University of Konstanz
Department of Computer and Information Science
Master's Degree Course Information Engineering

Master Thesis

A STATE MACHINE FRAMEWORK FOR POST-WIMP INTERACTION DESIGN

for the degree
Master of Science (M.Sc.)

Subject of Study:	Information Engineering
Specialization:	Computer Science
Topic:	Applied Informatics

by
Michael Zöllner
(01/622989)

1st Referee:	Prof. Dr. Harald Reiterer
2nd Referee:	Prof. Dr. Marc H. Scholl

Submission Date: April, 10th, 2012

Abstract

After years of dominance, classic desktop-based WIMP (Windows, Icons, Menus, Pointer) systems are slowly being replaced by modern post-WIMP systems. Such systems do not stick to a certain user interface or interaction paradigm, but rather contain a heterogeneous set of characteristics that stem from multiple fields of research. These characteristics induce a variety of different challenges that designers and developers of post-WIMP systems have to face and overcome. In this thesis, it is argued that formal methods, and in particular finite-state machines, are an important means to tackle certain of these challenges. In order to adapt finite-state machines to the requirements of post-WIMP systems and to improve their expressivity, specific additions to their default notation are suggested. One such addition allows the specification of animated transitions. The other addition is a notation to differentiate multiple input points. Although there is a consensus between many researchers and developers that finite-state machines are a rather natural formalism for the specification of complex interactive systems, their implementation is not yet supported appropriately by user interface toolkits or programming languages. Thus, a finite-state machine framework for post-WIMP interaction design, the Reactive State Machine framework, is presented in the main part of this thesis. Due to its declarative nature, it greatly facilitates the transformation of a graphical state machine model into code. The Reactive State Machine framework supports all important state machine concepts, such as states and transitions. What sets it apart from other similar frameworks is its full support for input events, its support for animated transitions and its support for the multi-point notation that is introduced in this thesis. To show the utility and value of the Reactive State Machine Framework its application in three assorted use cases is demonstrated. For one of the use cases, the Facet-Streams system, a comparison is conducted between the old naive implementation based on low-level implementation techniques and the revised implementation based on the Reactive State Machine framework. Finally, the threshold and ceiling of the Reactive State Machine framework are assessed in a brief informal evaluation. To conclude the thesis, its main contributions are summarized and an outlook on potential future work is given.

Zusammenfassung

Nach Jahren der Dominanz werden klassische desktop-basierte WIMP (Windows, Icons, Menus, Pointer) Systeme allmählich von modernen post-WIMP Systemen verdrängt. Diese Systeme basieren nicht auf einem bestimmten Benutzerschnittstellen- oder Interaktionsparadigma, sondern auf sehr heterogenen Charakteristika die aus vielen verschiedenen Forschungsfeldern stammen. Diese Charakteristika verursachen eine Vielzahl verschiedener Herausforderungen denen sich Designer und Entwickler von post-WIMP Systemen stellen müssen. In dieser Arbeit werden Gründe angeführt wieso formale Methoden, und im Speziellen endliche Automaten, ein geeignetes Mittel sind einige dieser Herausforderungen anzugehen. Um endliche Automaten an die Anforderungen von post-WIMP Systemen anzupassen und um ihre Expressivität zu verbessern, werden Ergänzungen zu ihrer Standardnotation vorgeschlagen. Eine solche Ergänzung erlaubt es animierte Übergange zu definieren. Die andere Ergänzung ist eine Notation, die die Differenzierung verschiedener Eingabepunkte erlaubt. Obwohl unter Forschern und Entwicklern die übereinstimmende Meinung herrscht, dass endliche Automaten ein ziemlich natürlicher Formalismus sind um komplexe interaktive Systeme zu spezifizieren, wird ihre Entwicklung von heutigen Entwicklungswerkzeugen und Programmiersprachen noch nicht ausreichend unterstützt. Darum wird im Hauptteil dieser Arbeit ein Framework zur Erstellung von endlichen Automaten präsentiert (das Reactive State Machine Framework). Auf Grund seiner deklarativen Natur, erleichtert es die Transformation des graphischen Modells eines endlichen Automaten in den Code außerordentlich. Das Reactive State Machine Framework unterstützt alle wichtigen Konzepte von endlichen Automaten, wie zum Beispiel Zustände und Übergänge. Es setzt sich ab von anderen ähnlichen Frameworks durch seine umfassende Unterstützung für Input Events, seine Unterstützung für animierte Übergange und seine Unterstützung der Multi-Point Notation, die in dieser Arbeit vorgestellt wird. Um die Nützlichkeit und den Wert des Reactive State Machine Frameworks zu zeigen, wird seine Anwendung in drei unterschiedlichen Anwendungsfällen demonstriert. Für einen der Anwendungsfälle, das Facet-Streams System, wird eine Gegenüberstellung durchgeführt zwischen der alten, naiven Umsetzung die mit konventionellen Mitteln erstellt wurde und der überarbeiteten Umsetzung die mit dem Reactive State Machine Framework erstellt wurde. Danach werden *Threshold* und *Ceiling* des Reactive State Machine Frameworks mittels einer informellen Evaluation beurteilt. Zum Abschluss der Arbeit werden ihre Hauptbeiträge zusammen gefasst und ein Ausblick auf potenzielle weitere Arbeiten gegeben.

Foreword

This thesis presents a state machine framework for post-WIMP interaction design. While I was first skeptical regarding the use of finite-state machines in the context of interaction design, I have been convinced of their strenghts by my advisor *Hans-Christian Jetter* and meanwhile turned into a strong advocate. The main motivation for the subject of this thesis stems from a research project that has been conducted by the Human-Computer Interaction Group of the University of Konstanz in cooperation with Microsoft Research Cambridge. I want to say *thank you* to *Prof. Harald Reiterer* and my advisor *Hans-Christian Jetter*, who gave me the opportunity to participate in this project and who made it possible that I could work on it in Cambridge for a couple of weeks. The outcome of this research project, the Facet-Streams system, was successfully published at the 2011 CHI conference in Vancouver. I am very thankful that I have been given the opportunity to visit this conference together with the other authors.

After 3 1/2 years of work in the HCI Group of *Prof. Reiterer*, I want to take this opportunity and say *thanks* to all members of this group who crossed my path. I had the opportunity to work with you on a variety of fascinating research projects and learned a lot during this time. Special thanks go out to my advisor *Hans-Christian Jetter* for his excellent support during the last 4 years, the research opportunities he provided me and all the inspiring discussions.

Last but not least I want to thank my family for their never-ending support during all the years of study and especially during the last two months of writing.

Contents

I Introduction **1**

II Post-WIMP Systems **5**

1 Characteristics of Post-WIMP Systems **6**
1.1 Devices . . . 9
1.1.1 Natural Interaction . . . 10
1.1.2 Direct Interaction . . . 11
1.1.3 Concurrent Interaction . . . 13
1.1.4 Tangible Interaction . . . 15
1.2 Graphics . . . 16
1.2.1 Flexible Layouts . . . 16
1.2.2 Flexible Scale . . . 17
1.2.3 Physical Behavior . . . 17
1.2.4 Animations . . . 18
1.3 Users . . . 19

2 Describing and Developing Post-WIMP Systems **21**
2.1 Challenges of Post-WIMP Systems . . . 21
2.1.1 Concurrent Interaction with Multi-Modal and Multi-Point Input . . . 22
2.1.2 Multi-User Interaction . . . 24
2.1.3 Discrete vs. Continuous Input . . . 26
2.1.4 Ambiguous Input . . . 27
2.2 Shortcomings of Development Tools . . . 29
2.2.1 Expressing Sequential Interaction . . . 29
2.2.2 Expressing Parallel Interaction . . . 30
2.2.3 Expressing Timing Contraints . . . 31
2.2.4 Summary . . . 32
2.3 Formal Methods . . . 33
2.3.1 Advantages of Formal Methods . . . 34
2.3.2 Criticism . . . 35
2.4 Finite-State Machines . . . 36
2.4.1 States in the User Interface . . . 37
2.4.2 States in the Interaction . . . 37

III Finite State Machines **41**

3 Features and Notation **42**
3.1 Modeling States . . . 42
3.2 Modeling Transitions . . . 43
3.3 Modeling Animations . . . 45
3.4 Modeling Multiple Input Points . . . 48
3.5 Omitted Statecharts Concepts . . . 51

4 Implementation **54**
4.1 Comparison of State Machine Frameworks . . . 57
4.1.1 Issues of Triggered Transitions . . . 58
4.1.2 Support for Animations . . . 63
4.1.3 Conclusions . . . 63

IV Required Libraries **64**

5 Reactive Extensions **65**
5.1 The Unified Programming Model . . . 65
5.1.1 Observable Collections . . . 66
5.1.2 Lifetime Phases . . . 67
5.1.3 Composing and Coordinating Observables . . . 69
5.1.4 Visualizing Observables . . . 70
5.2 Usage . . . 71
5.2.1 Subscribing to Observables . . . 71
5.2.2 Creating Observables . . . 73
5.2.3 Conversion Operators . . . 74
5.2.4 Filter Operators . . . 74
5.2.5 Projection Operators . . . 75
5.2.6 Time-based Operators . . . 75
5.3 Examples . . . 76
5.3.1 Selecting Input Events . . . 76
5.3.2 The `ReleaseLink` Behavior . . . 78

6 The Visual State Manager **79**
6.1 The `VisualStateGroup` Class . . . 79
6.2 The `VisualState` Class . . . 80
6.3 The `VisualTransition` Class . . . 81
6.4 Controlling the Visual State Manager . . . 81
6.5 Tool Support . . . 82

V Reactive State Machine 83

7 Overview 84
7.1 State Machine Management . . . 84
7.2 States . . . 85
7.2.1 Entry & Exit Actions . . . 86
7.3 Transitions . . . 88
7.3.1 Triggered Transitions . . . 88
7.3.2 Timed Transitions . . . 90
7.3.3 Automatic Transitions . . . 90
7.4 Animations . . . 91
7.5 Tracking Input Points . . . 93

8 Architecture and Implementation 95
8.1 Main Loop . . . 95
8.2 Configuration . . . 96
8.3 Enabling/Disabling Transitions . . . 97
8.4 Transition Flow . . . 99
8.5 Execution Context of Actions and Conditions . . . 100
8.6 Exception Handling . . . 100
8.7 Extension Mechanism . . . 100
8.8 Animations . . . 101
8.9 Tracking Input Points . . . 102

9 Use Cases 103
9.1 Facet-Streams - The Wheel . . . 103
9.1.1 Lifecycle . . . 106
9.1.2 Selection Behavior . . . 107
9.1.3 Implementation . . . 109
9.2 Facet-Streams - The `ReleaseLink` Behavior . . . 116
9.3 SmartShare . . . 117
9.3.1 The Login Control . . . 118
9.3.2 Getting Data onto the Table . . . 121
9.3.3 Getting Data from the Table . . . 124
9.3.4 Summary . . . 125
9.4 Informal Evaluation . . . 126

VI Conclusion 130

10 Conclusion **131**
10.1 Contributions 131
10.1.1 Conceptual Contributions 131
10.1.2 Development Contributions 132
10.2 Future Work 132
10.2.1 Support for Statecharts Elements 133
10.2.2 Support for Graphical FSM Models 133

References **144**

List of Figures **147**

List of Tables **147**

List of Listings **149**

A Appendix: Reactive Extensions **150**
A.1 Dualizing `IEnumerable<T>/IEnumerator<T>` 150
A.2 Additional Reactive Extension Operators 155
A.2.1 Additional Conversion Operators 155
A.2.2 Additional Filter Operators 156
A.2.3 Additional Projection Operators 156
A.2.4 Additional Time-based Operators 158
A.2.5 Aggregation Operators 158
A.2.6 Grouping and Windowing Operators 159
A.2.7 Coordination Operators 160
A.2.8 Other Operators 161

Part I

Introduction

> The good news about computers is that they do what you tell them to do.
> The bad news is that they do what you tell them to do
>
> Ted Nelson

Since the early days of computing the ways humans interact with computers have changed significantly. In the first decades, human-computer interaction went from plugging and soldering electrical circuits to encoding punch cards to textual command-line interfaces [Dourish, 2001]. Then, in the early 1980's, the first commercial graphical user interface (GUI), the Xerox Star [Smith et al., 1982], completely changed the computing landscape: It featured the WIMP (**W**indows, **I**cons, **M**enus, **P**ointer) paradigm as a way to generalize interaction with the machine, and the desktop metaphor as a means to facilitate access to computers for novices. Without any doubt, these two UI concepts turned out to be a major success. Even today, all major PC operating systems (Windows, OS X, Linux) still ship with graphical user interfaces based on WIMP and the desktop metaphor.

All classic WIMP user interfaces are based on a small set of defining principles: By using a ***P**ointing device* a user can select ***I**cons* from the desktop or the file system. These icons are associated to specific applications which are represented by ***W**indows*. Inside these application windows, objects can be created, deleted or edited which is usually performed by invoking actions from a ***M**enu* or similar graphical widgets.

While the interaction with a WIMP user interface is in many ways different from the interaction with a classic command line interface, the basic concept of both is the same: "They were all based on [...] an explicit dialogue between the user and the computer during which the user commanded the computer to do something" [Nielsen, 1993]. This is different to the way we mostly interact in the physical world where the object of interest can be manipulated directly with our hands without having any mediating proxy. Shneiderman tried to introduce such a more physical kind of interaction with his principle of *direct manipulation*. Direct manipulation advocates the use of "rapid, incremental, reversible operations whose impact on the object of interest is immediately visible" combined with a "continuous representation of the object of interest" [Shneiderman, 1983]. Although direct manipulation has been stated in the early 1980s, it could not influence too many of the interaction principles of WIMP user interfaces, *Drag & Drop* being a notable exception.

While the WIMP paradigm and the desktop metaphor evolved steadily and eventually became the de-facto standard in the desktop computing world, they have also been subject to criticism by researchers for various reasons early on (see for example [Nielsen, 1993; Gentner and Nielsen, 1996; van Dam, 1997] or [Kaptelinin and Czerwinski, 2007]). Some of these reasons have been

resolved over time thanks to principles such as direct manipulation, while others have been tenacious. Take for example the strict and literal usage of metaphors to bridge the gap of real world and computer world. Although criticized by many (e.g. [Nelson, 1990]), metaphors were a crucial part of the Macintosh Design Guidelines [Apple Computer, 1992] and kept coming back at regular intervals (remember Magic Cap[1] or Microsoft's Bob[2] ?). Unfortunately, the most fundamental points of criticism still apply today: WIMP user interfaces still do not allow more than one user interacting with the system and they leverage only a limited number of our motoric and sensory capabilities: "WIMP GUIs based on the keyboard and the mouse are the perfect interface only for creatures with a single eye, one or more single-jointed fingers, and no other sensory organs" - Bill Buxton via [van Dam, 1997]. This is also illustrated in figure 1 which shows a caricature of how we as users are seen by a WIMP system.

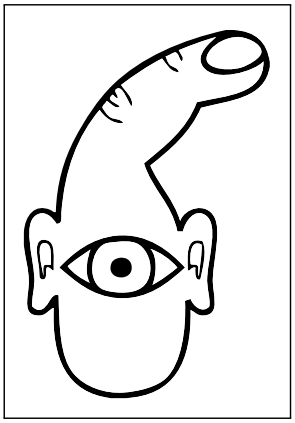

Figure 1: *How the computer sees us* [O'Sullivan and Igoe, 2004]

The first organized approach to tackle these problems was a workshop conducted at SIGGRAPH in 1990. There, several experts gathered to discuss so-called "non-WIMP" user interfaces, a term that refers to all user interfaces that do not stick to the WIMP paradigm and the desktop metaphor [Green and Jacob, 1991]. This workshop was not only motivated by the various points of criticism against those two, but also by the fact that the success of the desktop metaphor "has blinded user interface designers and researchers to other interaction styles", leaving the community with the belief that "a desktop direct manipulation user interface is the best style of user interface for all applications" [Green and Jacob, 1991]. Or, as Andries van Dam put it a few years later: "They are apparently sufficiently good for conventional desktop tasks that the field is stuck comfortably in a rut" [van Dam, 1997]. By specifying the characteristics of "non-WIMP" user interfaces, their soft- and hardware requirements and future research topics for the community, the members of the workshop laid the ground for many researchers thereafter and raised awareness to the fact that WIMP GUIs are not carved in stone.

[1] http://en.wikipedia.org/wiki/Magic_Cap
[2] http://en.wikipedia.org/wiki/Microsoft_Bob

In addition to research on specific "non-WIMP" user interfaces such as virtual reality systems or 3D user interfaces, there were various attempts at specifying more detailed what it is (or will be) that constitutes user interfaces that do not adhere to the WIMP paradigm. In 1993, Jakob Nielsen defined 12 dimensions "along which next-generation UIs may differ from previous generations of UIs" [Nielsen, 1993]. As the term "may" suggests, many of the dimensions that he envisioned for the next generation of user interfaces, were clever speculations based on other research outcomes of that time. A fair number of his assumptions became reality by now such as the input device explosion, the increase of bandwidth between the user and the interface, and the ubiquity of interfaces, while others are still on the agenda of today's research such as the interpretation of user actions to let the system do "what it deems appropriate". In 1996, Don Gentner and Jakob Nielsen came up with the "Anti-Mac User Interface" [Gentner and Nielsen, 1996], a set of user interface principles that provide an alternative to the Macintosh principles, and in a 1997 article, Andries van Dam coined the term "post-WIMP" user interfaces to denote all those interfaces whose concepts go beyond the classic WIMP paradigm [van Dam, 1997]. A commonality of all those attempts is that they relate themselves to the existing WIMP paradigm in a negative, destructive, and unspecific way. Describing a system as being "non-WIMP", "Anti-Mac" or "post-WIMP" is definitely easier than describing its characteristics in a concise way. On the other hand, this lack of specificity also gave rise to many different research directions in the past years which further diversified the field of HCI. Nielsen envisioned such a development when he stated that there will not be a single "canonical interface style" such as the WIMP paradigm, but rather tailor-made concepts that serve the individual task at hand [Nielsen, 1993]. In the mobile sector, this vision certainly has already come true: current smartphone user interfaces differ substantially from traditional WIMP user interfaces and do successfully serve the task at hand. Yet, in the world of classic, desktop-sized computers most advancements are only in their infancy.

Regarding the last two decades of UI development it is fair to say that the actual user interface revolution failed to appear. It is in fact rather a gradual evolution. Most changes are driven by advancements of the hardware industry which were quite substantial in the last two decades. Not only do we have all sizes and types of computing devices available today, we could theoretically also choose from a huge variety of input technologies to interact with these devices. Yet in practice, the integration of these input technologies into our present day computing devices is only beginning to happen. Recent developments, mainly established by the mobile industry, are gradually fusing with the classic world. For example, the upcoming Windows 8 operating system will for the first time feature an optional alternative to the default desktop that is targeted at multi-touch interaction[3]. Reasons for this slow adoption are various, ranging from the lack of theoretical principles to the lack of design and implementation guidance in practice. It is the prevalent opinion of many practitioners (including myself) that the design and development of post-WIMP systems is a very challenging task that requires skills from many different areas of computer science.

[3] http://blogs.msdn.com/b/b8/archive/2011/08/31/designing-for-metro-style-and-the-desktop.aspx

My goal with this thesis is to address certain parts of this challenge. The thesis is thereby focused on the interactive behaviors of post-WIMP systems which are typically complex to describe and implement. As is shown below, finite-state machines can greatly facilitate the development of certain of these behaviors. To enable their implementation, the state machine framework *Reactive State Machine (RSM)* has been created. It provides a low barrier of entry for developers to integrate finite-state machines into their systems and has additional support for specific post-WIMP characteristics.

The remainder of this thesis is structured as follows:

In part II, I first want to shed some additional light on post-WIMP systems. In section 1 on page 6, their basic characteristics are discussed. In section 2 on page 21, it is then shown that the design and development of such systems involves a variety of different challenges which are not yet supported appropriately by present-day user interface toolkits and programming languages. To conclude this part, it is then shown that formal methods, and in particular Finite-State Machines (FSMs), are an appropriate means to describe and develop specific interactive behaviors of these systems.

In part III, finite-state machines are then advanced further. Section 3 on page 42 shows what features and notational elements are required to adequately model finite-state machines for post-WIMP systems. In this section, additions to the default notation are introduced to deal with animations and multi-point input, two important characteristics of post-WIMP systems. In section 4 on page 54, the implementation of finite-state machines is then discussed briefly. It is shown that current state machine frameworks are not prepared for the implementation of post-WIMP systems as they lack some important features.

In part IV, two important libraries are presented that play a central role in the Reactive State Machine framework. The Reactive Extensions (Rx) library, which is presented in section 5 on page 65, enables full event support for the triggers of the state machine, whereas the Visual State Manager (VSM), which is presented in section 6 on page 79, is leveraged to enable animated transitions.

In part V, the main part of the thesis, the Reactive State Machine (RSM) framework is then presented in detail. In section 7 on page 84, an overview is given over all features of the RSM and its public API. Then, in section 8 on page 95, the architecture and implementation decisions are discussed. In section 9 on page 103, it is shown how the RSM is leveraged in state-of-the-art post-WIMP systems to realize certain interactive behaviors. Finally, a quick informal evaluation is provided to summarize the findings of these use cases.

In the final part of this thesis, starting at page 130, conclusions are drawn and an outlook on potential future work is given.

Part II

Post-WIMP Systems

> Never trust a computer you can't throw out a window
>
> Steve Wozniak

In this part of the thesis, the nature of *post-WIMP systems* is examined detailly. First, their various characteristics are discussed to get an impression what post-WIMP systems are constituted of. The subsequent section then shows that these characteristics bring along a set of challenges which are not adequately supported by present-day development tools. Finally, the application of formal methods, and in particular finite-state machines, is suggested as a means to design and develop the interactive behaviors of post-WIMP systems.

1 Characteristics of Post-WIMP Systems

In this section, the characteristics of post-WIMP systems are presented. While these characteristics portrait post-WIMP systems from the point of view of the user, the next section points out what consequences these characteristics have for designers and developers of such systems. The reader has to keep in mind that most current post-WIMP systems only contain a subset of these characteristics. As the evolution from WIMP to post-WIMP happens only gradually, new technologies and features blend in one by one, once they reach a mature state. This development is in keeping with the definition of Andries van Dam who stated that to him a post-WIMP interface is one "containing *at least* one interaction technique not dependent on classic 2D widgets such as menus and icons" (emphasis mine) [van Dam, 1997].

Terminology and Classification The most difficult part about discussing post-WIMP systems is to find a common ground in terms of terminology and classification. The research around post-WIMP systems, post-WIMP user interfaces, and post-WIMP interaction is not in itself a dedicated field in HCI, but is instead distributed over a huge variety of different research fields such as Multi-Modal Interaction, Tangible User Interfaces (TUIs), Interactive Surfaces, Augmented Reality (AR), Virtual Reality (VR), and many more. While many researchers made up their own theories of user interface and interaction principles within these fields, it is really hard to find a general paradigm or theme of interaction that is able to span all of them. Many of the systems that stem from these fields differ significantly in their style of interaction, which makes it impossible to come up with a concise paradigm such as WIMP that could capture the essence in just four characters. To find commonalities, it is necessary to take a step back from the fine-grained interaction technique level and to consider the overall interaction of the user with these systems in a more general way.

The most important common trait of all these different research fields is that ever more properties of the human and its environment are leveraged for the interaction with a system. One term that often pops up in this context is the notion of Natural User Interfaces (NUIs). While the heritage of this term can be attributed to Steve Mann[4], it was established mainly by the NUI Group[5], an open source community whose members share their knowledge and enthusiasm about new emerging technologies such as multi-touch tabletops. Meanwhile the term has arrived in both research and industry, where it is for example used to promote the new technologies of Microsoft. Wigdor and Wixon, who devoted a complete book to the design and development of NUIs, define the term as follows: A NUI is a user interface that mirrors the users' capabilities, meets their needs, takes

[4] http://en.wikipedia.org/wiki/Natural_user_interface
[5] http://nuigroup.com

full advantage of their capacities and fits their task and context demands [Wigdor and Wixon, 2011]. While this is a rather vague definition, it spans a substantial amount of systems. NUIs are often built around new emerging technologies, such as multi-touch tabletops, digital pen & paper or depth cameras. Although these are certainly important to attain the goals that NUIs promise, Wigdor and Wixen put their importance into the correct perspective: "Input and output technologies offer us the opportunity to create a more natural user interface; they do not, in and of themselves, define or guarantee it" [Wigdor and Wixon, 2011]. Instead, they stress that "the natural element of a natural user interface is not about the interface at all. Quite the opposite. We see natural as referring to the way users interact with and feel about the product, or more precisely, what they do and how they feel while they are using it" [Wigdor and Wixon, 2011]. In the domain of Natural User Interfaces, an attempt has been undertaken to create a basic set of metaphors and principles, similar to the WIMP paradigm. After a series of blog posts, Ron George and Joshua Blake published a workshop paper about what they called OCGM (Objects, Containers, Gestures, and Manipulations) [George, 2009; George and Blake, 2010]. They proclaimed that "WIMP is to GUI as OCGM (Occam) is to NUI" [Blake, 2009]. To this day, however, OCGM did not have any wide-ranging impact on the community.

An approach that provides scientific evidence to the goals that NUIs promise is the concept of Reality Based Interaction (RBI) [Jacob et al., 2008]. Jacob et al. observed that "all of these new interaction styles draw strength by building on users' pre-existing knowledge of the everyday, non-digital world to a much greater extent than before" [Jacob et al., 2008]. They further identified four *themes of reality* which are employed by these systems: (1) The theme of *Naïve Physics (NP)* is concerned with the fact that we have implicit and intuitive knowledge of how objects physically behave in the real world. (2) The theme of *Body Awareness and Skills (BAS)* refers to the fact that a user is aware of his physical body and has skills to control and coordinate it. (3) The theme of *Environment Awareness and Skills (EAS)* is based on the fact that users are integrated in a surrounding which they constantly manipulate and in which they constantly navigate and interact. (4) And finally, the theme of *Social Awareness and Skills (SAS)* highlights the fact that users are social beings that are aware of each other and interact and communicate with each other. While being mainly an explanatory theory, the concept of RBI and its four themes of reality can also act as a starting point for new designs and interaction techniques, as it is based on broad scientific evidence, has a well-defined structure and provides enough abstraction to be used in a variety of different contexts.

Strong support for the concepts of NUI and RBI comes from a line of research in cognitive sciences. This theory of the human mind, which is called *embodiment* or *embodied cognition*, opposes the old Cartesian dualism which states that body and mind are two separate entities. Instead, it is argued that body and mind heavily influence each other and that cognitive processes can not be viewed in isolation [Dourish, 2001]. Instead of treating cognitive processes as formal operations on abstract symbols, cognition is regarded as a highly embodied or situated activity which is influenced by the

body, the environment, and the actions we make therein [Anderson, 2003]. While the field of HCI was initially built upon the concept of a Human Information Processor [Card et al., 1983, 22ff.] which totally neglected the properties of the human body and its environment, the embodiment concept slowly establishes itself in the minds of HCI researchers. This can mainly be attributed to Dourish and his concept of *embodied interaction* which is based "on the understanding that users create and communicate meaning through their interaction with the system (and with each other, through the system)" [Dourish, 2001]. Embodied interaction addresses the fact that the interaction with a system is situated in a greater context and that the physical and social skills that we have learned since early childhood can facilitate and augment it greatly. The following quote of Wellner indicates that researchers in the nineties were already quite aware of these qualities of the real life, but it is only now that we have the technological capabilities to make use of them: "We live in a complex world, filled with myriad objects, tools, toys, and people. Our lives are spent in diverse interaction with this environment. Yet, for the most part, our computing takes place sitting in front of, and staring at, a single glowing screen attached to an array of buttons and a mouse" [Wellner et al., 1993].

The above concepts and theories provide an explanatory framework to discuss and analyze a great variety of different post-WIMP systems in a general, abstract way and independent from a specific input technology or interaction technique. Yet, they largely remain a tool for researchers. Eventually all these notions of *naturalness*, *reality*, and *embodiment* have to be broken down to concrete things. Thus it is necessary that the discussion of the characteristics of post-WIMP systems also mentions specific technologies and interaction concepts that are employed in real post-WIMP systems. Below, I therefore try to address both higher-level theories as well as conceptual and technological aspects. However, to ensure that the focus of this thesis is not too broad, I want to restrict the term *post-WIMP* in the following to those systems whose main output channel is a visual one (i.e. which have a display or screen). While this restriction shrinks down the amount of systems to a manageable amount, it still leaves enough different characteristics that span multiple fields of research.

Table 1 on the following page provides a quick summary of these characteristics and compares them to the characteristics of traditional WIMP systems. The following three themes are used as a framework to structure the discussion of the characteristics below: The theme *Devices* addresses the fact that ever more sophisticated devices can be used to interact with systems. It is shown how this influences the characteristics of a system. The theme *Graphics* shows which graphical characteristics many state-of-the art post-WIMP systems employ to enhance the interaction. And finally, the theme *Users* addresses the fact that post-WIMP systems afford collaborative and social interaction.

	WIMP System	Post-WIMP System
Devices		
Type of Input	Time-Multiplexed (Sequential)	Time-Multiplexed (Sequential) and Space-Multiplexed (Concurrent)
Number of Devices	Few (Mouse, Keyboard)	Multiple Devices and Multiple Points per Device
Physical Objects as Input	No	Yes
User Capacities Exploited	Few (disembodied)	Multiple (embodied)
Modalities Used	Few	Multiple
Interface Metaphor	Conversational	Model World
Directness of Interaction	Mostly Indirect	Indirect (Gestures) and Direct (Manipulations)
Graphics		
Layout	Rigid	Flexible
Scale	Rigid	Flexible
Physical Behaviors	No	Yes
Animations	Few	Many
Users		
Single- or Multi-User	Only Single-User	Single- and Multi-User

Table 1: Characteristics of WIMP and post-WIMP systems

1.1 Devices

Due to great advancements in the hardware industry, a myriad of diverse input (and output) devices got ready for the consumer market in the last few years and even more can be expected in the near future. For many users these devices are probably the most salient characteristics of post-WIMP systems, as changes in input devices are physically perceptible while most changes in user interfaces or interaction are only virtually and cognitively perceptible. Some of these new devices were in the pipeline of research for many years, but only recently achieved market maturity. Take for example the improvements in multi-touch technology. While research in this field has a long history[6], it was only until the iPhone appeared in 2007 that this technology gained notable market share. But since then it catapulted the mobile industry into a new era, with smartphone sales increasing steadily[7]. Another notable technology that currently impacts researchers as well

[6] http://billbuxton.com/multitouchOverview.html
[7] http://www.guardian.co.uk/technology/2011/nov/03/q3-2011-smartphone-growth-continues

as consumers are depth cameras such as the Microsoft Kinect[8] which enable non-invasive body tracking of multiple persons for a very low price. Then, there is also a huge market of special-purpose sensors, such as humidity-, infrared- or ultrasonic-sensors, that can be integrated very easily into computer systems by using platforms such as Arduino[9] or .NET Gadgeteer[10].

This is only a brief outline of technologies and devices that are currently available. Instead of focusing on particular of these, I want to work out some of their common characteristics in the following and point out what consequences these characteristics have for the users.

1.1.1 Natural Interaction

For the user the availability of these diverse input devices means that he can gradually exploit more and more of his sensory and motor skills in the interaction with a machine and thus gradually diverges from the creature of figure 1 on page 2 to a complete human. Such a development has also been observed by Jacob et al. who state that "emerging interfaces support an increasingly rich set of input techniques based on these skills, including twohanded interaction and whole-body interaction" [Jacob et al., 2008]. Their RBI theme of *Body Awareness and Skills* is a direct outcome of this observation. Often the interaction with such devices is said to be *natural*, a term that is very difficult to grasp and subject to many a discussion. In the physical world, it is quite obvious what natural interaction means: "When interacting with objects in the physical world, we take advantage of natural skills developed over our lifetimes. We use our fingers, arms, 3D vision, ears and kinesthetic memory to manipulate multiple objects simultaneously, and we hardly think about how we do this because the skills are embedded so deeply in our minds and bodies" [Wellner, 1993]. In my opinion, the last part of this quote is the most important: Interaction is *natural*, once it is deeply entrenched and we do not have to think about it any more. Currently, when people talk about *natural* user interfaces they are concerned with "how we leverage the potential of new technologies to better mirror human capabilities, optimize the path to expert, apply to given contexts and tasks, and fulfill our needs" [Wigdor and Wixon, 2011]. Yet, sometimes extensive training is needed to establish a specific interaction and render it *natural*. Take for example the act of writing on a keyboard. The *natural* feeling that expert users have with this device usually required a substantial amount of training beforehand. In this respect, the interaction with a classic WIMP UI can also become *natural* at some point. In my opinion, it is therefore often helpful to think of *natural* as being the amount of *previous knowledge* a user carries along. This renders *natural* a very individual property that is actually "external to the product itself" and "refers to the user's behavior and feeling during the experience" [Wigdor and Wixon, 2011]. Thus, in order to make an interaction *natural* for a group of users, it has to "mirror their capabilities, meet their needs, take full advantage of their capacities and fit their task and context demands" [Wigdor and

[8] http://www.microsoft.com/en-us/kinectforwindows/
[9] http://www.arduino.cc/
[10] http://www.netmf.com/gadgeteer/

Wixon, 2011]. As Wigdor and Wixon point out further: "The trick, of course, is in helping them to feel that way the moment they pick it up, instead of after decades of practice". As a consequence, novice users can therefore act as a benchmark for the *naturalness* of an interface or interaction. While new input and output technologies do not necessarily guarantee a natural interaction per se, they at least carry the potential to speed up the learning curve as they make extensive use of users' previous knowledge.

1.1.2 Direct Interaction

New emerging technologies also promise to make the interaction with the system more direct. The notion of *directness* has been introduced by Shneiderman with the aforementioned concept of *direct manipulation* [Shneiderman, 1983] and investigated more thoroughly by Hutchins et al. [Hutchins et al., 1985]. The latter identified two aspects that influence a users feeling of directness. The first aspect is concerned with the *semantical* and *articulatory distance* in the gulfs of execution and evaluation (see figure 2).

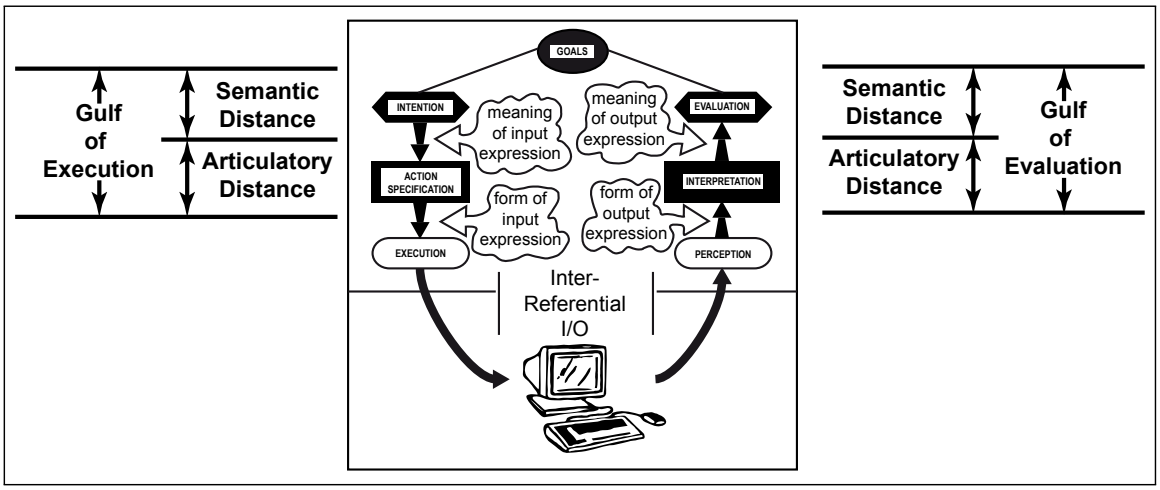

Figure 2: The articulatory and semantical distance of the gulfs of execution and evaluation influence the feeling of directness [Hutchins et al., 1985]

Hutchins et al. state that the feeling of directness is inversely proportional to the amount of cognitive effort it takes to manipulate and evaluate a system: "Cognitive effort is a direct result of the gulfs of execution and evaluation. The better the interface to a system helps bridge the gulfs, the less cognitive effort needed and the more direct the resulting feeling of interaction" [Hutchins et al., 1985]. The second aspect is concerned with the qualitative feeling of *engagement*, "the feeling that one is directly manipulating the objects of interest" [Hutchins et al., 1985]. Systems that have a high directness, "give the qualitative feeling that one is directly engaged with control of the objects – not with the programs, not with the computer, but with the semantic objects of

our goals and intentions" [Hutchins et al., 1985].

The influence of input (and output) devices on directness is mainly explained by the aspect of *articulatory distance* which is concerned with the relationship between the meaning of expressions and their physical form [Hutchins et al., 1985]. Input devices which establish such a relationship in a non-arbitrary way increase the feeling of directness. An example for such a relationship is the movement of an object with the finger on a touchscreen. Here, the form of the input directly resembles its meaning (i.e. to move the object from one place to the other). As the form of the output also directly resembles this meaning, directness is said to be high, both in terms of input and output. Classic input devices, such as mice or keyboards, do not always establish non-arbitrary relationships between input form and meaning and can therefore diminish the directness of the interaction. For example, issuing a textual command to alter the size of a graphical object on the screen is an indirect interaction, as the form of input has an arbitrary relationship to its meaning. The potential of new emerging input devices is that they can increase the directness in such situations as they support a greater variety of modalities, thereby making it possible to use a form of input that is as close as possible to the meaning.

Manipulations vs. Gestures In the context of new input devices, often the notion of gestures and gestural interaction pops up. While a lot of research has been conducted around gestures (e.g. to create consistent gesture sets for different input devices and domains [Frisch et al., 2009; Henze et al., 2010]), they are often criticized by researchers for being "unnatural" or "indirect" (e.g. [George, 2009; Norman, 2010; Jetter et al., 2010]). The reasons for this are manifold: First, gestures are typically symbolic or command-like. This means that their form is in an arbitrary relationship to their meaning, which increases the *articulatory distance* and reduces their directness. Depending on the type of gesture, it can also mean that a complicated form has to be employed to realize a conceptually simple action, which increases the *semantic distance* and further reduces their directness. Second, gestures are lacking appropriate feedback: "Because gestures are ephemeral, they do not leave behind any record of their path, which means that if one makes a gesture and either gets no response or the wrong response, there is little information available to help understand why" [Norman, 2010]. Thus, the success of a gesture can only be evaluated after it has been executed, which further reduces their directness as no feeling of *direct engagement* can be established. Because of their atomicity, gestures can not be undone partially and finally, gestures typically have to be learned by heart and are difficult to discover if no support is given.

For Norman it is unlikely that a whole system can be controlled solely by gestures [Norman, 2010]. He compares gestures to a language that only consists of verbs. As a result, certain subtleties of natural language, such as scope, range, temporal order, and conditional dependencies, can not be expressed with gestures and additional support from other input channels is needed to provide these missing elements. Yet, if applied carefully, gestures can also be a very successful means of

interaction. Norman for example states that "by their potential to engage the entire body, they can enhance the pleasure and engagement of participants" [Norman, 2010].

In stark contrast to gestures is the notion of manipulations, which represent continuous operations that lead to smooth changes in the system state. Typical manipulations are often found in multi-touch systems where they are for example used to drag, resize or rotate graphical objects. For Jetter et al. these manipulations are true representatives of the *direct manipulation* principles [Jetter et al., 2010]. Unlike gestures, manipulations are reversible as the user is in full control and has immediate feedback of his operations. Because manipulations have a very low *semantical* and *articulatory distance* and also support the feeling of *engagement*, their *directness* is very high.

Model World vs. Conversational Interface The dichotomy between manipulations and gestures is in fact rooted in a greater dichotomy of the overall interface metaphor a particular system employs. For Jetter et al. gestures are the "reincarnation of the conversation metaphor in NUIs", because gestures always mean "to talk about (intended) things" [Jetter et al., 2010]. Manipulations however are "intentional physical activities to change the surrounding world" [Jetter et al., 2010], which to Jetter et al. renders them *natural*. Manipulations always mean "to act in the world to make (intended) changes" [Jetter et al., 2010]. The underlying interface metaphor that permits such direct manipulations is that of a model world. In a model world, the user can be directly engaged with the objects of this world, whereas a conversational interface is always an intermediary between the user and the task [Hutchins et al., 1985]. A model world strongly supports the feeling of directness: At the input side, the user does not have to describe his actions but can perform them directly. At the output side, the results of these actions are directly visible, whereas in a conversational interface they are described by the system [Hutchins et al., 1985]. As a consequence, it can often be seen that post-WIMP systems employ a model world metaphor combined with direct manipulation techniques to create a *natural* and *direct* interaction for the user.

1.1.3 Concurrent Interaction

Interacting concurrently is natural for humans in the physical world. Take for example the act of driving a car: Many body parts and sensory organs are active at the same time to realize a set of different operations, such as steering, changing gears, accelerating and braking. There are three reasons why this is possible: First, we are (usually) equipped with multiple limbs and sensory organs whose coordination is learned in early childhood and established during the entire lifetime. Second, due to training, interactions can become automatic and unconscious. For example, during driving school we had to focus our attention at specific interactions more intensively, while after some months of experience all interactions are realized automatically and unconsciously. And third, the physical world (usually) permits it. We also have established a set of different strategies

and techniques to overcome problems with parallel activities. For example, if the cognitive system is overwhelmed of too many concurrent activities we automatically fall back into sequential mode by focusing our attention on the most important activity.

When it comes to the interaction with classic WIMP systems we can not benefit fully from our parallel interaction skills. WIMP applications accept but one input at a time which is provided by either the mouse or the keyboard. Thus, they make use of only few limbs and sensory organs (this limitation gave rise to the caricature in figure 1 on page 2). In 1986, Bill Buxton identified two reasons for this: "First, most of our theories about parsing languages (such as the language of our human-computer dialogue) are only capable of dealing with single-threaded dialogues. Second, there are hardware problems due partially to wanting to do parallel things on a serial machine" [Buxton, 1986]. While the second reason has dissolved in the meantime, the first still applies for most user interfaces today. The overall style of interaction is not designed for concurrent activities. The user is typically forced into a sequential back and forth of commands and answers, as described by the Seeheim model [Pfaff, 1985]. Thus, even if he wanted to use several different input devices concurrently, the system and its applications do not permit it. In order to make use of these, the style of interaction has to change significantly. One approach that several post-WIMP systems pursue is to employ the aforementioned model world metaphor instead of a conversational metaphor. Buxton's first reason does not apply to these, as the interaction in such model worlds completely avoids the notion of a dialogue. It can therefore be designed in a way that exploits our parallel interaction skills.

In post-WIMP systems, concurrent interaction can result from two different ways of input: First, several input devices can be used to target different motor functions and sensory organs. Such an interaction is usually termed *multi-modal* as multiple different input and output modalities are employed to interact with the system. Research in this area has established a separate field in HCI and dates back to the seminal "Put-that-there" system of Richard A. Bolt which employed voice and gesture input to move graphical objects on a screen [Bolt, 1980]. The second source of concurrency are input devices that produce multiple input points in parallel. This multi-point input can for example stem from multi-touch tabletops or body tracking cameras.

Redundancy In its ultimate form, concurrent interaction can be used to achieve *redundancy*. Redundancy means that the loss or misinterpretation of one sensor signal can be compensated by one or several others. While this is essential for people with impaired sensory function, it also greatly facilitates communication and interaction of people with intact sensory organs. For example, deixis, the act of referring to things, persons, locations or time with language, is often accompanied by pointing the hand towards the respective object or looking into the respective direction, thereby supporting situations where the communication partner could not fully understand the content or meaning of the sentence. This ultimately means that "the message is enormously redundant, and you can pull the signal out of any of many concurrent channels" [Negroponte, 1991].

Redundancy is investigated mainly in the research field of multi-modal user interfaces. Unfortunately, most findings of these research efforts have not yet been integrated into current WIMP systems and even into most post-WIMP systems which still leverage only a single modality at a time. In the end, achieving real redundancy is certainly an ambitious goal that involves a variety of conceptual and technological challenges.

1.1.4 Tangible Interaction

Interaction with purely digital artifacts does not have the same quality and fidelity as interaction with artifacts of the physical world. As a consequence, researchers began to explore the application of physical artifacts in the context of human-computer interaction. One of the first were Fitzmaurice et al. who referred to the resulting interfaces as *Graspable User Interfaces* [Fitzmaurice et al., 1995]. In order to make use of the rich affordances of physical artifacts in the interaction with their user interface, they attached little *bricks* to virtual elements on a screen. These bricks acted as handles and could be used to manipulate the respective virtual elements (i.e. to move, resize or rotate them). Later, Ishii and Ullmer expanded Fitzmaurice et al's concept with their notion of *Tangible User Interfaces (TUIs)* [Ishii and Ullmer, 1997]. With TUI's, not only physical objects (tangibles) are employed in the interaction, but also the physical environment where the interaction takes place is considered more thoroughly.

The benefits of using physical objects and the physical environment for the interaction with a system are manifold and can often be explained with findings in cognitive sciences such as the aforementioned embodiment theory [Shaer and Hornecker, 2010]. Klemmer et al. for example state that *motor* or *kinesthetic memory* is exploited when physical movements are dedicated to interface functions and physical feedback from objects helps users to distinguish interface functions kinesthetically [Klemmer et al., 2006]. An often stated advantage of tangibles is that they provide both affordances and constraints. Affordances refer to the "possibilities for action that we perceive of an object in a situation" [Shaer and Hornecker, 2010], whereas constraints "physically prevent certain actions or at least increase the threshold for an action" [Shaer and Hornecker, 2010]. Especially the application of constraints has been investigated thoroughly in Ullmer et al's article about *Token+Constraint* systems [Ullmer et al., 2005]. The most important advantage of TUIs is that tangibles enable true concurrent interaction and ultimately enable co-located multi-user interaction (which is discussed separately below in subsection 1.3 on page 19). The reason for this is that the interaction with tangibles is entirely space-multiplexed in comparison to the interaction in classic WIMP user interfaces which is a mix of time-multiplexed input and space-multiplexed output. With time-multiplexed input, one device is used to control different functions at different points in time, whereas with space-multiplexed input each function of the user interface has a dedicated transducer which occupies its own space [Fitzmaurice et al., 1995]. As the output of graphical systems is usually space-multiplexed, traditional WIMP systems have an inherent dissonance. This

dissonance is resolved in TUIs where input is distributed spatially [Terrenghi et al., 2007] and bimanual and concurrent interaction is enabled. Although such bimanual and concurrent interaction is also possible without physical objects (as discussed in the previous subsection), their inert properties enable entirely different qualities of interaction. Terrenghi et al. for example noticed differences in bimanual interaction when a task is performed with digital elements only, compared to when it is performed partly with the help of a physical artifact [Terrenghi et al., 2007]. While in the digital only interface users only performed symmetrical bimanual interactions, the hybrid interface elicited asymmetrical bimanual interactions. Such effects can largely be attributed to the *tangibility* of the physical artifact which allows the user to offload cognitive resources to the haptic system, making it possible to let one hand operate on a different task than the other.

Early on, Graspable and Tangible User Interfaces were often realized on horizontal surfaces as they offer the advantage that physical objects can be placed on top of. In parallel to this, so called *interactive surfaces* emerged due to the advancements in multi-touch technologies. As these *interactive surfaces* often provide means to sense and track not only fingers but other physical objects, it is only reasonable that both capabilities are used increasingly together to generate a "hybrid" interaction. Consequently, Kirk et al. referred to such systems as *hybrid surfaces* [Kirk et al., 2009]. These kinds of systems are probably the most widely used representatives of TUIs, although not the only ones. Other technologies that are used to implement TUIs are for example magic lenses or tool glasses (based on the early work of Bier et al. [Bier et al., 1993]) which are able to provide a different view on the information on a screen. Also, augmented reality systems, which date back to the early work of Wellner [Wellner, 1993], can be seen as belonging to the field of TUIs.

1.2 Graphics

1.2.1 Flexible Layouts

A common characteristic of many post-WIMP systems is the rejection of the strict layout rules of classic WIMP user interfaces. While these employ classic UI widgets such as lists, grids or tables to structure and position the information on the screen, many post-WIMP systems use rather loosely structured layout containers such as canvases or infinite landscapes that can flexibly adapt to the users' needs. This is a direct consequence of the shift from the conversational interface to the model world interface. In a model world, the user needs the opportunity to rearrange entities as he wishes, which is not possible with a static layout. While the free arrangement of objects provides more freedom for the actions of the user, it can also prove cumbersome as typically it is not the main task of the user to arrange objects on a screen. Several researchers therefore explored additional possibilities to facilitate the arrangement of objects in such loosely

structured layout environments. For example, the Bubble Clusters [Collins et al., 2009] and Bubble Sets [Watanabe et al., 2007] techniques allow the clustering of information items into organic structures. Different clustering and piling techniques have also been explored in the BumpTop system [Agarawala and Balakrishnan, 2006]. In Frisch et al's work, alignment tools and layouts can be created in a lightweight manner with the help of multi-touch manipulation techniques or pen and touch gestures [Frisch et al., 2011].

1.2.2 Flexible Scale

In post-WIMP systems that are based on the model world metaphor and that employ loose layout structures, the scale of the whole user interface or of individual components can often be adapted flexibly by the user. The main representative of such systems are Zoomable User Interfaces (ZUIs) whose origin dates back to the works of Perlin and Fox and their *Pad* system [Perlin and Fox, 1993]. ZUIs allow the user to explore the information space and access its information items by seamlessly panning and zooming through a 2.5D environment. They are motivated by Perlin and Fox with the assumption "that navigation in information spaces is best supported by tapping into our natural spatial and geographic ways of thinking" [Perlin and Fox, 1993]. In ZUIs, often the enlargement of information objects not only causes them to occupy more screen space (geometric zooming), but also to gradually reveal more and different content, a concept called *semantic zooming*.

A post-WIMP user interface paradigm that is based strongly on the concepts of ZUIs is the ZOIL paradigm (Zoomable Object-oriented Information Landscape) [Zöllner et al., 2011; Jetter et al., 2012]. It consists of a set of design principles to support the design and implementation of post-WIMP user interfaces. These design principles provide "patterns of solution as heuristics for choosing suitable conceptual models, visualizations and interaction techniques" [Zöllner et al., 2011]. The ZOIL software framework, which has been co-developed by the author, provides a great variety of reusable components and means to build systems according to the principles of the ZOIL paradigm. The validity of these principles and the utility of the ZOIL framework has been shown in various systems. Geyer et al. for example created a hybrid surface to support the collaborative design activity of affinity diagramming [Geyer et al., 2011]. Also, certain parts of the Facet-Streams system [Jetter et al., 2011], which is discussed in more detail in subsection 9.1 on page 103, are based on the ZOIL framework.

1.2.3 Physical Behavior

As suggested by Jacob et al. with their RBI theme of *Naïve Physics*, users can benefit greatly from an interaction that builds upon their tacit knowledge of physics. This does not necessarily mean that physical tokens or objects have to be used, such as with TUIs, but it can also mean that physical properties can be attributed to virtual objects or that a group of virtual objects abides by the

laws of physics. The benefit of these behaviors is that they can greatly augment and accelerate the interaction with a system. Forlines et al. for example observed that dragging an object with a finger across the table is inefficient, compared to the same operation with a mouse that has acceleration enabled [Forlines et al., 2007]. If that object however contained inertia, it could be flicked across the table with almost no physical effort. Such physical behaviors also greatly contributed to the success of the iPhone and similar smartphones as they allowed the quick scrolling of lists and collections. One of the first systems that made extensive use of physical behaviors was *BumpTop* [Agarawala and Balakrishnan, 2006] which is a 2.5D simulation of the default desktop, where the icons are "influenced by physical characteristics such as friction and mass" [Agarawala and Balakrishnan, 2006]. The user of the system can toss around the icons of the desktop to create loose clusters. If they collide with other icons or the wall, they "bump against and displace one another in a physically realistic fashion" [Agarawala and Balakrishnan, 2006]. Thus, the interaction with the system feels more like the interaction with physical objects. In contrast to the rigid and discrete way of interaction that is employed in the digital world, such a more continuous and analog way of interaction can especially prove beneficial for novice users as they can transfer their knowledge of the real world to a new domain. Wilson et al. state that the interaction on a multi-touch tabletop combined with physical behaviors provides "unusually high fidelity" for manipulations, as several forces can be applied simultaneously to the objects [Wilson et al., 2008]. They also note that the manipulation of multiple objects or 3D objects is facilitated greatly if physical behaviors are used.

Currently only little systematical research has been conducted to investigate the nature and the benefits of these physical behaviors for the interaction (e.g. [Wilson et al., 2008; Lee and Lee, 2009; Bragdon et al., 2010; Baglioni et al., 2011]). For Norman the fine tuning of physical behaviors is currently an art. In order for it to evolve, it has to be transformed into a science [Norman, 2010]. Yet, regardless of whether physical behaviors are backed by broad scientific works or not, the key factor for their application in user interfaces is to strike a balance. The creators of BumpTop for example state that their intention was to "leverage the beneficial properties of the physical world, but not be overly constrained by or dogmatically committed to realism" [Agarawala and Balakrishnan, 2006]. This is very important as a too strict realism can also hamper the interaction. A designer should not forget the advantages that the virtual world provides and that set it apart from the real world.

1.2.4 Animations

Many post-WIMP systems employ animations to support their interaction techniques. Such support can range from subtle indicators of property changes, such as the layout animations in BumpTop [Agarawala and Balakrishnan, 2006], to interaction techniques that rely strongly on animations, such as the zooming functionality in the ZOIL framework [Jetter et al., 2012]. In my opinion, animations are an important means for the success of many post-WIMP interaction techniques.

Several cognitive findings in literature strongly support this view. May and Barnard for example motivated animations for the user interface with a cognitive model that they also applied in cinematography to explain "the flow of information through the human cognitive system" [May and Barnard, 1995]. They state that "salient information or objects should not just appear or disappear from the screen, but should make some sort of entrance or exit" [May and Barnard, 1995]. This is backed by Chang and Ungar who state that "when the user cannot visually track the changes occurring in the interface, the causal connection between the old state of the screen and the new state of the screen is not immediately clear" [Chang and Ungar, 1993]. As a consequence, they used techniques from cartoon animation "to replace sudden changes with smooth transitions, offloading some of the cognitive burden of interpreting the change to the perceptual system" [Chang and Ungar, 1993]. Robertson et al. also justify the use of animations in their Cone Tree visualization with cognitive findings: "The perceptual phenomenon of object constancy enables the user to track substructure relationships without thinking about it. When the animation is completed, no time is needed for reassimilation" [Robertson et al., 1991]. This effect of shifting "some of the user's cognitive load to the human perceptual system" [Robertson et al., 1991] is backed by several cognitive theories. Chui and Dillon for example refer to the ecological or Gibsonian approach to perception/action research: "One output from this line of research is the view that temporally extended events in the environment are fundamental sources of stimulus for the perceptual system. Living organisms become attuned to various events that are specified by trajectories through time. As such, animation in the interface is likely to afford benefits to users seeking to identify locations and spatial arrangements" [Chui and Dillon, 1997]. One of the goals of post-WIMP user interfaces is to allow the user to focus on the task at hand and not the interface or interaction. Animations can support this goal: "By offloading interpretation of changes to the perceptual system, animation allows the user to continue thinking about the task domain, with no need to shift contexts to the interface domain" [Chang and Ungar, 1993].

However supportive animations can be, it must be taken care of that the user is not distracted or overwhelmed by them. Gonzales for example states that "to be an effective decision support tool, animations must be smooth, simple, interactive and explicitly account for the appropriateness of the user's mental model of the task" [Gonzalez, 1996]. Thus, using animations just for the sake of it may even hamper the interaction if they are inappropriate.

1.3 Users

With their RBI theme of *Social Awareness and Skills (SAS)* Jacob et al. highlight the fact that users are social beings: "People are generally aware of the presence of others and develop skills for social interaction. These include verbal and non-verbal communication, the ability to exchange physical objects, and the ability to work with others to collaborate on a task" [Jacob et al., 2008].

With current WIMP systems, however, a group of users can not interact simultaneously with the interface, but has to choose one master-user that is in control of the input device(s). Other group members can only contribute indirectly by giving instructions. As group work, such as group decision making, is usually a democratic undertaking, it is however required that every member of the group has equal possibilities to participate.

Co-located Settings It has been highlighted above that new emerging input technologies bring the necessary means to enable multi-user interaction on the same device (i.e. in a co-located setting). The reason for this is that they allow space-multiplexed or parallel input from multiple users. Also, new display form factors such as tabletops naturally afford face to face group collaborations as they invite users to gather around them and do not require a dedicated input device. The interaction of a group in such settings is greatly improved if tangibles are used for the interaction. Because of their physical presence, they provide important cues for the users to raise the general awareness of the system state and the awareness of what the other members of the group are doing. Such effects have for example been observed in the evaluation studies of the Facet-Streams system [Jetter et al., 2011].

In group work, users also sometimes want to escape from the group, to work individually on a subset of the problem, afterwards synchronizing their result with that of other group members [Dietz and Leigh, 2001]. It is therefore important that multi-user systems provide the respective means for their users to do this. On large interactive surfaces users for example automatically create *personal territories* in front of them to arrange and work with digital and physical items [Scott et al., 2004]. Systems that facilitate the creation of such personal territories, like the system of Klinkhammer et al. [Klinkhammer et al., 2011], can therefore greatly support the user in this important activity.

Distributed User Interfaces (DUIs) Not all tasks can be done on a small area of a tabletop and also certain privacy concerns emerge from such settings. To address these issues, some systems make it possible to distribute parts or whole of their user interface "across multiple monitors, devices, platforms, displays, and/or users" [Melchior et al., 2009]. With such Distributed User Interfaces (DUIs) members of the group can employ their own personal devices, such as laptops, pads, or phones, to work on the respective subset of the problem. DUIs also not necessarily require the group to be at the same location. Similar to shared workspaces or groupware systems, members of the group can be distributed all over the world. The aforementioned ZOIL paradigm and software framework for example facilitates the creation of such DUIs in that it provides a shared visual workspace that is synchronized between all participants [Zöllner et al., 2011; Jetter et al., 2012]. Each participant thereby has its own view onto this shared workspace and can navigate therein independently.

2 Describing and Developing Post-WIMP Systems

When it comes to the design and development of post-WIMP systems, each of the characteristics of the previous section brings along a set of challenges that a designer or developer has to face and overcome. From my experience in practice and the insight I gained after reviewing a lot of literature, I can safely state that there is no "silver bullet" [Brooks, 1987] (i.e. no single formalism or technique) for both the design and the development of such systems. Every challenge is of a different nature and requires different approaches. To show this, I provide a general overview of these challenges in subsection 2.1. Unfortunately, there is still little support in terms of tools or programming languages to tackle them. To underpin this observation, I want to discuss the shortcomings of current development tools in subsection 2.2 on page 29. After this, I then want to advocate the use of formal methods for the design of interactive behaviors in subsection 2.3 on page 33 and finally make the case for finite-state machines in subsection 2.4 on page 36.

2.1 Challenges of Post-WIMP Systems

In this subsection, I want to provide a broad overview over some important challenges, developers of post-WIMP systems are likely to face and discuss potential means to tackle these. The next subsection then more thoroughly deals with a selected set of problems that ultimately point into the direction of finite-state machines. As this thesis is concerned with the *interactive behaviors* of post-WIMP systems, I want to focus the following discussion on those challenges that are directly related to these interactive behaviors. As a consequence, I do not discuss any challenges that stem from the *Graphics* theme of the previous section. The characteristics from this theme, such as flexible layouts, flexible scales, and animations, are not directly related to interactive behaviors and are already well supported in most current user interface frameworks. Even the simulation of physical behavior is usually straightforward to integrate, as existing physics engines abstract away the underlying math.

It turns out that the remaining challenges from the *Devices* and *Users* themes are still manifold and span a variety of different research fields. In addition to this, they also cause difficulties at different levels of abstraction. Shaer et al. for example state that the designers and developers of Tangible User Interfaces "encounter several *conceptual*, *methodological* and *technical* difficulties" (emphasis mine) [Shaer and Jacob, 2009]. This classification is not exclusive to TUIs, but generally holds true for the challenges of post-WIMP systems which also have *conceptual*, *methodological* and *technical* aspects.

The main source for almost all challenges are the various different and heterogeneous input devices and technologies that are being used to interact with post-WIMP systems. As stated by van Dam, the interaction with these devices "involves multiple parallel high-bandwidth input and output channels operating full-duplex on continuous (not discrete) signals that are decoded and probabilistically disambiguated in real time" [van Dam, 2001]. Because it is not possible to address each class of input devices separately in this context, the challenges presented below are a generalization of the properties these devices have. At first I address the problem of multi-modal and multi-point input from a single user perspective. This kind of input can arrive concurrently at the application and is therefore difficult to handle by the developer. The next step from concurrent input of a single user is the concurrent input of multiple users. Here, the design and interaction concepts of the application have to be altered significantly and input potentially has to be mapped to a specific user. As envisioned by Green, Nielsen or Van Dam, the bandwidth of input devices is increasing continuously [Green and Jacob, 1991; Nielsen, 1993; van Dam, 1997]. While the amount of data that a pointing device, such as a mouse or a touchscreen, produces is easily manageable, devices such as the Kinect generate huge amounts of input data every second. These high-bandwidth devices usually produce continuous input which differs significantly from the discrete input of a keyboard or mouse button. The problem of discrete and continuous input is therefore addressed in more detail thereafter. Whereas discrete input is typically straightforward to interpret, continuous input is often fraught with uncertainty. In order to recognize gestures or natural language, potential ambiguities need to be resolved beforehand. Finally, the problem of ambiguous input is therefore addressed.

2.1.1 Concurrent Interaction with Multi-Modal and Multi-Point Input

In post-WIMP systems, input can stem from multiple devices or from multiple points of the same device. While the latter is called multi-point input, the former is often referred to as multi-modal input, because the usage of multiple devices requires the user to employ different modalities. The overall characteristic of both cases is that multiple input channels (i.e. devices or points) can produce values at unexpected times. Such input can therefore not expected to be sequential, because values of one channel can overlap or interfere with values of the other channel, which automatically creates situations of concurrency. To realize interactive behaviors it is often necessary to resolve these situations on different levels of abstraction, ranging from the input event level to the interaction technique level. The following paragraphs highlight the most important challenges that emerge from these situations.

Fusing different input devices A multi-modal system often has to combine two or more modalities (i.e. input from multiple devices) to create some particular higher-level interaction. It may for example be necessary to synchronize video and audio input or to combine free hand gestures

with audio input such as in the "Put-that-there" system [Bolt, 1980]. Such a task is usually called *fusing* and is one of the pillars of multi-modal interaction [Nigay and Coutaz, 1993]. The fusing of different input devices covers all cases where the data types of the different modalities or devices are different. The result thereof typically is some combination of all involved input events. Unfortunately, current programming languages or user interface toolkits do not provide any help for this important activity. Each device or modality is usually treated in isolation and synchronization and coordination mechanisms have to be established manually. Researchers therefore have proposed several different concepts and toolkits to aid in both the design and development of fusion mechanisms. Very well-known toolkits are for example the OpenInterface platform [Lawson et al., 2009] or Squidy [Rädle, 2011] which both provide reusable components and generic mechanisms to combine different modalities in a straightforward way.

Grouping input points When several input events occur at (nearly) the same time, they often have to be grouped together to form a higher-level interaction. Examples for such interactions are the well-known multi-touch manipulations that are used to drag, resize and rotate graphical objects. The input points that form such interactions usually have to be grouped together according to some spatial condition. It is for example necessary to group together all input events that currently affect the same user interface element in order to create the appropriate response. Especially with multi-touch manipulations it is often the case that fingers do not touch the screen at exactly the same time, but consecutively. It is then necessary to add the new input point to an already started manipulation, which may completely alter the semantics of the manipulation (i.e. from *drag* to *resize*). Current programming languages and user interface toolkits do not provide any special help for such tasks as every input point is treated in isolation. There is also no built-in means to specify that certain input events belong together and form some higher-level interaction. This requires the developer to set up these interactions and add or remove input points manually.

Detecting input sequences For other interaction techniques it is important that input of multiple devices or points arrives in a defined order. Examples for such interaction techniques are simple gestures such as a tap gesture or more complex gestures such as the diagram editing gestures of Frisch et. al [Frisch et al., 2009]. The detection of input sequences is also necessary when natural language is used in conjunction with other modalities, as the syntactical rules of the language (e.g. Subject-Predicate-Object) imply a certain order. The problem with such interactions is that state has to be preserved between subsequent input events in order to recognize the whole interaction. Each input event creates a certain situation, which permits or prevents certain follow-up situations. Once again, it is not possible with current programming languages to specify such interaction sequences in a concise way, as the input is treated in isolation and distributed over several event handler methods.

Specifying timing constraints Certain interaction techniques require the developer to specify timing constraints. For example, a double tap gesture requires the four events (Down-Up-Down-Up) of the two taps to occur in a certain timespan. Describing such timing constraints is usually straightforward in natural language or with formalisms such as finite-state machines. Yet, as current programming languages do not provide sufficiently abstract means to specify these constraints, the actual implementation often requires complex management of timers and their events and it is usually distributed over several different methods. As timers also often run in separate threads, additional situations of concurrency are emerging thereof.

2.1.2 Multi-User Interaction

While concurrent interaction per se is challenging to design and develop, it is likely to get more complex when a system promises to cater multiple users, which I deem the ultimate form of concurrent interaction. Devices such as multi-touch tabletops foster collaboration due to their form factor and spatially distributed input. However, in order to make use of these features the user interface and interaction techniques have to be designed specifically. A tabletop alone will not hinder application developers to design user interfaces that can only be controlled by a single user. With good reason, many people still deride devices such as the Microsoft Surface, as hardly any application that is presented for these supports substantial multi-user aspects and therefore justifies their high price. For me, this is not too astonishing as the design and development of true multi-user applications is very challenging and requires a totally different mindset in terms of interaction concepts and development techniques. In the following, I therefore want to give an impression what conceptual and technical challenges designers of such systems are likely to face.

Global State The most important point when designing true multi-user interfaces is to avoid global state. Such global state is usually inevitable in classic WIMP user interfaces as they are based on the notion of a single *focus* point. Nielsen stated in 1993 that "window systems and other attempts at application integration typically forced the user to 'be' in one application at a time, even though other applications were running in the background" [Nielsen, 1993]. Unfortunately, this has not changed since then. In a WIMP GUI only one window can be active, and in this window only one element has focus. A single focus point implies that global state is established as typically a composite command syntax is used to execute actions on objects. This composite syntax is either the verb-object syntax or the object-verb syntax. With the verb-object syntax the user first selects the action he wants to execute and then the object to which the action applies. With the object-verb syntax the user first selects the object of interest and then the action he wants to apply. With both approaches, state has to be maintained from one action to the next. As a single focus is used, this state is automatically global. For example with the object-verb syntax, every tool automatically applies to the one object being in focus. While this is unproblematic with a single

pointing device in a single-user system, it can cause unsolvable concurrency issues in a multi-user system: The selection of operand and operator is usually a two-step process. If a user selects an operand or an operator, his follow-up selection can interleave with the action of another user, so that his initial selection can mistakenly be attributed to the selection of the other user. To resolve this issue their actions needed to be sequentialized, which in turn renders the system useless for multiple users. In general, multi-user interfaces should therefore entirely avoid the notion of a focus point to avoid global state and global functionalities. To achieve this, entirely different concepts have to be applied. It is for example possible to couple the functionality a user can execute with an object with the appearance of the object. This approach stems from object-oriented user interfaces (OOUIs) [Collins, 1994]. Another option is to use tangibles (physical tokens) as tools to manipulate digital objects. Here the two-step process of selecting operator and operand is split up on the user interface and the physical environment. The tangible is unambiguously connected to the user who holds it, which prevents interleaving effects. In the Facet-Streams system for example, both tangibles and concepts from object-oriented user interfaces have been used to enable multi-user interaction [Jetter et al., 2011]. There, parts of a database query are represented by physical glass tokens with virtual functionality being directly attached to the digital representation of the glass token.

Eventually, there is no syntax involved in post-WIMP interaction at all. This was envisioned by Nielsen who stated that future interfaces will "to some degree be syntax-free". He referred to interactions such as writing with a digital pen which are natural to the user and do not require him to think about syntax at all [Nielsen, 1993].

Coordination When several users interact with an interface, they have to coordinate their actions in a meaningful way. In the physical world, we established social protocols to ensure that nobody interferes with the other. While these "standards of polite behavior" [Morris et al., 2004] still work when we interact in digital environments, they do not automatically prevent conflicts at the interaction level [Greenberg and Marwood, 1994]. Users may for example inadvertently *steal* or manipulate objects from another user, change the overall layout of the objects or occlude the view of another user. To classify these conflicts, Morris et al. have presented a framework of different coordination mechanisms for multi-user environments. These mechanisms are direct manipulation techniques that can be used to avoid conflicts and "help ensure that software has deterministic, predictable responses to multi-user interactions" [Morris et al., 2004]. While their mechanisms are mainly digital in nature, the interaction in a multi-user environment can also benefit from the use of physical objects and their well-known properties, such as object permanence and gravity. In the Facet-Streams system for example, the use of tangible glass tokens prevents conflicts at the interaction level, as they are considered to be associated with the user that put them on the screen [Jetter et al., 2011]. Because a user typically puts them on the screen directly in front of him, the position strongly communicates this association to the user. This phenomenon

has thoroughly been researched by Scott et al., which suggest that multi-user systems should consider appropriate territoriality "to help people coordinate their task and social interactions" [Scott et al., 2004]. Their research is based on the concept of territories to partition the workspace of a group into meaningful areas. While Scott et al. mainly consider the workspace to be a tabletop, Ballendat et al. extend this to the entire environment. They suggest the concept of *proxemics* to "regulate implicit and explicit interactions" in ubiquitous computing environments [Ballendat et al., 2010]. Proxemics are based on the interpretation of spatial relationships between objects and people. With proxemics the user interface and the interaction with the user interface may change depending on the configuration of people and objects in a room. One such problem is for example the orientation of objects on a tabletop. If the system knows where the user is located, it may automatically rotate the objects to his direction.

Mapping Input to Users All the coordination concepts from above require that users can be distinguished from each other and input can be attributed to its origin. Several technologies have been explored in the past to achieve this. Obvious are vision-based tracking systems which try to identify the bodies or hands of users [Dohse et al., 2008]. Another technique is the use of modulated electric fields to identify individual users, such as in the commercially available DiamondTouch[11] [Dietz and Leigh, 2001]. With such approaches, the main input (e.g. from a touchscreen) can be unambiguously attributed to a specific user. The challenge with these technologies is that two sources of input have to be related to each other in some way. Libraries such as Rx (see section 5 on page 65) or input frameworks such as Squidy [Rädle, 2011] can greatly help in such cases to combine and coordinate the two separate input streams.

A different technology to identify users are infrared sensor arrays. Their precision is not as high as with the former technologies, thus input can not be directly attributed to its origin. Yet, they can nonetheless be used to create personalized workspaces around a tabletop, such as in Klinkhammer et al. [Klinkhammer et al., 2011].

2.1.3 Discrete vs. Continuous Input

The signals that input devices produce are typically classified into two types: discrete and continuous. Discrete signals are usually called *events*, which are "atomic, non-persistent occurrences in the world, that is, we sense that they happen at a particular point in time" [Dix and Abowd, 1996]. Examples are the press of a button on a mouse or a `TouchDown` event on a touchscreen. Continuous signals are "phenomena which can be observed at any time" [Dix and Abowd, 1996]. Dix refers to them as *status*, which to him are "things that persist and we observe in the world, that is, they have a measurable value at any moment" [Dix and Abowd, 1996]. Others call these continuous phenomena *streams* or *data-flows* [Chatty, 1992]. Examples for such streams are the signals

[11] http://www.circletwelve.com/

a mouse generates during movement or the video stream of a depth camera. Due to the digital nature of computers, streams have to be discretized on compulsion. However it is suggested to "keep their continuous nature in mind" [Chatty, 1992]. This is backed by Dix who states that "a user interface specification should first reflect the events and status as they are perceived by the user rather than those of the implementation of the system" [Dix and Abowd, 1996]. Dix also notes that most formalisms to describe user interfaces do not consider this dichotomy between event and status phenomena. This is problematic as certain phenomena have to be described in ways that do not reflect their real nature which eventually leads to user interfaces that are specified incorrectly or whose specification can not be understand.

To resolve such issues, Dix therefore proposed a model to describe both event and status phenomena in user interfaces [Dix and Abowd, 1996]. As he notes, specifying a user interface with this model "highlights important design issues which would be missed if either status or event phenomena were not properly treated". While Dix only proposed a (rather complicated) formalism, others have built toolkits that consider both events and streams. In Whizz, a system for building animated interactive applications, continuous and sudden *evolutions* are used to build animations [Chatty, 1992]. Similar to this is Fran (Functional Reactive Animation) which uses events and *behaviors* for composing richly interactive, multimedia animations [Elliott and Hudak, 1997].

As of today, none of the ideas presented by Dix and those realized by Chatty or Elliott are incorporated in the default user interface toolkits. Still, the default way of integrating input into an application is by means of events, which means that every single input is handled separately by a dedicated event handler method [Dix and Abowd, 1996]. This can prove very cumbersome with continuous phenomena such as a stream of position data from a body tracker. If a developer for example wants to reduce the jittering that is inherent to such input, he has to consider several subsequent data values. While this is straightforward with toolkits that are focused on continuous signals, it involves a lot of extra code in the case of event-based input. Thus, the developer of a post-WIMP application, who has to deal with both distinct and continuous signals, is left with only one mechanism to deal with two different phenomena.

2.1.4 Ambiguous Input

Many of the *natural* input devices that are used to enable post-WIMP interaction do not produce distinct input data. Instead their input is inherently fuzzy and ambiguous and has to be recognized and interpreted by the system. Mankoff et al. differentiate three types of ambiguity: recognition ambiguity, target ambiguity, and segmentation ambiguity [Mankoff et al., 2000]. *Recognition ambiguity* refers to those cases where input is interpreted differently by a recognizer and thus produces differently rated results. Examples are the recognition of gestures captured by body or hand trackers and the recognition of language, either spoken or written. *Target ambiguity* occurs, when "the target of the user's input is unclear" [Mankoff et al., 2000]. This type of ambiguity can for

example occur on multi-touch systems, where the finger of a person can overlay multiple graphical elements. This problem is also often referred to as the "fat finger problem". Segmentation ambiguity refers to the problem that multiple input events can be grouped in various ways. An example is the segmentation of characters of pen-based input, which depends on factors such as spacing and word recognition.

Before these ambiguities can be resolved, they have to be represented appropriately in the input system: "Input event properties need to be expanded from a single fact to estimates representing a range of possibilities" [Schwarz et al., 2010]. This suggests the use of probabilistic methods, such as probabilistic mass functions (PMF), to classify and rate the various ambiguous outcomes of an input. Typically these probabilities are expressed as a confidence value which accompanies the actual value. For methods that do not produce such confidence values, Hudson and Newell suggest the use of history or training data, to compute the probability value of a detected result [Hudson and Newell, 1992]. This is for example used in the body tracking algorithm of the Kinect sensor, where hundreds of thousands of training images are fed into a machine learning system [Shotton et al., 2011]. The results of this training data are then used by the algorithms to produce a skeleton simulation of the depth data on demand.

Once confidence values have been attached to different outcomes of an input, ambiguities can be resolved. Several solutions have been suggested which differ mainly in the location where ambiguity is resolved. The most simple and obvious approach is to use the best rated result as soon as it is detected. This approach completely relieves the developer from dealing with ambiguity at all, which to him becomes a black box. As present-day user interface toolkits do not contain any help to resolve ambiguous input, it is also the only viable approach that works out of the box. Yet, this early handling of uncertainty is criticized by many. Schwarz et al. for example state that it leads to a "stunted interactive experience" [Schwarz et al., 2010]. A better approach is to model the uncertainty with formalisms such as probabilistic finite state machines, as these can be directly integrated into the interaction design of a system [Hudson and Newell, 1992]. To reduce target ambiguity, Schwartz et al. advocate the use of lazy methods which delay the evaluation of the input to the latest possible point in time [Schwarz et al., 2010]. Their system uses a *mediator* component which takes ambiguous input and constantly talks to possible user interface elements to negotiate which element will eventually receive the event. At the extreme end of the spectrum are approaches that delegate the resolving of uncertainty completely to the user by presenting him the different alternatives. This is for example used by Mankoff et al. to augment a sketch-based user interface builder [Mankoff et al., 2000].

2.2 Shortcomings of Development Tools

Current programming languages and user interface toolkits often have significant shortcomings when it comes to the implementation of post-WIMP interaction techniques. To underpin this observation, a few selected examples are demonstrated in the following. The commonalities of these examples are then summarized at the end of this subsection.

2.2.1 Expressing Sequential Interaction

Many higher-level interaction techniques require input events to appear in a certain sequence. Simple examples are a tap gesture which consists of a `TouchDown-TouchUp` sequence or a drag manipulation which consists of a `TouchDown-TouchMove-TouchUp` sequence. Expressing such sequences in a concise way is not possible with present-day programming languages. Take for example the implementation of the tap gesture in listing 1. It is detected when a `TouchDown` event is followed by a `TouchUp` event in less than 100 milliseconds. To realize this, state information has to be preserved explicitly from the first to the second event in order to detect the sequence and in order to calculate the time difference. This requires the developer to reserve dedicated fields that can be accessed from each event handler method. The actual sequence thereby gets distributed on two different methods and has to be established manually. While this is still manageable in the single-touch case, it gets more complex in the multi-touch case, where multiple fingers can simultaneously affect a user interface element and multiple tap gestures can be active simultaneously. In this case, state information has to be preserved on a per-point basis, to prevent interleaving and concurrency effects.

```
DateTime timestamp;
bool touchDown;

OnTouchDown(TouchEventArgs e)
{
    touchDown = true;
    timestamp = DateTime.Now;
}

OnTouchUp(TouchEventArgs e)
{
    var diff = (DateTime.Now - timestamp).TotalMiliseconds;
    if(touchDown && diff <= 100)
        RaiseTapGestureEvent(new TapGestureEventArgs(e.Source));
    touchDown = false;
}
```

Listing 1: Detecting a tap gesture on a single-touch device

2.2.2 Expressing Parallel Interaction

Parallel interactions such as the resize or rotate multi-touch manipulations are similar to express as sequential interactions. The reason for this is that parallel interaction is only perceived to be parallel, it is in fact sequential as the input events are all handled sequentially in the user interface thread. The difference to sequential interaction is that for parallel interaction all input points have to be grouped together. Take for example the basic implementation of a multi-touch drag and resize manipulation in listing 2. Here, each finger that is currently affecting the UI element has to be stored in a collection in the `TouchDown` event. In the `TouchMove` event it can then be differentiated if the current movement is a drag operation (only one finger) or a resize operation (more than one finger). In order to calculate the displacement vectors for these operations, the last point of each finger has to be stored in a `Dictionary`. Again, the implementation is distributed over several event handler methods and state has to be saved explicitly between these methods.

```
List<TouchDevice> touches;
Dictionary<TouchDevice, Point> oldPoints;

OnTouchDown(TouchEventArgs e)
{
    touches.Add(e.TouchDevice);
    oldPoints[e.TouchDevice] = e.GetPosition(null);
}

OnTouchMove(TouchEventArgs e)
{
    if(touches.Count == 1)
        DragObject(e.TouchDevice);
    else if (touches.Count > 1)
        ResizeObject(e.TouchDevice);

    oldPoints[e.TouchDevice] = e.GetPosition(null);
}

OnTouchUp(TouchEventArgs e)
{
    oldPoints.Remove(e.TouchDevice);
    touches.Remove(e.TouchDevice);
}
```

Listing 2: Basic implementation of multi-touch manipulations

2.2.3 Expressing Timing Contraints

Interaction techniques that rely on timing constraints are usually complex to implement. In simple cases, such as the above tap gesture, it suffices to calculate a time difference between two subsequent events. Yet, often timers have to be introduced into the implementation as independent entities. It usually does not suffice to start those timers and wait until they expire. As they can be interleaved by other events, it might be necessary to stop or restart them. Also, timers usually run in a separate thread, which means that threading issues are likely to appear. To show how timers currently have to be used to realize interaction techniques, let's consider the implementation of a behavior from the Facet-Streams system. Here, visual links are connected to nodes to form a graph (see figure 3a). If a user places his finger on a link for one second (b), the link is released from the node and can be dragged around freely (c). If the user lifts his finger prior to the expiration of the timespan, the link remains connected to the node.

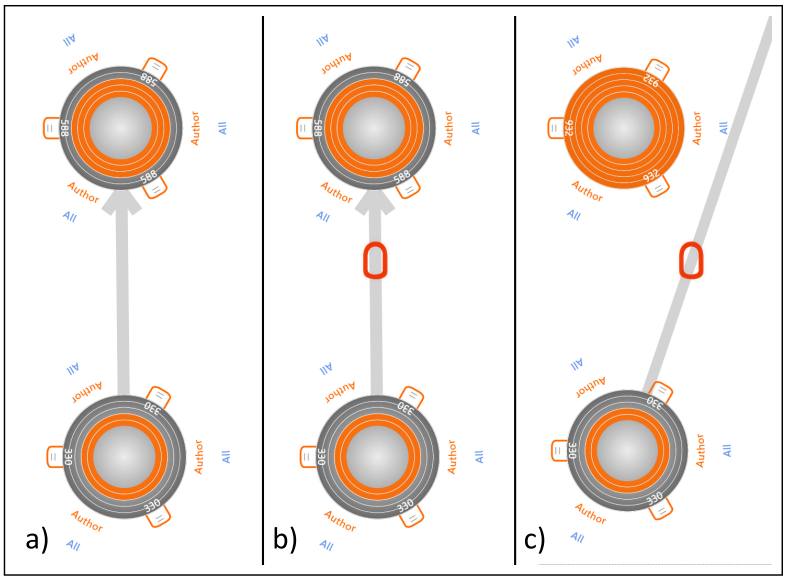

Figure 3: The `ReleaseLink` behavior of Facet-Streams

In the naive implementation of this behavior (see listing 3 on the following page), a timer has to be started in the `TouchDown` event handler (1). If the timer expires, the link can be released in the event handler of the timer (2). As the `ReleaseLink` method needs access to UI elements, it has to be executed in the `Dispatcher`. If the `TouchUp` event occurs prior to the expiration of the timer, it needs to be stopped and the link remains connected (3). In this implementation, a fair amount of timer management code has to be written to realize the timing constraint and the functionality is again distributed over several event handler methods.

```
private Timer _timer;

void OnTouchDown(object sender, TouchEventArgs e)
{
    //(1) start the timer
    _timer = new Timer(1000);
    _timer.Tick += OnTimerExpired;
    _timer.Start();
}

void OnTimerExpired(object sender, EventArgs e)
{
    //(2) release the link in the execution context of the Dispatcher
    Dispatcher.Invoke(() =>ReleaseLink());
}

void OnTouchUp(object sender, TouchEventArgs e)
{
    //(3) Stop the timer
    _timer.Stop();
}
```

Listing 3: Implementation of the `ReleaseLink` behavior with a timer

2.2.4 Summary

Distribution of functionality A salient characteristic of all presented examples is that functionality is distributed over several different places. The reason for this is that each input event is usually handled by a separate event handler method. Timers also contribute to this distribution in that their management is realized in several different places.

Explicit preservation of state A consequence of the distribution of functionality is that state has to be preserved between the distributed parts. Each situation that an event handler generates has to be marked explicitly. The next event handler then has to test against this situation to react appropriately. While all of this is possible with default programming language constructs and default data structures, a lot of decision logic code has to be written to ensure that all special cases are considered. The provided examples were also quite simple. Many complex interaction techniques require more state to be saved and involve a lot more events.

Abstraction gap The general problem with all presented examples is that the actual interaction is not represented in a way that closely resembles the mental model. What can be described in a single fact in natural language often has to be described with multiple distributed facts in the

programming language. For the developer it would be ideal, if he could directly translate his natural language specification into a similar expression in the implementation. This is however only possible if both descriptions represent the problem on the same level of abstraction. As the abstraction level of the natural language is far higher than that of the programming language, a gap remains which has to be bridged by the developer (see figure 4). Without any doubt, such a gap has always existed since the first interactive systems have been built. Yet, while the possible ways of interaction have been exploding in the last few years, the progress of programming languages is not taking place at the same speed.

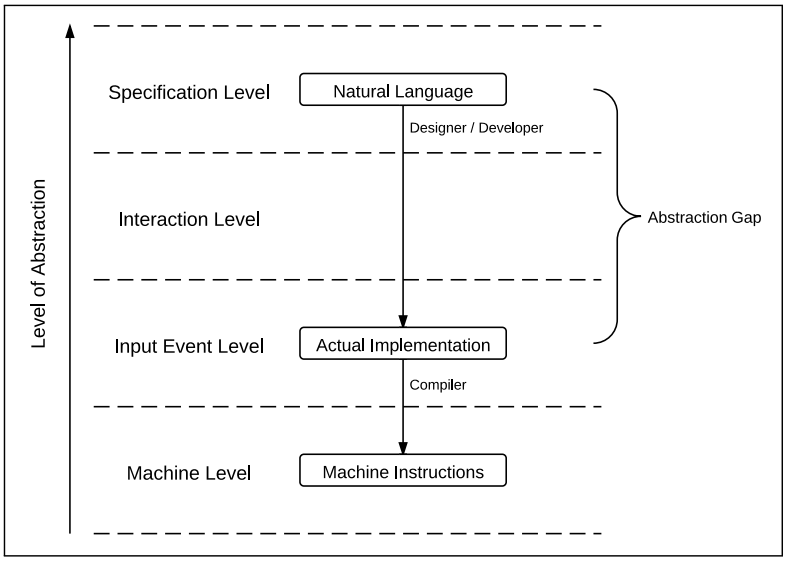

Figure 4: A huge gap exists between natural language description and implementation

To demonstrate how the implementation of an interactive behavior can be improved significantly when its level of abstraction is closer to that of the specification, the `ReleaseLink` behavior is addressed two more times in this thesis. One implementation is realized with the Rx library (see subsection 5.3.2 on page 78), while the other implementation is based on a finite-state machine and realized with the Reactive State Machine framework (see subsection 9.2 on page 116).

2.3 Formal Methods

Because of its inherent complexity, post-WIMP interaction has to be designed and specified thoroughly and systematically beforehand. A 2008 study by Myers et al. revealed that designers and programmers of user interfaces see a lack in tools that support the description and definition of interactive behaviors [Myers et al., 2008]. While according to Park et al. they tend to describe interactive behaviors with similar expressions and similar language constructs [Park et al., 2008],

there is still no common language to talk about interaction in all facets. Yet, especially with post-WIMP systems, which are highly interactive, it is important that all stakeholders of a project can communicate about the system using the same language. Since the early days of interactive computing, researchers advocated the use of formal methods to describe and specify the interactive behavior of systems (see for example [Newman, 1968; Parnas, 1969]). Despite being more mathematical than natural language, there is strong support in literature for the application of formal methods to describe interaction. In the following, I want to present a number of arguments in favor of formal methods, but also discuss some points of criticism. For a thorough investigation of formal methods in HCI, I recommend the various papers of Alan Dix ([Dix, 1991, 1993, 1995; Dix and Abowd, 1996; Dix, 2002, 2003; Dix et al., 2008]). Being one of the main advocates in this field, he defines formal methods as following: "Taken strongly, formalism in mathematics and computing is about being able to represent things in such a way that the representation can be analysed and manipulated without regard to the meaning. This is because the representation is chosen to encapsulate faithfully the significant features of the meaning" [Dix, 2003].

2.3.1 Advantages of Formal Methods

An important advantage of formal methods is that they are able to bridge the aforementioned abstraction gap in that they are able to describe interactive or dynamic behaviors. Thereby a formal notation requires the designer to be precise, while a natural language description often omits certain details. This automatically forces us into thinking about a problem more concisely, eventually leading to the consideration of issues that would otherwise be missed [Dix, 2003]. For example, potential ambiguities in the interaction can be resolved very early in the design process. Accot et al. consider this very important for specifying multi-modal interfaces, where parallel event flows "can lead to unpredictable incoherent situations in the same way as in multitasking concurrent programming systems" [Accot et al., 1996]. As thinking about an interaction problem usually happens while creating a formal model, insight is rather gained from the process than from the result.

While a formal notation helps to "sketch the details of interaction" [Dix, 2003], it also helps to abstract away unnecessary details. Abstraction is an equally important feature as precision, as too much detail often impairs comprehensibility, which Wasserman deems a key requirement for specifying interactive systems [Wasserman, 1985].

A system or behavior described by a formal method also allows formal analysis which means that the structure of the description can be analysed without knowledge of the meaning. This is not only useful to detect errors in the system itself, but also to "assess potential usability measures or problems" [Dix, 2003]. It can also save a lot of time, as the functionality of the system can be analysed before it is build [Dix, 2003] and making changes in this early stage is still safe and easy [Dix, 2002]. With formal analysis it is also possible to say whether or not a real system satisfies its

description [Dix, 1995].

Many formal notations are in fact rather graphical than textual. This facilitates the understanding of the formalism, as graphical representations are in 2D and thus have greater expressivity than the 1D representation of text or mathematical formulae. Thus, the use of a graphical representation can immediately prompt "a series of questions"[Dix, 2003]. Dix also states that a graphical representation per se can be formal, "depending on the meaning which is attached to the elements of the diagram", and that the structure of a diagram is "capable of formal analysis" [Dix, 1995].

By using formal methods the description of an interactive system is externalized. Externalization is an important support mechanism for our cognitive system. By outsourcing all parts of the interactive system to a permanent medium, the workload of the short-term memory is reduced drastically, which allows a designer to concentrate on certain aspects in greater detail. A more detailed description of a system in turn facilitates the designer-developer workflow, as a developer may implement the system exactly as the designer intended.

2.3.2 Criticism

Along with the advantages I listed above, there are some points of criticism that are raised against the application of formal methods. Myers for example argues that formal methods have a high threshold for developers, because they require programmers to learn new notations and languages and even new concepts [Myers et al., 2000]. While this is certainly true, it is always necessary for professional developers to learn new things to keep pace in this fast changing field. Yet, even Dix states that formal methods sometimes require "quite a high level of mathematical sophistication" and are thus "hard to 'give away' to the practitioner" [Dix, 1993]. An expert in the field of formal methods may be able to focus on the critical areas of a system, while the "proficient practitioner may become lost in the morass of detail" [Dix, 1993]. Even if a formal description is not mathematical, it may become complex structurally: "Formal descriptions, by making you be precise, can become complex through sheer level of detail" [Dix, 2003]. This has always been one of the most important points of criticism, especially for the state-transition diagram notation of finite-state machines. Yet, there usually exist methods that allow to reduce the complexity of a specific notation. The problem with complexity in general is that it is often inherent to the process or behavior we want to describe. It is thus no wonder that the resulting description is complex, too: "Whenever we capture the complexity of the real world in formal structures, whether language, social structures or computer systems, we are creating discrete tokens for continuous and fluid phenomena. In so doing we are bound to have difficulty. However, it is only in doing these things that we can come to understand, have valid discourse and design."[Dix, 2003]. Thus, a complex description of a system often means that there is inherent complexity to the whole system. It is therefore better to have complex behaviors documented in a formal description, than to just have them in code; they will be in the system anyway [Dix, 2003].

Despite the criticism, formal methods have successfully sneaked into the toolbox of developers in the recent years. A popular example is the Unified Modeling Language (UML) with its various diagrams and notations. Many of them have their roots in some formal notation, and yet they are just normal nowadays for the average software engineer. For Dix this comes without surprise: "[...] it is precisely because systems are large and complex that more formal notations are used. It is easy for programmers to get the details right, but they need support in understanding interactions and architecture" [Dix, 2003].

2.4 Finite-State Machines

Finite-state machines have been suggested as a reasonable formalism to specify interactive behavior ever since. The first to do so was William Newman in 1968 who used them to describe the behavior of graphical systems [Newman, 1968]. Newman correctly observed that in an interactive system, "the same action may cause a different reaction on different occasions" [Newman, 1968]. He therefore suggested the use of finite-state machines, as these are able to model both input and output of an interactive system depending on the state it is in. Since then, several extensions have been suggested for the default notation of finite-state machines, the state-transition diagram (e.g. [Parnas, 1969], [Wasserman, 1985], [Harel, 1987]). The most influential of these was the *Statecharts* notation of Harel which eventually became the basis of the UML State Diagram [Harel, 1987]. While these early works stem from an era where the interaction with the system was very limited, researchers in recent years also strongly suggested the use of finite-state machines for the description of interactive behaviors in post-WIMP systems (see for example [Thimbleby, 2007; Appert and Beaudouin-Lafon, 2008; de Haan and Post, 2009]).

While I strongly support the application of finite-state machines in the context of post-WIMP systems, it is certainly fair to point out that they are but one technique among many others. Although they are without any doubt very useful in specific scenarios, I can not imagine a whole user interface being built around finite-state machines alone. This is backed by Hudson, who states that "controlling a complete dialog on the basis of a state machine can have some significant drawbacks (such as promoting the use of modes)" [Hudson and Newell, 1992]. Having said that, I nonetheless want to make the case for state machines here as they are very useful for "controlling actions at a fine grain such as at the interaction technique level" [Hudson and Newell, 1992]. In the following, I therefore want to point out for which specific user interface and interaction problems of post-WIMP systems state machines are an appropriate means.

2.4.1 States in the User Interface

Every user interface element typically contains a number of different groups of states, such as a group of states that denotes its *Visibility* (Visible, Collapsed, Hidden) or a group of states that denotes if it is enabled or not (Disabled, Enabled). The different states of one group are typically differentiated graphically. For example, an element that is grayed out indicates that it is in the *Disabled* state. While finite-state machines are typically used to express the interactive behavior of a system, they can also be employed to express the graphical differences between individual states of a UI element. This is however not seen extensively in practice. The reason for this is probably the fact that it is currently significantly more complicated in most user interface toolkits to build a finite-state machine than it is to use some sort of decision logic to differentiate between a rather small set of states. There are however notable exceptions where finite-state machines are used to drive the different states of a UI element. These exceptions are the user interface elements that ship with Microsoft's WPF (Windows Presentation Foundation) and Silverlight. User interface elements of these toolkits leverage the Visual State Manager (VSM) to define multiple state groups and different states per state group. The VSM, which is presented in greater detail in section 6 on page 79, not only considers the individual states, but also the potential transitions between those states. These transitions are defined as animations which make it possible to smoothly change the visual properties of the respective UI element instead of setting them discretely.

In addition to the states that individual UI elements can attain, it is also the case that states are used in a global way that affects the entire user interface. These global states are often referred to as *modes*. They partition the entire user interface into distinct parts in which the look and feel differs substantially. While such modes are also used in post-WIMP systems (the Facet-Streams system for example differentiates between a Query Mode and a Result Mode), they are considered to be bad practice and the source of errors (see [Raskin, 2000] for a deep coverage of modes). For post-WIMP systems, the most important disadvantage of modes is that they introduce global functionality which is very problematic with multiple users (as discussed previously in subsection 2.1.2 on page 24). While finite-state machines are certainly the appropriate means to design and implement such global states or modes, it should first be investigated if the system could potentially do completely without them.

2.4.2 States in the Interaction

In addition to these differentiations of user interface states, finite-state machines are usually considered to be the ideal formalism to model the interactive behaviors of a system. It has been shown previously in subsection 2.2 on page 29 that higher-level interaction, which consists of multiple subsequent or parallel occurrences of input events, is not straightforward to realize with naive implementation techniques. Typical issues that arise are the need to preserve state between a dis-

tributed set of event handler methods and the management of timers to satisfy timing constraints. With finite-state machines these issues do not arise as they are able to model interactive behaviors on a higher-level of abstraction. Table 2 shows the main differences between naive implementation techniques and finite-state machines. While the functionality in naive implementation techniques is distributed over several event handler methods, a finite-state machine is self-contained and can be expressed in a coherent graphical model. This is backed by Appert and Beaudouin-Lafon who point out that state machines "reify the notion of interaction, which is otherwise represented by a disconnected set of event handlers" [Appert and Beaudouin-Lafon, 2008]. Also, the preservation of state over the course of a sequence of input events is inherently supported by finite-state machines. While naive implementation techniques require this state to be stored explicitly inside dedicated fields, finite-state machines implicitly store it inside their structure. Thereby each state automatically creates a certain situation, which permits or prevents certain follow-up situations. In a naive implementation technique, this situation has to be established manually, which requires the use of decision logic to determine the reaction to an event. Finite-state machines also help with the specification of timing constraints. While naive implementation techniques require an explicit management of timers (starting, stopping, ticking), timing constraints can be expressed naturally with finite-state machines by using timed transitions. The general advantage of finite-state machines is that their level of abstraction is significantly higher than that of naive implementation techniques. Thus, only few facts are needed with finite-state machines to describe one fact of the natural language description, while multiple facts are needed with naive implementation techniques.

	Naive implementation	**Finite-State Machine**
Distribution	distributed over several event handler methods	self-contained
Preserving State	explicit (in fields)	implicit (in structure)
Reaction to an event	determined by decision logic	determined by current state
Time Constraints	explicit (with timers)	implicit (timed transitions)
Level of Abstraction	low (one fact in specification represented by multiple facts)	high (one fact in specification represented by one or few facts)

Table 2: Comparison of naive implementation techniques and finite-state machines

While the above comparison showed that finite-state machines can express interaction on a higher level of abstraction, they are only a (graphical) formalism and eventually have to be transformed to code. Thus, the abstraction gap is only relocated and the responsibility is delegated to the respective FSM implementation technique (see figure 5 on the following page). In section 4 on page 54, it is shown how this abstraction gap can be bridged with an appropriate implementation technique.

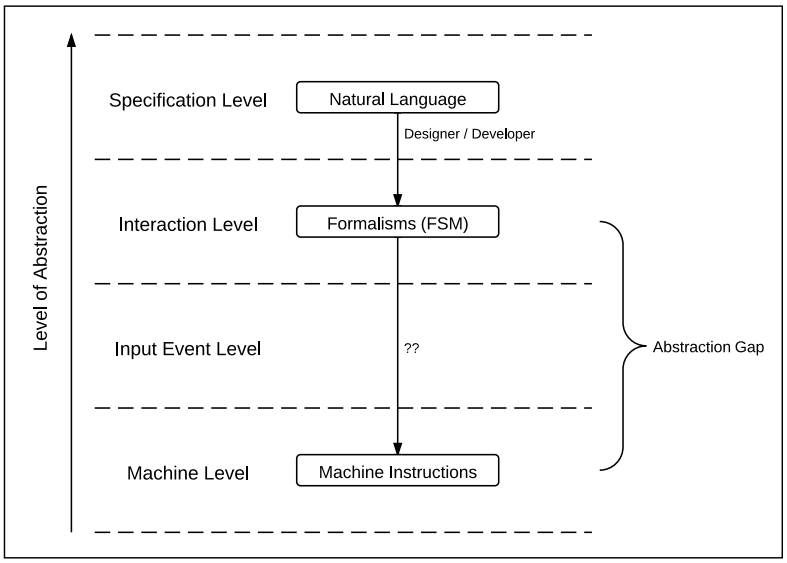

Figure 5: Finite-State Machines represent interaction at a higher level of abstraction

Finite-State Machines vs. Petri Nets In theory, a potential drawback of finite-state machines is their inability to represent real concurrent activities. Palanque thus makes a strong statement against finite-state machines and in favor of Petri nets [Palanque et al., 2011]. As he argues, Petri nets are one of the few formal description techniques that are able to represent concurrent interactive behavior: "they feature true-concurrency semantics, they are able to deal both with events and states and they provide several ways to represent quantitative time, [...], they represent state in a concise way as states are represented in intention and not in extension (as in state machines for instance) thus avoiding combinatory explosion when combining several devices" [Palanque et al., 2011]. While his objections are certainly correct, two arguments still speak for the application of finite-state machines in practice:

First, in many user interface toolkits there is no real concurrency, only perceived concurrency. In WPF for example, every operation that affects user interface elements must be delegated to the `Dispatcher`, a priority queue that posts these operation to a dedicated thread. Events from default input devices are always fed to the `Dispatcher` and as a consequence become sequenced. While input devices that are integrated into the application by third party libraries do often fire their events in a separate thread, these events have to be fed to the `Dispatcher` eventually, if they are used to drive the interaction of the user interface. This ultimately means that when dealing with interactive behaviors in (graphical) user interfaces concurrency is not an issue, although a user perceives his interaction as being parallel. As a consequence, many of the perceived parallel interactions can in fact be modeled with formalisms that do not specifically consider concurrency.

Second, Petri nets have a very high threshold. In the previous subsection, formal methods were

criticized for being too mathematical and therefore being too advanced for many developers. Petri nets are a great deal more complex than finite-state machines, which means that the threshold for average developers is far too high. This is backed by Appert and Beaudouin-Lafon who state that their "learning curve is steep, making the adoption of such a model by developers difficult" [Appert and Beaudouin-Lafon, 2008]. I therefore doubt that Petri nets will ever have much practical relevance for the development of post-WIMP interaction beyond the research community.

Part III

Finite State Machines

> Power ceases in the instant of repose; it resides in the moment of transition from a past to a new state [...]
>
> Ralph Waldo Emerson

In this part of the thesis, finite-state machines are advanced further. While the first section addresses different features and notations of finite-state machines, the subsequent section deals with their implementation. This whole part is based on a seminar paper of the author, in which a state-of-the-art analysis of the application of finite-state machines in post-WIMP user interfaces has been conducted [Zöllner, 2011b].

3 Features and Notation

Ever since finite-state machines have been suggested for the specification of interactive systems by William Newman in 1968, a lot of extensions have been proposed to their default notation, the state-transition diagram. Some of them were minor notational changes to improve the visual appearance, but others introduced new semantics into the notation to make it more expressive. In the aforementioned seminar paper, I gave an overview over these different suggestions researchers have made over the last decades [Zöllner, 2011b]. There, all features and notational elements are explained in greater detail. As I do not want to reproduce the findings of this seminar paper, I only address those notational elements in the following, which are also implemented in the Reactive State Machine framework and which I thus deem important for a wide range of post-WIMP interaction scenarios. Of course, all default notational elements of state-transition diagrams, such as states (see subsection 3.1), triggers, and transitions (see subsection 3.2 on the next page) are considered. In addition to that, I also introduce new notational elements to deal with animations (see subsection 1.2.4 on page 18) and multiple input points (see subsection 3.4 on page 48), two important characteristics of post-WIMP systems that have not yet been considered in any notation. Mainly for implementational reasons, some of the more advanced features of the Statecharts notation have not (yet) made it into the current implementation of the Reactive State Machine framework and are therefore only addressed briefly in subsection 3.5 on page 51.

3.1 Modeling States

States are the basic elements of finite-state machines. They represent a certain situation in the system [Hopcroft et al., 2001] in that they materialize the flow of input events that have happened before [Zöllner, 2011b]. As in the Statecharts notation, I suggest to draw states as rounded rectangles. This provides more space to include *entry* and *exit actions* inside the state. These special actions, which are executed when the state is entered or exited, are written inside the state rectangle and are prefixed with either of the keywords *entry* or *exit*. By default, *entry* and *exit actions* are unconditional, which means that they are always executed when the state is entered or exited. In certain scenarios it may however be beneficial to restrict their execution. I therefore suggest two further notational elements which can be used to set guards for *entry* and *exit actions*. The first element is used to restrict the execution of an action to situations where the source (or target) of a transition corresponds to a given reference state. For example, to express that an *entry action* may only execute if the state has been entered via some transition starting in state `X`, the `entry` prefix is changed to `entry [From=X]`. To express that an *exit action* may only execute if the state is exited via some transition to state `Y`, the `exit` prefix is changed to `exit [To=Y]`. Of course, the same behavior could be expressed by putting the operation of the *entry* or *exit action*

directly into the transition action of the respective transition. With these guards it is however possible to specify the action only once, instead of specifying it separately for every transition. Figure 6 highlights this difference.

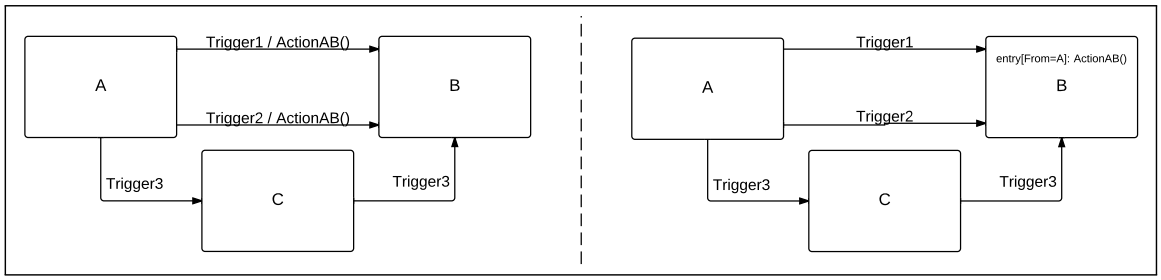

Figure 6: Specifying an *entry* guard (right) instead of multiple transition actions (left)

The second notational element that is introduced to constrain *entry* and *exit actions* is a general purpose guard, which behaves similar like a guard of a transition. Within square brackets a condition can be specified for each *entry* or *exit action*. If that condition is met, the action is executed, otherwise it is not. Note, that the state is always entered or exited, it is just the action that is guarded by the condition. Figure 7 shows how such a condition can be used.

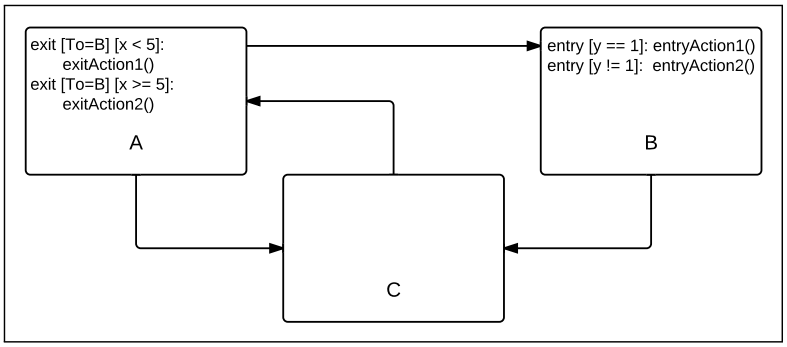

Figure 7: Specifying general purpose guards for *entry* and *exit actions*

Of course, it is also possible to move the conditions directly inside the respective action, instead of specifying them publicly in the notation. Yet, this notation makes conditional behavior explicit and visible, whereas otherwise it is implicit and hidden in the implementation.

3.2 Modeling Transitions

Transitions are the elements that introduce interactivity into state machines. They make it possible to change the currently active state of the state machine by reacting to external triggers. Each

transition is minimally a 3-tuple and maximally a 5-tuple consisting of the required elements *source state*, *target state*, and *trigger* and the optional elements *guard* and *transition action*. In the notation, the transition is represented as an arrow going from *source state* to *target state*. The *trigger* is written on top of the arrow, followed by the optional *guard* in square brackets and the *transition action* separated by a slash (/). Transitions can also be internal, which means that both *source state* and *target state* are identical. Figure 8 shows both the representation of a transition (top) and its execution flow (bottom).

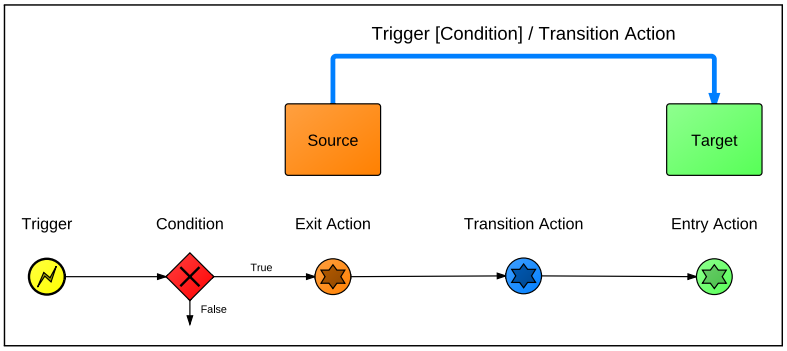

Figure 8: Notation of a transition (top) and its execution flow (bottom)

A particular transition is initiated, once its *trigger* fires. While triggers in the case of post-WIMP interaction are usually input device events, they can actually be any kind of external signaling mechanism. To differentiate transitions that react to such external signaling mechanisms from other transitions that are introduced below, I call them *triggered transitions* from now on. After the *trigger* of the *triggered transition* has fired, an optional *guard*, which is represented by a boolean condition, is evaluated. If the *guard* evaluates to `false`, the transition is not made. If it evaluates to `true`, the transition is started. The designer of a state machine has to be careful that multiple transitions originating in the same *source state* and initiated by the same *trigger* have guards that are mutual exclusive. Otherwise non-determinism is introduced into the state machine as more than one transition may be considered for execution. Once the transition is started, the first operation is to exit the *source state*. This is where the optional *exit actions* of this state are executed. Next, the optional *transition action* of the transition is executed. Then, the *target state* is entered, which is where its optional *entry actions* are executed. If the transition is internal, the state is neither exited nor entered and no *entry* or *exit action* is executed.

In addition to *triggered transitions*, there are also transitions that rely on a *timer* instead of an external *trigger*. This timer is started as soon as the state is entered, restarted when an internal transition is performed, and stopped when the state is exited prior to its expiration. When the timer fires, the transition is initiated. Such *timed transitions* are modeled by writing the keyword *After xx seconds* onto the respective arrow. Timed transition are important for all interactive behaviors that have a time component, such as the aforementioned tap gesture or for situations where the

user interface has to be reset after a given time of inactivity. Figure 9 shows how triggered and timed transition are used to realize the activation and deactivation of a control. Here, the control is activated (i.e. made visible) by the `TouchDown` event. It stays active (visible) after the user lifted his finger. Only after two additional seconds, it is eventually deactivated (i.e. invisible).

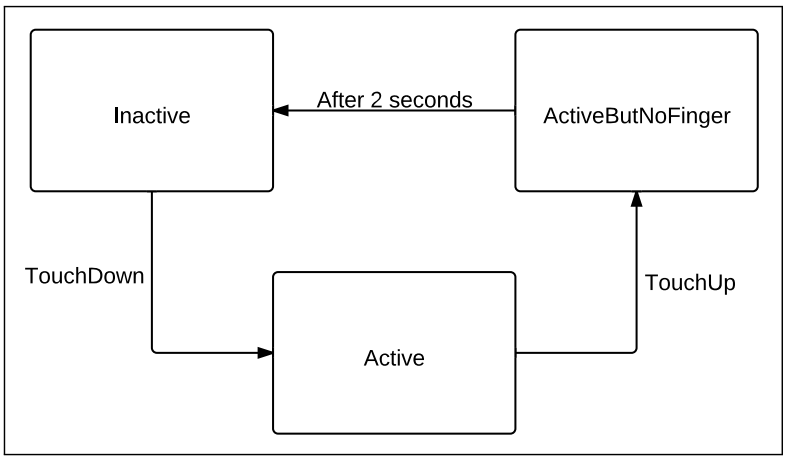

Figure 9: Triggered and timed transitions are used to model the activation and deactivation of a control

3.3 Modeling Animations

Animations are an important characteristic of post-WIMP systems. They can be used to smoothly alter the properties of user interface elements while transitioning from one state to another. Yet, to this day, animations, or *animated transitions* as they are called from now on, are not part of any state machine notation so far. The reason for this certainly has to do with the fact that animated transitions have a time component attached, whereas normal transitions of a state machine are thought to be instantaneous. Instantaneous in this case means that these transitions are atomic and can not be interleaved by any other operation. This is to prevent the concurrent initiation of other transitions during the execution phase of one transition. If *animated transitions* were also designed to be atomic, they would lock the state machine during their execution, thereby making it impossible for other triggers to change the current state. This is problematic as a user does not always want to wait until an *animated transition* finishes: "While the animation should be designed so that the user will not feel as if he is waiting for it to play out (it should be both fast enough and engaging enough so that the user isn't consciously aware of it), the user may want to move to the next interaction before an animation finishes" [Chang and Ungar, 1993]. Thus, in stark contrast to *triggered* or *timed transitions*, it must be assured that an *animated transition* can be cancelled inbetween and that the state machine then transitions to another state. Take for

example an opacity animation that smoothly fades in a user interface element. If a user decides inbetween that he does not want the user interface element to appear, it must be possible to cancel the animation and to revert the control back to the previous state (ideally using the reverse animation). To model this cancellation behavior, the state machine has to reside in a valid state during the animated transition, as a trigger used for cancelation can only cause a transition if the current state of the state machine is defined. If we wanted to model animated transition without major changes to the transition notation, one of the two options a) or b) of figure 10 would have to be applied.

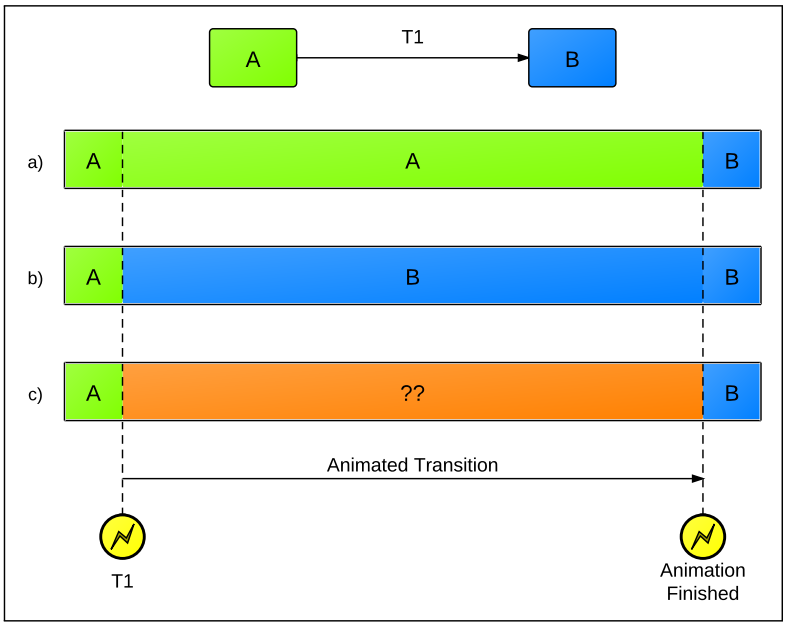

Figure 10: Possible concepts for animated transitions without changes to the default notation

With option a), the animation is started immediately after the trigger T1 fired. The state machine stays in state A until the animation is finished, then it transitions to state B. With option b), the state machine immediately transitions to state B after the trigger T1 fired. The animation is started thereafter and is executed while state B is active. Unfortunately, neither option is able to unambiguously specify what is to happen, when a trigger is fired during the animation. In both cases, the transition that is caused from a cancellation trigger has to be modeled with either state A or state B as starting point, since the state machine is currently residing in one of these. But then the same trigger would also cause this transition, if the state machine was just idling in this state, without having an animated transition running. Conceptually these are two separate cases: Take for example a button that can be pressed by touching it. If that button is about to disappear due to an animated transition and the user touches it during this fade-out animation, what is the desired behavior ? Most likely we do not want the button to be pressed, when it is barely visible, but rather want the animation to be reversed and have the button faded back to full opacity. Yet, when the

animation is not running we clearly want the button to be pressed by the same trigger. Thus, one trigger would cause two different reactions in this state, depending upon if the animation is running or not. This clearly indicates that animated transitions are in their own right some kind of state and that this fact can neither be expressed sufficiently by case a) nor case b).

Conceptually we want to have the situation of case c). Here, the state machine immediately transitions to some undefined intermediate state after the trigger T1 has fired and resides therein while the animation is running. Once the animation is finished, the state machine transitions to state B. Yet, leaving the state machine in some undefined state during the transition is not an option either, as we can not specify what is to happen if a trigger is fired during the transition. In order to do that, we have to materialize the undefined state and model it explicitly. Figure 11 shows how the definition of the state machine has to be altered to achieve this.

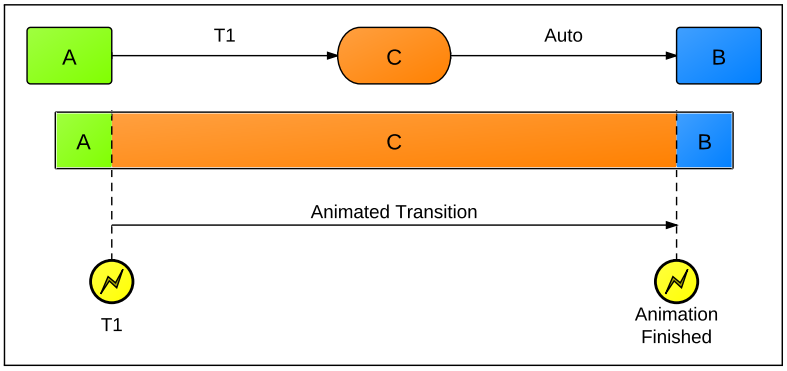

Figure 11: Animated transition with a materialized transitioning state

Note, that the shape of the transitioning state C is different to the shapes of the other states to indicate that this state is associated with an animation. After the trigger T1 has fired, the state machine transitions from state A to the transitioning state C, where it resides during the animation. After the animation has finished, the state machine transitions to state B. The reader may have noticed, that the arrow from state C to B is labeled with *Auto*. This indicates a special kind of transition which I call *automatic transition*. *Automatic transitions* are triggered as soon as all animations of the current state have finished and thereby make it possible to automatically leave the current transitioning state. Like all other transitions, *automatic transitions* can have optional guards and transition actions which are written on the arrow.

With this design it is ultimately possible to specify triggers that can cancel the current animation and thereby transition the state machine to another state. Figure 12 on the next page shows how the previous state machine, which was used to activate and deactivate a control, can be augmented with animated transitions. As can be seen from this model, it is also possible to transition from one transitioning state (*Fading In*, *Fading Out*) to another transitioning state (*Fading Out*, *Fading In*).

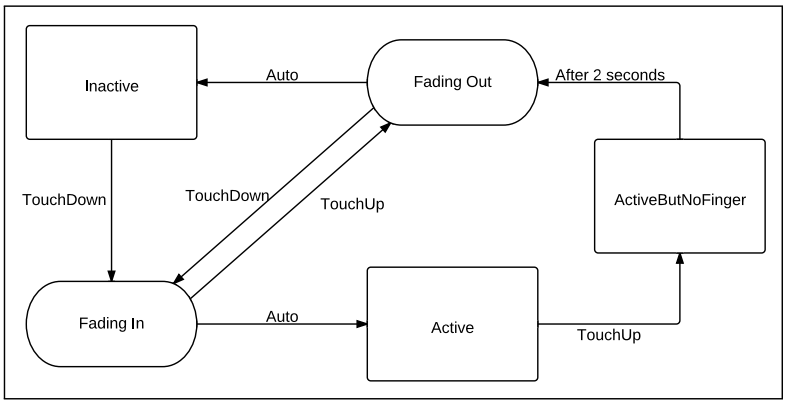

Figure 12: Activation and deactivation of a control with fade-in and fade-out animations

The advantage of this approach is that the designer of the state machine gains expressivity. He can now express clearly which transitions are animated and therefore need a transitioning state inbetween and how these transitioning phases have to react to potential triggers. Without the materialization of the animated transition this would not be possible.

As it turns out, the current concept of modeling animated transitions is in fact a combination of two features that were previously suggested by researchers for different purposes. The first feature are the *activities* of the Statecharts notation. These are operations that are associated with a state and take "nonzero amounts of time" [Harel, 1987]. To control the behavior of these activities, the Statecharts notation provides *start()* and *stop()* operators that can be used inside *entry* or *exit actions*. These activities are similar to animations which in the current concept also reside inside states and also take nonzero amounts of time. The other feature are *system actions* which were introduced by William Newman [Newman, 1968]. While these actions do not resemble longer running operations, they are capable of branching, which means that once they are ready, they transition the state machine to the next state. Unfortunately, Newman did not mention any notational element to visualize system actions. In the current concept, the branching mechanism of system actions is resembled by *automatic transitions* which also transition the state machine to the next state once the animation is ready. Thus, *animated transitions* are in some sense a combination of the *activities* of the Statecharts notation and the branching capabilities of *system actions*.

3.4 Modeling Multiple Input Points

The triggers of single-point input devices can be specified unambiguously in all FSM notations. Yet, when it comes to input devices that feature multiple input points, such as multi-touch table-

tops, there does not exist a dedicated notation that allows to differentiate between these different points.

There are two types of characteristics that can be used to differentiate input points: *explicit* and *implicit* characteristics. Explicit are those characteristics, that are contained in the event metadata of an input point, such as its location or the input element that is affected by the point. Such explicit information can be expressed with existing elements of the default FSM notation. For example, to select all input points that affect a specific user interface element, the name of the user interface element is added to the trigger with the keyword `on`, e.g. `TouchDown on Button A`. For more detailed differentiation, a *guard* can be used, e.g. `TouchMove[Position.X >= 100 && Position.X <= 200]`. Because the relevant information is contained in the event metadata, these explicit characteristics can also be implemented straightforwardly.

More difficult are implicit characteristics, such as relative timing information (i.e. the order of occurrence) or the total number of input points that currently affect a user interface element. They are often needed to realize more complex interactive behaviors, such as that a user interface element may only disappear when the last finger has left the element, or that the first input point on a user interface element has different effects than input points that occur later. Such relations are more difficult to express and implement as the required information is not contained explicitly in the event metadata, but has to be calculated on the fly or maintained in a data structure. Ideally, a hypothetical notation is able to comprehensibly model these implicit characteristics and can also be implemented straightforwardly. During the development of the Facet-Streams system we were forced to come up with such a notation, as we wanted to use a state machine to model some complex multi-touch behaviors which required relative timing information and the number of all input points that currently affect the element. At first, this notation was very much targeted at the particular scenarios of Facet-Streams. For this thesis, however, I tried to generalize it and thereby make it more comprehensible for other developers.

The notation is based on the concept of collections. Every input point that begins to affect a user interface element is thought to be added to a collection. It stays in this collection until it no longer affects the user interface element. The collection is sorted temporally in ascending order, which means that the oldest input point is the first one in the collection and the latest input point is the last one. Similar to the LINQ query mechanism of .NET collections, there are several operators in the notation that can be used to *query* the input point collection. To identify these operators, they are prefixed with a hash (#) sign. Currently, three types of operators exist: positional operators, the `#Contains` operator and the `#Count` operator. The positional operators check if the current input point matches a certain position inside the collection, the `#Contains` operator checks if the current point is contained in the collection and the `#Count` operator returns the overall count of input points in the collection. In order to avoid the introduction of additional syntactic elements into the notation, the operators can be used inside the conditional expression of a guard. The

positional operators thereby return a boolean value, which indicates if the current input point is at the respective position in the collection. The `#Contains` operator also returns a boolean value, and the `#Count` operator can be used for comparisons in a conditional expression. The following list presents all operators that are currently available:

#First Checks if the current point is the first or oldest point in the collection

#Initial Checks if the current point is the initial first point of the collection. Whereas the initial first point is always the first point, the current first point is not necessarily the initial first point, as the initial first point could have been removed already and thereby the subsequent point became the new first point. The initial first point is updated, when the collection was completely empty and a new point is added.

#Intermediate Checks if the current point is an intermediate point, which are all points that are neither first nor last

#Subsequent Checks if the current point is not the first point

#Last Checks it the current point is the last or newest point

#x Checks if the current point is the point at position x in the collection

#Contains Checks if the current point is contained in the collection

#Count Returns the number of input points that are currently in the collection

To show how the notation can be used in practice, I want to employ a simplified example of the Facet-Streams system below: the closing of the wheel. This example is detailed further in subsection 9.1 on page 103.

Closing the Wheel In Facet-Streams, a circular control, the *wheel*, is used to select a range of values. This wheel is opened by touching a special label with the finger. It is meant to stay open, if at least one finger is touching it. As soon as the last finger leaves the wheel, it begins to fade out gradually. If we wanted to express this behavior, we have to differentiate two different cases: (1) The `TouchUp` event occurs, but other fingers are still on the wheel. In this case, the wheel stays in the current state. (2) The `TouchUp` event occurs, but no other finger is left on the wheel. In this case, the wheel transitions to the *Fading Out* state. Figure 13 on the next page shows the small subset of the state machine that represents this behavior.

The `#Count` operator is used to tell the two cases apart. Note, that the comparison operand in both cases is not zero, but one. The reason for this is that the input point is only removed from the collection after all conditions have been checked. If it had been removed previously, it would not be possible to use the position operators to check the current point. The count therefore represents

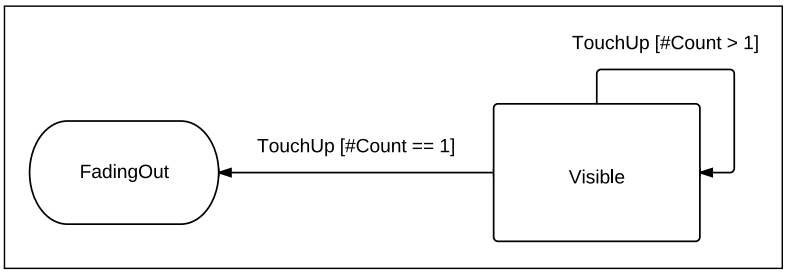

Figure 13: Usage of the `#Count` operator in Facet-Streams

the number of input points in the collection before the current input point was removed. Note, that the conditions are also evaluated before an input point is about to be added to the collection. While this has to be considered when using the `#Count` operator, the use of the position operators in this case is pointless, as the newly added point is always the newest.

3.5 Omitted Statecharts Concepts

The Statecharts notation features the three powerful concepts of hierarchy, orthogonality and broadcast communication to simplify complex state machines.

Hierarchy The concept of hierarchy introduces abstraction into state machines, by allowing the clustering of multiple states into a super-state (see figure 14 on the following page) and the refinement of a state into multiple sub-states (see figure 15 on the next page). By doing so, states can either be treated as black boxes or inspected more thoroughly. This facilitates the organization of a state machine, as the designer can concentrate specifically on certain aspects, while ignoring others. Eventually, the whole state machine is more readable and comprehensible. Unfortunately, the current implementation of the Reactive State Machine makes it impossible to model the concept of hierarchy. The reasons for this are explained in more detail in section 7 on page 84. As the use of hierarchy is only necessary if state machines grow too complex, it is not too much of an impairment for the definition of most interactive behaviors. This is backed by Appert and Beaudouin-Lafon who state that they have "not found compelling examples yet of its use in user interfaces" [Appert and Beaudouin-Lafon, 2008]. They also point out, that the state explosion "is not an issue when the state machine describes a single interaction" [Appert and Beaudouin-Lafon, 2008].

Orthogonality The concept of orthogonality deals with the fact, that two groups of states can be separated from one another if they are orthogonal or mutually exclusive. Consider for example

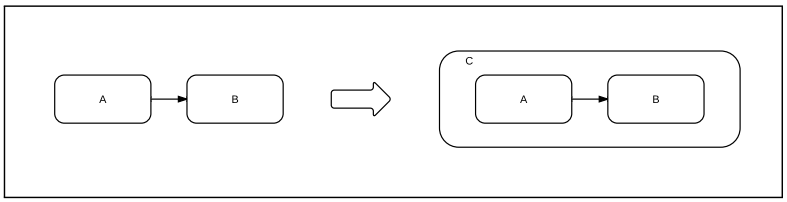

Figure 14: Clustering multiple states into a super-state

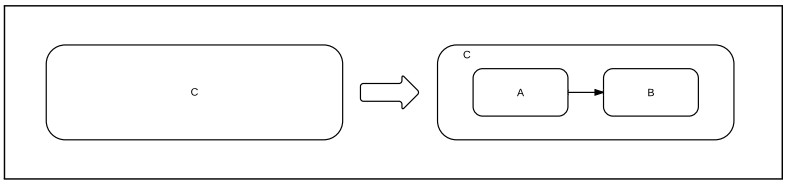

Figure 15: Refining a state by introducing sub-states

a user interface element which contains a group of states to indicate if it is enabled and a group of states to indicate if it is focused. Both groups of states are orthogonal, which means that they do not depend on each other. At each point in time, the UI element resides in some state of each state group. Instead of modeling these orthogonal groups of states as combinations in one huge state machine, the Statecharts notation suggests to separate the state groups into two independent small state machines to reduce the overall complexity (see figure 16 on the next page). Thereby concurrency is introduced, as the two small state machines are acting individually. In terms of notation, these different state machines are encapsulated in a higher level state (which requires the concept of hierarchy) and separated by a dashed line. As the Reactive State Machine does not currently support hierarchical states, it is also not possible to use the concept of orthogonality inside a single state machine. Yet, it is possible to express simple orthogonality in a different way. If two groups of states are orthogonal, they can as well be expressed by two separate state machines without being clustered inside another state. Thus, it is possible to simply use `n` state machines to model `n` orthogonal groups of states. This however only works if the orthogonal states are on the first level of hierarchy.

In my opinion these two elements of the Statecharts notation are not necessarily needed to specify post-WIMP interaction techniques, as their only purpose is the simplification of complex models. They do not necessarily add further semantics to the model. Appert and Beaudouin-Lafon also see their utility in practice very constrained, as they are "significantly more complicated and hard to learn than plain state machines" and therefore "user interface designers and developers have difficulties exploiting their power" [Appert and Beaudouin-Lafon, 2008].

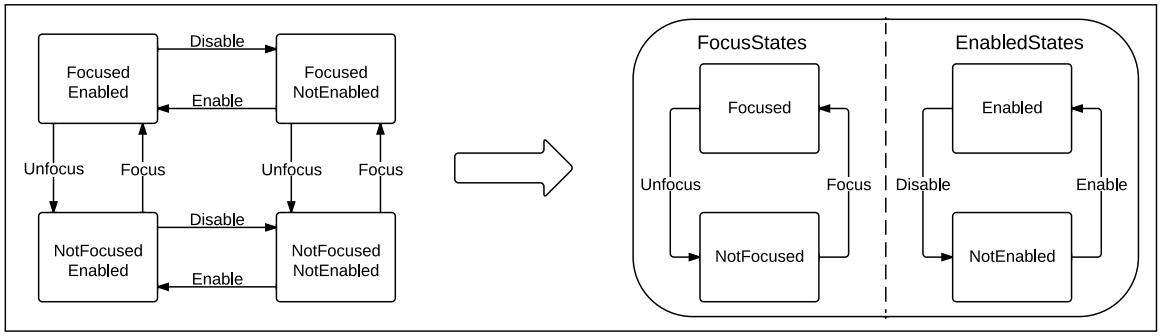

Figure 16: Orthogonality simplifies complex state machines by introducing concurrency

Broadcast communication The concept of broadcast communication is tightly connected to the concept of orthogonality. The idea is to emit events inside a transition action of one orthogonal state and use it as a trigger for a transition in another orthogonal state. Figure 17 shows an example of this functionality. Here, two orthogonal states are modeled which each contain two states. Each transition of the left state emits an event (E1, E2) which is in turn used as trigger for the transitions of the right state. The transitions of the right state therefore do not listen to external triggers, but to triggers emitted from within the state machine. The behavior of the right state thereby becomes tightly coupled to the behavior of the left state.

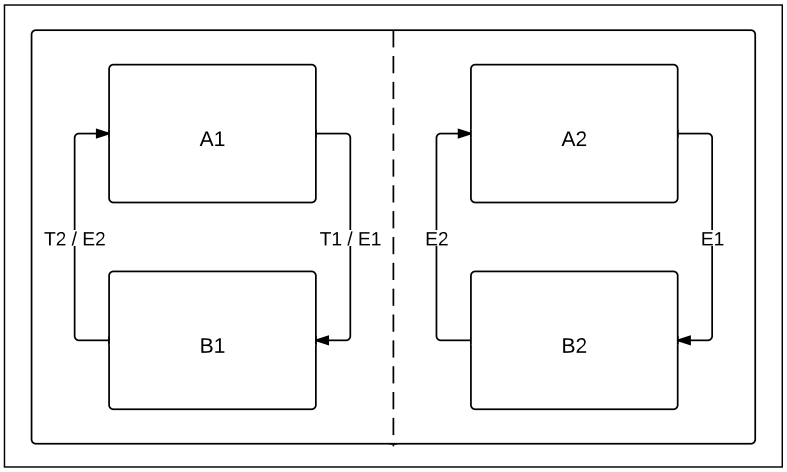

Figure 17: The broadcast mechanism of the Statecharts notation enables the coupling of orthogonal states

While orthogonal states per se can not be modeled with the Reactive State Machine framework, it is nonetheless possible to use this broadcast mechanism in combination with multiple state machine instances. This mechanism has for example been extensively used in the use case that is presented in section 9.3 on page 117.

4 Implementation

Researchers in the past were unanimous in stating that the implementation of finite-state machines to realize interactive behaviors was not supported by their then development tools and programming languages (e.g. [Parnas, 1969; Ackroyd, 1995; Dix, 2002; Appert and Beaudouin-Lafon, 2008]). Unfortunately these statements still hold true for most present-day user interface toolkits and programming languages. To overcome this lack of built-in support, researchers and developers have suggested different ways to realize a state machine implementation. In my seminar paper I classified these into two distinctive groups: low-level implementation techniques and third-party declarative state machine frameworks [Zöllner, 2011b]. It turns out, that declarative state machine framework are at a similar level of abstraction as the graphical model of the state machine (see figure 18). Thus, they are better suited to bridge the abstraction gap than low-level implementation techniques which typically work on the input event level. The reasons for this are discussed in the following.

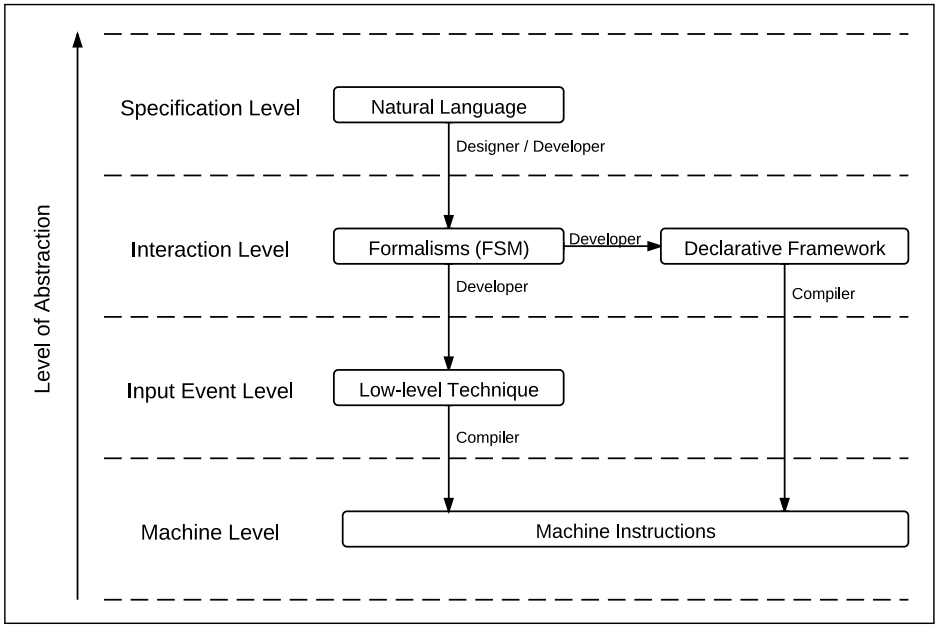

Figure 18: Declarative state machine frameworks are positioned on a higher level of abstraction than low-level implementation techniques

Low-level implementation techniques If a developer wants to create a state machine from scratch he has to establish all mechanisms by himself. Several patterns and best practices have been suggested from researchers and developers to aid in this process. They range from unstructured "twiggy tree" [Dix, 2002] approaches that employ procedural control flow mechanisms,

to maximally structured, object-oriented approaches that are backed by the State design pattern [Gamma et al., 1995, 338-348]. The most important advantage of these techniques is that a developer has the opportunity to deeply customize the behavior of the state machine and adapt it to his current needs. While this is certainly a favorable characteristic, there are also serious disadvantages that accompany these techniques: First, all elements and relations of the state machine have to be expressed with present-day programming language constructs. As they do not always represent the respective concept in a concise way, the resulting code tends to be bulky and complicated. Also, every technique requires the developer to follow a certain structure, be it procedural, table-based, object-oriented or some combination thereof. Yet, none of these structures faithfully resembles the real structure of a state machine. In fact, it is difficult to concisely map the structure of a two-dimensional diagram onto a linear textual representation. In the case of low-level implementation techniques additional complexity is created by the problem that not only the structure of a state machine has to be described in code, but also its dynamics or runtime behavior. Thereby structural and behavioral code constructs are mixed and the resulting implementation becomes very difficult to understand and dissect. The developer also has to ensure that all parts of a transition are always executed in the right order (as described in figure 8 on page 44). With many techniques this is a challenge, as the elements of the state machine are distributed over several classes and methods. Therefore errors in the execution flow of a transition are likely to occur and hard to detect. Consider for example the implementation of the state machine subset of figure 19 in listing 4 on the following page. The developer has to react to the trigger, employ decision logic constructs to *find* the correct transition and ensure that the order of the transition is correct. No special syntactical help is given by the programming language to achieve this, therefore the implementation looks complicated, although the represented concepts are simple.

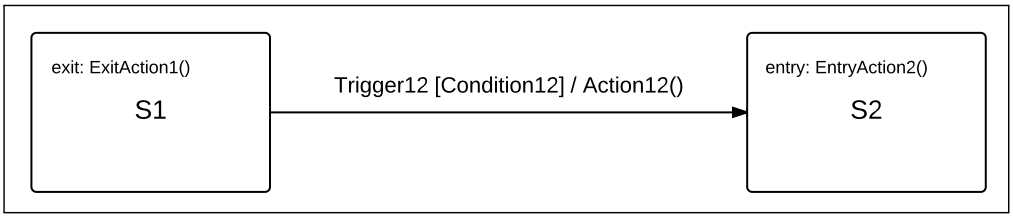

Figure 19: Example state machine subset

Declarative State Machine Frameworks The main characteristic of declarative state machine frameworks is that they abstract away the behavioral part of the state machine and offer the developer a declarative API to specify the structural part. This abstraction is made possible by the fact, that all state machines are conceptually identical. Regardless of how simple or complex a specific state machine is modeled, the basic operations are always the same and can therefore be hidden from the end-developer. If no behavioral code has to be written and the developer does not have to bother about any implementation details, the declarative part can be optimized greatly,

```
OnTrigger12()
{
    if(CurrentState == "S1")
        if(Condition12())
        {
            ExitAction1();
            Action12();
            CurrentState = "S2";
            EntryAction2();
        }
}
```

Listing 4: Structural and behavioral code gets mixed with low-level techniques

which eventually facilitates the problem of mapping a two-dimensional structure to a linear text representation. With the resulting interface, all elements and relations of the state machine can be expressed very concisely. The code that results in describing a state machine with such a framework is more comprehensible and maintainable and less error-prone than that of a self-built state machine. Consider for example the implementation of figure 19 on the previous page in listing 5. Here a hypothetical fluent API[12] is used to concisely describe all important elements and relations of the FSM. No programming language constructs other than method calls are used to describe the state machine, therefore the implementation looks very clean. As the method names directly represent the respective FSM concept, the implementation is also easily comprehensible for people who did not write the code.

```
var fsm = new StateMachine();
fsm.AddState("S1").AddExitAction(ExitAction1);
fsm.AddState("S2").AddEntryAction(EntryAction2);
fsm.AddTransition().From("S1").To("S2").On(Trigger12)
                   .Where(Condition12).Do(Action12);
```

Listing 5: Only structural code is needed with declarative state machine frameworks

Some frameworks even provide the possibility to use a special-purpose domain-specific language (DSL) to describe the state machine, eventually resulting in an even more concise description. These DSL descriptions are typically residing in a file external to the main application and are integrated at runtime. They can also be synchronized easily with a graphical diagram of the state machine, which renders them useful for model-based development scenarios.

In my seminar paper, I came to the conclusion that state machine frameworks are basically better suited for interaction designers than low-level implementation techniques. The reason for this is

[12] http://en.wikipedia.org/wiki/Fluent_interface

that interaction designers just want to straightforwardly realize the interactive behaviors they have modeled and do not want to worry about implementation details. Especially in the early phases of development where various different alternatives of an interactive behavior are tested, the use of a state machine framework pays off largely, as quick changes are easily possible and do not require great code refactoring. But also in the maintenance phase of an application, state machine frameworks are advantageous because their declarative code is easy to grasp even by developers who did not implement the behavior initially. I also stated in the seminar paper that a state machine framework based on an internal description is preferred, as interactive behaviors need access to internal (user interface) elements of the application, which is only difficult to achieve with external descriptions.

The problem with most current state machine frameworks however is that not all basic elements of state machines are supported sufficiently out-of-the box. Also, hardly any of the elements targeted at post-WIMP interaction design are supported at all. Therefore, designers currently still have to resort to low-level implementation techniques to realize their interactive behaviors. In my opinion, this is one of the main reasons why state machines "have often been used to describe user interaction [...] but rarely to actually directly program it" [Appert and Beaudouin-Lafon, 2008]. To underpin this observation, I compare a set of selected state machine frameworks in the following subsection and discuss their major shortcomings.

4.1 Comparison of State Machine Frameworks

There are many different state machine frameworks available for a variety of programming languages. A complete overview of those would probably go beyond the scope of this thesis. To enable a fair comparison, I chose to limit the selection to those frameworks that can be used inside state-of-the-art UI toolkits (Qt, Java Swing and WPF). Although only a few remain, they tend to be the ones containing the most features. The most advanced of these frameworks is the State Machine Framework that ships with the Qt library[13] (*Qt SMF*). Then, the *Swing States* framework for Java Swing follows [Appert and Beaudouin-Lafon, 2008]. Unfortunately, no dedicated state machine framework exists for WPF. Its built-in *Visual State Manager* can only be used to change the visual appearance of user interface elements and not to realize interactive behaviors. Yet, there are several other state machine frameworks available for the .NET runtime which can be alienated for the development of interactive behaviors in WPF applications. The most advanced of these is the *Stateless*[14] framework.

Table 3 on page 59 shows a feature comparison of these three selected state machine frameworks and the *Reactive State Machine (RSM)* framework which is presented in part V of this thesis. All

[13] http://doc.qt.nokia.com/latest/statemachine-api.html
[14] http://code.google.com/p/stateless/

state machine features that were presented in the previous section are considered in this table. As can be seen, no framework supports all available FSM features. *States* and their *entry* and *exit actions* are supported thoroughly by all frameworks. While *triggered transitions* are also supported by all frameworks, the fidelity of this support differs between the frameworks. This mainly has to do with restrictions in the way triggers have to be defined and how they are managed at runtime. In this particular aspect, all three frameworks have important limitations that restrain their expressivity. These limitations also affect the expressivity of *transition actions* and *guards*. In the following subsection, the issues that result thereof are discussed in greater detail. The definition of *timed transitions* is only supported by *Swing States* and the *RSM* and *automatic transitions* are only supported by the *RSM* as it is the only framework that supports the animation concept presented in the previous section. Although the *Qt SMF* and *Swing States* also offer support for animations, these are not backed by any concept (thus the downvote). This problem is addressed briefly below in subsection 4.1.2 on page 63. Furthermore, there is no built-in support for multiple input points in all current frameworks, except for the *RSM*. The *RSM* on the other hand offers only limited support for the concepts of the Statecharts notation. This part is dominated by the Qt SMF and *Swing States* which offer thorough support for all important Statecharts concepts.

4.1.1 Issues of Triggered Transitions

In post-WIMP systems, the triggers of triggered transitions are usually input events, stemming from some kind of input device. In general, an event in a user interface toolkit is made up of three different parts: (1) a name or type that represents the event, (2) the user interface element which is directly affected by the event, and (3) a set of metadata information associated with the event. The problem with many state machine frameworks is that they do not support all three parts of an input event. Many frameworks for example only use *flat* event tokens (strings or enumeration values) that stand for the name or type of the event. This requires the developer to feed these tokens into the state machine at runtime. To do so, he has to separately subscribe to the event, transform it into the representation that the state machine expects and pass it to the state machine. This not only requires a lot of setup code to be written to get the machine running, the code is also distributed over several event handler methods. The benefit of a compact FSM description is thereby rendered void. The Stateless framework, among others, employs this approach. Listing 6 on page 60 shows an implementation with Stateless of the example of figure 19 on page 55. While the actual state machine definition is very concise, the code to feed the state machine grows with every additional event trigger.

	Stateless	Qt SMF	Swing States	RSM
Platform / Language	.NET / C#	Qt / C++	Java Swing / Java	WPF / C#
States				
Entry / Exit Actions	++	++	++	++
Guards for Entry / Exit Actions	-	-	-	++
Transition Types				
Triggered Transitions	+	+	+	++
Timed Transitions	-	-	++	++
Automatic Transitions	-	-	-	++
Transition Properties				
Transition Actions	-	+	+	++
Guards	+	+	+	++
Post-WIMP Support				
Support for Animations	-	o	+	++
Support for Multiple Input Points	-	-	-	++
Statecharts Elements				
Hierarchy	++	++	++	-
Orthogonality		++	++	o[a]
Broadcast Communication		++	++	o[a]

[a] Only between separate state machines

Table 3: Comparison table of ready-to-use features available in selected FSM frameworks

```
/* FSM definition */
enum Triggers {Trigger12};
enum States {S1, S2};

var fsm = new StateMachine<States, Triggers>();

fsm.Configure(States.S1).OnExit(ExitAction1)
    .PermitIf(Triggers.Trigger12, States.S2, Condition12);

fsm.Configure(States.S2).OnEntry(EntryAction2);

/* Event handler */
OnTrigger12(Trigger12EventArgs e)
{
    fsm.Fire(Triggers.Trigger12);
}
```

Listing 6: Event tokens need to be feeded into the state machine of the Stateless framework

As can be seen from this example, the metadata information of the event (which is contained in the `Trigger12EventArgs` object) is not accessible by the condition and the transition action, as the event is flattened into a single token representation. To access metadata from within these methods, it needed to be stored beforehand in a separate data structure outside of the state machine. Listing 7 on the next page shows some example code of a drag operation that is modeled with the Stateless framework. As can be seen, the definition of the state machine does not suffice to capture the whole behavior. A lot of additional code has to be written to ensure that events are delivered to the state machine and that metadata information is accessible. In this example, the event subscription code has even been omitted for space reasons.

Both *Swing States* and the *Qt SMF* have a similar mechanism to get input passed into the state machine. Yet, they also offer out-of-the-box support for input events of classic input devices (mouse and keyboard). These do not have to be passed into the state machine but are registered automatically. However, when it comes to the events of non-standard input devices, they require the developer to create new classes for every custom event that is not supported out-of-the-box. As such non-standard input events are the norm rather than the exception in post-WIMP systems, a lot of additional code has to be written in these cases.

```
/* FSM definition */
enum State {NotDragging, Dragging};
enum Trigger {TouchDown, TouchMove, TouchUp};

var fsm = new StateMachine<State,Trigger>(State.NotDragging);

fsm.Configure(State.NotDragging).Permit(Trigger.TouchDown, State.Dragging);

fsm.Configure(State.Dragging).OnEntry(()=>MoveObject(_fingerPosition))
            .Permit(Trigger.TouchMove, State.Dragging)
            .Permit(Trigger.TouchUp, State.NotDragging);

/* temporal Storage of event metadata */
Point _fingerPosition;

/* Additional Code to drive the FSM */
OnTouchDown(object sender, TouchEventArgs e)
{
    fsm.Fire(Trigger.TouchDown);
}

OnTouchMove(object sender, TouchEventArgs e)
{
    _fingerPosition = e.GetPosition(null);
    fsm.Fire(Trigger.TouchDown);
}

OnTouchUp(object sender, TouchEventArgs e)
{
    fsm.Fire(Trigger.TouchDown);
}
```

Listing 7: Example of a drag operation modeled with Stateless

In *Swing States*, it is generally possible to get to the event metadata in the *guard* or *transition action*. Yet, the data is hidden in an object of the super-class and is not statically typed, which requires the developer to cast it to the correct type. Listing 8 on the following page shows exemplarily how event metadata can be accessed in a transition action of *Swing States*.

```
public State dragging = new State(){
    Transition t1 = new Move(){
        public void action(){
            /* metadata is hidden in triggeringEvent */
            MouseEvent evt = (MouseEvent)triggeringEvent;
            Point position = evt.getPoint();
        } }; };
```

Listing 8: Accessing event metadata in a transition of Swing States

In the *Qt SMF* the metadata of the event can also be acquired, yet it requires the developer to create a subclass of the transition class. The problem with the *Qt SMF* in general is that the developer has to create a subclass for every transition that contains a transition action, as C++ does not offer lambda functions or anonymous inner classes. The result thereof is that a lot of code is again distributed over several classes or files and references to user interface elements have to be passed to these distributed classes in order for the transition action to use them. Listing 9 shows exemplarily how metadata of a mouse event can be accessed in the *Qt SMF* by subclassing the `QMouseEventTransition` class.

```
class MySpecificTransition : QMouseEventTransitions
{
    virtual void onTransition(QEvent *e)
    {
        QMouseEvent *evt = static_cast<QMouseEvent*>(e);
        QPoint position = evt.globalPos();
    }
}
```

Listing 9: Accessing event metadata in a transition of Qt's SMF

In summary, the support for the integration of all event aspects into the state machine definition is mediocre in all three frameworks. While *Stateless* has no support at all, both *Swing States* and the *Qt SMF* have only basic support which requires the developer to write a lot of code and to use bulky language constructs to realize a rather simple problem.

4.1.2 Support for Animations

While *Stateless* offers no support for animations at all, *Swing States* has basic animation support and that of the *Qt SMF* is fairly advanced. The problem with the latter however is that their animations do not follow any conceptual considerations. Both frameworks are based on the Statecharts notation, which does not consider animations at all. Thus, their animation implementation is superimposed on the implementation of the Statecharts concepts. In *Swing States*, animations are separate entities that communicate with the state machine via their `Started` and `Stopped` events. It is however not communicated in which state the state machine resides during an animation and there is no information given how a running animation can be canceled. In the *Qt SMF*, animations can directly be associated with transitions. Here, the documentation[15] mentions a policy that is used to decide what is to happen if a state is exited before the animation has finished. This policy indicates that the animation is executed while the state machine is in the target state of the transition (which corresponds to case b) in figure 10 on page 46). In summary, animations and animated transitions are possible in both *Swing States* and the *Qt SMF*, but as they are not backed conceptually it is not explicitly defined how the behavior in certain situations will be.

4.1.3 Conclusions

The previous comparison not only revealed that current state machine frameworks are not feature-complete, but also that their API is often too bulky and complex. While the fluent API of *Stateless* is straightforward to use, not all features can be expressed with it, rendering it useless for post-WIMP interaction design. In contrast to this, *Swing States* and the *Qt SMF* offer more features, but often complex expressions have to be used and new classes have to be created to leverage these features. Especially with those two, often the impression is conveyed as if no declarative framework had been used, because the resulting code is distributed over several methods, classes and files, similar to a low-level implementation.

This lack of features and API usability leaves a gap for other frameworks. In part V of this thesis, I want to present such a framework, the Reactive State Machine (RSM). Although it is not entirely feature-complete (some elements of the Statecharts notation are missing), it provides a very concise fluent API with full support for input events, animations and multiple input points.

[15] http://developer.qt.nokia.com/doc/qt-4.8/statemachine-api.html#what-happens-if-a-state-is-exited-before-the-animation-has-finished

Part IV

Required Libraries

> If I have seen further it is by standing on ye sholders of Giants
>
> Isaac Newton

The Reactive State Machine (RSM) is built upon two third party libraries which enable support for all aspects of input events and animated transitions. As both features are key characteristics that tell the Reactive State Machine apart from other state machine frameworks, it is essential to understand the concepts and usage of these two libraries. Therefore, they are discussed thoroughly in the next two sections.

5 Reactive Extensions

Reactive Extensions for .NET (Rx) is a library that facilitates the orchestration of asynchronous operations in *reactive* applications. A reactive application is an application which has to deal with data that is produced at unexpected times. Building such applications in regular .NET is not supported comprehensively. As soon as several asynchronous computations, push-based messages or events need to be coordinated, the developer has to deal with ordering issues and the correct termination of each asynchronous operation, but also with failure cases or cancelation. Creating, understanding and maintaining such applications in regular .NET code is hard, as this code "doesn't follow normal control-flow" [Cloud Programmability Team, 2010]. With Rx, all kinds of reactive behaviors are unified into a consistent programming model and thereby made first-class citizens of the programming language. This effectively reduces the amount of composition and coordination code that needed to be written before and reduces or prevents the possibility of errors.

The Rx library consists of a set of .NET types that represent reactive behavior as asynchronous data streams and a set of operators that can be used to query, compose and coordinate these data streams. The library has been created by the Cloud Programmability Team at Microsoft[16]. It comes with a license that permits commercial use and can be downloaded for free[17].

5.1 The Unified Programming Model

Many aspects of our present-day computing are inherently asynchronous and reactive. Examples for reactive entities are push-based messages from a server in the cloud, the results of asynchronous operations running on the local machine or the events that an input device produces. Even though these examples stem from entirely different domains, they are conceptually very similar, as all require the developer to establish a dedicated entity that is notified asynchronously when new data is available. Unfortunately, this conceptual similarity is not yet reflected in present-day programming languages. In C# for instance, to schedule asynchronous work there is either the Asynchronous Programming Model (APM) which uses a `BeginXXX/EndXXX` method pair, the Event-based Asynchronous Pattern (EAP) which uses regular .NET events to notify completion, asynchronous methods that use delegates for the same purpose or the new `Task<T> API` which employs continuation methods. In the near future, there will even be native language support in C# for asynchronous tasks with two additional keywords (`async/await`). Input events on the other hand are implemented using regular .NET events or WPF Routed Events which use the well-known publish/subscribe pattern. All of these techniques more or less represent the same aspect, albeit for different purposes and with different strategies. As completion, fault, and the arrival

[16] http://msdn.microsoft.com/en-us/data/gg577609
[17] http://msdn.microsoft.com/en-us/data/gg577610

of new data is handled entirely differently in all techniques, the composition and coordination of several asynchronous tasks and events is very difficult. There is actually no common denominator that captures the essence of reactive behavior in .NET. This is the gap that Rx fills.

Rx thereby does not replace the existing sources of reactive behavior, instead it wraps and unifies them by providing a pair of interfaces (`IObservable<T>, IObserver<T>`) which are a generalization of the operations that constitute reactive behavior. These interfaces make it possible to specify generic operators that allow the coordination and composition of reactive entities. They also provide an object-oriented approach to reactive computing and render reactive entities first-class citizens of the language. First-class citizenship is generally attributed to those elements of a programming language that have "the fewest restrictions" [Abelson et al., 1996, 1.3.4]. This means that all elements with first-class status can be named by variables, passed as arguments to procedures, returned as the results of procedures, included in data structures and constructed at runtime [Abelson et al., 1996, 1.3.4]. In C#, first-class status is attributed to many programming constructs, such as primitive data types (int, double, ...), compound data types (objects, structs), and methods. The classic asynchronous programming constructs and event mechanisms however do not receive this status. As types that implement the two interfaces of Rx are regular .NET objects, reactive entities that are represented by these interfaces are eventually rendered first-class citizens of the C# language.

5.1.1 Observable Collections

The main idea of Rx is to treat every reactive entity as a collection of data that can be observed. A mouse for example can be seen as a hidden data source of points. While it usually communicates location changes by raising `MouseMove` events, with Rx, these location changes are represented by a collection that *pushes* them to potential observers. These collections are therefore called *observable collections* or simply *observables*. In comparison to a pull-based collection, an observable collection usually does not store data values. Instead they are pushed towards the observers, as soon as they are generated.

The advantage of treating reactive entities as collections is that almost all operators that apply to regular pull-based collections, equally apply to those push-based or observable collections. This especially includes the popular LINQ (Language INtegrated Query) technology that is used throughout the .NET framework to perform tasks such as filtering, element selection and aggregation. Some of these operators are presented below in subsection 5.2 on page 71, others can be found in Appendix A.2 on page 155

The reason why LINQ can also be applied to observable collections is rooted in a deeper mathematical relationship between regular pull-based collections and these push-based observable collections. The interfaces of observable collections can actually be derived by dualizing the interfaces

of regular pull-based collections, which makes those two collection types mathematical duals. For the interested reader this dualization process is explained in greater detail in Appendix A.1 on page 150.

Observable collections and their observers are based on the `IObservable<T>/IObserver<T>` pair of interfaces which is shown in listing 10. Every type that wants to be an observable collection has to implement the `IObservable<T>` interface and every type that wants to be an observer has to implement the `IObserver<T>` interface. These interfaces thereby are actually a realization of the popular Observer pattern known from object-oriented software engineering [Gamma et al., 1995, 326–337].

```
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<T>
{
    void OnNext(T next);
    void OnCompleted();
    void OnError(Exception e);
}
```

Listing 10: Definitions of `IObservable<T>` and `IObserver<T>`

5.1.2 Lifetime Phases

In the lifetime of an observable collection, the four different phases *Declaration, Subscription, Publication* and *Disposal of Subscription* can be differentiated.

Declaration At first, an observable collection has to be declared and created. This can be done by using any of the creation operators (see subsection 5.2.2 on page 73) which create primitive observable collections, or by wrapping existing .NET events or asynchronous methods into an observable collection using the conversion operators (see subsection 5.2.3 on page 74). Another option is to extend an existing observable collection using filters or other composition operators (see subsection 5.2.4 on page 74 and the following ones).

Subscription Next, potential observers can subscribe themselves to the observable collection by calling the `Subscribe()` method. An observable thereby behaves like a multicast delegate, as every value is pushed to every observer that has subscribed.

Publication Immediately after declaration an observable collection is ready to publish values. The actual publication of values is however only performed if at least one observer has subscribed itself to the observable. Observers are notified of new values by means of the `OnNext(T next)` method which is called by the observable, passing the new value as parameter. The observable may also signal completion by calling the `OnError()` or `OnCompleted()` methods. The protocol of observables in the publication phase is `OnNext*[OnError|OnCompleted]?`. This means that an arbitrary number of `OnNext()` calls is optionally followed by either an `OnCompleted()` or an `OnError()` call. Whenever an error occurs, the observable automatically shuts down and does not produce any additional values. The same is true when the official end of the observable is signaled via the `OnCompleted()` call. Note, that the observable can be infinite, which means that neither an `OnError()` nor an `OnCompleted()` call will occur.

Disposal of Subscription The last phase is the disposal phase, which is used to unsubscribe an observer from the observable collection. To enable disposing, the `Subscribe()` method returns an object of type `IDisposable`. The `Dispose()` method of this object can be used to dispose of, or unsubscribe the observer which was previously passed in as parameter to the `Subscribe()` method. Using the `IDisposable` type for this purpose has several advantages over other unsubscription mechanisms that are used throughout the .NET framework. Regular .NET events for example are subscribed and unsubscribed using the `+=/-=` syntax which resembles the `AddListener/RemoveListener` or `Subscribe/Unsubscribe` pattern of other languages and toolkits. In these patterns, nothing (`void`) is returned when subscribing to the source. Thus, to release the relationship between a publisher and a specific subscriber, a reference to the subscriber must be given in the unsubscribe method. If this approach had been used in Rx, the use of anonymous functions or lambda expressions as observers would have been complicated, because the observer would have to be stored in a field to reference it for unsubscription. A further disadvantage of this approach is that composition is complicated greatly. An observable that is composed of several other observables, automatically creates subscriptions to those. When the outer observable is disposed, it has to be assured that the subscriptions that the outer observable has established are also disposed. Thus, each part of a composition has to know how to undo or reset the subscriptions it made. This is not easily possible with a pair of `Subscribe()/Unsubscribe()` methods, as it is not always possible to get a reference to the observer. The advantage of the `IDisposable` approach in this case is, that it creates a closure over the observable and the observer, leaving back only a single object that is capable of releasing the relationship of those. This object can easily be stored in a data structure, where it can be accessed by other entities when needed.

5.1.3 Composing and Coordinating Observables

One of the most daunting tasks for a developer of a reactive application is the orchestration of multiple asynchronous operations and events. This can be attributed to the lack of composition and coordination support in all default programming models. Generally, being reactive in a compositional way is not easy: One reacts to one thing and then it is over. It is hard to maintain states between subsequent reactions. It is even harder to maintain states in cases of cancellation or abnormal termination. As observable collections unify the propagation of new data and the completion and abnormal termination into a single interface, the composition and coordination of multiple observables is greatly facilitated.

Composition in Rx means that there is a function (a combinator) which takes things and produces new things. To enable composition each operator has to act as both observer and observable. Thereby, it can observe one or multiple source observables that are passed in as parameter and return the combination thereof as another observable. As soon as one operator breaks out of the observable world by using a return type other than `IObservable<T>`, composition is no longer possible. Thus, almost all Rx operators again return an observable collection and can therefore be chained together. The observable that is returned not necessarily has to be of the same type as the observable(s) that were passed into the combinator. It is also possible that the combinator changes the type of the values that it receives. A `Select()` operator, for example, can take data values of some type `T` and project them onto data values of some other type `U`. Its signature therefore looks as follows:

```
IObservable<U> Select<T,U>(this IObservable<T> source, Func<T,U> selector)
```

Coordination addresses situations where multiple observables have to be combined in some way. Thereby the ordering, timing and concurrency of these observables has to be considered as well as the continuation actions after completion or abnormal termination. Examples for simple coordination operators are `TakeUntil()` or `SkipUntil()`, which take or skip values from the first observable until the other begins to produce values. Others, like `Merge()` or `Zip()` combine the values of two or more observables and return the combination as another observable.

Concurrency in Rx is controlled by means of the `IScheduler` interface. Classes that implement this interface are called schedulers. Each operator gets passed an instance of such a scheduler and the actual work of the operator is then delegated to this scheduler. Each scheduler thereby represents a special execution context inside or outside the system. Examples of such schedulers are the `ThreadPoolScheduler` which delegates work to a new `Thread` of the `ThreadPool` or the `CurrentThreadScheduler` which delegates work to the current `Thread`. With this mechanism it is possible to distribute work to several entities of the system, or even beyond system boundaries. Thus, multiple cores of a processor or even the computing power of the cloud can be exploited

easily in a reactive application. Since the results of each operator are again returned as observable collections, they can be composed easily with one another inside the main application. To eventually get the results back into the UI of the application, one of the special UI schedulers, such as the `DispatcherScheduler`, can be employed.

5.1.4 Visualizing Observables

To help communicating the characteristics of a specific observable or combination of observables, a special type of diagram, the *marble diagram*, can be employed. Figure 20 shows such a marble diagram for a simple observable. This observable produces integer values in a range from one to six which are filtered by a `Where()` filter that lets through only even values. The horizontal lines in this diagram mark the time going from left to right. Every operator has its own horizontal line. The bottom line usually represents the result values that are produced from the whole observable. The blue marbles represent the `OnNext()` calls or values that flow through the observable and its operators. Values that are passed from one operator to the next are connected by a dashed arrow. The black vertical bar at the right represents an `OnCompleted()` call which marks the end of the observable. Exceptions are represented by a cross, which can be seen in figure 21 on the next page. There, a composite observable is visualized which concatenates an integer range from one to four with an exception.

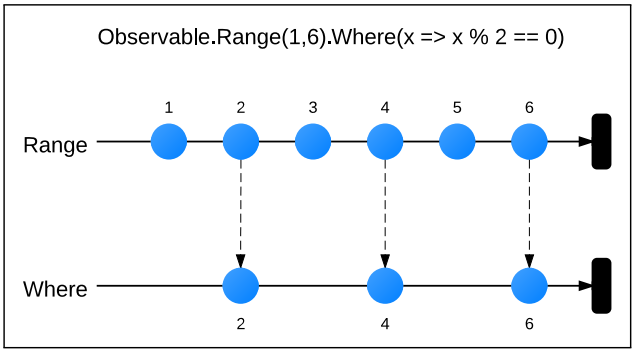

Figure 20: Marble diagrams are used to visualize observables

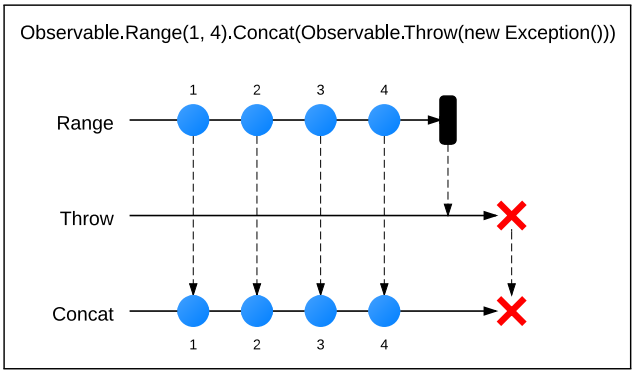

Figure 21: Marble diagram with an exception

5.2 Usage

The core interfaces of Rx (`IObservable<T>/IObserver<T>`) already ship with the current version of the .NET framework (4.0) in its Base Class Library. The full power of Rx however lies in its extension methods or operators that are located in separate assemblies. These extension methods mainly operate on the `IObservable<T>` interface. In the main Rx assembly, more than 100 distinct extension methods can be found with more than 400 parameter overloads. Additional extension methods can be created by any developer without further ado. One such contributor project is Rxx[18] which provides several additional extension methods.

Because of their large amount, not all extension methods can be addressed here. Thus, in the following only those extension methods are introduced that are used later for the definition of state machine triggers. For the interested reader I assembled a set of additional extension methods in Appendix A.2 on page 155. Many of these extension methods are also explained in greater detail in the official documentation[19], the Rx wiki[20] or on videos at Channel 9[21]

5.2.1 Subscribing to Observables

Before the actual operators are introduced, I shortly present how to subscribe to an observable collection. The default `Subscribe()` method expects an instance of `IObserver<T>` to be passed as parameter. Thus, the `OnNext()`, `OnError()` and `OnCompleted()` methods of the interface have to be implemented by the class that implements the interface. As it is not convenient to create a new type for every subscription that is made, Rx ships with a set of extension methods,

[18] http://rxx.codeplex.com
[19] http://msdn.microsoft.com/en-us/library/hh242985(VS.103).aspx
[20] http://rxwiki.wikidot.com/
[21] http://channel9.msdn.com/Tags/rx

that accept method delegates for the three methods of the observer. Inside this extension method an anonymous observer is automatically created which makes use of the method delegates that were provided. With these extension methods it is not necessary to provide delegates for all three methods, if a developer just needs to react to one of them. The delegates can either be a reference to a concrete method or an anonymous delegate or lambda expression, which also allows closuring over the current context. Listing 11 shows the difference between implementing an observer from scratch (1a), using one of the extension methods with a lambda expression (1b) and using the extension method with all three lambda expressions (1c). Obviously, the extension method approach is far more compact and concise than completely implementing an observer. Thus, the creators of Rx suggest to always use the extension methods and never implement the `IObserver<T>` interface by oneself.

```
//(1a) providing a full-fledged observer
public class ConsoleObserver : IObserver<int>
{
    public void OnNext(int next)
    {
        Console.WriteLine(next);
    }

    public void OnError(Exception e) { //not needed }
    public void OnCompleted() { //not needed }
}

someObservable.Subscribe(new ConsoleObserver());

//(1b) providing a lambda expression for OnNext
someObservable.Subscribe(next => Console.WriteLine(next));

//(1c) providing a lambda expression for OnNext, OnError and OnCompleted
someObservable.Subscribe( next => Console.WriteLine(next),
    error => Console.WriteLine(error.Message),
    () => Console.WriteLine("Observable completed"));
```

Listing 11: The `Subscribe()` method either takes a full-fledged observer or delegates for the three methods

5.2.2 Creating Observables

To create simple observable collections, basic operators are provided by Rx. The primitive creation operators `Never()`, `Empty()`, `Throw()` and `Return()` have only limited use in real-world scenarios. They are only needed for algebraic reasons. Figure 22 shows these operators as marble diagrams, together with the `Range()` and `Repeat()` operators which are slightly more useful. `Range()` produces a range of integers and `Repeat()` repeats a given value or sequence n-times or infinitely. Other simple creation operators are based on time. The `Timer()` operator produces exactly one value at a given time, or after a given timespan has passed, whereas the `Interval()` operator continuously produces values after a given timespan has passed (see figure 23).

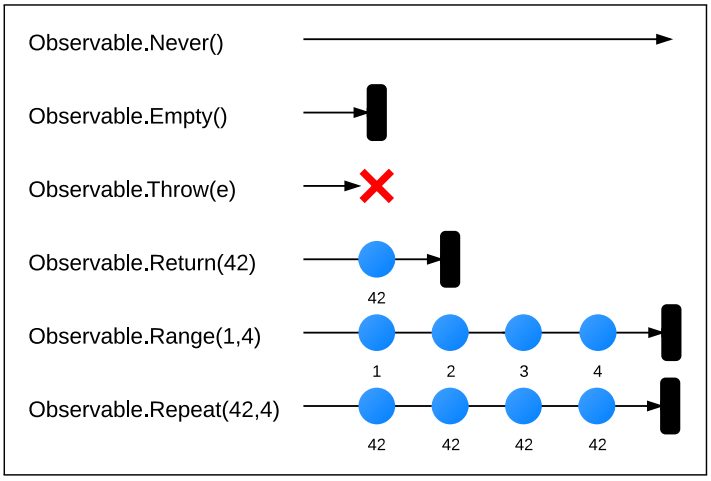

Figure 22: Primitive creation operators

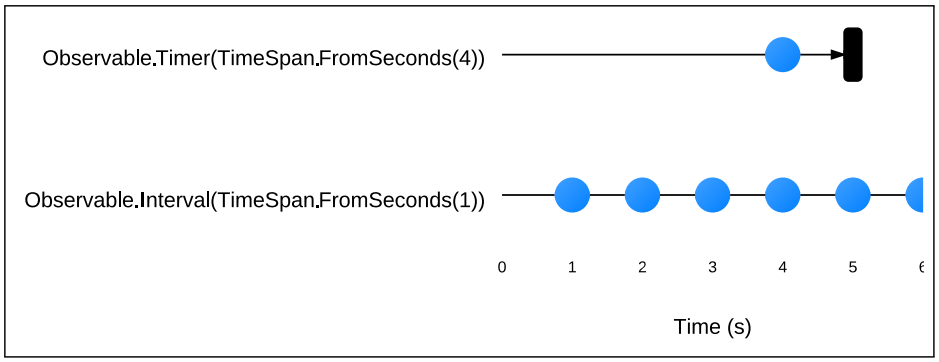

Figure 23: Creating time-based observables with `Timer()` and `Interval()`

5.2.3 Conversion Operators

Significantly more important than the operators which create observable collections out of thin air, are the operators that create observable collections out of existing reactive entities, such as regular .NET events or the asynchronous programming model. The operators thereby create wrappers for these reactive entities and provide their functionality as an observable collection. Since these operators enable developers to wrap input events into observable collections, they are very important for the Reactive State Machine, which requires triggers to be observable collections. Listing 12 shows exemplarily how a `TouchMove` event can be wrapped into an observable collection. Additional conversion operators can be found in Appendix A.2 on page 155.

```
IObservable<EventPattern<TouchEventArgs>> touchMoves;

touchMoves = Observable.FromEventPattern(window, "TouchMove");
```

Listing 12: Wrapping a `TouchEvent` into an observable

5.2.4 Filter Operators

Once an observable collection has been created, several filter methods can be used to extract only those data values that are needed. The most used extension method for filtering is the `Where()` operator, which evaluates a condition and only lets through values that meet this condition. With such an operator it is for example possible to create filtered events, such as in figure 24. This observable filters out all `TouchMove` events that are not stemming from a finger.

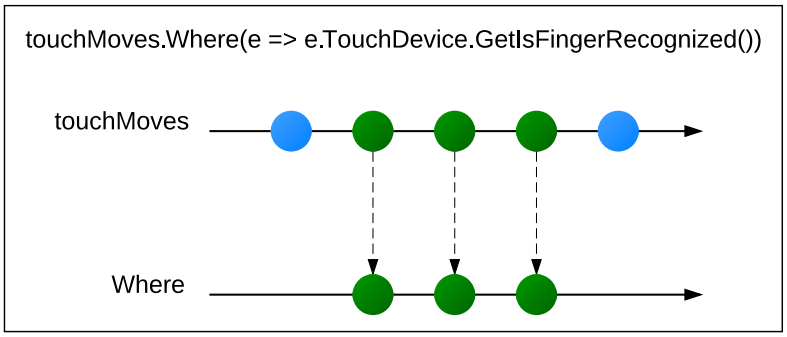

Figure 24: The `Where()` operator filters out values that do not satisfy a given condition

5.2.5 Projection Operators

The projection operator `Select()` takes each value from the observable and projects it onto a new value. Thereby it may change the type of the value. The projection is performed with a function that receives the value and returns another value. Figure 25 shows a marble diagram where the `Select()` operator is used to extract the position of the `TouchMove` event.

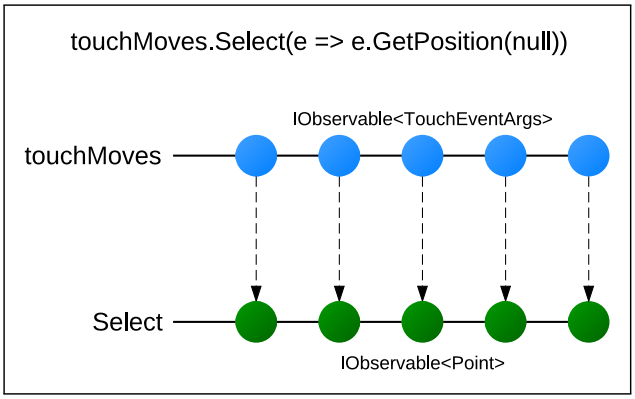

Figure 25: The `Select()` operator projects every value to a new one

5.2.6 Time-based Operators

As observables can produce values at unexpected times, it is important to have operators that introduce the notion of time. An operator that is used in the examples below, is the `Delay()` operator. It simply delays the propagation of every value for a given timespan as shown in figure 26.

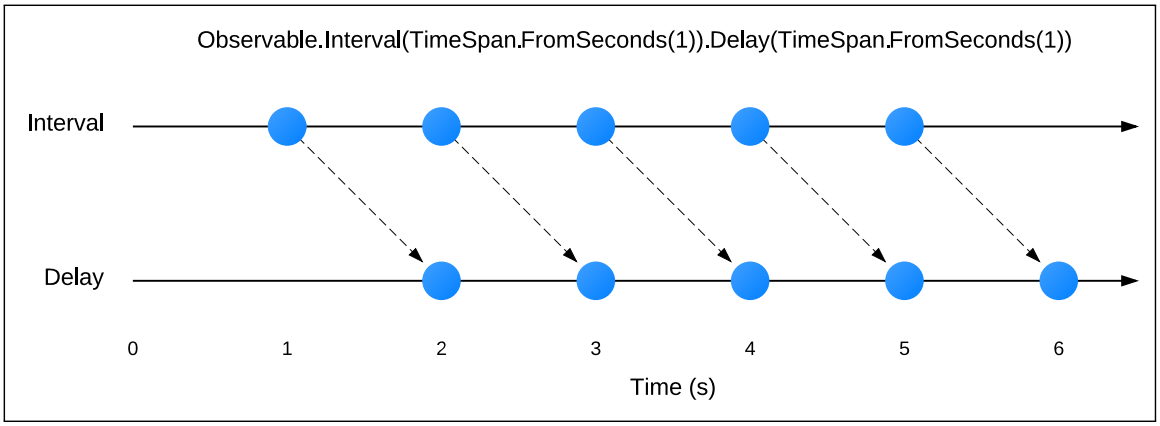

Figure 26: The `Delay()` operators delays each `OnNext()` call for a given timespan

5.3 Examples

To show how observable collections can contribute to the challenges that developers of interactive behaviors are facing, I present two examples in the following. The first deals with the selection of input events, which is not appropriately supported in present-day user interface toolkits. The second example then shows how a more complex interaction technique, the `ReleaseLink` behavior of Facet-Streams, can be realized with several Rx operators.

5.3.1 Selecting Input Events

The subscription to an input event usually implies a two-step process: First, the event has to be selected and then the selected event can be bound to an event handler method [Accot et al., 1996]. To a developer of a post-WIMP system it is very important that he can specify precisely what specific event he is interested in. He may for example be interested in all `TouchDown` events that occur on a specific interface element and where the orientation of the finger is inbetween a given angle. While nowadays every user interface toolkit allows the selection of events based on the interface element on which they occur, there is no general means of specifying criteria that an event has to meet. If a developer wants to specify such criteria he has to do so in the event handler method (see listing 13). This typically involves some sort of decision logic control structures which tend to bloat the event handler method and, when mixed with the reaction to the event, lead to confusing spaghetti code [Myers, 1991].

```
//(1) Definition of the event in the UIElement class
public event EventHandler<TouchDownEventArgs> TouchDown;

uiElement.TouchDown += OnTouchDown;    //(2) Subscription

//(3) Event Handler
private void TouchDown(object sender, TouchEventArgs e)
{
    var orientation = e.EventArgs.Device.GetOrientation(null);
    if(orientation >= 90 && orientation <= 270)
        Reaction();
}

uiElement.TouchDown -= OnTouchDown    //(4) Disposal of Subscription
```

Listing 13: With regular events, filters must be specified in the event handler method

According to Accot et al. it should be possible to use all properties of an event to specify in detail which event to select. They suggested a method to select events that allows designers to "explicitly express constraints, instead of implementing them by hand in the callbacks" [Accot et al., 1996]. I am not aware of any modern user interface toolkit that has adopted this approach and has built-in support for the fine-grained selection of input events. Fortunately the Rx framework provides operators to wrap events inside observable collections and specify conditions to filter out those events that do not meet the respective criteria. Listing 14 shows how such a filtered event can be specified with Rx. The advantage of this approach is that the filtered event can be reused for several purposes and that the resulting event handler method is very concise as it only contains the reaction to the event.

```
//(1) Wrapping the regular event inside an observable collection
var touchDownEvent = Observable.FromEventPattern(uiElement, "TouchDown");

//(2) Creation of a filtered event
var filteredEvent = touchDownEvent.Where(e =>
{
    var orientation = e.EventArgs.Device.GetOrientation(null);
    return orientation >= 90 && orientation <= 270
});

//(3) Subscription
IDisposable subscription;
subscription = filteredEvent.Subscribe(OnFilteredTouchDown);

//(4) Event Handler
private void OnFilteredTouchDown(TouchEventArgs e)
{
    Reaction();
}

//(5) Disposal of Subscription
subscription.Dispose();
```

Listing 14: With Rx operators, a filtered event can be specified straightforwardly

5.3.2 The `ReleaseLink` Behavior

As has been shown previously in subsection 2.2.3 on page 31, a special behavior was created in the Facet-Streams system to detach a visual link from a node. The naive implementation of this behavior required a good deal of timer management code and was distributed over several methods. With the help of Rx and its various operators, the definition of the behavior shrinks down to a few lines of compact and concise code (as shown in listing 15). In this implementation, first the two events `TouchDown` and `TouchUp` are wrapped inside observable collections. Thereby a filter is applied which filters out all events that do not stem from fingers (but from objects or tags) (1). In the next line, the trigger is defined (2). For the trigger, the `TouchDown` event is delayed for the given timespan (i.e. one second). The `TakeUntil()` operator then ensures that the trigger only fires if the `TouchUp` event did not occur before the timespan elapsed. The problem with the `TakeUntil()` operator however is, that it automatically finishes the observable collection when it is called. When this happens, the behavior can only be executed once. To ensure that the behavior can be executed over and over again, the trigger is made repeatable by means of the `Defer()` and `Repeat()` operators (4). In the last line, the execution context is switched to the `Dispatcher` by means of the `ObserveOnDispatcher()` method and the subscription to the behavior is established (5). Switching the execution context is necessary, because the `Delay()` operator opened a new execution context in another thread. As the final event handler needs to access UI elements, it has to be executed in the `Dispatcher`.

```
//(1) wrap the touchDown and touchUp events inside observable collections
touchDown = Observable.FromEventPattern(AssociatedObject, "TouchDown")
                      .Where(e => e.TouchDevice.GetIsFingerRecognized());

touchUp = Observable.FromEventPattern(AssociatedObject, "TouchUp")
                    .Where(e => e.TouchDevice.GetIsFingerRecognized());

//(2) define the trigger
trigger = touchDown.Delay(TimeSpan.FromSeconds(Delay)).TakeUntil(touchUp);

//(3) allow the repeated execution of the behavior
repeatableTrigger = Observable.Defer(() => trigger).Repeat()

//(4) subscribe to it on the Dispatcher
repeatableTrigger.ObserveOnDispatcher().Subscribe(ReleaseLink);
```

Listing 15: With Rx operators, the `ReleaseLink` behavior of Facet-Streams is specified concisely

6 The Visual State Manager

The Visual State Manager (VSM) is a set of classes[22] in WPF that can be used to realize state-based adaption of visual components. As the VSM can only change the appearance of visual components and not their behavior, it does not represent a full-fledged finite-state machine toolkit such as Swing States for Java Swing[23] or the State Machine Framework of the QT library[24]. The uniqueness and strength of the VSM is its strict usage of WPF's powerful animation system for all actions of the state machine. In spite of configuring the VSM and its animations directly in XAML code, the user can use Microsoft's Expression Blend[25] tool, which offers an easy-to-use graphical editor. By replacing the default VSM with a custom implementation, the Reactive State Machine is able to use the powerful animation support of the VSM and developers can leverage Expression Blend to define the visual appearance of their components in certain states or during transitions. In the following, the individual components of the Visual State Manager are explained in greater detail, along with examples that show how to implement them.

6.1 The `VisualStateGroup` Class

In the VSM, each state machine is of type `VisualStateGroup`. It can be attached to every visual component, such as a `Window`, a `Grid` or a `Button`. A `VisualStateGroup` can contain any number of states (of type `VisualState`) and transitions (of type `VisualTransition`). The VSM supports the concept of *orthogonality* by allowing more than one `VisualStateGroup` per visual component. This is useful for components which need two mutually exclusive state machines such as a `Button` which has *press*-states (not pressed, pressed) and *hover*-states (mouse over, mouse not over). Every `VisualStateGroup` is identified by a name which must be unique among all other groups of this particular visual component. The `VisualStateGroup` is usually attached to a visual component in the component's declarative XAML part and not in the code-behind or C# part. The reason for this are the animations, which are very complicated to create in C# code. Listing 16 on the next page shows the definition of two empty orthogonal groups, which are attached to a `Button`.

[22] http://msdn.microsoft.com/en-us/library/system.windows.visualstatemanager.aspx
[23] http://swingstates.sourceforge.net/
[24] http://doc.qt.nokia.com/latest/statemachine-api.html
[25] http://www.microsoft.com/expression/products/Blend_Overview.aspx

```
<Button>
  <VisualStateManager.VisualStateGroups>

    <VisualStateGroup name="pressStates"/>

    <VisualStateGroup name="hoverStates"/>

  </VisualStateManager.VisualStateGroups>
</Button>
```

Listing 16: Defining two orthogonal VisualStateGroups in XAML

6.2 The `VisualState` Class

Every state of a `VisualStateGroup` is represented by an instance of the `VisualState` class. It is identified by a name which must be unique among all other states of the same group. For every state the developer can specify an *entry action* using WPF animations. This *entry action* is started whenever the state is entered. If it is still running when the state is exited, it will be stopped. Listing 17 shows the definition of the *press*-states of the `Button`. The animations make sure that the background of the `Button` is *DarkGray* when it is pressed, and *LightGray* when it is not pressed.

```
<VisualStateGroup name="pressStates">

  <VisualState name="Pressed">
    <Storyboard>
      <ColorAnimation Storyboard.TargetProperty="Background.Color"
                      To="DarkGray"/>
    </Storyboard>
  </VisualState>

  <VisualState name="NotPressed">
    <Storyboard>
      <ColorAnimation Storyboard.TargetProperty="Background.Color"
                      To="LightGray"/>
    </Storyboard>
  </VisualState>

</VisualStateGroup>
```

Listing 17: Defining states in XAML

6.3 The `VisualTransition` Class

The enter action of the `VisualState` ensures that certain visual parameters are always set when the state is entered. With the `VisualTransition` class it is possible to specify what is supposed to happen when the state machine transitions from one state to the other. This concept is basically the same as the concept of transition actions of state-transition diagrams, with the exception that the VSM can only start WPF animations and not call arbitrary actions. Listing 18 shows how transitions can be specified in XAML. In this example the color animation is the same as the color animation of the previous example, but with an additional `Duration` parameter. This parameter ensures that the color changes smoothly over the duration of one second. The `From` and `To` parameters refer to the names of the corresponding states. It is also possible to underspecify transitions, by omitting the `To` or the `From` parameter. In this case the animation is applied to all transitions that start in a specific state or end in a specific state.

```
<VisualStateGroup.Transitions>
  <VisualTransition From="Pressed" To="NotPressed">
    <Storyboard>
      <ColorAnimation Storyboard.TargetProperty="Background.Color"
                      To="LightGray" Duration="00:00:01"/>
    </Storyboard>
  </VisualTransition>
  <VisualTransition From="NotPressed" To="Pressed">
    <Storyboard>
      <ColorAnimation Storyboard.TargetProperty="Background.Color"
                      To="DarkGray" Duration="00:00:01"/>
    </Storyboard>
  </VisualTransition>
</VisualStateGroup.Transitions>
```

Listing 18: Defining transitions in XAML

6.4 Controlling the Visual State Manager

While the definition of states and transitions is usually done in declarative XAML code, it is common to control the state machine from the code-behind part of the visual component. To trigger a transition from the current state to another, the `VisualStateManager` class provides the static method `GoToState()` which takes as parameter the visual component, the name of the target state and a flag indicating if `VisualTransitions` are to be used. Listing 19 on the following page shows how this method is used in the event handler method of the `TouchDown` and `TouchUp` event to transition between the *press*-states.

```
private void OnTouchDown(obect sender, TouchEventArgs e)
{
    VisualStateManager.GoToState(this, "Pressed", true);
}

private void OnTouchUp(obect sender, TouchEventArgs e)
{
    VisualStateManager.GoToState(this, "NotPressed", true);
}
```

Listing 19: Triggering a transition from code-behind

6.5 Tool Support

The VSM is supported by Microsoft's Expression Blend tool which is a WYSIWYG[26] editor that creates clean XAML code. With this editor it is possible to configure states and transitions without writing any line of code. When a state or transition is selected in the left panel, Expression Blend automatically switches into recording mode. In this mode, all changes to properties that can be made in the center and right panel are recorded into an animation. The animation can then be edited at the bottom in a timeline editor which is similar to editing an animation in Adobe Flash.

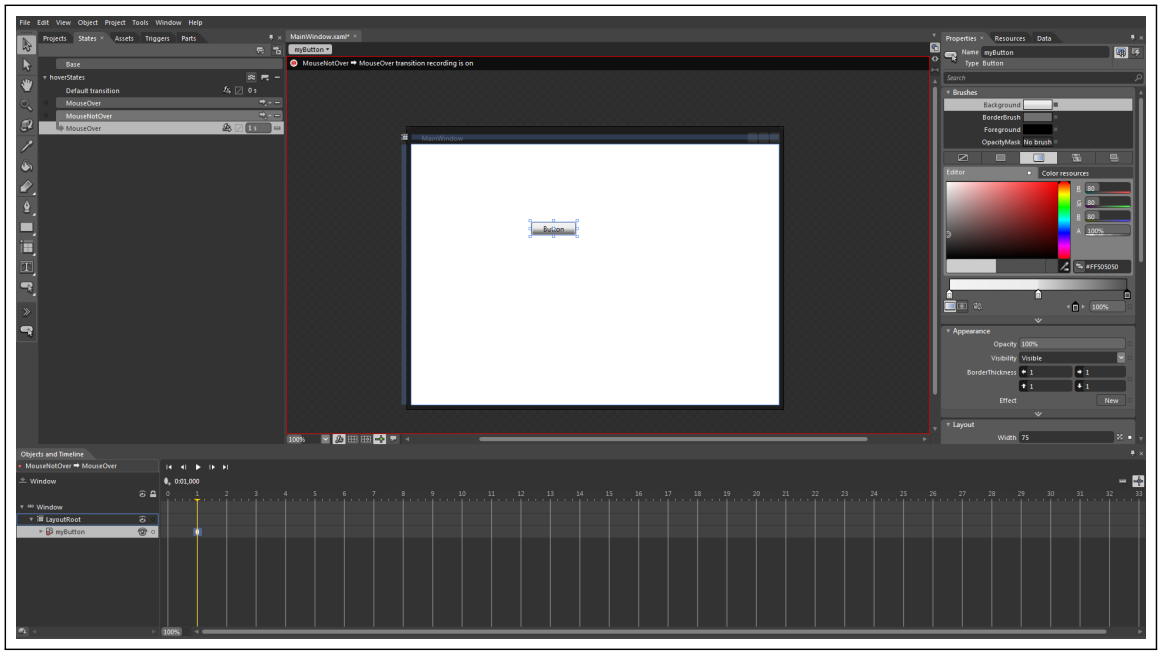

Figure 27: Configuration of states, transitions and animations in Expression Blend

[26] http://en.wikipedia.org/wiki/WYSIWYG

Part V

Reactive State Machine

> Never underestimate the power of a simple tool
>
> Craig Bruce

In this part of the thesis, the Reactive State Machine (RSM) is introduced. The RSM is a set of classes that enables the declarative definition of finite-state machines for post-WIMP interaction design. Special features are the support for all aspects of an event via Rx and the support for animated transitions via the Visual State Manager (VSM). In the next section, an overview over the features and public API of the RSM is given. After that, its architecture and implementation decisions are presented. The final section then presents assorted use cases that show how the RSM is employed in state-of-the-art post-WIMP user interfaces.

7 Overview

The Reactive State Machine framework (RSM) is a declarative state machine framework targeted at WPF and the .NET runtime to facilitate post-WIMP interaction design. Its development was motivated by the fact that low-level implementation techniques of state machines often create complicated and verbose code and that current state machine frameworks do not support all features that are needed for post-WIMP interaction design. The framework has been released as an open-source project under the BSD license and can be downloaded for free[27].

With the Reactive State Machine framework, internal descriptions of state machines can be created. Thereby, the state machines are directly integrated into the application and configured in plain C# code. As a consequence, they have direct access to all elements of the application, such as user interface elements and input events. The library consists of approximately 800 lines of code, which is rather small, compared to the 10.000 lines of Java code of Swing States [Appert and Beaudouin-Lafon, 2008], for example. Yet, in the Swing States library, many functionalities, such as the animation support, had to be created by the developers from scratch. As the RSM is based on the Rx library and the Visual State Manager, a lot of functionality can be used from these, which keeps the actual implementation of the framework comparatively small.

In the following, I want to present all features of the Reactive State Machine framework which can be configured via its public API.

7.1 State Machine Management

Each state machine in the RSM is an instance of the class `ReactiveStateMachine<T>`. The generic type parameter `T` denotes the type of the states that the machine accepts (see next subsection). After the instantiation of a state machine, it has to be configured first. Configuration calls can only be made when the state machine is stopped. This is to prevent that a half-configured state machine already begins to process triggers. Thus, the state machine is not running initially. After all configuration calls are finished, the state machine may be started via its `Start()` method. During startup, all pending configuration calls are executed to setup all elements and relations of the state machine. After this, the main loop of the state machine is started (see subsection 8.1 on page 95) and it transitions to the start state, which was passed as parameter to the constructor. Figure 28 on the following page shows an overview of all meta-states the state machine can attain. When the state machine is running, it can be paused with the `Pause()` method and resumed from the pause state with the `Resume()` method. Pausing a state machine is different than stopping it, as the state machine resides in the current state after a pause call, whereas after a stop call the

[27] http://reactivestatemachine.codeplex.com/

current state is reset to the start state. While the state machine is paused or stopped, it ignores all triggers. After a call to the `Stop()` method, the developer may again make configuration calls which are then executed during the next startup.

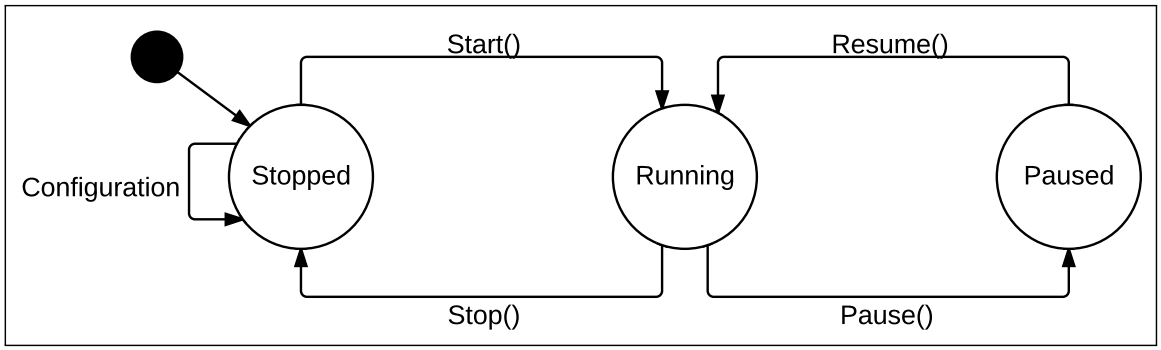

Figure 28: Meta-States of a `ReactiveStateMachine<T>` instance

7.2 States

A set of states can be represented by an arbitrary .NET type. While it is possible to use a complex type (i.e. a class or struct) or strings to denote the states, it is more common to employ a *flat* `enum` type. The advantage of using an `enum` type is that it is statically typed and spelling errors are detected at compile time. If strings were used and spelled incorrectly, the errors would only be noticed at runtime. Using a complex type to represent the states may be beneficial if specific information has to be saved with each state, which is more complicated to achieve with an `enum` type.

Listing 20 shows how an instance of a state machine is created, based on an `enum` of type `States`. The constructor of the state machine expects a name for this particular instance, which is needed for animation support, and a start state which becomes the active state as soon as the state machine is started.

```
enum States {A, B, C}

var rsm = new ReactiveStateMachine<States>("Example", States.A);
```

Listing 20: Creating a simple state machine instance

After the instantiation, no additional configuration calls have to be made to setup the states of the state machine. As states are only important in relation with transitions, it suffices to reference those states that are actually needed in the configuration calls of the transitions.

The property `CurrentState` always points to the currently active state of the state machine. Entities outside of the state machine can use this property to determine in what state the state machine is. The state machine also communicates that its state has changed via the `StateChanged` event. Any interested entity can subscribe itself to the event and get notified when the state has changed. As the `StateChanged` event can act as trigger for another state machine, it can be used to realize the aforementioned broadcast communication of the Statecharts notation.

Why the RSM currently does not support hierarchical states: In a hierarchical state machine, every state can have a random number of substates and every substate once again can have a random number of substates, eventually resulting in a hierarchical tree of states. It is usually expected that every set of states that resides in the same layer of the hierarchy is represented by a separate type. As the RSM uses a generic type parameter to represent states, it would be necessary to reserve type parameters for all these separate types at compile time. As each state machine instance has a different number of hierarchy levels, the number of generic type parameters that are actually needed can not be known at compile time. It is also not possible in .NET to specify a variable amount of type parameters. Thus, the RSM is currently not able to support hierarchical states. The support for hierarchical states in frameworks such as Swing States is usually realized by using string representations for the states, which makes it easy to nest multiple states.

7.2.1 Entry & Exit Actions

For every state of the state machine, entry and exit actions can be defined individually. As specified in the Statecharts notation, entry actions are always executed when a state is entered and exit actions are always executed when the state is exited. Besides this classic usage, entry and exit actions of the RSM can also be configured to use the guards introduced in section 3.1 on page 42. Listing 21 on the next page shows how the entry actions of figure 29 on the following page have to be configured (the definition of exit actions is analog). As can be seen from this example, the `AddEntryAction()` method has several parameter overloads that enable the different scenarios. The first parameter is always the state for which the entry action is set and the last parameter is always the entry action. The middle parameters are the optional two guards. The parameter for the entry action expects an object of type `Action` which denotes all methods that return `void` and take no parameters.

Multiple entry and exit actions can be specified for each state. When a state is entered or exited, all valid actions are determined and executed. An action is valid if its guards are evaluated positively.

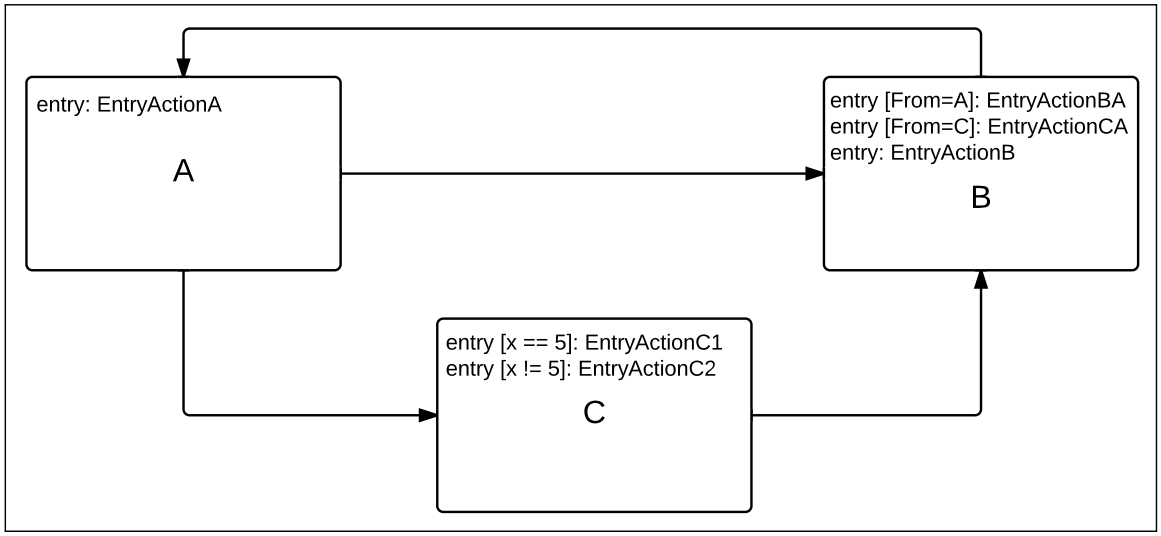

Figure 29: An example state machine with different entry actions

```
enum States {A, B, C};

var rsm = new ReactiveStateMachine<States>("Example", States.A);

//Entry Actions for State A
rsm.AddEntryAction(States.A, EntryActionA);

//Entry Actions for State B
rsm.AddEntryAction(States.B, States.A, EntryActionBA);
rsm.AddEntryAction(States.B, States.C, EntryActionBC);
rsm.AddEntryAction(States.B, EntryActionB);

//Entry Actions for State C
rsm.AddEntryAction(States.C, () => x != 5, EntryActionC1);
rsm.AddEntryAction(States.C, () => x == 5, EntryActionC2);
```

Listing 21: Specification of the entry actions of figure 29

7.3 Transitions

The RSM supports three different types of transitions: triggered transitions, timed transitions and automatic transitions. Regardless of its type, every transition has a start state and an end state and can take an optional guard and transition action. The only characteristic that separates the different types of transitions is the mechanism that initiates them: a triggered transition is initiated by a trigger, a timed transitions is initiated by a timer and an automatic transition is initiated automatically when all animations of the current state are finished. In the following, all three types are introduced in detail.

7.3.1 Triggered Transitions

A triggered transition is initiated by a trigger which usually is some kind of input event. As has been stated previously, many state machine frameworks do not make full use of all parts of input events. In the RSM, full event support is provided due to the usage of Rx. Each trigger of a triggered transition has to be an observable collection (i.e. an object of type `IObservable<TTrigger>`). It has been shown previously that native .NET events can easily be wrapped into observable collections. The type parameter `TTrigger` thereby represents the type of the event's metadata (i.e. `XXXEventArgs`). In the case of an event, the event metadata is passed as strongly typed parameter into the guard and transition action of a transition. This is one of the main differences between the RSM and other state machine frameworks which provide either no or only limited access to the event metadata. Another important disadvantage of other state machine frameworks is that they require the developer to deliver the input events to the state machine at runtime. This is not necessary with the RSM, as it is completely autonomous. After configuring the transitions and starting the state machine, it subscribes itself automatically to the observable collections of the triggers. It also makes sure that only those observable collections are active which can actually affect the current state (see subsection 8.3 on page 97 for details). This ensures that the state machine is not flooded with events that do not have any influence in the current state.

While it is natural to consider regular .NET events as triggers for the state machine, it is actually possible to use any arbitrary observable collection as trigger. Every asynchronous operation that can be wrapped inside an observable collection is therefore a potential trigger of a state machine. This opens up entirely different possibilities for state machines besides post-WIMP interaction. Consider for example a state machine that is partly driven by Twitter streams and Facebook conversations which are asynchronously pushed towards the application. Or a state machine that coordinates several asynchronous computations running on a cloud based service.

Triggered transitions are configured by means of the `AddTransition()` method. The first two parameters of this method are the start and end state of the transition. Then, the trigger of type

`IObservable<TTrigger>` follows. These first three parameters are mandatory. Next, an optional guard and transition action can be specified. The guard is of type `Func<TTrigger,bool>` which represents a method that returns `bool` and takes one parameter of type `TTrigger`. The parameter in this case is the metadata of the event. The transition action is of type `Action<TTrigger>` which represents a method that returns `void` and takes one parameter of type `TTrigger` which is again the event metadata. Listing 22 (top) shows all method overloads of the `AddTransition()` method.

In addition to these method overloads, the `AddTransition()` method also features a fluent API[28]. Such an API offers the benefit that the various parameters of the `AddTransition()` method can be offloaded to multiple method calls which each represent one particular aspect. These method calls can be chained together in arbitrary order, while the order of the parameters in the classic `AddTransition()` method is fixed. The advantage of a fluent API is its readability and comprehensibility. A well designed fluent API reads almost like a correct english sentence as every method call of the fluent API only deals with one aspect, whereas a method with several parameters usually deals with several aspects.

The fluent API in the RSM also uses the `AddTransition()` method, yet it only expects the trigger as parameter. After this, the methods `From()`, `To()`, `Where()` and `Do()` can be chained together to specify the start state, end state, guard and transition action (see listing 22 (bottom)).

```
//method overloads
rsm.AddTransition(startState, endState, trigger);
rsm.AddTransition(startState, endState, trigger, condition);
rsm.AddTransition(startState, endState, trigger, condition, action);
rsm.AddTransition(startState, endState, trigger, action);

//fluent API with all parameters
rsm.AddTransition(trigger)
   .From(startState)
   .To(endState)
   .Where(condition)
   .Do(action);
```

Listing 22: Overview of all options to configure triggered transitions

As the parameters in a fluent API are chained together with method calls, it is not possible to enforce the specification of mandatory parameters at compile time. The `From()` call could for example be omitted from the specification without causing a compile time error. This issue can only be resolved at runtime. At startup, the RSM therefore checks if all transition calls are complete and only considers those that have all parameters specified.

[28] http://en.wikipedia.org/wiki/Fluent_interface

7.3.2 Timed Transitions

A timed transition is initiated after the state machine has resided in the current state for a given timespan. While it would be possible to model such behavior with a triggered transition based on a timed observable collection, additional timer management has to be performed besides raising an event when the timer expires. With timed transitions, this timer management is done automatically without having the developer to bother about it. Timed transitions are based on a timer that is started when the state is entered. If that timer expires before another transition is made, the timed transition is initiated. If the current state is exited before the timer expired, the timer is stopped. If an internal transition is made before the timer expired, the timer is restarted.

The `AddTimedTransition()` method is used to configure a timed transition. As with triggered transitions, it first expects the start and end state of the transition. Then, the mandatory timespan has to be provided and an optional condition and transition action can be added. The `AddTimedTransition()` offers several overloads to specify all parameters. Again, a fluent API can optionally be used to specify the parameters in a more readable way. Listing 23 shows all possible method overloads and the fluent API.

```
//method overloads
rsm.AddTimedTransition(startState, endState, timeSpan);
rsm.AddTimedTransition(startState, endState, timeSpan, condition);
rsm.AddTimedTransition(startState, endState, timeSpan, condition, action);
rsm.AddTimedTransition(startState, endState, timeSpan, action);

//fluent API with all parameters
rsm.AddTimedTransition(timeSpan)
   .From(startState)
   .To(endState)
   .Where(condition)
   .Do(action);
```

Listing 23: Overview of all options to configure timed transitions

7.3.3 Automatic Transitions

Automatic transitions were introduced in subsection 3.3 on page 45 to facilitate the handling of animated transitions. An automatic transition does not depend on a trigger or timer. It is initiated automatically after all animations of the current state have finished. If no animations are associated with a state, it is initiated immediately after the current state has been entered and all entry actions have been executed. The configuration of automatic transitions is identical to the configuration of triggered or timed transitions, except that no trigger or timespan has to be

specified. Listing 24 shows all options to configure automatic transitions.

```
//method overloads
rsm.AddAutomaticTransition(startState, endState);
rsm.AddAutomaticTransition(startState, endState, condition);
rsm.AddAutomaticTransition(startState, endState, condition, action);
rsm.AddAutomaticTransition(startState, endState, action);

//fluent API with all parameters
rsm.AddAutomaticTransition()
   .From(fromState)
   .To(toState)
   .Where(condition)
   .Do(transitionAction);
```

Listing 24: Overview of all options to configure automatic transitions

7.4 Animations

Another pillar of the RSM is its support for animated transitions, based on the concept that was introduced in section 3.3 on page 45. As stated there, the behavior of the state machine during animated transitions can be modeled more precisely if animations are materialized as states. While the RSM does not treat these transitioning states separately, it is expected that animated transitions are modeled according to this concept. The specification of animated transitions is split into a logical and a visual part. While the logical part is realized with the RSM, the visual part is entirely realized with the Visual State Manager, which was introduced in section 6 on page 79. The resulting implementation thereby abides to the spirit of classic WPF development, in that the visual part is defined declaratively in XAML markup and the logical part is defined in C#.

A typical example to showcase the animation support is the appearance or disappearance of visual objects. Listing 25 on the next page shows a very simple definition of a RSM instance. It has two states (`Visible` and `Hidden`) which correspond to the visual states of the object that is to appear or disappear. This definition only specifies logically, that some trigger `T1` can transition the state machine from `Hidden` to `Visible` and some other trigger `T2` can transition it back from `Visible` to `Hidden`. Now, the VSM is used to define the visual behavior of the object that contains this state machine (see listing 26 on the following page). There, each `VisualStateGroup` corresponds to one RSM instance. The relationship is expressed by giving the same name to the `VisualStateGroup` and the RSM instance (here "VisibilityGroup"). The animations that are defined in this example ensure that the object smoothly moves into and out of the screen when the respective transition is initiated.

```
enum States {Visible, Hidden};

RSM = new ReactiveStateMachine<States>("VisibilityGroup", States.Hidden);

RSM.AddTransition(T1).From(States.Hidden).To(States.Visible);

RSM.AddTransition(T2).From(States.Visible).To(States.Hidden);
```

Listing 25: A simple RSM definition which models the visibility states of a UI element

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="VisibilityGroup">
    <VisualState x:Name="Visible">
      <Storyboard>
        <DoubleAnimation To="0" TargetProperty="(Canvas.Left)"/>
      </Storyboard>
    </VisualState>
    <VisualState x:Name="Hidden">
      <Storyboard>
        <DoubleAnimation To="-300" TargetProperty="(Canvas.Left)"/>
      </Storyboard>
    </VisualState>
    <VisualStateGroup.Transitions>
      <VisualTransition From="Hidden" To="Visible">
        <Storyboard>
          <DoubleAnimation Duration="0:0:1" To="0"
            TargetProperty="(Canvas.Left)"/>
        </Storyboard>
      </VisualTransition>
      <VisualTransition From="Visible" To="Hidden">
        <Storyboard>
          <DoubleAnimation Duration="0:0:1" To="-300"
            TargetProperty="(Canvas.Left)"/>
        </Storyboard>
      </VisualTransition>
    </VisualStateGroup.Transitions>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Listing 26: Definition of the UI element's visual appearance during an animated transition

Note, that every state in the VSM is represented by a `VisualState` element, identified by the same name as the corresponding state of the RSM. Analogously, every transition is represented by a `VisualTransition` element whose `From` and `To` properties point to the names of the respective states of the RSM. While this already establishes a conceptual mapping between both worlds, it is not yet expressed explicitly that this specific RSM instance is associated with that specific `VisualStateGroup`. This explicit mapping has to be created manually as shown in listing 27. I wanted this step to be as lightweight as possible. Thus, a developer just has to create an instance of a specific behavior, the `ReactiveStateMachineBehavior`, and add one mapping object to this behavior. The mapping object relates exactly one RSM instance (identified by the `StateMachine` property) to exactly one `VisualStateGroup` (identified by the `GroupName` property). The RSM instance can be acquired by using the default WPF data binding mechanism. It is currently not possible to establish a 1:N relation between one RSM instance and multiple `VisualStateGroups`. This has been highlighted by early adopters of the RSM, who wanted to use a single state machine to drive the animations of multiple UI elements in parallel.

```
<i:Interaction.Behaviors>
  <ReactiveStateMachine:ReactiveStateMachineBehavior>
    <ReactiveStateMachine:Mapping
      StateMachine="{Binding RSM}"
      GroupName="VisibilityGroup"/>
  </ReactiveStateMachine:ReactiveStateMachineBehavior>
</i:Interaction.Behaviors>
```

Listing 27: The logical and visual definition of the state machine are connected with a special behavior

7.5 Tracking Input Points

The RSM supports the tracking of multiple input points according to the notation that was introduced in section 3.4 on page 48. The implementation is a direct realization of the collection metaphor on which the notation is based. All operators are implemented literally, to provide the developer a seamless transition from model to implementation. As this functionality is not needed by all developers, it does not reside in the default `ReactiveStateMachine<T>` class, but in its subclass `TrackingStateMachine<T>`. This subclass implements the basic multi-point API, but redirects the actual tracking calls to a custom input point tracker of type `IInputPointTracker` which can be integrated via a plug-in mechanism. This allows the definition of input point trackers that are targeted at a specific input device. Currently, input point trackers exist for Windows Touch input and for the Microsoft Surface SDK 1.0. Listing 28 on the next page shows how the operators of the multi-point API can be used inside the guard of a transition. The example uses

the WindowsTouchStateMachine which integrates an input tracker for Windows Touch events.

```
var trigger = Observable.FromEventPattern<TouchEventArgs>(this, "TouchUp")
                        .Select(e => e.EventArgs);

var rsm = new WindowsTouchStateMachine<States>{"Example", States.A}

//#First
rsm.AddTransition(trigger).Where(e => rsm.First(e.TouchDevice));

//#Initial
rsm.AddTransition(trigger).Where(e => rsm.Initial(e.TouchDevice));

//#Intermediate
rsm.AddTransition(trigger).Where(e => rsm.Intermediate(e.TouchDevice));

//#Subsequent
rsm.AddTransition(trigger).Where(e => rsm.Subsequent(e.TouchDevice));

//#Last
rsm.AddTransition(trigger).Where(e => rsm.Last(e.TouchDevice));

//#x
rsm.AddTransition(trigger).Where(e => rsm.AtPosition(e.TouchDevice, x));

//#Contains
rsm.AddTransition(trigger).Where(e => rsm.Contains(e.TouchDevice));

//#Count == x
rsm.AddTransition(trigger).Where(e => rsm.Count == x);
```

Listing 28: Overview of all operators of the multi-point notation

8 Architecture and Implementation

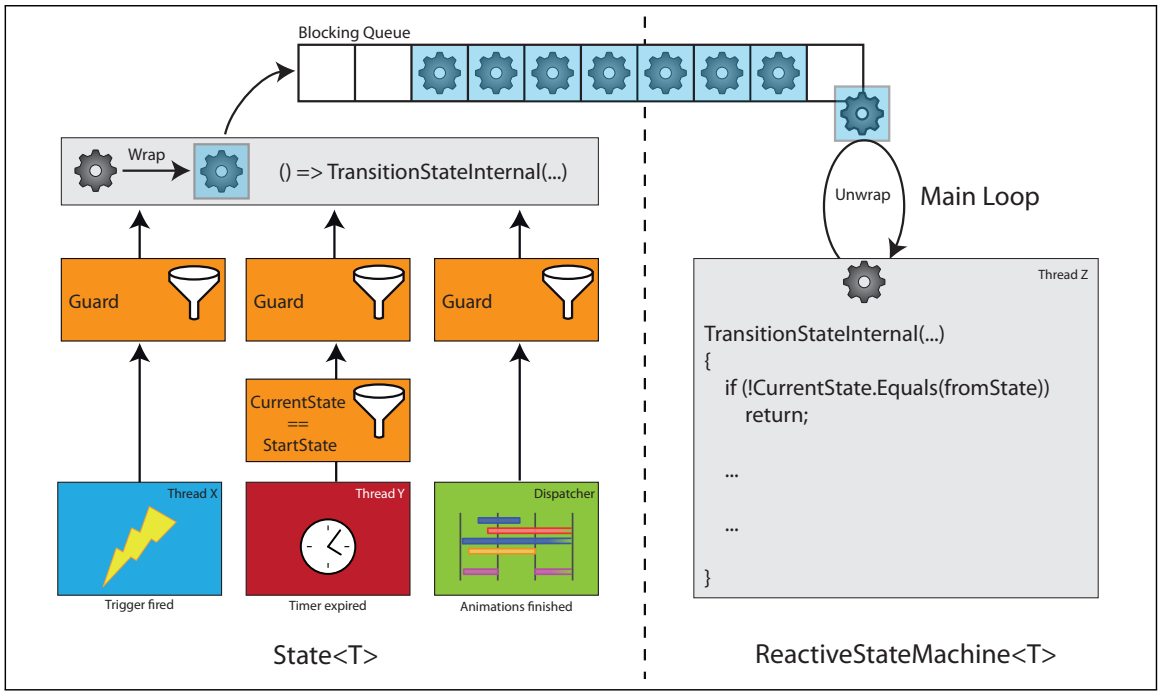

Figure 30: Diagram of the basic architecture of Reactive State Machine

8.1 Main Loop

Each instance of the `ReactiveStateMachine<T>` class has a main loop which runs in its own thread (see figure 30 on the right). This main loop is started when the `Start()` method is called and stopped when the `Stop()` method is called. The purpose of this main loop is to enable the sequential execution of the transitions of the state machine. Because the triggers of triggered transitions and timers of timed transitions may run in different threads and several of them may fire or expire at nearly the same time, there is the possibility that transitions are initiated concurrently. Concurrently running transitions however may produce unexpected behaviors and may leave the state machine in an incorrect state, which has to be avoided. To ensure that transitions are always executed sequentially, they are enqueued to a special thread-safe queue of the state machine once they are initiated by their triggers. The main loop constantly dequeues these transitions and executes them one after the other. While this design ensures that each transition is executed atomically and is not interleaved by another, it may generate situations where a transition is no longer valid once it gets dequeued, as a previous transition already transitioned the state machine to a

different state. Thus, before a transition is executed, the implementation checks if the current state of the state machine is still the same, as was previously when the transition had been enqueued. Transitions are added to the queue once their trigger fires (triggered transitions) or their timer expires (timed transitions) or when all animations of the current state have finished (automatic transitions).

8.2 Configuration

During the configuration phase of the state machine, a wrapper object of type `State<T>` is generated for every state `T` that is actually used in the state machine. Inside this wrapper object, all entry and exit actions of the respective state are stored in a collection. When the state is entered or exited, they can be retrieved from this collection by the transition mechanism. Also, all timed and automatic transitions that origin in the respective state are stored in a collection in the wrapper object. Timed transitions are retrieved from their collection when the state is entered, then their timer mechanism is started (see below). Automatic transitions are retrieved from their collection by the transition mechanism once all animations have finished. Then, they are immediately enqueued to the main loop's queue. While it is unproblematic to store timed and automatic transitions in a collection, this is not possible with triggered transitions. Because the type of their trigger `TTrigger` is not known at compile time, they cannot just be added to a statically typed collection. To resolve this issue, a subscription to the observable collection of the trigger is immediately established in the `AddTransition` method. This subscription is made using a lambda expression, which creates a closure over all important elements of the transition. Thereby they are implicitly stored in this closure as long as the subscription is active. Listing 29 on the next page shows sketchily how this mechanism works. The method that is presented there, resides in the `State<T>` wrapper class. It is called from the configuration methods in the main class. All important elements of the transition are passed into the method via the `TriggeredTransition<T, TTrigger>` parameter. At first (1), the observable collection of the trigger is stored in a collection. There it can be accessed later to enable and disable it (see below). Then (2), the subscription to the observable collection is established. The instructions inside the lambda expression are executed whenever a new value is pushed into the observable collection (i.e. when the trigger fires). There, the condition of the transition is evaluated at first (3). If it does not produce a positive value, the execution is cancelled here. Otherwise, the transition is wrapped into a lambda expression to conserve all important values (4) and finally enqueued to the main loop's queue (5).

```
void AddTransition<TTrigger>(TriggeredTransition<T, TTrigger> transition)
{
    //(1) Add observable to a collection
    _transitions.Add(transition.Trigger.Sequence);

    //(2) Subscribe to observable with a lambda expression
    transition.Trigger.Sequence.Subscribe(next =>
    {
        //(3) check the condition
        if (!transition.Condition(next))
            return;

        //(4) wrap the transition in a lambda expression
        var t = () => _stateMachine.TransitionStateInternal(
        transition.FromState, transition.ToState, next, transition.Action);

        //(5) enqueue the transition to the main queue
        _stateMachine.EnqueueTransition(t);
    });
}
```

Listing 29: Triggered transitions are immediately subscribed to conserve their parameters

8.3 Enabling/Disabling Transitions

To prevent the main loop's queue from getting flooded with transitions that have no chance of getting executed, at each point in time only those transitions are active that origin in the current state, all other transitions are inactive. Timed and triggered transitions are enabled when the state is entered and disabled when the state is exited. Automatic transitions do not participate in this mechanism, as they are initiated directly by the transition mechanism. In the following, it is explained how the enabling and disabling of timed and triggered transitions works.

Timed Transitions Timed transitions are internally implemented with timed observable collections (using the `Delay()` operator). This makes it possible to use the subscription and disposal mechanism of observable collections to enqueue the transition to the main loop and to recycle the timer. At each state entry, a new timer is created for every timed transition. Then, a subscription is made to this timer. All subscriptions of timed transitions are stored in a collection so that they can be accessed on state exit for disposal. Listing 30 on the following page shows sketchily how this mechanism is implemented. At first (1), the actual transition is stored in a lambda expression to implicitly store all important values. Then (2), the timer is created with the `Delay()` operator of the observable collection. The first `Where` filter (3) checks if the current state is still equal to the start state of the transition. This is necessary, as the state machine may already have transitioned

to the next state, rendering this transition useless. Then (4), the guard condition is evaluated. In the subscription expression (5), the actual transition is then enqueued to the main loop's queue. The whole subscription is then stored in a collection (6). At state exit, all current subscriptions are fetched from this collection and disposed (7).

```
/* -- State Entry -- */
//(1)
var t = () => _stateMachine.TransitionStateInternal(transition.FromState,
                    transition.ToState, args, transition.Action)

var subscription = Observable.Return<object>(null)
    .Delay(transition.Timespan) //(2)
    .Where(_stateMachine.CurrentState.Equals(transition.FromState)) //(3)
    .Where(transition.Condition) //(4)
    .Subscribe(_stateMachine.EnqueueTransition(t)); //(5)

_subscriptions.Add(subscription); //(6)

/* -- State Exit -- */
//(7)
foreach(var subscription in _subscriptions)
    subscription.Dispose();
```

Listing 30: Timed transitions are enabled on state entry and disposed on state exit

Triggered Transitions The mechanism of timed transitions could theoretically also be applied to the triggers of triggered transitions, as they are also based on observable collections. Yet, this is problematic for two important reasons: First, it is not possible to explicitly store the transitions. This is why a subscription to the trigger has already been established initially. If we disposed of this subscription, we could not subscribe again to the trigger as all important information would be lost. The second reason is the repeated execution of side effects. Even if we somehow managed to store the transition and to re-subscribe to its trigger, we would repeatedly execute subscription side effects. This can be very problematic, as the observable collection of the trigger is specified externally, possibly by a third party library. We can not know if the repeated execution of subscription side effects is safe and should therefore avoid it completely. To overcome both issues, the actual observable collection is wrapped at configuration time inside an object of type `IIgnoringObservable<T>`. This object is still an observable collection, but it has two additional methods, `Ignore()` and `Resume()`, which control an internal gate. If the gate is closed, no values are pushed to the observers and the whole observable is disabled. If the gate is open, all values are pushed to the observers. Thus, instead of disposing the subscription and re-subscribing again, it suffices to call the `Resume()` method of every trigger on state entry and to call the `Ignore()` method of every trigger on state exit.

8.4 Transition Flow

Regardless of its initiator (trigger, timer, automatic), an actual transition always follows the same set of instructions. This is one of the advantages of a state machine framework, as the order of these instructions can be specified internally and does not have to bother the end-user of the state machine. In the `ReactiveStateMachine<T>` class, the transition is performed in the `TransitionStateInternal` method. When a trigger fires, a call to this method is wrapped inside a lambda expression and added to the main loop's queue. It takes as parameters the start and end state of the transition, the value of the trigger (i.e. the event metadata) and the transition action. Note, that timed and automatic transitions do not have trigger values. In these cases, a `null` reference is passed into the method. The instructions of the `TransitionStateInternal` method are executed in the following order:

1. The virtual `TransitionOverride(TTrigger trigger)` method is called. This extension mechanism is explained below in more detail.
2. It is checked if the current state of the state machine is still the same as it was when the transition was initiated. If it is not, the transition is aborted here, which is safe as to this point no changes have been made to the state machine and no entry, exit or transition actions have been executed.
3. If an animation is associated with this transition, it is started. Since animations run in a different thread, we do not have to wait for the animation to complete and can continue with the following instructions.
4. The current state is exited if the transition is not internal. If the transition is internal it is not necessary to leave the state here. Exiting the state implies two things: First, all transitions that originate from the current state are disabled (as described above), as none of them has a chance to get initiated. From this point, no other transitions can be added to the main loop's queue. Second, all valid exit actions are executed. Valid exit actions are all exit actions of the current state whose conditions evaluate to true.
5. The transition action is executed (if provided). The value of the trigger is passed to the transition action as parameter.
6. The target state is entered if the transition is not internal. All valid entry actions are executed and all transitions that originate from the target state are enabled (as described above). From this point, new transitions can be added to the main loop's queue again.
7. The `CurrentState` property is set to the new state.
8. The `StateChanged` event is raised, which informs interested external entities of the transition that has just taken place.

9. If the animation that has been started previously has already finished, an optional automatic transition is immediately enqueued to the main loop's queue. If the animation is still running, a continuation action is registered that will be executed immediately after the animation signals completion. This continuation action will then enqueue an optional automatic transition to the main loop's queue.

8.5 Execution Context of Actions and Conditions

The main loop runs in its own thread or execution context. This automatically means that all actions (entry-, exit-, transition-) that are called from within the main loop are also running in this thread. In WPF, this however could cause serious problems as these actions typically affect user interface elements, which may only be affected from the thread of the `Dispatcher`. The same problem exists for the guards of a transition which are executed in the execution context of the trigger. To prevent runtime exceptions it has to be ensured that all actions and guards that affect user interface elements are enqueued to the `Dispatcher`. I wanted to relieve the developer from doing this manually, therefore all actions and guards are currently automatically wrapped inside a lambda expression that enqueues them to the `Dispatcher`. As not all actions and guards need access to user interface elements, a potential future improvement would be to let the developer choose the execution context of individual actions and guards. Thereby it would be possible to offload work from the dispatcher if the action or condition can safely run in another execution context.

8.6 Exception Handling

All actions and conditions might throw exceptions at runtime, which eventually causes the state machine to stop. As not all exceptions are critical, I chose to handle them internally, so that the state machine can continue working. Yet, as a developer may want to be informed about potential exceptions in his code, all exceptions are propagated by means of the `StateMachineException` event. Interested entities can subscribe to this event and be informed about all exceptions that are caused during the runtime of the state machine.

8.7 Extension Mechanism

An extension mechanism is provided which allows subclasses to intercept a transition. It is implemented via the virtual method `TransitionOverride(T fromState, T toState, TTrigger`

`trigger)` which is called as first instruction of a transition. While this method is empty in the base class, it can be overridden by any subclass to realize custom behaviors. The method gets passed all important information of the transition, which are the source and target states and the trigger metadata of the trigger that initiated the transition. This extension mechanism is for example used to enable the tracking of input points, but it can also be used to enable the debugging or visualization of the state machine or for other similar purposes.

8.8 Animations

It has been shown in the previous section that animations have to be created with the VSM and that the mapping between the VSM and the RSM is established via a special attached behavior. To establish this mapping internally, a special extension mechanism of the VSM is employed. It is possible to subclass the default `VisualStateManager` class and provide a custom implementation. When a user establishes a mapping between VSM and RSM, an instance of such a special subclass is created and registered as custom VSM implementation. Then, the RSM gets a reference to this custom VSM implementation which is used in the transition mechanism to start the animations of the VSM.

Once the animations of the VSM are started, they are running in parallel to the transition of the RSM. In order to start an automatic transition after the animations are finished, their end has to be signaled to the transition. This turned out to be more complicated as expected, as the VSM does not seem to have a reliable mechanism to signal the completion of its animations. Two possibilities were identified: Either the storyboard of the animation signals completion through its `Completed` event, or the VSM signals completion through its `CurrentStateChanged` event. To reduce the amount of coordination code that usually needs to be written in such cases, I chose to wrap the entire execution of the VSM animations inside a `Task<T>` object. This object represents a task that is running in a separate thread. Inside this task, the coordination mechanisms for the two events are established. If either of the two signals the completion of the transition, the task finishes. Thus, at the end of the transition in the RSM it is only checked if the task of the animations has already finished. If it has, a potential automatic transition can be enqueued. If it has not, the `ContinueWith()` method of the task object is used to schedule the enqueueing of the automatic transition to the point where the task has finished.

Fortunately the VSM helps to deal with another tricky concurrency issue: the canceling of a running transition. As explained in subsection 3.3 on page 45, it must be possible to escape from the current transitioning state while an animated transition is running. For the VSM it is a common scenario to cancel a currently running transition and proceed with the next one. It automatically takes care of canceling the old animation(s) and starting the new one(s). Thus, no coordination code had to be written for this scenario inside the RSM.

8.9 Tracking Input Points

The tracking of input points is enabled by the extension mechanism of the RSM that was presented previously. As shown in listing 31, the `TransitionOverride()` method is overridden by a subclass of the `ReactiveStateMachine<T>` class. This method is notified of each transition that the state machine performs. As it gets passed the metadata of the event trigger, it is possible to establish the input point collection that is needed for the input point notation. The actual tracking of the input points is realized by a special input tracker that is targeted at the respective input device.

```
void TransitionOverride(T fromState, T toState, TTrigger trigger)
{
    if (trigger is EventArgs)
        InputTracker.Track(trigger as EventArgs);
}
```

Listing 31: The tracking mechanism overrides the `TransitionOverride()` method to intercept transitions

Listing 32 exemplarily shows an input tracker implementation for Windows Touch events. In its `Track()` method, the input tracker determines the action of each event and adds or removes the respective input point to its internal collection.

```
public override void Track(EventArgs e)
{
    //we only accept Touch Events
    if (!(e is TouchEventArgs))
        return;

    var args = e as TouchEventArgs;

    var contactAction = args.TouchDevice.GetTouchPoint(null).Action;

    if (contactAction == TouchAction.Down)
        AddPoint(args.TouchDevice);
    else if (contactAction == TouchAction.Up)
        RemovePoint(args.TouchDevice);
}
```

Listing 32: Input point tracker implementation for Windows Touch input

9 Use Cases

9.1 Facet-Streams - The Wheel

The Facet-Streams project was the actual initiator for the creation of the Reactive State Machine framework. Facet-Streams is a tabletop system that supports the collaborative and faceted search of products using physical tokens and multi-touch interaction. It was created in cooperation with Microsoft Research Cambridge and published at the 2011 CHI conference [Jetter et al., 2011]. The overall design and implementation of Facet-Streams can be looked up in the technical report of my master project [Zöllner, 2011a].

In Facet-Streams, we had to create two fairly complex multi-touch controls (the `FacetWheel` and the `ValueWheel`). During the design of the controls' interaction, we came to a point where things got so complicated that we had to draw them in a structured way onto a whiteboard. We eventually ended up with a finite-state machine that captured the entire interaction. In this stage, we were also forced to come up with the multi-point notation for state machines, as certain aspects of the interaction required the differentiation between different fingers. When we wanted to implement the state machine that we had drawn, we had troubles finding a suitable implementation technique. As the then state machine frameworks did not cater our needs, we ended up implementing the state machine manually with a semi-structured low-level technique. Although the state machine did its job correctly, we were not quite satisfied with the implementation, as it was very complex and therefore hard to comprehend and maintain. As a consequence, the Reactive State Machine framework has been created to replace the old implementation. In the revised version of Facet-Streams, the Reactive State Machine is now fully integrated. With this new implementation, the maintainability of the state machine improved significantly: While the old implementation had about 850 lines of code, the revised implementation with the Reactive State Machine framework comprises about 350 lines of code. This is a saving of 500 lines of code or about 60%, which clearly indicates the value of using a state machine framework. Because of this improved maintainability, new developers have been introduced into the current implementation in a very short amount of time.

In the following, the state machine model of the two controls and its implementation are discussed. The basic interaction with these controls is simple: The user puts a glass token onto the tabletop and a visualization around this token appears (see figure 31 on the following page). To initiate the `FacetWheel`, the orange label has to be touched and to initiate the `ValueWheel`, the blue label has to be touched. When the label is touched, the respective wheel smoothly fades to full opacity, thereby overlaying parts of the former visualization (see figures 32 and 33 on the next page). While the wheel is visible, its segments can be selected with one or multiple fingers. The wheel stays visible as long as at least one finger resides on it, then it smoothly fades out again.

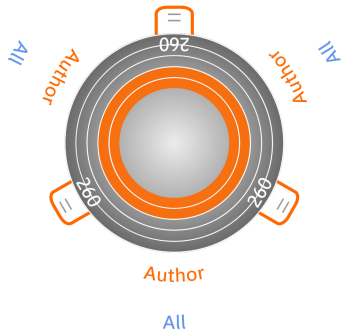

Figure 31: `FacetWheel` in the *FacetWheelCollapsed* state

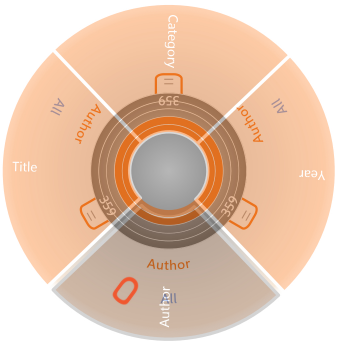

Figure 32: `FacetWheel` in the *FacetWheelFadingIn* state

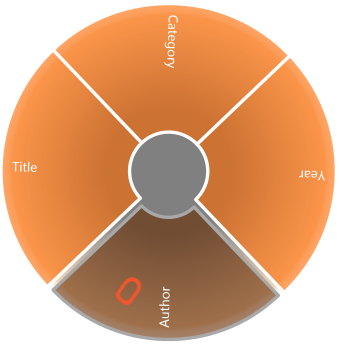

Figure 33: `FacetWheel` in the *FacetWheelVisible* state

While this brief description covered all basic aspects of the control's lifecycle, there are many details in the interaction that need to be described more concisely. The state machine in figure 34 on the following page provides such a precise and formal description for the `FacetWheel`. In its entirety the model looks very complex. Yet, it can be partitioned into two separate aspects that can be treated individually: The first aspect is the overall lifecycle of the wheel. It is discussed more thoroughly in subsection 9.1.1 on page 106. The second aspect is the selection behavior of the wheels which is discussed in subsection 9.1.2 on page 107.

After this conceptual introduction into the interaction design of the wheels, it is shown in subsection 9.1.3 on page 109 how the Reactive State Machine framework facilitates the implementation of the underlying state machines in comparison to the low-level implementation technique that was employed in the first version of Facet-Streams.

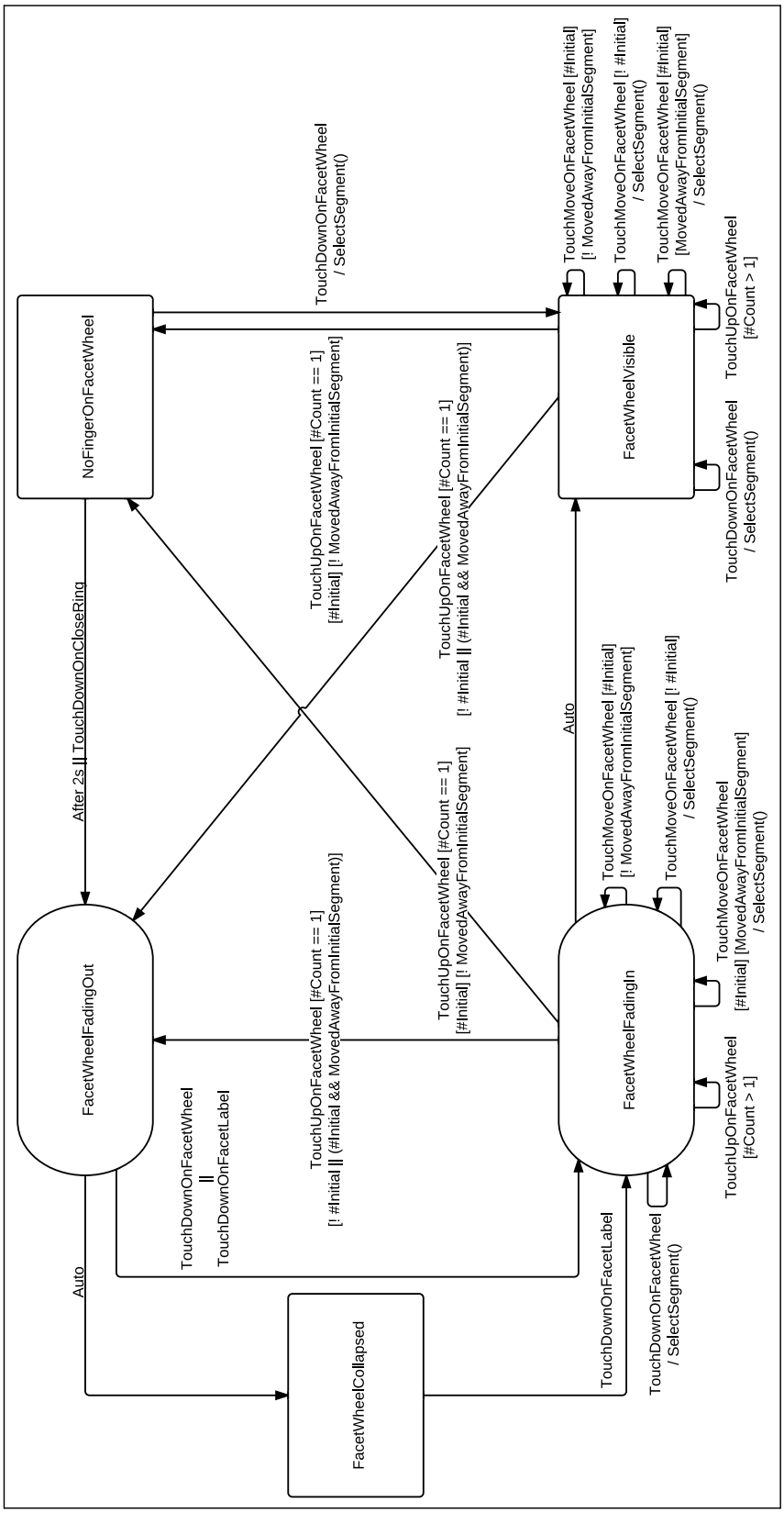

Figure 34: State machine model of the `FacetWheel`

9.1.1 Lifecycle

To explain the wheels' lifecycle, all transitions that are used to implement the selection behavior can be omitted for a moment from the state machine. The model of figure 35 shows the state machine after this step. A wheel can be in any of five states at a particular moment. Initially the wheel is in the *FacetWheelCollapsed* state, where it is invisible. As soon as a user puts his finger onto the orange label (the `FacetLabel`), the state machine transitions to the *FacetWheelFadingIn* state. In this transitioning state, which is indicated by the shape, an opacity animation smoothly fades the control to full visibility. If the animation finishes before a transition has been made to another state, the automatic transition to the *FacetWheelVisible* state is then initiated. In both the *FacetWheelFadingIn* and *FacetWheelVisible* state, the behavior of the state machine is identical when fingers are added or removed from the wheel: When an additional finger is added, the state machine remains in the current state. The same is true when a finger is removed and other fingers are still on the wheel (indicated by `[#Count > 1]`).

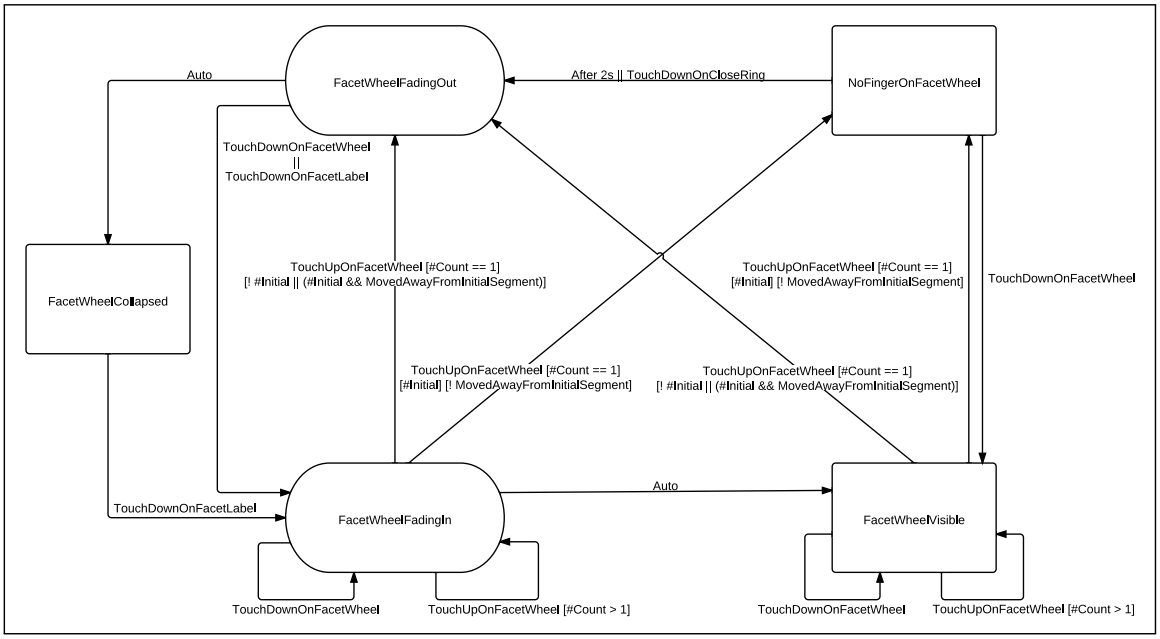

Figure 35: State machine model of the lifecycle of the `FacetWheel`

Things get more complicated when the last finger is removed (indicated by `[#Count == 1]`). Here, two cases have to be considered: If the removed finger is the finger that was initially put as first finger onto the wheel (indicated by `[#Initial]`), and it did not move away from its initial segment, then the state machine transitions to the *NoFingerOnFacetWheel* state. In this case we can be sure that no selection has been made by the user, as he did not move its initial finger away from its initial segment. He is therefore given another two seconds (in the *NoFingerOnFacetWheel*

state) to make a selection, before the wheel fades out again. If however the removed finger is not the initial finger or it is the initial finger but has moved from its initial segment, then the state machine transitions directly to the *FacetWheelFadingOut* state. In this case we can be sure that the user has made a selection, therefore we can accelerate the interaction by directly fading out the wheel. Note, that this behavior is different in the `ValueWheel`. As multi-selection is allowed there, we can not close the wheel after a selection has been made. We therefore always transition the state machine to the *NoFingerOnValueWheel* state if the last finger was removed. If the user wanted to accelerate the closing of the wheel he has to employ the close ring (see below).

When the user resides in the *NoFingerOnFacetWheel* state, he can go back to the *FacetWheelVisible* state by putting a finger on the wheel. If he does not put a finger on the wheel, the state machine automatically transitions to the *FacetWheelFadingOut* state after a timespan of two seconds has expired. The same transition from the *NoFingerOnFacetWheel* state to the *FacetWheelFadingOut* state is also made, when the close ring is pressed. This invisible ring is positioned outside of the wheel. It was introduced as an acceleration mechanism for expert users. Note, that the triggers of this transition are collapsed onto one arrow to reduce the visual complexity of the model.

In the *FacetWheelFadingOut* state, the wheel finally fades out again with a smooth opacity animation. If the animation completes, the automatic transition to the *FacetWheelCollapsed* state is initiated. If the user wants to intercept the fade out animation he can do so by putting his finger onto the wheel or the `FacetLabel`. This transitions the state machine back to the *FacetWheelFadingIn* state, where the wheel is eventually animated back to full opacity.

9.1.2 Selection Behavior

The wheels contain several segments which can be selected with the finger (see figure 36 on the next page for a screenshot of the `FacetWheel`). We wanted to achieve a rather natural selection behavior, especially when multiple fingers are acting in parallel. During the fine-tuning of the interaction we observed that our bimanual interaction with the control always followed a similar scheme: The initial finger, which was used to open the wheel, would rest on the position were it was placed initially and a finger of the other hand would select or deselect segments on the wheel. This is actually a rather common behavior of bimanual interaction in the physical world, where the non-dominant usually holds firm the object and the dominant hand is used for more fine-grained interaction. Yet, in our case this was problematic at first, as jitter or small movements of the initial finger caused the segment under this finger to be selected or deselected, although the intention of the user was different. To prevent this, we had to differentiate between the initial and the other fingers. Now, the initial finger is only allowed to select a segment if it moved beyond the segment which was initially underneath it (indicated by the predicate `[#Initial] [MovedAwayFromInitialSegment]`). Moving the finger this far clearly indicates that the user wants to perform a selection. If it is only moved inside the initial segment no selection is made

(indicated by the predicate `[#Initial] [!MovedAwayFromInitialSegment]`). All subsequent fingers are always allowed to select the respective segment, regardless of whether they moved beyond their initial segment (indicated by the predicate `[!#Initial]`).

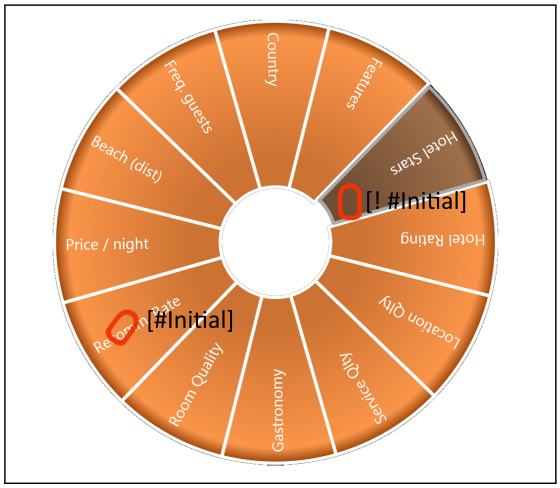

Figure 36: Two fingers affecting a `FacetWheel`

Figure 37 shows a small subset of the state machine of the `FacetWheel` which models the selection behavior with the help of the multi-point notation. Note, that it suffices to consider the *FacetWheelVisible* state as the behavior is identical in the *FacetWheelFadingIn* state.

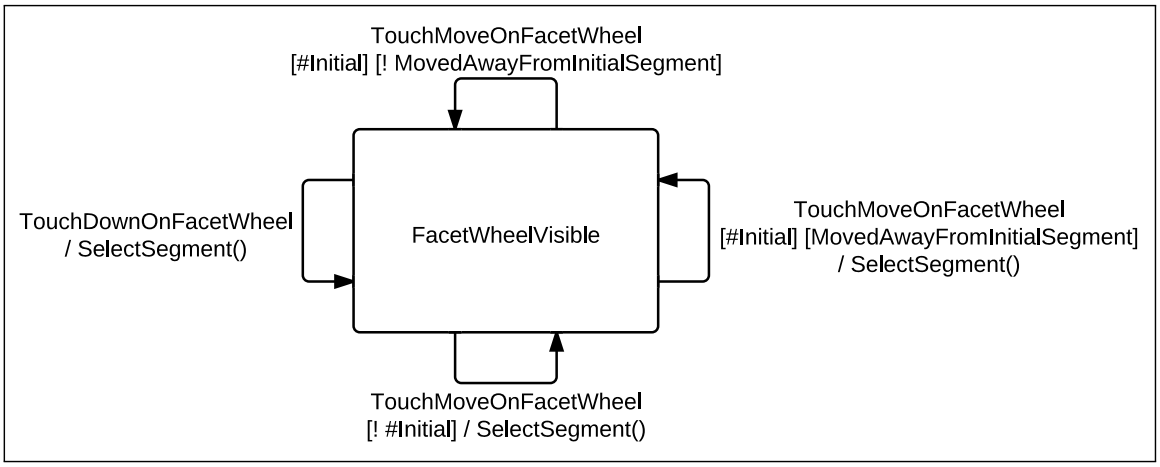

Figure 37: State machine subset of the selection behavior

9.1.3 Implementation

In this subsection, the implementation of the wheels' state machine is described. Instead of just focusing on the current implementation, which is based on the Reactive State Machine framework, I want to compare it to the old implementation, which was based on a low-level implementation technique. This comparison reveals several significant differences which show why a state machine framework is advantageous over low-level implementation techniques. Because of the sheer size of the state machine, it is however not possible to cover all implementation details. Instead, I want to focus on just one particular aspect here, which vividly describes the differences between the old and the new implementation. This aspect is the lifting of a finger in the *FacetWheelVisible* state. Listing 33 shows the current implementation with the RSM in its entirety. Listing 34 on the following page shows one part of the old implementation. The other part is presented in listing 35 on page 112.

```
fsm.AddTransition(TouchUpOnFacetWheel)
    .From(States.FacetWheelVisible).To(States.FacetWheelVisible)
    .Where(e => fsm.Count > 1);

fsm.AddTransition(TouchUpOnFacetWheel)
    .From(States.FacetWheelVisible).To(States.NoFingerOnFacetWheel)
    .Where(e => fsm.Count == 1)
    .Where(e => fsm.Initial(e) && ! MovedAwayFromInitialSegment(e));

fsm.AddTransition(TouchUpOnFacetWheel)
    .From(States.FacetWheelVisible).To(States.FacetWheelFadingOut)
    .Where(e => fsm.Count == 1)
    .Where(e =>
    {
     return (! fsm.Initial(e) ||
           (fsm.Initial(e) && MovedAwayFromInitialSegment(e)));
    }

fsm.AddTimedTransition(TimeSpan.FromSeconds(2))
    .From(States.NoFingerOnFacetWheel).To(States.FacetWheelFadingOut);
```

Listing 33: Current implementation of a state machine subset of Facet-Streams

```
private void OnFacetWheelContactUp(object sender, ContactEventArgs e)
{
    switch (CurrentState)
    {
        case States.FacetWheelVisible:

            if (contactStore.GetContacts().Count() == 1)
            {
                if(contactStore.First() == e.Contact)
                {
                    if (MovedAwayFromInitialSegment(e))
                    {
                        CurrentState = States.FacetWheelFadingOut;
                        facetWheel.AnimateOpacity(1, 0, 1000);
                    }
                    else
                    {
                        facetWheelVisibleTimer.Start();
                    }
                }
                else
                {
                    CurrentState = States.FacetWheelFadingOut;
                    facetWheel.AnimateOpacity(1, 0, 1000);
                }
            }

            contactStore.Remove(e.Contact);
            break;

        ...
    }
}
```

Listing 34: Old implementation of a state machine subset of Facet-Streams

Determining the transition The first important difference between the two implementations is the way transitions are determined. In the old implementation, a combination of `switch/case` and `if/else` control structures were used to determine the correct transition, which creates a lot of confusing spaghetti code. In the new implementation, the conditions that are defined in the graphical model can be transferred directly into the implementation, without changing their representation. These conditions are also directly attached to a specific transition, which greatly facilitates the readability, compared to the distributed code of the old implementation.

Transition order In the old implementation, the order of operations that take place during a transition has to be established manually by the developer. This includes the setting of the `CurrentState` variable, the execution of transition actions and the starting of animations. In the new implementation, the developer is completely relieved from this task, as the execution of the transition is taking place inside the framework. There it is ensured that the order of the transition's operations is always the same. The developer just has to provide the relevant information in the definition of the transition.

Timer management The reader may have noticed that the two implementations are actually not based on exactly the same state machine model. In the old version of Facet-Streams, there was no *NoFingerOnFacetWheel* state, instead the functionality of this state was included in the *FacetWheelVisible* state. The reason for this is that the timer that triggers the transition to the *FacetWheelFadingOut* state had to be managed explicitly, which is easier when the different calls to the timer (Start, Stop, Tick) are not distributed over several states. Yet, the explicit management of the timer nonetheless requires several extra steps of the developer: It has to be started manually in the event handler of the `TouchUp` event (see listing 34 on the preceding page) and its `Tick` event has to be handled separately (listing 35 on the next page). In this event handler, once again the `switch/case` decision logic has to be employed to determine the correct transition. Compared to the compact one-line definition of the timed transition in the new implementation, the explicit managing of the timer requires huge efforts. It also increases the distribution of functionality on additional event handler methods which further complicates the overall implementation.

Input point management The correct realization of the multi-point notation in the old implementation required us to explicitly add and remove input points from the input point collection in every event handler method (e.g. `contactStore.Remove(e.Contact)`). In the new implementation, this functionality is handled automatically in the state machine. The developer just has to ensure that the respective transition is defined.

```
private void OnFacetWheelTimerTick(object sender, EventArgs e)
{
    (sender as DispatcherTimer).Stop();

    switch (CurrentState)
    {
        case States.FacetWheelVisible:
            CurrentState = States.FacetWheelFadingOut;
            facetWheel.AnimateOpacity(1, 0, 1000);
            break;

        ...
    }
}
```

Listing 35: Old implementation of the timer event handler in Facet-Streams

Animations The management of animations in the old implementation was rather complex compared to the current implementation. Every step, from setting up the animation in the first place to starting and stopping it, had to be handled explicitly. Thereby, the implementation had to ensure that an already running animation is stopped, once another animation is started. Except for the definition of the animation, all other steps are not necessary in the new implementation, as the Visual State Manager takes care of these under the covers. Listing 36 shows how the animations of the VSM are mapped to the respective state machines in the code-behind and listing 37 on the next page exemplarily shows the definition of the animations that are executed automatically when the state machine transitions from the *FacetWheelVisible* to the *FacetWheelFadingOut* state.

```
<i:Interaction.Behaviors>

  <ReactiveStateMachine:ReactiveStateMachineBehavior>

    <ReactiveStateMachine:Mapping GroupName="FacetWheelStates"
      StateMachine="{Binding FacetWheelStateMachine}" />

    <ReactiveStateMachine:Mapping GroupName="ValueWheelStates"
      StateMachine="{Binding ValueWheelStateMachine}" />

  </ReactiveStateMachine:ReactiveStateMachineBehavior>

</i:Interaction.Behaviors>
```

Listing 36: XAML code to map the VSM to the RSM

```
<VisualStateGroup Name="FacetWheelStates">
  <VisualTransition From="FacetWheelVisible" To="FacetWheelFadingOut">
    <Storyboard Duration="00:00:01">

      <ObjectAnimationUsingKeyFrames Duration="00:00:00"
        Storyboard.TargetName="closeRing"
        Storyboard.TargetProperty="Visibility">
        <DiscreteObjectKeyFrame>
          <DiscreteObjectKeyFrame.Value>
            <Visibility>Collapsed</Visibility>
          </DiscreteObjectKeyFrame.Value>
        </DiscreteObjectKeyFrame>
      </ObjectAnimationUsingKeyFrames>

      <DoubleAnimation Duration="00:00:01"
        Storyboard.TargetName="facetWheel"
        Storyboard.TargetProperty="Opacity"
        To="0" />

      <ObjectAnimationUsingKeyFrames Duration="00:00:00"
        BeginTime="00:00:00.9"
        Storyboard.TargetName="facetWheel"
        Storyboard.TargetProperty="Visibility">
        <DiscreteObjectKeyFrame>
          <DiscreteObjectKeyFrame.Value>
            <Visibility>Collapsed</Visibility>
          </DiscreteObjectKeyFrame.Value>
        </DiscreteObjectKeyFrame>
      </ObjectAnimationUsingKeyFrames>

    </Storyboard>
  </VisualTransition>
</VisualStateGroup>
```

Listing 37: XAML definition of the VSM animations

Event Triggers The interaction of both wheels is mainly driven by touch events. The initial version of Facet-Streams was targeted at the first generation of the Microsoft Surface tabletop. It therefore used the special `Contact` events of the Surface SDK. The revised version of Facet-Streams is targeted at the second generation of the Microsoft Surface, whose SDK permits to use the built-in touch events of WPF. In the old state machine implementation, all events have been handled explicitly by means of event handler methods. In these methods, the operations of the state machine were then executed (as shown previously in listing 34 on page 110). In the new implementation, the events are handled internally by the state machine. They just have to be defined once as trigger of a transition. In order to do so, they first have to be wrapped inside observable collections. Usually a mechanism, such as in listing 38 is applied to wrap an event into an observable collection.

```
IObservable<TouchEventArgs> touchDown;
touchDown = Observable.FromEventPattern(uiElement, "TouchDown");
```

Listing 38: Wrapping a `TouchDown` event inside an observable collection

With this approach, the class where the state machine is implemented needs direct access to the user interface elements. To achieve this, the user interface code and the interaction logic have to be coupled very tightly. Because we did not want to have such a tight coupling, we employed the MVVM[29] pattern which uses WPF's data-binding mechanism to couple the user interface elements of the *View* in a loose manner to the interaction logic of the *ViewModel*. As a consequence, we also had to use a different approach to wrap the events into observable collections. The idea of this approach was to use a kind of mediator which can be instantiated in XAML. It handles the respective event and forwards it to an observer that is residing in the *ViewModel* and provided via data-binding. Listing 39 shows how this mediator (the `EventToObserver`) is used inside the user interface definition.

```
<i:EventTrigger EventName="TouchDown">
  <util:EventToObserver Observer="{Binding TouchDownOnFacetWheel}" />
</i:EventTrigger>

<i:EventTrigger EventName="TouchUp">
  <util:EventToObserver Observer="{Binding TouchUpOnFacetWheel}" />
</i:EventTrigger>
...
```

Listing 39: The `EventToObserver` element forwards input events to an observer

[29] http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx

In the *ViewModel*, where the state machine is defined, the observers have to be defined as properties, to ensure that they can be data-bound (see listing 40). While plain observers can not be used as triggers for the state machine, the observers that are used in this example are actually also observable collections. They are represented by the type `Subject<T>` which implements both the `IObservable<T>` and `IObserver<T>` interface. Thus, the observer component subscribes itself to the input event and forwards its values via its observable component. Interested entities can again subscribe themselves to these updates by means of the `Subscribe()` method of `IObservable<T>`. Figure 38 shows the complete mechanism that is used inside Facet-Streams to get the event from the user interface into an observable collection and finally into the state machine.

```
public Subject<TouchEventArgs> TouchDownOnFacetWheel
{
    get;set;
}

public Subject<TouchEventArgs> TouchUpOnFacetWheel
{
    get;set;
}
...
```

Listing 40: Events are represented by properties which are data-bound to the `EventToObserver`

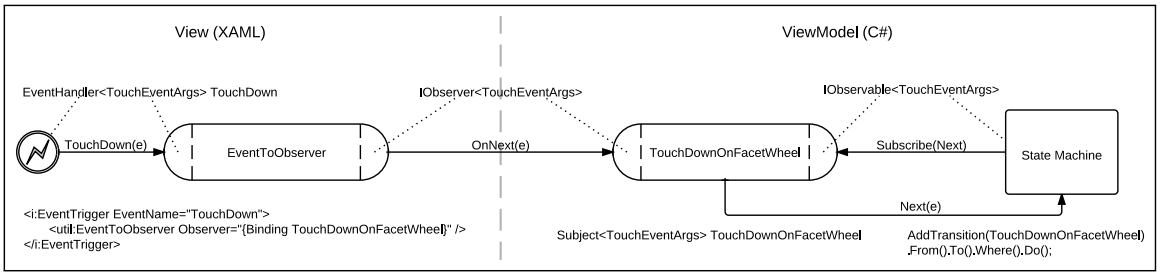

Figure 38: The entire mechanism to forward an event from the user interface to the state machine

9.2 Facet-Streams - The `ReleaseLink` Behavior

The `ReleaseLink` behavior has already been discussed two times before. While the naive implementation with timers in subsection 2.2.3 on page 31 requires the explicit management of the timer and the distribution of the functionality on several event handler methods, the implementation with observable collections in subsection 5.3.2 on page 78 requires substantial knowledge of the Rx operators. The most comprehensible implementation can be realized with a finite-state machine (see figure 39). Here, three states are used to differentiate the different possibilities. In the initial *Up* state, a `TouchDown` trigger transitions the state machine to the *Down* state. If the `TouchUp` trigger is fired within a second, the state machine transitions back to the *Up* state. If the timespan expires before a `TouchUp` trigger occurs, it transitions to the *Released* state, and the link is released in the transition action. Note, that the state machine of this behavior does not contain a transition which points away from the `Released` state. This is perfectly fine, as the state machine is destroyed once the link is released and created new when it is connected again.

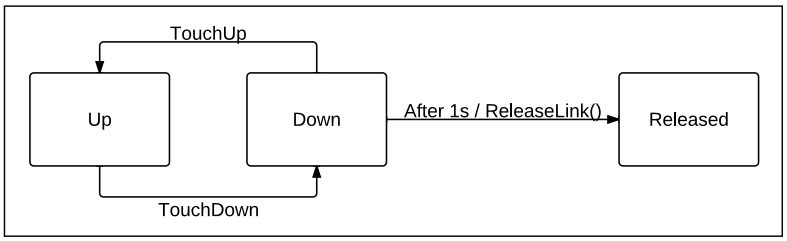

Figure 39: State machine model of the releasing of a link

Listing 41 on the following page shows the implementation of the behavior with the Reactive State Machine framework. Again, the respective events are first wrapped inside observable collections (1). Then, the state machine and its transitions are specified, using the `AddTransition()` and `AddTimedTransition()` methods (2).

As in the implementation with Rx, no additional code has to be written to manage the underlying timer. Yet, as the behavior is modeled with a state-transition diagram, the developer only needs to know how to specify triggers and transitions, whereas the Rx implementation requires the developer to know some advanced operators that are only difficult to visualize. Also, the specification of the state machine forces the designer to explicitly formulate what the behavior does and therefore makes it possible that all stakeholders talk about it on an abstract level. As the resulting implementation exactly mirrors the model, it is easy to comprehend and maintain.

```
//(1) wrap the touchDown and touchUp events inside observable collections
touchDown = Observable.FromEventPattern(AssociatedObject, "TouchDown")
                      .Where(e => e.TouchDevice.GetIsFingerRecognized());

touchUp = Observable.FromEventPattern(AssociatedObject, "TouchUp")
                    .Where(e => e.TouchDevice.GetIsFingerRecognized());

//(2) specify the state machine and its transitions
var fsm = new ReactiveStateMachine<States>(Up);

fsm.AddTransition(touchDown).From(States.Up).To(States.Down);

fsm.AddTransition(touchUp).From(States.Down).To(States.Up));

fsm.AddTimedTransition(Delay).From(States.Down).To(States.Released)
   .Do(ReleaseLink);
```

Listing 41: Implementation of the `ReleaseLink` behavior with the RSM

9.3 SmartShare

In the course *Blended Interaction*, held at the University of Konstanz in the Winter Term 2011/2012, the Reactive State Machine framework was used by one group to support the implementation of specific interactive behaviors of their post-WIMP multi-user system *SmartShare*. The prototype was based on an existing project of Klinkhammer et al. [Klinkhammer et al., 2011]. In this project, personal territories are created for every user standing around a tabletop. With the help of infrared tracking, each personal territory was configured to follow its user closely. Klinkhammer et al. used these personal territories in a museum exhibition to allow visitors to simultaneously explore the information items of the tabletop. In SmartShare, which is targeted at exhibitors of business fairs, the functionality of the personal territory is extended with a data exchange mechanism and a login mechanism based on QR codes and smartphones. The goal of SmartShare is to support a straightforward data exchange between exhibitors and potential clients.

Figure 40 on the next page shows the physical setting of the system. Each of the two users in the picture has a personal territory in front of him, which is divided into three parts (see figure 41 on the following page): On the left side (1), identification information of the user is displayed when he is logged in. The central part (2) is used to receive information items from a smartphone or a cloud storage service. From there they can be dragged to the main information area of the table. The right part (3) resembles a shopping cart. Information items of the table can be put into this shopping cart and sent to a smartphone or a cloud storage service. Each of these three parts is controlled by one or several state machines of the Reactive State Machine framework. In the following, the model and implementation of these is presented more detailly.

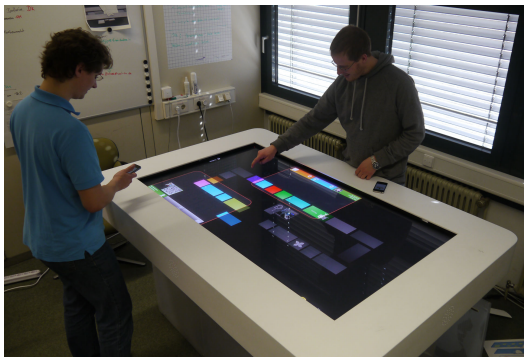

Figure 40: Physical setting of SmartShare

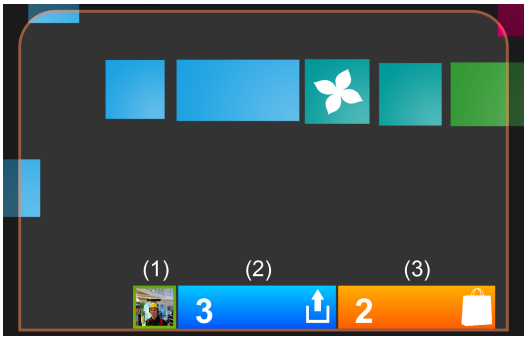

Figure 41: Personal territory of a user

9.3.1 The Login Control

SmartShare uses a login procedure to identify and differentiate the users of the system. With this, users not only know with whom they are currently interacting, it also enables the exchange of data between the user and the table. It is additionally planned to associate roles with each user, to allow different actions for different users. A user can trigger the login procedure by touching the green user icon of the personal territory (figure 42a). A QR code appears which the user has to scan with a custom app of his smartphone (figure 42b). When the QR code is scanned successfully, the user is logged in and the image and name of the user appear in the bottom left corner of the personal territory (figure 42c). Also the exchange mechanisms in the center and right part of the personal territory become active, which is indicated by a change in color.

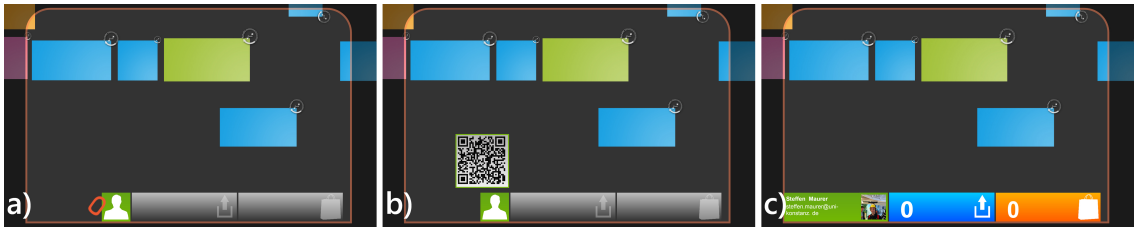

Figure 42: Overview of the login procedure in SmartShare

The login control itself consists of three separate controls, the `PictureControl`, the `QRControl` and the `NameControl`. The `PictureControl` contains the image of the user, the `QRControl` contains the image of the QR code and the `NameControl` contains two `TextBlock` controls showing the name and eMail address of the user. As the `PictureControl` is always visible, it does not need a state machine. The two other controls however can appear or disappear individually, therefore they are both backed by a separate state machine. Figure 43 on the following page shows both state machines, the left being that of the `QRControl` and the right that of the `NameControl`.

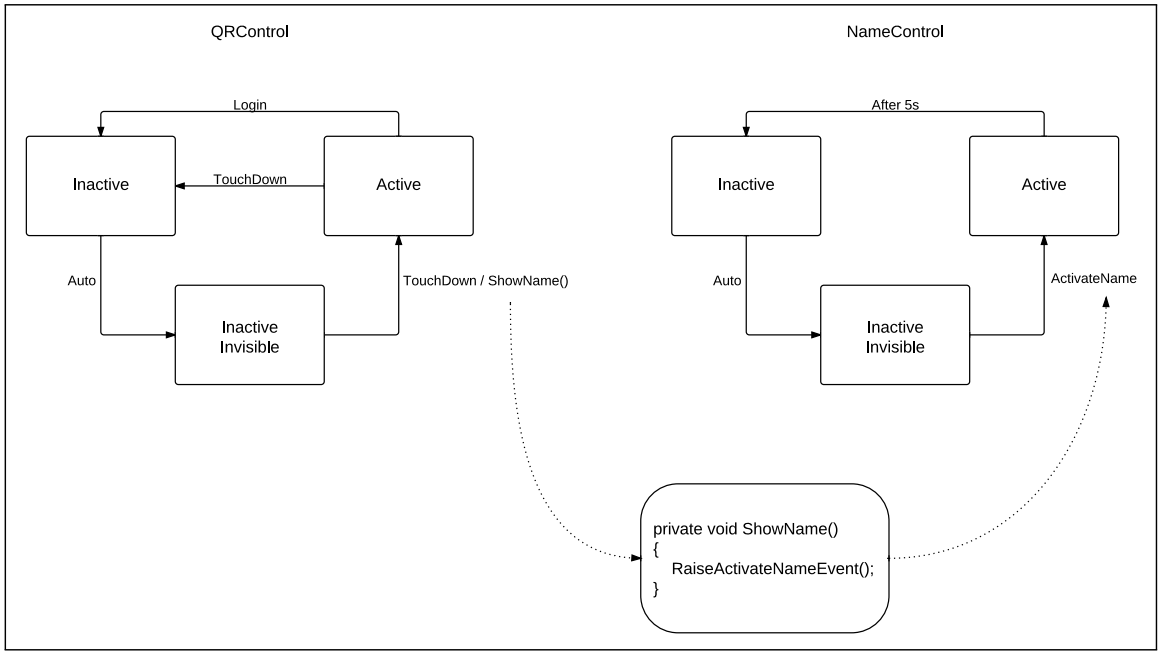

Figure 43: Overview of the state machines that control the `QRControl` and the `NameControl`

Both state machines have the same set of states: *Active, Inactive* and *InactiveInvisible*. Initially each control is in the *InactiveInvisible* state, where it is not visible. The `QRControl` transitions to the *Active* state when the picture representation of the user is touched. The opacity animation that is associated with this transition smoothly fades in the `QRControl`. To transition the control to the *Inactive* state, either the picture representation of the user has to be touched again or the user has to log himself in. To log in, the QR code has to be scanned with a special app of the smartphone. The smartphone automatically establishes a connection to the table and signals the recognition of the QR code. The main model of the application then raises the *Login* event which triggers the transition from the *Active* to the *Inactive* state. This transition is also animated, using the reverse opacity animation. After the animation is finished, the state machine automatically transitions to the *InactiveInvisible* state.

The state machine of the `NameControl` is almost identical to the that of the `QRControl`. The main difference is that it uses different triggers to transition the state machine to and away from the *Active* state. The transition to the *Active* state is triggered by the `ActivateName` event. This event is raised in the `ShowName()` method which is called by two different entities of the system. First, when the `QRControl` transitions from the *InactiveInvisible* state to the *Active* state, the transition action calls the `ShowName()` method (see figure 43). This ensures that the `NameControl` gets active whenever the *QRControl* is activated. Second, the `NameControl` of all personal territories are activated whenever a new user logs in. This mechanism ensures that the newly logged in

user can read the names and eMail addresses of all collaborators and all collaborators can read the name and eMail address of the newly logged in user. To achieve this, every personal territory listens to a global *UserConnected* event and in reaction calls the `ShowName()` method. During the transition to the *Active* state, the `NameControl` is animated smoothly from the bottom of the screen. While the `QRControl` had to be closed explicitly, the `NameControl` is closed automatically after it was in the *Active* state for five seconds. Again the reverse animation moves the control out of the screen and the state machine transitions back to the *InactiveInvisible* state after the animation finished.

While the underlying state machines of these controls are not as complex as in Facet-Streams, the developers of SmartShare established a broadcast communication mechanism between two state machines as suggested in the Statecharts notation. The implementation of both state machines (see listing 42 and 43) is rather straightforward, as only simple elements are used.

```
var touchDown = Observable.FromEventPattern(this.MainGrid, "TouchDown");

var login = Observable.FromEventPattern(this.DataContext, "Login");

fsm.AddTransition(touchDown)
   .From(States.InactiveInvisible).To(States.Active)
   .Do(e =>ShowName());

fsm.AddTransition(touchDown).From(States.Active).To(States.Inactive);

fsm.AddTransition(login).From(States.Active).To(States.Inactive);

fsm.AddAutomaticTransition()
   .From(States.Inactive).To(States.InactiveInvisible);
```

Listing 42: Implementation of the state machine of the `QRControl`

```
var acivateName = Observable.FromEventPattern(owner, "ActivateName");

fsm.AddTransition(acivateName)
   .From(States.InactiveInvisible).To(States.Active);

fsm.AddTimedTransition(TimeSpan.FromSeconds(5))
   .From(States.Active).To(States.Inactive);

fsm.AddAutomaticTransition()
   .From(States.Inactive).To(States.InactiveInvisible);
```

Listing 43: Implementation of the state machine of the `NameControl`

9.3.2 Getting Data onto the Table

Once a user is logged in, he can get data onto the table using his smartphone or a cloud storage service. Currently, only images are supported to demonstrate the functionality, but it is planned to add support for digital business cards and other data types. A special app on the smartphone lets the user select the image he wants to send to the table (figure 44a). Once transmitted, the image is added to the list in the center of the user's personal space (figure 44b) and automatically opened for presentation (figure 44c). The opened image can be dragged onto the main panel to share it with other users. All images that were sent to the table reside in the list of the respective user, where they can be accessed later on.

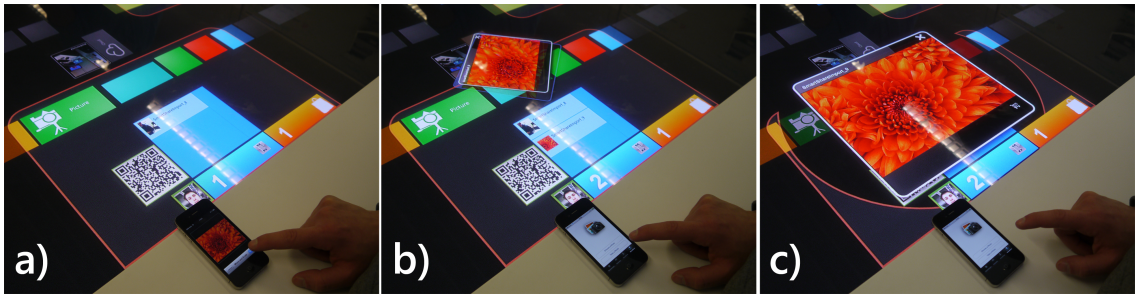

Figure 44: Overview of the steps to get data onto the table

If a user provides login information about his account with a cloud storage service, he may also access all images that are stored in a public folder of this service. To get there, the user has to switch the list with the arrow button (figure 45). The images in that list can again be dragged onto the main panel to share them with other users.

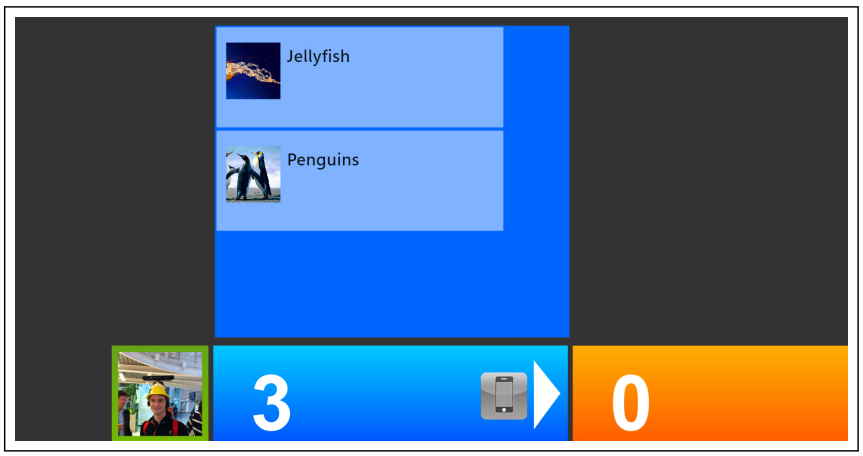

Figure 45: The user can switch between his smartphone and a public folder of a cloud storage service to get images onto the table

The center panel consists of three different parts: A bar at the bottom, which is always visible. In this bar, the `TargetSwitcher` is contained. It consists of two arrow buttons and an icon. Above the bar, the `DropControl` is placed. It consists of two lists which contain all information items that have been sent to the table via smartphone or cloud storage service. Both the `TargetSwitcher` and the `DropControl` are backed by an individual state machine (see figure 46). While the state machine of the `DropControl` is used to control its appearance and disappearance, similar to the state machines used in the login panel, the state machine of the `TargetSwitcher` is used to switch between the smartphone list and the cloud storage service list. The reason why the controls are split onto two state machines, is that the `TargetSwitcher` is also used in another panel (see below) and has therefore been encapsulated in a separate control.

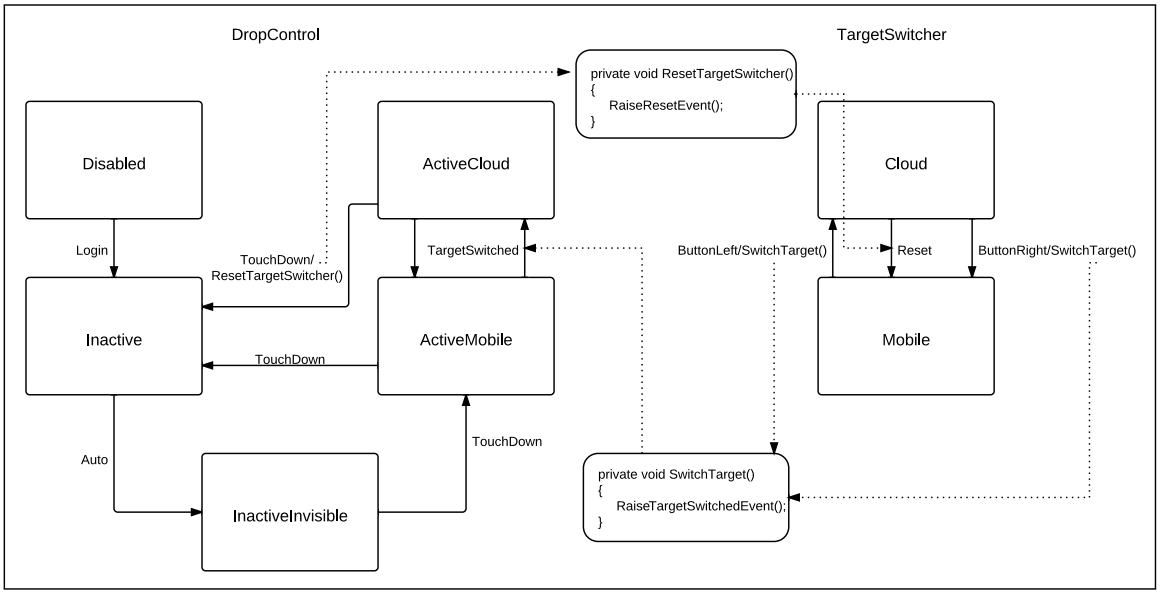

Figure 46: The state machines of the `DropControl` and the `TargetSwitcher`

The state machine of the `DropControl` initially resides in the *Disabled* state. In this state, only the bottom bar of the control is visible, but it is grayed out and does not receive user input. As soon as a user logs in into the personal territory, the state machine transitions to the *Inactive* state and from this automatically to the *InactiveInvisible* state. The bottom bar of the control changes its color to blue, to indicate that it now receives input events. When the user touches this bottom bar, the control transitions to the *ActiveMobile* state and a smooth opacity animation fades in the other elements of the control. In this state, the list of information items, that have been sent via the smartphone, is shown. If the user wants to switch to the list of the cloud storage service, he has to press the arrow button of the `TargetSwitcher`. This control is backed by its own state machine which only transitions forth and back between two states, triggered by presses of the arrow buttons. In the transition action of these transitions, the `TargetSwitched`

event is raised, which serves as trigger for the state machine of the `DropControl` to transition it between the `ActiveMobile` and `ActiveCloud` states. From each of those states the state machine can be transitioned back to the *Inactive* state, by touching the control again. During this transition the reverse opacity animation is applied which eventually fades the control out. If the state machine resided in the *ActiveCloud* state before transitioning to the *Inactive* state, the `ResetTargetSwitcher()` method is called in the transition action. This method raises the `Reset` event of the `TargetSwitcher` to set it back to the *Mobile* state. This mechanism ensures that the `TargetSwitcher` control always starts in the `Mobile` state, when the `DropControl` is activated.

Again, the developers of SmartShare established a communication between two state machines. This became necessary as they split the two parts of the panel apart. To keep them in sync, changes of one state machine have to be transmitted to the other and vice versa. Listing 44 shows the implementation of the state machine of the `DropControl`, which is again very straightforward, as only simple elements have been used. The state machine of the `TargetSwitcher` is omitted, as it contains only two states and is trivially simple.

```
var touchDown = Observable.FromEventPattern(grid1,"TouchDown");

var switched = Observable.FromEventPattern(switcher, "TargetSwitched");

var login = Observable.FromEventPattern(DataContext, "Login");

fsm.AddTransition(login)
   .From(States.Disabled).To(States.Inactive);

fsm.AddAutomaticTransition()
   .From(States.Inactive).To(States.InactivInvisible);

fsm.AddTransition(touchDown)
   .From(States.InactivInvisible).To(States.ActiveMobile);

fsm.AddTransition(touchDown)
   .From(States.ActiveMobile).To(States.Inactive);

fsm.AddTransition(touchDown)
   .From(States.ActiveCloud).To(States.Inactive)
   .Do(e => ResetTargetSwitcher());

fsm.AddTransition(switched)
   .From(States.ActiveMobile).To(States.ActiveCloud);

fsm.AddTransition(switched)
   .From(States.ActiveCloud).To(States.ActiveMobile);
```

Listing 44: State machine implementation of the `DropControl`

9.3.3 Getting Data from the Table

A user can get data from the table into his smartphone or a cloud storage service using the panel on the right side. To get information items into the list of this panel, a special button on the information item has to be pressed (figure 47a). The items can then either be sent to the smartphone of the user or to the cloud storage service (figures 47b and 47c). To switch between those two targets, again the `TargetSwichter` control is used. By touching a checkbox next to the information item, the element is selected and can be sent to the current target via the send button at the bottom.

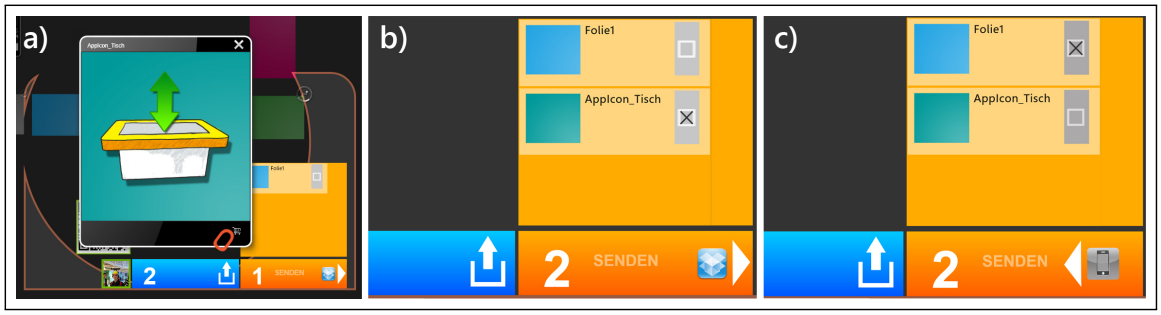

Figure 47: Overview of the steps to get data from the table into the smartphone or cloud storage service

The panel also consists of three parts: The bar at the bottom remains always visible. It contains the `TargetSwitcher`. Above the bar, the `SendControl` is placed, which consists of a list containing the information items that the user selected. Whereas the list of the `DropControl` was exchanged completely, when the user switched between the *Mobile* and *Cloud* states, the list of the `SendControl` always remains the same. Yet, to indicate that the user has switched the target, the checkboxes of the information items are exchanged with a smooth animation. This also ensures that the state of the checkbox (checked, not checked) remains valid when the user switches between the targets. To animate the switching of the checkboxes, each information item contains an individual instance of a state machine. It was not possible to drive the animations with the state machine of the `SendControl`, because currently only 1:1 mappings can be established between the VSM and the RSM. As the animations of the checkboxes reside in a different place than the animations of the `SendControl`, a 1:N mapping would have been needed. The developers therefore put a state machine in every information item and established a communication between these state machines and the `TargetSwitcher`. Figure 48 on the next page shows the resulting models. The implementations of the state machines are omitted because they are very similar to those that have been presented above.

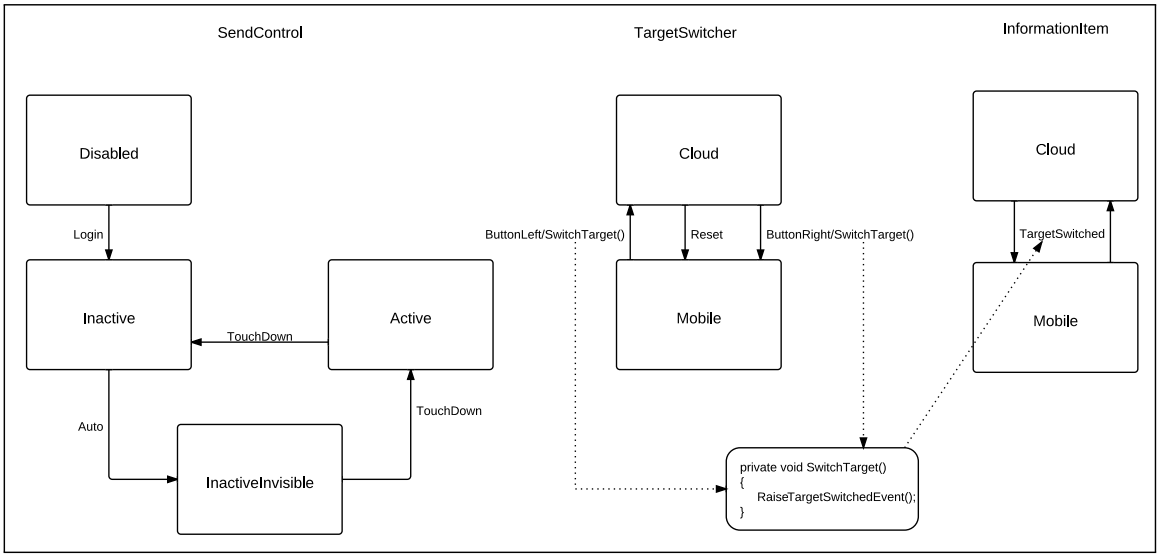

Figure 48: State machines of the `SendControl`, the `TargetSwitcher` and the `InformationItem`

9.3.4 Summary

The SmartShare prototype showed that the Reactive State Machine framework can be handed off without further ado to other developers. The developers of SmartShare were only introduced into the framework for about 15 minutes. They created six different state machines, which were not fairly complex, but helped them to realize the desired behaviors in a concise and straightforward way. The fact that they managed to establish an inter state machine communication, although it has not been shown to them, indicates that the concepts of the framework are easy to grasp and that even novice developers can make extensions to the default behavior.

While the state machines of the Facet-Streams system were mainly driven by multi-touch input, the state machines of SmartShare relied on all sorts of triggers, ranging from simple touch events to login events triggered by a QR code scan from a smartphone, to custom switch events of the `TargetSwitcher`. This broad range of different triggers clearly shows the flexibility that observable collections bring into the Reactive State Machine framework.

9.4 Informal Evaluation

The three use cases that were presented previously show that the Reactive State Machine framework facilitates the development of state-based behavior in post-WIMP user interfaces. As all features of the framework have been used and tested in these two applications, I want to provide a short and informal evaluation of the framework in the following. Myers et al. suggested to evaluate user interface tools by determining their *threshold* and *ceiling*: "The 'threshold' is how difficult it is to learn how to use the system, and the 'ceiling' is how much can be done using the system" [Myers et al., 2000]. Ideally, the threshold of a tool is low and its ceiling is high, which means that both novice users can learn the tool quickly and expert users can realize complex behaviors with it. It is however difficult to get hard numbers for these two characteristics, especially when only few use cases are under consideration. The following results are therefore estimates that are based on an extensive code review of the three use cases and on unstructured interviews with the developers of SmartShare.

Given the fact that the students which implemented SmartShare only had a short introduction of about 15 minutes into the framework, it can be stated that the threshold for novice users is fairly low. They used nearly all features of the framework with only few problems (see below). On the other hand, the Facet-Streams system shows that even complex post-WIMP interaction scenarios can be implemented straightforwardly with the framework and that the resulting code is still maintainable. I therefore state that the ceiling of the framework is higher than that of other similar frameworks. While these are the overall impressions, I want to further dissect the single parts of the framework in the following and point out what each feature contributes to the threshold and ceiling.

Table 4 on the following page summarizes the findings of the informal evaluation. In the first column, all important features of the framework are listed. The second column shows if the respective feature is straightforward to define in code or not. It turns out that all features are straightforward to define, yet some of them require additional care which is discussed more detailly below. The third column shows the threshold of the features. The threshold of a feature is not only dependent on the way it is defined but also on the complexity of the underlying concepts. While especially the classic features such as states and transitions have a low threshold, post-WIMP features such as animations or the multi-point notation are based on more complex concepts and therefore have a higher threshold. The fourth column describes the ceiling of each feature. Here, the classic features such as states and transitions do not contribute much to the ceiling of the framework, while the more advanced features are considered to raise it. In the following, I want to further dissect the features of the RSM and discuss how the assessments of table 4 on the next page are justified.

Feature	Definition	Threshold	Ceiling
States	straightforward	low	medium
Transitions	straightforward	low	medium
Guards	straightforward (care needed)	low to medium	high
Transition Actions	straightforward	low	high
Triggers	straightforward to complicated	low to medium	high
Animations	straightforward (care needed)	low to medium	high
Connecting VSM and RSM	straightforward	low	medium
Multi-Point Notation	straightforward	medium	high
Broadcast communication	straightforward	medium	high

Table 4: Summary table of the informal evaluation of the Reactive State Machine framework

Declarative Specification of States and Transitions The declarative API of the Reactive State Machine framework allows a direct translation of all graphical elements of a state machine model into code. The Reactive State Machine framework uses exactly one expression for one graphical element, while other frameworks often require the developer to use multiple expressions or even create new classes. The advantage of this approach is that the implementation scales linearly with the graphical model. As the direct translation of the graphical model requires the developer to write only structural code, the threshold to implement a regular state machine is very low, compared to low-level techniques which require the developer to mix structural and behavioral code. Most low-level techniques also require significant changes when the model of the state machine is changed. With a declarative specification, only the parts that have changed in the model have to be changed in the implementation and the overall complexity of the implementation remains constant, even when the model gets more complex.

Fluent API All developers immediately understood and appreciated the syntax and semantics of the fluent API. As each method call of the API directly expresses the underlying concept, a direct connection can be drawn from the graphical model to the implementation. This further supports the argument that the framework has a low threshold, as novice users can quickly transform their state machine models into code with the help of this API. It turned out, that none of the developers used the method overloads of the default declarative API, which suggests that it can be removed safely from the framework.

Guards While the guards of transitions can be implemented straightforwardly in the RSM, their specification in the model requires additional care. It is very important that guards are specified unambiguously on transitions that are triggered by the same trigger. If this is not considered, non-determinism may be introduced into the state machine, as more than one transition may be valid for the same trigger. Describing such guards unambiguously is not always easy as sometimes multiple levels of boolean logic have to be considered.

Triggers The use of observable collections as triggers for transitions is one of the main strengths of the Reactive State Machine framework. Although most developers probably do not know the concepts on which the Reactive Extensions are based, I suppose that every developer is able to wrap an input event into an observable collection to use it as a trigger. Once the developer is familiar with observable collections, the possible ways to specify triggers are endless, as arbitrary signaling mechanisms, ranging from regular events to all sorts of asynchronous computation, can be wrapped into an observable collection. Another argument in favor of Rx triggers is that they enable the use of the event metadata inside the transition action and the guard of a transition, which is not equally possible in other frameworks. Thus, I state that Rx triggers clearly raise the ceiling of the framework, as they facilitate several scenarios that would only be complicated to realize in other frameworks.

In addition to this Rx triggers also contribute to the low threshold of the framework. While other frameworks require that triggers are fed into the state machine at runtime, the Reactive State Machine automatically subscribes itself to the triggers, and processes the events that are happening. Thus, no involvement of the developer is required at runtime.

Animated Transitions In both projects, animations were an important part of the controls' interaction design. The feedback of the developers regarding the definition of animated transitions with the Visual State Manager and Expression Blend was uniformly positive. The same holds true for the mechanism that connects the VSM parts with the state machine in the code behind. This subjective impression of the developers is largely reflected in the implementation, which looks very clean. Yet, defining animations in XAML is sometimes tricky, despite the use of the Expression Blend tool. Especially novice users do not know that sometimes the XAML statements have to follow a specific order and that discrete properties, such as boolean properties, can also be set in an animation. Because they had problems with the correct specifications of their animations, the developers of SmartShare often put user interface code inside the transition actions of the state machine, instead of inside the animation. This tended to bloat the definition of the state machine. They also had to introduce an extra state into the state machine, as they did not know how to set the `Visibility` of a UI element at the end of an animation. For future users of the framework it is therefore important that they first understand the concepts of WPF animations and the Visual State Manager before they use animated transitions in their state machines.

Another issue with animations was discovered by the developers of SmartShare. They wanted to animate the checkboxes of the item list all at once, whenever the user switched between smartphone and cloud storage service. Currently only 1:1 mappings can be established between a state machine and a `VisualStateGroup` of the VSM. This meant, that every pair of checkboxes had to be driven by a separate state machine, instead of a single state machine that controlled all pairs. Although all state machines now listen to the same event, the animations of the checkboxes do not start at the same time, because the event is delivered to each state machine one after the other.

The resulting animations were therefore not synchronous and behaved differently every time. To resolve this, it should be possible to associated more than one `VisualStateGroup` with each state machine. The state machine could then start all animations of the VSM at the same time.

Multi-Point Notation So far, the multi-point notation was only used in the Facet-Streams system. There, the direct translation of the operators of the notation into code proved very helpful to realize the complex interactive behaviors and I therefore conclude that the notation raises the ceiling of the framework.

Yet, it turned out that the current implementation can sometimes cause problems, if the developer is not careful. As points are only added to and removed from the input point collection when a transition is made, sometimes input points are ignored when no transition is specified for the respective trigger. This is for example the case when a trigger can cause multiple transitions that only differ in their guards. If the specification of the guards is not precise, it can happen that no transition is triggered at all. Input points are also ignored when a state does not react at all to certain input points. This was initially the case with the *FacetWheelFadingIn* state of Facet-Streams. Although this state is only active for a very short amount of time, it can happen that a user places a second or third finger onto the control during the fade in animation. If no transition is specified for this case, the additional input points are never added to the collection. This results in incorrect behaviors as the operators of the notation rely on a correctly maintained input point collection.

To overcome this issue, the developer currently has to make sure that each input point causes a transition in all circumstances. This may mean that guards have to be specified unambiguously and that 'dummy' transitions have to be added, i.e. internal transitions that do not cause any action but only ensure that the input point is added or removed. While this requires more effort on the developer's side, it also forces him to think more precisely about the transitions, eventually resulting in a better model.

Broadcast Communication The broadcast communication mechanism that was mentioned in the Statecharts notation can easily be realized with the RSM framework, in that one state machine produces an event during a transition which is used as trigger by another state machine. Although this mechanism was not explained to them previously, the developers of SmartShare discovered it by themselves and made extensive use of it. Several different scenarios can be realized with this mechanism, such as the tight coupling or synchronization of state machines, therefore I state that it contributes to the high ceiling of the RSM framework.

Part VI

Conclusion

> I may not have gone where I intended to go, but I think I have ended up where I needed to be
>
> Douglas Adams, *The Long Dark Tea-Time of the Soul*

In this last part, the main contributions of the thesis are summarized shortly and an outlook on potential future work is given.

10 Conclusion

In the first part of this thesis, it has been shown that post-WIMP systems have various different characteristics that stem from multiple fields of research and issue huge challenges to designers and developers. Formal methods, such as finite-state machines, can prove very beneficial for certain of these challenges as they facilitate the description of interactive behaviors and require the designer to be precise. Yet, although finite-state machines have often been suggested as a valuable formalism to model certain post-WIMP behaviors, their implementation is not appropriately supported in present-day programming languages and user interface toolkits. This eventually leaves a gap that the framework presented in this thesis tries to fill. While the Reactive State Machine framework certainly makes the design and development of post-WIMP interaction easier in many ways, it must not be forgotten that it is but one tool in the toolbox of a post-WIMP interaction designer. Many more additional concepts and tools are needed to address all challenges of these systems. As the characteristics of post-WIMP systems will even expand further in the upcoming years, many more challenges will come up, requiring ever more diverse concepts and tools. Post-WIMP systems are actually only a transitional step towards an even greater vision. According to Weiser, computers in the age of *ubiquitous computing* will eventually be "invisible in fact as well as in metaphor" [Weiser, 1991]. As of today, it is not entirely foreseeable how the interactions with these devices is taking place someday. I assume that the changes are gradual and evolutionary rather than sudden and revolutionary. Thus, many of our current concepts and tools will also be there in five or ten years time, especially when they are based on such fundamental formalisms like finite-state machines.

10.1 Contributions

In the following, the main contributions of this thesis are summarized in terms of both concepts and development.

10.1.1 Conceptual Contributions

Concept for animations Animations or animated transitions have not been considered in any state machine notation so far. In section 1.2.4 on page 18, a concept was suggested to model these with only few changes to the default notation. The concept is based on the materialization of the transitioning phase into a dedicated state. As a consequence, the designer of a state machine has greater expressivity to model the behavior during the transitioning phase. Also, a new type of transition (automatic transitions) was introduced into the notation to deal with the completion of animations.

Concept for multi-point notation Post-WIMP systems often feature input devices that contain multiple input points, such as multi-touch tabletops. Yet, current state machine notations have no means of differentiating between individual points of these devices, which may become important to realize certain interactive behaviors. In section 3.4 on page 48, a notation was therefore suggested which allows the differentiation of multiple input points. The notation is based on a collection metaphor and features several operators that can be used in the guard of a transition to test if the current input point corresponds to a certain position in this collection. It has been shown that the application of this notation enables advanced interactive behaviors that could not be expressed previously with the default state machine notation.

10.1.2 Development Contributions

Full event support via Rx Probably the most important contribution in terms of development is the full event support that the RSM offers due to its usage of Rx triggers. These triggers can be specified declaratively and the state machine automatically registers itself with them. Thus, it is not required to feed events at runtime into the state machine. Also, all event metadata can be leveraged inside the transition action and the guard of a transition, which is very important for the realization of many interactive behaviors.

Animation support via VSM The RSM supports the definition of animated transitions based on the aforementioned animation concept. The Visual State Manager and tools such as Expression Blend thereby greatly facilitate the definition of animations in XAML code. This design contributes to an improved designer-developer workflow, as both the visual part and the logical part can be defined separately with optimal tool support.

Support for the multi-point notation The aforementioned multi-point notation is supported entirely in the RSM. Thereby the operators of the notation are transformed literally into method calls of the RSM, to ensure that the transformation from model to code remains straightforward. While input point trackers for two particular input devices are included in the library, the implementation also allows the integration of additional input point trackers.

10.2 Future Work

Although the Reactive State Machine contains a lot of features and thereby supports many post-WIMP scenarios, there is still room for improvements. I want to highlight two major aspects in the following which I deem very important.

10.2.1 Support for Statecharts Elements

In terms of features, the RSM does not have to fear comparison with other state machine frameworks, although it currently does not support two concepts of the Statecharts notation (hierarchy and orthogonality). I briefly pointed out that support for those two features is not highly required to model post-WIMP interaction, yet, it would further enhance the expressivity and overall value of the RSM. One of the main reasons behind the omission of these concepts were constraints resulting from generic type parameters. The application of generic type parameters in the RSM was a deliberate design decision, as they greatly simplify the configuration of regular state machines. If the additional Statecharts concepts had to be integrated, it would certainly be necessary to rethink this design decision. While the integration of these concepts would probably require only minor changes to the current design of the configuration API, there would certainly be major changes to the inner workings and semantics of the state machine, as the Statecharts notation has clearly defined rules for hierarchical and orthogonal states which ensure the correct and consistent behavior of the state machine in all situations. It is for example specified how substates are entered and exited correctly, how orthogonal states react to the same trigger and how history transitions work. These rules would have to be implemented and tested for correctness.

10.2.2 Support for Graphical FSM Models

Every developer who uses finite-state machines to model interactive behavior typically begins with the drawing of a graphical model of the state machine, as the graphical representation by far outweighs every textual representation in terms of expressivity and comprehensibility. Yet, to implement the graphical model he has to somehow convert it into a textual representation in a programming language. While this is more complex with low-level implementation techniques, it can be relatively straightforward with declarative state machine frameworks such as the RSM. As the RSM requires exactly one expression for every element of the graphical model, the transformation into code could easily be automated. Such automatic transformations of graphical models into code are one of the tasks that are performed in model driven engineering (MDE). There, UML (Unified Modeling Language) diagrams are used to express certain aspects of a system (its structure, behavior and interaction) in a platform independent way. With special tools, these diagrams are then converted to platform specific code representations. Depending on the degree of detail that has been modeled in the diagram, these code representations are of different fidelity. Usually it is more common to be abstract in the graphical, platform independent model and more detailed in the textual, platform specific model (i.e. the code). The results of such transformations are therefore mere code skeletons that have to be elaborated further by the developer. It is obvious that with such differences in level of detail, no roundtrip can be achieved. Roundtrips are however very supportive, as the different representations may complement each other in terms of expressivity. Also, developers rather make changes in code and designers rather make changes in graphical

models. If both were synchronized, the designer-developer workflow could be improved significantly. Fortunately, with the RSM, the graphical and textual representations are at almost the same level of detail. It should therefore be possible to achieve roundtrip scenarios with a tool that keeps both representations in sync. In my opinion, the only elements of state machines that need special treatment are actions and guards. While the structural and relational elements of state machines (i.e. the states and transitions) are straightforward to transform forth and back, actions and guards potentially contain a fair amount of instructions and references to other constructs of the application, such as user interface elements. It is not convenient to express these details in the graphical notation. Instead only the method name or the intention of the action/guard are usually written on the arrow of a transition. Therefore they can not participate fully in the transformation process. If this issue can be resolved adequately by a tool, roundtrip scenarios should be easily possible with the RSM.

Apart from these transformations that work on top of the model and the API, it is also important that the state machine in either representation is integrated neatly into the overall development environment. If a developer has to manually trigger the transformation process and copy/paste the resulting code into his source files, it would cause too much extra integration work and he would probably not do it. A tool to support model-code-model roundtrips therefore is ideally part of the default development environment and offers low-threshold access to all important functionalities. The developer should be able to seamlessly change between graphical and textual representation and changes to one should be synchronized immediately with the other. Such integration into the development environment requires more thought and effort than the simple transformation of the state machine elements into API calls: To which file is the generated code added? In which part of the file does the code reside? How can it be located for the reverse transformation? These and other questions have to be answered by the tool to ensure appropriate integration. It is therefore certainly wise to have a look at how current user interface editors work, as they also have to keep a graphical and textual representation in sync and must integrate neatly in the overall development environment.

The default development environment of .NET and WPF developers is Visual Studio. It has been shown previously[30] that state machines can be integrated with reasonable effort into Visual Studio and first attempts of the author in this direction have proven promising. In general, the integration of additional functionality into Visual Studio is possible via a powerful plugin mechanism. In its *Ultimate Edition*, Visual Studio also supports the creation of graphical models and the Visual Studio Visualization & Modeling SDK[31] offers support for Domain Specific Languages in terms of both graphical modeling and transformation into source code. Thus, the technical prerequisites for an integration of the Reactive State Machine framework into Visual Studio are given.

[30] http://mnsdc.de/programme/vsix/statepatternmodeler.htm
[31] http://archive.msdn.microsoft.com/vsvmsdk

References

Abelson, Harold, Sussman, Gerald J., and Sussman, Julie. *Structure and interpretation of computer programs*. MIT Press, 1996. ISBN 978-0684831305.

Accot, Johnny, Chatty, Stéphane, and Palanque, Philippe. A Formal Description of Low Level Interaction and its Application to Multimodal Interactive Systems. In François Bodart and Jean Vanderdonckt, editors, *Proceedings of DSV-IS'96*, pages 92–104. Springer, 1996. ISBN 3-211-82900-8.

Ackroyd, Michael. Object-Oriented Design of a Finite State Machine. *Journal of Object-Oriented Programming*, 8(3):50–59, 1995.

Agarawala, Anand and Balakrishnan, Ravin. Keepin' it real: pushing the desktop metaphor with physics, piles and the pen. In Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson, editors, *Proceedings of CHI'06*, pages 1283–1292. ACM, 2006. ISBN 1-59593-372-7.

Anderson, Michael L. Embodied cognition: a field guide. *Artificial Intelligence*, 149(1):91–130, 2003. ISSN 0004-3702. `doi:10.1016/S0004-3702(03)00054-7`.

Appert, Caroline and Beaudouin-Lafon, Michel. SwingStates: adding state machines to Java and the Swing toolkit. *Software: Practice and Experience*, 38(11):1149–1182, 2008. ISSN 1097-024X. `doi:10.1002/spe.867`.

Apple Computer, Inc. *Macintosh human interface guidelines*. Addison-Wesley Publishing Company, 1992. ISBN 978-0201622164.

Awodey, Steve. *Category theory*. Clarendon Press, 2006. ISBN 978-0198568612.

Baglioni, Mathias, Malacria, Sylvain, Lecolinet, Eric, and Guiard, Yves. Flick-and-brake: finger control over inertial/sustained scroll motion. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *Proceedings of CHI'11 Extended Abstracts*, pages 2281–2286. ACM, 2011. ISBN 978-1450302685.

Ballendat, Till, Marquardt, Nicolai, and Greenberg, Saul. Proxemic interaction: designing for a proximity and orientation-aware environment. In Krüger et al. [2010], pages 121–130.

Bier, Eric A., Stone, Maureen C., Pier, Ken, Buxton, William, and DeRose, Tony D. Toolglass and magic lenses: the see-through interface. In Whitton [1993], pages 73–80.

Blake, Joshua. WIMP is to GUI as OCGM (Occam) is to NUI [webpage]. 2009. URL: `http://nui.joshland.org/2009/12/wimp-is-to-gui-as-ocgm-occam-is-to-nui.html` [Last checked 2012-04-02].

Bolt, Richard A. "Put-that-there": Voice and gesture at the graphics interface. In Harvey Z. Kriloff James J. Thomas, Robert A. Ellis, editor, *Proceedings of SIGGRAPH '80*, pages 262–270. ACM, 1980. ISBN 0-89791-021-4.

Bottoni, Paolo, Rosson, Mary Beth, and Minas, Mark, editors. *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*. IEEE Computer Society, 2008. ISBN 978-1-4244-2528-0.

Bragdon, Andrew, Uguray, Arman, Wigdor, Daniel, Anagnostopoulos, Stylianos, Zeleznik, Robert, and Feman, Rutledge. Gesture play: motivating online gesture learning with fun, positive reinforcement and physical metaphors. In Krüger et al. [2010], pages 39–48.

Brooks, Frederick P., Jr. No Silver Bullet – Essence and Accidents of Software Engineering. *Computer*, 20:10–19, 1987. ISSN 0018-9162. doi:10.1109/MC.1987.1663532.

Buxton, William. There's More to Interaction than Meets the Eye: Some Issues in Manual Input. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 319–337. L. Erlbaum Associates Inc., 1986. ISBN 0898597811.

Card, Stuart K., Moran, Thomas P., and Newell, Allen. *The psychology of human-computer interaction*. L. Erlbaum Associates Inc., 1983. ISBN 978-0898598599.

Chang, Bay-Wei and Ungar, David. Animation: from cartoons to the user interface. In Scott Hudson, Randy Pausch, Brad Vander Zanden, and James Foley, editors, *Proceedings of UIST'93*, pages 45–55. ACM, 1993. ISBN 0-89791-628-X.

Chatty, Stéphane. Defining the Dynamic Behaviour of Animated Interfaces. In James A. Larson and Claus Unger, editors, *Proceedings of EHCI'92*, pages 95–111. North-Holland, 1992. ISBN 0-444-89904-9.

Chui, Michael and Dillon, Andrew. Who's zooming whom? Attunement to animation in the interface. *Journal of the American Society for Information Science*, 48:1067–1072, 1997. ISSN 0002-8231. doi:10.1002/(SICI)1097-4571(199711)48:11<1067::AID-ASI8>3.3.CO;2-2.

Cloud Programmability Team. Rx Design Guidelines. Technical report, Microsoft Corporation, 2010. URL: http://go.microsoft.com/fwlink/?LinkID=205219.

Collins, Christopher, Penn, Gerald, and Carpendale, Sheelagh. Bubble Sets: Revealing Set Relations with Isocontours over Existing Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1009–1016, 2009. ISSN 1077-2626. doi:10.1109/TVCG.2009.122.

Collins, Dave. *Designing Object-Oriented User Interfaces*. Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 978-0805353501.

de Haan, Gerwin and Post, Frits H. StateStream: a developer-centric approach towards unifying interaction models and architecture. In Graham et al. [2009], pages 13–22.

Dietz, Paul and Leigh, Darren. DiamondTouch: a multi-user touch technology. In Joe Marks and Elizabeth Mynatt, editors, *Proceedings of UIST'01*, pages 219–226. ACM, 2001. ISBN 1-58113-438-X.

Dix, Alan. *Formal methods for interactive systems*. Academic Press, 1991. ISBN 978-0122183157.

Dix, Alan. Formal Methods in HCI: Moving Towards an Engineering Approach. In *Proceedings of NDISD'93 - HCI: Making Software Usable*. 1993.

Dix, Alan. Formal Methods. In Andrew Monk and Nigel Gilbert, editors, *Perspectives in HCI: Diverse Approaches*, chapter 2, pages 9–43. Academic Press, 1995. ISBN 978-0125045759.

Dix, Alan. Formal Methods in HCI: a Success Story - why it works and how to reproduce it, 2002. Unpublished manuscript. URL: `http://www.alandix.com/academic/papers/formal-2002/`.

Dix, Alan. Upside down ∀s and algorithms – computational formalisms and theory. In John Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Mulitdisciplinary Science*, chapter 14, pages 381–429. Morgan Kaufmann, 2003. ISBN 978-1558608085.

Dix, Alan and Abowd, Gregory. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996. ISSN 0268-6961.

Dix, Alan, Ghazali, Masitah, and Ramduny-Ellis, Devina. Modelling Devices for Natural Interaction. *Electronic Notes in Theoretical Computer Science*, 208:23–40, 2008. ISSN 1571-0661. `doi:10.1016/j.entcs.2008.03.105`.

Dohse, K. C., Dohse, Thomas, Still, Jeremiah D., and Parkhurst, Derrick J. Enhancing Multi-user Interaction with Multi-touch Tabletop Displays Using Hand Tracking. In C. Dini, editor, *Proceedings of ACHI'08*, pages 297–302. IEEE Computer Society, 2008. ISBN 978-0-7695-3086-4.

Dourish, Paul. *Where the Action Is: The Foundations of Embodied Interaction*. The MIT Press, 2001. ISBN 978-0262541787.

Elliott, Conal and Hudak, Paul. Functional Reactive Animation. In Arthur Michael Berman, editor, *Proceedings of ICFP'97*. ACM, 1997. ISBN 0-89791-918-1.

Fitzmaurice, George W., Ishii, Hiroshi, and Buxton, William A. S. Bricks: laying the foundations for graspable user interfaces. In Irvin R. Katz, Robert L. Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen, editors, *Proceedings of CHI'95*, pages 442–449. ACM/Addison-Wesley, 1995. ISBN 0-201-84705-1.

Forlines, Clifton, Wigdor, Daniel, Shen, Chia, and Balakrishnan, Ravin. Direct-touch vs. mouse input for tabletop displays. In Rosson and Gilmore [2007], pages 647–656.

Frisch, Mathias, Heydekorn, Jens, and Dachselt, Raimund. Investigating multi-touch and pen gestures for diagram editing on interactive surfaces. In Gerald Morrison, Sriram Subramanian, M. Sheelagh T. Carpendale, Michael Haller, and Stacey D. Scott, editors, *Proceedings of ITS'09*, pages 149–156. ACM, 2009. ISBN 978-1-60558-733-2.

Frisch, Mathias, Langner, Ricardo, and Dachselt, Raimund. Neat: a set of flexible tools and gestures for layout tasks on interactive displays. In Rekimoto et al. [2011], pages 1–10.

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. ISBN 978-0201633610.

Gentner, Don and Nielsen, Jakob. The Anti-Mac interface. *Communications of the ACM*, 39:70–82, 1996. ISSN 0001-0782. `doi:10.1145/232014.232032`.

George, Ron. OCGM (pronounced Occam['s Razor]) is the replacement for WIMP [webpage]. 2009. URL: `http://blog.rongeorge.com/design/ocgm-pronounced-occams-razor-is-the-replacement-for-wimp/` [Last checked 2012-04-02].

George, Ron and Blake, Joshua. Objects, Containers, Gestures, and Manipulations: Universal Foundational Metaphors of Natural User Interfaces. In *Natural User Interfaces: The Prospect and Challenge of Touch and Gestural Computing – A CHI'10 Workshop*. 2010.

Geyer, Florian, Pfeil, Ulrike, Budzinski, Jochen, Höchtl, Anita, and Reiterer, Harald. AffinityTable – A Hybrid Surface for Supporting Affinity Diagramming. In Pedro F. Campos, T. C. Nicholas Graham, Joaquim A. Jorge, Nuno Jardim Nunes, Philippe A. Palanque, and Marco Winckler, editors, *Proceedings of INTERACT'11*, volume 6946 of *Lecture Notes in Computer Science*, pages 477–484. Springer, 2011. ISBN 978-3-642-23773-7.

Gonzalez, Cleotilde. Does animation in user interfaces improve decision making? In Bonnie A. Nardi, Gerrit C. van der Veer, and Michael J. Tauber, editors, *Proceedings of CHI'96*, pages 27–34. ACM, 1996. ISBN 0-89791-777-4.

Graham, T. C. Nicholas, Calvary, Gaëlle, and Gray, Philip D., editors. *Proceedings of the 1st ACM SIGCHI symposium on Engineering Interactive Computing System , EICS 2009, Pittsburgh, PA, USA, July 15-17, 2009*. ACM, 2009. ISBN 978-1-60558-600-7.

Green, Mark and Jacob, Robert J. K. SIGGRAPH '90 Workshop report: software architectures and metaphors for non-WIMP user interfaces. *ACM SIGGRAPH Computer Graphics*, 25(3):229–235, 1991. doi:126640.126677.

Greenberg, Saul and Marwood, David. Real time groupware as a distributed system: concurrency control and its effect on the interface. In Jim Herbsleb and Gary Olson, editors, *Proceedings of CSCW'94*, pages 207–217. ACM, 1994. ISBN 0-89791-689-1.

Harel, David. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. ISSN 0167-6423. doi:10.1016/0167-6423(87)90035-9.

Henze, Niels, Löcken, Andreas, Boll, Susanne, Hesselmann, Tobias, and Pielot, Martin. Free-hand gestures for music playback: deriving gestures with a user-centred process. In Marios C. Angelides, Lambros Lambrinos, Michael Rohs, and Enrico Rukzio, editors, *Proceedings of MUM'10*, pages 16:1–16:10. ACM, 2010. ISBN 978-1450304245.

Herbsleb, James D. and Olson, Gary M., editors. *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004*. ACM, 2004. ISBN 1-58113-810-5.

Hopcroft, John E., Motwani, Rajeev, and Ullman, Jeffrey D. *Introduction to automata theory, languages, and computation*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2nd edition, 2001. ISBN 978-0201441246.

Hudson, Scott E. and Newell, Gary L. Probabilistic state machines: dialog management for inputs with uncertainty. In Jock Mackinlay and Mark Green, editors, *Proceedings of UIST'92*, pages 199–208. ACM, 1992. ISBN 0-89791-549-6.

Hutchins, Edwin L., Hollan, James D., and Norman, Donald A. Direct manipulation interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985. ISSN 0737-0024. doi:10.1207/s15327051hci0104_2.

Ishii, Hiroshi and Ullmer, Brygg. Tangible bits: towards seamless interfaces between people, bits and atoms. In Steven Pemberton, editor, *Proceedings of CHI'97*, pages 234–241. ACM/Addison-Wesley, 1997. ISBN 0-201-32229-3.

ITL Education Solutions Limited. *Introduction to Database Systems*. Pearson Education, 2010. ISBN 978-8131731925.

Jacob, Robert J.K., Girouard, Audrey, Hirshfield, Leanne M., Horn, Michael S., Shaer, Orit, Solovey, Erin Treacy, and Zigelbaum, Jamie. Reality-based interaction: a framework for post-WIMP interfaces. In Mary Czerwinski, Arnold M. Lund, and Desney S. Tan, editors, *Proceedings of CHI'08*, pages 201–210. ACM, 2008. ISBN 978-1-60558-011-1.

Jetter, Hans-Christian, Gerken, Jens, and Reiterer, Harald. Natural User Interfaces: Why We Need Better Model-Worlds, Not Better Gestures. In *Natural User Interfaces: The Prospect and Challenge of Touch and Gestural Computing – A CHI'10 Workshop*. 2010.

Jetter, Hans-Christian, Gerken, Jens, Zöllner, Michael, Reiterer, Harald, and Milic-Frayling, Natasa. Materializing the Query with Facet-Streams - A Hybrid Surface for Collaborative Search on Tabletops. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *Proceedings of CHI'11*. ACM, 2011. ISBN 978-1-4503-0228-9.

Jetter, Hans-Christian, Zöllner, Michael, Gerken, Jens, and Reiterer, Harald. Design and Implementation of Post-WIMP Distributed User Interfaces with ZOIL. *to appear in International Journal of Human-Computer Interaction IJHCI (Special Issue on Distributed User Interfaces)*, 2012.

Kaptelinin, Victor and Czerwinski, Mary, editors. *Beyond the desktop metaphor: designing integrated digital work environments*. MIT Press, 2007. ISBN 978-0262113045.

Kirk, David, Sellen, Abigail, Taylor, Stuart, Villar, Nicolas, and Izadi, Shahram. Putting the physical into the digital: issues in designing hybrid interactive surfaces. In *Proceedings of BCS HCI'09*, pages 35–44. ACM, 2009.

Klemmer, Scott R., Hartmann, Björn, and Takayama, Leila. How bodies matter: five themes for interaction design. In John M. Carroll, Susanne Bødker, and Julie Coughlin, editors, *Proceedings of DIS'06*, pages 140–149. ACM, 2006. ISBN 1-59593-367-0.

Klinkhammer, Daniel, Nitsche, Markus, Specht, Marcus, and Reiterer, Harald. Adaptive Personal Territories for Co-located Tabletop Interaction in a Museum Setting. In Rekimoto et al. [2011], pages 107–110.

Krüger, Antonio, Schöning, Johannes, Wigdor, Daniel, and Haller, Michael, editors. *ACM International Conference on Interactive Tabletops and Surfaces, ITS 2010, Saarbrücken, Germany, November 7-10, 2010*. ACM, 2010. ISBN 978-1-4503-0399-6.

Lawson, Jean-Yves Lionel, Al-Akkad, Ahmad-Amr, Vanderdonckt, Jean, and Macq, Benoit. An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. In Graham et al. [2009], pages 245–254.

Lee, Sang-Su and Lee, Woohun. Exploring effectiveness of physical metaphor in interaction design. In Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg, editors, *Proceedings of CHI'09 Extended Abstracts*, pages 4363–4368. ACM, 2009. ISBN 978-1605582474.

Mankoff, Jennifer, Hudson, Scott E., and Abowd, Gregory D. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In Thea Turner and Gerd Szwillus, editors, *Proceedings of CHI'00*, pages 368–375. ACM, 2000. ISBN 1-58113-216-6.

May, Jon and Barnard, Philip J. Cinematography and interface design. In Knut Nordby, Per H. Helmersen, David J. Gilmore, and Svein A. Arnesen, editors, *Proceedings of INTERACT'95*, pages 26–31. Chapman & Hall, 1995. ISBN 0-412-71790-5.

Melchior, Jérémie, Grolaux, Donatien, Vanderdonckt, Jean, and Van Roy, Peter. A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications. In Graham et al. [2009], pages 69–78.

Morris, Meredith Ringel, Ryall, Kathy, Shen, Chia, Forlines, Clifton, and Vernier, Frederic. Beyond "social protocols": multi-user coordination policies for co-located groupware. In Herbsleb and Olson [2004], pages 262–265.

Myers, Brad A. Separating application code from toolkits: eliminating the spaghetti of call-backs. In James R. Rhyne, editor, *Proceedings of UIST'91*, pages 211–220. ACM, 1991. ISBN 0-89791-451-1.

Myers, Brad A., Hudson, Scott E., and Pausch, Randy. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7:3–28, 2000. ISSN 1073-0516. doi:10.1145/344949.344959.

Myers, Brad A., Park, Sun Young, Nakano, Yoko, Mueller, Greg, and Ko, Andrew. How designers design and program interactive behaviors. In Bottoni et al. [2008], pages 177–184.

Negroponte, Nicholas. Beyond the Desktop Metaphor. In Albert Meyer, John Guttag, Ronald Rivest, and Peter Szolovits, editors, *Research Directions in Computer Science – An MIT Perspective*, chapter 9, pages 183–190. MIT Press, 1991. ISBN 978-0262132572.

Nelson, Ted. The Right Way to Think About Software Design. In Brenda Laurel and S. Joy Mountford, editors, *The Art of Human-Computer Interface Design*, pages 235–243. Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN 978-0201517972.

Newman, William M. A system for interactive graphical programming. In *AFIPS 1968 Spring Joint Computing Conference*, pages 47–54. Thomson Book Company, Washington D.C., 1968.

Nielsen, Jakob. Noncommand user interfaces. *Communications of the ACM*, 36:83–99, 1993. ISSN 0001-0782. doi:10.1145/255950.153582.

Nigay, Laurence and Coutaz, Joëlle. A design space for multimodal systems: concurrent processing and data fusion. In Stacey Ashlund, Kevin Mullet, Austin Henderson, Erik Hollnagel, and Ted N. White, editors, *Proceedings of INTERCHI'93*, pages 172–178. ACM, 1993. ISBN 0-89791-574-7.

Norman, Donald A. Natural user interfaces are not natural. *interactions*, 17(3):6–10, 2010. ISSN 1072-5520. doi:10.1145/1744161.1744163.

O'Sullivan, Dan and Igoe, Tom. *Physical computing: sensing and controlling the physical world with computers*. Thomson, 2004. ISBN 978-1592003464.

Palanque, Philippe A., Barboni, Eric, Martinie, Célia, Navarre, David, and Winckler, Marco. A model-based approach for supporting engineering usability evaluation of interaction techniques. In Fabio Paternò, Kris Luyten, and Frank Maurer, editors, *Proceedings of EICS'11*, pages 21–30. ACM, 2011. ISBN 978-1-4503-0670-6.

Park, Sun Young, Myers, Brad, and Ko, Andrew J. Designers' natural descriptions of interactive behaviors. In Bottoni et al. [2008], pages 185–188.

Parnas, David L. On the use of transition diagrams in the design of a user interface for an interactive computer system. In Solomon L. Pollack, Thomas R. Dines, Ward Sangren, Norman R. Nielsen, William G. Gerkin, Alfred E. Corduan, Len Nowak, James L. Mueller, III Joseph Horner, Pasteur S. T. Yuen, Jeffery Stein, and Margaret M. Mueller, editors, *Proceedings of ACM'69*, pages 379–385. ACM, 1969.

Perlin, Ken and Fox, David. Pad: an alternative approach to the computer interface. In Whitton [1993], pages 57–64.

Pfaff, Günther E., editor. *User Interface Management Systems*. Springer-Verlag New York, Inc., 1985. ISBN 038713803X.

Pierce, Benjamin. *Basic category theory for computer scientists*. MIT Press, 1991. ISBN 978-0262660716.

Raskin, Jef. *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-37937-6.

Rädle, Roman. *Squidy: A Zoomable Design Environment for Natural User Interfaces*. AV Akademikerverlag, 2011. ISBN 978-3639385991.

Rekimoto, Jun, Koike, Hideki, Fukuchi, Kentaro, Kitamura, Yoshifumi, and Wigdor, Daniel, editors. *ACM International Conference on Interactive Tabletops and Surfaces, ITS 2011, Kobe, Japan, November 13-16, 2011*. ACM, 2011. ISBN 978-1-4503-0871-7.

Robertson, George G., Mackinlay, Jock D., and Card, Stuart K. Cone Trees: animated 3D visualizations of hierarchical information. In Scott P. Robertson, Gary M. Olson, and Judith S. Olson, editors, *Proceedings of CHI'91*, pages 189–194. ACM, 1991. ISBN 0-89791-383-3.

Rosson, Mary Beth and Gilmore, David J., editors. *Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007*. ACM, 2007. ISBN 978-1595935939.

Schwarz, Julia, Hudson, Scott, Mankoff, Jennifer, and Wilson, Andrew D. A framework for robust and flexible handling of inputs with uncertainty. In Ken Perlin, Mary Czerwinski, and Rob Miller, editors, *Proceedings of UIST'10*, pages 47–56. ACM, 2010. ISBN 978-1-4503-0271-5.

Scott, Stacey D., Sheelagh, M., Carpendale, T., and Inkpen, Kori M. Territoriality in collaborative tabletop workspaces. In Herbsleb and Olson [2004], pages 294–303.

Shaer, Orit and Hornecker, Eva. Tangible User Interfaces: Past, Present, and Future Directions. *Foundations and Trends in Human-Computer Interaction*, 3(1–2):1–137, 2010. ISSN 1551-3955. doi:10.1561/1100000026.

Shaer, Orit and Jacob, Robert J. K. A specification paradigm for the design and implementation of tangible user interfaces. *ACM Transactions on Computer-Human Interaction*, 16:20:1–20:39, 2009. ISSN 1073-0516. doi:10.1145/1614390.1614395.

Shneiderman, Ben. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16:57–69, 1983. ISSN 0018-9162. doi:10.1109/MC.1983.1654471.

Shotton, Jamie, Fitzgibbon, Andrew, Cook, Mat, Sharp, Toby, Finocchio, Mark, Moore, Richard, Kipman, Alex, and Blake, Andrew. Real-Time Human Pose Recognition in Parts from Single Depth Images. In *Proceedings of CVPR'11*, pages 1297–1304. IEEE, 2011. ISBN 978-1457703942.

Smith, David Canfield, Irby, Charles, Kimball, Ralph, Verplank, Bill, and Harslem, Eric. Designing the star user interface. *BYTE*, 7(4):242–282, 1982.

Terrenghi, Lucia, Kirk, David, Sellen, Abigail, and Izadi, Shahram. Affordances for manipulation of physical versus digital media on interactive surfaces. In Rosson and Gilmore [2007], pages 1157–1166.

Thimbleby, Harold W. *Press on – principles of interaction programming*. MIT Press, 2007. ISBN 978-0262201704.

Ullmer, Brygg, Ishii, Hiroshi, and Jacob, Robert J. K. Token+constraint systems for tangible interaction with digital information. *ACM Transactions on Computer-Human Interaction*, 12(1):81–118, 2005. ISSN 1073-0516. doi:10.1145/1057237.1057242.

van Dam, Andries. Post-WIMP user interfaces. *Communications of the ACM*, 40:63–67, 1997. ISSN 0001-0782. doi:10.1145/253671.253708.

van Dam, Andries. Post-WIMP User Interfaces: the Human Connection. In Rae Earnshaw, Richard Guedj, Andries Van Dam, and John Vince, editors, *Frontiers of human-centered computing online communities and virtual environments*, pages 163–178. Springer, 2001. ISBN 978-1852332389.

Wasserman, Anthony I. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 11:699–713, 1985. ISSN 0098-5589. doi:10.1109/TSE.1985.232519.

Watanabe, Nayuko, Washida, Motoi, and Igarashi, Takeo. Bubble clusters: an interface for manipulating spatial aggregation of graphical objects. In Chia Shen, Robert J. K. Jacob, and Ravin Balakrishnan, editors, *Proceedings of UIST'07*, pages 173–182. ACM, 2007. ISBN 978-1-59593-679-0.

Weiser, Mark. The computer for the 21st century. *Scientific American*, 3(3):3–11, 1991.

Wellner, Pierre. Interacting with paper on the digitaldesk. *Communications of the ACM*, 36(7):87–96, 1993. ISSN 0001-0782. doi:10.1145/159544.159630.

Wellner, Pierre, Mackay, Wendy, and Gold, Rich. Back to the real world. *Communications of the ACM*, 36(7):24–26, 1993. ISSN 0001-0782. doi:10.1145/159544.159555.

Whitton, Mary C., editor. *Proceedings of the 20st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1993*. ACM, 1993. ISBN 0-89791-601-8.

Wigdor, Daniel and Wixon, Dennis. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*. Morgan Kaufmann Publishers Inc., 2011. ISBN 978-0123822314.

Wilson, Andrew D., Izadi, Shahram, Hilliges, Otmar, Garcia-Mendoza, Armando, and Kirk, David. Bringing physics to the surface. In Steve B. Cousins and Michel Beaudouin-Lafon, editors, *Proceedings of UIST'08*, pages 67–76. ACM, 2008. ISBN 978-1595939753.

Zöllner, Michael. Developing Facet-Streams - A hybrid surface for collaborative search on tabletops, 2011a. Technical Report of the Master Project. URL: http://hci.uni-konstanz.de/downloads/studentrepos/SS11/masterprojekt_zoellner.pdf.

Zöllner, Michael. Using State Machines For Interaction Design - A State Of The Art Analysis, 2011b. Master Seminar Paper. URL: http://hci.uni-konstanz.de/downloads/studentrepos/WS1112/MA_Seminar_Zoellner.pdf.

Zöllner, Michael, Jetter, Hans-Christian, and Reiterer, Harald. ZOIL: A Design Paradigm and Software Framework for Post-WIMP Distributed User Interfaces. In José A. Gallud, Ricardo Tesoriero, and Victor M. Ruiz Penichet, editors, *Distributed User Interfaces - Designing Interfaces for the Distributed Ecosystem*, Human-Computer Interaction Series, chapter 10, pages 87–94. Springer-Verlag, 2011. ISBN 978-1-4471-2270-8.

List of Figures

1 *How the computer sees us* [O'Sullivan and Igoe, 2004] 2

2 The articulatory and semantical distance of the gulfs of execution and evaluation influence the feeling of directness [Hutchins et al., 1985] 11

3 The `ReleaseLink` behavior of Facet-Streams 31

4 A huge gap exists between natural language description and implementation 33

5 Finite-State Machines represent interaction at a higher level of abstraction 39

6 Specifying an *entry* guard (right) instead of multiple transition actions (left) 43

7 Specifying general purpose guards for *entry* and *exit actions* 43

8 Notation of a transition (top) and its execution flow (bottom) 44

9 Triggered and timed transitions are used to model the activation and deactivation of a control 45

10 Possible concepts for animated transitions without changes to the default notation . 46

11 Animated transition with a materialized transitioning state 47

12 Activation and deactivation of a control with fade-in and fade-out animations 48

13 Usage of the `#Count` operator in Facet-Streams 51

14 Clustering multiple states into a super-state 52

15 Refining a state by introducing sub-states 52

16 Orthogonality simplifies complex state machines by introducing concurrency 53

17 The broadcast mechanism of the Statecharts notation enables the coupling of orthogonal states 53

18 Declarative state machine frameworks are positioned on a higher level of abstraction than low-level implementation techniques 54

19 Example state machine subset 55

20 Marble diagrams are used to visualize observables 70

21 Marble diagram with an exception 71

22 Primitive creation operators 73

23 Creating time-based observables with `Timer()` and `Interval()` 73

24 The `Where()` operator filters out values that do not satisfy a given condition 74

25 The `Select()` operator projects every value to a new one 75

26 The `Delay()` operators delays each `OnNext()` call for a given timespan 75

27 Configuration of states, transitions and animations in Expression Blend 82

28 Meta-States of a `ReactiveStateMachine<T>` instance 85

29 An example state machine with different entry actions 87

30 Diagram of the basic architecture of Reactive State Machine 95

31 `FacetWheel` in the *FacetWheelCollapsed* state 104

32 `FacetWheel` in the *FacetWheelFadingIn* state 104

33 `FacetWheel` in the *FacetWheelVisible* state 104

34 State machine model of the `FacetWheel` 105

35 State machine model of the lifecycle of the `FacetWheel` 106

36 Two fingers affecting a `FacetWheel` 108

37 State machine subset of the selection behavior 108

38 The entire mechanism to forward an event from the user interface to the state machine 115

39 State machine model of the releasing of a link 116

40 Physical setting of SmartShare 118

41 Personal territory of a user 118

42 Overview of the login procedure in SmartShare 118

43 Overview of the state machines that control the `QRControl` and the `NameControl` . 119

44 Overview of the steps to get data onto the table 121

45 The user can switch between his smartphone and a public folder of a cloud storage service to get images onto the table 121

46 The state machines of the `DropControl` and the `TargetSwitcher` 122

47 Overview of the steps to get data from the table into the smartphone or cloud storage service 124

48 State machines of the `SendControl`, the `TargetSwitcher` and the `InformationItem`125

49 Dualizing a method by interchanging return types and parameters 151

50 The `Take()` and `Skip()` operators filter out a given amount of values at the end or beginning of a collection 156

51 The `SelectMany()` operator produces one observable for every value and flattens them into the result observable 157

52 `Sample()` and `Throttle()` limit the number of values based on time 158

53 Marble diagram of the sliding sum aggregate of listing 53 on page 159 159

54 The `Concat()` operator concatenates two observables sequentially 160

55 The `Zip()` operator aggregates pairs of two observables into one observable 161

56 The `CombineLatest()` operator takes the latest values of each observable and puts them into one observable 161

List of Tables

1 Characteristics of WIMP and post-WIMP systems 9

2 Comparison of naive implementation techniques and finite-state machines 38

3 Comparison table of ready-to-use features available in selected FSM frameworks . . 59

4 Summary table of the informal evaluation of the Reactive State Machine framework 127

List of listings

1 Detecting a tap gesture on a single-touch device 29
2 Basic implementation of multi-touch manipulations 30
3 Implementation of the `ReleaseLink` behavior with a timer 32
4 Structural and behavioral code gets mixed with low-level techniques 56
5 Only structural code is needed with declarative state machine frameworks 56
6 Event tokens need to be feeded into the state machine of the Stateless framework . . 60
7 Example of a drag operation modeled with Stateless 61
8 Accessing event metadata in a transition of Swing States 62

9 Accessing event metadata in a transition of Qt's SMF 62
10 Definitions of `IObservable<T>` and `IObserver<T>` 67
11 The `Subscribe()` method either takes a full-fledged observer or delegates for the three methods 72
12 Wrapping a `TouchEvent` into an observable 74
13 With regular events, filters must be specified in the event handler method 76
14 With Rx operators, a filtered event can be specified straightforwardly 77
15 With Rx operators, the `ReleaseLink` behavior of Facet-Streams is specified concisely 78
16 Defining two orthogonal VisualStateGroups in XAML 80
17 Defining states in XAML 80
18 Defining transitions in XAML 81
19 Triggering a transition from code-behind 82
20 Creating a simple state machine instance 85
21 Specification of the entry actions of figure 29 on page 87 87
22 Overview of all options to configure triggered transitions 89
23 Overview of all options to configure timed transitions 90
24 Overview of all options to configure automatic transitions 91
25 A simple RSM definition which models the visibility states of a UI element 92
26 Definition of the UI element's visual appearance during an animated transition 92
27 The logical and visual definition of the state machine are connected with a special behavior 93
28 Overview of all operators of the multi-point notation 94
29 Triggered transitions are immediately subscribed to conserve their parameters 97
30 Timed transitions are enabled on state entry and disposed on state exit 98
31 The tracking mechanism overrides the `TransitionOverride()` method to intercept transitions 102
32 Input point tracker implementation for Windows Touch input 102
33 Current implementation of a state machine subset of Facet-Streams 109
34 Old implementation of a state machine subset of Facet-Streams 110
35 Old implementation of the timer event handler in Facet-Streams 112
36 XAML code to map the VSM to the RSM 112
37 XAML definition of the VSM animations 113
38 Wrapping a `TouchDown` event inside an observable collection 114
39 The `EventToObserver` element forwards input events to an observer 114
40 Events are represented by properties which are data-bound to the `EventToObserver` 115
41 Implementation of the `ReleaseLink` behavior with the RSM 117
42 Implementation of the state machine of the `QRControl` 120
43 Implementation of the state machine of the `NameControl` 120
44 State machine implementation of the `DropControl` 123

45 The interfaces `IEnumerable<T>/IEnumerator<T>` are used to iterate over collections 150
46 Dualizing `IEnumerable<T>` in 4 steps 152
47 Dualizing `IEnumerator<T>` in 5 steps 153
48 Definitions of `IObservable<T>` and `IObserver<T>` 154
49 Observable wrapper to asynchronously observe the console 155
50 Converting pull-based collections to observables 155
51 Examples of operators that convert observables back into the non-observable world . 156
52 `SelectMany()` can be used to create cross-products of several collections 157
53 The `Scan()` operator creates a sliding sum aggregate 159
54 `GroupBy()` groups values according to a common characteristic 159
55 `Window()` produces sliding windows of subsequent values 160

A Appendix: Reactive Extensions

A.1 Dualizing `IEnumerable<T>/IEnumerator<T>`

The .NET framework includes many different collection types, such as index-based arrays, lists, stacks, queues and dictionaries, that can be used for various purposes. The essence of all these collections is captured by a pair of interfaces (see listing 45) whose purpose is to iterate over a collection. Iterators are a well-known concept both in object-oriented software engineering, where the Iterator pattern is specified [Gamma et al., 1995, 289–304], and in database systems, where they are known as cursors [ITL Education Solutions Limited, 2010, 166-167].

```
public interface IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T>
{
    public bool MoveNext();
    public T Current {get;}
}
```

Listing 45: The interfaces `IEnumerable<T>/IEnumerator<T>` are used to iterate over collections

In .NET, every collection type implements the `IEnumerable<T>` interface. It contains a single method `GetEnumerator()` which returns the iterator (of type `IEnumerator<T>`) that can be used to iterate over the collection. The `IEnumerator<T>` interface works as follows: With a call to `MoveNext()` the user of an iterator can check if there are any elements left in the collection. If it returns `true`, the user can then access the next element via the property `Current`. If it returns `false`, the user knows that he reached the end of the collection. The contract of this interface thereby resembles a pull-based protocol, where the consumer of the interface continuously asks for new data until no more data is available. One of the characteristics of this protocol is that the call to `MoveNext()` blocks the current thread. This is always the case, yet only noticed when the data of the collection is not stored in main memory, but has to be received from some slower place (disk, network, etc.) or computed during iteration using the `yield` pattern. A process that is waiting for the answer of `MoveNext()` effectively wastes resources and, depending on the implementation, also potentially blocks other parts of the application such as the UI. The nature

of this pull-based protocol is therefore inherently synchronous. It is obvious that the blocking of a thread is a highly unwanted behavior in reactive applications. In fact, what is desired is the opposite behavior. Instead of waiting for a `MoveNext()` call to produce a value, the value should be pushed towards the respective entity, once it has been produced, while in the meantime the entity does not waste any resources to wait for the value. To get this desired behavior, the creators of Rx decided to dualize the `IEnumerable<T>/IEnumerator<T>` interface pair, as "the dual of getting stuck is getting too much"[32]. The following paragraphs, which are based on a Rx Keynote[32], show the process of dualizing `IEnumerable<T>/IEnumerator<T>` into another pair of interfaces (`IObservable<T>/IObserver<T>`) which is able to express reactive or push-based behavior.

Duality is a term of category theory in mathematics, which deals with mathematical structures and relationships between them. In category theory, duality "expresses the fact that one category is the opposite of another" [Awodey, 2006, 14]. The dual of a category is obtained by reversing the relationships. In most cases this means that the arrow or composition is reversed. An example is the function $f : A \longrightarrow B$ whose dual is the function $f' : B \longrightarrow A$, or the composition $f \circ g$ whose dual is the composition $g \circ f$ [Pierce, 1991, 8]. Translated to methods of a programming language, dualization means that the arguments and return types of the methods are interchanged. Instead of returning an object of type T, the dual method now receives an object of type T and vice versa (see figure 49).

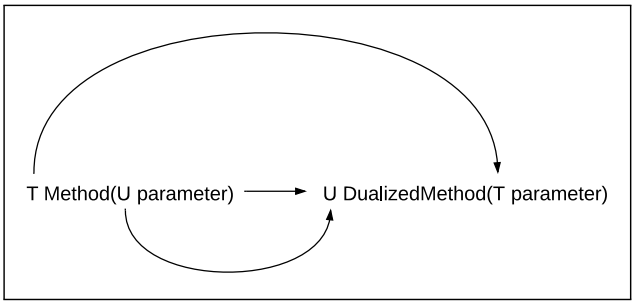

Figure 49: Dualizing a method by interchanging return types and parameters

Listing 46 on the following page shows all steps of dualizing `IEnumerable<T>`. As the original definition (1) hides some details that are implicitly contained after compilation, the explicit pseudo code (2) therefore more accurately resembles the actual interface definition. One such implicit assumption is that the return type of `GetEnumerator()`,`IEnumerator<T>`, also implements the `IDisposable` interface. Thus, the return type of `GetEnumerator()` is actually a combination of both `IEnumerator<T>` and `IDisposable`. Since the functionality of `IDisposable` is not important for the asynchronous aspect, it is therefore not dualized. Another implicit assumption is,

[32] http://channel9.msdn.com/Blogs/codefest/DC2010T0100-Keynote-Rx-curing-your-asynchronous-programming-blues

that if no argument is passed into a method, this can be seen as passing the empty type `void` as argument. After the first step (3) of dualizing `IEnumerable<T>` to its dual `IEnumerableDual<T>`, the name of the `GetEnumerator()` method has changed to `SetEnumerator()` and this method now receives an `IEnumerator<T>` as parameter. In the second step (4) the redundant `void` is removed and the `IEnumerator<T>` is replaced by its dual, that will be created next.

```
public interface IEnumerable<T>          //(1)
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable<T>          //(2)
{
    (IEnumerator<T> & IDisposable) GetEnumerator(void);
}

public interface IEnumerableDual<T>        //(3)
{
    (void & IDisposable) SetEnumerator(IEnumerator<T> enumerator);
}

public interface IEnumerableDual<T>        //(4)
{
    IDisposable SetEnumeratorDual(IEnumeratorDual<T> enumeratorDual);
}
```

Listing 46: Dualizing `IEnumerable<T>` in 4 steps

Listing 47 on the next page shows all steps of dualizing `IEnumerator<T>`. Again, the original definition of this interface (1) is made more explicit first (2). As can be seen, the property `Current` is actually converted by the compiler to a method `GetCurrent(void)` which returns the next object of the iterator. It is also made explicit that both `MoveNext()` and `Current` can return or throw an exception, a fact that is hidden in the implicit definition. In Java such a fact would be explicitly marked with the `throws Exception` keyword. Next (3), further aspects are made explicit: The boolean return type of `MoveNext()` can be resolved to either `true` or `false`. However as returning `true` is equal to returning the actual value of type `T` it can be omitted and replaced by `T`. This renders the `GetCurrent()` method useless. The resulting pseudo code definition expresses compactly what the `IEnumerator<T>` interface does: Either `MoveNext()` returns a value of type

T, an Exception or false which marks the end of the collection.

```
public interface IEnumerator<T> : IDisposable        //(1)
{
    bool MoveNext();
    T Current {get;}
}

public interface IEnumerator<T> : IDisposable        //(2)
{
    (bool | Exception) MoveNext(void);
    (T | Exception) GetCurrent(void)
}

public interface IEnumerator<T> : IDisposable        //(3)
{
    (T | false | Exception) MoveNext(void);
}

public interface IEnumeratorDual<T>                  //(4)
{
    void GotNext(T | false | Exception);
}

public interface IEnumeratorDual<T>                  //(5)
{
    void OnNext(T next);
    void OnCompleted();
    void OnError(Exception e);
}
```

Listing 47: Dualizing IEnumerator<T> in 5 steps

After the first dualization step (4) the return types and parameters of MoveNext() have been interchanged and its name has been altered to GotNext(). This expresses the fact that values are now passed into the method instead of returned from it. However the type combination that is currently passed into the method is no valid construct in .NET. Therefore the method is split into three methods, which consume the parameters (5). Note that the false parameter can be

omitted completely, as it never changes and the semantics are made clear by naming the method `OnCompleted()`, as its purpose is to signal that the collection has no further elements.

Finally, both interfaces are renamed to `IObservable<T>` and `IObserver<T>`. Every type that implements the `IObservable<T>` interface is said to be an observable collection, or simply an *observable*. Every observable can be observed by any type that implements the `IObserver<T>` interface. Additionally, the `SetEnumeratorDual()` method of the `IObservable<T>` interface is renamed to `Subscribe()`. With this renaming it is made clear that the `IObservable<T>/IObserver<T>` pair of interfaces resembles the popular Observer Pattern known from object-oriented software engineering [Gamma et al., 1995, 326–337].

```
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<T>
{
    void OnNext(T next);
    void OnCompleted();
    void OnError(Exception e);
}
```

Listing 48: Definitions of `IObservable<T>` and `IObserver<T>`

A.2 Additional Reactive Extension Operators

A.2.1 Additional Conversion Operators

In addition to the wrapping of events, asynchronous methods can be wrapped into observables. Listing 49 shows how input from the console can be observed asynchronously with an observable wrapper.

```
//observing the console input as a stream of integers
var inputStream = Console.OpenStandardInput();
var buffer = new byte[1024];
IObservable<int> consoleInput;

consoleInput = Observable.FromAsyncPattern<byte[], int, int, int>(
    inputStream.BeginRead, inputStream.EndRead)(buffer, 0, 1024);
```

Listing 49: Observable wrapper to asynchronously observe the console

Another option is to convert a pull-based collection to an observable. This `ToObservable()` operator is actually an extension method of the `IEnumerable<T>` interface. It can be handy in many scenarios, for example when a user interface is about to be unit tested. A collection of input events can be created manually and fed to the application as an observable collection. As the application can also receive regular input events by means of an observable, this mocked input data can be integrated rather easily. Listing 50 shows how this operator can be used.

```
IObservable<int> intObservable = new List<int>(){1,2,3,4}.ToObservable();
IObservable<char> charObservable = "Characters".ToObservable();
IObservable<MouseEventArgs> mouseMoves = new List<MouseEventArgs>(){...}
                                        .ToObservable();
```

Listing 50: Converting pull-based collections to observables

Because of the duality between the two interface pairs `IEnumerable<T>/IEnumerator<T>` and `IObservable<T>/IObserver<T>` it is also easily possible to reverse the conversion and create pull-based collections out of an observable. Extension methods are included for a few popular collection types, such as `List<T>`, `Array<T>` or `Dictionary<T,U>`. Beyond that, there even exist operators that create regular .NET events and methods of the asynchronous programming model out of observables. Thus, a whole roundtrip is possible with almost all asynchronous sources.

Listing 51 shows examples of these operators, that convert the observables of listing 50 on the previous page back into the non-observable world.

```
List<int> intList = intObservable.ToList();
char[] charArray = charObservable.ToArray();

IEventSource<EventPattern<MouseEventArgs>> mouseMoveEvent;
mouseMoveEvent = mouseMoves.ToEventPattern();
mouseMoveEvent.OnNext += myMouseHandler;
```

Listing 51: Examples of operators that convert observables back into the non-observable world

A.2.2 Additional Filter Operators

In addition to the `Where()` filter, there exist several other filter extension methods. For example, the `Take()`, `TakeWhile()` and `Skip()`, `SkipWhile()` operators, which take or skip elements from an observable, or take or skip elements while some condition is true (see figure 50). Then there is the `Distinct()` operator which leaves back only distinct values and throws all duplicate values away or the `DistinctUntilChanged()` operator which does the same for subsequent distinct values.

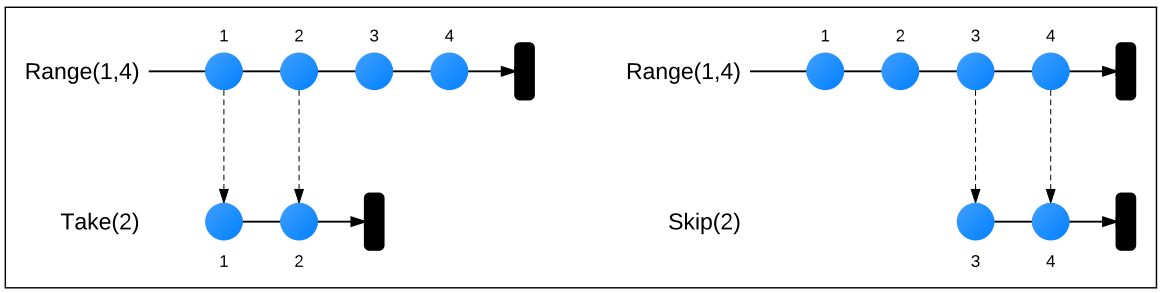

Figure 50: The `Take()` and `Skip()` operators filter out a given amount of values at the end or beginning of a collection

A.2.3 Additional Projection Operators

Similar to the `Select()` operator, the `SelectMany()` operator also applies a function to every value of the observable. This function however does not simply return a single modified value, but another observable. So, for every value in the source observable another observable is returned, resulting in a nested set of observables. These nested observables are immediately flattened, so

that the return value of the `SelectMany()` operator is again a simple observable. Figure 51 shows an example of the `SelectMany()` operator. The main usage of this operator is to create cross-products of two or more observables. In listing 52 the `SelectMany()` operator is used to model a drag&drop operation. The example can be read as follows: "For every `MouseDown` event take all `MouseMove` events, until a `MouseUp` event occurs". It is written in both LINQ syntax (1), which more concisely expresses the intent of the operation and extension method syntax (2), which is equivalent to the former, but not as expressive.

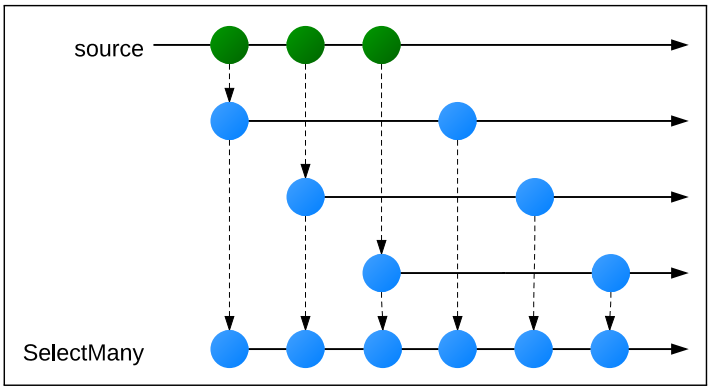

Figure 51: The `SelectMany()` operator produces one observable for every value and flattens them into the result observable

```
IObservable<MouseButtonEventArgs> mouseDowns = ...;
IObservable<MouseEventArgs> mouseMoves = ...;
IObservable<MouseButtonEventArgs> mouseUps = ...;

//(1) LINQ Syntax
var dragOperations = from mouseDown in mouseDowns
                     from mouseMove in mouseMoves.TakeUntil(mouseUps)
                     select mouseMove

//(2) Extension Method Syntax
var dragOperations = mouseDown.SelectMany(
                        mouseDown => mouseMoves.TakeUntil(mouseUps));
```

Listing 52: `SelectMany()` can be used to create cross-products of several collections

A.2.4 Additional Time-based Operators

Several time-based operators exist in addition to the `Delay()` operator. With the `Sample()` operator it is possible to sample the collection every n milliseconds if too many values arrive in a short time span. With `Throttle()`, a value has to wait a given timespan before it is forwarded. If another value is coming in during this timespan, the former value is discarded (see figure 52 for a marble diagram of `Sample() and Throttle()`).

Additional operators are the `Timestamp()` operator which attaches a timestamp to every value, or the `Timeout()` operator which throws an exception if the observable does not produce any value within a given timespan.

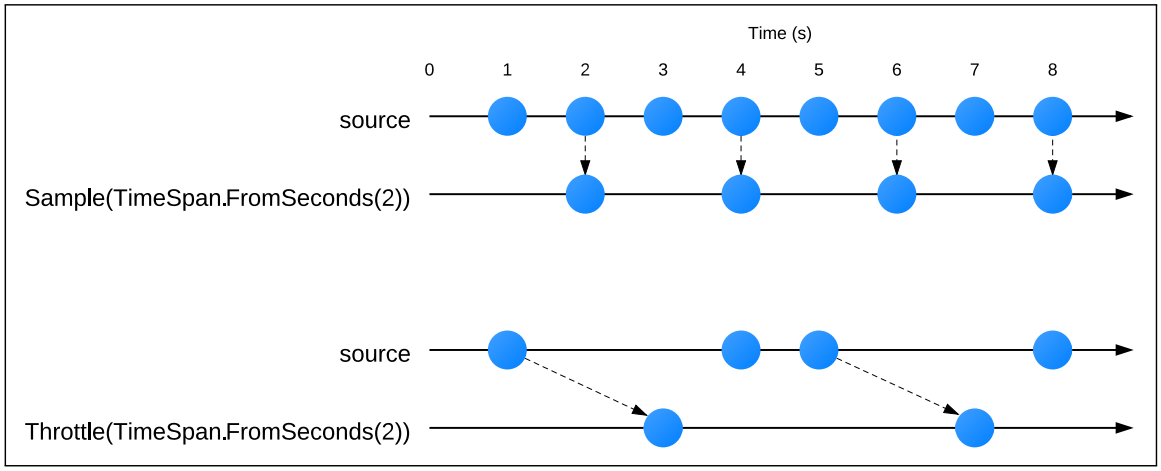

Figure 52: `Sample()` and `Throttle()` limit the number of values based on time

A.2.5 Aggregation Operators

The aggregation operators take all values of an observable and aggregate them into a single value. Thereby they break the compositionality of the observable. Examples of such aggregation operators are the `Sum()`, `Count()`, `Average()`, `Min()` and `Max()` operators, whose semantics are obvious. There is also a generic `Aggregate()` operator which can be used to compute individual aggregations. Similar to this is the `Scan()` operator, which however calculates the aggregated value every time a new value arrived at the source observable, thereby creating a sliding aggregate. See listing 53 on the following page for a sliding sum aggregator, which returns the sum of all previous values whenever the source observable produces a value.

```
Observable.Range(1, 4).Scan((sum, i) => sum + i);
```

Listing 53: The `Scan()` operator creates a sliding sum aggregate

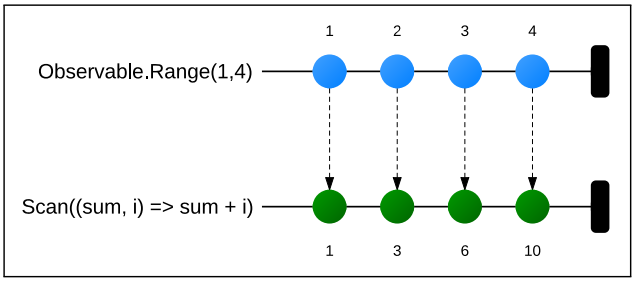

Figure 53: Marble diagram of the sliding sum aggregate of listing 53

A.2.6 Grouping and Windowing Operators

These types of operators are used to combine several values of one observable into an observable of aggregations of these values. The way the aggregation is performed depends on the operator. The `GroupBy()` operator aggregates values according to a group characteristic. All values that have the same group characteristic are put into the same group. See listing 54 for an example where strings of the same length are put into individual groups.

The `Window()` and `Buffer()` operators perform aggregation on subsequent values. While the `Window()` operator returns its aggregations again as observables, the `Buffer()` operator creates regular lists of the aggregations. Listing 55 on the following page shows how a sliding window can be implemented with the `Window()` operator to create a sliding average of three subsequent `double` values. The size of the window and its steps can be adjusted by parameters.

```
var strings = new[]{"one", "two", "three", "four"}.ToObservable();

var sizeGroups = strings.GroupBy(s => s.Length);
//produces an observable with three groups (for string lengths 3, 4 and 5)
```

Listing 54: `GroupBy()` groups values according to a common characteristic

```
var data = new[] {3,4,2,3,2,2,2,4,3,2,4}.ToObservable();
var averages = data.Window(3,1).SelectMany(window => window.Average());
//produces the values 3, 3, 2.33, 2.33, 2, 2.66, 3, 3, 3
```

Listing 55: `Window()` produces sliding windows of subsequent values

A.2.7 Coordination Operators

Several operators exist to coordinate the behavior of multiple observables. One part of coordination is the correct ordering of multiple observables. The `Concat()` operator concatenates two observables sequentially, thereby creating a defined order (see figure 54). This only works when the source observable completes.

`TakeUntil()` and `SkipUntil()` take or skip values from the source observable until a second observable produces any value. Then, the observable completes. The `TakeUntil()` operator has been used above for the drag&drop example (see listing 52 on page 157), where values have been taken from the `MouseMove` observable until the `MouseUp` observable produced a value.

Another part of coordination is to handle parallel observables. With the `Merge()` operator, two or more observables are flattened into one. Here, the resulting observable is of the same type as the source observables. In contrast to this, the `Zip()` operator takes a value from every input observable, applies a function to both and pushes the result of the function in an output observable (see figure 55 on the next page). While `Zip()` always selects distinct pairs of values, `CombineLatest()` always takes the latest value from one observable to combine it with the latest value of the other (see figure 56 on the following page).

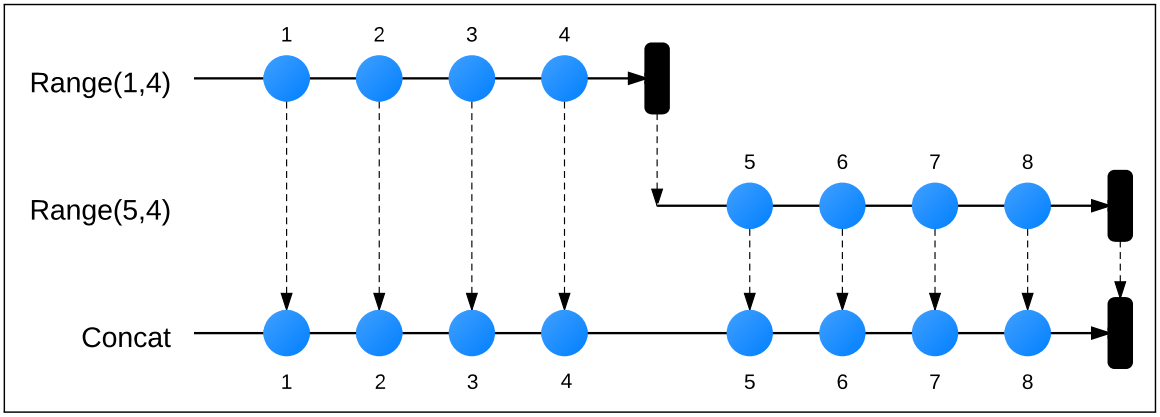

Figure 54: The `Concat()` operator concatenates two observables sequentially

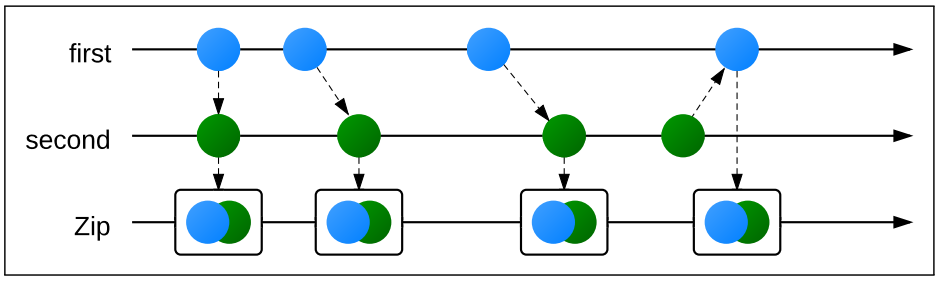

Figure 55: The `Zip()` operator aggregates pairs of two observables into one observable

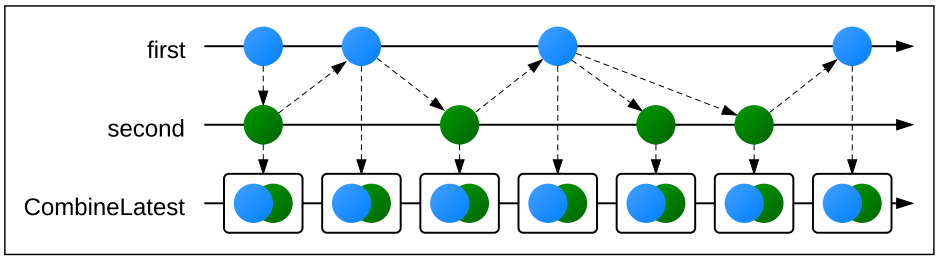

Figure 56: The `CombineLatest()` operator takes the latest values of each observable and puts them into one observable

A.2.8 Other Operators

Many additional operators exist, which can not be addressed in the necessary detail here. Some of them are rather simple, such as those which check if specific elements are contained in the observable or those which provide access to specific elements. Others are inherently complex, such as the join operator which resembles the join operator of relational databases. Then, there are operators that allow the sharing of a subscription to bypass subscription side effects. And finally there are also meta-operators that translate an observable collection of type `T` into an observable collection of type `Notification<T>`. Thereby these notifications represent the `OnNext()`, `OnError()` and `OnCompleted()` calls of the underlying observable.