices, can extend these interfaces with new behavior and turn them into domain-independent interfaces.

The user must be in control of how devices are connected

3. Due to the dynamic nature of domain-independent interfaces, it is impossible to predict how devices are combined in the future. Thus, the user must become the ultimate arbiter and control how and when devices are connected.

Recombinant computing challenges the fixed interfaces predominant in today's systems. The main argument underlying the approach is that it is impossible to predict how devices, applications, and services are combined, and thus this combination should be as flexible as possible and ultimately controlled by the user. In this thesis, this thought is reflected in the open nature of application state as discussed in the next chapter, which can be migrated between different applications to support greater flexibility when migrating tasks between devices.

The idea of Recombinant computing has inspired the flexible state exchange between applications discussed in the next chapter

3.2.4 Model-based Migration and Distribution

In *Model-driven engineering* software is developed based on an abstract representation of the application domain in the form of one or more *domain models*. A domain model provides a structural view of the key concepts and the relationships between all aspects of a particular problem domain. Since models are abstract representations of the software, they are independent of a specific device. Therefore, models can be used to develop multiple versions of the same application that are adapted to different devices.

In Model-driven engineering software development is guided by domain models

Model Transformation

The Transformation Environment for inteRactivE Systems representAtions (*TERESA*) by Mori et al. [2003] is an integrated development environment (IDE) that integrates the creation of user interaction models at different levels of abstraction into the development process. The abstraction levels that are considered by TERESA are: (1) Task models, identifying logical activities, (2) abstract (device-independent) user interfaces, (3) concrete (device-dependent) user interfaces, and (4) source code. The IDE also provides automatic transformations between these abstraction levels, which can be used to generate a user interface in a multi-step process. Each transformation in this process and the intermediate results can be customized to accommodate for different device properties.

TERESA is a programming environment that offers automatic transformations between different levels of abstraction of an application's user interface

Mori et al. [2003] describe how TERESA can be used to implement the user interface of a nomadic application, i.e., an application that can be executed on different devices with an adapted user interface. To this end, the developer first

TERESA aids the development of nomadic applications that can be adapted to different device modalities

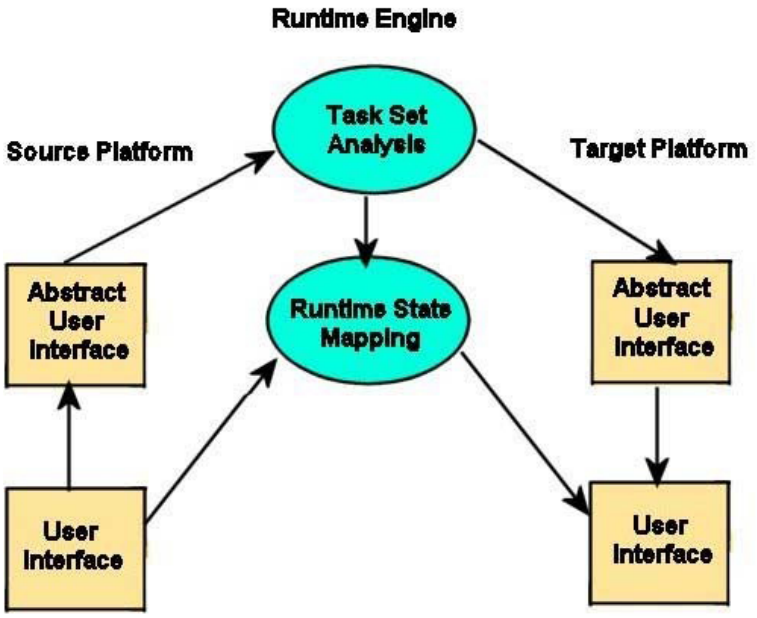

Figure 3.26: To transform the state between two potentially different user interfaces of a nomadic application, their properties are matched against each other and the active page is determined. Property matching is done via a special algorithm that considers the runtime state and platform data. The active page is determined from the aspect of the task model that the last-used user interface element represents. Picture taken from Bandelloni and Paternò [2004].

specifies which parts of the task model can be executed on which target platform. Then, user interfaces for these platforms are generated using TERESA's transformations, which can be further adapted to the circumstances of the specific device. At runtime, the application environment then selects the most appropriate user interface to be displayed on the device.

A nomadic application can be migrated between different devices by inspecting, transforming, and recreating the user interface

Bandelloni and Paternò [2004] extended above approach to allow users to seamlessly transition a nomadic application between different devices at runtime. To this end, a migration service was developed that performs the transition of the nomadic application. First, the service captures the runtime state of the nomadic application. Then, a new instance of the application is spawned on the target device. Finally, the application is configured to resume operation at the captured state.

The UI state of a nomadic application must be transformed during a transition to match the adapted user interface

Since the user interface of a nomadic application is adapted to its host device, the UI of a nomadic application can change significantly during a device transition. Consequently, the UI state of the nomadic application must be transformed to match the adapted UI. This transformation is enabled by the models that were previously used to generate the different UIs. In particular, the UI elements are matched against each other by identifying the aspects of the

task model that they represent. This process is illustrated in Figure 3.26.

Model transformations can be used to quickly generate diverse UIs but maintaining these model to reflect manual changes of the UI can be laborious

Automatic model transformations allow developers to quickly generate multiple models and concrete user interfaces for different devices. However, generated UIs often suffer from low usability. To address this issue, TERESA offers different ways to customize the UI at each level of abstraction. Yet, it is not clear how the different models can be maintained to reflect these customizations and enable the matching of widgets in different version of the UI.

Modeling Distribution

The development of distributed user interfaces can be assisted by a model of the distribution

Melchior et al. [2011] extended above approach by explicitly modeling the distribution scenario alongside the other domain models. This distribution scenario is created based on a model of the users and the environment, as well as a concrete user interface model for each supported device. These models are then connected using the following distribution primitives, which represent the basic operations to configure the distributed user interface:

- *Set* the value of a widget property.
- *Display* a widget on a given device.
- *Copy* a widget onto a different device.
- *Move* a widget, possibly to another device.
- *Replace* a widget with another widget.
- *Merge* multiple widgets into a composite widget.
- *Separate* a composite widgets into its parts.
- *Switch* the position of two widgets, possibly on different devices.
- *Distribute* multiple widgets into multiple containers by means of an external algorithm.

The distribution model alongside the concrete user interface models can be used to generate distributed applications and to configure the distribution at runtime

To demonstrate the application of distribution scenarios for the development of distributed systems, Melchior et al. [2011] developed a toolkit that automatically generates user interfaces and distribution capabilities from above models. Additionally, the distribution modeling language is made available as a console to the user to customize the configuration of the distributed user interface at runtime.

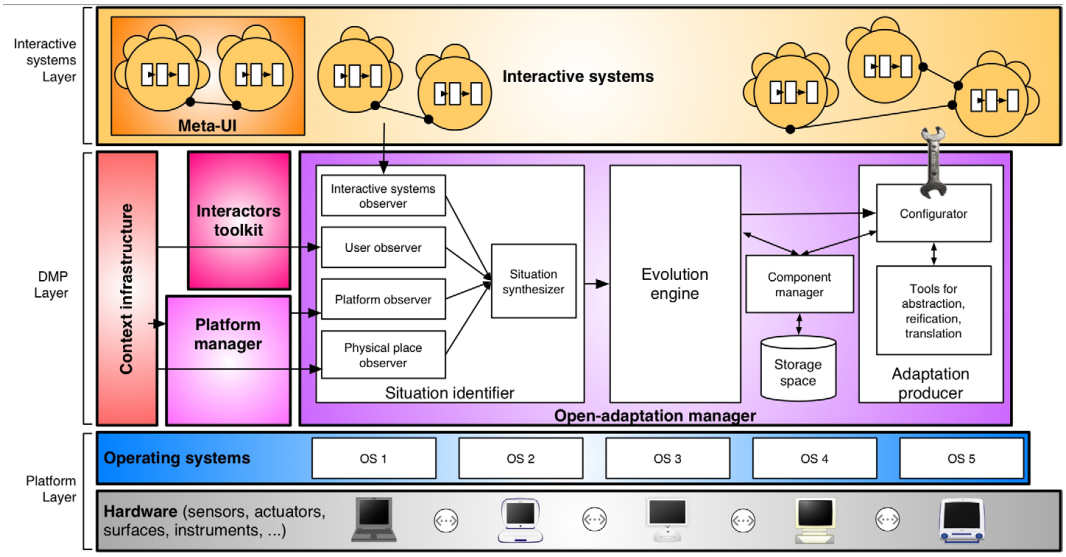

Figure 3.27: The CAMELEON-RT architecture reference model is structured into three layers of abstraction: The interactive system layer contains the applications that users interact with including the meta-UI. The DMP layer contains the middleware that maintains diverse models of the environment and supports dynamic formation of heterogeneous clusters including UI adaptation. The Platform layer includes the hardware and operating systems of the available devices. Picture taken from Balme et al. [2004].

CAMELEON-RT

CAMELEON-RT is an architecture reference model for that guides the design of systems supporting multi-device interaction

CAMELEON-RT by Balme et al. [2004] is an architecture reference model for solutions supporting distributed, migratable, and adaptive (plastic) user interfaces. It consists of three layers of abstraction as illustrated in Figure 3.27. The most important layer is the middle layer called Distribution-Migration-Plasticity (DMP) layer, which provides support for multi-device interaction in the form of three services:

Context infrastructure: maintain a model of the physical space

- The *context infrastructure* creates and maintains a model of the physical space based on the information gathered from physical sensors and available devices.

Platform manager and interaction toolkit: support resource discovery, platform independence, and UI distribution and migration

- The *platform manager* provides a layer of abstraction for the available devices to exchange information between each other. It provides automatic device discovery and hides the heterogeneity of the different devices from each other. The *interaction toolkit* provides a collection of interactive widgets that support UI distribution and migration.

- The *open-adaptation manager* determines when and how running applications should be migrated between or distributed across the available devices based on the information provided by the context infrastructure. This adaptation can be performed automatically according to learned or preset rules, or initiated by the user through the meta-UI.

Open-adaptation manager: trigger and support UI adaptation

Coutaz [2010] introduced two additional principles for the development of the interactive applications that make up the interaction layer:

Two principles guide the design of interactive applications for CAMELEON-RT:

1. The first principle suggests that close-adaptiveness (the ability of the application to adjust to situational changes) must cooperate with open-adaptiveness (adaptation beyond the ability of the application). To this end, the open-adaptation manager should have access to the models and mechanics used in the design process of the application to generate variations of the user interface.

Close-adaptiveness must cooperate with open-adaptiveness

2. This requirement leads to the second principle, which suggests that interactive applications must be represented as graphs of models that are related by mappings and transformations. These models are then used to generate any missing user interfaces for adaptation.

Interactive applications are composed of graphs of models

Like the approaches based on Software agents, model-based approaches fundamentally challenge the way software and particularly user interfaces are developed. Instead of shaping the final product directly, models of the product are created at varying levels of abstraction. Today's applications are typically designed the other way around: First, a concrete representation of the user interface is created to test and evaluate the proposed interactions. Then, the application is designed to match the proposed user interface. Even though it is not impossible to combine these two approaches, it is not straight-forward to do so efficiently.

Model-based approaches challenge the way software is developed

3.2.5 User Interface Migration and Distribution

Approaches presented in this section do not attempt to migrate or distribute the application process like the previous section but instead only migrate the application's user interface. To the user, migrating the user interface makes no

difference to migrating the application as the user interface is the only point of interaction that the user encounters.

Display Replication

The X Window System is a windowing system with transparent access to networked user interfaces

The *X Window System* (or *X11*) by Scheifler and Gettys [1986] is a windowing system for that provides a basis for rich, networked, and device-independent graphical user interfaces. Applications built with X11 can transparently access and utilize networked graphics displays, such that the application can run on a different device than its user interface. The communication between the application and the display server is defined by the X Window System core protocol. Through this protocol applications can create windows and draw graphics and text inside these windows. The display server visualizes these windows and forwards all user events, like mouse clicks and keyboard events, to the application.

User interace façades allow users to replicate and rearrange an application's graphical user interface

Stuerzlinger et al. [2006] presented a system called *User interface façades* that allows users to adapt and combine the user interfaces of X11 applications. User interface façades are created by selecting one or more existing user interface components to be replicated on a designated window. These replicated components can be freely arranged in the designated window to shape the UI to the current need. Through the replicated components the original application can be controlled as if operating directly on the application's UI.

Pixel Replication

Virtual Network Computing replicates a desktop environment at the pixel-level over the network

Virtual Network Computing (*VNC*) by Richardson et al. [1998] gives users access to an entire desktop environment from a remote computer. VNC implements a simple remote display protocol that replicates the rendered graphical user interface of the desktop environment over the network. This replication is done at the pixel-level, making it independent of the underlying operating system and device platform. Users interact with the replicated user interface through their local input devices, which generate events that are forwarded to the remote machine.

WinCuts by Tan et al. [2004] uses pixel replication to allow users to create copies of parts of a graphical user interface. To create a WinCut, the user selects a region of the screen to be replicated in a new window. The WinCut window can be move freely on the screen without affecting the linked region. Users can use WinCuts to rearrange and combine user interfaces of one or more running applications on a single computer. In addition, WinCuts can be distributed across networked devices, allowing users to display and arrange parts of a user interface on remote devices.

WinCuts enables users to replicate and share areas of the screen through pixel replication

IMPROving MDE's Potential to support Tasks that are genUine (*IMPROMPTU*) by Biehl et al. [2008] is an interaction framework for group collaboration in multi-display environments (MDEs). It uses pixel replication to replicate applications windows across multiple devices. These replicated windows are used to share access to an application among multiple workers and to visualize an application to the entire group by projecting it onto a large shared display.

IMPROMPTU uses pixel replication to augment regular applications with collaboration features

Web Application Migration

Web applications are applications that are hosted on a web server and accessed by one or more clients via a web browser. They are structured in three layers: presentation, application, and storage. The presentation layer is sent to the client and displayed in the web browser. It consists of a user interface specification in HTML that is often augmented with interactive elements specified in the scripting language JavaScript. The application layer runs on the web server and generates the user interface that is then sent to the web browser. The storage layer serves as a persistent storage for the application layer, typically in the form of a database.

Web applications are distributed applications that are accessed via a web browser

Web applications are accessed via a uniform resource locator (URL), which consists of the web server's Internet address and a path and query string that identifies the application and its state. This URL can be used to open the same web application from diverse clients. Consequently, a web application can be migrated between different devices by transferring its URL and reopening it in a local web browser. However, the URL cannot store all of the application's state. In particular, local storage (e.g., cookies), form data, UI manipulations that occurred at runtime, and JavaScript state are not or only partially captured by the

Web applications can be migrated by transferring their URL, but not all of the application's state is preserved this way

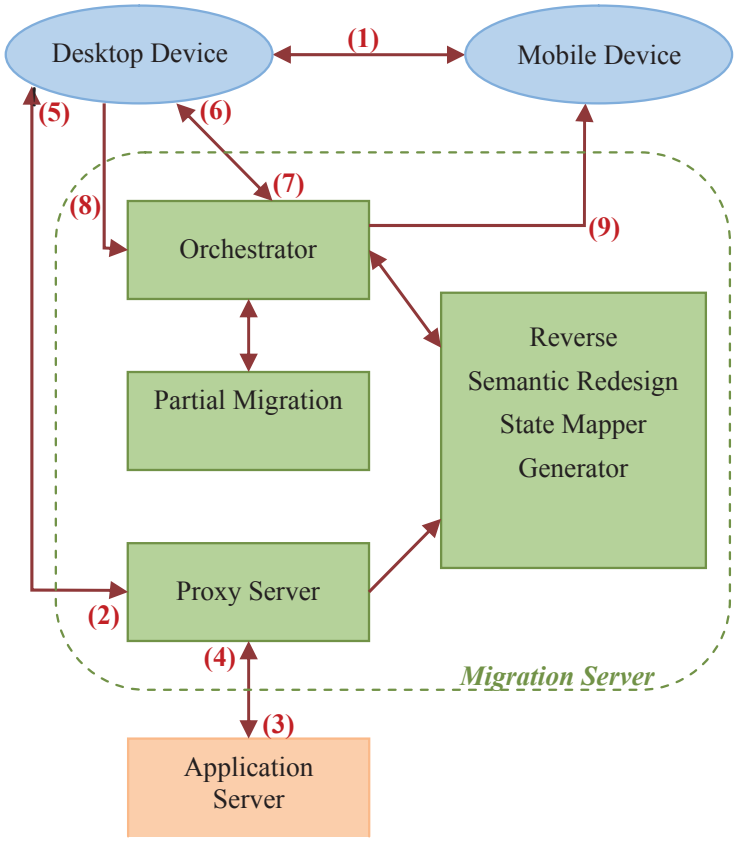

Figure 3.28: A web application can be migrated from a desktop to a mobile device in nine steps: (1) Devices are discovered. (2-5) The desktop client requests the user interface of the web application from the application server, which is mediated by a proxy server that injects code for tracking state. (6-7) To initiate a migration, the desktop client requests the logical structure of the web application from the orchestrator and presents it to the user to select the parts of the user interface that should be migrated. (8) The state of the web application is extracted through the previously injected code. (9) The redesigned web application is sent to the mobile device. Picture taken from Ghiani et al. [2010].

URL. This state is lost if a web application is migrated via its URL, which can break the seamlessness of a transition between devices.

Additional state can be considered in the migration of web applications by capturing key user interactions

Ghiani et al. [2010] created a system that allows users to migrate web applications while preserving more state information than is contained in the URL. In addition, users can restrict the migration to a specific part of the user interface, which allows users to focus on the essential task when migrating a large web application to small device such as a smartphone. The system uses a migration server to deduce the logical structure of the web application's user interface and extract its state. The deduced logical structure serves two purposes: It is presented to the user to select the parts that should be included in the migration and it is used as a basis to redesign the partial website and adapt it to the target device. The state is extracted via code injection, i.e., the web application's user interface is augmented with inspection code that captures and transmits the state

of all elements upon request. The captured state includes mainly user-entered form data, such as entered text and selected check boxes. The migration process is described in Figure 3.28.

JavaScript State

The migration server was extended by Bellucci et al. [2011] to preserve more state in the migration. The migration follows the same steps as described in Figure 3.28. However, step 8 is extended to transfer the complete program state of the user interface. This program state is composed of all global JavaScript variables, the complete document object model (DOM) tree, and special objects such as timers. These program objects are serialized including individual references between objects and transferred to the target device to perform the migration. Here, the state is restored by reconstructing all stored objects and connections. This migration of the complete JavaScript state is possible because of the open nature of JavaScript, where all objects can be inspected and most system functions can be overridden to track some of the inner workings of a JavaScript application.

3.2.6 Legacy Application Support

Several solutions have been proposed that integrate multi-device capabilities into existing applications. These solutions make use of a mix of techniques to query and control the state of legacy applications.

Pebbles

Pebbles supports the development of mobile user interfaces to control legacy applications

The *Pebbles* project by Myers [2001] allows users to control legacy applications on Microsoft Windows from a mobile device. Figure 3.29 gives an overview of the Pebbles architecture. One example application is the *SlideShow Commander*, a mobile application that can be used to control a Microsoft PowerPoint presentation running on a connected desktop computer. The mobile user interface displays the current slide and provides navigation buttons to jump to the next or the previous slide. Pebbles uses the Microsoft Windows *Common Object Model interface* to access and manipulate the state of Microsoft PowerPoint A different example allows users to scroll any system window by moving the finger on the touch screen of the mobile device. Pebbles

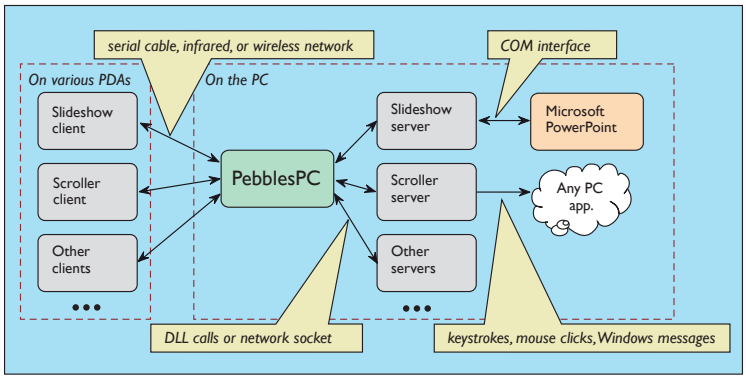

Figure 3.29: Pebbles supports the development of mobile user interfaces to control legacy applications on a desktop computer. To this end, the mobile applications communicate with legacy applications through different servers that simulate user events or utilize special programming interfaces that the applications provide. Picture taken from Myers [2001].

translates the finger movement into scroll events that are emitted to the currently active window.

Activity-based Computing

Activity-based computing extends the desktop environment with activity-based capabilities

Activity-based computing by Bardram et al. [2006] augments the Microsoft Windows desktop with activity-based capabilities. These capabilities allow users to organize their work according to activities, which can span multiple applications and documents. Activities are managed from the activity bar, which is located at the top of the window. Here, all running activities are listed, and the user can switch between different activities by clicking on them. Switching an activity suspends the current activity by closing all windows and applications and resumes the new activity by restoring the windows and applications of that activity.

The state of legacy applications is accessed via the COM interface or by changing the source code

To enable a seamless transition when resuming a task, the system extracts and stores the state of all applications associated with the task upon task suspension. This stored application state is then user to restore the applications to the stored state upon task resumption, making the transition seamless. The system employs two different methods to access and store the state of legacy applications: If the application implements the COM interface, a custom application wrapper can be developed to extract the application's state via this interface. If full access to the application's source code is available, the state extraction can be implemented directly in the application.

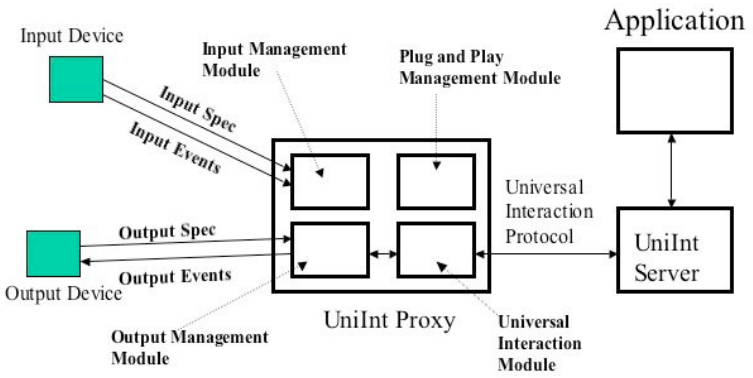

Figure 3.30: The Universal Interaction system allows users to control legacy applications from mobile devices by transferring their user interfaces and generating user events based on mobile input. Picture taken from Nakajima [2006].

CoWord

CoWord augments Microsoft Word with collaborative editing capabilities

CoWord by Xia et al. [2004] extends the word processing software Microsoft Word with collaborative document-editing capabilities. Multiple authors can use CoWord each on their own device to edit a document collaboratively such that any changes to the document on any of the connected devices are reflected on all other devices.

CoWord implements oOperation transformation on top of Word's COM interface

CoWord synchronizes documents across multiple devices through an implementation of the *Operational transformation algorithm* by Ellis and Gibbs [1989]. The algorithm ensures that concurrent editing operations yield a consistent result by transforming the operation parameters according to preceded operations. CoWord implements concurrent document editing on top of an abstract representation of the live Word document, which is based on Word's COM interface.

Legacy Applications in Ubicomp Systems

Nakajima [2006] describe an approach to reuse existing interactive applications in ubiquitous computing environments. The approach allows users to employ mobile devices as input and output devices for legacy applications. This dynamic assignment of input and output to applications is enabled through a middleware that mediates between mobile devices and legacy applications by converting input and output channels between the different device modalities.

The Universal Interaction system controls legacy applications by replicating the UI and simulating user events

Figure 3.30 gives an overview of the system architecture that was implemented following the proposed approach. For each legacy application a *UniInt server* is created to mediate between legacy applications and the *UniInt proxy* via the *Universal Interaction Protocol*. The different servers act as gateways to the applications, forwarding rendered user interfaces to the proxy and user events to the application. The proxy is responsible for maintaining the mapping between devices and applications, and for mediating and converting input and output between mobile devices and the UniInt servers. In particular, it forwards the rendered user interface to mobile devices with appropriate output capabilities and user events from these devices back to the legacy applications.

Approaches that replicate the user interface can be improved by augmenting the UI representation with meta-data

Dixon et al. [2011] show how pixel-based approaches, such as Universal Interaction system, can be improved by augmenting the pixel representation of the user interface with additional meta-data. They present a method to deduce the type and properties of a visible widget based on its pixel representation. Chang et al. [2011] improve this method by also considering the accessibility information underlying a user interface to improve text recognition and the extraction of non-visible user interface elements. Accessibility information is provided by the application developers through a standard programming interface that is used to drive assistive applications like screen readers. These approaches offer great potential to improve the reuse of legacy applications for multi-device interaction by allowing the middleware to gain a better understanding of the structure of the legacy application's user interface.

3.2.7 Logical Framework for Multi-Device User Interfaces

Paternò and Santoro [2012] presented a classification framework for systems supporting multi-device user interfaces. They have identified the following ten dimensions to reflect the various aspects of multi-device user interfaces:

UI Distribution: Is it possible to distribute the UI across multiple devices?

- *UI Distribution* analyzes whether the user interface of the system can be distributed across multiple devices. The distribution id *dynamic* if it can be changed at runtime. Otherwise, it is *static*.

Aspect \ Tool	Web Migration[1]	DeepShot[2]	Myngle[3]	P2P Distrib. UIs[4]	Dygimes[5]	Multimod. Distrib.[6]
Distribution	–	–	–	Dynamic	Dynamic	Dynamic
Migration	UI Elem., Func.	UI Elem.	History	UI Elem.	–	UI Elem.
Granularity	Entire UI, Groups	Entire UI, Groups	Entire UI	Entire UI, Groups	Entire UI, Groups	Entire UI, Groups
Trigger	Mixed	User	System	User	System	System
Sharing	Moving	–	–	Moving	–	–
Timing	Immediate	Mixed	Mixed	Immediate	Immediate	Immediate
Modalities	Transmod.	Monomod.	Monomod.	Monomod.	Monomod.	Multimod.
Generation	Runtime	Mixed	Runtime	Runtime	Mixed	Mixed
Adaptation	Transduc., Transfor.	Scaling	Scaling	Transduc.	Transduc., Transfor.	Transduc., Transfor.
Architecture	Client-Server	Client-Server	Client-Server	Peer-to-Peer	Client-Server	Client-Server

Table 3.1: Classification of different approaches for multi-device user interface: (1) Ghiani et al. [2012] (2) Chang and Li [2011] (3) Sohn et al. [2011] (4) Melchior et al. [2009] (5) Vandervelpen and Coninx [2004] (6) Blumendorf et al. [2010] Table taken from Paternò and Santoro [2012].

- *UI Migration* analyzes to what extent ongoing interactions can be seamlessly continued when changing devices. This continuation extent is described with the number of elements that are preserved when the UI is migrated.

UI Migration: Is there some continuity when changing devices?

- *UI Granularity* analyzes at what level of granularity a UI can be distributed or migrated. Possible values are: *Entire UI*, *Groups of UI elements*, *Single UI elements*, *Components of UI elements*

UI Granularity: At what granularity can the UI be distributed or migrated?

- *Trigger Activation Type* analyzes how multi-device interactions are triggered. If the trigger originates from the *user*, it can occur from the source device (*push*) or from the destination devices (*pull*). Otherwise, the *system* deduces appropriate times for transitions and executes them automatically. If the system approach

Trigger Activation Type: Is the cross-device interaction triggered by the user or the system?

relies on the user for a final decision, the trigger is called *mixed*.

Timing: Is the effect of a transition immediate or deferred?

- *Transition Timing* analyzes when the device change occurs after triggering a transition. If there is no delay between the effect and the trigger, the transition is *immediate*. Otherwise, the user can specify a target device ahead of time and the transition is *deferred*. If both immediate and delayed transitions are available, the timing is called *mixed*.

Interaction Modalities Involved: Are devices with different modalities supported?

- *Interaction Modalities Involved* analyzes whether the system supports transitions between devices with different interaction modalities. *Mono-modality* means that all devices must employ the same interaction modality. *Trans-modality* means that devices can employ different interaction modalities, but each device can employ only one modality at a time. *Multi-modality* means that devices can employ multiple interaction modalities at the same time.

UI Generation Phase: When is the user interface generated?

- *UI Generation Phase* analyzes when the distributed or migrated user interface is generated. The UI can be generated at *design-time* when implementing the system, or it can be generated dynamically at *runtime*. A *mixed* mode generates the UI dynamically at runtime based on information that was prepared at design-time.

UI Adaptation Aspects: How is the user interface adapted to different devices?

- *UI Adaptation Aspects* analyzes how the user interface is adapted to a new device. There are three different approaches to UI adaptation: *Scaling* adapts the size of the UI to the new device without changing anything else. *Transducing* adapts individual UI elements without changing the overall structure of the UI. *Transforming* modifies both the UI elements and the overall structure.

Architecture: What architectural strategy is pursued?

- *Multi-device System Architecture* analyzes the strategy with regard to the migration or distribution architecture that is pursued. A client-server architecture employs a central unit that manages all cross-device requests. A peer-to-peer architecture relies on the individual devices to negotiate the distribution and migration among themselves without a central unit.

Table 3.1 classifies six different approaches to multi-device user interfaces according to the presented ten dimensions.

3.3 Discussion

Systems designers must acknowledge the diversity and evolutionary nature of multi-device interaction techniques

The excerpt of interaction techniques for multi-device interaction presented in this thesis can only give a small insight into the available diversity of interaction techniques. In addition, there is a stable growth of this diversity consisting of several novel and creative interaction techniques that are suggested every year. The main lesson that can be learned from this list is to acknowledge the existing diversity and the evolutionary nature of multi-device interaction techniques. In consequence, systems must not only support as many of the existing techniques as possible but provide ways to extend and adapt the system to support new ways of interacting with multiple devices.

Previous system solutions were not designed to address the unique challenges of multi-device interaction in the wild

The analysis of previous approaches to enable multi-device interaction has revealed that none of them constitutes an ideal solution for the unique challenges of multi-device interaction in the wild. This is not surprising as none of the approaches were developed to address these challenges but instead focus on systems research, multi-device environments, or collaborative use. However, they all solve the core problem underlying multi-device interaction: to enable the migration and distribution of tasks across multiple devices. Thus, it is important to study and understand these approaches to extract successful and recurring patterns and adapt them to the domain of multi-device interaction in the wild.

A recurring pattern to enable multi-device operations is to use program or UI state as a representation of ongoing tasks

One such recurring pattern is the use of state to enable the seamless transition of tasks across multiple devices: Migratory applications transmit the state of the Software agent and the user interface in the suitcase. Model-based techniques extract and map the state across different versions of a user interface. Deep Shot relies on an explicit programming interface to access the state of a third-party application. In all of these approaches the state of a program or user interface is used as a snapshot representation of the ongoing task. The task is then migrated by extracting the state, transferring it to the target device, and restoring it in a new application process, potentially with an adapted user interface.

Similar to how state is used to represent tasks, the file is used to represent content, which has lead to a separation of authoring and management tools for content

This concept of using state as a persistent container for ongoing tasks is similar to the how the file is used as a container for user content. The file, however, has evolved beyond a mere technical means – it has become a ubiquitous conceptual model that aids designers and users alike when creating or using interactive systems. In particular, the file facilitates the separation of content authoring from content management: Many of today's tools for the consumption, authoring, and editing of content are completely separated from the tools to organize and share this content. For example, the Microsoft Office[4] suite provides versatile tools to edit documents of various types but does not include many facilities to organize and share these documents. Instead, users save the documents to files and use external tools, such as the Windows Explorer or Dropbox[5], to organize and share their content. This separation is possible because users understand that content is stored and consequently represented by files.

Applying the properties of the file to state offers compelling opportunities for the domain of multi-device interaction in the wild

A separation between using task applications and managing task applications is also beneficial for multi-device interaction in the wild. This separation would allow designers to create and evolve task applications separate from task management applications and vice versa. At the same time, it would allow users to select the most appropriate combination of task applications and task management applications independent of one another. The next section explores how this separation can be conceptualized using application state as a conceptual model to represent tasks.

[4] http://office.microsoft.com
[5] http://dropbox.com

Chapter 4

Interacting with State

There is a lack of support for migrating and distributing tasks across multiple devices

As established in chapter 2 users are confronted with more and more interactive devices and benefit from using them in concert to work on their everyday tasks. However, current interactive systems do not provide any general concepts for combining multiple devices towards a common task. If multi-device operations are not explicitly considered by an application, users must manually coordinate the employed devices to address the given task.

Digital content can be transferred and distributed across devices via the file

Digital content, on the other hand, can be distributed across multiple devices independent of the application used to author or gather the content. Most of the applications used to gather or create digital content can save this content to files. These files can then be organized and distributed through other applications dedicated to managing files. The file acts as a general concept for handling digital content by separating the acquisition and consumption of content from its management.

The properties of the file can be applied to tasks by making task state persistent

This chapter investigates the use of application state as a container to make the state of a user task persistent. Application state captures the interaction state of an application in a way that it can be stored persistently and transferred to other devices. Similar to how the file acts as a general concept for managing digital content, application state serves as a general concept to manage running applications by separating the execution of the application from its management including cross-device operations.

Chapter outline

The chapter starts with a description of the conceptual model as a design tool. Then the conceptual model of the file and its interaction properties are described in detail. Afterwards, the chapter describes the conceptual model of application state and discusses how it addresses the needs of multi-device interaction in the wild. This effectiveness of the conceptual model as a design tool is evaluated in two

ways: First, the results of a brainstorming session with students shows how designers can develop innovative interaction techniques based on application state. Second, a prototype called Tangible Windows that reflects the properties of application state at the window level is described and evaluated in four scenarios.

4.1 Conceptual Model

Definition: *Conceptual model*

Norman [1986] defines a conceptual model as a mental model that is formed by humans to explain how a system works. Conceptual models are used to predict possible actions of a system and explain their impact on the environment. Having an accurate conceptual model of a system allows users to quickly find desirable actions and accurately predict their effect, leading to almost intuitive usage of the system. Having an inaccurate conceptual model, on the other hand, leads to user frustration because desirable actions are hard to find or mixed up with other actions leading to erroneous and confusing results. Thus, it is one of the most important tasks of a system designer to communicate a good conceptual model of the system to the users.

Three different conceptualizations must be considered in the design of interactive systems:

Design model

- The *Design model* is the designer's conceptualization of the system. It describes the true capabilities of the system and how the system functions as a whole.

System image

- The *System image* is the image that is transmitted by the physical aspects of the system to its users. It is shaped by the physical appearance of the system, its documentation and other instructional materials, and the nature of the interaction.

User model

- The *User model* is the model of the system that is constructed by the user. This model is based on the user's interpretation of the System image. It describes how the user understands the system and shapes the user's predictions and assumptions about the system.

The conceptual model accompanies the design and usage of an interactive system

Figure 4.1 illustrates the interplay between these different conceptualizations of an interactive system. The designer first develops the Design model of the system. Based on this model the designer then creates the system and with it the System image. When the system is used for the first

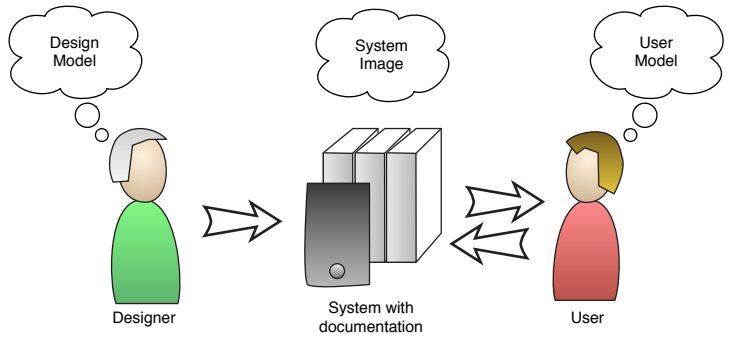

Figure 4.1: A conceptual model of a system consists of the Design model, the System image, and the User model.

time, the user interprets the system by means of the System image and creates the User model. This model is then used to guide all user actions when using the system.

User models are reused for systems that appear similar

When users have formed a User model for a system, this model is often applied to similar systems. Designers can use this fact to ease the introduction of a new system by grounding its Design model in a previous system and employing a similar Design model. Through this similarity, users can apply their known User model to the new system and are likely to understand the system quicker.

The conceptual model introduced in this chapter is intended as an aid for designers to build systems that support multi-device interaction in the wild. This model helps designers build consistent systems that users have to learn only once and can subsequently apply their model to the other systems following the same model.

4.2 The File

The basic conceptual model of current systems: Users manipulate digital content via task applications and manage and distribute it via the file

In typical interactive systems today users perform tasks by interacting with digital content through digital tools. These tools are manifested in applications, which can be launched and interacted with through the application's user interface. Digital content, on the other hand, is manifested in files. Typically, users can instruct applications to save the active digital content into a file, which stores the digital content persistently. Later, users can instruct the same or a different application to load the content from the file and restore the digital content. Figure 4.2 illustrates this conceptual model.

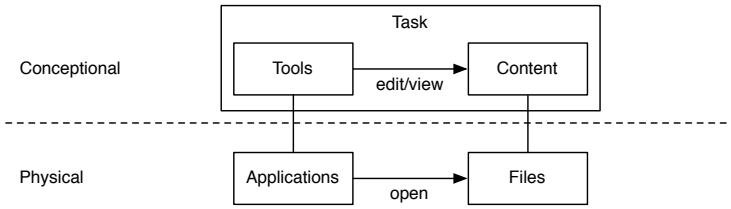

Figure 4.2: In current systems users perform tasks by interacting with digital tools to view and manipulate digital content. These tools are manifested in applications and digital content is manifested in files. Tasks are not manifested in this system.

History of the file

The term *file* originates from the domain of clerical work, where physical files are used to organize paper documents. In the context of computing the file was first mentioned in an advertisement in Popular Science magazine [1950] to refer to a mechanism to store digital information in a memory tube. Later, the hardware used to store information, such as punch cards and physical drives, were sometimes called files. Since then, the file has become the predominant concept to represent stored information in almost all computing platforms.

Definition: *File in computing*

A file in computing is "a collection of data, programs, etc. stored in a computer's memory or on a storage device under a single identifying name" (*File in computing*, Oxford English Dictionary Online. Oxford University Press, n.d. 1st December, 2013) As such, a file can be used as a universal container for digital content: Any type of digital content can be stored in a file and later restored with the guarantee that the restored content is the same as the stored content.

The file turns the concept of digital content into a first-class interactive object

In addition to these technical properties, a file acts as a reification of digital content: It turns the abstract concept of content into a first-class interactive object. Since this conceptual model is used in most computing systems today, many users have gained a profound understanding of the model by now: Experienced computer users understand that the file contains digital information, which can be restored by opening the file with an application. In particular, users are aware of the distinction between applications and files, allowing them to open a file selectively in the most appropriate application for any given task. Additionally, users can organize and distribute their digital content by organizing and distributing files.

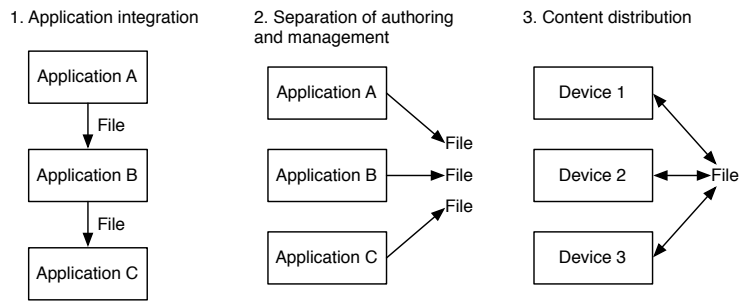

Figure 4.3: Files offer three distinct properties: First, they integrate applications into task flows by allowing multiple applications to work on the same content. Second, they separate content authoring from content management. Third, they allow content to be distributed across devices and people.

Changes to files and file systems can be observed through file system events

Modern file systems allow third-party applications to observe files and receive notification events when files are changed. These events are triggered, when files are relocated, copied, or deleted, or when content is written to a file. Through these events, applications can respond to changes to specific files or collections of files organized in folders. For example, cloud synchronization tools make use of file system events to synchronize files located in specific folders with a cloud representation of these folders. This way, all changes to files contained in the folder can be replicated on all devices connected to the cloud, making these folders appear synchronized on all devices.

4.2.1 Properties of the File

The file offers several unique properties for users and designers

Files offer many unique properties, which help users in performing their everyday tasks and ease the development of interactive applications: First, files allow users to work on digital content with multiple applications. This allows designers to design applications for specific aspects of a task and rely on other applications for supplemental functions. Second, files allow digital content to be organized in a central system independent of the source of the content. In consequence, designers do not need to explicitly consider how content is organized in the application and users can store everything in one place. Finally, files allow content to be distributed across multiple devices and shared among multiple people. Based on the file designers can create innovative systems that enhance digital content access for various situations, which users can directly benefit from. Figure 4.3 illustrates these properties.

Files allow multiple applications to work on the same digital content

Files can be accessed by any application running on the same device. Therefore, files can be used to share content between multiple applications and work collaboratively on the content. This has lead to the design of many small applications that are dedicated to perform very specific tasks, which are combined with other applications to create a complete task flow. A good example of such a specific application is OmniGraffle[1], which allows users to visually layout a diagram. OmniGraffle can open files containing vector graphics, e.g., created in Adobe Illustator[2] and then export a diagram to be embedded in a document, e.g., authored with LaTeX[3]. These small applications are easier to create and maintain than complex systems that try to address all aspects of the task. At the same time, users can selectively pick the most appropriate applications for any given aspect of a task and combine these applications opportunistically. For example, when editing images for a presentation in some cases a vector-based application is preferable to a pixel-based application, while in other cases the situation is reversed. Allowing applications to be interchanged this way gives users greater flexibility when choosing the tools for their tasks.

Files separate content management from content authoring

Applications that are designed to visualize or manipulate digital content typically do not include mechanisms to organize this content. Instead, the content is saved as a file and organized in other applications, which are specifically designed to organize files. This way, designers do not need to consider the organization of content when designing applications, which eases the overall development of applications. At the same time, designers can create innovative new ways to organize content by creating applications that organize files. An example of such an application is Yojimbo[4], which lets users organize files with tags, labels, and collections. These applications can then be used to organize all content that users already have in the form of files. Users benefit from this separation because they can organize their digital content in a single system, allowing them to keep content together that is semantically related independent of its type.

[1] http://omnigroup.com/omnigraffle
[2] http://adobe.com/illustrator
[3] http://latex-project.org
[4] http://barebones.com/products/yojimbo

Files can also be used to transmit digital information to other devices. This transmission is done through special communication applications. For example, many email clients allow users to attach files to an email and send them alongside the email. Another example is a network drive, which can be accessed from remote devices to exchange files with other devices. Nowadays, cloud sharing services are becoming more and more common, which transmit files to a central server "in the cloud" and distribute the file from there to all connected devices. Through the file, designers have created and are likely to continue creating innovative ways of sharing everyday content among people and devices. Users directly benefit from these innovations, because they can employ these new techniques with their existing content.

Files can be used to distribute content across people and devices

The listed properties of the file are especially useful for multi-device interaction in the wild. Files can be transferred between different devices using a large variety of tools, giving users the flexibility to arrange content and devices as needed. In addition, the separation of digital content from authoring tools enabled by the file allows multi-device interaction techniques for distributing content to be developed independent of the applications used to view and edit the content. Finally, the ability of different applications to work on the same content through the file facilitates the use of applications that are adapted to the new device when changing devices.

The properties of the file facilitate multi-device interaction in the wild

4.3 Application State

In the conceptual model of the file, there is no manifestation of tasks. There is no concept that allows users to suspend an ongoing task and store its state persistently. Files only store the state of the digital content, and applications only store the tools of interaction, not their state. The concept introduced in this thesis fills this gap by providing a manifestation of tasks in the form of a persistent container called *application state*. Application state enables the extraction, storage, and restoration of the state of a task that is performed in a single application including the state of the application's user interface and the digital content included in the task. Similar to how the file can be used to store digital content, application state can be used to store the state of a task: Users save the state of a task by saving the state of all

The conceptual model introduced in this thesis: Users migrate and coordinate tasks across multiple devices by transferring and distributing the state of task applications

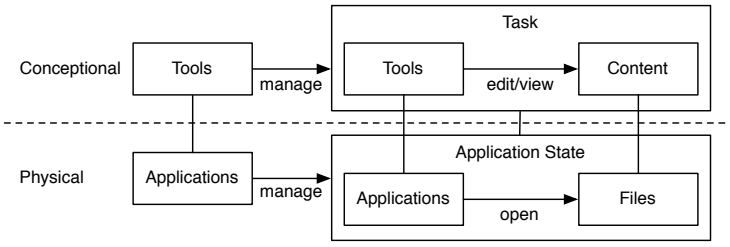

Figure 4.4: Application state extends the conceptual model of current systems by giving tasks a concrete representation in the form of application state. Users can manage application state by interacting with tools designed to manage application state.

applications included in the task and subsequently restore the task by restoring all applications. Figure 4.4 illustrates the conceptual model of application state.

Definition: Application state

In the context of this thesis application state is defined as a persistent container for the aspects of an application that represent the state of an ongoing user task. In particular, application state contains the state of the user interface and the related user content at the time of extraction.

Application state is different from program state

The definition of application state in this thesis is different from the concept of program state in the computer science literature. Program state includes all stored information, including the stack, the hash, and all program pointers. Consequently, program state also captures all aspects of application state. However, program state is tightly coupled with the application and its underlying operating system, making it hard to migrate the state to another application or platform. In addition, program state contains no semantic knowledge about the program and thus does not support breaking down the state into those aspects related to specific tasks. Application state, on the other hand, captures everything that is relevant to the user because it stores all aspects of the task, while potentially preserving independence of a specific system and application by storing the state in a semantically meaningful way.

Application state turns the concept of a running task into a first-class interactive object

Similar to how the concept of the file turns digital content into a first-class interactive object, application state turns the state of an ongoing task into a first-class interactive object. At any point in time, users can store the state of a running application into a persistent container. This container can be managed through special state management tools, similar to how files are managed through special tools. In particular, these tools allow users to organize application state in an organizational system and transfer it between

multiple devices. At a later point in time, the application state can be restored from the information stored in the persistent container, allowing the user to resume operation at the exact point where the task was interrupted.

Tasks can span multiple applications and applications can span multiple tasks

Since tasks can span multiple applications and applications can be used to work on multiple tasks in parallel, it is important to consider application state at different levels of granularity. Multiple application states can be aggregated to represent tasks that span multiple applications. These aggregated application states act as a composite, i.e., they allow the same operations as a basic application state. At the same time, application state can be broken down into the parts of the state that reflect individual tasks that are currently executed in the application. This way, users can extract only the part of the application's state that represents the task that they are currently concerned with. These partial application states can then be aggregated to form a composition of states representing a task that spans multiple applications unexclusively.

Changes to application state can be observed through state change events

State management tools can register to receive change events for specific application states. These events inform the tool about changes to the state of the running application, which can be used to maintain a synchronized, persistent copy of the application state. Like application state, state change events can be transferred to remote devices, allowing application state to be synchronized across these devices. Additionally, state change events can be used to synchronize one application with multiple replicas of that application running on different devices: First, the state of the original application is extracted, transferred, and restored on the devices that should host the replicas. Then, all changes to the original application are forwarded to the connected applications.

4.3.1 Properties of Application State

The previously mentioned properties of the file can be applied to application state

The properties of the file discussed in the previous section can be applied to the concept of application state. First, similar to how the file allows combining applications to work on the same digital content, application state can be used to combine applications to work on the same task. Based on application state designers can create applications that perform only a specific aspect of a task and seamlessly integrate into other applications that address the other aspects

of the task. Second, similar to how files allow the separation of managing and editing digital content, application state can be used to separate working on tasks from managing tasks. Designers can create special applications that maintain the states of tasks and allow task switching and migration independent of the applications used for the task. Third, similar to how files allow content to be distributed among multiple devices and people, application state can be used to distribute tasks among devices and people. By transferring the state of a task to another device, the task can be migrated to that device. By synchronizing the state across multiple devices, multiple devices can be combined to work on a common task in parallel.

Multiple applications can be combined to work on the same task by sharing application state

Similar to how files can be used to employ multiple applications to work on digital content, application state can be used to seamlessly combine multiple applications towards a common task. Application state can be restored in diverse applications if the applications share an understanding of the stored state. Application state can be shared in two ways: sequentially and simultaneously. In the sequential case, the state is extracted from one application and restored in another application, allowing the user to transition between the two applications without loosing the task state. In the simultaneous case, multiple applications operate on a shared, synchronized state such that any change to any application is reflected in all other applications.

Application state separates task management from working on tasks

Application state enables the separation between task applications and task management. With application state users employ their typical applications to work on their tasks. To manage their tasks, however, they use specialized task manager applications, which operate on the application state extracted from the task applications. Based on this separation designers can develop task applications separate from task management applications, allowing them to ignore aspects of task management in the former and aspects task execution in the latter. Users, on the other hand, can opportunistically combine task applications with task managers according to their needs.

Tasks can be distributed among devices and people by distributing application state

Finally, application state enables ongoing tasks to be transferred between different computing devices. To this end, the state of the original application is extracted and stored in the persistent application state container. Then, the persistent container is transferred to the target device and restored in an appropriate application. Since the state of the

task is conserved during the transfer, users can seamlessly continue working on the task on the target devices. Additionally, tasks can be synchronized across multiple devices by also transferring change events to the remote devices. This way, the states of applications running on multiple devices can be kept in synchronization allowing the applications to be coordinated to work on a common task.

4.4 Multi-device Interaction in the Wild with Application State

Application state addresses all work modes and challenges of multi-device interaction in the wild

Application state can be used to enable support for multi-device interaction in the wild. As usual, users can employ their devices individually to work on distinct tasks sequentially or simultaneously. Users can migrate tasks between devices by migrating application state to work with multiple devices on a common task sequentially. Users can coordinate applications running on multiple devices through a shared and synchronized application state to work with multiple devices on a task simultaneously. With application state these multi-device interactions are supported in a matter that addresses the additional challenges of multi-device interaction in the wild as described in chapter 2: Multiple devices can be opportunistically rearranged, transitions are robust, and they can be performed in ad-hoc situations.

Application state enables the opportunistic rearrangement of devices and tasks

Based on application state, designers can create innovative interaction techniques that seamlessly integrate into the work situation of users and adapt to their needs. Since state managers can initiate and perform the entire process of migrating applications between multiple devices, the user does not need to be involved in this process. At the same time, application state captures everything that is relevant to restore the appearance of the application at a later time, making a device transition seamless because the application can resume operation with the same appearance. In consequence, interaction techniques based on application state can be used efficiently, and a device migration appears seamless to the user. These two properties of application state eliminate the typical barriers for users that keep them from opportunistically rearranging their devices and tasks.

Device transitions based on application state are robust

When an application is migrated between devices, there must be an active network link between these devices to transfer the application state. Once the application state is transferred and restored, however, this network link is no longer needed. The target application runs independent of the original device and application, allowing the user to designate that device to other tasks or remove it from the setup entirely. The transition between devices based on application state is robust.

Application state transitions can be used in ad-hoc situations

Interaction techniques based on application state that initiate and control application migration can derive a large benefit from having access to contextual information through designated infrastructure as it exists, e.g., in active spaces. However, this contextual information is only optional. To migrate an application between devices, there are only two basic requirements: It must be possible to discover the target device, and both devices must be able to communicate with one another. These requirements can be realized with technologies that are entirely embedded in the devices, such as Bluetooth or Ad-hoc wireless networking. Thus, task transitions based on application state can be performed independent of any external infrastructure and effectively support ad-hoc situations.

4.5 Validation

The effectiveness of the conceptual model of application state was measured in two ways: as a tool for designers and as a an aid for users

The conceptual model presented in this chapter pursues two main goals: First, it serves the designer as a tool to shape novel interaction concepts in a way that users can easily understand and use. Second, it serves as an effective and reusable model for users to enabling multi-device interaction in the wild. To evaluate the conceptual model, its effectiveness towards these two goals is measured. To this end, the effectiveness conceptual model of application state was evaluated in two ways: First, a workshop was conducted to evaluate the ability of designers to understand and use the conceptual model to design new multi-device interaction techniques for current systems. Second, several prototype systems have been implemented and evaluated with users to test the conceptual model in realistic situations with users.

4.5.1 Design Workshop

To measure the effectiveness of application state as a design tool, a student design workshop was organized. In the workshop, 20 students from computer science were introduced to the conceptual model of application state and encouraged to conceive new interaction techniques based on application state.

In the design workshop, students were asked to brainstorm new interaction techniques based on application state

The conceptual model of application state was introduced in a ten minute presentation held by the author at the beginning of the workshop. The presentation included a demonstration of the Nomadic Desktop prototype described in chapter 6. After the presentation, the students were asked to form groups of 4-6 participants and brainstorm new ways of using interactive systems based on application state.

The students showed a solid understanding of application state and were able to conceive several novel interaction techniques based on application state

All of the student groups quickly understood the presented concept and were able to conceive diverse and novel ways of interacting with tasks through application state. For example, one group suggested a gesture similar to Superflick (see 3.1.2), which was used to transition the active task from a mobile device to another device in the direction of the gesture. Another group suggested allowing users to "pull" a task from a previously used device to the current device and thus continue the task that was interrupted on the other device. The term "application state" was quickly adopted by the students in their discussions, which strengthens above observation that the concept was readily understood and accepted by the students.

The students also conceived new ways of managing tasks on a single or across multiple devices

The students also raised several ideas for ways of managing tasks by means of application state. For example, one group suggested using application state as a template to capture a desirable initial state of an application to be reused as a starting point for recurring activities. Another group suggested allowing users to move dialogs from complex software products to external devices and thus enable quicker optimization of parameters by removing the steps needed to confirm and reopen the dialog. In total, more than 20 ideas were presented during the design workshop, many of which extending our existing interactive systems with novel ways of managing devices and tasks that enable new ways of operating our devices. In conclusion, the workshop showed that application state has the potential to generate new ideas for single and multi-device interaction.

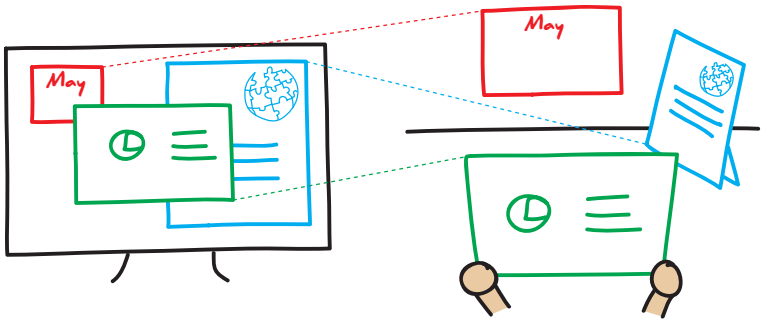

Figure 4.5: Tangible Windows are portable devices that represent virtual windows in the physical world

4.5.2 Tangible Windows

Tangible Windows turn windows into first-class interactive objects that are manifested in portable devices

Tangible Windows is a multi-device interaction concept that was developed based on application state. Tangible Windows turn the virtual windows of a window-based operating system into first-class interactive objects that users can exchange between devices. The window becomes tangible once it is transferred to a mobile device, where the device itself becomes the manifestation of the window. The concept is illustrated in Figure 4.5.

Virtual windows allow users to arrange multiple applications on a single screen

In a window-based operating system applications are visualized in windows, which are rectangular areas on the computer screen that contain the application's user interface. Multiple applications can be active in parallel by showing one or more distinct windows for each application on the screen. Users interact with the applications through their user interfaces inside the windows. Each window also contains a frame that allows users to move, resize, and minimize the window.

Tangible Windows embody the state of an application that is related to a single virtual window

Many applications distribute their user interface into multiple virtual windows that are logically organized. For example, most word processors use one window for each open document, enabling users to open and arrange multiple documents in parallel on their screen. Tangible Windows make use of this implicit organization of the application by allowing users to transfer the aspects of an application that are related to a single window to a separate device. In terms of application state, Tangible Windows capture, transfer, and restore the application state related to a single window.

With Tangible Windows portable devices become the manifestation of virtual windows

Technically, most mobile operating systems, including Android and iOS, are window-based, but they can show at most one window at a time. This window possesses no

user interface and thus cannot be moved or resized. Instead, the window size is fixed to the screen size such that each application always uses the entire screen of the device. In consequence, when transferring a virtual window from a desktop computer to a mobile device, the window user interface disappears and the application user interface fills the whole screen. Instead of showing a virtual window, the portable device becomes a tangible manifestation of the window.

Tangible Windows explores the use of tablet-sized displays in an office environment

The Tangible Windows prototype was designed to simulate an office environment where tablet-sized displays are ubiquitously available and can be opportunistically used as Tangible Windows. The prototype design and evaluation focuses on exploring multi-device interaction between tablet-sized displays in the form of sequential and simultaneous use. To this end, the prototype introduces six operations to facilitate the arrangement and coordination of Tangible Windows, which are accessible from a designated bar at the side of the device.

The Tangible Windows operations were evaluated in a user study

The understanding and utility of these operations was evaluated in a user study, where 14 participants performed a planning task with the prototype. The prototype outperformed a reference setup on a desktop computer in terms of user satisfaction and revealed interesting strategies for dealing with Tangible Windows. The prototype and the study are described in detail in a technical report by Diehl and Borchers [2013].

System Design

The Tangible Windows prototype provides six operations to allow users to coordinate their applications across multiple tablet-sized devices:

Copy/Retrieve transfers applications between devices

- *Copy* replicates the active application on another device while preserving the application's state. *Retrieve* represents the inverse of copy – it allows the user to replicate a remote application on the active device. The target device is selected by holding the *Target* button on that device.

Link synchronizes the appearance of two devices

- *Link* replicates the active application on the target device and synchronizes the appearance of both devices by keeping the application state synchronized. Users

can interact with the application running on either device and see the changes reflected on the other device. Tapping *Target* on either device or transferring another application state cancels the link.

Back/Forward gives users quick access to previously used applications

- *Back* reveals the previously used application on the same device. *Forward* reverts the back command. These commands are included to allow users to quickly undo a multi-device operation.

Home allows users to launch new applications

- Tapping *home* resets the application to the home screen, showing a list of all available applications, which can be launched by tapping on them. This command serves as the entry point to launching new applications, similar to the home button of many popular smartphones and tablet computers. The built-in home button could not be used as it would close the Tangible Windows application.

Target is used to specify the target device for a multi-device operation

- Holding the *Target* button marks the active device as a target for a multi-device operation. The operation is initiated by tapping copy, retrieve, or link on another device. Additionally, target can be used to initiate a remote operation. By holding target while performing a compatible operation in an application running on a separate device, the application is transferred to the target device and the application operation is executed there. This operation allows users to interact with an application and see the effect on a different device, similar to the "open link in new window" operation that exists in many web browsers.

The Tangible Windows prototype is an iPad application that shows various task applications next to the Tangible Windows bar

The Tangible Windows prototype was implemented as an Apple iPad application. After launching the Tangible Windows application, the home screen is shown on the main part of the screen. From the home screen, users can launch any of the included task applications by tapping on the application icon. Once an application is launched, the home screen is replaced with the user interface of the application, and the user can operate the application. The Tangible Windows bar, which is always located at the side of the device, provides several buttons to access the multi-device operations discussed above. Figure 4.6 shows the Tangible Windows prototype with the home screen active.

Figure 4.6: The Tangible Windows prototype shows a bar next to any task applications that is used to trigger multi-device operations.

Several task applications with limited functionality were integrated into the prototype

The following task applications were developed: A maps application based on http://maps.google.com[5], a notes application, an email client, and a calendar application. Additionally, several popular web sites were directly accessible from the home screen. The applications were only implemented as far as necessary to support the tasks of the user study: The maps application has two different icons, which focus the map onto a different area of the world. The notes application can only show a single note document. The email and calendar applications consist of several screen shots of the native iOS applications that were adapted towards the study.

The task applications are integrated as web or custom views into the prototype

All of the task applications except for the maps application were implemented as web applications to ensure a consistent state across all accessing devices. The maps application was realized with a custom implementation to give it a native feeling and improve support for linking maps in

[5] `GoogleMaps`

multiple devices with each other. All task applications are represented by a URL. The task application is rendered inside the Tangible Windows prototype by opening its URL in a web view or by showing the built-in maps application at the coordinates encoded in the URL.

Multi-device operations are mediated by a server

All multi-device operations are coordinated through a server that also hosts the web applications. Each device maintains a permanent connection to the server and all communication between the devices is done through the server. In particular, the server is kept up to date about the state of each device, including the current URL and whether the target button was pressed. When the user initiates a multi-device operation, the device sends a request containing the active URL to the server, which then forwards the request to all currently targeted devices. If the operation establishes a link between two devices, the server remembers this link and keeps either device informed about changes to the state of the other device.

Evaluation

The Tangible Windows prototype was evaluated by measuring the Task Load Index of a planning task

The prototype was evaluated with 14 participants (2 female, 12 male, mostly students with an average age of 24.5 years) in a user study. Each participant was given 6 Apple iPads with the prototype pre-installed and an introduction explaining the multi-device operations and the available task applications. The task was to identify interesting sights to visit in an unknown city (Sidney or Buenos Aires) by means of a Wikipedia article, and to plan a route around these sights in the maps application. All users planned routes for both cities, one on the prototype and one on a reference system consisting of a desktop computer, with the order of cities and systems randomized. At the end of the test, each participant was asked to fill out a questionnaire and rate the system performance according to the Task Load Index.

During the planning task all participants were interrupted

All participants were informed before the test that they should respond to any emails they might receive during the test. This interruption was scheduled for all participants some time during the planning task on each system. In the email the user was instructed to determine the postal code of a distant city and to find a free time for an appointment. The first task is easily solved with the Google Search application or web site, and the second task was solved with

the calendar application. After the interruption, the participants continued with the planning task.

Participants were able to employ the Tangible Windows operations easily, efficiently, and comfortably

For the planning task, the participants rated the ease-of-use, efficiency, and comfort slightly higher for the Tangible Windows prototype than for the reference setup. However, this difference is not significant between the test conditions. Still, the comparable performance of the Tangible Windows prototype with the desktop computer demonstrates the high potential of the Tangible Windows concept: All participants were very familiar with desktop computers and use them regularly to address various tasks. The prototype system, on the other hand, was completely new to all participants and still allowed them to match the performance of the desktop computer with only minimal training. The high rating of the usefulness of having Tangible Windows available as extra information screens and the good usability of the Tangible Windows operations suggests that the concept can have great value for its users.

The interruption was handled with ease on both systems

All participants were easily able to cope with the interruption on both systems. On the desktop computer, most participants opened a new tab in the browser to address the interruption and closed the tab afterwards to resume the planning task. On the Tangible Windows prototype, participants addressed the interruption by taking a new device or reusing the device that appears least important for their current task. After the interruption, the participants resumed their original task by switching their attention back to the previously used Tangible Windows. One participant explicitly commended the Tangible Windows system for helping the resumption of the previous task because of the spatial arrangement of devices. In summary, the strategies for addressing an interruption and resuming work after the interruption were very similar on both systems with the added benefit that tablets can be arranged spatially which appears to help task resumption.

The measured Task Load Index was smaller for the Tangible Windows prototype than the reference setup

The Task Load Index (TLX) is a multidimensional assessment tool developed by Hart and Staveland [1988] that measures the user-perceived workload of a task. It consists of the weighted average of the following six dimensions: mental demand, physical demand, temporal demand, performance, effort, and frustration. For the Tangible Windows prototype, the average TLX across all participants was with 4.81 slightly lower than the TLX for the reference system with 5.43. On average, participants rated the tem-

poral demand, performance, and effort as the most important dimensions and physical demand as the least important dimension. Interestingly, frustration was rated second highest in the reference condition and second lowest in the test condition. At the same time, with the exception of a single outlier the average score in frustration was significantly higher in the reference condition than the test condition ($\mu_{ref} = 8.00$, $\mu_{test} = 4.46$, $p < .05$). Thus, it appears that the Tangible Windows prototype was able to meet the high efficiency of the well-known desktop system, while creating less cause for frustration, reflected in the better rating and the lower awareness.

The questionnaire yielded three significant findings: With Tangible Windows it is easier to replicate a window, notice the interruption, and find open applications

In the questionnaire all participants were asked to rate the overall ease-of-use, efficiency, and comfort of using either system, and additional questions regarding the performed task and interruption. In particular, the questionnaire inquired the ease of handling multiple windows and how well the system assisted them in switching between the task and the interruption. Most of the ratings were high for both systems (≥ 4 on a 7-point Lickert scale) with a slight preference towards the Tangible Windows prototype. A Wilcoxon Signed-rank test, however, reveals only three significant differences between the ratings of both systems: First, in the Tangible Windows system it was easier to configure a new window to show the same content as another window ($M_{ref} = 4$, $M_{test} = 7$, $p < .05$), which was directly supported by the Copy and Retrieve operations and not supported on the reference machine. Second, in the Tangible Windows system, users found it easier to notice the interruption ($M_{ref} = 5$, $M_{test} = 7$, $p < .05$), which was much more invasive than on the reference system. Finally, users found it easier to find the window containing a specific application on the Tangible Windows system ($M_{ref} = 5$, $M_{test} = 6$, $p < .05$). While the first two results are unsurprising, the last result strengthens the claim that the spatial arrangement of tablets is beneficial for task resumption.

The study of Tangible Windows has shown that an interaction concept based on application state can encourage users to pursue creative and effective new ways of operating multiple portable devices

Most of the participants arranged multiple Tangible Windows devices spatially to reflect a logical meaning. For example, devices placed next to each other showed related contents like a map and a description of a sight as seen in Figure 4.7. One user even stacked devices on top of each other to memorize the order in which he wanted to visit the sights. Thus, the participants made use of the tangibility and the spatiality of the available device to address

Figure 4.7: An example arrangement of multiple Tangible Windows from the user study. Each group of three devices represents two identified sights: two devices show the Wikipedia articles about the sights, and the third device shows a map that can be switched between the sights via the back and forward operations.

their tasks. In this context the multi-device operations provided by the prototype worked very well, as reflected in the high ratings of the questionnaire. In conclusion, the study of Tangible Windows prototype has shown that an interaction concept based on application state can be introduced into portable devices and by doing so reveals new and compelling ways of interacting with portable devices.

4.5.3 SketchIt

SketchIt is a sketching application that facilitates group brainstorming through a combination of Tangible Windows and a shared display

Fraikin [2011] developed the sketching application *SketchIt* with support for Tangible Windows interactions under the supervision of the author. SketchIt demonstrates how the Tangible Windows interaction concept can be applied to facilitate collaborative activities by example of brainstorming. During a brainstorming session with SketchIt, each participant uses a personal device to create sketches, which are shared by transferring them using the Tangible Windows multi-device operations to the shared display.

System Design

The mobile SketchIt prototype visualizes the active sketch on the main part of the screen, which is manipulated via direct touch

The SketchIt application was developed for the Apple iPad and a Mac OS X computer connected to a large touchscreen. Figure 4.8 shows a screen shot of the mobile SketchIt version. The main area of the screen is used to show and edit the sketch. If the brush tool is selected, the user can draw with the selected color on the sketch canvas

Figure 4.8: The mobile SketchIt prototype running on the Apple iPad. The main area of the screen is used to draw sketches via direct touch. From the bar at the bottom, the user can manage their sketches, configure the drawing brush, activate the annotation tool, and initiate multi-device operations.

by dragging the finger on the screen. If the annotation tool is selected, tapping the screen creates a text annotation that is edited using the on-screen keyboard. The appropriate tool is selected from the bottom bar, which is also used to manage all user sketches, configure the brush settings, and initiate multi-device interactions.

The multi-device operations in SketchIt include the Copy and Target operations of the Tangible Windows system

The multi-device operations "Send" and "Receiving" work similar to the "Copy" and "Target" operations of the Tangible Windows system described above. Tapping the "Receiving" button toggles the receiving status of the device, marking it as a target for multi-device operations of all other devices. Tapping the "Send" button initiates a copy of the active sketch to all devices currently marked as target. The intended mode of operation for a brainstorming scenario is to activate the "Receiving" function on the shared display and send sketches from the participant's devices to

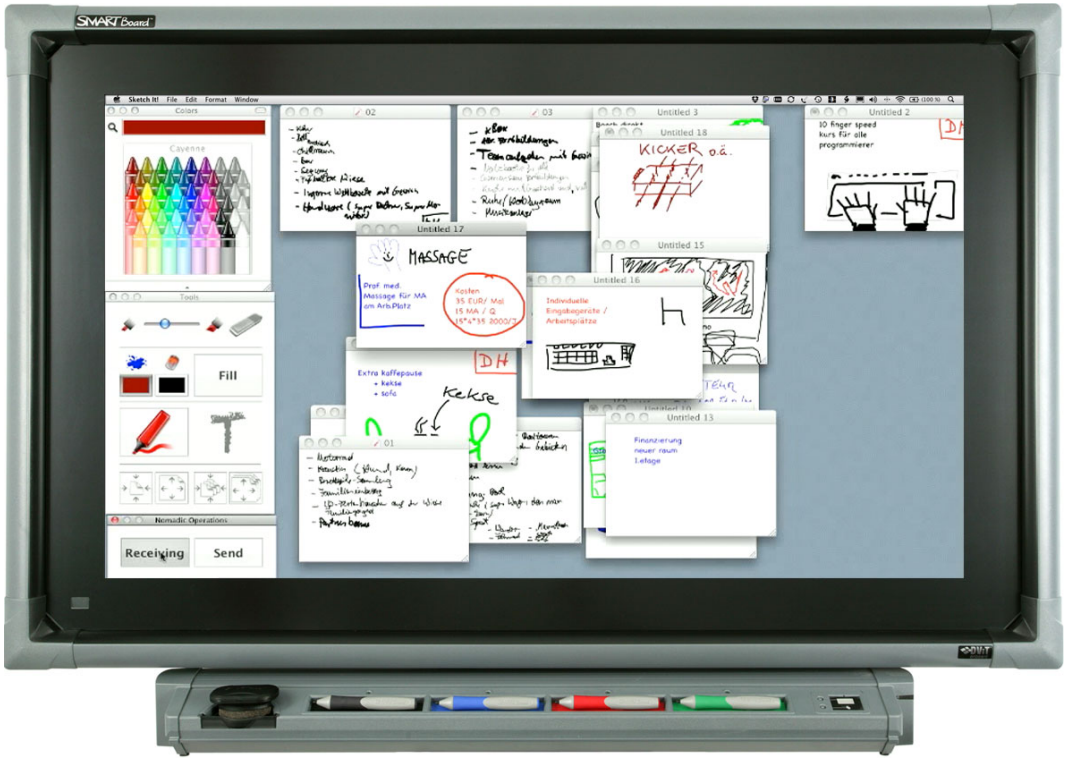

Figure 4.9: The SketchIt prototype running on a large shared display. Sketches are shown in virtual windows, which can be arranged freely on the screen. Users can edit the sketches and annotations via the touch-screen and configure the drawing brush from a special window at the side of the screen. The two buttons at the bottom of the screen trigger the multi-device operations "Receiving" and "Send".

the shared display via the "Send" function. However, it is also possible to mark any of the mobile devices as targets and thus distribute sketches to more than one target. This way, it is also possible to transfer a sketch from the shared display to a portable device.

SketchIt on the large shared display shows multiple sketches in virtual windows that ca be arranged on the screen

On the large display sketches are opened and visualized in virtual windows. Figure 4.9 shows a screen shot of SketchIt running on a shared screen. Each sketch that is sent to the large display is opened in a new virtual window. The windows can be moved freely by dragging the title bar and resized, which resizes the sketch contained in the window. Similar to the mobile version, users can edit the sketches by drawing on them via the touch-screen and annotate sketches with the annotation tool. Users switch between these tools and configure the drawing brush from a special window at the side of the screen. A small window at the bottom of the screen shows two buttons to toggle the "Receiving" function and initiate a "Send" from the shared dis-

play. Finally, sketches can be exported as PDF documents to allow users to archive and reuse sketches created during the brainstorming session.

Evaluation

SketchIt was evaluated in a brainstorming session with seven employees of a private company

The SketchIt system was evaluated with 7 employees at a small software company. They used the system to facilitate a brainstorming session and gather ideas how the motivation and productivity of the employees can be increased with an imaginary budget. The study was conducted in a meeting room at the company's site using 6 tablet computers and a touch-sensitive wall-screen.

The brainstorming session was observed silently

At the beginning of the study, the SketchIt application and its multi-device interactions were explained and demonstrated. Then, the participants were silently observed during the actual brainstorming session. Six of the seven participants used the tablets to draw sketches of their ideas. The final participant acted as a moderator and facilitated the process of sharing ideas via the wall-screen. All participants presented their ideas at some point by transferring the idea sketch to the wall-screen and explaining the idea. Sometimes during this idea presentation other participants had sudden insights and generated new ideas on their tablets, which were shared and discussed immediately after the presented idea.

Participants were satisfied with the brainstorming session and the system support provided by SketchIt

After the study, all participants were asked to fill out a questionnaire to rate their experience with the system on a Lickert scale from 1 (worst) to 5 (best). Overall, all participants were satisfied with the structure of the brainstorming session ($M = 4$), the quantity ($M = 4$) and quality ($M = 3$) of their contribution, and their own overall brainstorming performance ($M = 4$). In addition, participants rated the integration of the multi-device operations "Receiving" and "Send" as very useful for the brainstorming activity ($M = 5$). In summary, the participants received the SketchIt application and its brainstorming capabilities via the offered multi-device interactions very well.

The study of SketchIt showed that multi-device interaction via application state has potential for collaborative activities

The participants showed interesting new behavior during the user study that was caused by the introduction of the prototype system. One user marked his device as a target for all multi-device operations early on to receive all of the sketches that were passed around and keep an archive

of them on his personal device. Another user held up his tablet at one point to present his idea to the group instead of transferring it to the large display. Several users commended the system for its ability to efficiently edit sketches after they had been transferred to refine or leapfrog of ideas. In addition, the participants suggested many other application domains for the SketchIt application, especially in training and teaching situations. The diversity in which users did or imagined they could use SketchIt as a Tangible Window in a collaborative scenario illustrates the versatility of the underlying multi-device operations based on application state. In conclusion, having multiple devices and being able to inter-operate between these devices using application state can pose a great benefit for collaborative activities, such as the demonstrated brainstorming session.

4.5.4 Nomadic Whiteboard

The Nomadic Whiteboard system allows users to exchange virtual windows between a desktop computer and a large display

In parallel to SketchIt and also under the supervision of the author Busch [2011] developed the *Nomadic Whiteboard* system. This system allows users to exchange virtual windows between a regular desktop computer and a large display. It is related to Tangible Windows in that the "Copy" and "Receive" operations are used but not to transfer content between portable devices but to transfer virtual windows between two devices, which both run a window-based operating systems. The goals of the Nomadic Whiteboard and its study were to show that the multi-device operations introduced for Tangible Windows can be integrated into regular operating systems and that they can be beneficial in a single-user work scenario.

System Design

The desktop version of the Nomadic Whiteboard allows users to exchange application windows from the desktop computer with remote devices including large displays

The Nomadic Whiteboard system was developed to run on a regular Mac OS X desktop computer and on a large touch-sensitive display. The desktop version resides in the menu bar of the operating system, where it can be accessed from any running application. Through the menu, users can access the multi-device operations needed to transfer a window to or from the device. The multi-device operations always apply to the front-most window, i.e., the application window that is currently in use. Other than the new menu, there are no changes to the desktop operating system.

Figure 4.10: The Nomadic Whiteboard prototype running on a large touch-screen. The applications on the screen are organized in virtual windows with a special canvas that allows the user to trigger a window transfer directly from the window's frame. In addition, the multi-device operations can be triggered from a special window at the bottom of the screen.

The Nomadic Whiteboard prototype on the large display shows the multi-device operations in a designated window and augments the standard window borders with an option to trigger the "Send" operation on that window

On a large display, the Nomadic Whiteboard prototype shows the multi-device interactions in a designated window at the bottom edge of the screen. Showing the operations directly on the screen is preferable to a menu in this setting because operating a menu via direct touch is often inconvenient. Additionally, the prototype replaces the standard window borders of supported applications with a special border that gives users direct access to the "Send" operation for the window. Figure 4.10 shows a large display running the Nomadic Whiteboard prototype with two open windows.

The Nomadic Whiteboard allows users to transfer a window to another device via the Copy and Retrieve commands or by selecting the target from all discovered devices

The Nomadic Whiteboard supports two different modes of exchanging windows between devices: Just like SketchIt users can transfer a window by marking a device as a target and initiating a send from the sending device. Additionally, users can directly send a window to a remote device by selecting the device from a list of discovered devices, i.e., devices on the same network. These different ways of

initiating multi-device interactions were included to support two different ways of using the system: The first way of transferring windows is preferred for exchanging windows between devices that are in reach in a very natural way. The second way was added to support sending windows to remote devices that cannot be easily reached.

The desktop prototypes supports several legacy applications through the *AppleScript* scripting interface. AppleScript is a technology included in Mac OS X, which allows third-party applications to interface with running applications and extract useful information such as open documents and window properties. The Nomadic Whiteboard prototypes makes use of this interface to identify the document that is shown in the front-most window and the position and size of the window. This information, including a copy of the document, is then transferred to the target device, where the document is opened in the same application and the window is configured to match its original appearance. This approach fails if the application does not support AppleScript and it contains only a small part of the state in the form of the window configuration. However, the approach can give the impression of an almost seamless transition between multiple devices in many cases, which suffices for a study of the appropriateness of the technology.

The desktop prototype integrates legacy applications via AppleScript

The Nomadic Whiteboard prototype also includes a special version of the SketchIt application described above, which can be transferred between devices without losing any state. Additionally, a simple text editing application was developed that also supports a seamless transition by preserving the scrolling and cursor position. These custom applications communicate their state through a special AppleScript interface, which is used by the Nomadic Whiteboard to collect the extra state. On the large display, these two applications also include the special frame that allows users to directly send the window from a button inside its header.

The Nomadic Whiteboard system includes a custom version of SketchIt and a simple text editor that both support a seamless transition between devices by providing additional state information

Evaluation

The Nomadic Whiteboard was evaluated in a contextual study with two participants for two weeks each. The goal of the study was to explore if and how the participants would make use of a large display in their daily routine that

The Nomadic Whiteboard was evaluated in a contextual study with two users

is connected to their regular work devices via the multi-device operations provided in the Nomadic Whiteboard system. Both participants were male students between 20 and 30 years of age. Before the study, a 40" display was installed in the office of each participant and configured with the Nomadic Whiteboard prototype. Additionally, each participant received a copy of the desktop prototype to be installed on their personal devices. At the beginning of the study, the usage of the large display and the software prototype was explained to both participants. Other than that, no instructions were given as to how to use the system, nor were any tasks given to be performed during the two-week period. At the end of the testing period, both participants were interviewed about how they used to system.

The first user employed the Nomadic Whiteboard mostly for collaboration in small groups, where he transfered information to the large display and sketches back to his personal device

For the first participant the large display included a touch-screen and was placed on a stand next to his desk. Even though in this position he could not look at the large display from his sitting position, he preferred it at that place because it allowed him to comfortably stand in front of it and work on it collaboratively. During the testing period he mostly used the large display to collaborate in small groups. When a student enters his office, he transfers a window containing the current work topic to the screen and creates sketches about the topic with the SketchIt application collaboratively. After the meeting, he transfers the sketches back to his personal device to archive them. Without the prototype, he had to take pictures of the sketches on a paper whiteboard, which he felt was a nuisance. Additionally, he commended the ability to prepare meetings on his personal device and quickly transfer the windows to the large display when the meeting started. Overall, the participants regarded the Nomadic Whiteboard as valuable system to quickly visualize information for collaborative use and transfer sketches created on the large display back to his personal device.

The second user used the system to temporarily store windows that should remain in sight but out of focus

The second user had the large display set up at the side of his desk such that he could see and reach it from his regular sitting position. He preferred operating the large display with a mouse and a keyboard and thus no touch-screen was installed for him. The participant mostly used the large display as a storage space for windows that he wanted to remain aware of but should not interfere with his current work. The participant sent these windows from his desktop computer to the large display and arranged them there

as reminders of future activities. During a typical day the user had a project plan and several web pages on the large display. The project plan reminded him of the work he still needed to do and the web pages were reminders to read the content at a convenient time. To resume his work on one of these "parked" windows, he transfered the window back to his desktop computer. In summary, the participant used the large display mostly as a passive display to keep information in sight but out of focus.

The Nomadic Whiteboard prototype demonstrates how multi-device interactions based on application state can be integrated into a user's everyday routine

The two very different behaviors that the study participants exhibited demonstrate the flexibility of the multi-device interactions provided by the Nomadic Whiteboard prototype. At the same time, the tight integration of the system into the regular work devices of both participants shows that these multi-device operations can be implemented on top of existing interactive devices and facilitate a seamless exchange of applications between these devices. Thus, the design and study of the Nomadic Whiteboard prototype show that multi-device interaction based on application state are compatible with the user's every routine and everyday devices.

4.5.5 NoteCarrier

NoteCarrier is an interactive classroom system that employs Tangible Windows to allow students to follow and communicate with a presenter during a lecture

NoteCarrier is an interactive system that aims at assisting students in following a lecture. It was developed by Nazir Sheikh [2012] under the supervision of the author. The main goals of NoteCarrier are to support the learning process of students and to enable students to ask questions and give feedback about the presented materials during a lecture. Each student uses a Tangible Window to view and annotate the slides and to communicate with the presenter in a subtle way during the lecture. By means of these Tangible Windows, NoteCarrier transforms the conventionally one-sided communication during a lecture from the presenter to the audience into a bi-directional conversation.

NoteCarrier facilitates bi-directional, anonymized communication between students and lecturer, is compatible with legacy software, and can be setup with little overhead

The system was designed with several key issues in mind: First, students are often hesitant to ask questions during a lecture because they feel embarrassed or they do not want to interrupt the presentation. NoteCarrier addresses this issue by allowing anonymous communication from the students to the lecturer in a non-invasive way. Second, lecturers are reluctant to switch to new presentation software or hardware because all of their materials have already been

created for the known systems. Consequently, NoteCarrier was tightly integrated into existing presentation software, similar to how the Nomadic Whiteboard system is integrated into common operating systems. Finally, it is important that the presenter and the students can quickly and conveniently use the system without any lengthy preparation. To this end, the multi-device operations offered by Tangible Windows where extended to support synchronizing multiple student devices with the presenter device in an opportunistic fashion.

System Design

The NoteCarrier service extends the presentation software with multi-device operations to enable support for Tangible Windows

The NoteCarrier system consists of a service running on the presentation device that extends the presentation software with multi-device support and two mobile applications, one for the students and one for the presenter. The service runs on any Mac OS X computer and interfaces with the presentation software http://apple.com/keynote[6] via AppleScript. Keynote's AppleScript interface provides all the necessary functions to access and remote control a presentation: It can be used to determine the slide set and the current slide of a presentation and can be used to advance the presentation or switch to a specific slide.

Students can follow the presentation or browse it manually on their Tangible Windows and take notes or provide feedback directly on the slides

Using the mobile application the students and the presenter can request a linked copy of the presentation on their mobile device, which is kept in synchronization with the presentation software through the NoteCarrier service. The student version of NoteCarrier allows students to browse the slide set in two ways: In the synchronized mode the mobile device always shows the slide that is currently presented. In the off-line mode the student can browse the slides manually. At any time, students can take notes directly on a slide by drawing on it with their finger. Finally, the mobile client gives students two options to provide feedback to the presenter during a lecture: First, they can mark a slide as unclear, indicating to the lecturer that they would like additional clarification about the presented topic. Second, they can enter a question related to a specific slide. All feedback generated via NoteCarrier is anonymous.

[6] `AppleKeynote`

The presenter uses NoteCarrier as a presentation remote that also keeps the presenter informed about pending questions and clarification requests

The presenter version of NoteCarrier always shows the presentation's current slide on the device and allows the presenter to go forward or backward in the presentation via a simple gesture. In addition, the lecturer is notified with a subtle visual hint about questions or clarification requests raised by the students. The presenter can chose to directly read and respond to these questions or postpone them to the end of the presentation where they are summarized again.

Evaluation

NoteCarrier was evaluated in a lecture held by a university professor in front of 22 students

NoteCarrier was evaluated in an actual computer science lecture with a university professor and 22 students. The lecturer installed the NoteCarrier service on his laptop computer and received an Apple iPhone running the lecturer version of the mobile NoteCarrier application. Two iPhones and three iPads were given out to the students, all of which running the student version of the NoteCarrier application. Since not enough devices were available for everyone, the students were asked to share the devices during the lecture. After the lecture, the lecturer and the students were interviewed individually about their experience with the prototype.

The lecturer was able to exploit the new technology of the prototype immediately in his presentation

The lecturer commended and made use of all of the features provided by the NoteCarrier system. He appreciated having a mobile device to control the presentation and review feedback from the audience. Especially the clarification requests from students that appeared on his device were a new experience for him, which he was able to successfully integrate into his presentation style. Overall, he was very positive about the system and the provided functions.

Most of the students used the prototype to follow the slides and indicate unclear topics to the presenter

Each student used the mobile devices to follow the presentation when it was their turn. Some students were so focused on the device that they rarely made eye contact with the presenter, which is arguably not a desirable outcome for the presenter. At the same time, it demonstrates the students' engagement in the technology. Many of the students marked one or more slides as unclear and appreciated that their signal to the presenter was received and the topic was clarified. Other students did not see any value in this feature as they preferred asking the presenter in person. Overall, the majority of students found the systems useful.

NoteCarrier demonstrates how multi-device interactions based on application state can be integrated into legacy systems and augment these with useful new functionality

NoteCarrier demonstrates how the concept of Tangible Windows and thus an interaction technique based on application state can be applied to aid the communication between students and lecturer. The evaluation shows that the lecturer is able to quickly adapt his style of presentation to the new technology because of the tight integration of the system into his typical work environment. Likewise, the students were able to immediate draw a benefit from the system Thus, NoteCarrier highlights the ability of multi-device interactions based on application state to integrate into legacy systems and augment them with new and useful functionality.

Chapter 5

The State Exchange Architecture

Today's systems lack support for multi-device interaction

Today's operating systems have very limited technical support for developing multi-device interaction techniques based on application state. As discussed in chapter 2 the user content that is authored and edited in running applications can be extracted and shared via the file. However, the file is not sufficient to represent the task itself because it typically does not preserve the interaction state that is embodied by the application. Many operating systems also support a standard programming interface, such as AppleScript, for extracting information about applications at the programming level. However, this mechanism was not designed to enable application state and consequently does not provide all the information needed to ensure a seamless transition of the application between devices.

This thesis proposes that the operating system should provide the means to extract and restore application state, which is then used by third-party interaction techniques to implement different ways of migrating and distribution applications

To address this issue, this thesis proposes an extension to common operating systems that provides a well-defined way to extract and restore the state of applications. The extracted state can be stored in a container, which inherits the properties of the file. In particular, it serves as a first-class interactive objects that resembles that state of a task. Through this state object users can capture, store, transfer, and resume a task. These multi-device interactions are enabled through special interaction techniques that are designed based on application state. The operating system provides the mechanisms to capture and restore as well as a persistent container to hold the application state, while the interaction techniques provide the interactions needed to migrate or distribute the state between different systems.

Chapter outline

This chapter describes the design process that lead to the state exchange architecture, which illustrates how support for application state can be implemented on top of a common operating system. First, the requirements for such

a system are derived from the challenges of supporting multi-device interaction in the wild and the analysis of the application state conceptual model. Then, a first version of the state exchange system architecture that addresses most of these challenges with a strong focus on the mobile operating system Android is described. Finally, the second and final version of the state exchange architecture is described, which fulfills all of the given requirements in a way that can be implemented on top of most common operating systems.

5.1 Requirements

Application state is a conceptual model that addresses the challenges of multi-device interaction in the wild through several unique properties

As discussed in chapter 4, the conceptual model of application state addresses the challenges of multi-device interaction in the wild by providing a first-class interactive object to represent tasks, which can be migrated between and distributed across multiple devices. The key properties of application state that enable multi-device interaction in the wild are: First, application state enables the seamless combination of multiple application towards a common task by allowing users to switch between applications while preserving state. Second, application state separates managing tasks from actually working on tasks. Third, tasks can be migrated and distributed by migrating and distributing application state.

The requirements in this section guide the design of system solutions

The following set of system requirements summarize the capabilities a system must include to enable these properties of application state:

R1 *State Extraction*: It must be possible to extract the state of a running application into a *state object*. The state object contains all the information needed to seamlessly resume the task that is currently being executed in the application at a later point.

R2 *State Persistence*: State objects can be stored persistently. It is possible to create a copy of a stored state object, and stored state objects can be transferred over the network using common transfer protocols.

R3 *State Restoration*: State objects can be restored in an application. Restoring a state object configures the application in a way such that the user can seamlessly resume the task that was executed when the

state object was extracted. The restored application must be able to run autonomously, i.e., without needing to communicate with the application where the state was extracted, after state restoration.

R4 *State Composition*: In the case that multiple tasks are performed with an application, it is possible to apply above state operations only to a one specific task. This can be realized in two ways: Either it is possible to only extract the part of the application's state that is related to the specified task or it is possible to extract only one specific task from a state object that contains multiple tasks.

R5 *State Sharing*: State objects can be shared between different applications. Multiple applications can be designed to work with the same state such that users can extract the state in one application and restore it in another application while preserving the overall task state.

R6 *Platform Interoperability*: State objects can be transferred between diverse computing platforms. The state objects preserve their ability to restore a compatible application at the stored task state independent of the computing platform.

R7 *State Synchronization*: Multiple copies of state objects can be kept in synchronization including state objects that are distributed across multiple devices connected by a network connection. Synchronized state objects maintain a consistent state by forwarding each change as it occurs to all replicated state objects. Conflicts must be resolved automatically.

R8 *Separation of Control*: Operations on the state of an application including state extraction, restoration, and synchronization can be executed from third-party applications.

R1 - R4 enables the sequential use of multiple devices for a common task

R1, R2, R3, and R4 ensure that it is possible to employ multiple devices for a common task sequentially: The state of the task is captured by extracting state objects of all applications participating in the task (R1). These state objects are then transferred to the target device (R2) and restored (R3), allowing the user to continue operating the application on the new device. R4 ensures that the system can distinguish between multiple tasks that are executed in a single appli-

cation, such that the user can select the task that should be transferred.

R5 and R6 ensure that multi-device interaction can occur between diverse applications and devices

R5 and R6 guarantee that users can switch between diverse applications running on diverse devices without losing task state. R5 describes that it is possible to design applications that share a common state, which can be used to seamlessly switch between these applications using the capabilities described by R1 - R3. In addition, R6 describes that state objects can be transferred between different computing platforms, allowing the development of systems for each platform that can exchange state objects and thus perform cross-platform state exchange.

R7 enables the simultaneous use of multiple devices for a common task

R7 ensures that there is a mechanism to keep multiple state objects in synchronization across different devices. Each application is informed when and how a connected application changes its state. This information is sufficient to keep the appearance of multiple applications in synchronization. In addition, this communication channel can be used to control certain aspects of an application from a remote device. The synchronization and remote control via a synchronized state are demonstrated at the end of this chapter.

R8 ensures that task management applications can be developed separately from the actual task applications

Finally, R8 ensures that all of the multi-device interactions can be executed from external applications. This separation enables the separate development of task management applications from task applications. Task applications can be extended with new functionality and new state capabilities independent of the tools used to migrate and distribute them. At the same time, new task management tools can be developed that represent new ways of interacting with multiple devices independent of the task applications. In consequence, the separation of control enables the development of a diversity of task and task management applications, from which the user can always chose the most appropriate combination depending on the situation.

5.2 First Iteration of the State Exchange Architecture

Under the supervision of the author, Plücken [2012] developed a system that enables support for multi-device interaction between a mobile and a desktop operating system. He describes the main goals of his system in four scenarios:

1. When working on a computer, it is not always possible to finish all the work in time, before one has to leave. The system proposed in this thesis should allow users to continue the left-over work when they are in transit on a mobile device. Therefore, the solution should be able to transfer one or more documents and their editing state from a desktop computer to a mobile device.

   Allow users to continue work that was started on a fixed computer on a mobile computer

2. In meetings whiteboards are often used to create and capture notes and sketches. The system should allow meeting participants to transfer these sketches to their personal devices, edit them, and later transfer them back to the whiteboard to share their contributions with the group.

   Facilitate combined use of whiteboards and tablet computers in a collaborative setting

3. Many users enjoy listening to music on their smartphones when they are in transit. When they arrive at home, however, most users prefer to employ their home stereo. The proposed system should provide a seamless transition of music playback between these two devices such that users can switch between the devices without interrupting their listening experience.

   Enable seamless switching of music players

4. When reading a book, the reader wants to share an interesting passage with a friend. The system should allow the reader to transfer the book at its current page to a device that belongs to another person.

   Enable seamless switching of devices while reading

All of these scenarios have in common that users want to switch between different devices while preserving the state of their interaction across this switch. Thus, a solution based on application state should be a good fit to enable these scenarios. This section describes the design and implementation of the system proposed by Plücken [2012] and analyzes its effectiveness to support multi-device interaction in the wild with respect to the requirements described above.

5.2.1 System Design

The system consists of a service that runs in the background of a device and allows applications to initiate and respond to state transfers. To initiate a state transfer between multiple devices, a state object is created and transferred to

The system employs a background service that applications can use to initiate and respond to state transfers

the target device, where it can be restored in a compatible application. The state object is configured from the application by setting all needed attributes via an interprocess communication interface provided by the service. The service then either transfers the constructed state directly to the target device or stores the information in a file, which can be transferred using third-party file sharing applications. When a state object is received, the service determines an appropriate application to restore the state, launches the application, and passes the state object to it. If multiple installed applications fit the state, the user can decide which application to open.

State objects must contain all the information needed to seamlessly continue the captured task

State objects include all information that is needed to seamlessly resume operation of the application at the state of extraction. Consequently, state objects must include all visible settings, i.e., the configuration of all widgets that are currently shown on the screen, and any additional custom information that is needed to restore operation at the given state. State objects should not include any information that is implicitly defined in the application, such as the layout of the user interface or bundled resources like icons.

State objects are stored in trees, where each branch is identified by a named key

State objects are organized as trees with each branch identified by a named key. This allows values to be logically grouped by placing them in the same subtree. At the same time, each value can be unambiguously referenced by a path, which is an ordered list of all keys that are traversed to reach the value. The tree structure was chosen because it is very flexible and there are several standardized human-readable formats available for this type of structure.

State trees are organized in five levels: context, group, object, attribute, value

Plücken [2012] suggests the following standard scheme for state trees: At the top level, each state object contains one or more context branches. Each included context defines the type of application state that is stored in the branch. At the next level, the application can use any number of groups to structure the different aspects of the application state into semantically coherent groups. The next level describes the name of the object that is stored. Each object is then stored using one or more attributes that have a name and a value. Figure 5.1 shows an example state tree of a text editor.

A context can define a standard format for the subtree, which serves state sharing

Each application can use one or more context keys to store its state in one or more ways. These context keys define the standard structure of the subtree and its groups, objects, and attributes. Each context represents a specific class

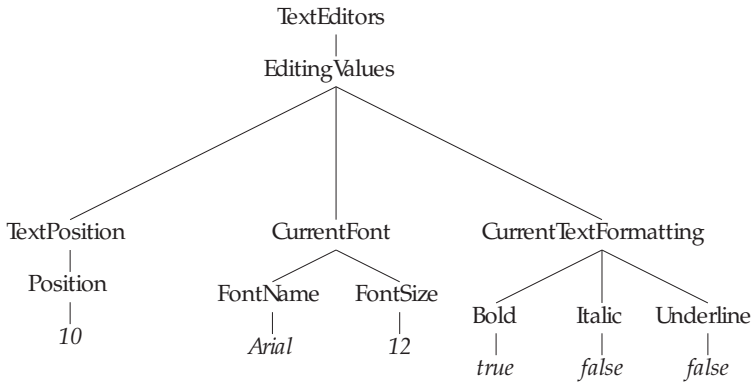

Figure 5.1: Example state of a text editor: In the context "TextEditors", there is only one group named "EditingValues", containing three attributes: "TextPosition", "CurrentFont", and "CurrentTextFormatting". Picture taken from Plücken [2012].

of applications that share a common functionality. For example, "TextEditors" can be a standard context for applications that include text editing functionality. It defines the group "EditingValues" with the objects "TextPosition", "CurentFont", and "CurrentTextFormatting", which are all common aspects of text editors. If multiple text editors supply their state in this context, they can interpret the state of each other, allowing users to seamlessly switch between different applications using state objects. At the same time, the context does not limit the subtree structure to the defined values. Text editors with extended capabilities can add their own groups and objects to the existing context, as long as they do not interfere with the default structure. These custom extensions are simply ignored by other applications upon restoration.

Documents are included in the state in the form of references that can be resolved as needed

Many tasks include authoring and editing documents or consuming media. These documents must be included in a transition or otherwise the task cannot be resumed. At the same time these documents can be large in size and stored on diverse storage drives, including cloud services. To prevent slow and redundant transfer of these documents when transferring state, the state object includes references to documents instead of copies of the documents. A document reference consists of the document's system path, type, and a checksum, which can be used to verify the integrity of the document after a transfer.

State objects are transferred as a bundle including all documents or separately

State objects can be transferred in two different ways: If the communication channel between the devices supports only asynchronous communication, such as email, the state object and all referenced documents are stored and into an archive, which is then transferred. If the communication channel support synchronous communication, such as a

socket network connection, only the state object is transferred initially and the referenced documents are sent only on request.

5.2.2 Implementation

The implementation consists of two services (Android and Windows/Linux) that share a common state object management library

Two services were implemented in the programming language Java to demonstrate above design. The first service was implemented as a background service on the Android operating system. The second service was implemented as a platform-independent application, which can be executed on Microsoft Windows, Linux, and Mac OS X machines. Both services share common functionality in the form of a library for creating and managing state objects and a second library for communicating between different devices.

State Management Library

The state management library contains the necessary functions to create, manipulate, and export state objects. The provided functions are used by applications via interprocess communication to extract their state into state objects and later restore their state from a state object. In particular, the library provides the following functionality:

Create state object and retrieve its ID

- A state object is created and assigned a state ID, which can later be used to manipulate the state object.

Assign state attributes via the state object ID, the path, and the attribute value

- The attributes of a previously created state can be set by providing the state ID, the path to the attributes, and its value. Intermediate branches (context, group, object) are created automatically. Acceptable value types are integers, booleans, strings, and files.

Read state attributes via the state object ID and the path

- State attributes can also be read by providing the state ID and the path to the attribute.

Create a state file that contains a finished state object

- A finished state object can be stored into a state file. This file can be transferred via the networked or accessed directly to transfer it using third-party applications.

Create a state archive that contain the state file and all referenced documents

- A state archive can be created that contains the state file and all referenced documents. The state archive is used to store a persistent copy or to transfer state asynchronously between devices.

State attributes are stored in the file system

While state objects are constructed, their values are stored in the file system following the branch structure of the state: An attribute value is stored in a file located at */context/group/value/attribute.attrib* inside a unique folder for each state ID. The state file is generated by traversing the file structure for the state ID and reading all attribute values from the attribute files. File attributes are stored in special files called *attribute.file*, which contain the referenced document. When generating a the state file, for each file attribute a reference to the stored document is created instead of including the document data.

State archives are tar archives with the state file at the top of the archive followed by the referenced documents

State archives are creating using the Java Gnutar[1] library. The created tar archive contains the state file at the head of the archive, followed by the referenced documents. This way, the receiver can extract the state file first and decide based on the contained information whether the remaining files need to be extracted from the archive.

Communication Library

The communication library enables state files and referenced documents to be transferred between devices

The communication library provides the functions needed to transfer state objects and referenced documents to remote devices. It was designed to support synchronous, bi-directional communication between two devices using a variety of transport technologies, including wireless networking and Bluetooth. In addition, the communication library automatically discovers all available devices that can be the target of a state transfer.

The communication library supports various communication channels to discover peers and exchange data

The library supports discovering peers and establishing a network connection via TCP networking and Bluetooth. The TCP networking support uses multicast DNS[2] to discover other services on the network. For data transfer, standard network sockets are used which allow devices to exchange data synchronously over a network link. The Bluetooth support uses the built-in device discovery mechanism and transfer protocols to enable discover and data transfer. The library can be extended to support additional communication channels, such as near-field communication.

The best communication channel is automatically selected for a state transfer

The communication library automatically selects the best available communication channel to perform state trans-

[1] `http://code.google.com/p/javagnutar`

[2] `http://multicastdns.org`

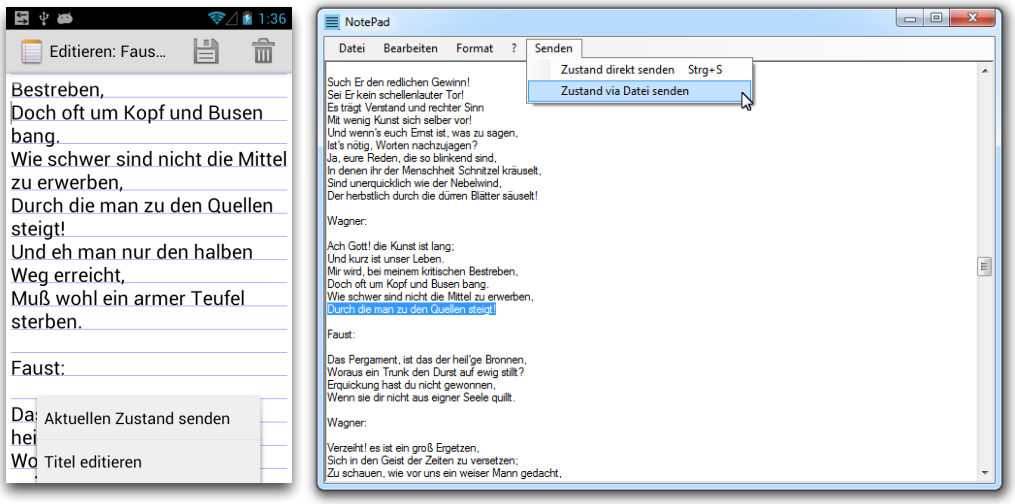

Figure 5.2: Example text editor on Android (left) and Microsoft Windows (right). On Android the main screen is used to view the text, which can be edited via an on-screen keyboard. On Windows the text is shown in a window and can be edited via the keyboard. The state transfer operations are accessed via the context menu (Android) or the window menu (Windows). Pictures taken from Plücken [2012].

fers. Upon discovery, each discovered service is queried for a unique ID that is used to associative the available communication channels with a single service. From then on, each service can decide based on the priority setting of the discovered communication channels, which channel to use for a state transfer.

State is migrated by first transferring the state file followed by all document files upon request

The library defines a simple communication protocol, which allows each service to send a file to the other end or a request a specific file. To migrate a state from one device to another, the sending device simply transfers the state file to the receiving device. The receiving device then analyzes the state file and sends requests for each referenced document file that it is missing. These requests are answered by another file transfer containing the requested file.

5.2.3 Example Applications

Two example applications have been developed on each platform to demonstrate how to implement support for state extraction and restoration in a task application. The example applications are a text editor and a PDF viewer. These task applications are also responsible for initiating a state transfer.

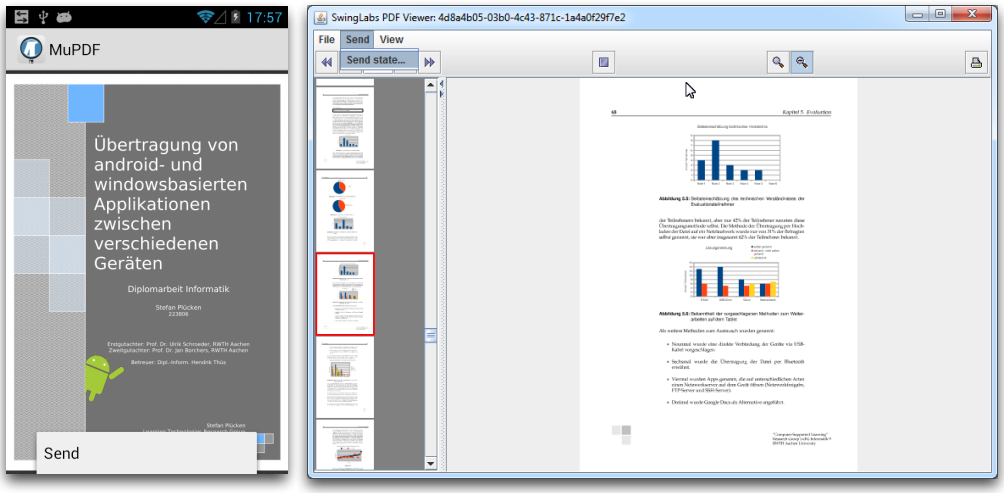

Figure 5.3: Example PDF viewer on Android (left) and Microsoft Windows (right). On both operating systems existing open source PDF viewers were extended to support extracting, restoring, and initiating state transfer. Pictures taken from Plücken [2012].

A text editor that supports state transfer was developed from scratch on Android and Windows

The text editor allows users to open, view, and edit text files. On Android the text is shown on the main screen of the device, where it can be edited via an on-screen keyboard. On Windows the text is shown in a window and edited with an attached keyboard. Both applications offer the current state of the application to be transferred to another device from a built-in menu. On Android the context menu accessible through a hardware button is used, while on Windows the application menu at the top of the window is used. Figure 5.2 shows a screen shot of both applications.

Existing open source PDF viewers were extended to support state transfer on Windows and Android

The second example application that was developed is a PDF viewer. Both on Windows and on Android an existing open source PDF viewer was extended to support state transfer using the proposed system. On Android the MuPDF viewer[3] was used, and on Windows the SwingLabs PDF Renderer[4] was used. Both applications were augmented with capabilities to extract the current state including the visible page in the PDF document and a mechanism to initiate a state transfer similar to the text editor. Figure 5.3 shows a screen shot of both PDF viewers.

The example applications include three different ways of transferring state: send via file, send to device, and send directly

Three ways of sending state were implemented in above example applications. First, users can store the state in

[3] `http://mupdf.com`
[4] `https://java.net/projects/pdf-renderer`

a file and transfer the file to the target device using any means they deem appropriate. There, the file is restored by opening it directly with the service, which launches the appropriate application and configures it to resume operation from the stored state. Second, users can transfer the state to a target device by selecting the target from a list of discovered devices. The service then extracts and transfers the state to the target device, where it is immediately restored. Finally, users can configure a service as "receiving" and then send a state directly to this service by initiating a "direct send".

5.2.4 Discussion

The system architecture presented in this section fulfills many of the requirements for system support for multi-device interaction based on application state

The presented architecture enables users to migrate tasks between mobile and desktop devices by transferring the state of applications between these devices and resuming operation of the task stored inside the application state (R1-R3). The approach has a strong focus on supporting state sharing between different applications via a sophisticated tree structure that allows the definition of standards for common application classes (R5). Finally, the entire architecture was developed in Java, which ensures platform interoperability (R6).

The architecture does not provide sufficient support for state composition, state synchronization, and separation of control

The other requirements, however, are not fulfilled entirely. The architecture supports multiple documents to be referenced in a single state but it is not clear how the state should be structured to reflect a relationship between open documents and state attributes. Thus state composition (R4) is not explicitly supported in the current version of the architecture. Similarly, there is no explicit mechanism for state synchronization (R7). Two applications running on different devices can be kept in synchronization with some delay by continuously transferring and restoring their states. However, this mechanism is slow and prone to conflicts. Finally, the architecture does not consider separation of control (R8) but instead integrates the interaction technique to trigger multi-device operations into the task applications.

The proposed system successfully demonstrates how multi-device interaction can be integrated into modern operating systems

Despite these shortcomings, the proposed system successfully demonstrates how multi-device interactions can be integrated into current operating systems with only small changes to the task applications. For example, the adaptation needed to make the PDF viewer on Android took only 70 lines of code. Additionally, the exploration of standard

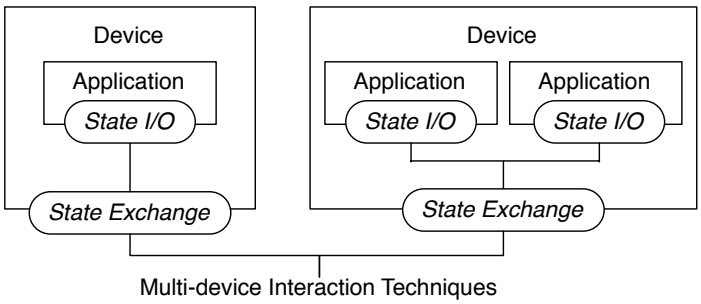

Figure 5.4: The state exchange system architecture: The State I/O programming interface exposes the state of running applications. The State Exchange programming interface exposes the states of all supported applications to network clients implementing multi-device interaction techniques.

state formats and the mechanism to reference and transfer files independent of the application state had a strong impact on the next iteration of the system, presented in the next section.

5.3 Final Iteration of the State Exchange Architecture

The final version of the state exchange architecture was designed and implemented based on the first iteration of the system architecture described in the previous section. Its goal was to fulfill all of the requirements listed in section 5.1. Special focus was placed on those requirements that the previous approach did not fulfill: supporting state composition, state synchronization, and separation of control.

5.3.1 System Design

The state exchange architecture employs a service to manage the communication with other devices and the State I/O programming interface

Similar to the previous design, the state exchange architecture also employs a service called state exchange service to coordinate the communication between different devices. In addition, two programming interfaces are defined for the communication between the service and task applications and the service and other services or multi-device interaction techniques. *State I/O* defines the methods needed for the service to extract, restore, and synchronize the state of a task application. *State Exchange* defines the methods needed to exchange state across devices and initiate multi-device interaction techniques. This architecture is illustrated in Figure 5.4.

Method & Parameters	**Description**
register(application)	Registers an application with the state exchange service. The parameter contains various information about the application, including a unique identifier.
extractState(path, includeDocuments, observe)	Extract and return the current state of the application including the user interface state and open documents. The path parameter describes the location for a partial state extraction. The includeDocuments parameter defines whether document copies should be included in the state. The observe parameter adds an observer to the extracted state.
restoreState(state, path, observe)	Restore the application to the given state. The path parameter describes the location for a partial state restoration. The observe parameter configures the application to emit state change events.
stopObservingState(path)	Remove a state observer from the location described by the given path.
Event	**Description**
stateChanged(path, state)	This event is triggered when the state changes after the caller has registered as a state observer. The path parameter describes the location of the change and the state object describes the actual change.

Table 5.1: The methods of the State I/O programming interface to expose the state of an application.

State I/O Programming Interface

The State I/O programming interface exposes the interaction state of a single running application. This interaction state is comprised of everything that is needed to restore the tasks that are currently executed in the application, including all open documents and the configuration of the user interface. Through the State I/O interface a third-party application can trigger state extraction, restoration, and observation on the task application. Table 5.1 lists the methods defined in the State I/O programming interface.

Applications maintain a persistent connection with the state exchange service

Each task application running on the system that supports state exchange maintains a persistent connection with the state exchange service also running on the device. After connecting, the application registers itself as a task application by providing various information about the application including a unique identifier. The state exchange service maintains a list of all connected applications and initiates the other State I/O operations as needed.

When state extraction is triggered, the application collects the information needed to make the current interaction state of the application persistent. What is included in the state of an application is left at the discretion of the application developer. At a minimum, an application's state should always contain the user content that is currently accessed by the application and the state of the application's user interface. The extracted state is stored in a state object and returned to the caller.

Upon state extraction the application collects and stores the current state in a state object

The state object is structured as a tree with each branch identified by a key, such that individual subtrees or leafs can be described by a path that lists all keys on the way from the root to the subtree. The tree contains three specific keys at the root level to represent different aspects of the state. The remaining structure of the tree is flexible such that applications can add their own keys as needed. The standard root-level elements are:

The state object is represented by a tree with key references to individual branches

- The *application* element contains information about the application that created the state. This information serves the purpose to identify the type of the state, aiding the decision of how to use the state. At a minimum, the application identifier and type identifier of the state must be set. Other useful keys include the path of the application's name and an Internet URL where the application can be obtained.

  Application contains information about the application that was used to extract the state

- The *global* element contains the global state, i.e., the application state that does not depend on any open document. The state is stored as a subtree, which is defined by the application developer. It is recommended to use meaningful key names and organize the state in a semantic way.

  Global element

- The *documents* element contains a list of all open user documents and their associated state. Each document must be referenced with a unique identifier as the key to its subtree and a *document* element that describes the document. A typical document description contains the document name, path, and a modification time stamp or checksum. Other than that, the structure of the state tree associated with the document is up to the application developer just like the global state subtree.

  Documents elements

The state extraction can be configure to include a copy of all referenced documents alongside the state object. If doc-

The service can request copies of the user documents when extracting application state

ument copies are requested, the application must create the copies of the document and make them somehow available to the service. This can be done by returning the document data or a reference to the documents alongside the state object.

It is possible to extract a partial state by providing the path to the subtree that should be extracted

Additionally, the state extraction can be configured to extract a specific subtree identified by a path. If this path is passed to the application, it is the responsibility of the application to create a state object that represents the subtree at the given path. This option was added to allow applications to request only the part of the state that is needed to potentially reduce the transmission overhead. Additionally, this option is used when synchronizing state, which is described below.

When extracting application state, the service can request to stay informed about changes to the part of the state that was extracted

Finally, the state extraction can be configured to add an observer for the extracted state. This observer then informs the service about changes to the state as they arise by sending change events that include the path where the change occurred and an object describing the change. If the state extraction was performed on a part of the state, the observer is only added for that part as well. The path and state objects that are included in the change events are compatible with the state restoration method described below. Previously added observers can be removed later on by providing the path that was used during state extraction.

State objects can be made persistent and transferred over the network

Extracted state objects can be stored persistently such that it is guaranteed that the information stored in the state is not lost. It is also possible to duplicate state objects and transfer them over a network connection. Additionally, state objects can be inspected and manipulated. In particular, it is possible to read the different root-level elements and extract a specific document or other subtree from a state object and store it in a new state object. This way, a state object containing multiple tasks can be divided into individual state objects for each task.

A previously extracted state object can be restored

When an application is requested to restore the state contained in a state object, the application should configure itself to resume operation of the task at the time when the application state was created. After the restoration, the state of the application should reflect the state stored in the state object. A state restoration can also only affect a part of the application state: By providing an optional path the state object is applied to the subtree that is located at the path.

If appropriate, the application state is extended with a new branch where the state object is inserted. This way, a new document window can be opened and restored to the state stored in the state object. Finally, the application can be configured to emit state change events after a restoration. This way, two applications can be synchronized in both directions by forwarding state change events from either end to the other.

State Exchange Programming Interface

The State Exchange programming interface allows third-party clients to access and manipulate the state of the applications running on the device. Clients of the State Exchange interface are typically applications that implement a multi-device interaction technique based on application state. The State Exchange interface includes all of the methods of the State I/O interface with an additional parameter that describes the application that is affected. Additionally, the interface provides methods to launch and terminate running applications and events to keep the remote end informed about which applications are currently running. Table 5.2 lists all of the methods included in the State Exchange interface.

The State Exchange interface provides clients with a list of running applications and methods to start and stop applications

When a client connects to the State Exchange interface, it must first register with a unique identifier. The service then sends a list of running applications, which is updated via events every time a new application launches or a running application is terminated. The client can also directly launch and terminate applications on the device. This way, clients can be kept informed about running applications, allowing them to access their states as needed.

All State I/O methods can be initiated via State Exchange by providing an additional application parameter

State exchange offers all of the methods of the State I/O interface but each method call is extended with an application parameter that identifies the application that is affected by the method. Similarly, all events are replicated by adding the application parameter that identifies the application where the event originated from. The application parameter is compatible with the list of applications and the application launch or terminate events that are transmitted to the client. The service implementing state exchange also ensures that observer events are forwarded to the appropriate clients. To this end, it must remember which client

Method & **Parameters**	**Description**
register(client)	Register a new client with the service. The client parameter describes the client application, which includes at least a unique identifier.
startApplication(application)	Launch the application that is represented by the application parameter.
stopApplication(application)	Terminate the application that is represented by the application parameter.
extractState(application, path, includeDocuments, observe)	Extract and return the current state of the specified application. If no application parameter is given, the top-most application is used. The path, includeDocuments, and observer parameters are used as described in the State I/O interface.
restoreState(application, state, path, observe)	Restore the application to the given state. If no application parameter is given, an appropriate application is determined from the state object If the application is not running, it is launched before restoration. The path and observe parameters are used as described in the State I/O interface.
stopObservingState(application, path)	Remove a state observer for the given path from the specified application.
Event	**Description**
applicationStarted(application)	This event is send when a new application is started.
applicationStopped(application)	This event is send when an application was stopped.
stateChanged(application, path, state)	This event is sent when the state of the given application changes after an observer was added to that application. The application parameter identifies the application, the path describes the location of the change, and the state object represents the updated state.

Table 5.2: The methods and events of the State Exchange programming interface.

added which observer and forward the events from the application to the appropriate clients.

State Exchange Service

The state exchange service mediates between external clients via the State Exchange interface and applications running on the same system as the service via the State I/O interface. All applications that support state exchange connect to the local service and register with their credentials. The service maintains a list of connected applications and also identifies other applications installed on the system, which can be launched if necessary. This list is shared with every client that connects to the service via the State Ex-

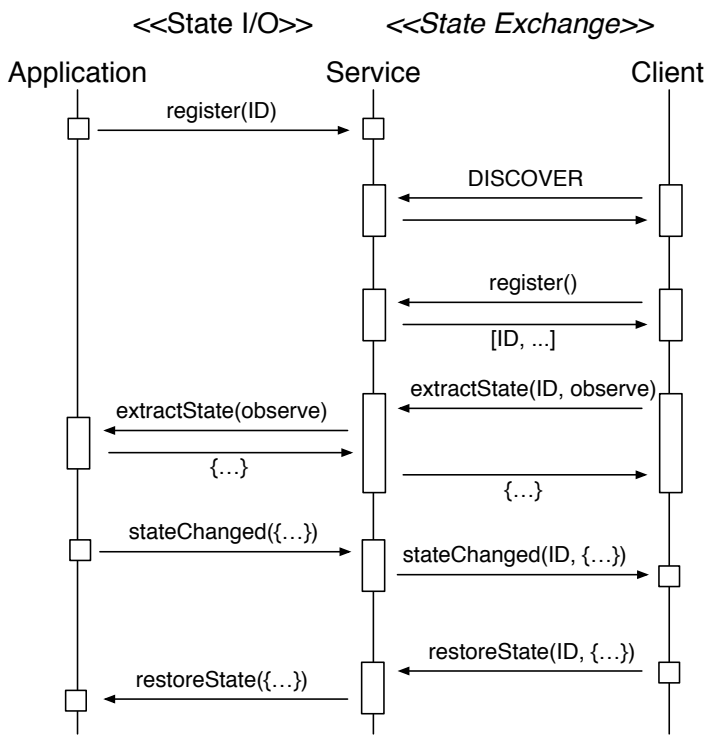

Figure 5.5: Sequence diagram of a typical communication sequence between applications, clients, and the state exchange service. First, the application registers with the service. Then, a client discovers and registers with the service. Afterwards, the client requests that the state of a specific application is extracted, which is forwarded by the service to the application. The application responds with a state object and state changed events, which are forwarded to the client. Finally, the client requests a state to be restored, which is again forwarded to the application.

change interface. All State I/O requests that are initiated by any of the clients are forwarded to the appropriate application and the response, including change events from observers, are sent back to the client. Figure 5.5 illustrates a typically message flow between task applications, clients, and the state exchange service.

5.3.2 Implementation

The state exchange service was implemented as a *Node.js* application. Node.js is a software platform based on Chrome's JavaScript runtime. Distributions of Node.js exist for all major platforms including Windows, Mac OS X, and diverse Linux distributions. It was chosen because of its extensive support for socket networking and asynchronous behavior, which makes it a prime candidate for any service-oriented application.

The state exchange service was implemented in Node.js

In addition, a shared library was developed that supports application developers in implementing the State I/O interface. This library manages the connection to the state exchange service and defines the application state data struc-

Applications are supported in their implementation of State I/O with a shared library

ture. Two versions of the library were implemented: one for Mac OS X and iOS applications and a second for web applications.

State Exchange Service

The service creates two servers that wait for connections from applications and clients

The state exchange service was developed with *Node.js* version 0.8.16 from December 13th, 2012. Upon launching, the state exchange service creates two servers, each hosting a socket and a web socket (using the *ws* package version 0.4.23). The first server accepts connections from local applications, which communicate with the service via the State I/O interface. The second server accepts connections from local and remote clients, which communicate via the State Exchange interface.

Device discovery is done via Bonjour

Before a remote client can connect to the state exchange service, it must discover the service and the device over the network. To this end, the service is advertised using Bonjour[5]. Clients can discover the service if they are running on a device that is connected to the same network. Once discovered, a client can connect to the service and communicate via the state exchange interface.

The state exchange message protocol is based on JSON

Both servers of the state exchange service use a simple protocol to exchange messages via the socket connection with applications or clients. These messages are encoded as UTF-8 strings, and individual commands are terminated by a \0 character (ASCII code 0). The messages themselves are formatted using the JavaScript Object Notation (JSON). There are three different types of messages: requests, responses, and events. A request is used to execute a command. It is always followed by a response that can be associated with the original request via a unique ID number. An event describes an event that occurred and does not expect a response. The following listing shows an example of a request message:

```
1 { "id": 1,
2   "request": "registerApplication",
3   "object": {
4     "identifier": "com.example.app" } }
```

This request is sent by an application after connecting to the service to register the application with the service. As-

[5] http://www.apple.com/support/bonjour

suming that no error occurs, the service will answer with the following response:

```
{ "id": 1,
  "response": "OK" }
```

Using the ID number, the response can be associated with the original request. The following listing shows an example of an event:

```
{ "event": "applicationLaunched",
  "object": {
    "identifier": "com.example.app" } }
```

This event is sent when a new application was launched while a client is connected to the service. The object describes the launched application with a unique identifier.

Both servers also offer a web socket connection to serve web applications and web clients

In addition to the socket, both of the servers included in the state exchange service also offer their services via a *web socket*. This web socket was added to allow web applications and web clients to easily participate in state exchange. A web socket uses the connection of a web request as a bidirectional communication channel by keeping the connection open after the original web request has terminated. It employs a communication protocol that is very similar to the one described above and thus compatible with the message types employed by the state exchange service.

State I/O Support Library

The *State I/O support library* can be loaded by a task application to assist with the implementation of the State I/O interface. It creates and maintains the connection to the state exchange service running on the same machine and provides an abstraction of the State I/O interface to be implemented by the application. Additionally, it defines the state object that can be configured and sent in response to state extraction requests.

The support library maintains the connection to the service and forwards messages to the application in the form of callbacks

The library is initiated with a unique application identifier and an application type. Once initiated the library creates a connection to the local service and uses the identifier to register the application. As long as the connection is established, any requests from the service are forwarded to the application via two callback functions: *extractState* is called

when a request for state extraction was sent. *restoreState* is called when a state to be restored was received from the service. Both callbacks include all of the parameters from the State I/O methods except for the observing flag.

The application can use a special state object to create, configure, and inspect application state

The application can make use of a special state object to create and configure its current state. The state object reflects the tree structure described in the previous section. The *applications* element is automatically configured with the identifier and type used to initialize the application and information gathered from system frameworks. The *global* element can be freely configured by the application. The *documents* element can be filled with one or more states that must contain a document reference. The finished state object is simply returned by the appropriate callback, which transforms the stored state into a JSON object and transmits it to the service. Similarly, a received state is transformed into a state object and passed on to the application for inspection and restoration.

State observation is supported by calling a library method every time the application state changes

To support state observation and thus synchronization, the application must call a method defined by the library every time its state changes with a state object representing the changed state. The support library remembers which parts of the state have been extracted with the observing flag enabled and forwards the matching state change events to the service.

5.3.3 Example Applications

To demonstrate how State I/O can be implemented in modern applications based on above implementation, two open source applications have been augmented with State I/O support: TextEdit and Skim. In addition, custom applications were created that share the state with these applications to demonstrate state sharing and synchronization.

TextEdit

TextEdit is the default text editor included in Mac OS X. It is implemented in Objective-C using the Cocoa framework. Its source code is available at the Apple Developer Library[6]. TextEdit contains $3,324$ source lines of code (ignoring comments) in 14 classes.

[6] `http://developer.apple.com/library/mac/#samplecode/TextEdit`

The state of TextEdit contains all open documents, the window configuration of these documents, and the cursor position and text selection

Text edit allows users to open text files and display their contents in a separate window for each open text file. To edit the text, users navigate the text cursor to the appropriate position in the text and start typing. Users can also select a portion of the text to highlight and quickly manipulate that portion of the text. All of this state information is dependent on the active document. Thus, the complete state of TextEdit is stored under the documents element and the global element is left empty. The state information that is stored for each document is:

- *file*: the document file
- *selectedRange*: the range of the current selection – the cursor position is the starting index of the selection range
- *windowFrame*: the location and size of the window
- *scrollingRect*: the visible portion of the document in the window

TextEdit also allows text to be formatted and stored in the rich text format. For the sake of simplicity this advanced behavior was ignored in the implementation of State I/O. The following listing shows an example of an extracted TextEdit state:

```
{ "application": {
    "identifier": "com.apple.TextEdit",
    "name": "TextEdit",
    "type": "texteditor" },
  "global": {},
  "documents": [{
    "file": {
      "modificationDate": "2013-03-18T15
          :40:32Z",
      "edited": false,
      "path": "/Users/demo/example.txt",
      "type": "public.text" },
    "selectedRange": {
      "location": 11,
      "length": 0 },
    "windowFrame": {
      "y": 880,      "x": 349,
      "width": 565, "height": 468 },
    "scrollingRect": {
      "y": 0,        "x": 0,
      "width": 550, "height": 389 } }] }
```

Integrating State I/O support into TextEdit required only two classes to be modified

After including the State I/O library, only two classes had to be modified to implement State I/O: the *Controller* and the *DocumentWindowController*. The Controller class is the application delegate, which is the central object that is notified about all aspects of the application's life cycle. It was modified to initialize the the State I/O library upon launching the application. The DocumentWindowController class is responsible for the window of a text document. Two methods were added to this class that allow the extraction and restoration of the associated document's state. The Controller classes uses these methods to collect the states of all open documents or create a new document and restore a given state upon request.

State synchronization is enabled by creating state change events based on cursor changes

To support state synchronization, TextEdit has to inform the service about any changes that occur to its state. To this end, the DocumentWindowController class was modified to observe the cursor position in the document's text view and the position and size of the window and the scrolling view. These notifications are sufficient to capture all state changes as any text modification is always accompanied by a change of the cursor location. Based on the change information included in the notification, the DocumentWindowController can generate a state change event that reflects the change of the state.

State change events include information about text modification to improve performance

TextEdit also includes textual changes to the document in the change events. These change descriptions are generated by comparing the previous version of the text with the current version. The change description is included to avoid the high overhead of retransmitting a changed file over and over to synchronize its contents across multiple applications. The change description is included using the virtual key *file.content*. The following listing illustrates a change event where the character "a" was inserted into the text:

```
1 { "event": "stateChanged",
2   "object": {
3     "application": "com.apple.TextEdit",
4     "file": {
5       "content": {
6         "insert": {
7           "location": 11,
8           "text": "a" } } },
9     "selectedRange": {
10      "location": 12,
11      "length": 0 } } }
```

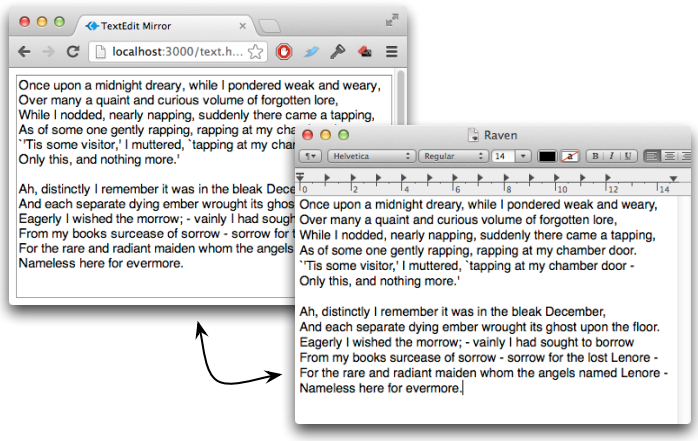

Figure 5.6: State synchronization between TextEdit and WebTextEdit: Both applications generate state change events when their state changes. If these events are forwarded to the other application, the appearance of both applications can be kept in synchronization.

A second text editor called WebTextEdit was implemented as a web application that shares state with TextEdit. WebTextEdit can only show a single open document. The document's contents are displayed in a *textarea* element, which the user can edit similar to the original TextEdit application.

WebTextEdit and TextEdit share the same application state but not all aspects of the state are used by both applications

When a state from TextEdit is restored in WebTextEdit, the first document contained in the state is displayed and can be edited. This document reflects the document that was currently active in TextEdit when the state was extracted. In addition, the cursor position is restored from the state such that the user can seamlessly continue editing the text after the transition. Because the window position and size is handled by the browser in the web application, the state information about the window and scrolling view are ignored in the restoration.

WebTextEdit can be synchronized with TextEdit such that all edit events are reflected in both applications simultaneously

WebTextEdit also supports state synchronization. When the user types a character or changes the cursor position inside the text, WebTextEdit generates a state change event that includes the same information as the TextEdit state change event. Through these events, TextEdit and WebTextEdit can be set up to synchronize their appearance such that their user interfaces appears the same at all times as illustrated in Figure 5.6. To enable state synchronization, the service on both ends must be configured to forward state change events to the other side. Note that this solution ignores potential conflicts, when both ends are manipulated at the same time. However, as discussed earlier there are technical solutions to automatically resolve such conflicts.

Skim

Skim[7] is an open source PDF document viewer for Mac OS X. It is developed in Objective-C using the Cocoa framework. Its source code is available at SourceForge[8] The implementation described in this section is based on build 7809 from November 11th, 2012 and contains 55,776 source lines of code (ignoring comments) in Objective-C.

The state of Skim contains all open documents, the window configuration, and the current page index

Skim allows users to view and navigate multiple PDF documents. Each PDF document is displayed in a designated window. Users can navigate the PDF documents by page or by zooming into a specific page and panning the view. Similar to TextEdit, all of Skim's state depends on the open document and thus its state is also stored entirely in the documents element. The relevant state information for each PDF document is:

- *file*: the document file
- *pageIndex*: the page that is currently visible
- *windowFrame*: the location and size of the window
- *scrollingRect*: the visible portion of the document in the window

This list can be extended to include other aspects of Skim's state, such as the selected navigation tool and any customizations of its user interface. For the sake of simplicity, however, these additional aspects were skipped in the current implementation. The following listing shows an example of an extracted state from Skim:

```
1 { "application": {
2     "identifier": "net.sourceforge.skim-app.
          skim",
3     "name": "Skim",
4     "type": "pdfviewer" },
5   "global": {},
6   "documents": [{
7     "file": {
8       "modificationDate": "2013-03-18T15
          :44:44Z",
9       "edited": false,
10      "url": "file://localhost/Users/demo/
          example.pdf",
11      "name": "main",
```

[7] http://skim-app.sourceforge.net
[8] http://sourceforge.net/projects/skim-app/develop

```
12        "type": "com.adobe.pdf" },
13      "pageIndex": 2,
14      "windowFrame": {
15        "y": 150,      "x": 1113,
16        "width": 803, "height": 1028 },
17      "scrollingRect": {
18        "y": 0,        "x": 0,
19        "width": 803, "height": 957 } }] }
```

State I/O was implemented in two classes: the application delegate and the document controller

Similar to TextEdit only two classes had to be modified to implement State I/O support. The application delegate called *SKApplicationController* was used to initialize the library upon launching Skim. State extraction and restoration was implemented in the *SKMainDocument* class, which is responsible for a single open PDF document. This class was extended with two methods to extract and restore state. Like TextEdit, these methods are used to extract the state of each document from the SKApplicationController.

State synchronization was implemented by observing the window and the current page index

To implement support for state synchronization, the SKMainDocument was configured to observe changes to the window, the scroll view, and the current page index. Based on these notifications, a state change object is generated and sent to the service. Since Skim cannot be used to edit PDF files, the file contents of the an open file cannot change. Thus, state synchronization only applies to the user interface showing the PDF document, most importantly the page that is currently shown.

SkimRemote allows users to navigate the current page of Skim from a remote device

A second application called SkimRemote was implement that shares state with Skim. SkimRemote runs on a mobile device and does not display a PDF document. Instead, it was designed to be synchronized with a running instance of Skim on a separate device that is remote controlled through SkimRemote. To this end, SkimRemote displays the current page index of the synchronized state and offers two buttons to increment and decrement this page index. Pressing a button, sends a state change event to Skim that reflects the change in the page index. Skim reacts to this change event by updating its current page to the state in the change event. In consequence, SkimRemote can be used to navigate the current page of PDF document viewed in Skim from a remote device. Figure 5.7 illustrates how SkimRemote and Skim can be used in combination.

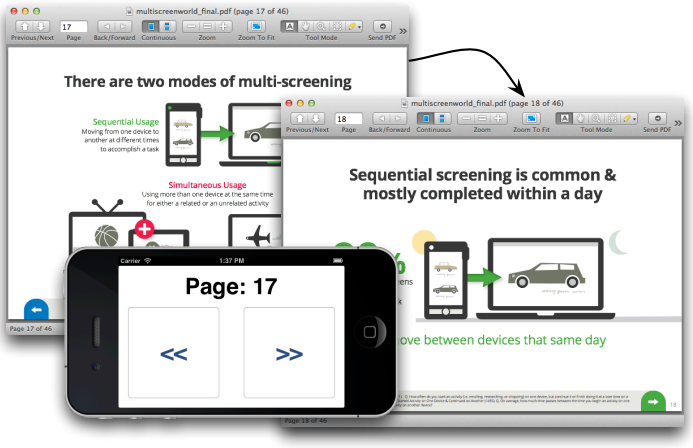

Figure 5.7: State synchronization between Skim and SkimRemote: When SkimRemote is synchronized with Skim, it displays the index of the visible page and offers to buttons to increment and decrement this page index. Upon tapping a button, the state of Skim is changed to reflect the new page index and thus navigate the PDF document.

5.3.4 Example Clients

Three different example clients have been implemented to illustrate how different interaction techniques can be developed separate from the task applications. These clients communicate with the state exchange service via the State Exchange programming interface to initiate and control multi-device interactions.

Web Control Center

The web control center gives advanced users direct access to the functions of the state exchange interface

The *web control center* lists all applications running on a device and allows users to extract the state of each running application. The extracted state can be inspected, stored as a file, and later restored to resume operation of the task stored in the state. The web control center was designed to assist the development of the state exchange service. As such, it can be a valuable tool for advanced users, who can use to gain unlimited access to all of the functions of the state exchange programming interface. Figure 5.8 shows a screen shot of the web control center.

Application Integration

State synchronization between SkimRemote and Skim is initiated directly from the SkimRemote application

The client to set up state synchronization between SkimRemote and Skim was integrated into SkimRemote. After launching SkimRemote, a list of all discovered devices is shown where a state exchange service is running. When

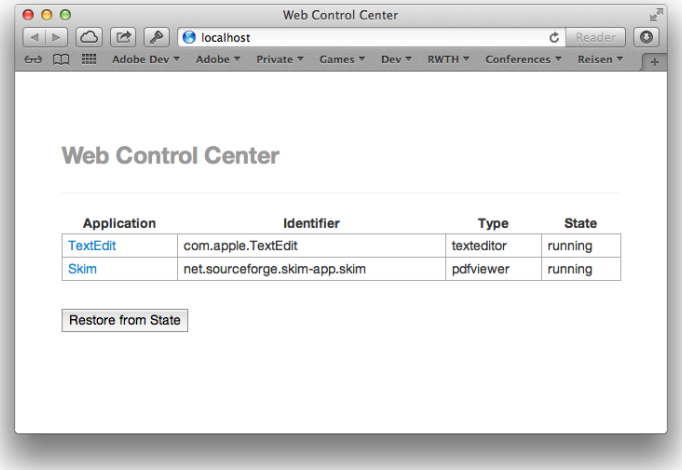

Figure 5.8: The web control center provides direct access to the state exchange interface. It shows a list of all supported applications and allows the state of each application to be extracted by clicking on the application. The extracted state can be inspected or saved to a file, which can be restored by clicking the button at the bottom of the control center.

the user selects a device, SkimRemote connects to the device via the state exchange interface and waits until Skim is launched on the device. Once Skim is launched, SkimRemote requests the state of Skim including the observe option such that from then on, the service forwards all state change events to SkimRemote. When the user presses a button, the state change event generated by SkimRemote is sent back to Skim as a request to restore the partial state.

MagicPad

The MagicPad allows users to transition tasks between a mobile device and a desktop computer by placing the mobile device on a designated pad

The *MagicPad* demonstrates how innovative new multi-device interaction techniques can be implemented based on application state. MagicPad is a hardware pad that can be sensed by mobile devices. Once a mobile device is placed on the pad, the state of the application running on the mobile device is transferred to and restored on the nearest desktop computer. When the mobile device is lifted, the application state is extracted from the desktop computer and transferred back to the mobile device, where it is restored. This way, a user can simply place her mobile device on this pad to transition an ongoing task to the desktop and lift it up again to transition the task back to the mobile device.

The MagicPad emits a strong magnetic field that can be detected by the mobile device to initiate a state transfer

Due to restrictions of the iOS mobile operating system, support for the MagicPad could not be implement for generic tasks. Instead, it was integrated into a custom text editing application for iOS. This mobile text editor shares state with TextEdit such that its state can be restored in TextEdit and

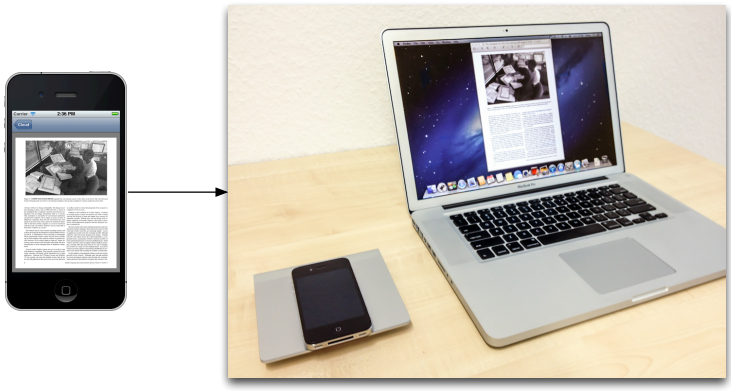

Figure 5.9: The magic pad automatically initiates a state transfer between a mobile device and a desktop computer. When the mobile device is placed on the pad, the active application on the mobile device is migrated to the attached desktop computer. When the mobile device is lifted, the application is migrated back to the mobile device.

vice versa. The MagicPad contains a strong magnet, which is detected by the mobile device using the built-in digital compass. Once the MagicPad is detected, the mobile text editor triggers a state extraction for itself and transfers the state to a pre-configured device. Afterwards, as soon as the MagicPad is no longer detected, the mobile text editor triggers a state extraction at the remote device and transfers the state back to itself, where it is restored. The need to preconfigure a target device for the MagicPad can be circumvented by connecting the MagicPad to the desktop computer and watching for synchronous changes of the magnetic field around the pad.

5.3.5 Discussion

The state exchange system architecture supports state extraction, persistence, and restoration

Similar to the first iteration, the final iteration of the state exchange system architecture enables task migration between devices by allowing the state of applications to be extracted (R1), transferred (R2), and restored (R3). Furthermore, the presented system implementation shows that the architecture can be integrated into a typical operating system using only cross-platform technologies. Thus, the solution also supports platform interoperability (R6).

State composition is enabled by the semantic structure of the extracted application state

The semantic structure of the state that reflects which aspects of the state are related to an open document enables state composition (R4): All tasks that are conducted with an application can be distinguished in the extracted application state by the document that is affected by the task. Since applications are not designed to support multiple tasks being performed on a single document, the deduction of a

task from a document is typically unambiguous. Clients can make use of state composition and remove the parts of the state that are not needed in a transfer. Tasks that span multiple devices can be captured by extracting the state of all related applications. By combining these two methods, a task spanning multiple applications unexclusively can be captured and migrated or coordinated.

Application state can be shared across multiple applications by employing a consistent structure for the same state type

The extracted state also supports state sharing, i.e., multiple applications sharing a single state definition such that tasks can be migrated between these applications by migrating state. Each application state includes a state type, which can be used to deduce how the state will be structured. These types can also be used to define standard types for commonly used application classes that all applications of that class are encouraged to support. At the same time, the state format is flexible enough to also support the use of custom keys to augment a standard state structure with additional information relevant only for specific applications.

Two applications can be synchronized by forwarding state change events between both applications

The state between two compatible applications can be kept in synchronization (R7) by transferring the state from one application to the other and then forwarding all state change events between the two applications. This way, changes to the original application are transmitted via change events that were requested upon extraction to the replicated application. Changes from the replicated application are transmitted to the original application via change events that were requested upon restoration. As demonstrated by SkimRemote, state synchronization can be used to enable other means of coordination by developing custom applications that provide a special coordination interface, such as a remote control interface, for an existing application.

Separation of control is enabled by the distinction between State I/O and State Exchange

Finally, the architecture provides a clear separation between task applications and clients (R8). Task applications are responsible for implementing the State I/O programming interface and thus respond to state extraction and restoration requests. Clients, on the other hand, use the state exchange programming interface to initiate state extraction and restoration requests on available task applications and manage synchronization and adaptation procedures of the extracted states. This separation was demonstrated with the development of several multi-device interaction clients, separate from any task applications. At the same time, the separation is not exclusive, as the integrated

client demonstrates that allows client functionality to be executed from inside a task applications.

5.4 Validation

This section argues that the proposed system architecture is a valid solution for supporting multi-device interaction in the wild in a way that is compatible with today's interactive system architectures. In summary, the argument is as following: Any solution that fulfills the requirements listed at the beginning of this chapter provides adequate support for multi-device interaction in the wild. Since the proposed system architecture fulfills all of these requirements, any interactive system that integrates this architecture supports multi-device interaction in the wild. The final section will then discuss limitations of the approach.

5.4.1 Requirements Validation

The requirements presented in this chapter accurately describe a system that supports multi-device interaction in the wild through application state

The requirements described in section 5.1 accurately describe a system that provides support for multi-device interaction through application state. Task migration is enabled by extracting, transferring, and restoring application state, which can be migrated across a diversity of applications and devices. In the stored state multiple tasks running in a single application can be distinguished and multiple states can be combined to reflect tasks spanning multiple applications. Task coordination is enabled through state synchronization, which gives applications a communication channel to coordinate their execution across multiple instances. This communication channel can be used to keep the appearance synchronized. However, it is also possible to use it for other coordination activities, such as sharing control. Finally, the separation of control enables designers to design and users to use multi-device interaction techniques separate from task applications.

The first iteration of the proposed system enables task migration with special focus on application interoperability

The first iteration of the state exchange system architecture fulfills a large part of these requirements. The system provides task applications with the means to extract, transfer, and restore state. The extracted state can be transferred between different system and restored in diverse applications. The organization of application state into different contexts, enables applications to provide support for

as many application classes as they deem fit. Defining a standard context structure for typical application classes ensures that applications can share their state just by adhering to this standard. At the same time, the state structure is flexible enough to support individual differences of applications without interfering with one another or the standard.

The second iteration of the proposed system fulfills all of the requirements

The second and final iteration of the state exchange system architecture extends the previous iteration by integrating support for state synchronization, state composition, and separation of control. State synchronization is enabled by allowing state to be observed and mediating change events between the applications that are coordinated. This implicit coordination channel can be used to keep the appearance of multiple task applications in synchronization or to create specialized coordination interfaces for existing applications. State composition allows tasks to be captured at a finer level of granularity by allowing tasks that span multiple applications unexclusively to be extracted and migrated or coordinated. Finally, separation of control enables the separate development of task applications and multi-device interaction techniques by separating them through two well-defined programming interfaces. Separation of control helps integrating multi-device interaction into the complex daily routine of users, where tasks, applications, and device setups frequently change.

The state exchange system architecture is one possible solution for the implementation of the application state conceptual model

In consequence, the architecture presented in this chapter fulfills all the requirements and thus is suitable to be used for multi-device interaction in the wild. However, just like application state is only one possible conceptual model to assist multi-device interaction in the wild, this architecture should also be considered as one possible implementation of the application state conceptual model. Just like any solution, the state exchange system architectures comes with several advantages and disadvantages. The advantages have been largely discussed in this section and the requirements analyzes of the individual iterations. The following section will analyze the disadvantages of the solution in the form of limitations.

5.4.2 Limitations

The architecture proposed in this chapter has several limitations.

Device discovery is limited to local devices

The device discovery mechanisms employed by the system are limited to devices that are close to each other (Bluetooth) or within the same technical infrastructure (Bonjour). It is not possible to discover or remember remote devices such that it is not trivial for clients to initiate connections with devices at remote locations. However, accessing remote devices might be beneficial for users in some situations, e.g., when they need to transition a task from their home computer to their computer at work. To support remote discovery, the system could be extended with a central device registry, where all of the user's devices are registered and update their network location whenever they are online.

Task coordination is limited to coordinating multiple devices via a synchronized state

Task coordination is limited to the types of coordination that can be realized using state synchronization. State synchronization provides an implicit communication channel between multiple devices that keeps the state of a task consistent across all connected devices. This implicit communication is sufficient to perform several coordinated activities as demonstrated with the two examples in the previous section. However, it does not allow the remote execution of a particular piece of functionality. For example, it is not easily possible to print a document on a printer that is connected to a remote computer from a mobile device. In some cases, the execution of remote commands can be enabled by extending the application state with otherwise unused keys. The remote device then listens to change events to these special keys and triggers a pre-configured functionality when an event is received. However, this workaround reduces the semantic coherence of the state and should thus be applied only sparingly.

Task synchronization can lead to conflicts, which are ignored by the system

Task synchronization can lead to unresolved conflicts: If two parties change the state of two synchronized applications at the same time, the change can lead to a conflict. In the current implementation, these conflicts are ignored and the latest change simply overwrites any intermediate changes. In most cases, this will be sufficient, as the different devices are used in a coordinated fashion, such that the users can easily detect and circumvent any synchronization conflicts. The TextEdit example, however, would benefit from a more sophisticated synchronization procedure. To this end, the operational transformation algorithm by Ellis and Gibbs [1989] could be used to merge conflicting events automatically.

Legacy task applications are not supported without modifying their source code

The system does not support legacy task applications without changing their source code. State extraction, restoration, and synchronization must be implemented inside the task application to allow it to participate in state exchange. Thus, applications with inaccessibly source code cannot be supported. This is a severe limitation for any system that strives at supporting multi-device interaction in the wild, as users are very reluctant of changing their tools for their daily routine. In the next chapter, some approaches to integrate support into legacy applications that do not need access to the source code are presented. However, the optimal solution is to encourage application developers to integrate support for their applications. This can be only achieved, if state exchange becomes a standard in all operating systems with ample support such that application developers are strongly encouraged to implement the needed methods.

The overhead of implementing support for state extraction, restoration, and synchronization lies with the application developer

Finally, state extraction, restoration, and synchronization must be customized to the application logic and implemented by the application developer. This places much of the burden of enabling multi-device interaction on the application developer. The author strongly believes that this limitation cannot be eliminated completely since applications exhibit very different behavior, which is reflected in very diverse extraction, restoration, and synchronization mechanisms. However, it is possible to assist application developers further in implementing the necessary methods by relying on common behavior throughout applications that is rooted in commonly used system frameworks. Supporting application developers this way is demonstrated by example of the Cocoa framework in the next chapter.

Chapter 6

Integrating State Exchange into Legacy Systems

The biggest hindrance of establishing a system like state exchange is to support a sufficient number of task application for the system to become useful for the user. In the state exchange system architecture, this support is enabled by implementing the State I/O programming interface inside the task application. In this chapter, different approaches are presented and evaluated that simplify the integration of State I/O into task applications.

The first approaches ease communication with the state exchange service; the other approaches make use of system frameworks to automate different parts of the state exchange implementation

The first two approaches provide useful abstractions to application developers, which ease the communication with the state exchange service. These abstractions can be implemented in any system environment. The other approaches are based on the unique system frameworks included in Mac OS X and cannot be immediately replicated in other systems. These approaches highlight by example of Mac OS X how features of the operating system can be used to deduce information about the state of applications. To port these approaches, similar functionality must be identified and exploited in other systems.

6.1 State I/O Support Library

The State I/O support library presented in section 5.3.2 provides several useful abstractions to simplify the implementation of the State I/O interface into task applications. It manages the communication with the state exchange service, provides abstractions for the interface methods, and a data structure for state.

The State I/O library allows application developers to focus on implementing state extraction and restoration

By using the State I/O support library, application developers do not need to be concerned about how to communicate with the state exchange service. Instead, they can focus

on how to implement state extraction and restoration. Using the state data structure, the extracted state can then be transferred between the application and the service without the application developers needing to concern themselves with the underlying data format.

6.2 State Object

In the State I/O library state synchronization is supported by calling a designated method every time the application state changes. This method call must be accompanied by an object describing the change. The goal of the state object is to combine state extraction and state synchronization into a single object.

The state object holds the application's current state in a tree structure

The state object holds the complete application state at any given time. The aspects of the application state are stored in a tree with named branches that reflects the structure of application state introduced in section 5.3.1. All of the values stored in the tree can be observed, i.e., when they are changed a notification is posted, which can inform the observer about the change.

The state object contains all the information needed for state extraction and synchronization

The state object is used to enable state extraction and synchronization. When state extraction is triggered, the current state object is simply exported and transferred to the state exchange service. When a state observer was added to a given state, any modifications of the state by the application are immediately transferred to the state exchange service as state change events. There is no need for the application developer to implement any of this behavior.

Application developers keep the state object updated by replicated any state changes in the object or using it as a model object

When using the state object, the application developer must ensure that the state object always reflects the state of the application. Thus, whenever the application's state changes, this change must also be applied to the state object. For example, when a user interface element changes its state, the application developer must implement a mechanism that duplicates this change in the state object. Since in most cases the application must respond to such a change to its UI in any case, updating the state object can be simply integrated to the existing response logic. In other cases, the state object can be used as a model object for the UI to enable loose coupling between UI and application logic. For example, a tool selection palette can be configured to change the "selectedTool" property of the state object. The

application logic can then observe the state property and update the cursor accordingly without needing a direct connection to the user interface.

By maintaining a state object application developers do not need to implement state extraction or synchronization explicitly

The state object allows application developers to entirely neglect state extraction and synchronization. Instead, they must maintain the state object and ensure that it always reflects the state of the application. Depending on the application, this programming pattern can significantly ease the implementation of State I/O. In some cases, the pattern can even improve the underlying application architecture because it facilitates loose coupling between user interface objects and application logic.

6.3 Automatic Document Extraction

In the state exchange system application state is organized by open documents to enable clients to distinguish between individual tasks that are executed in parallel in a single application. Automatic document extractions aids the application developers with this organization by determining all open documents and preparing the application state accordingly. The approach is based on the Cocoa document architecture.

The Cocoa document architecture is a system architecture for multi-document application

The *Cocoa document architecture* is a system architecture that application developers are encouraged to use to implement any kind of application that can open one or more files, called documents, and show each document in one or more designated windows. The architecture defines a mechanism to manage documents in the application and a life-cycle for the documents: All documents are managed by a single controller class that is instantiated by the system. This controller maintains a list of all open documents and is used to create and initiate new documents, which are represented by document objects. The life-cycle of a document defines how the document is created including the configuration of its user interface, how its contents are loaded from or save to a file, and how it is closed.

Application developers create a subclass of the abstract document class for each document type they want to support

To aid developers in implementing this architecture, Cocoa includes a system framework that automates many parts of the implementation. The system framework includes a fully-functional standard implementation of the controller that manages the open documents. In addition, the framework defines an abstract document class that defines the

frame for implementing the document objects. To make use of this standard functionality, application developers create a subclass of the document class for each document type they want to support and configure the application to associate their subclass with the appropriate file type. Then, they implement the logic needed to load/save the document contents from/to a file. Finally, the user interface of the document is set up and any additional needed functionality is implemented in the document class.

The Cocoa document architecture is exploited to enable automatic document extraction

If an application uses the Cocoa document architecture, the standard document management and life-cycle can be used to automate the organization of application state according to documents. To this end, the state extraction mechanism is split up into two parts: The global state is configured from the application delegate, which is a central object that manages the entire application. The state of each document included in the documents element is then configured from each of the document objects that are managed by the document controller. State restoration can be split up in the same way. In other words, state extraction and restoration is integrated into the document life cycle.

Automatic document extraction splits state extraction and restoration into semantically coherent parts

Integrating state extraction and restoration into the document life cycle is beneficial for application developers, because it splits up the process into semantically coherent parts: The application delegate is extended with two methods that extract and restore the state of the application that is independent of any documents. The document objects are extended with two methods that extract or restore the state that is relevant for the document. When state extraction or restoration is triggered, the system can call these methods on the appropriate objects to construct or restore the complete application state. If needed, new documents can be created using the document controller and configured by calling the state restoration method.

6.4 Automatic User Interface State Extraction

While the previous sections focused on supporting application developers in their implementation of state extraction and restoration, this section explores how some aspects of the state can be extracted and restored automatically. In particular, the section explores how the inspection and con-

figuration capabilities of standard UI elements can be exploited to automatically derive their state.

User interface elements are represented by view objects, which are organized in a tree structure

In Cocoa, the UI is represented by view objects. A view object contains the capabilities to render a widget on the screen and listen to user events that affect the rendered widget. View objects are organized in a tree structure that reflects the structure of the user interface. The root element is always a window object, representing the window that the views are placed in. The tree structure and all properties of standard view objects can be inspected at runtime.

User interface state can be extracted by replicating the tree structure of the view objects in the state and storing all view properties at the appropriate elements

To extract the state of the UI, it is sufficient to extract the state of all view elements in the UI in a way that reflects their organization in the UI. Since both the UI and application state are organized as a number of trees, the structure of the UI can be reflected in an equal structure in the application state. This tree structure can then be used to store all properties of the view objects that are relevant for its state. For example, the state of a button includes at a minimum the button label and the "pushed" state of the button. Custom view objects, i.e., view objects defined by the application developers, cannot be included in the state extraction because their properties are unknown.

User interface state is restored by updating the user interface elements according to the stored state

To restore the automatically extracted UI state, the stored view properties are read from the state and applied to the appropriate view elements. This process assumes that the UI exists at the time of restoration and that its structure is the same as when it was extracted. In other words, the automatic UI state extraction can only work with a static UI structure, where it is guaranteed that the tree structure containing all view elements does not change. The state can then be restored by traversing both the stored state structure and the view structure and applying the stored view properties to the view objects.

The UI can be created from the Cocoa document architecture

The assumption that the UI must exist at the time of restoration is fulfilled if state restoration is integrated into the Cocoa document architecture as described in the previous section. By using the Cocoa document architecture, the system can create a new document from the document referenced in the state, which will instantiate the default UI for that document. In a second step, this UI is then updated according to the stored view properties.

Differences in the UI structure between the time of extraction and the time of restoration can lead to incomplete or erroneous restoration

The assumption that the created UI must have the same structure as the UI at the time of extraction, however, cannot be guaranteed for all applications. If the created UI differs from the stored UI, the state restoration may be incomplete or erroneous. An incomplete restoration occurs if the location of a UI element is different between the time of extraction and the creation time. In this case, the state of the UI element is not restored. An erroneous restoration occurs, if a different UI element is located at the place of a stored UI element. If the difference between these UI elements is not detected, the restoration can result in a wrongly configured UI element or in a runtime error that terminates the application. To avoid erroneous restoration, the application state should include the type of the UI element (name of the view class) and match this type with the type of the element during restoration.

Restoring view properties can lead to inconsistencies between the UI and the application, which are avoided by triggering view update events

Finally, extracting and restoring only the state of UI elements can lead to inconsistencies between the user interface and the application. For example, the UI may be restored to show the "annotate" tool as active while internally the "selection" tool is active by default. This inconsistency can be avoided by triggering the same mechanisms that are used when the user changes the value of a view property, when restoring the view property from a state. In above example, the UI restoration can trigger a change event after restoring the tool selection, which will then update the application logic to match the tool selected in the UI. In Cocoa, this is the default behavior when setting view properties.

The approach takes away the need to implement state extraction and restoration for the user interface at the cost of a semantically meaningful structure of the state

Using the described approach, application developers no longer need to provide a custom implementation to extract and restore the user interface state of an application. Instead, they can focus on how to extract and restore the remaining application state and ensure that the automatic UI restoration does not fail because of the limitations mentioned above. In many cases, this will likely decrease the overall effort of implementing complete support for state extraction and restoration. However, the structure of the automatically generated UI state does not provide a good semantic representation of the state. Since view objects in the UI are structured as an ordered tree without named keys for the branches, the automatically extracted UI state also cannot contain meaningful keys to reference the UI elements. Thus, the automatically extracted state is comprised of unlabeled lists of elements that are hard to distin-

guish from one another. Consequently, automatic UI extraction can help application developers implement support for state extraction and restoration but only at the cost of reduced semantic coherence of the extracted state.

6.5 Automatic State Extraction via Resume

Resume is a feature of Mac OS X that allows terminated applications to be resumed at the state when they were terminated. To support Resume, an application must create a persistent copy of its state on the disk before termination, which is then loaded to restore the state upon resumption. This feature can be used to enable automatic state extraction and restoration.

Resume requires application developers to create a persistent copy of the application's state, which is organized according to the internal structure of the application

It is the application developer's responsibility to extract and store the state of the application before termination such that the application can restore its appearance from the stored state upon resumption. The state extraction is done in a multi-step process, where different parts of the application are asked for the part of the state that is relevant to them. These parts are closely related to the Cocoa document architecture: The application delegate is queried for any global state, and each document, window controller, and all view objects for their state. However, the extraction process does not include user content. Instead, user content is stored using the *Autosave* feature, which ensures that all user content is always stored persistently on the disk to avoid loss of data.

The Resume feature can be used to enables state migration and state composition but lacks immediate support for the other requirements of multi-device interaction in the wild

The state that is extracted through the Resume mechanism contains everything that is needed to resume operation of the application. This information is stored in a way such that different parts of the application, including all open documents, can be distinguished. Consequently, the state can be used to enable task migration with support for distinguishing between the tasks that are currently executed in the application. However, state extraction can only be triggered by the system, which is only done upon application termination. Additionally, the extracted state is typically not organized in a semantically coherent format making it is hard to interpret and convert the state to enable state sharing or platform interoperability. Finally, state synchronization is not supported by Resume.

Separation of control can be integrated by injecting custom functionality into the application

The need to terminate an application to extract its state can be eliminated using a custom script that triggers state extraction from inside the application. This script must be injected into the application via AppleScript as described in the next section. Afterwards, the script can be executed from an external process to extract the state of the application into a file, which can then be transferred to a remote device. In addition to the state file generated via the Resume feature, the autosaved documents are also extracted, using the approach based on the Cocoa document architecture described above. Finally, the script can be executed to restore its state from a previously extracted state by supplying the location of the state file and the related user content.

State sharing is possible but complicated by the format in which the state is stored and the way it is extracted

The format of the state files is stored in the *NSCoding* format, which is the standard Cocoa data format. However, this format cannot be stored in a human-readable format, because it directly represents the runtime object structure, which can contain circular relationships between objects. Additionally, the state information is not stored in a semantically abstract way but close to the representation of the application on the platform where it was extracted. For example, the position of a viewer application in a PDF document is stored as a scrolling offset in pixels instead of the page number. Consequently, even though all necessary information is contained in the state file, it is hard to extract and interpret the relevant information for the task to create custom applications that share the same state.

The Resume enables state migration at the cost of reduced semantic abstraction of the state

The Resume feature represents the strongest opportunity for completely automatic state extraction and restoration on Mac OS X. It provides the complete state of the application in a way that can be extracted at runtime and distinguished at the task level. The state can be stored persistently and transferred to other systems, where it can be restored to restore the application to the state of extraction. However, state extraction using the Resume feature comes at the cost of greatly reduced semantic abstraction of the state. The automatically extracted state is stored in a way that closely resembles the internal organization of the application, which makes the process of standardizing state for specific application types and converting state between applications much harder. Thus, even though the Resume feature can be used to enable state migration where it would otherwise not be supported, it is not a sustainable

solution to just rely on this feature for state migration in the future.

6.6 Enabling Third-party State I/O Integration

The previous approaches have focused on simplifying the implementation of State I/O for application developers. In this section, a different approach is taken to increase opportunities for State I/O adaptation: By making it possible to implement support for legacy applications without access to the application's source code, third-party developers can integrate State I/O on behalf of the application developers. This way, more people are empowered to extend existing applications with support for state exchange.

Runtime toolkit overloading enables third-party developers to modify the behavior of legacy applications by providing six abstractions

The approach is based on *Runtime toolkit overloading* by Eagan et al. [2011], which allows developers to inject custom functionality into legacy applications to change the behavior of these applications. Runtime toolkit overloading provides six abstractions that help developers in creating these modifications:

- *Window or widget hooks* allow developers to adapt the appearance of the UI before it is rendered.
- *Event funnels* provide mechanisms to intercept and respond to arbitrary events.
- *Glass sheets* are overlays on top of the existing UI that can be used to augment or replace the UI with custom widgets.
- *Dynamic code support* enables developers to load custom code into the running application and replace and extend existing functionality at the class level.
- *Object proxies* allows developers to override and extend existing functionality at the object instance level.
- *Code inspection* provides several tools to inspect the structure of the application and its user interface at runtime.

Dynamic code support can also be used to integrate support for State I/O into legacy applications

These abstractions assist developers in two ways: They enable developers to create modifications without needing a deep understanding of the application structure, and they help developers in gaining a deeper understanding of the

application if necessary. In the context of this thesis dynamic code support is used to enable third-party developers to integrate State I/O support into legacy applications.

A ScriptingAddition extends all applications on a system with custom functionality that can be executed via AppleScript

On Mac OS X any application written with the standard system framework (Cocoa) can be extended with custom functionality through a *ScriptingAddition*. A ScriptingAddition is a static library that contains compiled code and an AppleScript interface. By installing the ScriptingAddition in the system, the AppleScript interface is automatically injected into any application as soon as the application is started. By executing the commands defined in the AppleScript interface of the library on a third-party application, the library is loaded into the program context of the application and the custom functionality is executed.

AppleScript commands are executed from a Cocoa application or a command line tool by sending the command and parameters as a string to the target application

AppleScript commands can be executed in diverse ways. Cocoa provides several classes to assist the creation and execution of an AppleScript interface. In addition, commands can be executed from a designated command line tool. An AppleScript command can contain parameters and return data including complex objects. However, the command, the parameter, and the returned value are encoded as a string and thus should not contain long segments of binary data.

State I/O support can be loaded into a legacy application through a ScriptingAddition

Through a ScriptingAddition, the State I/O support library or any of the approaches mentioned above can be loaded into legacy applications on Mac OS X. State extraction and restoration can then be executed in two different ways: The application can be configured to connect to the state exchange service as described in section 5.3.1, or the AppleScript interface can define two methods to execute state extraction and restoration directly. Connecting to the state exchange service has the advantage of a continuous communication channel between the application and the service, which can be used for state synchronization. However, due to restrictions of the application sandbox[1], many applications are not allowed to open a socket connection to another service. The AppleScript interface is not affected by this limitation.

A ScriptingAddition can also be used to load a custom implementation of the State I/O interface

In addition to a support library, the ScriptingAddition can also be used to load a custom implementation of the State I/O interface. Through this custom implementation, a

[1] `http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide`

legacy application can be augmented with full support for State I/O without needing access to the source code. The code inspection tools from Runtime toolkit overloading can help third-party developers to find the appropriate state values to be included in the extracted state and to update these values when given a state for restoration.

6.7 Example Implementation

Different approaches to integrate State I/O have been implemented in a prototype

Several aspects of the approaches presented in this chapter have been implemented in a prototype on Mac OS X. The prototype is composed of three components:

- The *NomadicApps ScriptingAddition* loads the NomadicApps library or a custom implementation of State I/O and defines an AppleScript interface to trigger state extraction and restoration.
- The *NomadicApps library* is loaded into applications to assist or automate the implementation of state extraction and restoration.
- The *NomadicDesktop* application allows users to initiate task migration by extracting, transferring, and restoring state from a menu in the system menu bar.

The prototype is not compatible with the state exchange architecture

Due to the limitation of the application sandbox that prevents many applications from communicating with services via a socket and the incompatibility of the automatically extracted state with the state format suggested in section 5.3.1, the prototype is not compatible with the state exchange architecture.

6.7.1 NomadicApps ScriptingAddition

The NomadicApps ScriptingAddition defines an AppleScript interface that is injected into all applications on the same system. The interface defines the following four methods:

1. Return a list of all documents that are currently opened in the application.
2. Close the specified document.
3. Extract the application state that is related to the specified document into a file and return the file path.

4. Restore the document and its related application state from the file at the specified location.

The NomadicApps ScriptingAddition defines the methods needed for state extraction and restoration

The first two methods are used to identify and manage the open documents in an application. The last two methods are used to extract and restore the application state related to a specific document. The extracted state is returned by means of a file in the file system, avoiding the overhead of sending large binary data via the AppleScript interface.

Upon triggering the AppleScript interface a custom or automatic implementation of the interface methods is loaded and executed

The first time, any of these methods is executed, the ScriptingAddition loads the NomadicApps library into the running task application. Then, a custom implementation of state extraction and restoration, if available, or the automatic implementation from the NomadicApps library is used to perform the appropriate command. A custom implementation is provided via a plug-in, i.e., a static library that includes an implementation of the state extraction and restoration methods and a description of the application that it is appropriate for.

6.7.2 NomadicApps Library

The NomadicApps library is designed to be loaded into legacy applications and support application developers in implementation state extraction and restoration or provide an automatic implementation based on the Resume feature. It contains two classes:*NAState* and *NADocumentCoder*.

The NAState class is used to create, access, and manage an application state stored on the disk

The NAState class represents a file structure that is used to stored extracted application state on the disk. The application state is stored in three files: The info file contains a description of the application and the document. The document file contains the user content in the format defined by the application. The supplement file contains the exported application state from the Resume feature. In addition, the file structure can include a screen shot of the application at the time of extraction. The NAState object implements several convenience methods to create and manage state files on the disk.

The NADocumentCoder class extracts and restores state by saving or loading the document and extracting or restoring the related application state

The NADocumentCoder class implements automatic state extraction and restoration based on the Cocoa document architecture and the Resume feature. The list of open documents and a mechanism to close a document can be readily obtained from the document controller included in the document architecture. State extraction and restoration is

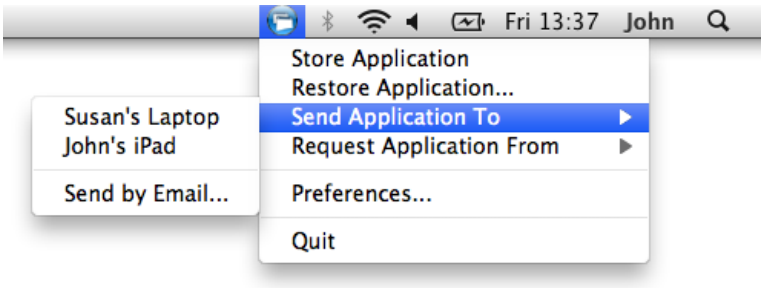

Figure 6.1: The menu included in NomadicDesktop allows users to extract the state of the active document and store it as a file or send it directly to a peer device.

done by first storing the specified document to NAState's document file and, finally, extracting and storing the application state related to the document in the supplement file. State restoration first opens the document contained in the NAState object and then updates the document's user interface according to the state stored in the supplement file.

6.7.3 NomadicDesktop

NomadicDesktop is an application that can be used to initiate application migration in two ways: First, it creates a menu in the Mac OS X menu bar, which allows users to extract the state of the active document and store it in a file or transfer it to a remote device. Second, it allows users to extract the state of an arbitrary document and application running on the system by selecting it from a list. An extracted state stored in a file can be restored by opening it with NomadicDesktop and initiating state restoration.

The NomadicApps menu allows users to interact with the state of the currently active document and application

Figure 6.1 shows the menu included in NomadicDesktop. Using the menu, users can only extract the state of the active document. The state can be either stored in a file or sent to a remote device. All remote devices running the NomadicDesktop application on the same network are automatically discovered and shown in the list of possible transfer targets. By clicking on a target device, the state of the current document and application is migrated to the target device in one step. Finally, the menu can be used to migrate the active document and application running on the remote device to the local device.

The application browser is used to inspect and extract the state of any running application

Figure 6.2 shows a screen shot of the NomadicDesktop application browser. In the left column, all running applications are listed and can be selected by clicking on the application name. Once selected, the main part of the screen reveals a list of all open document in the selected applica-

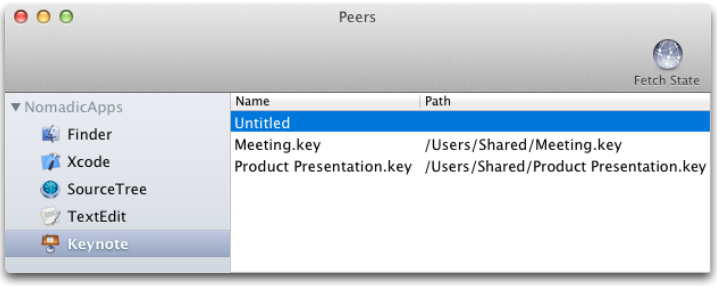

Figure 6.2: The NomadicDesktop application browser allows users to select and extract the state of any active application and document.

tion. These documents can be selected and extracted into a state file. In addition to the local application browser, it is also possible to open an application browser on a remote device. This allows users to inspect, extract, and transfer the state of applications running on remote devices.

The automatic state extraction and restoration of the NomadicDesktop prototype works with a variety of applications

The automatic implementation of state extraction and restoration was tested successfully with the 18 different applications listed in Table 6.1. This list includes many of the applications that are bundled with Mac OS X. Additionally, the iWork application suite from Apple is also supported. The remaining applications are made of a mix of smaller commercial and open-source applications for the Mac. Large application bundles, such as Microsoft Office or Adobe Creative Suite, are not supported because they are not developed in Cocoa.

6.8 Discussion

The goal of this chapter was to explore different approaches to simplify the integration of State I/O into legacy applications. This goal was addressed from three sides: Simplify the integration process for application developers, automate State I/O integration, and empower third-party developers to integrate State I/O on behalf of the application developers.

To convince application developers to integrate support for State I/O, the barriers of integration should be lowered as discussed in this chapter and the functionality should be anchored in the operating system

The approaches focused on supporting application developers are universally applicable. They can be readily ported to other operating systems and device classes. These approaches, however, still require work on behalf of the application developers. The best way to encourage application developers to implement this support is to lower the barriers of implementation as demonstrated with the discussed approaches and to integrate them deeply into the

Application	Version
AppleScript Editor	2.4.1
Automator	2.2.1
Chess	2.4.3
Grab	1.6
Grapher	2.2
Preview	5.5.1
Quicktime Player	10.1
TextEdit	1.7
BibDesk	1.5.4
CocoPad	1.0
GarageBand '11	6.0.4
Keynote '09	5.1
Numbers '09	2.1
OmniGraffle	5.3.2
OmniOutliner	3.10.3
Pages '09	4.1
Skim	1.4.4
TeXShop	3.04

Table 6.1: The listed applications can be migrated using the NomadicDesktop prototype.

operating system. Through this deep integration, users will become acquainted with the features enabled by state exchange. In consequence, users will actively demand that their applications' fulfill the requirements to participate in state exchange and thus give application developers sufficient incentive to make their applications compatible.

To empower third-party developers or enable automatic implementation of State I/O support can help the adoption of state exchange, but it should not be considered a permanent solution

The other approaches focused on integrating State I/O support into legacy applications on behalf of the application developers, should be considered as a bridge between today where no support is provided and a future where hopefully the application developers will readily implement State I/O support. Automating state extraction and restoration suffers from severe drawbacks in the form of reduced semantic meaning of the extracted state and its implications on state sharing, platform interoperability, and synchronization. Pushing the efforts onto third-party developers or the "community" can mitigate some of these problems, but the solutions created by third parties are hard to maintain and thus prone to compatibility issues between different versions of the software.

Chapter 7

Conclusion

The thesis provides contributions in the areas of understanding, supporting, and integrating multi-device interaction in the wild

When developing support for users in their everyday behavior, it is important to understand the user behavior that is encouraged and define a solution that embeds itself tightly into their everyday infrastructure. This thesis pursued the goal of improving everyday work with multiple devices in this matter: First, the thesis sought a deeper understanding of user behavior when confronted with multiple devices. Then, an interaction concept to facilitate effective multi-device interaction with everyday devices was designed. Finally, the tight integration of this concept into the existing tools was demonstrates with a system architecture that augments existing interactive devices with the multi-device operations provided by the concept. This thesis provides several unique contributions in each of these areas, which are summarized in the following sections.

7.1 Multi-device Interaction in the Wild

There is ample room to improve support for multi-device interaction in the wild

This thesis established that multi-device interaction in the wild as conducted today is far from its full potential. The main hindrances of this evolving behavior are a lack of support in current systems that leaves the overhead of managing and coordinating tasks across multiple devices entirely to the user and a misalignment of current research efforts in the area of multi-device interaction. If these hindrances can be overcome and efficient multi-device interaction can be integrated into common systems and used for common tasks, these solutions have the potential to reach an increasing number of users and improve their daily lives.

The thesis classifies multi-device interaction operations and uncovers its challenges

Before designing a solution for multi-device interaction in the wild, it is important to understand the unique properties and challenges of the domain. This thesis contributes to the understanding of multi-device interaction in the wild

in two ways: First, it proposes the multi-device interaction matrix, which identifies two categories of multi-device interaction in the wild: timing and task. Second, it uncovers the unique challenges of multi-device interaction in the wild: support opportunistic rearrangement of devices and tasks, transitions must be robust, support ad-hoc situations. These contributions can be used to guide the design of future solutions to address the unique demands of multi-device interaction in the wild.

By raising awareness of multi-device interaction in the wild, the author hopes to push more research into the direction of this promising domain

The most important goal of this thesis in analyzing multi-device interaction in the wild, however, is to raise awareness of this domain in the research community. Past research efforts in multi-device interaction appear to be caught in the domains of collaborative work and meeting room technology. At the same time, workers have developed their own way of using multiple devices in their everyday routine, without much support from research. By highlighting this evolving behavior, the author of this thesis hopes to encourage more work on multi-device interaction in the wild in an attempt to realign research efforts with current practices.

7.2 Exposing Application State

This thesis presents an interaction concept that addresses the challenges of multi-device interaction in the wild

In addition to raising awareness of multi-device interaction in the wild, this thesis also pursued first steps into solving the challenges of this promising domain. To this end, an interaction concept was developed that meets the challenges of multi-device interaction in the wild. This concept demonstrates how the behavior described above can serve as a basis for the design of an interactive system to address multi-device interaction in the wild. At the same time, it demonstrates a new approach to multi-device interaction that both designers and users can benefit from.

The development of the interaction concept demonstrates how the challenges underlying multi-device interaction in the wild can be leveraged in the design process

By describing not only the final solution but also the process of how it was developed, this thesis demonstrates how the insights about multi-device interaction in the wild can be used to design new solutions that are tailored to this evolving behavior. The process description validates the effectiveness of the classification of multi-device interaction in the wild and the described challenges as a design tool. Thus, it can serve as a guide for future authors to apply the

theoretical considerations around multi-device interaction in the wild in novel interaction concepts.

The interaction concept can guide the design of future multi-device interaction techniques for everyday tasks

At the same time, the concept itself is very generic and can be applied to manifold systems and situations where both users and designers can benefit from the consistent application of the concept: For users it defines a standard way of integrating multi-device interaction into their everyday tasks by turning application state into a persistent, manipulatable object. For designers it facilitates the design process of creating multi-device interaction techniques that integrate into common interactive systems by separating this integration process from the actual design of the interaction technique. The author hopes that this concept will assist the consistent adaptation of existing multi-device interaction techniques from the literature into everyday tasks, as well as the design of novel interaction techniques.

7.3 System Architecture

The thesis proposes a system architecture for exposing state that integrates into modern operating systems

To demonstrate that the concept of exposing application state as a first-class interactive object can be integrated into common interaction system, the thesis finally explored a system architecture that implements above concept on top of a typical operating system. This implementation relies on application developers to implement a standard programming interface that makes the state of the application accessible. Interaction designers can then use this interface to implement interaction techniques that can operate on all applications that support the interface.

The implementation would benefit from being integrated into operating systems at the system level

Ideally, the implementation proposed in this thesis or a similar implementation should be integrated at the system-level of modern operating systems. As with any technology that is pushed by operating system developers, integrating state exchange at the system level has the potential to encourage more application developers to implement support for state exposition in their application. Additionally, many of the implementation challenges encountered in this thesis are easily circumvented by the extensive access that operating system developers have over common applications.

Community-based and automatic integration of exposing application state lowers the initial adoption of multi-device interaction into everyday systems

With the discussion of community-based and automatic integration of state exposition, this thesis describes several methods to improve initial adoption of the new technology into modern systems – a challenge that is especially critical for introducing technology that affects people's everyday lives. Pursuing automated approaches has the potential to greatly increases the number of initially supported applications and thus ease the adoption of the new technology. Similarly, by giving third-party developers the opportunity to implement exposing application state on behalf of application developers empowers enables a large community to create custom support for multi-device interaction. However, these approaches should be considered as an intermediate step with the ultimate goal of reaching application developers to implement support for state exchange into their applications.

Appendix A

External Resources

Resource	URL
State Exchange Project Website	http://hci.rwth-aachen.de/stateexchange
Ying Zhang's website	http://hci.rwth-aachen.de/zhang
Mario Fraikin's website	http://hci.rwth-aachen.de/fraikin
Sören Busch's website	http://hci.rwth-aachen.de/busch
Ahsan Nazir's website	http://hci.rwth-aachen.de/nazir
Stefan Plücken's website	http://learntech.rwth-aachen.de/Pluecken

Bibliography

Tube with a Memory Keeps Answers on File. *Popular Science*, page 96, 1950.

Till Ballendat, Nicolai Marquardt, and Saul Greenberg. Proxemic Interaction: Designing for a Proximity and Orientation-aware Environment. In *Proceedings of the International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 121–130. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0399-6.
URL http://dx.doi.org/10.1145/1936652.1936676

Lionel Balme, Alexandre Demeure, Nicolas Barralon, Joëlle Coutaz, and Gaëlle Calvary. CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. In *European Symposium on Ambient Intelligence*, pages 291–302. Springer Verlag, Berlin / Heidelberg, 2004. ISBN 978-3-540-23721-1.
URL http://dx.doi.org/10.1007/978-3-540-30473-9_28

Renata Bandelloni and Fabio Paternò. Flexible Interface Migration. In *Proceedings of the International Conference on Intelligent User Interfaces*, IUI '04, pages 148–155. ACM, New York, NY, USA, 2004. ISBN 1-58113-815-6.
URL http://dx.doi.org/10.1145/964442.964470

Jakob Bardram, Jonathan Bunde-Pedersen, and Mads Soegaard. Support for Activity-based Computing in a Personal Computing Operating System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 211–220. ACM, New York, NY, USA, 2006. ISBN 1-59593-372-7.
URL http://dx.doi.org/10.1145/1124772.1124805

Patrick Baudisch, Edward Cutrell, Dan Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson, and Alex Zierlinger. Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch- and Pen-Operated Systems. In *Proceedings of Human-Computer Interaction*, INTERACT '03, pages 57–64. IOS Press, 2003. ISBN 1-58603-363-8.

Patrick Baudisch and Ruth Rosenholtz. Halo: a Technique for Visualizing Off-Screen Locations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 481–488. ACM, New York, NY, USA, 2003. ISBN 1-58113-630-7.
URL http://dx.doi.org/10.1145/642611.642695

Michel Beaudouin-Lafon. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 446–453. ACM, New York,

NY, USA, 2000. ISBN 1-58113-216-6.
URL http://dx.doi.org/10.1145/332040.332473

Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering JavaScript State Persistence of Web Applications Migrating Across Multiple Devices. In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 105–110. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0670-6.
URL http://dx.doi.org/10.1145/1996461.1996502

Krishna Bharat and Marc H. Brown. Building Distributed, Multi-user Applications by Direct Manipulation. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '94, pages 71–80. ACM, New York, NY, USA, 1994. ISBN 0-89791-657-3.
URL http://dx.doi.org/10.1145/192426.192454

Krishna A. Bharat and Luca Cardelli. Migratory Applications. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '95, pages 132–142. ACM, New York, NY, USA, 1995. ISBN 0-89791-709-X.
URL http://dx.doi.org/10.1145/215585.215711

Jacob T. Biehl and Brian P. Bailey. ARIS: An Interface for Application Relocation in an Interactive Space. In *Proceedings of Graphics Interface*, GI '04, pages 107–116. Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. ISBN 1-56881-227-2.
URL http://dx.doi.org/10.1145/642611.642666

Jacob T. Biehl, William T. Baker, Brian P. Bailey, Desney S. Tan, Kori M. Inkpen, and Mary 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. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-011-1.
URL http://dx.doi.org/10.1145/1357054.1357200

BITKOM. IT-Nutzung an deutschen Arbeitsplätzen nur noch Mittelmaß. 2011.
URL http://www.bitkom.org/72889_72885.aspx

BITKOM. Tablet Computer verbreiten sich rasant. 2012.
URL http://www.bitkom.org/de/presse/64050_70631.aspx

Marco Blumendorf, Dirk Roscher, and Sahin Albayrak. Dynamic User Interface Distribution for Flexible Multimodal Interaction. In *International Conference on Multimodal Interfaces and the Workshop on Machine Learning for Multimodal Interaction*, ICMI-MLMI '10, pages 20:1–20:8. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0414-6.
URL http://dx.doi.org/10.1145/1891903.1891930

Sören Busch. *Nomadic Interfaces in UbiComp*. Diploma thesis, RWTH Aachen University, 2011.
URL http://hci.rwth-aachen.de/busch

Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 286–297. ACM, New York, NY, USA, 1995. ISBN 0-89791-692-1.
URL http://dx.doi.org/10.1145/199448.199516

Tsung-Hsiang Chang and Yang 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. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0228-9.
URL http://dx.doi.org/10.1145/1978942.1979257

Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '11, pages 245–256. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0716-1.
URL http://dx.doi.org/10.1145/2047196.2047228

comScore. MobiLens. 2012.
URL http://de.statista.com/statistik/daten/studie/219258

Joëlle Coutaz. User Interface Plasticity: Model Driven Engineering to the Limit! In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 1–8. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0083-4.
URL http://dx.doi.org/10.1145/1822018.1822019

David Dearman and Jeffery 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. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-011-1.
URL http://dx.doi.org/10.1145/1357054.1357177

Jonathan Diehl and Jan Borchers. Tangible Windows. Technical report, RWTH Aachen University, 2013.
URL http://hci.rwth-aachen.de/diehl

Morgan Dixon, Daniel Leventhal, and James Fogarty. Content and Hierarchy in Pixel-based Methods for Reverse Engineering Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 969–978. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0228-9.
URL http://dx.doi.org/10.1145/1978942.1979086

James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '11, pages 225–234. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0716-1.
URL http://dx.doi.org/10.1145/2047196.2047226

W. Keith Edwards, Mark W. Newman, Jana Sedivy, Trevor Smith, and Shahram Izadi. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 279–286. ACM, New York, NY, USA, 2002. ISBN 1-58113-486-X.
URL http://dx.doi.org/10.1145/570645.570680

C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407. ACM, New York, NY, USA, 1989. ISBN 0-89791-317-5.
URL http://dx.doi.org/10.1145/67544.66963

Mario Fraikin. *Collaborating with Tangible Windows - Idea Generation and Information Exchange in Small Groups*. Diploma thesis, RWTH Aachen University, 2011.
URL http://hci.rwth-aachen.de/fraikin

Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. On-demand Cross-device Interface Components Migration. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, pages 299–308. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-835-3.
URL http://dx.doi.org/10.1145/1851600.1851653

Giuseppe Ghiani, Fabio Paternò, and Carmen 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. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1287-5.
URL http://dx.doi.org/10.1145/2254556.2254563

Tony Gjerlufsen, Clemens Nylandsted Klokmose, James Eagan, Clément Pillias, and Michel 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. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0228-9.
URL http://dx.doi.org/10.1145/1978942.1979446

Carl Gutwin, Saul Greenberg, and Mark Roseman. Workspace Awareness Support With Radar Views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '96, pages 210–211. ACM, New York, NY, USA, 1996. ISBN 0-89791-832-0.
URL http://dx.doi.org/10.1145/257089.257286

Sandra G. Hart and Lowell E. Staveland. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, pages 239–250. North Holland Press., Amsterdam, 1988. ISBN 978-0-444-70388-0.

Mountaz Hascoet. Throwing Models for Large Displays. In *Human Computer Interaction*, HCI '03, pages 77–108. British HCI Group, Bath, UK, 2003.

Ken Hinckley. Synchronous Gestures for Multiple Persons and Computers. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '03, pages 149–158. ACM, New York, NY, USA, 2003. ISBN 1-58113-636-6.
URL http://dx.doi.org/10.1145/964696.964713

David Holman, Roel Vertegaal, Mark Altosaar, Canada Kl, Nikolaus Troje, and Derek Johns. PaperWindows: Interaction Techniques for Digital Paper. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 591–599. 2005. ISBN 1-58113-998-5.
URL http://dx.doi.org/10.1145/1054972.1055054

Brad Johanson and Armando Fox. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, WMCSA '02, page 83. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1647-5.
URL http://dx.doi.org/10.1109/MCSA.2002.1017488

Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *Pervasive Computing*, 1:67–74, 2002a. ISSN 1536-1268.
URL http://dx.doi.org/10.1109/MPRV.2002.1012339

Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '02, pages 227–234. ACM, New York, NY, USA, 2002b. ISBN 1-58113-488-6.
URL http://dx.doi.org/10.1145/571985.572019

Amy K. Karlson, Shamsi T. Iqbal, Brian Meyers, Gonzalo Ramos, Kathy Lee, and John 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. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-929-9.
URL http://dx.doi.org/10.1145/1753326.1753631

Amy K. Karlson, Brian R. Meyers, Andy Jacobs, Paul Johns, and Shaun K. Kane. Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker. In *Proceedings of the International Conference on Pervasive Computing*, Pervasive '09, pages 398–405. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01515-1.
URL http://dx.doi.org/10.1007/978-3-642-01516-8_27

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP. Springer-Verlag, 1997. ISBN 978-3-540-63089-0.
URL http://dx.doi.org/10.1007/BFb0053381

Clemens Nylandsted Klokmose and Michel 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. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-246-7.
URL http://dx.doi.org/10.1145/1518701.1518833

Shin'ichi Konomi, Christian Müller-Tomfelde, and Norbert A. Streitz. Passage: Physical Transportation of Digital Information in Cooperative Buildings. In *Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, CoBuild '99, pages 45–54. Springer-Verlag, London, UK, UK, 1999. ISBN 3-540-66596-X.
URL http://dx.doi.org/10.1007/10705432_5

Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '09, pages 101–110. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-587-1.
URL http://dx.doi.org/10.1145/1551609.1551630

N. Marquardt, T. Ballendat, S. Boring, S. Greenberg, and K. Hinckley. Gradual Engagement between Digital Devices as a Function of Proximity: From Awareness to Progressive Reveal to Information Transfer. In *Proceedings of Interactive Tabletops and Surfaces*, ITS '12. ACM, 2012a. ISBN 978-1-4503-1209-7.
URL http://dx.doi.org/10.1145/2396636.2396642

Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '11, pages 315–326. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0716-1.
URL http://dx.doi.org/10.1145/2047196.2047238

Nicolai Marquardt, Ken Hinckley, and Saul Greenberg. Cross-Device Interaction via Micro-mobility and F-formations. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '12, pages 13–22. ACM, New York, NY, USA, 2012b. ISBN 978-1-4503-1580-7.
URL http://dx.doi.org/10.1145/2380116.2380121

Cathy Marshall and John C. Tang. That Syncing Feeling: Early User Experiences with the Cloud. In *Proceedings of the Designing Interactive Systems Conference*, DIS '12, pages 544–553. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1210-3.
URL http://dx.doi.org/10.1145/2317956.2318038

Jérémie Melchior, Donatien Grolaux, Jean Vanderdonckt, and Peter Van Roy. A Toolkit for Peer-to-peer Distributed User Interfaces: Concepts, Implementation, and Applications. In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 69–78. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-600-7.
URL http://dx.doi.org/10.1145/1570433.1570449

Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy. A Model-based Approach for Distributed User Interfaces. In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 11–20. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0670-6.
URL http://dx.doi.org/10.1145/1996461.1996488

Giulio Mori, Fabio Paternò, and Carmen Santoro. Tool Support for Designing Nomadic Applications. In *Proceedings of the International Conference on Intelligent User Interfaces*, IUI '03, pages 141–148. ACM, New York, NY, USA, 2003. ISBN 1-58113-586-6.
URL http://dx.doi.org/10.1145/604045.604069

Brad A. Myers. Using Handhelds and PCs Together. *Communications of the ACM*, 44(11):34–41, 2001. ISSN 0001-0782.
URL http://dx.doi.org/10.1145/384150.384159

Miguel Nacenta, Carl Gutwin, Dzimitri Aliakseyeu, and Sriram Subramanian. There and Back again: Cross-Display Object Movement in Multi-Display Environments. *Human-Computer Interaction*, 24(1):170–229, 2009. ISSN 0737-0024.
URL http://dx.doi.org/10.1080/07370020902819882

Miguel A. Nacenta, Dzmitry Aliakseyeu, Sriram Subramanian, and Carl Gutwin. A Comparison of Techniques for Multi-display Reaching. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 371–380. ACM, New York, NY, USA, 2005. ISBN 1-58113-998-5.
URL http://dx.doi.org/10.1145/1054972.1055024

Miguel A. Nacenta, Samer Sallam, Bernard Champoux, Sriram Subramanian, and Carl Gutwin. Perspective Cursor: Perspective-based Interaction for Multi-display Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 289–298. ACM, New York, NY, USA, 2006. ISBN 1-59593-372-7.
URL http://dx.doi.org/10.1145/1124772.1124817

Tatsuo Nakajima. How to Reuse Exisiting Interactive Applications in Ubiquitous Computing Environments? In *Proceedings of the Symposium on Applied Computing*, SAC '06, pages 1127–1133. ACM, New York, NY, USA, 2006. ISBN 1-59593-108-2.
URL http://dx.doi.org/10.1145/1141277.1141546

Ahsan Nazir Sheikh. *NoteCarrier: A Nomadic Application for Bi-Directional Class-Room Communication.* Master's thesis, RWTH Aachen University, 2012.
URL http://hci.rwth-aachen.de/asheikh

Donald A. Norman. Cognitive Engineering. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design: New Perspectives on Human-computer Interaction*. CRC Press, 1986. ISBN 978-0898598728.

Antti Oulasvirta and Lauri 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. ACM, New

York, NY, USA, 2007. ISBN 978-1-59593-593-9.
URL http://dx.doi.org/10.1145/1240624.1240795

J. Karen Parker, Regan L. Mandryk, and Kori M. Inkpen. TractorBeam: Seamless integration of local and remote pointing for tabletop displays. In *Proceedings of Graphics Interface*, GI '05, pages 33–40. Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. ISBN 1-56881-265-5.

Fabio Paternò and Carmen Santoro. A Logical Framework for Multi-device User Interfaces. In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 45–50. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1168-7.
URL http://dx.doi.org/10.1145/2305484.2305494

Stefan Plücken. *Übertragung von android- und windowsbasierten Applikationen zwischen verschiedenen Geräten*. Diploma thesis, RWTH Aachen University, 2012.
URL http://learntech.rwth-aachen.de/Pluecken

Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceedings of the International Conference on Ubiquitous Computing*, UbiComp '01, pages 56–75. Springer-Verlag, London, UK, UK, 2001. ISBN 3-540-42614-0.
URL http://dx.doi.org/10.1007/3-540-45427-6_7

Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974. ISSN 0001-0782.
URL http://dx.doi.org/10.1145/361011.361073

Adrian Reetz, Carl Gutwin, Tadeusz Stach, Miguel Nacenta, and Sriram Subramanian. Superflick: a Natural and Efficient Technique for Long-Distance Object Placement on Digital Tables. In *Proceedings of Graphics Interface*, GI '06, pages 163–170. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 2006. ISBN 1-56881-308-2.

Jun Rekimoto. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '97, pages 31–39. ACM, New York, NY, USA, 1997. ISBN 0-89791-881-9.
URL http://dx.doi.org/10.1145/263407.263505

Jun Rekimoto, Yuji Ayatsuka, and Michimune Kohno. SyncTap: An Interaction Technique for Mobile Networking. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '03, pages 104–115. Springer, 2003. ISBN 978-3-540-40821-5.
URL http://dx.doi.org/10.1007/978-3-540-45233-1_9

Jun Rekimoto and Masanori Saitoh. Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments. In *Proceedings of the SIGCHI*

Conference on Human Factors in Computing Systems, CHI '99, pages 378–385. ACM, New York, NY, USA, 1999. ISBN 0-201-48559-1.
URL http://dx.doi.org/10.1145/302979.303113

Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *Internet Computing*, 2(1):33 –38, 1998. ISSN 1089-7801.
URL http://dx.doi.org/10.1109/4236.656066

Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5:79–109, 1986. ISSN 0730-0301.
URL http://dx.doi.org/10.1145/22949.24053

Dominik Schmidt, Fadi Chehimi, Enrico Rukzio, and Hans Gellersen. PhoneTouch: A Technique for Direct Phone Interaction on Surfaces. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '10, pages 13–16. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0271-5.
URL http://dx.doi.org/10.1145/1866029.1866034

B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69, 1983. ISSN 0018-9162.
URL http://dx.doi.org/10.1109/MC.1983.1654471

James E. Smith and Ravi Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005. ISSN 0018-9162.
URL http://dx.doi.org/10.1109/MC.2005.173

Timothy Sohn, Frank Chun Yat Li, Agathe Battestini, Vidya Setlur, Koichi Mori, and Hiroshi Horii. Myngle: Unifying and Filtering Web Content For Unplanned Access Between Multiple Personal Devices. In *Proceedings of the International Conference on Ubiquitous computing*, UbiComp '11, pages 257–266. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0630-0.
URL http://dx.doi.org/10.1145/2030112.2030147

Statistisches Bundesamt. Private Haushalte in der Informationsgesellschaft (IKT). 2012. Fachserie 15 Reihe 4.

Norbert A. Streitz, Jörg Geißler, Torsten Holmer, Shin'ichi Konomi, Christian Müller-Tomfelde, Wolfgang Reischl, Petra Rexroth, Peter Seitz, and Ralf 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. ACM, New York, NY, USA, 1999. ISBN 0-201-48559-1.
URL http://dx.doi.org/10.1145/302979.303010

Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User Interface Façades: Towards Fully Adaptable User Interfaces. In *Proceedings of the Annual Symposium on User Interface Software and Technology*, UIST '06, pages 309–318. ACM, New York, NY, USA, 2006. ISBN 1-59593-313-1.
URL http://dx.doi.org/10.1145/1166253.1166301

Takashi Suezawa. Persistent Execution State of a Java Virtual Machine. In *Proceedings of the Conference on Java Grande*, JAVA '00, pages 160–167. ACM, New York, NY, USA, 2000. ISBN 1-58113-288-3.
URL http://dx.doi.org/10.1145/337449.337536

Desney S. Tan, Brian Meyers, and Mary Czerwinski. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 1525–1528. ACM, New York, NY, USA, 2004. ISBN 1-58113-703-6.
URL http://dx.doi.org/10.1145/985921.986106

Peter Tandler. The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments. *Journal of Systems and Software*, 69(3):267–296, 2004. ISSN 0164-1212.
URL http://dx.doi.org/10.1016/S0164-1212(03)00055-4

Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java Programs with Distribution Capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1):1:1–1:40, 2009. ISSN 1049-331X.
URL http://dx.doi.org/10.1145/1555392.1555394

Peter van Roy, editor. *Proceedings of The Second International Conference on Multiparadigm Programming in Mozart/Oz*. MOZ '04. Springer-Verlag, Secaucus, NJ, USA, 2004. ISBN 978-3-540-25079-1.
URL dx.doi.org/10.1007/b106627

Chris Vandervelpen and Karin Coninx. Towards Model-Based Design Support for Distributed User Interfaces. In *Proceedings of the Nordic Conference on Human-computer interaction*, NordiCHI '04, pages 61–70. ACM, New York, NY, USA, 2004. ISBN 1-58113-857-1.
URL http://dx.doi.org/10.1145/1028014.1028023

S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine*, 35(2):46 –55, 1997. ISSN 0163-6804.
URL http://dx.doi.org/10.1109/35.565655

Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991.
URL http://dx.doi.org/10.1145/329124.329126

Andrew D. Wilson and Raman Sarin. BlueTable: Connecting Wireless Mobile Devices on Interactive Surfaces Using Vision-based Handshaking. In *Proceedings of Graphics Interface*, GI '07, pages 119–125. ACM, New York, NY, USA, 2007. ISBN 978-1-56881-337-0.
URL http://dx.doi.org/10.1145/1268517.1268539

Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, COOTS'96, pages 17–17. USENIX Association, Berkeley, CA, USA,

1996.
URL https://www.usenix.org/legacy/publications/library/proceedings/coots96/wollrath.html

Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging Single-user Applications for Multi-user Collaboration: The Coword Approach. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW '04, pages 162–171. ACM, New York, NY, USA, 2004. ISBN 1-58113-810-5. URL http://dx.doi.org/10.1145/1031607.1031635

Ying Zhang. *A Classification of Interaction Styles that Span Multiple Systems*. Master's thesis, RWTH Aachen University, 2012.
URL http://hci.rwth-aachen.de/zhang

Index

Curriculum Vitae

Personal Data	Jonathan Diehl Media Computing Group RWTH Aachen University
Email	diehl@cs.rwth-aachen.de
24. Mai 1980	Born in Kiel, Germany
Oct 2001 – Sept 2006	Diploma in Computer Science at RWTH Aachen University, Germany
Aug 2007 – Nov 2013	Doctoral Candidate at the Media Computing Group, Department of Computer Science, RWTH AAchen University, Germany Advisor: Prof. Dr. Jan Borchers

Publications of the Author

2013

Jonathan Diehl and Jan Borchers. **Tangible Windows**. Technical report, RWTH Aachen University, June 2013.

2012

Can Liu, Stéphane Huot, Jonathan Diehl, Wendy E. Mackay, and Michel Beaudouin-Lafon. **Evaluating the Benefits of Real-time Feedback in Mobile Augmented Reality with Hand-held Devices**. In *Proceedings of CHI '12*, Austin, TX, USA, May 2012. ACM Press.

2011

Jonathan Diehl and Jan Borchers. **Mobile HCI and Hospitality**. In *Mobile HCI '11 Workshop on Mobile Interaction in Retail Environments*, Stockholm, Sweden, September 2011. ACM Press.

Can Liu, Jonathan Diehl, Stéphane Huot, and Jan Borchers. **Mobile Augmented Note-taking to Support Operating Physical Devices**. In *Mobile HCI '11 Workshop on Mobile Augmented Reality*, Stockholm, Schweden, September 2011. ACM Press.

Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. **Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency**. In *Proceedings of UIST '11*, Santa Barbara, CA, USA, October 2011. ACM Press.

2010

Bernd Theiss, Jan Borchers, Markus Jordans, and Jonathan Diehl. **Gute Bedienung**. *connect*, 11:12-17, November 2010. Weka Media Publishing.

Jan-Peter Krämer, Thorsten Karrer, Jonathan Diehl, and Jan Borchers. **Stacksplorer: Understanding Dynamic Program Behavior**. In *Extended Abstracts of UIST '10*, New York, NY, October 2010. ACM Press.

Chatchavan Wacharamanotham, Jonathan Meyer, Jonathan Diehl, and Jan Borchers. **The Interactive Bracelet: An input device for bimanual interaction**. In *Mobile HCI '10 Workshop on Ensembles of On-Body Devices*, Lisbon, Portugal, September 2010. ACM Press.

Jonathan Diehl, Thorsten Karrer, and Jan Borchers. **Interactive System Architecture for Layered Applications**. In *Interactive System Architecture Workshop*, September 2010.

2009

Jonathan Diehl. **Associative Personal Information Management**. In *Extended Abstracts of CHI '09*, Boston, MA, USA, 2009. ACM Press.

Max Möllers, Jonathan Diehl, Markus Jordans, and Jan Borchers. **Towards Systematic Usability Verification**. In *Extended Abstracts of CHI '09*, Boston, MA, USA, 2009. ACM Press.

2008

Jonathan Diehl and Jan Borchers. **Associative Information Spaces**. In *Extended Abstracts of CSCW '08*, San Diego, CA, USA, November 2008.

Jonathan Diehl, Max Möllers, and Jan Borchers. **Improving List Selection Performance with Pressure-Sensitivity on a Scroll Ring**. In *Extended Abstracts of UIST '08*, Monterey, CA, USA, October 2008.

Jonathan Diehl, Jan-Peter Krämer, and Jan Borchers. **A Framework for using the iPhone as a Wireless Input Device for Interactive Systems**. In *Extended Abstracts of UIST '08*, Monterey, CA, USA, October 2008.

Mei-Fang Liau, Jonathan Diehl, and Jan Borchers. **DISCO: Disk-based Interface for Semantic Composition**. In Ulrike Lucke, Martin Christof Kindsmüller, Stefan Fischer, Michael Herczeg, and Silke S, editors, *Workshop Proceedings der Tagungen Mensch & Computer 2008, DeLFI 2008 und Cognitive Design 2008*. Lübeck, Germany, 2008. Logos Verlag.

Jonathan Diehl, Deniz Atak, and Jan Borchers. **Associative Information Spaces**. In Niels Henze, Gregor Broll, Enrico Rukzio, Michael Rohs, and Andreas Zimmermann andd Susanne Boll, editors, *Mobile HCI '08 Workshop on Mobile Interaction with the Real World*, Amsterdam, Netherlands, 2008. BIS-Verlag.

Eileen Falke, Jonathan Diehl, and Jan Borchers. **The Associative PDA 2.0**. In *Extended Abstracts of CHI '08*, Florence, Italy, April 2008. ACM Press.

2007

Bernd Theiss, Markus Eckstein, Jan Borchers, Jonathan Diehl, and Markus Jordans. **Systemkritik: Der Handy-Bedientest**. *connect*, 9:14-23, September 2007. Weka Media Publishing.

2006

Jonathan Diehl. **The Associative PDA - An Organic User Interface for Mobile Personal Information Management**. Diploma thesis, RWTH Aachen University, 2006.

Jonathan Diehl and Jan Borchers. **The Associative PDA: An organic user interface for mobile**. In Extended Abstracts of Ubicomp '06, Newport, CA, USA, September 2006. ACM Press.

Typeset: 1st December, 2013