Nicolai Marquardt

Developer Toolkit and Utilities for Rapidly Prototyping Distributed Physical User Interfaces

Bauhaus-University Weimar, Germany
Faculty of Media
Media Systems Science

Bauhaus-Universität Weimar

Developer Toolkit and Utilities for Rapidly Prototyping Distributed Physical User Interfaces

Diploma Thesis

Nicolai Marquardt
Date of birth: November 30, 1979
in Bad Nauheim, Germany

Student ID: 10599

1st Reviewer: Prof. Dr. Tom Gross
2nd Reviewer: Prof. Dr. Bernd Fröhlich

Date of Submission: March 10, 2008

Nicolai Marquardt
mail@nicolaimarquardt.com
Student ID: 10599

Developer Toolkit and Utilities for Rapidly Prototyping Distributed Physical User Interfaces

Diploma Thesis

Cooperative Media Lab
Bauhaus-University
Bauhausstr. 11, 99423 Weimar
Germany

1st Reviewer: Prof. Dr. Tom Gross
2nd Reviewer: Prof. Dr. Bernd Fröhlich

Abstract

Distributed physical and tangible user interfaces represent the vision of building embedded computing systems that move off the desktop, in order that the interaction takes place in—rather than apart from—our everyday environment. These systems take advantage of our practical skills and senses to provide intuitive interfaces. For instance, they facilitate the communication over distance, sharing of digital media, and interaction with electronic devices. The systems are often built as *information appliances* that are specialised for a specific task. They are assembled as units consisting of input controls, sensors, actuators, and displays. These appliances, however, are still difficult to build; especially if distributed devices are connected over a network.

In this thesis I focus on the research of methods and tools to support developers to rapidly prototype these distributed physical user interfaces. The developed Shared Phidgets toolkit integrates distributed sensors and actuators, and provides easy to use programming strategies for developers to build their envisioned interactive systems. The *runtime platform* of the toolkit hides the complexity of hardware integration and network synchronisation. The implemented *developer library* as well as the introduced programming strategies address developers with diverse development skills. To support the testing, debugging, and deployment of appliances, diverse *utilities* allow the monitoring and control of all connected components at runtime. For instance, visualisations can be used to explore the distributed hardware components and the built appliances in their geographical context. These utilities allow gaining insight into the internal communication processes of the distributed infrastructure. Furthermore, the simulation utilities facilitate the testing and debugging of the developed appliances. Finally, appliance case studies illustrate the applicability of the toolkit and the provided utilities to support the rapid prototyping process. A critical discussion of the toolkit and the built information appliances concludes the thesis.

Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Tom Gross for his support and guidance throughout my university studies. I really appreciate his advice for this thesis as well as his help with all my questions and concerns.

I would like to thank Prof. Dr. Bernd Fröhlich for his support and for being the second reviewer of this thesis.

I would also like to thank Dr. Schalbe for his help; even in moments when I had trouble with the planning of my thesis and defence.

Many thanks to Prof. Dr. Saul Greenberg for his guidance as advisor of my project at the GroupLab, for making this great research visit in Calgary possible, and for his helpful advice for this thesis.

I would like to thank Christian for his help, and Petra for her support with this thesis layout as well as her answers to the many emails I have sent.

Thank you to all students at the Cooperative Media Lab, the GroupLab, and the Interactions Lab: I'm very glad that I was able to spend the time of my university research with these great friends.

Many thanks to Chester and Patrick for building fantastic electronic hardware devices and for their help with questions I had about Phidgets hardware.

I would like to thank my parents for their patience and support throughout my university studies.

Finally, I would like to thank my fiancée Kristin, for being there for me whenever I needed her. Without her motivation I would still think about the first sentence of this thesis.

Publications

Materials, ideas, and figures from this thesis have appeared previously in the following publication:

MARQUARDT, N. AND GREENBERG, S. 2007. Distributed Physical Interfaces with Shared Phidgets. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction - TEI 2007 (Baton Rouge, LA, USA)*. ACM Press, New York, NY, USA, 13–20.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This thesis addresses the problem of how developers can rapidly prototype physical user interfaces with distributed hardware devices. To ground this research project, the chapter begins with a motivation of the research area and a description of the research problem. Then, an overview of the solution to address this problem is provided. The section that follows briefly summarises the contributions of the thesis. Finally, the chapter is concluded with an overview of the subsequent chapters.

1.1 Motivation

In recent years, computing technology is becoming increasingly important in our work environment, as well as our domestic surrounding. Inspired by Mark Weiser's vision of *ubiquitous computing* [Weiser, 1991] this leads to seamless integrated computing technology in our everyday environment that is disappearing and pervasive. Weiser envisioned a future with embedded computing technology of different form factors that recedes into the background and provides users helpful information exactly where it is needed [Weiser and Brown, 1997]. The technology should work within—rather than apart from—the everyday practices of people [Dourish, 2001].

Many of these visions of ubiquitously available systems in our environment can be described as *information appliances*: they are built and optimised to support specific tasks of the users, and are available in the environment for when they are needed [Norman, 1999]. These information appliances enhance our social and domestic activities, for instance by letting people browse and share digital me-

dia like photos, videos, and music [Fitzmaurice et al., 1995; Barrett and Maglio, 1998]; send and receive messages or reminders [Elliot et al., 2007; Sellen et al., 2006]; stay in contact with distant family members and friends by using messengers or ambient displays [Wisneski et al., 1998; Nagel et al., 2001; Mynatt et al., 1998; Consolvo and Towle, 2005]; and assist them in the smart home [Kidd et al., 1999; Helal et al., 2005; Babulak, 2006; Brumitt et al., 2000].

To facilitate the easy and non-disruptive interaction of users with these *reactive environments*, and to *"build upon users' existing skills, rather than demanding the learning of new ones"* [Buxton, 1997], the implementations of these systems avoid the users' interaction with mouse and keyboard and instead move off the desktop into people's everyday environment [Weiser, 1991]. The systems often implement a *physical or tangible user interface* that allows users to interact more intuitively by addressing the users' physical senses and skills [Ishii and Ullmer, 1997; Greenberg and Fitchett, 2001]. These physical user interfaces are usually built with components like sensors, actuators, and displays. These components enable the system to react to explicit (e. g., touch sensors, buttons) and implicit input (e. g., motion sensor, temperature sensor). In turn, the system can generate a response with actuators (e. g., servo motors) or displays (e. g., small embedded colour displays). For many information appliances these sensors and actuators are not only located at one place. With the creation of *distributed physical interfaces* developers can create a series of devices that can intercommunicate, or a device that may have its subcomponents physically distributed in remote located places. This could be for instance a series of sensors and actuators situated and embedded in several rooms in the domestic space.

Researchers already have created many examples of such information appliances, but the problem is that these technologies are still very difficult to build, especially when distributed hardware components should be integrated. Developers of distributed physical user interfaces have to deal with a variety of complex problems when creating even a simple device [Abowd, 1996; Weiser, 1993; Greenberg and Fitchett, 2001]. They need to build electronics hardware, write software to access the hardware, use networking protocols, solve networking issues to synchronise distributed components, manage runtime robustness and failure issues, control the distributed hardware and appliances, and so on. Not only is this hard to do, but in practice very few developers have all the skills required to take on such an onerous task [Greenberg and Fitchett, 2001; Helal, 2005]. This inhibits the developer's ability to easily build prototypes. Experimenting with such rapid prototypes, however, is essential for the exploration of alternative designs of the developer's envisioned information appliances. [Greenberg, 2007].

To make it easier for developers to deal with these issues, rapid prototyping toolkits facilitate the development process. Besides the low-fidelity prototyping with

pen and paper [Liu and Khooshabeh, 2003; Rudd et al., 1996], sketching [Buxton, 2007], and storyboards [Preece et al., 2002], the development of physical high-fidelity prototypes is an important part of the iterative design process. These high-fidelity prototypes let designers easily explore initial designs and improve upon good ones by letting them try multiple ideas variations across an iterative design cycle [Buxton, 2007; Greenberg, 2007]. For the development of such a toolkit for physical interfaces it is important to consider the previous experience of developers with the prototyping of graphical user interfaces (GUI) [Myers et al., 2000].

Although there already exist toolkits that facilitate the integration of local physical hardware elements like sensors and actuators into custom software [Greenberg and Fitchett, 2001; Lee et al., 2004; Villar and Gellersen, 2007; Klemmer et al., 2004], they do not address and facilitate the problems of the development with distributed hardware. Moreover, existing toolkits that support the distributed development introduce high-level abstractions that allow the assembly of new applications based on these abstractions [Salber et al., 1999; Matthews et al., 2004]. These systems, however, do not facilitate the direct composition of the underlying distributed sensors and actuators to these high-level abstractions. Therefore, a toolkit is needed that is positioned in between these two prototyping toolkit categories and that provides adequate development utilities for the building of prototypes with access to remotely located sensors and actuators.

1.2 Solution Overview

To address the problems described in the previous section of the chapter, the objective of this thesis project is the development of a development toolkit that facilitates the development of distributed physical user interfaces. The developed Shared Phidgets toolkit hides the complexity of the hardware access, device exploration, network communication and synchronisation from the developer of information appliances. The developers can access the distributed physical sensors and actuators through an easy to use object-oriented *Application Programming Interface (API)* of the Shared Phidgets *developer library*. Therewith, developers can build their envisioned distributed information appliances, and do not have to deal with low level implementation issues like device access and network synchronisation.

The toolkit provides a *runtime platform* that automatically integrates locally attached hardware sensors and actuators and makes these devices accessible over the network. Thus, developers can easily instantiate *infrastructures* that comprise multiple network connected computers and all the connected physical interface

components. The software of the Shared Phidgets toolkit runtime platform works autonomously in the background of all client computers, and manages the network connections to a common shared data space. The synchronisation between all client computers and the connected hardware devices is handled via data exchange over this shared data space.

The Shared Phidgets toolkit is built upon existing research projects of the University of Calgary, including the local *Phidgets* toolkit [Greenberg and Fitchett, 2001; Phidgets Inc., 2008]. Phidgets stands for *Physical Widgets*, as they represent the hardware equivalent to widgets that are available in interface builders for graphical user interfaces (GUI). They are a collection of hardware building blocks for physical user interfaces (e. g., motion and distance sensors, RFID readers, motors, servos, displays, buttons, sliders, and more) that developers can use with little or no electronics engineering knowledge[1]. The Shared Phidgets toolkit integrates all these currently available Phidget devices, and makes them easily accessible over the network. The toolkit is, however, not limited to Phidget hardware devices and allows the easy integration of custom hardware into the runtime architecture. Therefore, the Shared Phidgets toolkit includes the implementation of further hardware devices, for instance graphical colour displays and receivers of the Global Positioning System (GPS) satellite signals.

The design of the developer library follows the principle that simple things should be easy to build, and hard or complex things should be still possible [Greenberg, 2007]. It is important to avoid a steep learning curve for programmers [Myers et al., 2000]; instead the toolkit should address the increasing requirements of developers with adequate tools. Therefore an important design objective of the Shared Phidgets toolkit is a *low threshold* [Myers et al., 2000] for developers with no previous experience of programming physical hardware[2], in order that they can learn how to efficiently use the toolkit. The strategies to achieve this low threshold are the easy to use exploration and discovery utilities, object-oriented programming proxy objects, event-based architecture, graphical interface skins, advanced infrastructure visualisations, and a close integration into the Integrated Development Environment (IDE). The toolkit furthermore provides a *high ceiling* [Myers et al., 2000] for experienced developers[3]. These developers can program information appliances more efficiently with the direct access to the underlying shared data model, integrate custom hardware with the extensible plug-in architecture, or use high-level events for the development of abstract appliances.

1 As a former research project of the University of Calgary, the hardware is now manufactured and distributed by Phidgets Inc. (http://www.phidgets.com/) in Calgary. The company sells these hardware components to academic and industrial research labs for implementing physical user interfaces.

2 This category of developers is subsequently described as *average developers*.

3 This category of developers is subsequently described as *expert developers*.

As the testing, debugging, and deployment especially of distributed information appliances is a difficult task for developers [Klemmer et al., 2004; Matthews et al., 2004], a set of utilities allows the monitoring and control of all connected components at runtime. These utilities allow the access to diverse abstraction layers of the toolkit, for instance the direct access to the shared data model, graphical interface representations for hardware devices, or an overview of all currently running information appliances. An advanced visualisation utility can be used to explore the distributed hardware components and the built appliances in their geographical context, and allows insights into the internal processes of the distributed infrastructure with *details on demand* [Shneiderman, 1996]. Furthermore, simulation utilities support developers to test and debug appliances [Li et al., 2007; Dey, 2000; Klemmer et al., 2004], which is also useful for the case that not all of the used hardware devices are available. These simulation user interfaces make it easier for developers to test the functionality of their created appliances, before testing it with all the remotely located embedded hardware components in the environment.

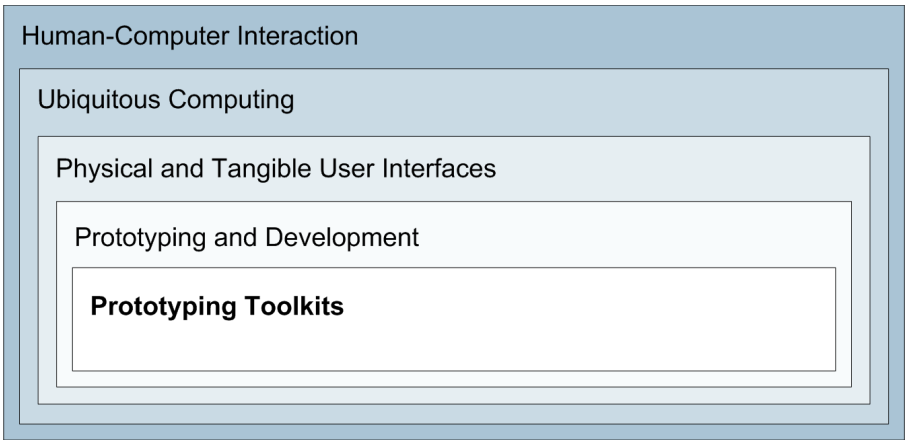


Figure 1.1: The scope of the thesis research project.

Collectively, this thesis project provides a toolkit for the development of physical user interfaces, with focussing on the support for the prototyping of *distributed* information appliances. The thesis explores the process of building prototypes of physical user interfaces, and provides helpful utilities and visualisations to facilitate the developers' tasks. Figure 1.1 shows the research scope of this thesis: the research is in the field of ubiquitous computing that is a part of the human-computer interaction (HCI) research. A specialised research area of ubiquitous computing is the development of physical and tangible user interfaces. The thesis addresses especially the rapid prototyping of these physical interfaces by means of development toolkits.

1.3 Thesis Contributions

In summary, the thesis provides the following three major contributions:

1. Development of a *runtime platform* that provides access to distributed sensor and actuator hardware. By introducing hardware model abstractions in a shared data space, the platform allows the access from distributed client machines to the shared hardware devices. The platform handles all connected hardware devices autonomously, allows the easy integration of custom hardware, and provides a comprehensive set of already supported hardware building blocks.
2. Providing a *developer library* that allows programmers to rapidly prototype distributed physical user interfaces of information appliances. This developer library hides the complexity of the hardware and network access. This enables developers with no previous experience of programming physical hardware the implementation of their envisioned interface ideas. Furthermore, the toolkit extensibility and advanced development strategies address the needs of experienced developers.
3. To facilitate the debugging of developed information appliances, the toolkit includes *development utilities to monitor, control, and simulate* the distributed hardware devices as well as the built information appliances of the distributed infrastructure. These utilities allow the access to the diverse abstraction levels of the runtime architecture. Furthermore, they provide means to get insights into the internal status and events of the distributed infrastructure.

1.4 Thesis Overview

The thesis consists of the following chapters:

Chapter 2

An overview of the research field that forms the framework for this thesis is presented in Chapter 2. It introduces the vision of ubiquitous computing, the early prototypes of appliances, as well as the concepts of physical and tangible user interfaces. The review of system prototypes in the related work and the overview of prototyping techniques conclude the chapter.

Chapter 3

Chapter 3 introduces the requirements for a toolkit that allows the rapid prototyping of distributed physical user interfaces. Subsequently, the related work of existing prototyping toolkits is discussed.

Chapter 4

This chapter introduces the concepts of the Shared Phidgets runtime platform and gives an overview of the general toolkit architecture, as illustrated with the lower four layers of Figure 1.2. It explains the distributed architecture and the abstract hardware device representation in the shared data space. Finally, the details of the platform implementation are explained.

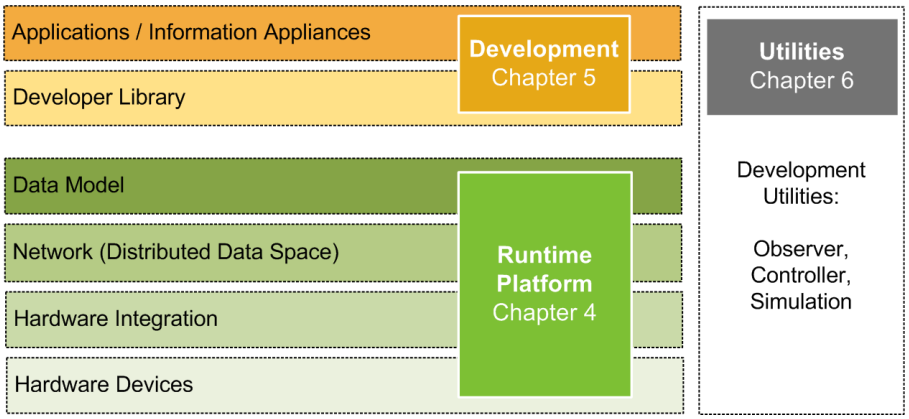


Figure 1.2: Structure of the Shared Phidgets toolkit.

Chapter 5

This chapter explains the details of the development support of the Shared Phidgets toolkit, as illustrated with the upper two layers of Figure 1.2. This includes details of the developer library, integration into the development environment, and advanced programming concepts. At the end of the chapter, the details of the developer library implementation and extensibility are described.

Chapter 6

To support developers with the debugging of the distributed information appliances, this chapter introduces a set of advanced development utilities. These tools access the various abstraction levels of the toolkit, as illustrated in the right side of Figure 1.2. This includes utilities to monitor and control the shared data model, the distributed hardware devices, and the built appliances.

Chapter 7

The case studies in Chapter 7 demonstrate the applicability of the toolkit. The various information appliances illustrate the range of possible systems that can be built with the toolkit. The remainder of the chapter discusses the created prototypes and the overall toolkit architecture.

Chapter 8

This chapter describes possible areas for future work to continue the research of this thesis. An overview of the contributions and the summary conclude the thesis.

1.5 Conventions

Typesetting: A `monospaced` typeface is used for names of classes, interfaces, objects, as well as source code in general. The *italic* typesetting is used for highlighting (e. g. the name of a system that occurs the first time, or important parts of a sentence). The Palatino serif typeface is used for the thesis text, whereas the Arial sans-serif typeface is used for the text in the figures.

Spelling and Citations: This thesis is written in British English. The *ACM Journals/Transactions* citation style[4] is used for all references in this thesis. References for citations, chapters, sections, and subsections are links to the according referenced page or bibliography entry in the thesis document.

Software: The developed toolkit software, source code files, and additional documents of this thesis project are included on the enclosed CD (also available for download at the Shared Phidgets toolkit website [Marquardt, 2008]). Appendix D provides an overview of the included documents and development projects.

4 Instructions and downloads of the ACM Journals/Transactions citation style are available at `http://www.acm.org/pubs/submissions/latex_style/index.htm`

CHAPTER 2

Background and Foundations

In this chapter the background and foundations of the thesis research project are introduced. This includes an overview of the early visions of ubiquitous and tangible computing. These visions describe information systems that move off the desktop, so that our interaction with information technology can take place in our everyday environment. The chapter also introduces the concepts behind embodied interaction and context awareness, and describes the principles of information appliances. Subsequently, various prototype systems are described that were built by researchers as implementations of the ubiquitous and tangible computing visions. A summary highlights the characteristics of these prototypes. The review of prototyping techniques concludes the chapter.

2.1 Ubiquitous Computing

The early visions of the embedded and ubiquitously available computing technology were first introduced by Mark Weiser while working at Xerox PARC [Weiser, 1991]. He used the term *ubiquitous computing* (UbiComp) to describe a new generation of computing technology. Weiser envisioned a variety of networked computing devices, accessible and embedded everywhere in our environment that can support users with computing technology just at the right places. The computing technology should assist users with their everyday tasks and activities, while it should be unobtrusive and invisible at the same time.

Weiser already predicted that in the next years there will be many small and embedded computers used per person [Weiser, 1991]. This would be the consequent evolution of the usage of computing devices: from the time when users had

to share computing power with other users (e. g., the multi-user mainframe systems), to the era of the desktop computer where the user works in front of a single computer, and finally to an environment with many embedded computing devices that are networked and accessible everywhere [Weiser, 1991]. Weiser describes this as *"the age of calm technology, when technology recedes into the background of our lives"* [Weiser, 1996].

An important characteristic of this ubiquitous technology is the fact that it should be invisible, as the *"most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."* [Weiser, 1991]. When Weiser describes the interface to this computing technology as invisible, this does not necessarily mean that the computers themselves are invisible. He rather envisioned an intuitive interaction with the devices, and states that the *"highest ideal is to make a computer so embedded, so fitting, so natural, that you can use it without even thinking about it."* [Weiser, 1991]

Figure 2.1: The tab prototype [Weiser, 1996].

Weiser and his colleagues at Xerox PARC have created various research prototypes to implement and evaluate this concept of embedded and ubiquitous technology. They categorise computers in three different form factors: the small, palm sized *tabs*; the paper sized *pads*; and the larger, wall mounted *boards* [Weiser, 1991]. Weiser predicted for the future that there will be hundreds of these computers spread out in a single room, each of these devices suited for a different task.

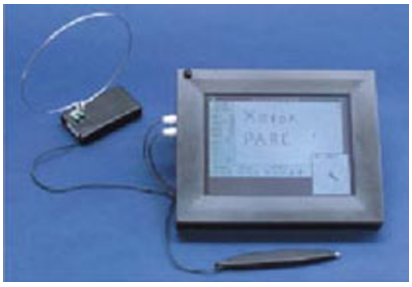


Figure 2.2: The paper sized pad prototype [Weiser, 1996].

Figure 2.1 illustrates the inch-scaled *tabs* that are described by Weiser as *active Post-it notes*. These devices can be utilised as calendars, diaries, reading devices, dictionaries, responsive environment controls, and weather displays [Schilit et al., 1993]. Furthermore, the researchers at Xerox PARC integrated the same technology into the *active badge* [Want et al., 1992]; an electronic name tag with infrared (IR) technology that can locate the position of people in buildings. This allows application scenarios like the automatic opening of doors, the for-

warding of telephone calls, or the adjustment of computer display configurations according to the users' preferences [Weiser, 1991][1].

Weiser describes the paper sized *pads* (illustrated in Figure 2.2) as *"scrap computers"*, whereby he has in mind that they have no individualised identity or importance [Weiser, 1991]. Users can take any of the *pad* computers and for example spread them out on a table and use them like paper. They are used temporarily for a particular task and can be later used by someone else. With this characteristic they differ from the way people normally use notebooks or tablet computers today. Notebook computers are personalised computers of their owners and are carried around with them. The *pads* on the other hand can be grabbed and used anywhere. When users finish their task they leave the *pad* for other users [Weiser, 1991]. The research prototypes built by the XEROX PARC researchers were called the *ScratchPad*, *XPad*, and *MPad*; and they included a X-Window-based software and a drawing surface [Weiser, 1993].

The *boards* shown in Figure 2.3 provide the digital equivalent to blackboards or bulletin boards [Weiser, 1991]. They are large wall screens with the ability to write on the screen. The *boards* are useful for collaborative work of small groups (e. g., viewing documents, presentations, videos). A *board* could also be used to display public information and adapt the content depending on the people in the room (identified with their *active badges* [Want et al., 1992]).

Figure 2.3: The Xerox PARC Liveboard prototype [Weiser, 1996].

Mark Weiser identified three requirements for the success of the technology: *cheap and low-power computers with high quality displays*, *appropriate software* for these computers, and an all *connecting network infrastructure* [Weiser, 1993]. Although the first and third requirement (cheaper and low-power hardware and an ubiquitously available network) already become more and more reality in recent years, the second requirement—the development of adequate software to control ubiquitous computing applications—becomes even more important and is therefore a very active academic research area.

1 Although the *tabs* have similarities with the today's widespread *Personal Digital Assistants* (PDA), they differ in an important characteristic. PDA computers are highly personalised devices for managing the user's contacts, schedule, and emails (and are carried around with them). In contrast, the *tabs* are not intended as personal devices for a specific user, rather than being ubiquitously available tools in the environment to be used by everyone when they are needed for a specific task.

As Weiser points out, the advantage of Ubiquitous Computing does not emerge from many distributed computers alone, but from the interconnection between all of them. *"Like the personal computer, ubiquitous computing will produce nothing fundamentally new, but by making everything faster and easier to do [...] it will transform what is apparently possible"* [Weiser, 1991]. In summary, *"ubiquitous computing enhances computer use by making many computers available throughout the physical environment, while making them effectively invisible to the user"* [Weiser, 1993]. Mark Weiser's work identified visions for a next generation of computing devices and frames the foundations of the work in this thesis.

2.2 Tangible User Interfaces

The work of Hiroshi Ishii and Brygg Ullmer at the Tangible Media Group of the MIT[2] was influenced by Mark Weiser's research of ubiquitous computing; however, they introduce the so called *Tangible Bits* with a different understanding of the user's interaction with digital information. They try to *"bridge the gap between cyberspace and the physical environment by making digital information (bits) tangible"* [Ishii and Ullmer, 1997]. As illustrated in Figure 2.4(a), their goal is to turn the environment into the interface, so that the boundaries between humans and the digital world dissolve; in contrast to the interaction with the desktop computer.

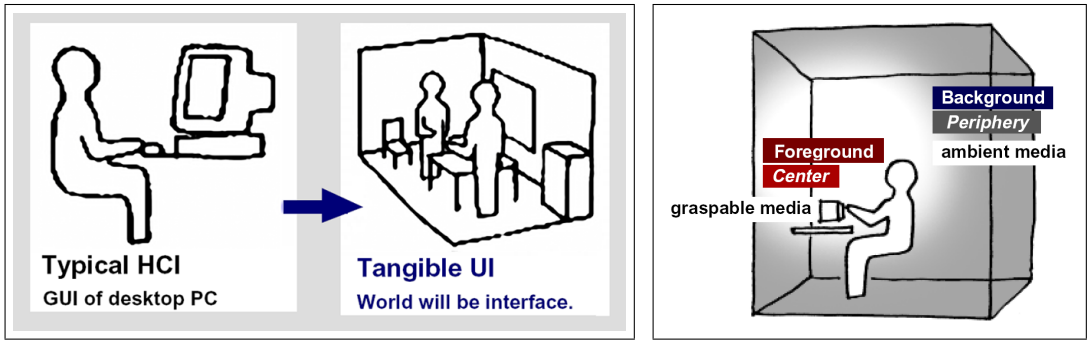
Paul Dourish summarises this concept as follows:

> *"The intuition behind tangible computing is that, because we have highly developed skills for physical interaction with objects in the world - skills of exploring, sensing, assessing, manipulating, and navigating - we can make interaction easier by building interfaces that exploit these skills."* [Dourish, 2001]

Therefore, the user interaction with tangible interfaces differs from traditional computer interfaces in the following ways: the *input* works through the manipulation of graspable, physical objects (e. g., blocks made out of wood, glass, or metal), and the system can in turn change the physical objects as *output* (e. g., by augmentation or physical manipulation) [Ishii and Ullmer, 1997]. Therewith, the physical artefacts are input and output of the tangible user interface at the same time.

Ishii and Ullmer [1997] further explain their concept of tangible interactions that is illustrated in Figure 2.4(b): the foreground interaction with *graspable media*

2 Massachusetts Institute of Technology



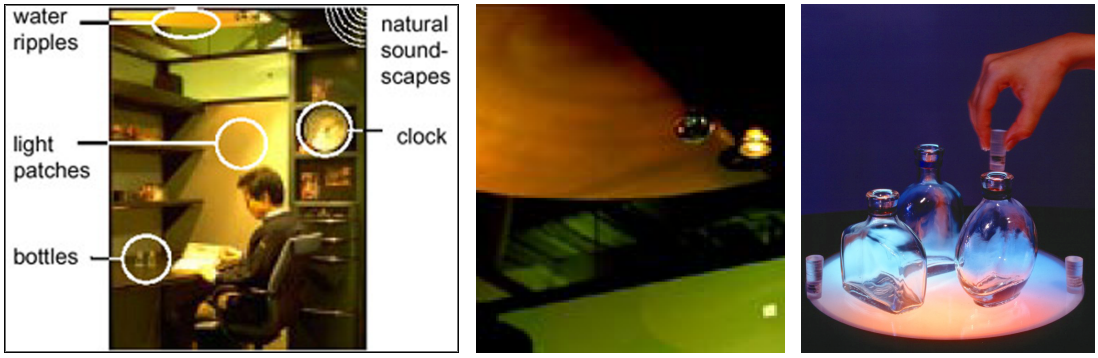
(a) From desktop computing to tangible user interfaces.

(b) Centre and periphery of user's attention.

Figure 2.4: Principles of tangible user interfaces [Ishii and Ullmer, 1997].

(the *coupling of bits and atoms*) and *interactive surfaces* (e. g., wall, desktops, ceilings), and the *ambient media* in the background. The foreground interaction requires users to have the focus of their attention on the current task, whereas the changes in the background are only perceived in the periphery of the user's attention. Buxton [1995] describes a human-centric model that builds on the background and foreground interaction, and how systems should support seamless transitions between these forms of interaction. Buxton explains that these systems can reduce the load on the user and the necessary technological knowledge, if the systems make use of the context and the knowledge of the application domain.

Instances of the foreground interaction are the *metaDESK* [Ullmer and Ishii, 1997], the *transBOARD*, and the *mediaBlocks* [Ullmer and Ishii, 2000]. They have in common that they allow the users to access and work with digital information by manipulating graspable, physical objects. Furthermore, the *ambientROOM* project illustrated in Figure 2.5(a) highlights the innovative ideas behind the *ambient media* concept. On the one hand, the system uses light, shadows, projections, sound, air flow, and water movement to change the environment (cf. Figure 2.5(a) and 2.5(b)), and therewith provide event notifications at the periphery of the user's perception. On the other hand, the ambientROOM provides physical artefacts that control an associated part of the digital world. These are for instance bottles, clocks, and books as tangible controls for the manipulation of the *digital bits*. The bottles illustrated in Figure 2.5(c) represent digital information that can be accessed by opening the bottles, and the hands of the clock illustrated in Figure 2.5(a) can be manipulated by the users to access past events. These computationally augmented artefacts in the environment can make the interaction with information technology more intuitive for users.



(a) The ambientROOM overview.

(b) Water lamp projections as ambient notification.

(c) Bottles as physical handles to control music and light.

Figure 2.5: The ambientROOM project of the Tangible Media Group [Ishii and Ullmer, 1997].

In short, with the tangible interaction Ishii and Ullmer introduced an influential concept for the representation of digital information in the physical environment and the interaction with these *tangible bits*. This research and the diverse experimental prototypes were inspirations for this thesis research work and the development of the Shared Phidgets toolkit.

2.3 Embodied Interaction

With *embodied interaction* Dourish [2001] introduces a concept that applies knowledge and theories of philosophical and sociological research to the emerging visions of tangible computing. Dourish states that *"embodiment is the property of our engagement with the world that allows us to make it meaningful"* [Dourish, 2001]. People are better in using systems that are meaningful to them and are strongly connected to their social practices.

Dourish explains that *"[e]mbodied interaction is the creation, manipulation, and sharing of meaning through engaged interaction with artifacts"*. This does not only mean physical presence of these artefacts in the environment, but moreover that the *"occasion within a setting and a set of specific circumstances gives it meaning and value"* [Dourish, 2001]. Dourish further describes that *"technology and practice cannot be separated from each other; they are coextensive and will coevolve. Practices develop around technologies, and technologies are adapted and incorporated into practices."* [Dourish, 2001]. Therefore it is crucial to consider and evaluate the ways of how technology is integrated into the everyday routines of people. This is another argument of why prototyping of interactive applications

is so important: it can lead to findings about the integration of the technology into social practices of people that might be different to the expectations of the developers and designers.

Dourish does not define a strict set of guidelines for designers and developers of how to create systems to support embodied interaction:

> *"Embodied interaction is not a technology or a set of rules. It is a perspective on the relationship between people and systems. The question of how it should developed, explored, and instantiated remain open research questions."* [Dourish, 2001]

Nonetheless, a summary of design principles highlights important concepts. For instance, that *"users, not designers, create and communicate meaning"* and *"users, not designers, create coupling"* [Dourish, 2001]; principles that stress the importance of considering that the users integrate the technology into their social practices. Another principle is that the *"embodied technologies participate in the world they represent"* [Dourish, 2001], which is especially important for the tangible user interfaces.

With the developed Shared Phidgets toolkit it is intended to support developers to easily build and evaluate embodied technology that is integrated into the social practices of people. Developers should be able to focus on finding the right forms of embodied interaction through the physical user interface, rather than the technological difficulties of the development.

An important aspect of the embodied technology is the system's awareness of the surrounding context in a way that the system can adapt and react to changes of the context. Therefore, the next section introduces the concept of context awareness.

2.4 Context Awareness

Context awareness is another important concept of interactive systems, and describes their ability of being able to sense the context in the environment of the user and the system. This context *"typically includes the location, identity, activity and state of people, groups and objects"* [Salber et al., 1999]. Therefore, these systems can use this information and dynamically respond to the changes of this context. A system that reacts and adapts to this context information can interpret this information to execute services. This enables the system to provide information that is useful or important to the user in the current context. The system

could also store the context information for later data retrieval [Salber et al., 1999].

An example of a context aware application is the automated phone call forwarding of the ActiveBadge system [Want et al., 1992] mentioned earlier. Based on the identity of the caller and the current location of the receiver of the phone call, the system can use this information to forward or reject the call. Another example of a context-aware system is an electronic city guide device that knows the current location of the user and can therefore provide information about the nearby landmarks and restaurants.

In summary, context aware systems are able to react to changes in the environment in meaningful ways. To implement this context awareness, physical sensors can be integrated, so that the system can derive context related information from these sensor values. To support the development of context aware applications, the Shared Phidgets toolkit should facilitate the integration of (and access to) physical sensors and actuators.

2.5 Information Appliances

Donald Norman describes the concept of *information appliances* in his book *The Invisible Computer* [Norman, 1999]. The key characteristic of information appliances is that they are specialised for a certain task, and that they are interconnected based on an underlying network infrastructure. He follows Mark Weiser's vision [Weiser, 1991] insofar as an *"ideal system buries the technology that the user is not even aware of its presence."* [Norman, 1999]. The specialisation of the functionality of the device has the advantage that *"learning how to use [the device] is indistinguishable from learning the task"* [Norman, 1999]. Norman argues that *"difficult tasks always have to be taught. The trick is to ensure that the new technology is not part of the difficulty"* [Norman, 1999]. Information appliances, however, lack many of the advantages of the desktop computer, because of their specialised design. This is for example the high flexibility and possible recombinations of software on the desktop computer that allows the uses in initially unintended ways.

Norman [1999] stresses the fact, like Weiser previously [Weiser, 1993], that an important characteristic of the ubiquitously available technology is their interconnection with an underlying network infrastructure. The success emerges from the combination and fluent interaction between appliances. Nonetheless, not all appliances necessarily have to work together. In fact, it is important to find categories of appliances that benefit from working interconnected (e. g., sharing dig-

ital artefacts like photos and videos); whereas others work separated from each other (e. g., appliances that access highly personal data, like bank accounts). This exploration of novel assemblies and connections between appliances is one of the application areas of the Shared Phidgets toolkit.

The concept of information appliances—specialised for specific tasks and interconnected via a network—describes key characteristics of the physical user interfaces that developers can prototype with the proposed toolkit. The Shared Phidgets toolkit helps developers to experiment with appliances, change their interaction methods, focus on the device design, implement connectivity with other appliances, and evaluate the users' interaction with the systems. In the next section of this chapter various example prototypes of such information appliances are described in detail.

2.6 Applications and Prototype Systems

Based on the mentioned visions of ubiquitous and tangible computing, researchers have developed various prototypes of systems that explore this research field. Therefore, the following subsections give an overview of selected prototypes that have been developed in the last years. These diverse prototypes help to identify the requirements for the toolkit, and to comprise the feasible application areas of prototyping toolkits. The examples fall into the following categories: *Facilitating Communication*, *Providing Awareness with Ambient Displays*, *Technology in the Domestic Space*, and *Tangible Digital Information and Media*.

2.6.1 Facilitate Communication

An important research area is the exploration of how to support people to communicate over distance with their family, friends, and colleagues. Based on this research several prototype systems have been developed that are now briefly described.

The *HomeNote* system [Sellen et al., 2006] introduces a situated display in the home to support family communication. The system supports a different approach for communication, as people can send messages to a *place*, rather than to a particular person. Messages can be sent to the HomeNote display from mobile phones, or scribbled on the write-sensitive screen, and the messages than appear on the display (with additional information as the phone number or an image of the sender of the message). Sellen et al. [2006] have investigated the value of

placing such displays at specific locations in the home (e. g., the mantelpiece, the hallway). The results of an evaluation of the HomeNote system show the various types of communication that are supported with a situated display. For instance, with *calls for action* family members have used the system to call more than one person to take some action. The system also supports *awareness and reassurance* of actions and activities (e. g., children tell where they are), as well as sending *reminders* and *passing on messages*. This research points out the importance of evaluating the prototypes of the situated technology in the home, and how the utilisation of the device can change depending on the location.

The *ActiveHydra* is an example of a context-aware media space that can facilitate communication between distant collaborators [Greenberg and Kuzuoka, 2001]. The system includes a video screen and microphones to allow communication between remotely located co-workers, as well as various sensors so that the system can derive the current context. For instance, the system is able to switch between an awareness mode (slow video update, no audio) and a high-quality communication (video and audio) if users draw near to the ActiveHydra. Therefore, the integrated sensors let the communication appliance react to changes in the environment (*context awareness*).

With the *LumiTouch* prototype Chang et al. [2001] investigate how to bridge the distance between remote located family members or relatives. Their system consists of two digitally enhanced picture frames; each equipped with a set of touch sensitive sensors and lights around the frame. With these sensors the system is aware of people holding, touching, and moving the picture frame. These activities are shown on the second (remote located) picture frame by illuminations and sounds. Chang et al. [2001] were especially interested in ways to provide awareness of the activities of distant relatives, and how technology can support the people to feel that they are closely connected. Therefore, the LumiTouch picture frames provide an informal and light-weight kind of communication.

The *digital family portrait* [Mynatt et al., 2001] and *CareNet* [Consolvo et al., 2004] are systems that provide awareness information between senior adults and their extended family members. This awareness system can support elderly adults to stay independently at home, and gives their caregivers (e. g., their children) information about their health and activities. With the digital family portrait, it is possible to choose between different kinds of information to display in the border area of the picture frame: health, activity, special events, and relationship. These information are visualised as icons around the digital photo, where the size of the icons represent a certain value or percentage. The CareNet system shown in Figure 2.6(a) provides more in-depth information once a user touches the photo (e. g., information of the following categories: activities, meals, mood, medications) [Consolvo et al., 2004]. The challenges are in capturing this information,

in the methods to abstract the values to preserve privacy on both sides, and finding out how to present this information on the displays (maybe with abstract representations, as for instance ambient lights). Although Mynatt et al. [2001] and Consolvo et al. [2004] did mention the value of integrating sensors to derive the related information (e. g., activity, taken medication) to display on the frames, the systems so far are not integrating any physical sensors. This integration of sensors will raise further questions of how to abstract the sensed values and preserve privacy. The developed Shared Phidgets toolkit facilitates exactly this integration of physical sensors into the prototypes, and therewith allows the researchers to focus on the evaluation and modification of the prototypes.

2.6.2 Providing Awareness with Ambient Displays

Another category of prototypes are the ambient displays and systems to provide awareness. The most important characterisation of ambient displays is the ability to provide awareness of the occurrence of events (e. g., reminders, notifications, presence of people) in the *periphery of the user's attention*. Intentionally, ambient displays should be embedded and integrated in the user's everyday environment, so that they can give useful information without being intrusive.

The *Live Wire* [Weiser and Brown, 1996] installation at Xerox PARC shown in Figure 2.6(b) is one of the early prototypes of an ambient display. It gives the network traffic of the Xerox PARC network a *physical presence*. To do this, the installation is using a motor to control a dangling string hanging from the ceiling. The rotations of the motor depend on the current measured network traffic in the research lab. With it, the Live Wire gives people that are walking by an awareness of the current (usually invisible) network activity of the research lab.

In the *TOWER* environment [Prinz and Gross, 2001], ambient displays are utilised to provide awareness of activities in a collaborative workspace. These are for instance large screens showing recent workspace changes, or a water-lamp illustrated in Figure 2.6(c) to indicate web page or web-cam access to the visitors in a coffee room. Prinz and Gross [2001] conclude, that the augmentation of groupware systems with ambient displays can *"increase the awareness and cohesion in distributed teams"*, and therefore *"support[s] communication and cooperation between distributed teams in an adequate manner compared to co-located teams"* [Prinz and Gross, 2001].

Besides providing visual awareness information by using displays, lights, etc., Mynatt et al. [1998] have explored if soundscapes with background auditory cues can provide serendipitous information in the workplace. By hearing music, speech, and sound effects on the edge of background awareness, this audio

feedback can give users information about their co-workers presence and physical actions in the workplace. In their developed *AudioAura* system Mynatt et al. [1998] use the active badge system [Want et al., 1992] to retrieve presence information of the users, and sensors on the software side (e. g., calendar and email). The implementation of the system requires the in-depth exploration of sensor configurations and their placement, as well as the timing between sensors and audio signals.

(a) The CareNet system [Consolvo et al., 2004].

(b) Livewire installation at Xerox PARC [Weiser, 1996].

(c) Water lamp of the TOWER environment [Prinz and Gross, 2001].

Figure 2.6: Ambient displays and remote awareness systems.

The visualisation projections of the *ambientROOM* [Ishii and Ullmer, 1997] are also examples of ambient displays to provide awareness information. Initially, Ishii and Ullmer [1997] have used the sound of rain to give a user ambient feedback of the traffic on a website (with a volume controlled playback of the rain sound effects). They found out, however, that these sound effects distract users from their work (as the sound is too intrusive), and therefore they decided to evaluate an alternative design: the water ripple projection, as illustrated in Figure 2.5(b). An actuator element connected to a water tank can cause ripples on the water (in this case, dependant on the web traffic). A light source is shining through the water tank and is projecting the water surface onto the ceiling of the ambientROOM. The result is a very calm and unintrusive ambient visualisation of the awareness information in the periphery of the user's attention.

The *location-dependant information appliances* [Elliot et al., 2007] enhance the concept of ambient displays in two aspects. First, the information appliances are aware of their location, and therefore appliance can provide diverse awareness information depending on the location. Second, the appliances let the users easily move between background awareness information and foreground interaction with detailed information [Buxton, 1995]. Users can easily change the information source that is displayed on the ambient display (e. g., weather, traffic, messages), and associate this information to an information appliance and/or a

particular location. This allows end users to benefit from ambient displays at the locations that make sense to them.

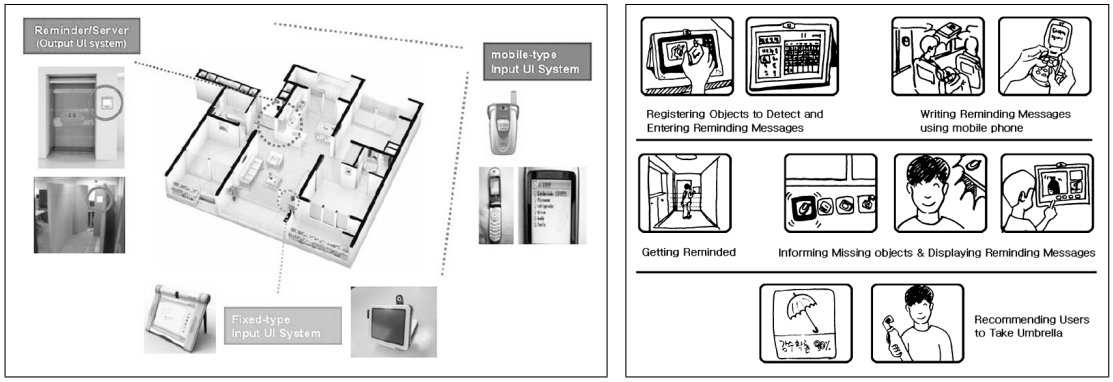
2.6.3 Technology in the Domestic Space

Researchers have investigated the advantages and tradeoffs of living in homes with embedded information technology, which is integrated to support people's everyday practices. Many issues arise with the usage of computing technology in homes, mainly because of the differences between the social practices at home compared to the office (which is the traditional application domain for information technology) [Elliot et al., 2007]. With the following research projects, these differences and the usefulness of applications in the domestic space have been explored.

The *Aware Home* [Kidd et al., 1999] is a *living laboratory* for ubiquitous computing research of the Georgia Institute of Technology. The intention of the research is to build a house that is aware of its occupants' activities, and supports the everyday activities of the family members. By integrating distributed sensors, displays, and actuators in the home, Kidd et al. [1999] explored various application scenarios. For instance, a *finding lost objects* service helps users to find lost wallets or keys in the home. They also implemented and evaluated identification techniques (e. g., the Smart Floor), digital augmentation of objects, monitoring and security applications, and care facilities (e. g., the previously mentioned *digital family portraits* [Mynatt et al., 2001]).

The *Gator Tech Smart House* [Helal et al., 2005] also evaluated the support of embedded computing technology in the home. The applications range from a *smart mailbox* (senses mail arrival and notifies occupants), *smart blinds* and *thermostats* (can automatically adjust to personal light and temperature preferences), to *home security monitors* (monitors windows, doors). In general, they explored and evaluated the utilisation of sensors and actuators in the domestic space, with considering technological and sociological constraints.

The *Gate Reminder* [Kim et al., 2004] is another example of how context aware information appliances can be integrated into households to support the everyday routines of the family. The system is designed to remind the family members of forgotten objects (e. g., key, wallet, mobile phone), things to do (e. g., shopping list, books due in the library), or to display other messages and information (e. g., the weather report with recommendations to take the umbrella when it is raining).



(a) Deployment of the displays in the home.

(b) Scenarios of the Gate Reminder system.

Figure 2.7: The Gate Reminder system: situated reminder displays in the home [Kim et al., 2004].

The system includes embedded displays (e. g., at the front door of the home as seen in Figure 2.7(a) on the left side) that can display reminders for the family members when they enter or leave the house. The displays receive messages from the household members (e. g., via a cell phone; Figure 2.7(a) on the right side), and they use RFID tags as well as video cameras and microphones to recognise the person that is leaving or entering the building. They also use various motion and door sensors to get detailed information about the activity at the house entrance, and RFID tags for the detection of objects [Kim et al., 2004]. With these sensors the Gate Reminder system can derive context information, so that it can provide the useful reminders at the right time (e. g., when leaving) and place (e. g., at the front door).

Figure 2.7(b) illustrates a series of scenarios of the Gate Reminder system. Users can register objects (e. g., books) in the system to get reminded of taking these objects with them when they leave at a particular date. They can also send messages and reminders with their mobile phone. In another scenario, a user sees the weather report on one of the situated displays and takes the umbrella with her when leaving [Kim et al., 2004]. When implementing the Gate Reminder system to evaluate these scenarios, Kim et al. [2004] were facing several difficulties: determine the intentions of a person (e. g., leaving, entering, directions), placement of the sensors and displays, and the optimal timing of the reminders. In this case the utilisation of a rapid prototyping toolkit would be helpful, as it can support the evaluation of different implementations, and facilitate the testing of design alternatives. The developed Shared Phidgets toolkit facilitates the development with distributed sensors and displays as they are used in the Gate Reminder system.

Elliot et al. [2007] summarise the importance of using *contextual locations* inside homes, and how these locations can support and enhance the information management of the family members. For instance, messages (e. g., sticky notes) of the family members are placed at particular locations within the home (e. g., on the fridge in the kitchen, at the front door, on the television screen) to act as *memory triggers*, provide *awareness*, or help to *coordinate resources* [Elliot et al., 2007]. In many cases, the *location* of these messages provides important context information in itself. For instance, when messages are located at the inside of the front door, they might require a particular action when leaving the house. The implemented *StickySpots* system [Elliot et al., 2007] supports this type of location-based messaging by placing displays at various meaningful locations inside the home, and allowing the family members to send handwritten notes to a particular display. With this prototype it is possible to evaluate the integration of the system into the routines and practices of the families.

2.6.4 Tangible Digital Information and Media

This category covers two example prototypes that make digital information physical and graspable. These systems directly address Ishii's vision of the *Tangible Bits* [Ishii and Ullmer, 1997].

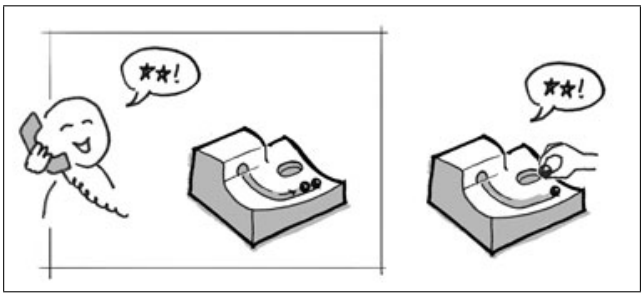


Figure 2.8: The marble answering machine by Durrell Bishop.

One example of a tangible user interface is the *marble answering machine* by Durrell Bishop [Crampton Smith, 1995]. Unlike conventional phone answering machines, this one does not overwhelm users with any buttons or displays. Instead, the user interacts *physically* with the device: for every new message that a caller leaves on the machine, the system drops a marble into a small container on the front, as illustrated in Figure 2.8 [3]. When the user returns and wants to hear one of the recorded messages, he/she can simply grab one of the marbles and put

3 Source of the illustration: http://www.interakt.nu/home/images/MarbleAMschematic.jpg (Last access: October 15th, 2007)

it into an indentation of the machine to playback the recorded message. If the marble is placed near an augmented phone, the number of the caller is dialled automatically. Thus, the marble answering machine is an excellent example of how to give digital information a physical presence, and how to interact with this tangible interface.

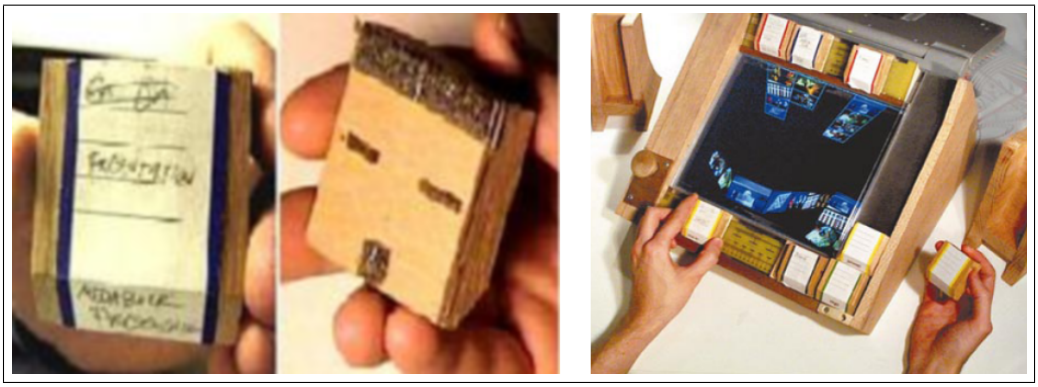


Figure 2.9: Tangible digital information with mediaBlocks [Ullmer and Ishii, 2000].

The *mediaBlocks* system [Ullmer and Ishii, 2000] is another example of how to connect digital information with a physical representation. The mediaBlocks are small wooden digitally tagged blocks (shown in the left side of Figure 2.9) associated with digital media; for instance videos, photos, or presentations. Users can interact with these physical tokens (*phicons*) in the following ways. First, they can use the mediaBlocks to easily transfer media between devices. Second, by arranging the blocks in a *sequence rack* as illustrated in the right side of Figure 2.9 users can modify and control the digital content (e. g., sequences of video clips). The mediaBlocks therefore explore possible techniques to work with digital media in the physical world.

2.6.5 Summary of Prototype Characteristics

In summary, these introduced prototype systems are example implementations of ubiquitous computing systems with physical or tangible user interfaces that move off the desktop and into the user's everyday environment. A few of these systems are built as local information appliances, but most of them are integrating various remotely located displays, sensors, and actuators. The number of utilised hardware components ranges from single sensors or actuators (for instance in the *Live Wire* or *Digital Family Portrait* projects) to systems that integrate larger collections of hardware devices (for instance the *Gate Reminder* or *ambientROOM* projects).

The researchers that have built these systems had to deal with low level implementation issues that occur when building these distributed prototypes. This makes the systems difficult to build, and even more important: it can hinder the exploration of alternative designs and prototype variations. Therefore the main objective of this thesis is to provide developers a toolkit that allows the rapid prototyping of similar distributed physical user interfaces like these of the four categories outlined in this section of the chapter. Before the requirements of the toolkit are introduced in Chapter 3, the prototyping strategies for the development of interactive appliances are described in the next section.

2.7 Prototyping Techniques

When designing interactive systems as those described in the previous section of the chapter, developers and designers can choose between diverse prototyping techniques. Two general categories of these techniques are the low-fidelity (e. g., paper-based) and the high-fidelity prototypes (e. g., software development, GUI builders) [Preece et al., 2002; Rudd et al., 1996].

Low-fidelity prototypes "tend to be simple, cheap, and quick to produce" [Preece et al., 2002], and therefore they are the ideal means to explore alternative designs at the early development stages of a project. Various techniques are available to create low-fidelity prototypes. First, with *sketches and drawings* the developers can illustrate important aspects of the system [Buxton, 2007; Preece et al., 2002]. Second, *Storyboards* are a series of drawings that illustrate use cases or scenarios of the system (e. g., how a user interacts with the envisioned physical appliance) [Preece et al., 2002]. Therefore, with storyboards it is possible to capture the dynamics of the interaction process. Third, *Wizard of Oz prototyping* describes a technique where the developer is simulating the response of the system. When users are interacting with the prototype they get the illusion of a working system, although the system is not yet implemented [Li et al., 2007; Preece et al., 2002]. Finally, with *videos and animations* it is possible to illustrate the interaction and usage of a system; however, this technique is the most time consuming of the four low-fidelity prototyping strategies [Buxton, 2007].

High-level prototypes on the other hand are already working implementations of the systems. To practically implement a high-fidelity prototype, it's functionality has to be reduced compared to the full working system. The two strategies to achieve this are *horizontal* and *vertical* prototypes. On the one hand, the horizontal prototypes implement the features of the full system but limit the implementation details of these features. On the other hand, a vertical prototype offers only

a limited set of the features; however, these features are then implemented in a higher level of detail than with the horizontal prototype [Nielsen, 1993].

High-fidelity prototypes are useful for the testing of the user's interactions with the system [Rudd et al., 1996]. They help to identify problems with the current prototype design and support the developer with decisions between implementation alternatives. This is becoming even more important when the development does not only include software (e. g., websites, user software), but hardware components as well (e. g., information appliances). As Liu and Khooshabeh [2003] pointed out, there can be design weaknesses of ubiquitous computing systems that were only revealed by the evaluation of interactive high-fidelity prototypes. Therefore, the high-fidelity prototypes are a good complement to the design sketches and storyboards of the early design phase.

The advantages and disadvantages of high- and low-fidelity prototyping are summarised in Table 2.1 [Rudd et al., 1996]. The most significant disadvantage of the high-fidelity solution is the *time-consuming* and *expensive development*. Another drawback of the high-fidelity prototyping is the tendency of users to assume that the prototype is already the *final system*. Developers are maybe considering *fewer alternatives* because they have already developed a working prototype that users like [Preece et al., 2002]. Therefore, a prototyping development system should facilitate the exploration of design alternatives for developers during the design process and allow the easy and flexible exchange of all used hardware interface elements and components [Abowd, 1999]. The Shared Phidgets toolkit minimises the drawbacks of the development of high-level prototypes (lower right section of Table 2.1). The toolkit makes the prototype programming as easy as possible, to minimise the necessary time developers have to spend on the programming of the prototype, and to encourage the exploration of design alternatives.

In summary, the following issues have to be considered when developing a high-fidelity programming prototyping toolkit to minimise the limitations of the high-fidelity prototype development:

- Minimise the effort required to build high-level prototypes [Rudd et al., 1996].
- Support the exploration of implementation alternatives [Preece et al., 2002; Abowd, 1999].
- Allow the flexible substitution of all the used hardware interface components [Abowd, 1999].

Rapid prototyping is an inherent part of the development and design process of interactive systems. Both—low- and high-fidelity prototyping—have certain advantages and are particularly suitable at different stages of the design process.

Type	Advantages	Disadvantages
Low-Fidelity Prototype	◦ Lower development cost. ◦ Evaluate multiple design concepts. ◦ Useful communication device. ◦ Address screen layout issues. ◦ Useful for identifying market requirements. ◦ Proof-of-concept.	◦ Limited error checking. ◦ Poor detailed specification to code to. ◦ Facilitator-driven. ◦ Limited utility after requirements established. ◦ Limited usefulness for usability tests. ◦ Navigational and flow limitations.
High-Fidelity Prototype	◦ Complete functionality. ◦ Fully interactive. ◦ User-driven. ◦ Clearly defines navigational scheme. ◦ Use for exploration and test. ◦ Look and feel of final product. ◦ Serves as a living specification. ◦ Marketing and sales tool.	◦ More expensive to develop. ◦ Time-consuming to create. ◦ Inefficient for proof-of-concept designs. ◦ Not effective for requirements gathering.

Table 2.1: Relative effectiveness of low- vs. high-fidelity prototypes [Rudd et al., 1996].

The studies and discussions identified the high value of low-fidelity prototyping for the early design stages, as well as the high-fidelity prototypes for building systems for evaluations with users [Rudd et al., 1996; Preece et al., 2002]. The drawbacks of high-fidelity prototyping strategies are considered in the design of the Shared Phidgets toolkit.

2.8 Chapter Summary

This chapter introduced the background and foundations of the thesis research. The research foundations were introduced with Weiser's vision of ubiquitous computing as the calm technology in the everyday environment, as well as Ishii's work on coupling the digital with the physical world. Embodied interaction stresses the importance to build appliances that fit to the social practices of people, and context awareness introduced the concept of systems that react dynamically to changes in the environment. Information appliances summarise the concept of situated physical user interfaces that are built to support a specific task. The

summary of previous research prototypes has highlighted the complexity of the development, and the common characteristics of these systems. Finally, the comparison of low- and high-fidelity prototyping techniques highlighted important aspects to consider when designing the toolkit.

The next chapter focuses on the introduction of the requirements for a toolkit that facilitates the rapid prototyping of distributed physical user interfaces. This also includes the review and discussion of related work on the development of prototyping toolkits.

CHAPTER 3

Requirements and Toolkit Research

In the previous chapter, after outlining the research area of ubiquitous and tangible computing, the need for a toolkit that supports the distributed development of information appliances was motivated. In this chapter, the requirements for such a development toolkit are analysed. Thereafter, existing toolkits for the rapid prototyping of physical and tangible user interfaces are reviewed.

3.1 Important Toolkit Strategies

To achieve a better understanding of necessary requirements for the Shared Phidgets toolkit, this section takes a closer look to the successful strategies for toolkit development. Toolkits for graphical user interfaces (GUI) have a long history, and have been proven successful of facilitating the development of the applications GUI [Myers et al., 2000; Myers, 1986; Salber et al., 1999]. These toolkits have dramatically simplified the tasks required to build the application GUI by introducing libraries of *widgets*. Widgets are reusable, encapsulated building blocks that implement a single graphical interface element for user interaction (e. g., buttons, pull-down menus, and checkboxes).

These toolkits have been successful because they significantly reduce the necessary time developers have to spent when building new graphical user interfaces. They facilitate the building of rapid prototypes and allow the iteration of the design process to improve the quality of these prototypes [Nielsen, 1993; Myers et al., 2000]. Salber et al. [1999] have summarised the main advantages of these toolkits and the provided widgets:

1. The widgets *hide the details of the access to the hardware input devices*, and can provide abstract events to the applications. For instance, there is no difference for an application if users choose the mouse to click a menu item, or if they use keyboard shortcuts.
2. They *encapsulate the details of the interaction*, as they are handling the widget specific details themselves, and applications only have to register for events to receive notifications.
3. They *provide reusable building blocks*. These building blocks can be utilised and combined, to fit the needs of the developed application.

Component systems like JavaBeans [Sun Microsystems, Inc., 2007] or *.NET* [Microsoft Corporation, 2007d] provide libraries of ready-to-use programming objects to the developer. For instance, this could be a library of GUI widgets that can be integrated into graphical UI design software. This tool (for instance the interface builder of Visual Studio [Microsoft Corporation, 2007d]) allows *"people who are not professional programmers to create sophisticated and useful interactive applications"* [Myers et al., 2000].

In their evaluation of software development tools and toolkits, Myers et al. [2000] have identified further key characteristics of toolkits that are important for the success of the tools. They stress the importance of two characteristics for the toolkits: *threshold* and *ceiling*. The *threshold* is an indicator of how difficult it is for beginners learning to use the system or tool. The *ceiling* describes the maximum level of complexity that can be implemented with the system—or simply: *"how much can be done using the system"* [Myers et al., 2000]. They emphasise the importance of a low threshold for toolkits:

> *"The lessons of past tools indicate that in practice high-threshold systems have not been adopted because their intended audience never makes it past the initial threshold to productivity, so tools must provide an easy entry and smooth path to increased power."* [Myers et al., 2000]

Toolkits should provide means to gradually increment the available functionality of the tools for the developer, and therefore adapt to the increasing requirements of expert developers. Myers et al. [2000] explain that *"it remains an important challenge to find ways to achieve the highly desirable outcome of systems with both a low threshold and a high ceiling at the same time"*. Therefore, it is an important design objective of the Shared Phidgets toolkit to minimise the entry barriers for average programmers, but at the same time ensure the efficient and powerful programming for expert developers.

The functionality of the toolkit and the outcome of toolkit functions should be predictable and transparent, which is difficult when the toolkit tries to take too much

control about the application development away from the programmer. Myers et al. [2000] summarise that *"tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers"*. For the toolkit design, it is also important to consider the *path of least resistance*: because tools that are used by developers influence the kind of systems or user interfaces built, the toolkit should try to prevent developers from choosing the wrong implementation decisions. Moreover, for the implementation of a toolkit it is important to encapsulate the most successful development strategies as reusable software components (e. g., the mentioned interface widgets, event-based architectures).

Event-based architectures of toolkits are a very successful concept and are widely used [Myers et al., 2000]. These systems raise events for the actions that occur in the system (e. g., a user typing on a keyboard or using a mouse). Event-based architectures allow the loosely coupling of the software components. Therefore, the event-based architecture is also a key basis of the Shared Phidgets toolkit. The toolkit implements a transparent event architecture to allow the notification of observers if specific events are raised (e. g., sensor changes).

In summary, the previous research of toolkits for the GUI development influences the requirements and design decisions for the Shared Phidgets toolkit. With the consideration of the successful toolkit design strategies it is possible to optimise the architecture of the proposed Shared Phidgets toolkit to fit the needs of average and expert developers.

3.2 Developer-Centred Toolkit Design

For the specification of the toolkit requirements it is important to analyse the envisioned toolkit characteristics in further detail. Roseman [1993] applied rules of the user-centred design [Norman and Draper, 1986] to the development process of the *GroupKit* toolkit [Roseman, 1993]. This set of rules is analogous to the traditional process of user-centred design, with the difference that the users are substituted with the developers that use the toolkit. This thesis follows this strategy to encompass the important requirements for the Shared Phidgets toolkit.

Specify Toolkit Domain:

As mentioned previously, the toolkit domain is the rapid prototyping of distributed physical user interfaces. The systems that are built with the toolkit therefore include various sensors and actuators, distributed across rooms and buildings. The toolkit aims to facilitate the composition of these physical hardware components to new information appliances.

Identify Developers:

The toolkit will be mainly used by developers of academic and industrial research labs. Presumably these developers are familiar with the fundamental object-oriented programming (OOP) concepts, and the development of GUI applications. However, it is not expected that they are familiar with the development of tangible or physical interfaces, or distributed networking applications. Therefore, the toolkit addresses the needs of average developers (*low threshold*) as well as expert developers (*high ceiling*) [Myers et al., 2000].

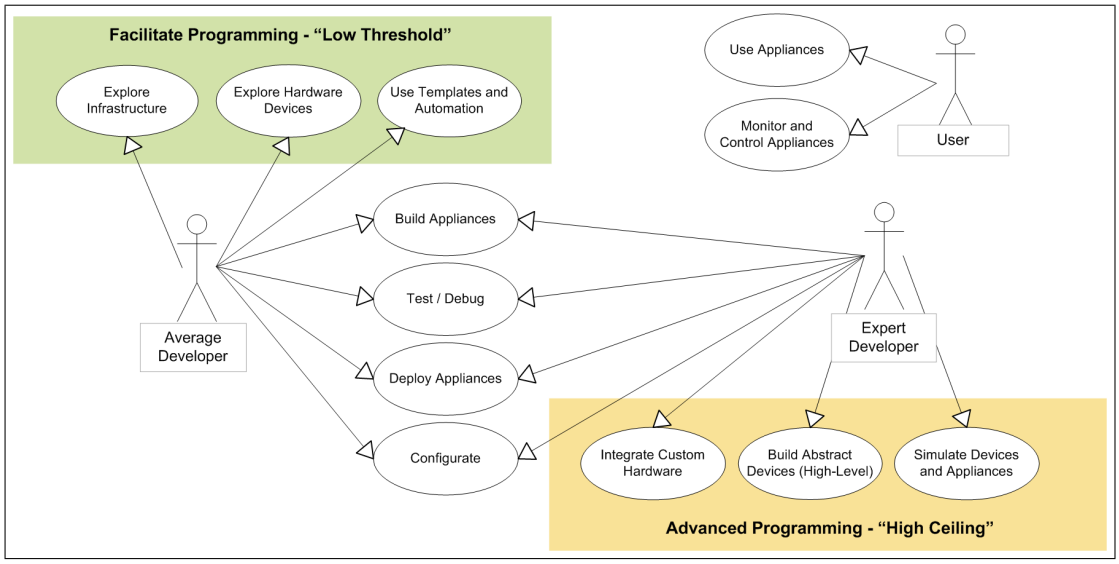


Figure 3.1: UML use case diagram.

Identify Use of Toolkit:

The UML diagram in Figure 3.1 illustrates the important *use cases* that the toolkit should support. *Average developers* with basic programming skills as well as *expert developers* will build, test, debug, deploy, and configure information appliances. Furthermore, the *average developers* need support to learn the distributed programming of physical hardware, to explore the toolkit's capabilities, and to use templates and automation features (*low threshold*, box on the upper left corner of Figure 3.1). On the other hand, *expert developers* need advanced programming support for efficient and powerful programming: integration of custom hardware, building abstract high-level appliances, and using simulations (*high ceiling*, box on the lower right corner of Figure 3.1). Although the developers are the main target group of the toolkit, *users* will implicitly use the toolkit through the built appliances. They might need to reconfigure the settings of the appliances, as illustrated in the upper right corner of the use case diagram in Figure 3.1.

Consider Target Applications:

A few categories of target applications have been described in Section 2.6. For instance, these are information appliances to support communication, handling of digital media, support the remote awareness, provide smart reminders, and control the smart home. The toolkit aims to support development of applications in these areas, but other target areas are imaginable as well. In general, the system should support the development of interactive user interface prototypes that integrate multiple sensor and actuator hardware elements (that might be local and/or remotely located). In contrast, it is important to note that the toolkit is not specifically intended for the development of mobile ubiquitous devices (e. g., sensor equipped mobile phones), large-scale sensor networks (e. g., large-scale temperature measuring networks), or hardware input controllers (e. g., game controllers, 3D controllers).

Design for Proper Use:

Roseman [1993] points out that a "*toolkit provides more than an alphabetic list of routines in a library to the developer*". Therefore, the toolkit should encourage developers to implement systems in a proven efficient and structured way. The toolkit documentation, example programs, development templates, seamless integration into the IDE, and programming tutorials that are provided by the Shared Phidgets toolkitare helping developers to benefit from the toolkit's functionality.

Apply Design Affordances:

This is a transferred interpretation of Donald Norman's term *affordances*[1] [Norman, 1988]. Roseman [1993] herewith describes that the developed toolkit should self-reveal its functionality to the developer. The toolkit's API should be designed in a way that it *suggests* possible uses, and therefore shows *affordances* to the developer. Especially the very close integration into the developer's IDE provides affordances of how to utilise the toolkit. Diverse available programming strategies ensure that developers can choose between multiple options to implement their solutions.

Iterate Design:

Similar to the iterative design cycle that the toolkit explicitly supports when developers build appliances [Greenberg, 2007], the toolkit itself also undergoes the same iterative process during its development. It is crucial to iterate the design of

1 *Affordance* is defined as the properties that *suggest particular uses to users*. An example for *affordance* in Graphical User Interfaces is a button that suggests (with the visual appearance of a 3D button) the user to *push* this button to trigger an action. [Norman, 1988]

the toolkit, and to improve it based on experiences with previous versions. The evaluation of the toolkit, the experience with the appliance implementations, and the feedback from other developers using the toolkit were helpful sources for the iterative design cycle to improve the toolkit's architecture and provided functionality.

3.3 Requirements of the Toolkit

Based on the toolkit's fundamentals presented in this chapter, the developer-centred toolkit design, the previous work of researchers developing interactive prototypes (cf. Section 2.6), and our research group's own experiences of prototyping distributed physical user interfaces, the following list summarises the requirements for the architecture of the Shared Phidgets toolkit, the developer library, and the utilities to support the distributed development:

Runtime Platform and Infrastructure Requirements:

1. **Distributed Hardware Access:**
   Provide a flexible runtime platform that allows the remote access to local and distributed hardware components over the network.
2. **Hidden Implementation Details:**
   Hide the details of the hardware access and network layer implementation, as well as the synchronisation of network events.
3. **Flexibility:**
   Allow the attachment and detachment of hardware components at any time (*plug-and-play*).
4. **Extensibility:**
   Facilitate the integration of custom hardware sensors and actuators with an extensible framework.

Developer Library and Prototyping Requirements:

1. **Event-Based Architecture:**
   Use an event-based notification architecture that allows the registration for any event that occurs at one of the distributed hardware devices.
2. **Encapsulation and Reusability:**
   Provide a library of hardware proxy objects with an easy to use API. These hardware proxy objects should reflect the current hardware status, and include event definitions for the hardware's specific events.

3. **Self Revealing Toolkit Functionality:**
   Provide means to facilitate the exploration of the toolkit's capabilities.
4. **Diverse Programming Strategies:**
   Support developers that have no previous experience with the programming of physical user interfaces (*low threshold*). Furthermore, consider and integrate advanced programming strategies for expert developers (*high ceiling*).
5. **Reconfiguration:**
   Support the easy reconfiguration of the developed information appliances (e. g., changing the hardware components that an appliance is connecting to at runtime).

Development Utilities Requirements:

1. **Monitoring and Debugging:**
   Provide monitoring and debugging utilities for the distributed hardware of the infrastructure. These utilities should provide detailed information at different abstraction levels of the toolkit, and therefore support different stages of the development.
2. **Infrastructure Exploration:**
   Integrate utilities that allow insights into the internal status of developed and deployed information appliances. This includes their current configuration, the addressed hardware, the occurred events, and their distribution across remote locations.
3. **Simulations:**
   Allow the simulation of hardware devices to facilitate the testing and debugging of information appliances.

Before the subsequent chapters explain the concept and implementation details of the Shared Phidgets toolkit to fulfil these requirements, the next section reviews the related work of existing development toolkits for prototyping information appliances. These toolkits are analysed with a focus on their support for the development of distributed physical user interfaces.

3.4 Toolkits for Prototyping Interactive Systems

This section reviews the previous research projects in the area of prototyping toolkits for building physical user interfaces. The important aspects of the systems are described as well as their support for the prototyping of distributed physical

user interfaces. The summary at the end of the section provides an overview of the available functionality of these toolkits.

3.4.1 Phidgets

When developing a networked media-space for collaboration between co-workers (the *Active Hydra* project introduced in Subsection 2.6.1), Greenberg and Kuzuoka [2001] were facing the difficulties of integrating hardware sensors and actuators into their system. The hardware of the prototype was very difficult to build, and it was even more complicated to modify the hardware configuration once the first prototype was built.

The experience with the development of this physical prototype motivated Greenberg and Fitchett [2001] to build a toolkit which facilitates the development of applications that integrate sensor and actuator hardware. The *Phidgets* toolkit [Greenberg and Fitchett, 2001; Phidgets Inc., 2008] provides physical building blocks for developers to prototype these systems. Phidgets include a collection of physical sensors (e. g., movement, light, force, temperature) and actuators (e. g., motors, displays). These hardware components can be connected to the USB port of any computer, and are immediately available as input or output devices. Programmers can address the hardware components on their local computer by using a simple application programming interface (API) that is included with the toolkit. Proxy objects that are added to the source code of the application represent the hardware objects. The developers can easily register for events (e. g., sensor changes) or control properties of the hardware component (e. g., servo movement) [Greenberg and Fitchett, 2001].

The easy integration of the physical building blocks enabled even programmers with moderate programming skills to experiment with physical interfaces. A few examples of the creative prototypes developed by students of an undergraduate HCI course can be seen in Figure 3.2. These examples include *tangible picture frames* (a), *interactive illuminations* (b), *meeting reminders* (c), *ambient lights* (d), *tangible music player* (e), and a digitally *augmented storybook* (f). The easy to use toolkit allowed the students to concentrate on the physical design of the information appliance, as well as the testing of various implementation strategies to implement their envisioned ideas [Greenberg, 2007].

This thesis project is closely related to the previous research of the Phidgets toolkit at the University of Calgary. The developed Shared Phidgets toolkit extends the successful Phidgets architecture in such a way that developers can prototype systems with distributed hardware like they have previously implemented them with the local Phidgets toolkit. As described earlier in Chapter 1, the important dif-

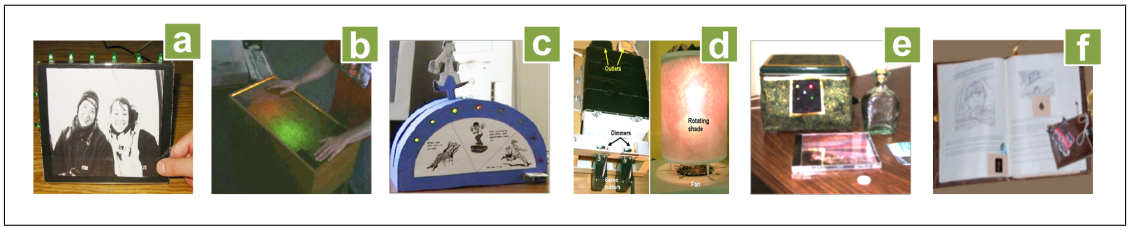


Figure 3.2: Prototyping examples of the Phidgets toolkit [Greenberg, 2005, 2007].

ference is the support of *distributed* hardware, and the support of the challenges that arise with the distributed development. The further advanced development strategies and programming utilities to support this distributed development are explained in the subsequent chapters.

3.4.2 Context Toolkit

The *Context Toolkit* [Dey, 2000] introduced high-level abstraction layers on top of the raw hardware access layer. As identified by Salber et al. [1999] in their paper about the Context Toolkit, the difficulties of building context aware application are as follows [Salber et al., 1999]:

- Applications need access to sensors like GPS devices, or the Active Badge system.
- The raw sensor information must be abstracted to provide high-level context information.
- Systems may be distributed, so that the sensors and the computer systems are at distant locations of a network.
- The systems are dynamic, and therefore the application must adapt to these changes.

The Context Toolkit addresses these difficulties with the introduction of the *Context Widgets*. These widgets hide the access to the underlying hardware. They provide abstract events to the application, and they build a library of reusable and customisable building blocks [Salber et al., 1999]. The Context Widgets are also instantiated in a distributed architecture, and are running permanently to monitor the environment at any time. Salber et al. [1999] have built a collection of widgets, for instance the *IdentityPresence* widget that informs the application of the arrival or leaving of people. The *Activity* widget can observe a location and sends events to the application if the activity level at this location changes.

With this architecture of Context Widgets, Salber et al. [1999] have built context aware applications to demonstrate the applicability of the toolkit. For instance, the *In/Out Board* informs a group of people in the office of the presence of their colleagues. Therefore, this application can be seen as a visualisation of the IdentityPresence widget. The underlying technology of detecting the presence can easily be exchanged as long as the IdentityPresence widget preserves its API.

The architecture of the Context Toolkit significantly influences the development of the Shared Phidgets toolkit. The encapsulated widgets for the hardware access are an important concept for hiding the implementation details. At the same time, the aggregation and interpretation of raw sensor values is important to provide abstract high-level events. The programming objects of the Shared Phidgets toolkit hide the low-level hardware access in a similar way. They also facilitate the building of appliances that provide similar functionality as Context Widgets by supporting high-level events. Nonetheless, the Context Toolkit builds upon a collection of high-level widgets and allows their composition and assembly. The Shared Phidgets toolkit, however, is in between the high-level Context Toolkit and toolkits for building local sensor-based applications. It explicitly supports programming concepts to facilitate the assembly of distributed sensors and actuators. Therefore the Shared Phidgets toolkit includes the access to the events and properties of all sensors and actuators, and it provides a library of programming building blocks to build appliances that can implement similar functionality as the Context Widgets.

3.4.3 Peripheral Displays Toolkit

Matthews et al. [2004] introduced with the *Peripheral Displays Toolkit* (PTK) a system that supports the development of peripheral and ambient displays. The toolkit is based on the Context Toolkit, and it provides abstraction layers and programming support to build these ambient notification systems based on physical sensors and actuators (similar to the systems introduced in Subsection 2.6.2). It facilitates the development by *abstracting* raw sensor values to high-level interpretations (the *notification levels*), and using *transitions* events to change actuators.

The *abstractions* describe the interpretation of raw sensor values to abstract representations (e. g., recognizing phone rings in an audio stream). *Notifications* are the events that are generated based on the changes in the *abstracted* interpretations of the sensor values. They address different awareness and attention levels (e. g., ambient, alerting), and they are based on the changes in the *abstracted* events (e. g., exact matches, delta changes). Finally, the *transitions* describe the mapping of these *notifications* to the changes of actuators (e. g., motors, lights).

The *transitions* generate a series of events for the actuator that lead to fluent transitions between the previous and the current status of the actuator element.

Although the introduced abstractions facilitate the development of peripheral displays, they are only partial applicable in a generic toolkit for physical user interfaces. The abstractions concept is, however, important for the Shared Phidgets prototyping toolkit as it represents the interpretation of raw sensor values to high-level events. The *notification levels* and *transitions*, however, are a specific concept related to the development of peripheral displays, as they define the attention and awareness levels that map the occurred events to the notifications on displays. Therefore, the PTK introduces a higher abstraction level with the programming objects it provides that limits the scope of the developed applications. Furthermore, the PTK *"does not have any support for managing or learning about the status and connections between various inputs, outputs, and other components"* [Matthews, 2005]. The Shared Phidgets toolkit explicitly supports these tasks with a collection of advanced development utilities.

3.4.4 Papier-Mâché

Klemmer et al. [2004] have built a toolkit that facilitates the development of tangible user interfaces. The *Papier-Mâché* toolkit does not only allow to use physical objects with associated electronic tags (e. g., RFID) for creating tangible interfaces, but uses computer vision so that developers can integrate any physical object (e. g., scissors, pencils) into the system. The toolkit hides the complexity of the object identification (e. g., RFID reader hardware, computer vision) from the developer. It provides high-level events of the detected physical objects that developers can integrate into their custom applications. The Papier-Mâché radically simplifies the development of tangible user interfaces; however, the integration of physical interface hardware is limited to the detection of objects via RFID and computer vision. Here, the Shared Phidgets toolkit is intended to support a wider range of input and output devices (e. g., accelerometer, servo motors, and displays). Furthermore, the Papier-Mâché toolkit is limited to local applications, whereas the Shared Phidgets toolkit explicitly supports the distributed development.

Papier-Mâché includes utilities to support the TUI application development. To facilitate the debugging and testing of these applications, the toolkit offers monitoring windows for current input objects, overview of the computer vision recognition, and Wizard of Oz (WOz) simulation functionality. Therewith, developers can easily simulate hardware that is not available, or reproduce scenarios for testing the developed applications. Although Klemmer et al. [2004] apply these

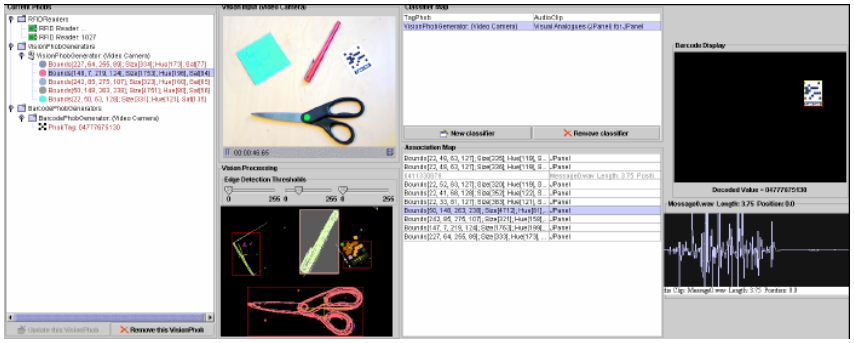


Figure 3.3: Papier-Mâché overview of available physical objects and computer vision recognition [Klemmer et al., 2004].

utilities to the development of local tangible user interfaces, the objective of the Shared Phidgets toolkit is the introduction of similar monitoring and simulation utilities for the development of distributed physical user interfaces.

3.4.5 Calder and BOXES

The *Calder* toolkit [Lee et al., 2004] allows designers to combine simple physical user interface components (e. g., buttons, RFID tags) with prototyping tools like Macromedia Director. Therewith, application designers can focus on the creation of the physical device and easily try different combinations of interactive hardware into these mock-ups. An important feature of the Calder toolkit are the miniaturised hardware building blocks (with wired and wireless connections) that make it easier to integrate the hardware into small physical objects. Although this simplifies the development of novel interface controllers, the toolkit architecture does not support the development of distributed appliances.

An extension to the Calder toolkit is BOXES (Building Objects for eXploring Executable Sketches) [Hudson and Mankoff, 2006]. It simplifies the prototyping process by allowing the coupling of physical interface elements to the software controls of GUI applications, similar to the *WidgetTap* Phidgets project [Greenberg and Boyle, 2002]. Calder and BOXES allow designers to prototype their systems with a very short *design/implement/test cycle*. The toolkit therefore encourages testing and evaluating alternative designs in the early design stages of the project, as changes of the prototype take only *"seconds rather than minutes or hours"* [Hudson and Mankoff, 2006]. This is also a very important objective of the Shared Phidgets toolkit, in that it provides developers (instead of interaction designers) a similar easy to use prototyping tool that allows the rapid exploration of

implementation ideas, and minimises the efforts of using high-level prototyping toolkits.

3.4.6 Equator Component Toolkit

The *Equator Component Toolkit* (ECT) [Greenhalgh et al., 2004] addresses the developers of ubiquitous computing environments. The toolkit integrates distributed hardware into an event-based data space and provides a component programming model for the development. These reusable programming components (e. g., JavaBeans [Sun Microsystems, Inc., 2007]) simplify the access to the hardware components and can be used by programmers in a similar way as the Phidgets hardware proxy objects [Greenberg and Fitchett, 2001]. The network architecture of the system is based on a distributed tuple-space with a subscriber and notification mechanism, similar to the iRoom's EventHeap [Johanson and Fox, 2002]. The toolkit includes utilities to observe the network events and components capabilities (left side of Figure 3.4), but does not provide means for controlling or simulating hardware. The system, however, provides an editor tool that allows the assembly of components to appliances as illustrated in the right side of Figure 3.4 [Egglestone et al., 2006,?]. This tool provides a limited control over the hardware and the event flow between these components, and is therefore primarily a tool for end users rather than developers.

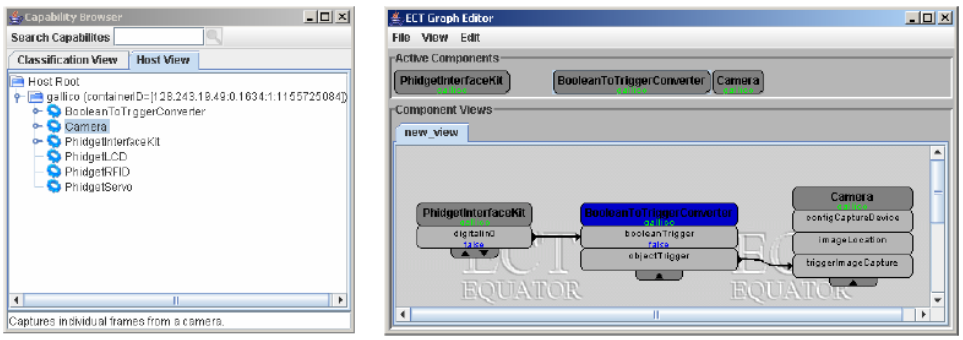


Figure 3.4: ECT capability browser and graph editor [Egglestone et al., 2006].

3.4.7 Voodoo IO Toolkit

The research of Villar and Gellersen [2007] with the *Voodoo IO* toolkit emphasises the importance of the spatial arrangement and location of physical user interface controls. Their introduced hardware technology allows the free arrangement of

buttons, lights, and sliders on a substrate material. Therefore, developers and end users can easily combine groups of controls to build meaningful and customised controls as illustrated in Figure 3.5. The substrate material is a flexible pad that can have any physical two-dimensional shape, and all physical controls have contact pins that connect them with the communication network once they are pinned onto the substrate. The electronic controls are using a one-wire communication protocol, and a programming interface enables the access to the connected hardware components [Villar and Gellersen, 2007].

Figure 3.5: Using Voodoo IO to transform everyday surfaces into control areas [Villar and Gellersen, 2007].

They illustrate the applicability of their toolkit with various examples of customised user controls attached around a computer setup. The hardware controls are assembled in various configurations, for instance to control a music player, navigate a map on a wall display, and build a game controller [Villar and Gellersen, 2007]. Their examples greatly enhance the user experience with traditional desktop applications, and illustrate how specialised physical user interfaces can simplify the interaction with information technology. The hardware system, however, is also limited to the specific set of available hardware controls, and it focuses on the development of local user interfaces extending the usage of desktop applications. Therefore, the system does not include specific support for the distributed development of physical user interfaces.

3.4.8 iStuff Toolkit

With the *iStuff toolkit* Ballagas et al. [2003] introduce a runtime platform and developer tools that support the building of novel input devices to control GUI software. Besides wired components (like most of the Phidgets hardware), the toolkit supports wireless input devices (e. g., wireless connected sliders, switches). The iStuff architecture is built on top of the Event Heap [Johanson and Fox, 2002],

which provides a networked tuple-space and an event notification mechanism. Using this network system, the toolkit handles hardware as event publishers and listeners. The software that handles a hardware component raises events when the hardware senses changes, and in return they change the hardware actuators if incoming events are received. Although the event-based programming model of the toolkit is a very lightweight technique for experienced developers, the toolkit does not offer a programming library with proxy objects (and the corresponding properties, methods, and events) for the hardware access. Moreover, the developers need to know the event protocol to access the hardware and have to handle the network events directly.

The *PatchPanel* [Ballagas et al., 2004] application is an extension of the iStuff toolkit, and allows developers to rapidly reconfigure the data flow between connected sensors and actuators. It provides methods to map and translate the incoming sensor events to outgoing events (e. g., to control an actuator). This is a very powerful mechanism to reconfigure the event flow between hardware and applications. The development utilities of the Shared Phidgets toolkit integrate a similar mapping functionality between hardware and appliances. It does not, however, integrate the scripting functionality of the PatchPanel, as the focus of the Shared Phidgets toolkit

With the *iStuff Mobile* toolkit, Ballagas et al. [2007] furthermore introduce a visual editor for designers to prototype new interactions with mobile devices (for instance mobile phones). Physical sensors (e. g., accelerometer, pressure sensor) can be attached to the mobile phone, and the visual editor allows designers to rapidly develop applications based on the sensed events. Designers can compose the data flow between sensors, actuators, and the application in the graphical editor, by adding proxy objects in the editor and defining the links between these objects (in a similar way as the graph editor of the ECT [Egglestone et al., 2006]). Ballagas et al. [2007] introduce various example applications with sensor equipped mobile phones. For instance, this includes an application for the scrolling of photo collections based on tilting the phone, text input supported by phone tilting, automatic changes of the ring tone profile, and interaction of the mobile phone with large screens [Ballagas et al., 2007].

The iStuff Mobile toolkit addresses interaction designers using the graphical editor user interface, whereas the Shared Phidgets toolkit addresses developers programming information appliances. Nonetheless, the graphical representation of the data flow between the physical components is a helpful visualisation for developers as well; for instance for monitoring, testing, and debugging of the developed appliances. The visualisation utilities of the Shared Phidgets toolkit are providing a similar view of the data flow of the distributed appliances.

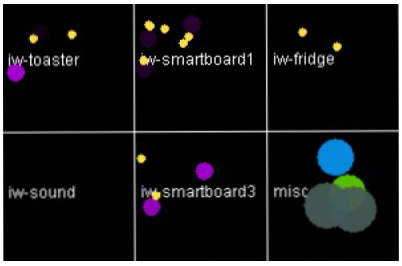


Figure 3.6: Event Heap visualisation for debugging [Morris, 2004].

In terms of support for the distributed development and debugging, Morris [2004] developed a visualisation for the Event Heap infrastructure. Although this visualisation is not directly integrated into the iStuff toolkit, it uses the same network event system (the Event Heap), and it introduces a valuable utility that supports the debugging of distributed applications. The developers of ubiquitous computing applications can use this utility to get a visualisation of all connected computer systems (the grid in Figure 3.6), and observe the events these systems raise in the distributed Event Heap network (the circle shapes in Figure 3.6), where the position, size, and colour of these shapes characterise the event attributes (e. g., event type, time to live). Therewith, the visualisation *"assisted novice users of [the] ubicomp environment with debugging and system understanding"* [Morris, 2004]. They can easily find out if a system is not responding, which systems are available, or if wrong events are raised or forwarded. The observe and control utilities of the Shared Phidgets toolkit address similar issues of the distributed development that are introduced in Chapter 6.

3.4.9 Overview of the Reviewed Toolkits

The previously reviewed toolkits vary in their supported prototyping techniques and provided programmer support for the development of distributed physical user interfaces. A comparison of the toolkit's implemented functionality cannot be completely balanced, as they are partially designed to address slightly diverse target applications. Nonetheless, all of the toolkits do support the development of local tangible and physical user interfaces, and partially they also support the distributed development. Table 3.1 summarises the toolkit characteristics in terms of the three categories introduced with the requirements in Section 3.3.

First, the toolkits vary in their implementation of a *runtime platform and infrastructure* that allows the access to distributed hardware. They all hide the implementation details of the hardware access from the developer, and in the majority of the cases they allow the attaching/detaching of sensor or actuator hardware dy-

	Phidgets (Greenberg and Fitchett, 2001)	**Context Toolkit** (Dey, 2000, Salber et al., 1999)	**Peripheral Display Toolkit** (Matthews et al., 2004)	**Papier Mâché** (Klemmer et al., 2004)	**Calder, BOXES** (Hudson and Mankoff, 2006)	**Equator - ECT** (Greenhalgh et al., 2004)	**Voodoo IO Toolkit** (Villar and Gellersen, 2007)	**iStuff Toolkit** (Ballagas et al., 2003)
Runtime Platform and Infrastructure								
Hide hardware access	■	■	■	■	■	■	■	■
Adding/removing hardware dynamically (plug&play)	■	■	■	■	▪	■	■	■
Integrate distributed networking layer		■	■	▪		■		■
Flexible runtime reconfiguration		■		▪		▪		■
Facilitate extensions and integration of custom hardware (e.g., plug-in based)		■	▪	■		▪		■
Development and Prototyping Support								
Development library and OOP concepts	■	■	■	▪	■	■	■	■
Event-driven architecture	■	■	■	■	■	■	■	■
Software proxy objects for hardware components (e.g., JavaBeans, .NET)	■	▪			▪	■	▪	
Visual user interface representations of hardware components	■			▪			▪	
Transparent accessable distributed data model		■				▪		▪
High-level abstractions / events		■	■	■		▪		■
Metadata integration		▪	■			▪		
Seamless integration into development tools (e.g., infrastructure exploration as IDE plug-in)	▪			■		■	▪	
Development Utilities (Monitoring, Controlling, Debugging, Simulating)								
Infrastructure explorer / observer	■	▪	▪	■	▪	■	▪	■
Controlling and initialising of hardware		▪				▪	■	
Application / appliance observer and control						▪		▪
Event visualisations (e.g., network)		▪		■		▪		▪
Testing and debugging support (e.g., *Wizard of Oz* simulations, test cases)	▪	■	▪	■				

Table 3.1: Overview of the prototyping toolkits in the related work (■ = supported/implemented, ▪ = partially supported/implemented).

namically. They do, however, only partially implement a distributed networking layer, or allow the dynamic reconfiguration of running applications. Another important requirement, the support of extensions for integrating custom hardware, is also not implemented in all reviewed toolkits.

Second, the support for the *development and prototyping* differs between the toolkits. Although all of them offer the object-oriented programming support and an event-driven architecture, only a few of them encapsulate the specific hardware's behaviour in reusable software building blocks with an easy to use API (e. g., as *.NET Common Language Runtime* (CLR) components [Meijer and Gough, 2000]). Only three of the toolkits implement GUI representations of the hardware device status. Partially, the toolkits allow the access to the distributed data model (e. g., to handle network events directly), the publishing of high-level or abstract events (e. g., building interpreters for raw sensor data), or the utilisation of metadata entries (e. g., to assign more specific location information to a sensor). Only partially supported is the seamless integration of the toolkit or its utilities into the programming environments (e. g., the Java Eclipse IDE, or Visual Studio).

Third, the toolkits provide diverse *development utilities*. On the one hand, all nine toolkits implement some sort of exploration utility to monitor the local or distributed infrastructure of current hardware. On the other hand, only few of them allow the interactive control or initialisation of the hardware with a GUI, the exploration of built appliances, or the visualisation of appliance events. Furthermore, five of the toolkits implement means for the simulation of hardware or appliances.

The review in this section and the comparison in Table 3.1 have highlighted the diversity of the existing toolkits. The review also showed that the available toolkits only partially fulfil the requirements for distributed physical user interfaces. Therefore, the next chapter begins with the in-depth description of the Shared Phidgets toolkit to fulfil the requirements.

3.5 Chapter Summary

This chapter introduced the background of prototyping concepts, summarised the important requirements of a rapid prototyping toolkit for distributed physical user interfaces, and gave a review of related work. In this review, the concepts and implementations of previous developed toolkits have been described, with a focus on their developer support for programming distributed hardware components. The next chapter now introduces the architecture of the Shared Phidgets toolkit that addresses the requirements described earlier in this chapter.

CHAPTER 4

Runtime Platform

In the previous chapter the requirements of a rapid prototyping toolkit for the development of distributed physical user interfaces were introduced. Furthermore, the related work of prototyping toolkits has been reviewed. These toolkits either only address the development of local physical user interfaces, or they allow the assembly of distributed interfaces on a high-level basis. None of the previous toolkits addresses all the requirements explained in Section 3.3, whereas the most important requirement is the facilitation of the programming of distributed physical user interfaces for average programmers as well as expert developers. Therefore, this chapter introduces the concept of the Shared Phidgets toolkit that supports the rapid prototyping of distributed information appliances.

4.1 Overview of the Shared Phidgets toolkit

This section covers the description of the overall architecture of the Shared Phidgets toolkit (that is illustrated in Figure 4.1), and therefore outlines the contents of this and the following two chapters.

The Shared Phidgets toolkit consists of *three main parts* that facilitate the development of distributed physical interfaces. First, it includes a *runtime platform* that allows the access to the physical user interface components from distributed client machines over a network. Second, a *developer library* supports the development of distributed information appliances. Third, a set of *development utilities* allow the monitoring, controlling, simulating, and debugging of developed appliances and distributed hardware devices.

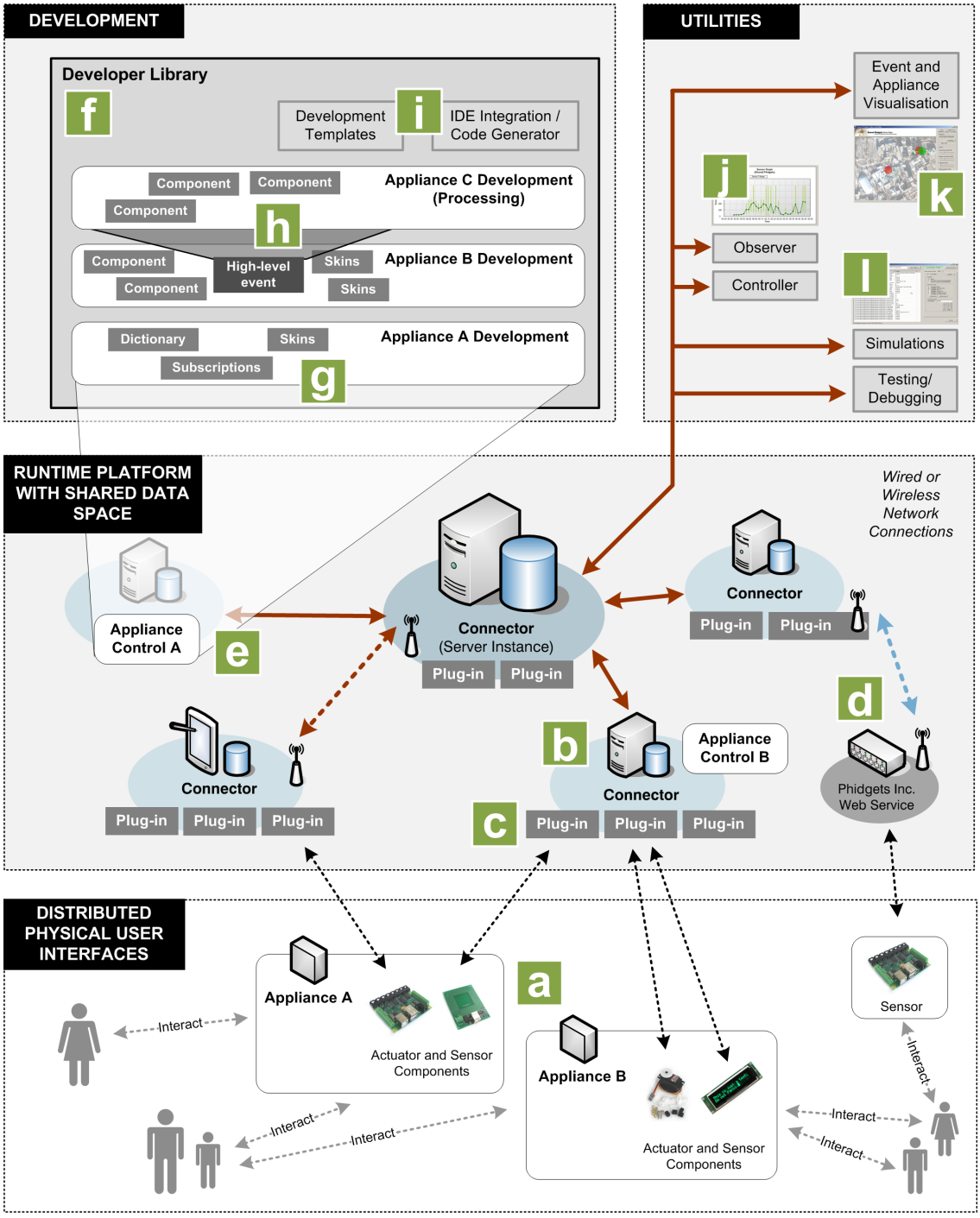


Figure 4.1: Architecture and main components of the toolkit.

The distributed sensors and actuators illustrated in Figure 4.1(a) are essential parts for the physical user interface of the information appliances. The end users

work with these information appliances through the interaction with the physical user interface. By using the Shared Phidgets *runtime platform* (illustrated in the centre of Figure 4.1) developers can easily instantiate infrastructures of network connected physical hardware sensors and actuators with minimal administrative efforts. The *Connector* software shown in Figure 4.1(b) maintains a shared data space that allows the distributed access from the client computers to an abstract data model of each utilised hardware device. Plug-ins of the *Connector* mediate between this shared data space and the hardware devices (shown in Figure 4.1(c)). These plug-ins either access locally connected hardware or they connect to other network services that in turn access the hardware devices as shown in Figure 4.1(d). Concept and implementation of this shared data space, as well as details of the *Connector* software and plug-in architecture, are introduced in this chapter.

To build their envisioned information appliances for the physical interactions, developers can write an appliance control software (shown in Figure 4.1(e)). This software can be executed on any of the networked client machines, and observes and controls the hardware via the shared data space. To facilitate the development, the Shared Phidgets toolkit provides a developer library that is illustrated in Figure 4.1(f)). This library allows the programming of information appliances by means of diverse programming strategies (cf. Figure 4.1(g)). The toolkit also allows the utilisation of high-level events (cf. Figure 4.1(h)) to support appliances that publish aggregated or interpreted high-level information to the shared data space. These strategies, as well as the appliance development, the seamless IDE integration (cf. Figure 4.1(i)), and the implementation issues of the extensible class framework are introduced in the subsequent Chapter 5.

Finally, the third main part of the toolkit is the integration of diverse development utilities, illustrated in the upper right corner of Figure 4.1. First, the monitoring and control utilities shown in Figure 4.1(j) allow the access to diverse abstraction levels of the toolkit's architecture. Second, the geographical infrastructure visualisation utility illustrated in Figure 4.1(k) provides insights into the internal processes and the spatial distribution of appliances. Third, the simulation utilities shown in Figure 4.1(l) facilitate the testing and debugging of distributed appliances. Concept and implementation details of these development utilities are introduced in Chapter 6.

This was a general overview of the built Shared Phidgets toolkit architecture and the outlook of the next three chapters. The chapter continues with a detailed introduction of the *runtime platform* concept.

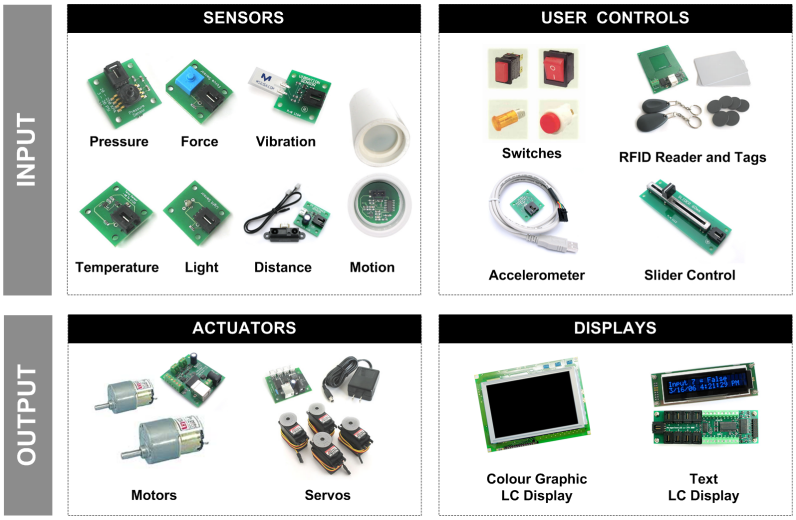


Figure 4.2: Overview of the physical user interface input and output components.

4.2 Runtime Platform Concept

This section introduces concepts of the Shared Phidgets toolkit runtime architecture. This includes details about how the toolkit provides distributed access from client computers to the shared physical hardware devices.

4.2.1 Hardware Integration

As illustrated in Figure 4.1a, hardware devices are assembled to act as information appliances. These appliances are situated in the environment and provide the physical interfaces for the interaction with end users. Appliances can comprise various types of input and output hardware. The following sensor and actuator elements illustrated in Figure 4.2 are available as hardware building blocks for developed appliances:

- *Input Sensors:* motion, temperature, force, proximity, magnetic field, light, vibration, acceleration, GPS satellite signals
- *Input Controls:* switches, dials, sliders, touch controls, joysticks, key fobs, RFID readers
- *Output Actuators:* servos, motors, solenoids
- *Output Displays:* text displays, graphic displays, lights

By using *Input Sensors*, appliances can get implicit sensor data about the context of the environment. This enables the appliances to interpret this data to derive high-level context information [Salber et al., 1999]. For instance, light and motion sensors can provide useful context information about the current utilisation of a meeting room (e. g., daylight and movement can be interpreted as a normal meeting, dimmed lights and less movement as an ongoing presentation, and darkness and no movement as an empty meeting room).

Input Controls are used for direct feedback of users to the information appliance. With the integration of these physical input controls, developers can build appliances that suggest possible actions to users. Norman [1988] describes this as *affordances*: the perceived action possibilities (as described earlier in Section 3.2). For instance, a physical button can *suggest* users to push the button to trigger an action, as the users fall back on their previous experiences with such objects.

To provide explicit feedback to users, appliances can change the location or status of objects in the environment with *Output Actuators*. For instance, servo motors can be used to rotate objects or to open and close a container.

The *Output Displays* are the other form of actuators to provide feedback. Small text displays can show messages and information, whereas graphic displays are also able to show digital images or videos. Although these displays provide visual feedback similar to conventional computer screens, they are used differently: the small output displays are part of the appliance, and they are more flexible in their application (for example they can be used as ambient displays or to display digital photos).

These various sensors and actuators are usually connected to the USB or serial port of a computer. For instance, each of the Phidgets devices is connected to a USB port, and the devices are then accessible from the local computer. Alternative connection methods are the serial, parallel, or infrared port of a computer; bluetooth; wired or wireless network; or custom radio frequency (RF) transmission protocols.

To allow the access to these locally connected hardware sensors and actuators, the Shared Phidgets toolkit runtime platform includes the *Connector* software (cf. Figure 4.1(b)) that is running on the client computers. This software, in combination with a set of *plug-ins*, acts as a mediator between the network of the Shared Phidgets infrastructure and the hardware that is connected to the client computers. The plug-in architecture of the *Connector* addresses the required extensibility of the runtime platform (cf. Section 3.3). To allow the easy integration of custom hardware, the specific implementation for each hardware device is encapsulated in the plug-ins. A basic set of plug-in reference implementations is provided by the toolkit. These reference plug-ins integrate various physical user interface compo-

nents: Phidgets hardware devices [Phidgets Inc., 2008], a GPS[1] receiver, a GSM[2] modem, and graphic LCD screens. Custom plug-ins can be developed by experienced developers when new hardware should be integrated; the details of the plug-in implementation are described at the end of this chapter.

In summary, the toolkit offers the developer the access to a large collection of physical hardware devices that can be integrated into their distributed appliances. The *Connector* software and integrated plug-ins access the local hardware devices and allow the distributed access from the client computers over the network. This network access with a distributed data space is introduced in the following subsection.

4.2.2 Shared Distributed Data Space

To allow the distributed access to shared hardware over the network, the Shared Phidgets toolkit maintains a shared data space with abstract representations of all distributed toolkit components. This shared data space is provided by a *shared dictionary* of the .NETWORKING toolkit [Boyle and Greenberg, 2005]. This toolkit provides a shared dictionary server as a central instance of the shared data structure. Client instances of the shared dictionary open connections to the server instance to read or write data objects to/from the dictionary. These operations are similar to accessing a local hashtable data structure, with the difference that the instances of the shared dictionaries are synchronised between all client machines and the server over the network. The data objects in the shared dictionary are represented as hierarchical key/value pairs. The keys of these data entries are organised as path expressions (e. g., `/root/subentry/`). The values can be primitive data types (e. g., boolean, integer, float) as well as complex data structures (e. g., lists, vectors, maps).

The shared dictionary also provides a *notification* mechanism that works similar to the Stanford *Event Heap* [Johanson and Fox, 2002] and the *Elvin* server [Fitzpatrick et al., 2002]. Clients can register to receive notifications of changes in the data structure. These subscriptions also define which part of the shared dictionary is synchronised between the client and the server: a client can subscribe only for a particular subset of entries from the dictionary, and hence only changes of these entries are transmitted from the server instance to the client.

The Shared Phidgets toolkit automatically creates all necessary connections to the shared dictionary, subscribes for events, adds event handlers, and interprets and

1 Global Positioning System
2 Global System for Mobile Communications

forwards the shared data space events. Although developers can decide to access the shared dictionary directly (one of the programming strategies explained in Subsection 5.1.1), typically the handling of these network connections and events is hidden from the developer.

Choosing the centralised architecture of the .NETWORKING notification server as the network basis of the Shared Phidgets toolkit implies certain drawbacks though. Client-server architectures usually do not scale very well, and with a high number of connected client instances the server can be the bottleneck of the distributed architecture. The routing of all sensor events through the single server instance could lead to latency problems. The synchronisation of dictionary values between the server and clients also cause high network traffic. Furthermore, a centralised server is also a weak point that affects the complete infrastructure in the case that failures occur.

Nonetheless, in previous research projects of prototyping toolkits, client-server architectures with notification servers have been proven to provide adequate performance for the demands of the developed applications (e. g., in ECT [Greenhalgh et al., 2004], Context Toolkit [Salber et al., 1999], or PTK [Matthews et al., 2004]). The centralised notification servers avoid the complex management of decentralised peer-to-peer and ad hoc communication systems (as it is often used for large scale sensor networks). Moreover, a notification server architecture could be replaced by more advanced shared peer-to-peer architectures, or by advanced servers implementing load balancing strategies [Bienkowski et al., 2005], as for instance later versions of the IBM TSpace server [Lehman et al., 2001]. To minimise the drawbacks of the client-server architecture and the costly synchronisation between clients and server, the Shared Phidgets toolkit is automatically managing fine grained subscriptions for events of the shared dictionary. This ensures that information is only synchronised between the server and client, if the data of this shared dictionary entry is needed by the software running on the client.

4.2.3 Distributed Model-View-Controller

To allow multiple client machines to observe and control the hardware devices, the access is mediated with a *distributed Model-View-Controller* pattern [Greenberg and Roseman, 1999]. Figure 4.3 illustrates the shared model representation with multiple views and controllers. The *model* comprises the abstract representations of all shared hardware devices, whereas the hardware and the controlling software can operate as *views* and/or *controllers*. Hardware devices work as a *controller* (illustrated in Figure 4.3(a)) if they change entries in the *model*, for

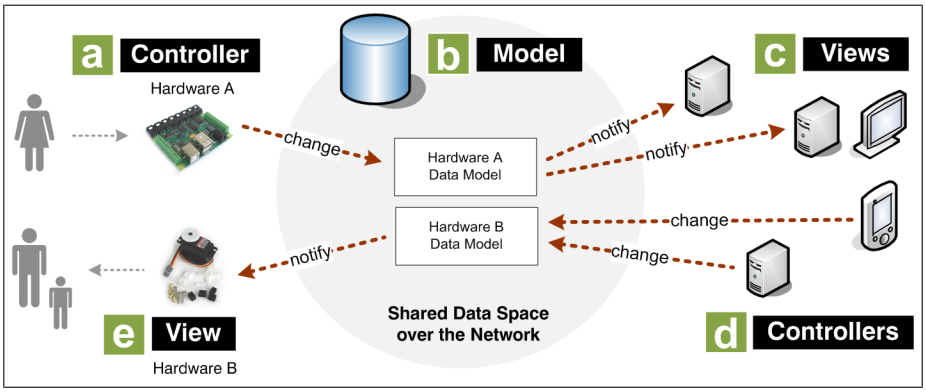


Figure 4.3: Distributed Model-View-Controller pattern.

instance because the current value of a sensor is updated. The shared data *model* of the hardware covers a complete abstract representation of the hardware and the current status as shown in Figure 4.3(b). Based on this abstract model, the software on the client side can create multiple views onto this hardware model as illustrated in Figure 4.3(c). A view for the software would be for instance a graphical visualisation of the hardware status. The software can also operate as *controller* (cf. Figure 4.3(d)); in this case it changes the hardware's data *model* representation. In turn, the hardware changes its status according the changed *model* entries (cf. Figure 4.3(e)).

In summary, the *distributed Model-View-Controller* pattern allows the abstract representation of the shared hardware with the current status and properties in the shared *model*. Multiple clients can operate as *views* and *controllers* at the same time. The detailed structure of the hardware data model is described in the next subsection.

4.2.4 Hardware Data Model

Each shared hardware device is represented by a unique set of entries in the shared data space. These entries define the device type, specific hardware properties, and the current state.

The structure of these entries in the shared data model is illustrated in Figure 4.4. All entries that describe a particular hardware device are subentries of the following path hierarchy in the data structure: the first part is always the root path of the toolkit (`/sharedphidgets/`), the second part is the type identifier of the hardware (e. g., `/servo/`), and the third part is the serial number (e. g., `/418/`). With this path (e. g., `/sharedphidgets/servo/418`) each hardware device can be iden-

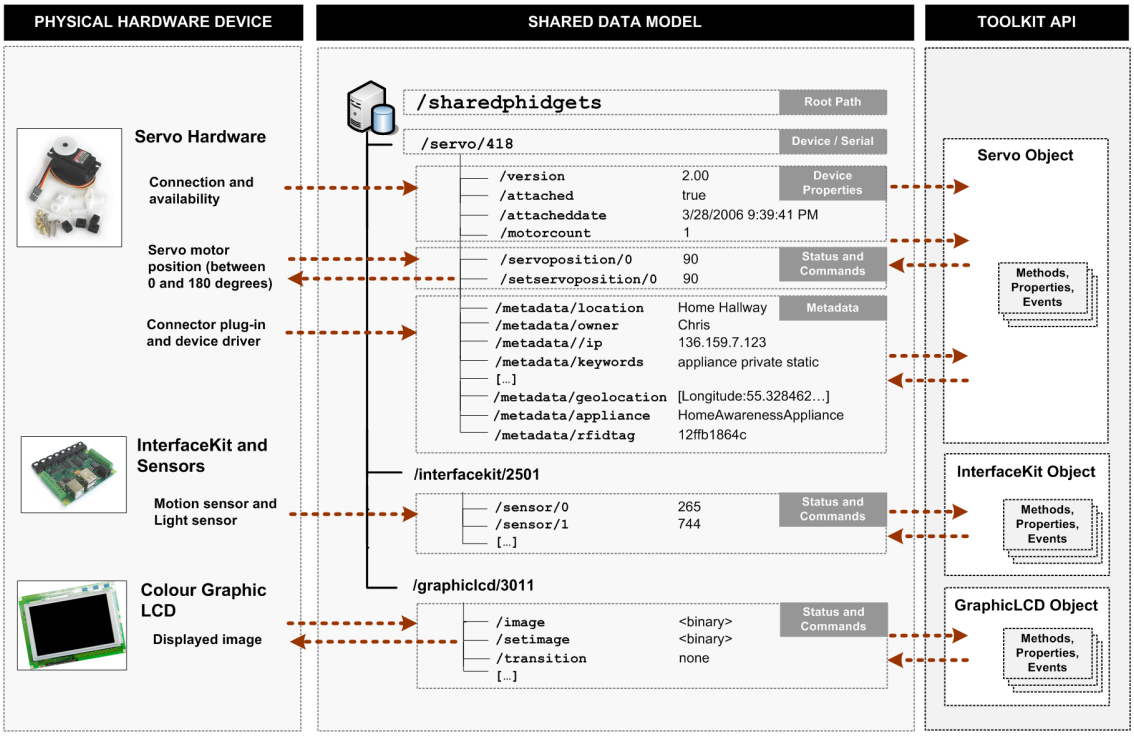


Figure 4.4: Abstract hardware model in the shared data space.

tified unambiguously in the shared data model, as the combination of hardware type and serial is a unique identification. All following parts of the path hierarchy correspond to the particular attributes of the hardware. These complete path expressions are the keys of the shared data structure to access the value entries. The possible entries fall into one of the following three categories: *device properties*, *status and commands*, and *metadata*.

The first three entries of the *device properties* are mandatory for all hardware devices: they specify if the hardware is currently attached to one of the local computers, the time stamp when the hardware was attached, and the version number of the hardware. Further entries can define particular static hardware properties (e. g., the total number of servo motors or digital inputs).

The device *status and command* entries represent the available input and output functions of the hardware (shown in the upper part of Figure 4.4). Each input of the hardware is represented by a separate entry in the data structure (e. g., the value of a specific sensor can be accessed with the path `/sharedphidgets/interfacekit/2501/sensor/0`). The outputs of the hardware that can be controlled by the remote client machines are each represented by two entries: the first entry represents the current status of the output, and the second entry can be accessed by remote clients to change this output (e. g., the LCD display's current image is

represented by the entries `/image` and `/setimage`). This distinction is important since there can be a delay between the command that changes the output, and the actual confirmation of the output change. This delay depends on the actuator hardware and can vary between milliseconds up to a few seconds (e. g., the rotation of a servo motor can take up to a few seconds).

The *metadata* entries (illustrated in the centre of Figure 4.4) contain additional information about each hardware device. All of the metadata entries remain in the `/metadata/` subpath of the hardware data model. The metadata information of each hardware device includes five static properties. The location description (`/location`) is a string entry describing the current location, whereas the geographical location (`/geolocation`) can be added if the exact location of the hardware is known in longitude and latitude coordinates. It is possible to specify the owner of the hardware (`/owner`), the IP address of the local computer the hardware is connected to (`/ip`), and a set of keywords to describe the hardware (`/keywords`). Custom metadata entries can be added from the plug-in that integrates the hardware, as well as from any client machine. For instance, an appliance can add metadata entries to hardware devices for identification and to specify groups of hardware; or a separate tool can assign special RFID tags to the hardware that facilitate the exploration of these hardware components (a technique that is described in the following chapter).

In summary, the entries of each hardware device in the shared data model represent this hardware's attributes and current status. The entries define a unique *model* of each hardware device, and allow the distributed access of *views* and *controllers* on the client machines to observe sensors and control actuators.

4.2.5 Appliance Concept and Data Model

Developers can combine the shared hardware devices to *information appliances*. All these appliances comprise two parts: first, the assembly of hardware sensors and actuators (cf. Figure 4.1(a)) that can be combined into a single unit or distributed across remote locations; and second, a developed software unit that observes and controls the hardware (cf. Figure 4.1(e)). Furthermore, this controlling software implements the appliance logic of how to handle and interpret the sensor events. The software controls how the physical actuators or displays should be modified in response to the measured values. As previously mentioned, this controlling software unit can be executed from any client machine, because a synchronised access to the shared data model is provided. While Chapter 5 explains the development process of appliances in further detail (cf. Figure 4.1(f)),

at first the general concept of appliances and their representation in the shared data space is introduced.

Similar to the representation of each hardware device in the abstract shared data model, each appliance represents itself as a collection of entries in the path hierarchy of the shared dictionary, as listed in Figure 4.5. In order to create unique path entries for each appliance the path expressions include a globally unique identifier (GUID)[3] [Leach et al., 2005]. The entries of each appliance include information about the name, a connection timestamp, and IP address (shown in lines 1–3 of Figure 4.5), as well as a listing of all hardware sensors and actuators that are addressed from this appliance (cf. lines 5–9 of Figure 4.5). The latter is important for the dynamic configuration of the appliances: by setting the `/externalserial/` subpath of any of the registered devices of the appliance, it is possible to map this implementation to a different hardware device of the same type. Furthermore, the listing of the addressed hardware devices also allows the observation of appliances and their incoming and outgoing events; this is fundamental for the observer utilities introduced in Chapter 6.

Appliances can be implemented as aggregators or interpreters of incoming sensor events and therewith can provide a similar functionality as the *Aggregates* introduced by Dey [2000] and Salber et al. [1999]. Therefore, the appliance's data model also includes entries to publish high-level events. The entries of the `/processing/` subpath comprise all these events of the appliance (cf. lines 11 and 12 of Figure 4.5). These high-level values can be for instance results of a calculation based on various incoming sensor events or an identified pattern in an observed series of sensor values. Any client that is connected to the shared data space can then register to receive notifications about changes of these high-level values.

```
1  /appliance/<guid>/appliancename                    = Sticky Spots
2  /appliance/<guid>/timestamp                        = 20/10/2007 04:56:45
3  /appliance/<guid>/ip                               = 192.168.178.20
4
5  /appliance/<guid>/components/<cid>/type            = rfid
6  /appliance/<guid>/components/<cid>/serial          = 2967
7  /appliance/<guid>/components/<cid>/externalserial  = 2967
8  /appliance/<guid>/components/<cid>/path            = /sharedphidgets/rfid/2967/
9  /appliance/<guid>/components/<cid>/timestamp       = 20/10/2007 05:22:07
10
11 /appliance/<guid>/processing/<subpath1>            = 42
12 /appliance/<guid>/processing/<subpath2>            = True
```

Figure 4.5: Entries of the abstract appliance data model (<guid> = globally unique identification number for appliances, <cid> = unique component identification.

3 The GUID is a randomly created 128-bit number that can be used as unique identification of software objects, as the total number of possible keys is so large that the probability of two identical generated keys is very small [Leach et al., 2005].

In summary, these abstract representations of the appliances facilitate the dynamic runtime configuration, the observation of the appliance events, and the re-use of high-level events between appliances. The details of the appliance development are explained in Subsection 5.2.1 of the subsequent chapter.

4.2.6 Data Persistence

The Shared Phidgets runtime platform includes a persistent storage of the specific entries in the shared data space. This persistent storage, integrated in the *Connector* software, saves entries of the shared data space and loads all stored entries when the central server instance of the shared data space is started (e. g., after a computer restart).

Whether or not entries of the shared data space are kept persistent depends on the type of these entries. On the one hand, the *metadata entries* of hardware devices and appliances are stored by default, as they represent custom added device properties. On the other hand, the *status and command* entries as well as the *device properties* (cf. Subsection 4.2.4) are not stored persistently. These entries represent the current status of the hardware device and they are only available when the hardware is attached to one of the client machines. Therefore, the platform currently does not store the history of occurred events. As introduced in the following chapter, the developer toolkit explicitly emphasises the utilisation of the event-driven programming methods. If the history of events is needed though, custom storage methods for these values have to be implemented by the developer.

4.2.7 Security and Privacy

For the development of ubiquitous computing applications, security and privacy issues have to be considered [Weiser, 1991; Abowd and Mynatt, 2000]. *Security* issues of a system include the need for restricted access to the system, and the secure transfer and storage of the information (e. g., by password protection, and encryption). *Privacy* aspects include the limited and monitored access of others to the private data a system might store about users. It is also necessary that users have knowledge about the information that the system gathers and derives from the environment (e. g., sensor values). Although the developed Shared Phidgets toolkit as a rapid prototyping toolkit implies lower demands on the security aspect of the systems, it nevertheless implements basic security and privacy fea-

tures: *password protection*, *hiding of hardware from the shared data space*, and *transparent feedback for users*.

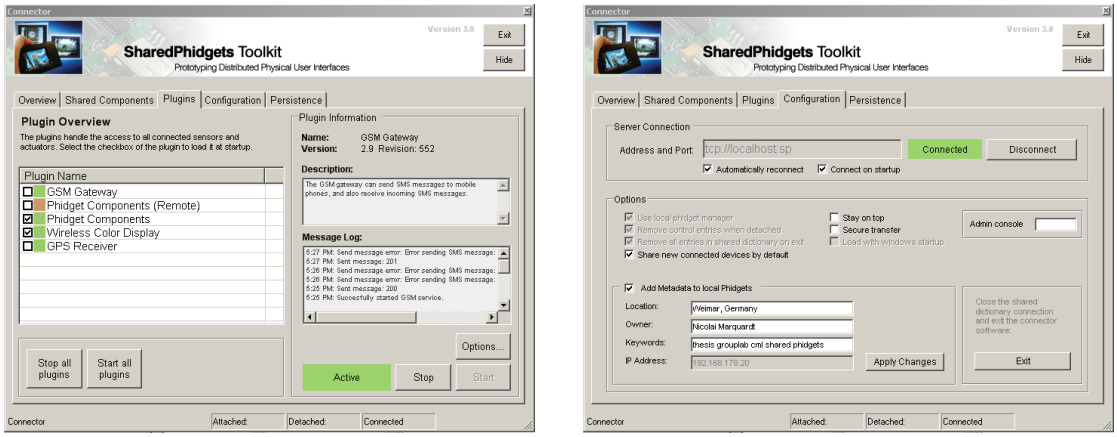
By using the *password protection*, users and developers can limit the access to the shared dictionary. This password security is provided by the .NETWORKING toolkit [Boyle and Greenberg, 2005] and allows restricting the access to the shared data space. With the *hiding of hardware from the shared data space* users can limit the access of others to their own hardware devices. In the *Connector* software, each device can be set as private and is then only accessible from the local computer. Consequently, this allows users to control the outgoing information of the locally connected sensors. Finally, the *transparent feedback for users* ensures that they can easily monitor the status of the distributed infrastructure. These monitoring utilities are introduced in Chapter 6; they allow insights into the transmitted events between hardware devices and the software.

Nonetheless, it is important to note that these integrated features of the toolkit only provide a basic level of security and privacy protection. The prototyping toolkit does not fulfil all security demands of deployed ubiquitous computing systems [Abowd, 1996; Abowd and Mynatt, 2000]. The toolkit does not yet include a fine grained access control (e. g., with user roles that define the user's right to access information in the shared data space), just as it does not support network connections secured with encryption algorithms. Developers have to be aware of these restrictions that the prototyping toolkit implies, and should consider additional security and privacy techniques before deploying systems.

4.3 User Interaction with the Platform

The most important characteristic of the runtime platform is that it automates the process of sharing and managing the hardware and works autonomously with a minimum of necessary user[4] interaction. The *Connector* software runs on each client machine and resides in the background of the operating system. It autonomously manages the available plug-ins that in turn handle the access to the locally connected hardware, and it registers each hardware device in the shared dictionary. Typically, the developers will rarely notice the *Connector* and the plug-ins, as they automatically find and share new hardware components when they are connected to a local computer (e. g., sensors or displays connected with a USB cable). Nonetheless, the user can access the GUI of the *Connector* to change the settings, start and stop plug-ins, and set options for the shared hardware.

4 *Users* in this case are the prototype developers. However, the *Connector* software of the runtime platform can also be used by people with no programming experience if they use a deployed information appliance.



(a) Plug-in overview (b) Configuration settings

Figure 4.6: User interface of the *Connector* software.

Figure 4.6 shows the *Connector* user interface. The plug-in overview interface is shown in Figure 4.6(a) and provides options for activating and deactivating plug-ins, deciding which plug-ins are activated automatically when the software starts, viewing the message log, and opening the plug-in configuration windows. In this configuration window, all settings of the plug-in can be changed. The local hardware devices that are currently managed by the plug-ins are visible in the shared device overview. The user can view the hardware settings, and change the metadata attributes for each local hardware device. Users can also set the hardware to be excluded from the shared data space (i. e., the hardware is then only available locally and is not available from other computers connected over the network as described earlier in Subsection 4.2.7). Finally, the network configuration (cf. Figure 4.6(b)) can be changed (e. g., the shared dictionary address, local or remote server, password) and users can alter the options of the persistent storage.

While this section has briefly introduced the user interface to configure the *Connector* and the *plug-ins*, usually the software runs hidden in the background of the operating system. This is important to minimise the necessary workload for developers when creating a networked infrastructure of hardware devices that they can integrate into the appliances to built.

4.4 Implementation Details and Extensibility

By using the plug-in architecture of the *Connector* tool, developers can easily integrate custom hardware components into the toolkit architecture. The plug-ins

encapsulate the access to the hardware components, and therefore intermediate between the hardware and the data model in the shared dictionary. While this architecture is usually hidden from the user, expert developers can extend the toolkit with custom plug-in implementations to integrate additional hardware sensors or actuators.

The Shared Phidgets toolkit software is developed on Microsoft Windows XP using *C# 2.0* of the Microsoft *.NET CLR* [Meijer and Gough, 2000]. The code examples in this thesis are also written in *C#.NET*. Nonetheless, all available *.NET CLR* development languages can be used alternatively (i. e., *C++*, *C#*, *Visual Basic*, and *J#*). More information about the development requirements of the toolkit can be found in Appendix A.1.

4.4.1 Plug-in Architecture

The plug-in architecture defines the interfaces between the plug-in host component (the *Connector* software) and the plug-in implementations. The architecture also provides base classes and templates, to minimise the efforts for the developers when implementing custom plug-ins. The following section describes this plug-in architecture and the plug-in reference implementations.

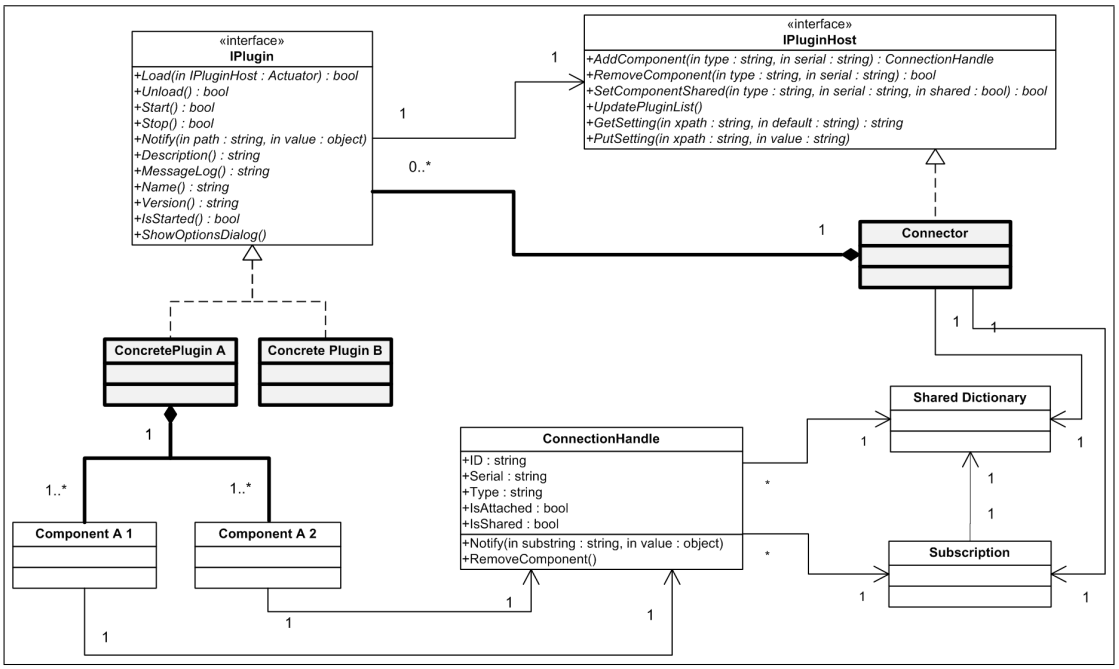


Figure 4.7: UML class diagram of the plug-in architecture.

The class diagram in Figure 4.7 gives an overview of the *Connector* plug-in architecture. The interfaces define the class responsibilities: the `IPlugin` has to be implemented by all plug-in components (e. g., the `PluginPhidgets`), and the `IPluginHost` interface is implemented by the class that manages all plug-ins (in the toolkit implementation this is the `Connector` class). A single plug-in can manage more than one hardware device at a time, and the connection to the shared dictionary for each hardware device is implemented with a `ConnectionHandle` instance. All interfaces and base classes are part of the `GroupLab.SharedPhidgets.PluginCore` namespace.

```
1  namespace GroupLab.SharedPhidgets.PluginCore {
2      // Interface definition for connector plug-ins.
3      public interface IPlugin {
4          // Properties of the plug-in
5          string  Name{get;}
6          string  Description{get;}
7          string  Version {get;}
8          string  MessageLog {get;}
9          bool    IsStarted {get;}
10
11         // Methods to manage the plug-in lifetime and status
12         bool    Load(IPluginHost host);
13         bool    Unload();
14         bool    Start();
15         bool    Stop();
16
17         // Show the options dialogue of the plug-in
18         void    ShowOptionsDialog(IWin32Window window);
19 }   }
```

Figure 4.8: IPlugin interface.

The `IPlugin` interface (listed in Figure 4.8) defines three types of properties and methods for the plug-in. First, the plug-in properties are read by the plug-in host on startup (e. g., name, description, version) and while the plug-in is executed (e. g., logging messages). Second, the methods `Load/Unload` are called once the plug-in assembly is loaded by the host, and `Start/Stop` are called to activate and deactivate the plug-in. Third, the `ShowOptionsDialog` is called by the plug-in host to display a dialogue window to set the plug-in properties. By implementing this interface, classes can be loaded dynamically by the plug-in host, and plug-ins get a reference to the `IPluginHost` with the `Load` method to call methods of the host.

The `IPluginHost` interface in Figure 4.9 provides methods for adding and removing hardware components that are handled by the plug-in. For each hardware device that a plug-in handles, it calls the `AddComponent` method of the host to receive a `ConnectionHandle` instance. This connection handle wraps the communication with the shared dictionary: it forwards events for all updated entries of the handled hardware component in the shared dictionary via the `Notified`

```
namespace GroupLab.SharedPhidgets.PluginCore {
    // Interface definition for the host connector application
    public interface IPluginHost {
        // Add a new component; returns ConnectionHandle
        ConnectionHandle AddComponent(string type, string serial);

        // Remove component
        bool RemoveComponent(string type, string serial);

        // Handle settings in the XML configuration file
        Settings GetSettings();
        string GetSetting(string xpath, string def);
        void PutSetting(string xpath, string value);

        [...]
}   }
```

Figure 4.9: IPluginHost interface.

event and provides methods to update entries in the shared dictionary by calling `NotifySharedDictionary`. The emphasis of this structure is on the limited access to the shared data space: the plug-in and the class that handles the hardware get only access to the subset of data entries for this hardware and can also send updates only to these entries. Once the hardware component is detached, the plug-in only calls the `RemoveComponent` method, and all entries of the data model are removed from the shared dictionary.

To save and load settings (e. g., port settings, device properties) the plug-in can access the `GetSetting` and `PutSetting` methods of the plug-in host. The plug-in host saves these settings in an XML file, including XPath entries [World Wide Web Consortium (W3C), 1999] for all settings added by the plug-ins. Although plug-ins can implement their own persistent storage of settings, these methods can be used as centralised storage of all configuration settings.

The UML activity diagram in Figure 4.10 illustrates the registration process of a plug-in, and the wrapped access to the shared dictionary with the `Connection-Handle`. First, the `Connector` class creates the `SharedDictionary` instance and loads the class instances (that implement the `IPlugin` interface) from the assembly files. The `Connector` loads and starts the plug-in. For each hardware device that this plug-in handles, the plug-in class creates a `Component` instance and registers this hardware device at the `IPluginHost` to get the corresponding `ConnectionHandle`. This handle wraps the access to the `SharedDictionary` and the `Subscriptions`. When the hardware status changes (e. g., events triggered by a motion sensor) the component implementation notifies the `ConnectionHandle` that updates the entries in the dictionary to notify all subscribers. On the other hand, when an entry of the hardware's data model in the shared dictionary is changed (e. g., the command to change the position of a servo motor) the `Con-`

nectionHandle notifies the component implementation of the plug-in to handle these changes.

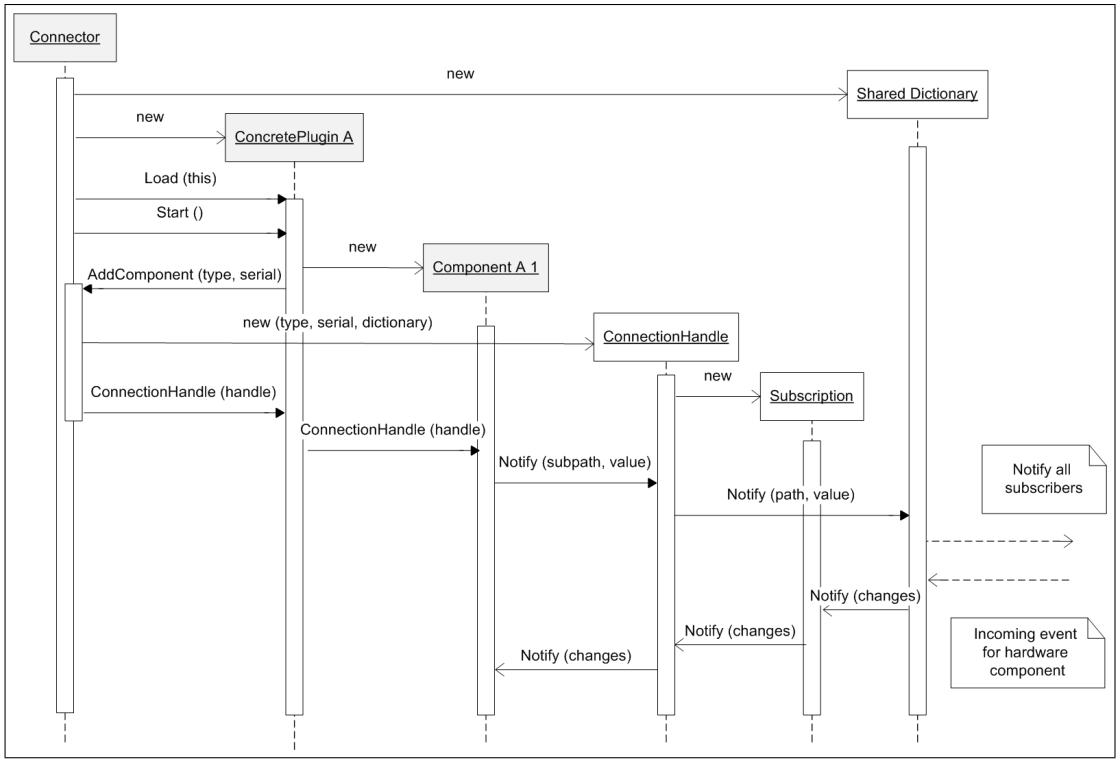


Figure 4.10: UML activity diagram of the plug-in architecture: loading of plug-ins and event registration.

To implement custom plug-ins developers have to import the `PluginCore` library file and can then compile the implemented plug-in as Dynamic-Link Library (DLL)[5] assembly. These assemblies can then be copied to the `Plugins` directory of the `Connector`. On startup the `Connector` tool scans all assemblies in this directory and searches for the implementation of the `IPlugin` interface. The assemblies with class files that implement this interface are integrated dynamically with the `Activator.CreateInstance()` method. The following section briefly describes the plug-in reference implementations.

5 *Dynamic-Link Libraries*: these shared libraries are linked to an executable program at runtime. They are the library assemblies of the *Common Language Infrastructure* [Meijer and Gough, 2000].

4.4.2 Plug-in Reference Implementations

The Shared Phidgets toolkit includes five plug-ins that handle the integration of hardware components to the runtime platform. The following list gives an overview of these plug-ins and highlights implementation details:

- **PluginPhidgets:**
  This plug-in can integrate all available Phidgets hardware components to the shared data space [Phidgets Inc., 2008]. The plug-in observes the local computer for any Phidgets hardware that is connected to one of the USB ports. Once a hardware component is found, it is immediately registered at the plug-in host and the data model of the hardware device is added to the shared dictionary. The plug-in includes a base class for generic properties and methods (`PhidgetDevice`) and derived classes for specific implementations of the Phidgets hardware types. These derived classes manage the forwarding of the occurred sensor events to the shared dictionary by using the `ConnectionHandle`. They respond in return to all changes of the hardware data model in the shared dictionary and control the actuators accordingly. The plug-in accesses the hardware via the Phidgets Inc. 2.1 driver library [Phidgets Inc., 2008].

- **PluginPhidgetsRemote:**
  This separate plug-in is included in the `PluginPhidgets` assembly, and manages the connection to the Phidgets Inc. socket servers [Phidgets Inc., 2008] (e. g., Linux servers running on embedded hardware). Users can specify the remote IP addresses of these socket servers in the plug-in's options window. The plug-in then automatically connects to these servers and observes them for connected hardware. Once a Phidgets hardware device is found, the plug-in integrates this device into the shared data space similar to the plug-in for local Phidgets. Hence, this plug-in integrates the external Phidgets Inc. servers into the platform and enables the access via the Shared Phidgets toolkit API.

- **PluginGraphicLCD:**
  For the integration of colour graphic LC displays into the architecture, this plug-in can remotely access a client software running on Windows Mobile 5 systems to display colour images on the screen. The plug-in provides a socket server, so that the mobile displays can register themselves as client displays by opening a socket channel to the plug-in (both USB and WLAN connections are supported). The client software on the mobile displays is drawing any images sent from the plug-in on the screen and can use transitions for the changes between two images (e. g., sliding the image to the right side of the screen). The plug-in automatically rescales oversized im-

ages, and compresses images for faster wireless transmission. The utilisation of wireless PDA devices is only intended as preliminary solution. It allows a very light-weight integration of wireless graphic displays into the developed appliance prototypes. Without this encapsulated access to the display hardware, it would be necessary for developers to implement complex custom software for the mobile operating system on the client device. The next version of this plug-in will support the *ezLCD*[6] screens as separate hardware to display colour graphic images.

- **PluginGPS:**
  This plug-in integrates GPS receiver hardware that delivers GPS data in the National Marine Electronics Association (NMEA) GPS format. The plug-in communicates via a serial port connection to access the data stream from the GPS receiver. It parses the `$GPGGA` information (Global Positioning System Fix Data) to extract the longitude and latitude coordinates of the current location. The plug-in converts this location that is given in degrees and minutes to decimal coordinates, and notifies these coordinates to the shared dictionary. However, it does not yet implement the complete NMEA command set (e. g., detailed satellite information).

- **PluginGSM:**
  By using this plug-in, developers can integrate phone text messaging into their applications. The plug-in opens the connection to a GSM modem, and forwards incoming text messages to the shared dictionary. The plug-in can also send text messages with the GSM modem. Therefore, developers can send text messages from within their applications and can get notifications of incoming text messages.

This collection of already implemented reference plug-ins ensures that a basic collection of physical user interface hardware can be easily connected to any client machine and is instantly available over the shared data model. Developers can directly use this collection of hardware, and they do not have to implement any hardware drivers or the network synchronisation to allow the shared access to the devices[7]. Furthermore, if any hardware component is needed that is not already implemented, the plug-in architecture allows the straightforward integration of the new hardware's data model into the shared data space.

6 Colour graphic liquid crystal displays (LCD) http://www.ezlcd.com/

7 A complete listing of all hardware devices that are implemented with these plug-ins can be found in Appendix C, and a listing of all *.NET* components of the developer library can be found in Appendix B

4.5 Chapter Summary

This chapter introduced the architecture of the Shared Phidgets toolkit that allows the distributed access to connected hardware components. The shared data space contains abstract *model* representations of the hardware. Client machines can generate *views* onto this model or operate as *controllers* to change the hardware status. This distributed Model-View-Controller pattern is implemented with the toolkit's *Connector* software that is running on all client machines and manages the network access to a shared data space. The plug-in architecture allows the integration of custom hardware, and five reference plug-ins are implemented to already integrate a collection of physical user interface hardware into the Shared Phidgets system.

The following chapter now introduces the toolkit's developer library that provides a collection of programming building blocks that facilitate the access to this shared data model.

CHAPTER 5

Toolkit Developer Library

The runtime platform that was introduced in the previous chapter enables the distributed access to the shared hardware devices. While this provides the foundation for the distributed access to the hardware, it does not facilitate the tasks that developers have to cope with when prototyping new information appliances with these hardware devices. Therefore, this chapter introduces the Shared Phidgets *developer library* that facilitates these programming tasks by means of diverse programming strategies, an object-oriented API, and a seamless integration into the development environment.

The chapter begins by explaining the structure of the developer library. Next, the fundamental programming strategies are introduced: direct access to the shared data model, using the object-oriented API of the proxy objects, and utilising interface skins. Next, details of the information appliance development are explained. Finally, details of the implementation are described.

5.1 Library Structure and Development Strategies

For the design of the developer library of the Shared Phidgets toolkit it is an important objective and requirement (cf. Section 3.3) to provide a *low threshold* for average developers and a *high ceiling* for expert developers, as previously described in Section 3.1 [Myers et al., 2000]. To achieve this goal, the toolkit implements a framework of programming components that addresses developers with diverse programming skills. With it, the toolkit supports the development based on varying abstraction levels, in that it allows average developers with moderate programming skills to easily learn and use the toolkit's capabilities. At the

same time it gives expert developers the tools for efficient and advanced programming of the hardware. The next sections of the chapter introduce the supported programming concepts as illustrated in Figure 5.1. It is important to note that developers can easily mix and match any of these development strategies.

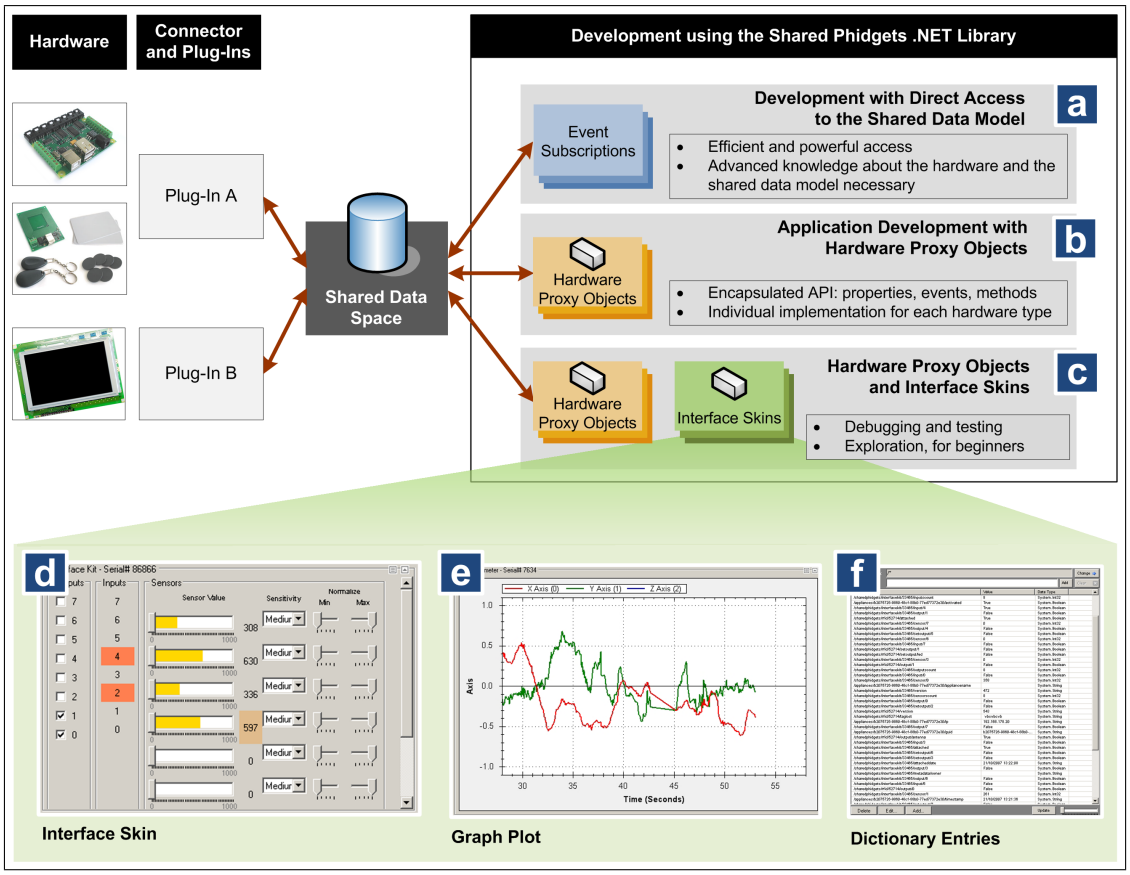


Figure 5.1: Developer library structure and programming strategies.

5.1.1 Programming via the Abstract Data Model

With this first programming strategy, developers can access the abstract data model of the hardware devices in the shared dictionary directly as illustrated in Figure 5.1(a). Therewith, developers gain powerful control of the infrastructure events and the distributed hardware. The programmers, however, have to be familiar with the distributed model-view-controller pattern and the abstract data model of the Shared Phidgets toolkit; and they need to know how to use the .NETWORKING shared dictionary API.

If developers choose this implementation strategy, they receive notifications of updated entries in the shared dictionary path hierarchy by adding subscription objects to their software. Furthermore, the developers need to add the corresponding event handlers for the notifications, parse and interpret the incoming events, and in turn modify entries in the shared data space to control actuator hardware. With these techniques, developers can easily iterate through collections of shared data model key/value pairs. Here, developers can utilise the pattern expressions provided by the .NETWORKING toolkit that allow path description patterns similar to regular expressions. Thus, the '?' character works as a placeholder for a single path hierarchy level, whereas the '*' character can stand for multiple levels. For instance, the following path expression generates matches for all entries of the servo's abstract model that control the current position of the motors.

```
/sharedphidgets/servo/?/setservoposition/?/
```

Developers can iterate through these collections of dictionary entries to control multiple hardware devices at the same time by altering the actuator control entries (e. g., reset the position of all servo motors by setting the value to 0). Another example of the pattern matching technique would be a collection of analog sensor values that can be for instance useful when calculating the average value of a collection of connected temperature sensors:

```
/sharedphidgets/*/sensor/0/
```

Furthermore, the direct access to the shared data model allows the exploration of the metadata entries. Developers can easily find collections of metadata entries that match a particular pattern, for instance the geographical location entries of all connected hardware devices.

```
/sharedphidgets/*/metadata/geolocation/
```

Overall, the programming based on event notifications and entries in a shared data space provides a powerful programming technique and has been previously applied in toolkits for the rapid prototyping (e. g., in the iStuff toolkit [Ballagas et al., 2003]). This technique, however, requires advanced programming knowledge. Therefore, it does not provide a *low threshold* for average programmers that learn the programming of information appliances. For this reason, the developer library includes a collection of high-level programming components that are introduced in the next chapter.

5.1.2 Hardware Proxy Object API

The developer library includes a collection of proxy objects that encapsulate the specific properties of each of the hardware elements (cf. Figure 5.1(b)). For

each of the different hardware types exists a corresponding class in the developer library that provides class properties[1] to access the hardware status, event notifications to monitor the status changes, and methods to control the hardware. Internally, the proxy object accesses the abstract model of the hardware, and the public API of the class hides the underlying networked infrastructure as well as the distributed data model from the developer.

For each of the sensor and actuator hardware types that are shared to the network data space with the plug-ins described in Subsection 4.4.2 (the left side of Figure 5.1) a corresponding class for the hardware *proxy object* is implemented in the developer library. This proxy objects implement an API similar to the one that was introduced by Greenberg and Fitchett [2001] for the local Phidgets toolkit. Therefore, developers can address the distributed hardware components as they have used local hardware, while the network architecture and synchronisation is hidden from the developer. Although the API of the proxy objects remains mostly unchanged compared to the local Phidgets toolkit, the internal implementation is replaced. The proxy objects connect to the shared data space and are built on top of the abstract shared data model of the hardware. The details of this implementation are explained at the end of the chapter in Section 5.4.

Because the hardware can be located anywhere on the network, it is essential to identify and map the physical hardware components to their software-side proxy object. As mentioned earlier in Subsection 4.2.4, the combination of the hardware type (e. g., `Servo`, `GraphicLCD`) and the serial number defines a unique identification for each physical device. Developers can easily address a particular hardware by setting the serial number of the hardware; or one device of a collection of hardware components by specifying multiple serial numbers (lines 1 to 3 in Figure 5.2). Alternatively, they can set the location filter property to match a hardware at a particular location (line 4 in Figure 5.2). For the advanced observation of connected devices, the `DeviceManager` provides the `attached` and `detached` events that notify registered listeners by forwarding the event to their event handler method (call-back method). In this method developers can check the device properties and metadata entries for detailed information about the hardware (lines 6 - 11 in Figure 5.2).

The object-oriented API of the hardware proxy objects matches the programming paradigms familiar from conventional GUI programming. Developers can easily alter the hardware's specific attributes by modifying the class properties. For instance, in lines 1 to 4 of Figure 5.3 a new proxy object for a graphic LC display is created, the displayed text and the transition effect are changed, and a bitmap is shown on the screen. The API also allows the easy access to the metadata entries associated with the wrapped hardware as illustrated in lines 6 and 7 of Figure 5.3.

1 *.NET* framework concept for combined get/set methods of a class.

```
1  TextLCD lcd = new TextLCD();
2  lcd.FilterSerialNumbers.Add("12933");
3  lcd.FilterSerialNumbers.Add("8374");
4  lcd.FilterLocations.Add("Building 221, Floor 1");
5
6  DeviceManager manager = new DeviceManager();
7  manager.Attach += new DeviceManagerEventHandler(manager_Attach);
8
9  private void manager_Attach(object sender, DeviceManagerEventArgs e) {
10   // Check the event attributes for details about the attached device.
11 }
```

Figure 5.2: Using the API to create proxy objects.

First, it is checked if the `location` metadata entry matches a particular location; and second, a new metadata entry is added that assigns this hardware to a particular appliance. Finally, developers can easily get notifications of the hardware's events by registering one or more event handlers. For instance, a new RFID reader proxy object is instantiated in line 9 (Figure 5.3). An event handler is registered for the `Tag` event in line 10 (Figure 5.3) that provides notifications if an RFID tag is near the reader and provides the ID of the tag as event parameter.

```
1  GraphicLCD display = new GraphicLCD();
2  display.Text = "Message text here.";
3  display.Transition = GraphicLCD.TRANSITION_LEFT;
4  display.Image = new Bitmap( [...] );
5
6  if(display.DeviceDescription.GetMetadata("location") == "Kitchen") { [...] }
7  display.DeviceDescription.AddMetadata("appliance", "Awareness Display");
8
9  RFID rfid = new RFID();
10 rfid.Tag += new RFIDTagEventHandler(rfid_Tag);
```

Figure 5.3: Using properties and event handlers.

Using the API of the proxy objects significantly facilitates the access of physical hardware devices from within the custom software projects. In contrast to the previously introduced programming technique via the shared dictionary directly, developers need no advanced knowledge about the underlying shared data space, the abstract hardware model, or the distributed MVC pattern. Moreover, the available properties, methods, and events easily reveal the hardware's capabilities and empower the programmers to develop the controlling software part with familiar OOP techniques[2].

2 An overview of the most commonly used hardware proxy objects and their API can be found in Appendix C, as well as in the developer documentation and tutorials [Marquardt, 2008].

5.1.3 Interface Skins

When developers are creating the software that implements the program logic of the distributed user interface they can add a GUI representation of their appliance that allows the monitoring and control of the appliance, even from a remote location. In order that developers have not to build this GUI from scratch, the developer library provides *interface skins* for the hardware devices of the infrastructure (illustrated in Figure 5.1(c)). These interface skins are wrapper objects around the proxy objects introduced in the previous section and they basically display a graphical representation of the hardware's current status. They include common GUI widget controls (e. g., buttons, sliders) so that users can directly control the hardware. With the GUI interface builder (e. g., the *form designer* in Visual Studio [Microsoft Corporation, 2007d]) the developers can drag and drop these GUI counterparts of the hardware onto their designed user interface.

These interface skins that can be used for the appliance control software are not necessarily a direct part of the envisioned interaction with the end user. In most cases the interaction with the appliance rather works entirely through the physical interface and not the software side. In the experience of our research group, however, these interface skins have been proven to be helpful for developers in understanding the capabilities and limitations of sensors, observing their behaviour, and initialising the hardware. Especially developers with no previous programming experience of distributed physical user interfaces can benefit from the graphical representation of the hardware's status. The graphical interface skins are an intuitive starting point that encourages the experimentation with the available hardware at the beginning of the development. They also work as an effective debugging and testing mechanism in the later stages of the project development. Furthermore, they are fundamental for the development of the utilities that are introduced in Chapter 6.

Each of the available hardware device types can be represented by multiple interface skins. These skins can visualise various aspects of the hardware and implement controls to alter the hardware properties. For instance, the connected sensors, inputs, and outputs of the Phidgets InterfaceKit hardware can be visualised with the control skin illustrated in Figure 5.1(d) that displays the current status of the inputs and let the user activate or deactivate the digital outputs. Alternative interface skins for this hardware are the graph visualisation in Figure 5.1(e) that shows a graph plot of the sensor values in the last 30 seconds, or the table illustrated in Figure 5.1(f) that lists all the entries in the shared dictionary that correspond to the InterfaceKit hardware.

For each of the available hardware devices of the toolkit various interface skins have been implemented to ease the exploration and control of the hardware. A

detailed overview of the implemented interface skins can be found in Appendix C. Developers can also create their custom interface skins; this is explained in the implementation part at the end of this chapter.

5.2 Appliance Development

These previously introduced development strategies and the toolkit's API are essential to facilitate the development of the distributed appliances. Based on these foundations, the next sections of the chapter introduce the concept details of the appliance development.

5.2.1 Appliance Development Overview

As all appliances include a representation in the shared data model (cf. Subsection 4.2.5), the developer library includes base classes, templates, and helper classes to automate the necessary appliance registration. Overall, these classes implement and hide the appliance model in a way that the developers have no additional efforts when building new appliances.

When developers start with the implementation of a new information appliance, first of all their appliance control software need to have a network connection to the shared data space that holds the abstract representations of all currently active hardware devices. The Shared Phidgets developer library provides base classes that manage the connection to the network. The easiest way to start a new appliance control software is by deriving from the `Appliance` base class. This class can open and close network connections and registers the appliance at the shared data space (line 2 of Figure 5.4). It also provides a basic set of properties and methods to control the appliance. Developers can set the appliance name (line 5 of Figure 5.4), the shared dictionary address (line 6 of Figure 5.4), and open the connection (line 7 of Figure 5.4).

```
1 // Derive from the appliance base class
2 public class Example : GroupLab.SharedPhidgets.Appliances.Appliance {
3   // Constructor; set appliance properties and open connection
4   public Example(){
5     this.ApplianceName = "Appliance Example";
6     this.SharedDictionaryURL = "tcp://localhost:sp";
7     this.OpenConnection();
8 } }
```

Figure 5.4: Creating the appliance control software.

Alternatively to this implementation developers could also choose to add a `ConnectionManager` or `ConnectionManagerSkin` object to their class. Both of these components are able to manage the connection to the shared dictionary. Programmers could furthermore decide to handle the shared dictionary directly; however, it is then necessary to manually assign the reference of this shared dictionary to all of the hardware devices and other components that need network access.

From here, developers can start with the implementation of their control software that implements the appliance logic by choosing one of previously described programming strategies. A programming walkthrough is outlined in Section 5.3, while more sophisticated example case studies are introduced in Chapter 7.

5.2.2 High-level Events

To support the encapsulation of programming objects and the building of reusable building blocks, the developed information appliances can furthermore publish high-level events to the shared dictionary. Thus, other clients on the network can subscribe for these events and in turn use these high-level values in their own software application logic. These high-level events represent various types of information. For instance, an appliance could implement a simple gesture recogniser and could publish the detected gestures (line 3 of Figure 5.5). Another appliance could calculate the average temperature out of multiple sensors and publish this calculated value as well (line 7 of Figure 5.5).

```
1 // Add a boolean high level event to the shared dictionary
2 // Complete path: /appliance/<id>/processing/gesturepattern/12/found
3 this.PublishProcessingValue("gesturepattern/12/found", True);
4
5 // Add a float value as high level event to the shared dictionary
6 // Complete path: /appliance/<id>/processing/averagetemperature
7 this.PublishProcessingValue("averagetemperature", 54.7768);
```

Figure 5.5: High-level appliance events.

Other appliances (located anywhere on the network) can access these high-level events of the appliance; either by using the `ApplianceObserver` helper class, or by subscribing for the corresponding dictionary path manually. Moreover, developers can even build interface skins for these high-level events of the appliances. Consequently, this is a light-weight, albeit powerful, method for the implementation of abstract devices. It simplifies the re-use of processed information of the appliances (e. g., calculations, context information).

5.2.3 Seamless IDE Integration

For the development of a programming toolkit it is not only important to build an API with the collection of reusable programming objects, but also to provide means to help developers exploring the functionality of the toolkit [Roseman, 1993; Greenberg, 2007]. As introduced in Section 3.3 it is a basic requirement to lower the *threshold* for developers that have no previous experience with programming physical user interfaces. It is important to build on their existing knowledge and familiarity with development tools [Greenberg, 2007]. To achieve this goal, the Shared Phidgets toolkit is closely integrated into the programming IDE. The toolkit provides the following means to simplify the programming process, to support the infrastructure exploration, and to reveal the toolkit's capabilities:

1. **Project Template:**
   The template provides an initial development starting point for a new appliance and developers can select this template directly from within the Visual Studio IDE, as illustrated in Figure 5.6(a). The template includes the basic class files for a new appliance project, the references to all necessary libraries, the classes for accessing the shared dictionary, and the appliance base methods to publish high-level events. It creates a new class framework and derives from the `Appliance` base class as described earlier in Subsection 5.2.1.

2. **Visual Designer Integration:**
   The Shared Phidgets toolkit integrates all available class components and user controls of the library into the IDE toolbox, as shown in Figure 5.6(b). Therefore, developers can easily add all components (e. g., the hardware proxy components) and the user controls (e. g., the interface skins) to the project by using the visual designer of the IDE. This can speed up the time to integrate new hardware proxy objects, but also facilitates the development by means of the property editors and automatic creation of event handler callback methods.

3. **Tutorials and Examples:**
   The toolkit offers various programming tutorials and step-by-step programming walkthroughs[3] as illustrated in Figure 5.6(c). Furthermore, the provided collection of example applications can help developers to get started with the development; even if they are unfamiliar with the programming of physical hardware.

4. **Infrastructure Exploration from within the IDE:**
   The exploration of the infrastructure is available from within the IDE. With

3 Available on the Shared Phidgets toolkit project website [Marquardt, 2008]

it, developers can easily explore the currently available hardware, monitor the current status, and control the hardware's properties; as for instance for an Phidget Interface Kit shown in Figure 5.6(d). Developers can explore available hardware components, view their current status, and control the hardware's properties.

5. **Automatic Code Framework Generator:**
   An extension of the previously mentioned explorer view in the IDE is the automatic code framework generator as shown in Figure 5.6(e). Once developers have chosen the needed hardware components, this extension generates the source code to access these hardware from within the application code. The extension can also create the interface skins and subscriptions for shared dictionary events.

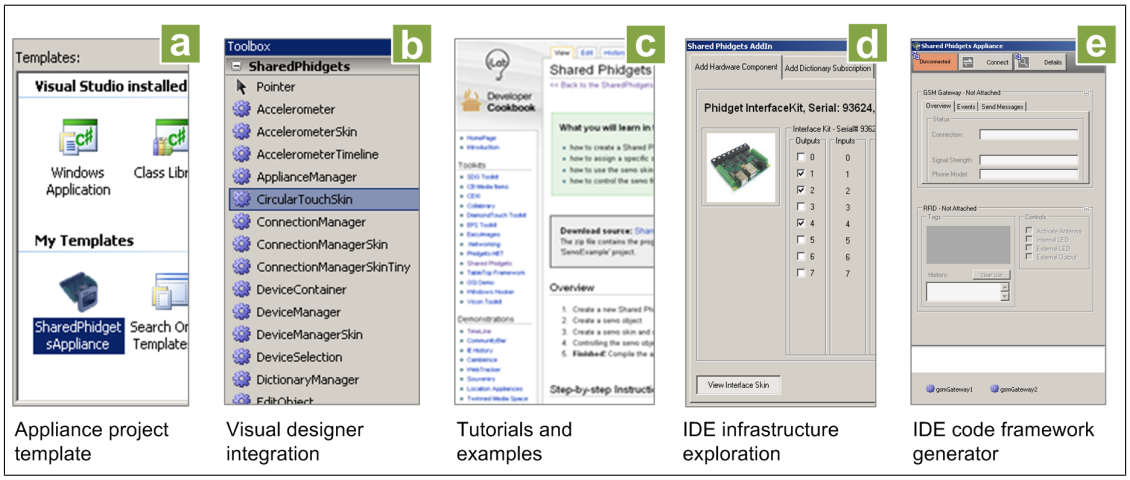


Figure 5.6: Methods for the seamless integration into the IDE.

All of these integrations into the familiar programming environment of the developers work together to minimise the necessary task of the developers. They all supplement the programming experience within the IDE by allowing the easy exploration of the toolkit's functionality, and they automate reoccurring and tedious programming tasks.

Especially the automatic code framework generator *add-in*[4] reduces the necessary work when programmers start with the development of an appliance. The development of this add-in is driven by the motivation to minimise the frequent programming tasks that occur in the starting phase of the appliance prototype development. The add-in extension simplifies the creation of new appliance develop-

4 The term add-in instead of plug-in is used here because it is the official term for this extension in Microsoft Visual Studio according to the Microsoft Developer Network (MSDN) documentation [Microsoft Corporation, 2007d].

ment projects as it can automatically generate a basic appliance code framework. Once the add-in started, it connects to the shared dictionary server instance and provides a list of all attached hardware components as shown in Figure 5.7(a). The add-in can also display the graphical interface skin of each of these components to easily test the hardware or to view the hardware's status.

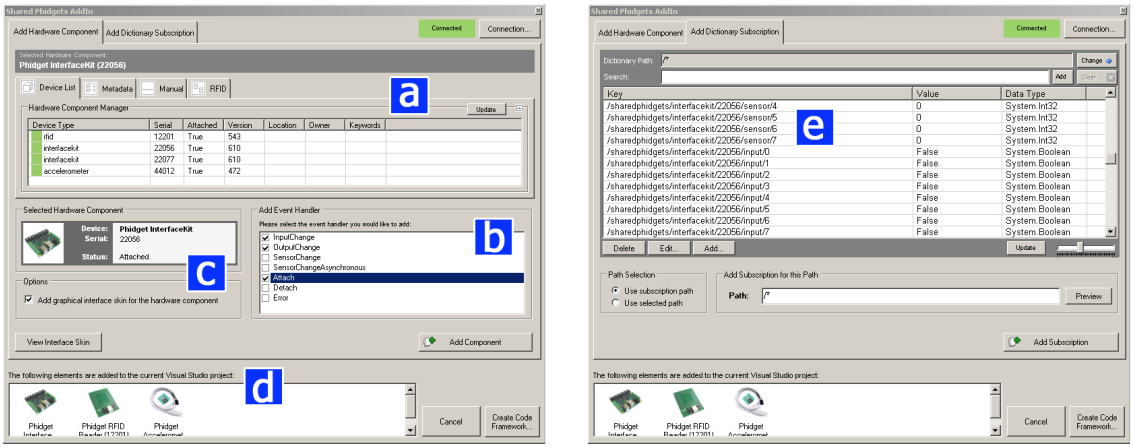


Figure 5.7: User interface of the IDE add-in.

The creation of the basic code framework for the development of a new information appliance works as following. First, the developer chooses the hardware component that he/she would like to use in the new project (cf. Figure 5.7(a)). Next, he/she can choose which event handlers (cf. Figure 5.7(b)) should be implemented (e. g., for the sensor change event), and if an interface skin should be added to the application form as well (cf. Figure 5.7(c)). Next, this hardware can be appended to the list of used hardware components (cf. Figure 5.7(d)). These steps can be repeated for all other needed hardware components. Once the developer has completed the list of needed hardware, the add-in automatically creates a code framework, whereas it automates the following tasks:

1. The add-in integrates the needed libraries to the development project.
2. For all selected hardware devices, the corresponding proxy component classes are added to the project. All class properties (e. g., serial number) are set accordingly.
3. The event handlers (for all of the selected events) and the callback methods are added to the project source code.
4. If selected, the corresponding interface skin of the hardware device is added to the application form.
5. For the integration of high-level appliance events, and for the facilitating of the shared dictionary event architecture in general, the add-in also sup-

ports the automatic addition of shared dictionary *event subscriptions* to the project. With it, developers can add subscriptions for high-level appliance events by just selecting the entry from the shared dictionary view as shown in Figure 5.7(e).

These steps have to be usually implemented by the developers themselves. However, as the add-in takes over these frequent programming housekeeping tasks, it therewith minimises the barriers of implementing high-fidelity prototyping projects (cf. Section 2.7).

Thus, this IDE extension simplifies the development in the following ways: it *allows the infrastructure exploration* from within the IDE, it *automates the integration of hardware proxy objects and event handlers*, and it provides means to *access the abstract data model* in the shared dictionary. An important characteristic of these integrations into the developer's IDE is the fact that even though they automate many frequent and tedious programming tasks, they do not intervene to deep into the programming concepts or try to automate the programming of the appliance logic themselves (cf. Section 3.1).

In summary, the seamless integration of the Shared Phidgets toolkit into the development environment of the programmers is an important part to *lower the threshold* for building high-fidelity prototypes. With it, programmers can easily discover and learn the toolkit's capabilities. Furthermore, the integrated add-ins take over a part of the frequent and sometimes tedious tasks of the development by automating parts of the code generation.

5.3 Programming with the Developer Library

Now that the previous sections of this chapter have outlined the available programming strategies, this section explains the utilisation of the toolkit developer library from the perspective of the prototype developer. This is explained as a walkthrough of the programming steps that are necessary to develop a new information appliance prototype. It begins with the basic setup to create a working infrastructure and proceeds with the basic steps to develop a new information appliance.

1. **Installation:**
   The Shared Phidgets toolkit has to be installed on all networked client computers. The *setup*[5] software automatically installs the required utilities and

5 The setup can be downloaded from the Shared Phidgets project website [Marquardt, 2008]

developer libraries. It also automatically integrates the toolkit into the Microsoft Visual Studio IDE.

2. **Infrastructure Setup:**
   The *Connector* software has to be started on all clients that have attached hardware devices. One of the *Connector* instances must be chosen to act as the server instance; therefore, the address of the *Connector* software on all other client machines must point to this IP address. All other aspects of the hardware integration are now handled by the software autonomously. By attaching any hardware devices to the client computers (e. g., Phidgets hardware or graphic displays) they are automatically added to the shared data space—ready for integrating them into custom projects by using the toolkit API.

3. **Appliance Implementation:**
   Developers can choose one of the variations of building an appliance framework as previously explained in Subsection 5.2.1. This is important to handle the network connection.

4. **Integrating Sensors and Actuators:**
   Developers can use any of the previously introduced programming strategies in Section 5.1 to integrate the hardware devices into their software project. For even simpler integration of hardware, developers can utilise the Visual Studio add-in.

5. **Programming Control Structures:**
   Developers now need to add their custom control logic into their appliance software code. This could be the interpretation of sensor values and the corresponding control of actuators.

6. **Compiling and Deployment:**
   After the compilation of the appliance software it can be started on any of the network connected computers. The appliance software handles the connection to the shared dictionary.

7. **Testing and Debugging:**
   The testing of developed appliances and the debugging to find appliance failures are important programming tasks. The Shared Phidgets toolkit provides a collection of utilities for these tasks; introduced in Chapter 6.

8. **Iterations:**
   By iterating on the design of the first prototype, developers could now try diverse implementation ideas. All hardware proxy objects can be easily exchanged and new ones can be added to the project.

By following these simple steps developers can easily build custom information appliances. Case studies of the prototype development are explained in Chapter 7.

5.4 Library Implementation and Extensibility

Similar to the runtime platform, the developer library is implemented in *C#.NET*. The library provides a collection of programming components that can be used with the *.NET CLR*, and is compiled as DLL assembly [Meijer and Gough, 2000].

The Shared Phidgets toolkit developer library is extensible in various ways. As mentioned in the previous chapter, developers can easily integrate custom hardware by implementing a `Connector` plug-in. While Section 4.4 already explained how to implement these plug-ins, this section now explains details of the developer library implementation. This includes extensions developers can build based on the the extensible class framework.

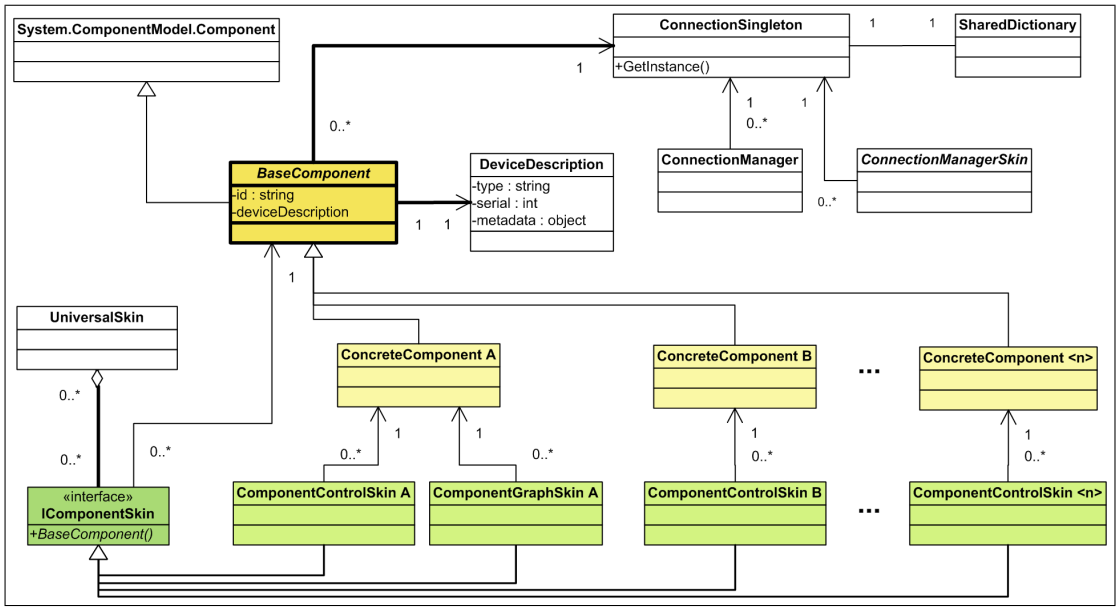


Figure 5.8: UML class diagram of the main components of the developer library.

The fundamental subset of the class architecture of the developer library is illustrated in Figure 5.8. The `BaseComponent` is the abstract base class for all implemented concrete proxy components for the hardware access. This abstract class derives from the `Component` class of the *.NET* component model, so that it can be used as component for the *.NET* form based development, for instance in the visual designer of Visual Studio [Microsoft Corporation, 2007d]). The

BaseComponent implements the basic set of methods and properties of the proxy components. Therefore, this base class minimises the effort of implementing the concrete proxy objects. The functionality of the BaseComponent comprises the following aspects:

- The class handles the connection to the SharedDictionary of the .NETWORKING toolkit, and instantiates the Subscriptions for events from the shared data space. It requests the SharedDictionary instance from the ConnectionSingleton as this is the default connection for all components. This dictionary instance is also accessed from the Appliance base class, the ConnectionManager or the ConnectionManagerSkin. Therewith, a single connection is shared by all the objects instances by default. If developers wish to use multiple network connections though, they can create multiple instances of the SharedDictionary connection and assign these connections to the corresponding property of the BaseComponent.
- The BaseComponent provides common properties of all implemented proxy components: the DeviceDescription with the hardware type identifier, the serial number of the hardware as well as access to the metadata entries.
- Finally, it provides a basic set of events and event handler definitions. These are for instance the Attach or Detach events that occur when the hardware is attached or detached. Another example is the SerialChanged event that is raised when the serial number of the proxy is changed and it therefore controls a different hardware.

Derived from the BaseComponent, the ConcreteComponent classes are the implementations of the specific proxy objects for the hardware devices (as introduced earlier in Subsection 5.1.2 of this chapter). For each of the available hardware devices a separate class implementation implements the specific hardware attributes (e. g., the current displayed Image property of the GraphicLCD or the Tag event of the RFID reader). To respond to the incoming notifications of the changes in the hardware's abstract data model (e. g., if the Connector plug-in of the hardware publishes updated data values of a sensor), the concrete proxy components override the abstract method subscription_Notified. In this method the developers can implement the code that handles these shared dictionary events and can in turn raise events of the implemented proxy component to notify interested subscribers (e. g., the SensorChange event of the InterfaceKit that is raised if the value of one of the analog sensors is changed). To furthermore simplify the development, the toolkit provides a set of helper classes in the GroupLab.SharedPhidgets.Utility namespace that are for instance utilised to parse dictionary path expressions, or to convert sensor values.

For the implementations of the proxy component events, developers can choose between two implementation strategies: *synchronous* and *asynchronous* event notifications. The *synchronous* events are thread-safe and therefore allow the direct access to GUI elements. Admittedly, they decrease the program execution performance, because for all raised events the notification method invokes the event handler on the same thread as the target class. They ensure, however, that developers can access all class members from within the event handler methods without the need of implementing the thread-safe access to these members themselves. The *asynchronous* notifications on the other hand are executed on a separate thread, which results in higher performance, but they are not thread-safe. All event implementations of the Shared Phidgets toolkit are *synchronous* by default, for the reason that developers do not have to care about the thread-safe implementation by themselves (with the cost of a slightly lower performance). Nonetheless, for all events with frequent updates of the notifications (e. g., sensor values, accelerometer changes), a second asynchronous event is implemented. This lets developers choose to use the asynchronous event notifications if they implement the thread-safe handler code by themselves.

The interface skins for the proxy components (cf. Subsection 5.1.3) are derived from the *.NET* `UserControl` base class that provides the basic container for encapsulated GUI components (illustrated in the lower part of Figure 5.8). Each interface skin implements a wrapper around one of the proxy components, and implements a graphical interface to monitor and control the hardware's properties. Interface skins implement the `IComponentSkin` interface that defines the access to the `BaseComponent` object that is wrapped by the skin. As mentioned earlier, Appendix B provides a complete list of the implemented .NET components and Appendix C a detailed overview of the available interface skins.

Overall, developers can extend the core functionality and implemented class components of the toolkit by deriving from the base classes and using the utility classes and methods provided by the toolkit. Further details about the development and extensibility can be found in the tutorials, the documented toolkit source code, and the development examples [Marquardt, 2008].

5.5 Chapter Summary

This chapter gave an overview of the Shared Phidgets developer library and the programming strategies that developers can use to implement distributed physical user interfaces. One the one hand, the hardware proxy objects, graphical skins, and the seamless IDE integration simplify the development for the average programmers that are unfamiliar with physical user interfaces. On the other hand,

the direct access to the shared data model, high-level events, and the extensibility of the class framework addresses the demands of expert developers.

As the handling of distributed hardware infrastructures can be troublesome, the next chapter introduces a collection of advanced utilities to assist developers with the monitoring of the infrastructure, simulating of hardware, and debugging of appliances.

CHAPTER 6

Development Utilities

The *developer library* introduced in the last chapter and the *runtime platform* described in Chapter 4 are fundamental components of the Shared Phidgets architecture. Nonetheless, the development of distributed information appliances raises further issues that the developers have to cope with, as *"a significant difficulty in debugging is the limited visibility of application behavior"* [Klemmer et al., 2004]. Therefore, the toolkit introduces utilities for the monitoring, controlling, and simulating of the distributed hardware; as well as for the management, reconfiguration, and debugging of developed information appliances. This chapter at first explains the concept of these high-level utilities, before the implementation details are explained at the end of the chapter.

6.1 Monitoring and Controlling Utilities

The toolkit utilities are useful for testing and debugging, and help the developers to observe the developed information appliances as well as all occurred events at runtime. With their discussion of the Papier-Mâché toolkit, Klemmer et al. [2004] stressed the importance of providing such feedback to users and developers: *"Providing visual feedback about the system's perception of tracked objects helped users compensate for tracking errors."* [Klemmer et al., 2004]. As in the previously mentioned visualisation (cf. Subsection 3.4.8) of the Event Heap infrastructure [Morris, 2004], these tools can provide valuable feedback of the system events to the developers. To facilitate the handling of the distributed infrastructure the following tasks should be supported by the toolkit utilities:

- Observing of the shared data model and the event forwarding between the networked components.
- Monitoring of the distributed infrastructure of hardware devices, as well as controlling the properties of the distributed hardware.
- Exploring of the specific hardware properties and capabilities. This can help developers to understand the hardware device's capabilities and limitations, and this in turn leads to decisions about which hardware is suitable for the desired functionality of the appliance.
- Monitoring and reconfiguration of the developed appliances. This can facilitate the testing and debugging of appliances, and the configuration adjustment to a changed hardware infrastructure.

Three fundamental *access layers* to the hardware and appliances have been introduced in the two previous chapters. First, it is possible to address the hardware by means of the abstract model in the shared data space (cf. Figure 6.1(a)). Second, the proxy objects of the developer library can be used to access the hardware API, with class properties, methods, and events (cf. Figure 6.1(b)). Third, the shared data model of all developed appliances allows the access to the structure of the composed information appliances (cf. Figure 6.1(c)). The developed utilities of the Shared Phidgets toolkit allow the exploration and access to these three abstraction layers. Subsequently, the specific utilities and their integration into the development process are described in detail.

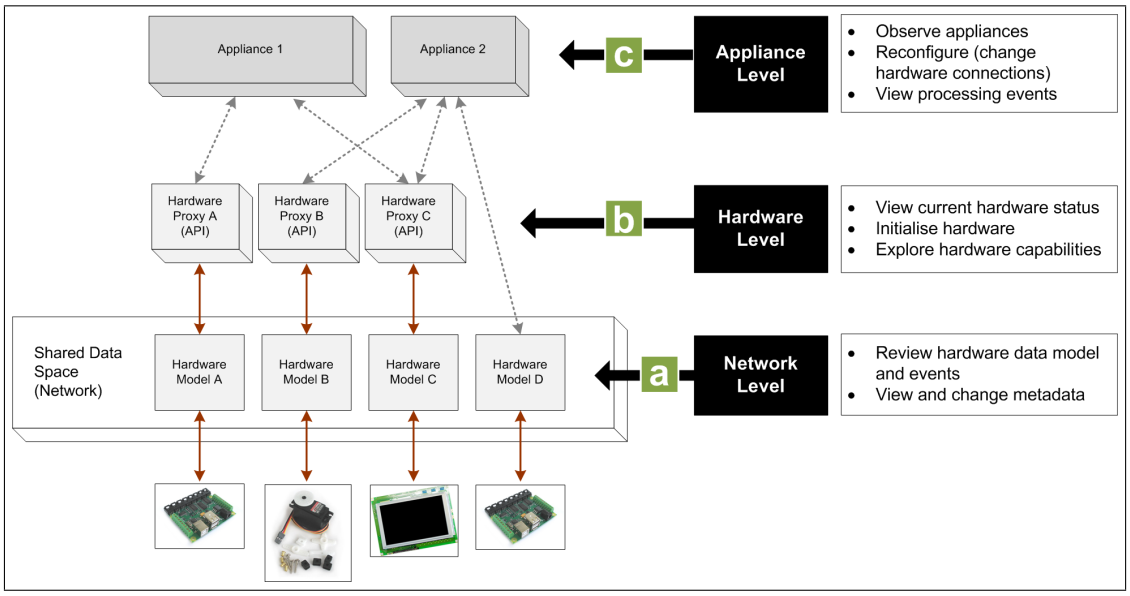


Figure 6.1: Three access levels to the shared infrastructure.

6.1.1 Network Level

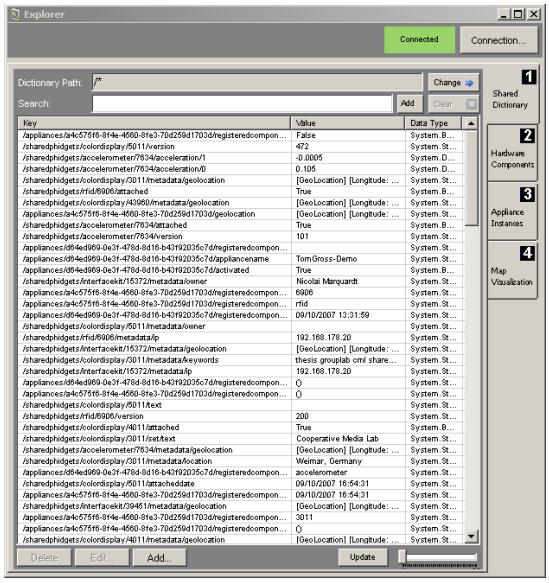
At the network level (cf. Figure 6.1(a)) developers can monitor the hierarchy of entries in the shared data space as illustrated in Figure 6.2(a). All entries of the shared data space are listed with their hierarchy path expression, their current value, and the data type of the value. This allows the observation of all occurred events (e. g., sensors changing their value). With this view to the shared dictionary developers are able to access the hardware data model directly. This represents the most abstract view to the infrastructure of network connected components and can be a helpful utility for experienced developers. For instance, this view allows the modification of all metadata entries of a hardware device, the modification of entries in the data model of a hardware device (e. g., for testing), and the addition or deletion of entries.

To facilitate the utilisation of this abstract view the following techniques are implemented. The list of entries can be *sorted* by path expression, values, or data type. Developers can easily *edit* the dictionary entries (with all supported data types), *add* new entries, or *delete* existing ones. It is also possible to *search* for all entries that match one or more key words. Once the key words in the search field are entered, the view changes automatically to display all matching results (search starts with the first letter that is entered). Multiple search terms are concatenated with the *AND* conjunction. The search automatically searches for matches in the path expression, the value, and the data type. This allows, for instance, finding all boolean metadata entries by entering the keywords *"metadata boolean"*, all digital inputs that are currently activated with *"input true"*, or the current location entries of all RFID readers with *"rfid location"*.

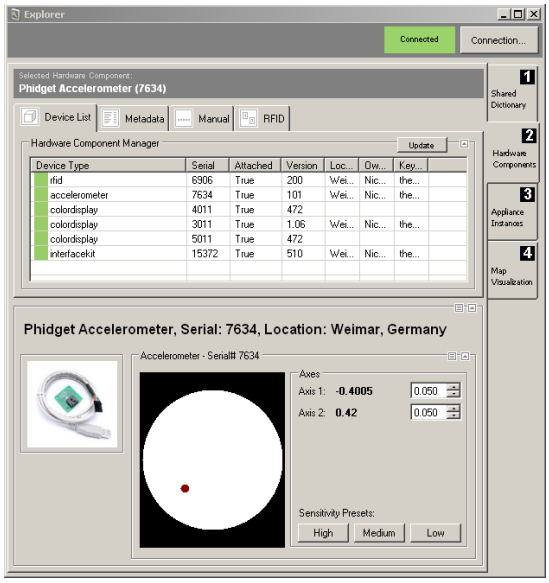
In summary, the access to the network level allows the direct view to the shared data model, and is the most powerful utility. The handling of the shared data model, however, requires advanced knowledge of the path hierarchy and the represented data model (cf. Subsection 5.1.1). Moreover, incorrect changes of the entries in the data structure might cause unpredictable behaviour of the infrastructure, for instance hardware devices that do not work correctly. Therefore, the following utilities allow a high-level view of the infrastructure.

6.1.2 Hardware Level

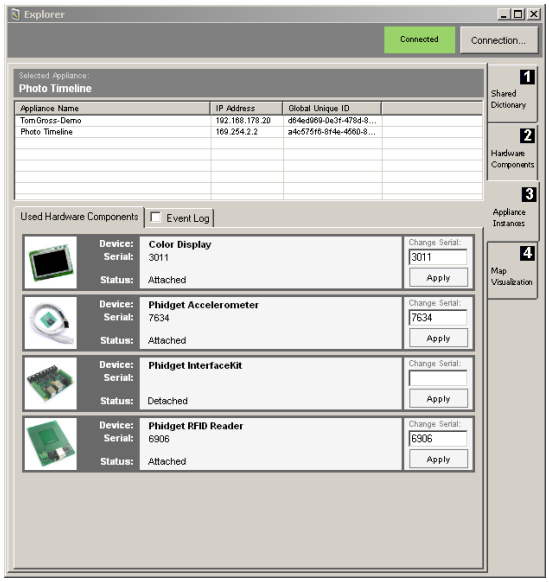
With the view to the hardware level of all distributed devices (cf. Figure 6.1(b)), developers can easily observe and control all hardware devices that are connected to any of the network machines. This helps viewing the current hardware status, initialising components (e. g., reset the position of a servo motor), and ex-



(a) Network level view.



(b) Hardware level view.



(c) Appliance level view.

(d) Geographical visualisation view.

Figure 6.2: Monitoring and controlling utilities.

ploring the hardware capabilities (e. g., to view the range of values a connected motion sensor triggers). Once a hardware device is selected the corresponding interface skin (cf. Subsection 5.1.3) is created and allows the view of all hardware properties and occurred events as well as the modification of the properties.

Figure 6.2(b) illustrates the view to an accelerometer and visualises the current positions of the accelerometer axis.

As it can be difficult to identify and to find the hardware a developer needs to control, the following options are available to explore all attached hardware devices:

- **Type List:**
  All attached hardware components are shown in a list view with information about the hardware device type, the serial number, and several metadata entries (e. g., location, owner).
- **Metadata:**
  The user can easily browse all devices based on the metadata entries in the shared dictionary. Users can also search for particular metadata entries, and see a list of all devices that match the search query.
- **Location:**
  All devices can be explored based on their location. This technique is described in more detail in Section 6.2.
- **Manual Input:**
  The user can manually enter the type and the serial number to identify the hardware.
- **Automatic Identification:**
  RFID tags attached to the hardware are used to identify the device type and serial number. Thus, users can identify all hardware devices by just placing them near an RFID reader.

The latter mentioned exploration technique is the easiest method to identify a hardware device. By attaching unique RFID tags to each of the hardware components, and by associating this RFID tag to the hardware component's dictionary data model, all components can be easily identified by bringing an RFID reader near them. These associations of the physical components with their digital representation are inspired by Want et al. [1999] research about adding identification tags to objects in the environment to facilitate the interaction with these objects. Therewith, the RFID tagged hardware can be more easily identified as the user does not need to know the device type or serial number.

In summary, the access to the hardware level allows the observation and control of connected devices via the graphical interface skins. By using the integrated search methods the needed hardware can be easily found and the applied RFID tagging of the hardware moreover simplifies the hardware exploration.

6.1.3 Appliance Level

As the hardware components can be combined to logical units (the *appliances* introduced in Subsection 5.2.1), the utility access to the appliance level (cf. Figure 6.1(c)) allows the exploration of all available appliances. Here, the addressed hardware components can be viewed as shown in Figure 6.2(c). This control view also allows changes of the *mapping* between hardware and appliances. That is important to let users modify appliances at runtime and adapt appliances to changes of the infrastructure like different attached hardware devices.

Furthermore, all events of the used hardware and the appliance itself can be viewed: these are all *incoming sensor events*, *outgoing actuator control events*, and the *high-level events* of the appliance. This helps to monitor the internal processing of an appliance. Nonetheless, it is still difficult for developers to monitor an appliance with this event view, which motivated the development of the advanced spatial visualisation of the overall infrastructure (cf. Figure 6.2(d)) that is introduced in the following section.

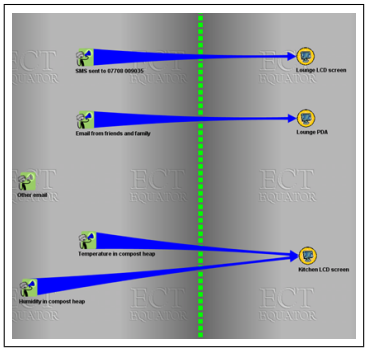
6.2 Revealing the Invisible: Advanced Spatial Visualisation

The distribution of the hardware sensors and actuators at remote located spaces (e. g., different rooms or buildings) makes it difficult for developers to observe and control the distributed hardware devices. Although the three earlier introduced utilities allow the control of separated aspects of the infrastructure (i. e., shared dictionary, hardware devices, appliances), they do not provide an easy to use interface to observe appliances, get insights into the transmitted events, or view the relations between the distributed components of the appliances. Therefore, the next section of this chapter introduces an advanced visualisation that combines the functionality of the previous introduced utilities with a visualisation of the appliance networks in their geographical context.

6.2.1 Overview

Overview visualisations of the assemblies of hardware devices to appliances enable the developer to easily monitor the incoming and outgoing connections of an appliance. Furthermore, it allows changing the mapping of the hardware devices

(a) Orbital browser user interface [Ducheneaut et al., 2006].



(b) Composing UI of the Equator Component Toolkit [Egglestone et al., 2006].

(c) SenseWeb map visualisation [Kasal et al., 2007].

Figure 6.3: Visualisations of distributed hardware devices.

to specific appliances. For instance, if an appliance is observing the activity in several rooms by the use of motion sensors, it is possible to change these addressed sensors at runtime. This technique is previously used in graphical programming environments of ubiquitous computing applications, as for instance in the editor user interfaces of the iStuff mobile project by Ballagas et al. [2007], the CollaborationBus project [Gross and Marquardt, 2007]), the Orbital Browser shown in Figure 6.3(a) [Ducheneaut et al., 2006], and the ECT user interface illustrated in Figure 6.3(b) [Egglestone et al., 2006]). These visualisations reveal the internals of the appliances and allow the control of the data flow between the incoming sensor values and the outgoing control values. Usually, these utilities integrate a graphical programming environment where end users can program and configure information appliances. Although all of these systems address end users, the principles of these visualisations can support developers as well.

Therefore, the advanced geographical visualisation of the Shared Phidgets toolkit provides developers of distributed physical interfaces similar visualisations of the configuration and *internal data flow* of the developed appliances. This data flow includes all events that are transmitted between the assembled components of the appliance. The visualisation combines the monitoring view of the appliances and the hardware with the visualisation of the geographical context. Similar to the geographical visualisations that have been previously utilised for the visualisation of large scale sensor networks (e. g., the SenseWeb visualisation in Figure 6.3(c) [Santanche et al., 2006; Kasal et al., 2007]) these geographical visualisations can also support the development of distributed physical interfaces. The following list gives an overview and the rationale for the integration of the previously men-

tioned aspects into the advanced geographical visualisation that is illustrated in Figure 6.4:

- **Geographic Context:**
  The geographic context (e. g., rooms, buildings, cities) is important for developers as they can analyse the distributed infrastructure based on the physical location of the components.
- **Metadata and Regions:**
  With defining regions and applying metadata developers can group hardware devices, and add high-level context information to the environment (e. g., private areas in buildings, where sensors are inactive or hidden). Developed appliances can utilise these metadata information (as introduced with the example case study in Subsection 7.1.1), which simplifies the development, and allows the dynamic runtime configuration of appliances.
- **Hardware Devices:**
  This layer visualises all sensors and actuators at their current location. Therewith, it is easier to find hardware devices that are nearby located to others and to find sensors at a specific location.
- **Appliances and Connections:**
  The addressed hardware components of an appliance are visualised as a network of connections. At a glance, the connections of appliances can be observed. It is important that the visualised connections represent the *logic control network*, not the underlying *physical network connections*.
- **Events and Data Flow:**
  The visualisation of occurred events can be beneficial for the debugging of appliances. All incoming events of sensors, outgoing control events for actuators, transmission of metadata, and generation of high-level events are visualised as animations on top of the drawn connections of the previous layers. For instance, developers can verify that a hardware component is working and transmitting events or that an appliance is sending actuator control commands in response to incoming sensor values.
- **Detailed View and Control:**
  Finally, detailed views and control windows for the visualised components (hardware, appliances) are available if required.

The integration of these mentioned visualisation layers follow the *Visual Information Seeking Mantra* by Ben Shneiderman: *"Overview first, zoom and filter, then details-on-demand"* [Shneiderman, 1996]. The *overview and context* in the visualisation is provided by the geographical map layer. Users can *navigate and zoom in* and select the hardware components and appliances they are interested in (*filter*).

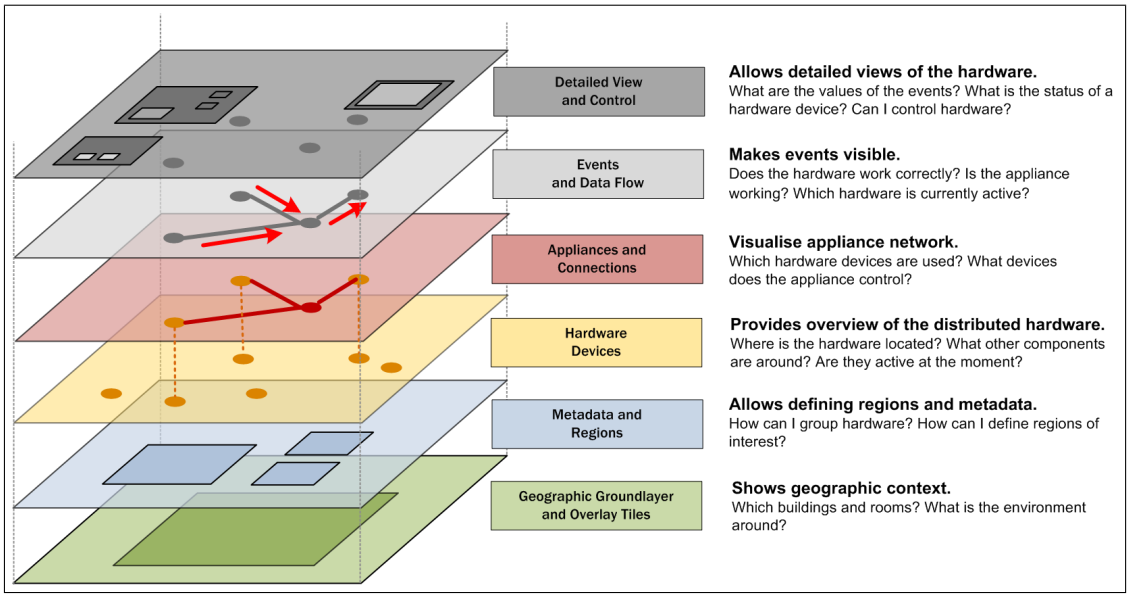


Figure 6.4: Layer architecture of the spatial infrastructure visualisation.

Finally, they can request *details-on-demand* about the hardware devices and appliances, with the floating window panels that can provide the interface skin, data plots, shared dictionary events, or metadata views. The next section describes the user interface of this visualisation tool.

6.2.2 Using the Infrastructure Visualisation

This section briefly describes the user interaction with the Shared Phidgets toolkit infrastructure visualisation. When the utility is started, the geographical map view is shown, and the developer can navigate and zoom in to display a particular region. Hardware devices are visualised at their spatial location, as with the orange markers in Figure 6.5. If the location of a hardware device is not specified (i. e., it was not automatically added by an GPS device or with RFID identification) the developer can manually specify the location on the map. For status information about the hardware, the developer can click on the hardware marker on the map to see the status panel of this device that is shown in Figure 6.5(a). Here, the developer can activate the detailed views of the hardware device's properties (previously described in Subsection 5.1.3): the *interface skins* as shown in Figure 6.5(b), the view of the hardware's *abstract data model* entries that can be seen in Figure 6.5(c), and the *graph plot views* of sensor values that is visualised in Figure 6.5(d). These information windows are floating panels on top of the map, and they can be collapsed in order to not occlude parts of the map (cf.

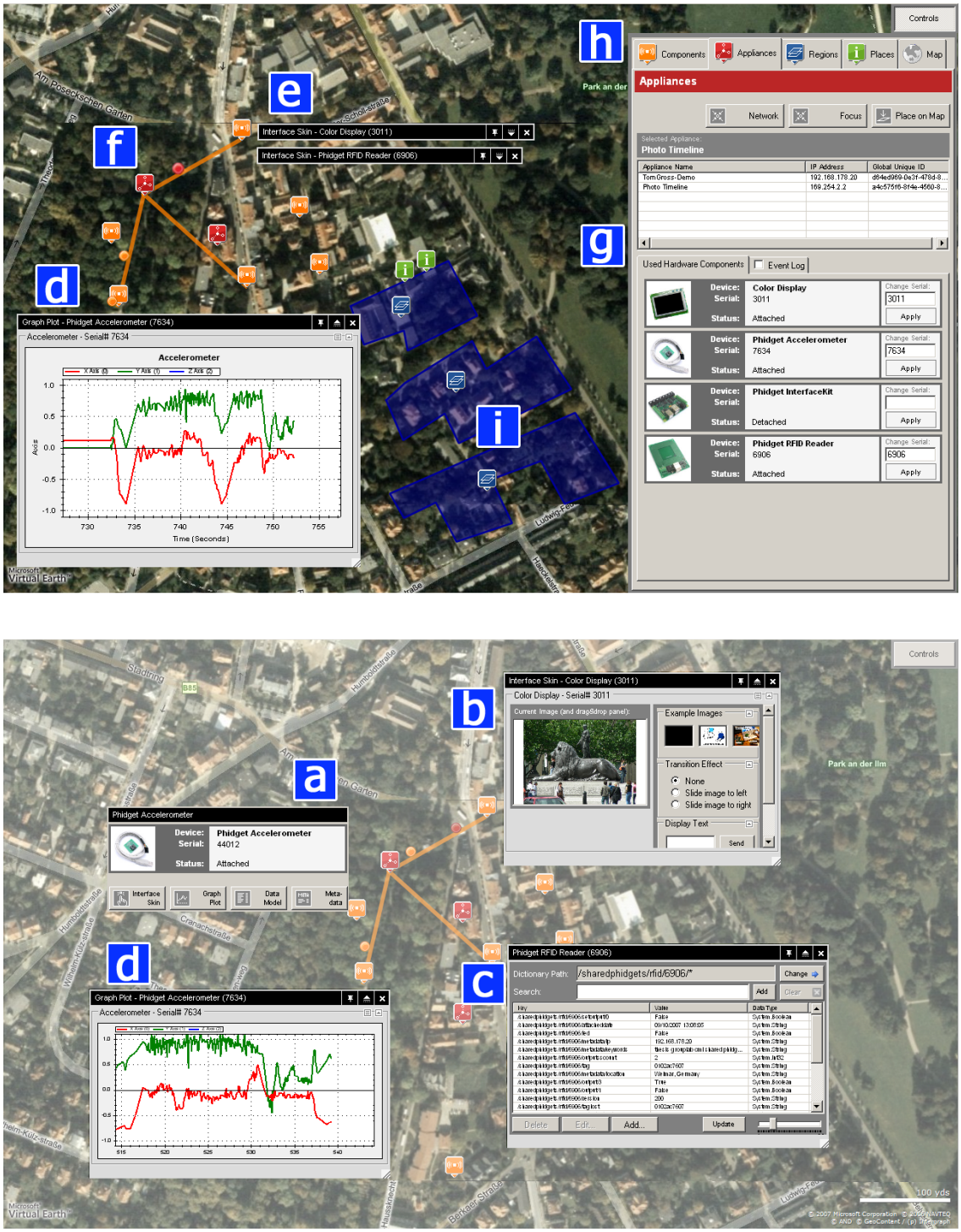


Figure 6.5: User interface of the spatial infrastructure visualisation.

Figure 6.5(e)). Their position is linked to the current map location (this means that they automatically follow this map location when the map is moved). The preference of the location can be, however, set to a fixed screen location (e. g., to retain the detail views of a sensor on the screen while the map is moved).

Appliances are visualised as the network of connected hardware components, as shown in Figure 6.5(f). Sensor events and actuator control events are displayed as animations on top of the drawn connections, so that developers can easily recognise these incoming and outgoing events. The detailed view of appliances (shown in Figure 6.5(g)) is available in the map control window (cf. Figure 6.5(h)). This appliance view is similar to that introduced in Subsection 6.1.3.

With the configuration window shown in Figure 6.5(h) the developer has access to detailed information about all hardware devices, appliances, map options, and metadata regions. The latter are the geographical regions (the blue areas in Figure 6.5(i)) that a developer can draw on the map and specify metadata information for these regions (e. g., context information about this area). The metadata entries can then be assigned to all hardware devices that are located inside of the area covered by the region. Therewith, this is a helpful functionality to apply metadata tags to groups of devices and to modify the metadata entries of hardware devices depending on their geographical location.

In summary, developers can use the spatial visualisation utility to monitor and control hardware *in place* and get detailed information about the *internals* of the developed appliances. To further support the debugging and testing the next section introduces the simulation utilities of the toolkit.

6.3 Testing with Simulated Hardware

Testing of the developed physical interfaces is an integral part of the development process [Klemmer et al., 2004; Dow et al., 2005; Dey, 2000; Sohn and Dey, 2003]. In this section the integration of simulation interfaces into the toolkits architecture is described. This facilitated the process of testing and debugging appliances during development.

6.3.1 Wizard of Oz Simulations

Wizard of Oz (*WOz*) simulations describe a technique where developers can simulate the input events for developed applications. Therewith, the testing of appli-

cations is possible even if the current input device (e. g., sensor hardware, input controller) is not available [Li et al., 2007].

A common application of the *WOz* method is the simulation of a part of the application that is not yet implemented; *"from simulating the entire system to simulating sensors"* [Dow et al., 2005]. Dow et al. [2005] describe the application of *WOz* simulations throughout the prototyping cycle. Their developed DART system illustrates the integration of *WOz* tools into an event-based architecture. With these controls the developers can easily simulate the occurrence of events by using controls of a graphical user interface (that is automatically generated based on the event types). Klemmer et al. [2004] also added the *WOz* functionality to their Papier-Mâché toolkit. With these *WOz* controls developers can simulate events (in this case, events from computer vision or an RFID reader) for the input of the TUI prototyping toolkit. The local Phidgets toolkit [Greenberg and Fitchett, 2001] also included a simulation mode; here, the simulation control is activated in the case that the corresponding "real" hardware is not available.

In summary, the *WOz* simulations support the prototyping process in the following ways. First, they can simulate hardware that is not available or remotely located [Dow et al., 2005]. Second, it is possible to test applications and to reproduce scenarios during the development and debugging [Klemmer et al., 2004]. Third, they can even simulate applications and high-level events that are not yet implemented [Dey, 2000]. The following subsection describes the integration of these techniques into the Shared Phidgets toolkit.

6.3.2 Toolkit Simulation Utilities

To support the testing and debugging of distributed information appliances with the Shared Phidgets toolkit, the toolkit includes simulation utilities and *WOz* simulations as well. These simulations have the following characteristics. First, all simulations are built on top of the network data model. Therewith, they can provide simulations based on the hardware data model in a way that they simulate the entries of this hardware in the shared dictionary. This means that they add all corresponding key/value pairs to the dictionary and respond to the changes of these entries similar to the real hardware component. Second, the *WOz* simulations provide a graphical user interface that provides user controls to simulate the hardware behaviour (e. g., sensor changes). Third, to allow the testing of applications that utilise larger number of sensors, the utilities provide a method to record and playback a series of events (similar to a macro recorder for GUI software). This utility can record any event that occurs in the shared data model and saves a series of events with their according time intervals.

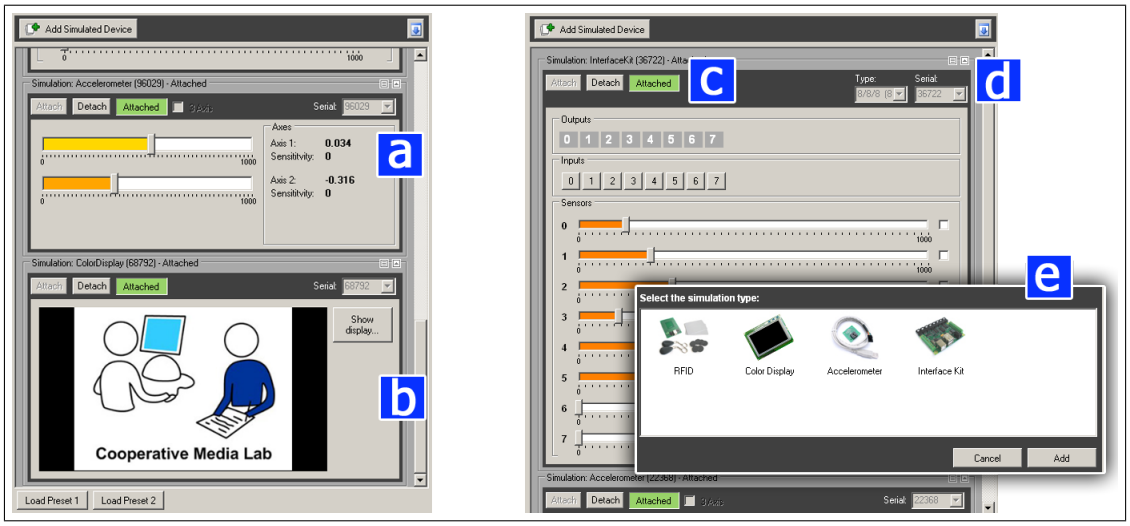


Figure 6.6: Simulating hardware with Wizard of Oz interfaces.

The following simulation functionality is available in the Shared Phidgets toolkit:

- **Wizard of Oz Hardware Simulations:**
  The developers can create hardware simulations that provide similar attributes and events like the physical hardware components. For instance, the developer can create simulated RFID readers and set the received ID tags of the reader. All these simulations are managed by the `SimulationManager` utility (cf. Figure 6.6). They provide a GUI control for interaction (e. g., simulate the acceleration changes of an accelerometer in Figure 6.6(a) and a graphic LC display in Figure 6.6(b)). Each simulated hardware interface provides controls to set the hardware as attached/detached (cf. Figure 6.6(c)) and to specify the serial number (cf. Figure 6.6(d)). The `SimulationManager` contains all the simulation GUI controls so that the developers can easily add simulated devices (cf. Figure 6.6(e)).

- **Recording and Playback of Network/Hardware Events:**
  When developers prototype applications that involve many distributed sensors, it can be difficult to test the developed applications even with the graphical *Wizard of Oz* interfaces. For this case, the toolkit provides an easy to use recording and playback utility, the `SimulationRecorder` illustrated in Figure 6.7. This utility can record events that occur in the shared dictionary (e. g., sensor events, actuator changes) and playback this sequence of events later for testing. For instance, the developer can record the events of various motion, light, and distance sensors, and playback these events later while testing the developed application to verify the application's reactions.

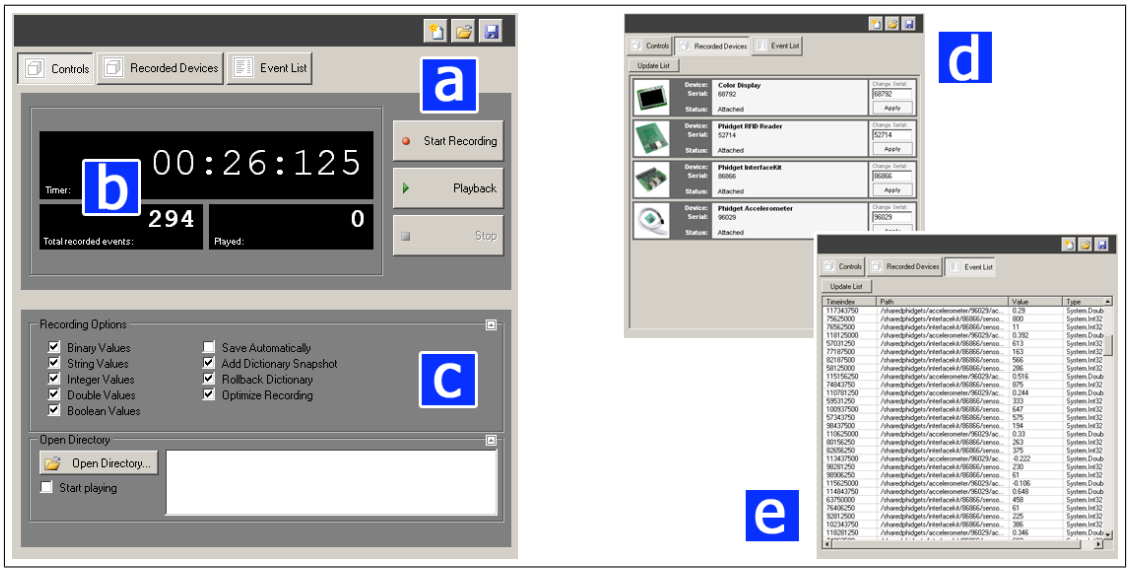


Figure 6.7: Recording and reproducing hardware events.

The user interface of the `SimulationRecorder` provides controls to record and playback a simulation recording (cf. Figure 6.7(a)). It also includes a timer for the recorded time and total number of events (cf. Figure 6.7(b)). Options (cf. Figure 6.7(c)) are available to specify the type of the events that are recorded (e. g., only network events of a specific data type). Once a simulation is recorded the GUI displays a list of all recorded devices (cf. Figure 6.7(d)) with the option to change the serial number of the devices. Furthermore, a table view provides a list of all recorded events (cf. Figure 6.7(e)) to review the recording and apply changes to the dataset.

With these utilities developers are able to easily simulate hardware devices that are currently not available or they can record *test cases* of a series of hardware events, and recall them later when testing and debugging applications.

6.4 Scenario

The following scenario illustrates the utilisation of the previously introduced development utilities. Chris develops an appliance that visualises the activity of a working team in distributed office rooms to provide *presence awareness* between the remote located group members. The appliance avoids using cameras because of privacy considerations. Instead it uses *motion* and *IR distance* sensors to measure the activity in the rooms.

When beginning the project, Chris starts the infrastructure visualisation of the `Explorer` software and immediately gets an overview of the available distributed sensors in the offices. He selects the interesting sensors and chooses the graph plot views to monitor the sensor values. This helps him to implement the interpretation logic of the sensor values in his appliance code as the graph plots show the changes of the sensors over time. The observation of the sensors and the view of their location in the environment also helps Chris to decide which sensors to include. He decides to use metadata entries to tag all sensors his application should observe. These metadata entries are easily added to the sensors with the list view of the *shared data model* and the developed appliance subscribes to receive notifications of these metadata entries.

When the implementation is finished, Chris starts the compiled appliance control software. He can easily monitor the appliance with the infrastructure visualisation as a network between all utilised hardware components. As no others are in the offices at that time Chris starts the `SimulationManager` to quickly simulate events of the motion and distance sensors in the *WOz* manner. During this simulation the sensor events are shown on the map view. With the interface skin views of the controlled actuator displays, Chris verifies that his developed appliance is working correctly.

Although this is only a simple scenario, it demonstrates how the implemented utilities can support the development process at various stages: *finding sensors and actuators* when beginning with the development, *monitoring* concrete sensor values, easily *integrating metadata* information, *observing* appliances, and *testing* with simulated sensors. The remainder of the chapter now explains the implementation of these utilities.

6.5 Implementation

This section briefly describes aspects of the utility implementation[1]. All utilities of the Shared Phidgets toolkit are built in *C#.NET* using the developer library that was introduced in Chapter 5. For the implementation they utilise the software building blocks of the developer library (e. g., shared dictionary access, appliance observer, hardware proxy objects, interface skins).

1 Details of the implementation can be found in the developer documentation and the source code; available on the thesis project DVD or as download on the Shared Phidgets project website [Marquardt, 2008]

Monitoring and Controlling Utilities

The core functionality of the monitoring and controlling utilities has been implemented as separate *.NET* `UserControls`. Therefore, these controls can be easily added to custom software projects. This includes the following controls. First, the `DictionaryManager` provides a list of dictionary entries, search functionality, and commands to add, edit, or remove dictionary entries. Second, the `DeviceManager` implements the view of the available hardware devices. Finally, the `AppllianceManager` implements the interface for monitoring and reconfiguration of appliances. The utility functionality is compiled as the `Explorer` application, and also includes the geographical infrastructure visualisation.

Infrastructure Visualisation

The `MapExplorer` application is the implementation of the spatial infrastructure visualisation of the Shared Phidgets toolkit. It includes the integration of the Microsoft Virtual Earth web service [Microsoft Corporation, 2007c]. Virtual Earth provides satellite imagery and street layout information. Moreover it allows the integration of custom data and information layers with the access to the API of the Virtual Earth SDK [Microsoft Corporation, 2007a]. The Virtual Earth web service is integrated into the *C#* application with a web browser control that opens an HTML/JavaScript website. The implemented JavaScript code acts as a bridge between the *C#* application and the Virtual Earth web service [Microsoft Corporation, 2007b]. This implementation allows the usage of all components and interface skins implemented in the Shared Phidgets developer library (including all custom controls created by developers), and the integration of these controls as the floating observer and control windows into the application. Therefore, the visualisation tool utilises the Virtual Earth web service from within *C#* but is not implemented as web service itself.

As the default resolution of the Virtual Earth satellite images is only around two meters per pixel, the *Map Cruncher* utility [Microsoft Research, 2007] is used to create high resolution overlay maps. This tool calculates an overlay map file based on specified coordinate marker points on custom map images that define the absolute location of the custom map in relation to the Virtual Earth map layer. It then generates a layer description XML file that can be integrated into the application from within the JavaScript code. Currently, a predefined set of custom layers is added to the visualisation; however, a dynamic loading method of the layer description files could be added in a future release of the toolkit.

The implementation of the spatial visualisation is illustrated in Figure 6.8. The described Virtual Earth maps and the additional geographic layers (created with the Map Cruncher tool) are the base layers of the visualisation. Additional layers

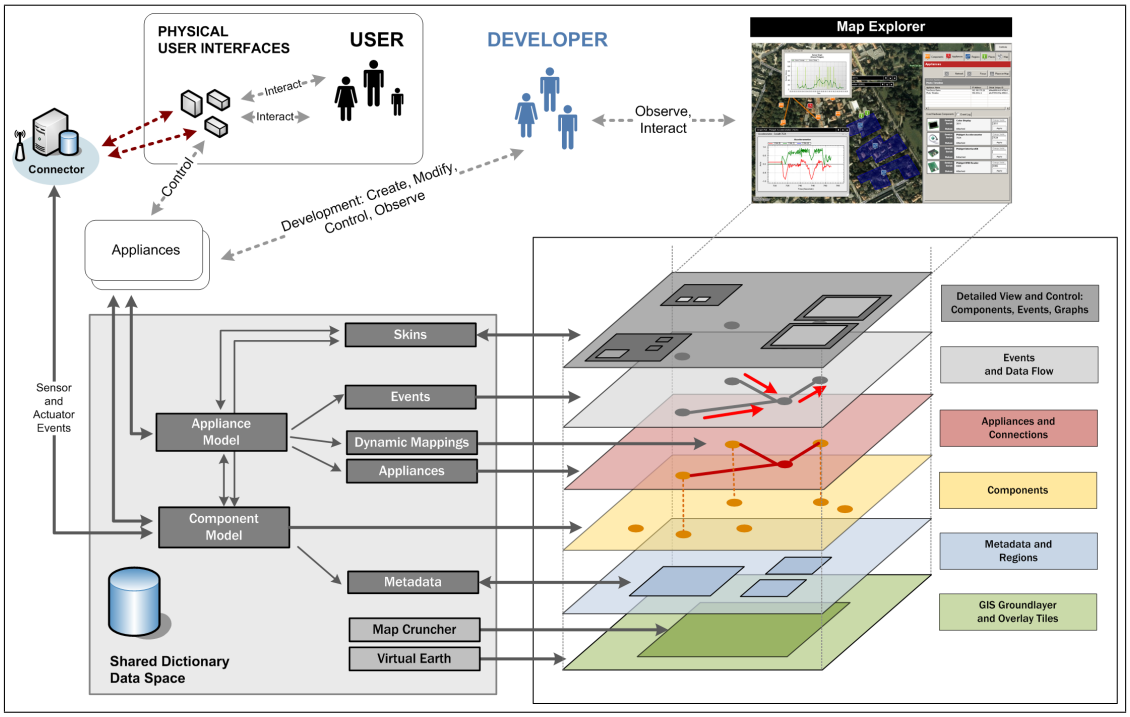


Figure 6.8: Implementation of the spatial infrastructure visualisation.

are the metadata regions and the hardware components. To visualise the appliance networks, the utility monitors the model of the appliances in the shared data space (by using the `ApplianceObserver`) and draws connections of all hardware components to the appliance location. If the appliance has no specified location, it is automatically positioned in the centre of all utilised hardware components. The utility also subscribes for all events of the sensors and actuators, in order that all incoming and outgoing events can be visualised.

Simulations

The *WOz simulation interfaces* implement the user controls of simulated hardware devices. They are built as *.NET* `UserControls` and implement the `ISimulation` interface. The `SimulationBaseComponent` class provides a wrapper for the access to the shared dictionary. This class has a set of methods to easily add the entries of the simulated hardware to the shared dictionary and to receive notifications when values of the simulated hardware data model have changed. Five reference implementations for a *WOz* simulation interface have been implemented: the `SimulationAccelerometer`, `SimulationGraphicLCD`, `SimulationInterfaceKit`, `SimulationGSM`, and `SimulationRFID`. Each of these classes includes a set of user controls that visualise the status of the simulated device and allow changes of the simulated device. For instance, slider controllers are added to the `SimulationAc-`

celerometer to allow the simulated change of the current axis positions; or the SimulationRFID interface allows drag and drop selection of the simulated detection of RFID tags. For the development of custom *WOz* interfaces, developers can use the code framework of one of the five reference implementations as a starting point.

The SimulationRecorder implements the functionality to record sensor and actuator events from the infrastructure. It uses subscriptions for shared dictionary events to store these entries in a hashtable data structure. This data structure can be serialised/deserialised to save/load the event collection to/from the local computer.

6.6 Chapter Summary

This chapter introduced the collection of utilities that facilitate monitoring and controlling of the distributed hardware, as well as the testing and debugging of developed appliances. Furthermore, an advanced visualisation utility provides deeper insights into the interconnections between all distributed hardware devices and the appliances. Finally, the integrated simulation utilities allow the Wizard of Oz simulation of hardware, as well as the recording and playback of event series for testing and debugging. The next chapter introduces example appliances that use the *runtime platform*, are implemented by means of the *developer library*, and are monitored and tested with the *utilities* introduced in this chapter.

CHAPTER 7

Case Studies and Discussion

In this chapter, the applicability of the toolkit for the prototyping of distributed physical interfaces is evaluated. The example case studies illustrate how developers can use the toolkit for the rapid prototyping of user interfaces involving various distributed sensors and actuators. For each of the case studies the important aspects of the implementation are explained. Thereafter, the implementation and limitations of the toolkit are discussed. The chapter concludes with a summary.

7.1 Appliance Case Studies

This chapter begins with a discussion of three case study implementations that illustrate the applicability of the toolkit. Previous toolkits in the related work have been typically evaluated by the spectrum of prototype systems they support, as well as their support for an easy reimplementation of interactive systems from the literature [Klemmer et al., 2004; Greenberg and Fitchett, 2001; Ballagas et al., 2003, 2007]. Therefore, the appliance case studies address various ubiquitous computing scenarios, and partially reimplement ideas of systems that were introduced earlier in Section 2.6.

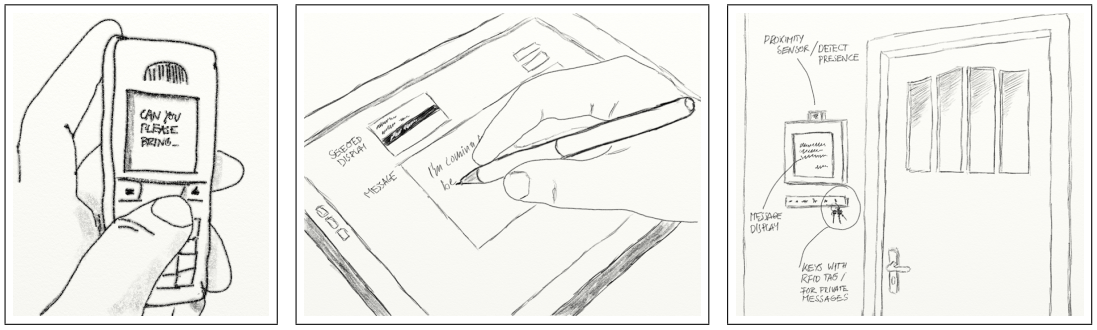
7.1.1 Location-based Messaging

The following prototype implements a system that allows users to send messages to displays that are distributed at various locations. It integrates ideas and concepts of previous research projects, like the situated displays of *HomeNote* [Sellen et al., 2006], the *Sticky Spots* system [Elliot et al., 2007], and the *Gate Reminder*

[Kim et al., 2004]. The motivation behind this example is to illustrate the rapid prototyping of such a location-based messaging system, and to highlight possible extensions of the first prototype.

Implementation

The prototype[1] implements the following functionality: users are able to send messages to situated displays from their mobile phones by using text messages, as illustrated in Figure 7.1(a). Alternatively they can use a graphical software front end (cf. Figure 7.1(b)) to write or draw messages and choose a display from a list to send the message to (similar as in the *Sticky Spots* project [Elliot et al., 2007]). The message displays are implemented with the wireless graphic LCD, and can be easily distributed across multiple rooms in the home; for instance near the front door as illustrated in Figure 7.1(c). The software organises the routing of messages to the graphic LC displays. Thus, this system implements a first iteration of a prototype that allows situated messaging.



(a) Sending messages from phones. (b) Writing messages. (c) Situated messaging display.

Figure 7.1: Location-based messaging appliance.

In the following, three implementation aspects of the system are briefly described: the *dynamic integration of distributed displays*, the *handling of incoming text messages*, and the *forwarding of messages* to displays. The system *dynamically integrates new connected graphic LC displays* into the application by using the `Attach` event of the `DeviceManager`. The event handler method checks for the correct device type (line 3 in Figure 7.2) and the specification of two metadata tags: *appliance* and *location* (lines 6 and 7 in Figure 7.2). The `appliance` metadata entry tags all hardware devices that are assigned to this appliance, while the *location* metadata entries specify the location of the display (and these names are added to

1 Development project: `LocationBasedMessaging` [Marquardt, 2008].

a list of the appliance user interface). Next, the *incoming text messages from mobile phones* are handled via an event callback method of the `ReceivedSms` event of the `GSMGateway` (lines 13 to 19 in Figure 7.2). In the event handler, the first part of the messages is interpreted as the destination location for the message. The remaining part of the messages is the text for the display. Finally, the `SendMessage` method (lines 21 to 24 in Figure 7.2) forwards messages to the displays by changing the according entry of the abstract data model of the display in the shared dictionary.

```
1  private void manager_Attach(object sender, DeviceManagerEventArgs e) {
2    // Check device type for attached graphic LC displays.
3    if (e.DeviceDescription.Type == Constants.DEVICE_GRAPHICLCD) {
4      // If the location property of the display is set, add the display
5      // to the device list.
6      if (e.DeviceDescription.MatchMetadata("appliance", "stickyspots") &&
7          e.DeviceDescription.ContainsMetadataKey("location"))
8        this.listViewLocations.Items.Add(new ListViewItem(new string[] {
9          e.DeviceDescription.GetMetadata("location"),
10         e.DeviceDescription.SerialNumber }));
11 } }
12
13 private void gsmGateway_ReceivedSms(object sender, SmsEventArgs e) {
14   // Event handler for incoming message from a mobile phone
15   string[] elements = e.Message.Split(':');
16   if (elements.Length > 1) {
17     this.SendMessage(elements[0].Trim(),
18       e.Message.Substring(elements[0].Length + 1).Trim());
19 } }
20
21 private void SendMessage(string location, string message) {
22   [...]
23   // Set the text message of the display
24   this.sharedDictionary[devicepath + "/set/text"] = message;
25 }
```

Figure 7.2: Source code for location-based messaging appliance.

Discussion

Although this prototype system does not implement all the functionality of the mentioned location-based messaging systems in the related work, it demonstrates that a working prototype can be built with the Shared Phidgets toolkit by programming only 26 lines of code for the implementation of the application logic.

The software implementation of the prototype illustrates how event handlers (e. g., to get notifications of attached display devices or incoming SMS text messages) and metadata information (e. g., for assigning devices to the appliance, and for location information) can be efficiently applied. The metadata entries allow the dynamic integration of additional displays at runtime. It is only necessary to

attach a new display to one of the client computers, and to add the two metadata entries *location* and *appliance*, for instance by using the utilities of the toolkit (cf. Section 6.1).

The prototype can be easily extended with additional functionality. For instance, the system could support sending private messages for a particular person to the situated displays (by using RFID tags for identification). It could also implement the reminder functionality of the Gate Reminder system [Kim et al., 2004], by adding additional sensors to gather context information (e. g., people moving by, or standing in front of the display).

In summary, this prototype illustrated the application of the toolkit to build a distributed information appliance. A few aspects of the software implementation highlighted the easy to use API with proxy objects for the addressed hardware. The rapid prototyping technique of the Shared Phidgets toolkit allows developers to focus on the design of the appliance, instead of dealing with low level implementation issues.

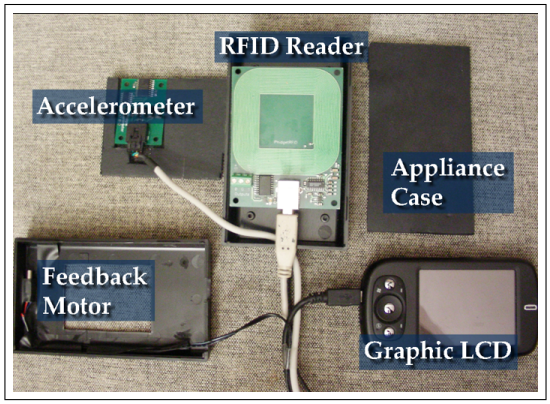
7.1.2 Tangible Digital Media

The second example case study explores different ways of the user's interaction with digital media. The intention of the developed appliance is to allow the coupling of objects in our everyday environment—the *physical world*—with information from the *digital world* [Ishii and Ullmer, 1997]. With the developed prototype users can browse through collections of digital photos with a small tangible controller and assign these photos to digital photo frames distributed in the environment. Therefore, this prototype allows the tangible exploration of digital information in the user's everyday environment [Fitzmaurice, 1993].

Implementation

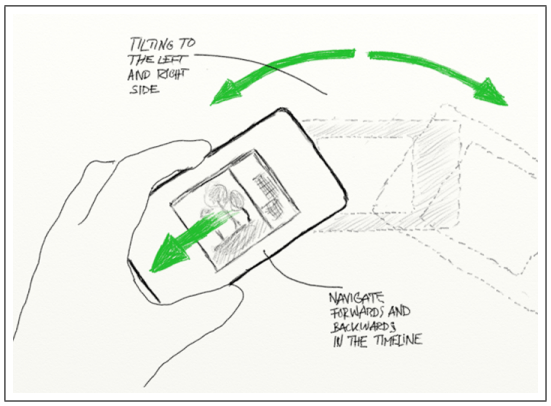
The developed appliance[2] consists of a hardware unit with a graphic LC display, an RFID reader, and an accelerometer (illustrated in Figure 7.3(a)). Furthermore multiple additional graphic LC displays work as distributed digital image frames as in Figure 7.3(b). To support the development of the appliance, interface skins for all these hardware devices are added to the software-side user interface as illustrated in Figure 7.4(a).

2 Development project: `PhotoTimeline` [Marquardt, 2008].

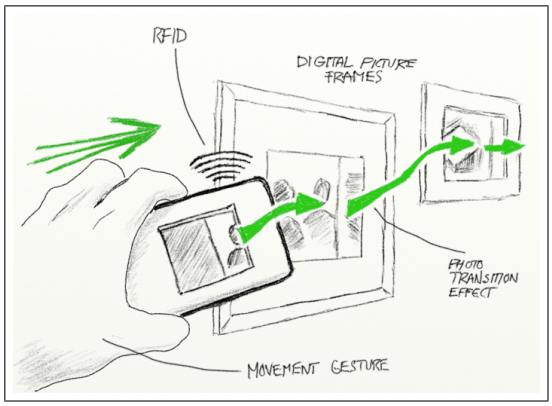


(a) Assembly of the information appliance.

(b) Appliance assigning photos to digital picture frames.



(c) Navigate through photo collections.



(d) Distribute media with simple gestures.

Figure 7.3: Tangible digital media appliance.

When browsing the collection of digital photos, the controller allows users to move forward or backwards in time by just tilting the small screen to the left or the right side (cf. Figure 7.3(c)). This method is inspired by the interaction technique for small displays introduced by Rekimoto [1996] and Harrison et al. [1998]. They have added solid state gyro sensors to a small display and presented example applications for selecting menus or navigating a map by tilting the small display screen. The prototype built with the Shared Phidgets toolkit is implemented with a similar technology. A Phidget accelerometer (attached to a graphic LCD) is used to determine the current orientation of the display. The tilting of the device controls the scrolling through the timeline of the digital photo collection (illustrated in Figure 7.3(c)). The visualisation of the accelerometer status with the interface skins in Figure 7.4(b) supports the integration of this sensor into the appliance control software. The `AccelerationChange` event handler notifies the appliance software of orientation changes of the accelerometer. The event han-

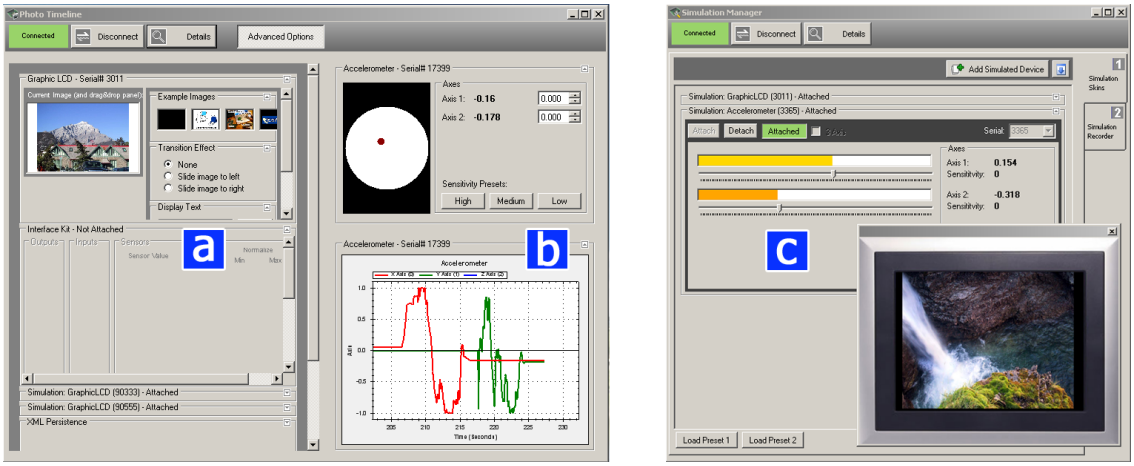


Figure 7.4: Appliance control user interface and simulations.

dler method observes if the value passes a certain threshold (lines 7 and 10 of Figure 7.5). If the value is above the threshold, the displayed image is changed by specifying the transition and assigning the new image of the digital image collection (lines 8/9 and 11/12 of Figure 7.5). The simulation utilities shown in Figure 7.4(c) can be used to easily create test cases to evaluate and debug these methods.

```
1  private void accelerometer_AccelerationChange
2    (object sender, AccelerationChangeEventArgs e) {
3    // Only proceed if the value of the first axis of the accelerometer is changed
4    if (e.Index == 0) {
5      // If the value is higher/lower than the threshold, set the transition
6      // of the display and change the displayed image
7      if (e.Acceleration < -0.5) {
8        this.graphicLCD.Transition = GraphicLCD.TRANSITION_RIGHT;
9        this.graphicLCD.Image = this.imageCollection.GetNextImage();
10     } else if (e.Acceleration > 0.5) {
11       this.graphicLCD.Transition = GraphicLCD.TRANSITION_LEFT;
12       this.graphicLCD.Image = this.imageCollection.GetPreviousImage();
13 } } }
```

Figure 7.5: Source code for the navigation through an image collection.

Furthermore, the RFID reader attached to the display and accelerometer is used to identify objects that are nearby; for instance the digital photo frames, or tagged digital artefacts or photo printouts. The `TagManager` notifies the appliance if new tags are found. On the one hand, if the associated information of this tag represents a digital image file, this image is shown on the mobile display (line 3 and 4 of Figure 7.6). On the other hand, if the associated information of the tag represents a graphic LC display of a digital photo frame (lines 8 and 9 of Figure 7.6),

the current selected image of the appliance unit is transferred to this digital photo frame as illustrated in Figure 7.3(b).

```
1  void tagManager_FoundTag(object sender, FoundTagEventArgs e) {
2    // If the RFID tag is associated with a file: display the image
3    if (e.TaggingType == TaggingType.File)
4      this.graphicLCD.Image = this.imageCollection.GetImage(e.Value);
5
6    // If the RFID tag describes a hardware device, check if the type
7    // of the device is a graphic display, and set the displayed image of this
       display
8    else if (e.TaggingType == TaggingType.Device) {
9      if (Util.GetTypeFromPath(e.Value) == Constants.DEVICE_GRAPHICLCD) {
10       this.sharedDictionary[e.Value + "set/transition"] = GraphicLCD.
       TRANSITION_NONE;
11       this.sharedDictionary[e.Value + "set/image"] = this.graphicLCD.Image;
12 } } }
```

Figure 7.6: Source code for using RFID tags as identifications for artefacts and hardware.

Discussion

The appliance source code illustrates the usage of the toolkit API (e. g., registering event handlers for the accelerometer changes, and the detected RFID tags) as well as the direct access to the shared data model (e. g., the access to the photo displays by changing the hardware model entries). The scenario also illustrates the application of the RFID tagging mechanism: the physical artefacts (e. g., photo printouts and other objects) are coupled with the digital representation of the photos, and RFID tags are also used to identify the hardware of the digital photo frames (by specifying the unique path to the data model of the hardware).

Developers can easily extend the application and explore other combinations of hardware. For instance, the accelerometer can be used to detect simple gestures (e. g., shaking, waving)[3] and interpret these simple gestures to assign the photos to the digital photo frames, as illustrated in Figure 7.3(d). A vibration motor can be used to provide tangible feedback, for instance when an image is found (left side of Figure 7.3(a)). The accelerometer could also be replaced with a circular touch sensor to navigate through the photo collection. A GPS sensor could be used with the appliance to assign digital information to physical locations. Besides digital photos, the appliance could also be extended to integrate different media and information (e. g., sound, web links).

In summary, the presented prototype integrates simple to use methods for the interaction with the tangible interface (*tilting*, *proximity*, *pointing*). It also brings

3 Development project: `PhotoLines` [Marquardt, 2008].

the digital information (that usually resides on the desktop computer) into the everyday environment. The easy to use API and the encapsulated software building blocks allowed the rapid development of a prototype as well as iterations in the design to experiment with various forms of tangible interaction.

7.1.3 Remote and Ambient Awareness

This third introduced appliance[4] is an example of an ambient display that provides awareness information about a person at a distant location. For instance, the awareness display could be situated in the home of a family and indicates if the working parent is sitting at her desk, is around in the office, or is absent.

Implementation

The first part of the appliance is located in the office room and includes a proximity sensor near the desk to determine if someone sits at the desk. A force sensor at the door observes if the door is opened or closed. These two sensor values are utilised to estimate the availability of the working person. The counter part of the appliance is illustrated in Figure 7.7 and includes a figurine representation mounted on top of a servo motor. The position of the figurine can represent the availability status (facing the front = *Available*, side = *Around*, back = *Absent*). The current status is furthermore displayed on a connected text LC display.

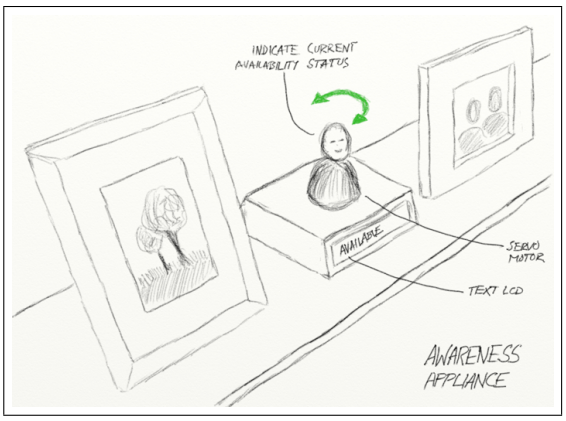


Figure 7.7: Awareness appliance implementation with an ambient display.

The infrastructure visualisation utility shown in Figure 7.8 is an important utility to support the developer with the integration of the remote located hardware sensors and displays. By means of the visualisation developers can get an overview of existing hardware (the markers in the centre of Figure 7.8), review the current appliance configuration (the lines between the markers in the centre of Figure 7.8), and control or view details of the hardware by using the interface skins (left and right side of Figure 7.8).

4 Development project: `AmbientAwareness` [Marquardt, 2008].

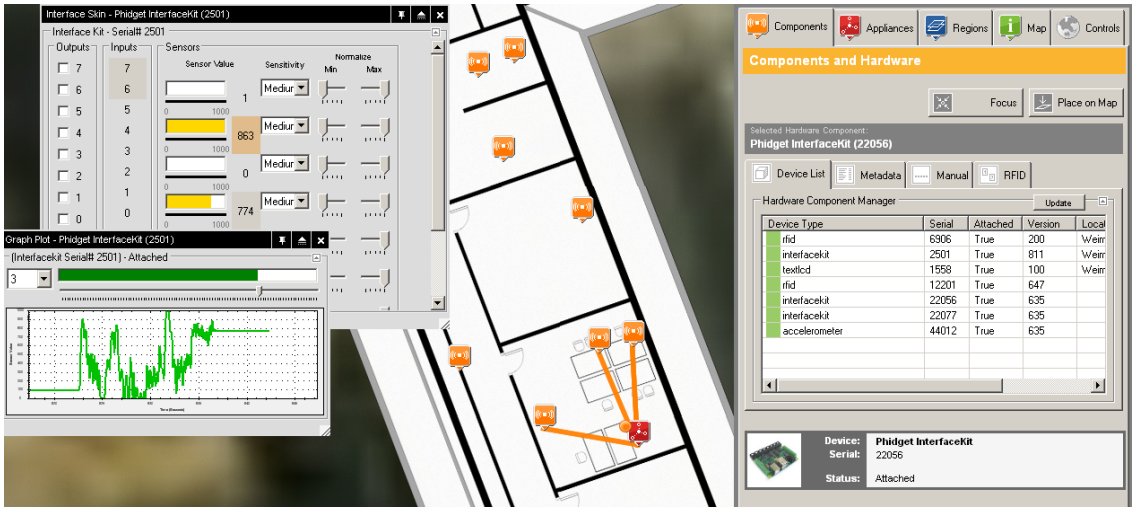


Figure 7.8: Infrastructure visualisation of the distributed hardware.

The implemented source code of the appliance is shown in Figure 7.9. At first, the callback method for the sensor uses a separate method to determine the current availability status. In this `Aggregate` method (lines 14-20 in Figure 7.9), the sensor values are interpreted and the method returns the estimated availability status. This is then used to change the servo position and the display text accordingly (lines 8 and 9 in Figure 7.9). Finally, the appliance is also adding a high-level entry to the shared data space (line 10 in Figure 7.9).

Discussion

While this prototype is only a simple implementation of an awareness display, it highlights the process of aggregating sensor values to high-level interpretations [Salber et al., 1999; Dey, 2000] and indicating awareness information (abstracted from the sensors) to remote located actuators. With it, this example comprises concepts of the *Physical but Digital Surrogates* [Greenberg and Kuzuoka, 2001] and the *Door Mouse* [Buxton, 1997]. The example appliance illustrated the processing of sensor data, interpretation of the raw sensor values, and deriving context information (i. e., the presence of a person). This information is easily published to the shared data model, and therefore it is available as high-level event to all other connected appliances.

Because the toolkit facilitates the access to the sensor information (local and remote located), developers can explore the possible aggregations and interpretations of sensor data, to derive high-level context information. These can be important steps to evaluate the applicability of such an appliance. Especially when

```
1  // Enumerate three availability states; labels for LCD
2  enum Availability {Present = 0, Around = 1, Absent = 2 };
3  string [] labels = new string [3] {"Present", "Around", "Absent"};
4
5  //New sensor values received; update the appliance
6  private void iK_SensorChange(...){
7    int status = this.Aggregate(iK.Sensors[0].Value, iK.Sensors[1].Value);
8    textLCD.Display = labels [status];
9    servo.Motors[0].Position = 90 * status;
10   this.PublishProcessingValue("presence-status/" + username, labels[status]);
11 }
12
13 // Aggregate the sensor values into an availability state
14 private int Aggregate (int forceSensor, int proximitySensor) {
15   bool door = (forceSensor < 50);         // door opened if force < 50
16   bool seated = (proximitySensor < 300); // seated if proximity < 300
17   if (door && seated) return (Availability.Present);
18   else if (door && !seated) return (Availability.Around);
19   else return (Availability.Absent);
20 }
```

Figure 7.9: Source code for awareness appliance.

sensors are embedded in the environment of users it is important to consider privacy and security issues in the appliance design. Therefore, the toolkit supports developers to find adequate abstractions that provide useful context information, but preserving privacy at the same time.

7.1.4 Further Appliance Examples

This subsection covers additional example prototypes that illustrate the spectrum of diverse appliances developers can build by means of the Shared Phidgets toolkit.

The *Location-Dependent Object Controller*[5] takes up on the ideas of the *Ubicomp Browser* research project [Beigl et al., 1998], as well as the *Point and Click* prototype [Beigl, 1999]. It is a tool for end users to obtain information about the interactive interface components embedded in their environment with a simple tangible device. As introduced earlier (cf. Subsection 6.1.2), hardware devices can be attached with an RFID tag for identification. Once the user brings the tangible controller near to a device, the display shows information about this hardware on the screen of the controller. This allows end users to monitor and control embedded sensors. This appliance could be extended to create *data flow links* between components. These links would then represent the application logic and would define the forwarding of events from a sensor source to an actuator.

5 Development project: `LocationDependentObjectController` [Marquardt, 2008].

Therefore, users would be able to create simple connection links between physical user interface components without programming a single line of code and only by using the tangible controller.

The *Augmented Map* appliance[6] discovers the idea of augmenting a physical map with digital information; whereas this prototype covers the research ideas of Reitmayr et al. [2005] and Reilly et al. [2006], as well as the research of the *Cameleon* prototype developed by Fitzmaurice [1993]. To implement the system, multiple RFID tags are attached onto the backside of a paper map (e. g. a city plan). These tags are associated with digital information, for instance digital photos, text, or Uniform Resource Locators (URL). This digital information are then in turn displayed once the RFID reader of the tangible controller (a device similar to the tangible photo browser of Subsection 7.1.2) covers a certain area of the map. The attached RFID reader identifies the tag and the appliance displays the corresponding digital content on the display of the tangible controller. The prototype implementation therewith facilitates the exploration of the user interaction with these digitally augmented maps.

The tangible controller (cf. Subsection 7.1.2) can be extended[7] with an easy to use mechanism that allows users to add new associations of *digital information* to *physical artefacts* by themselves; similar to ideas of Yonezawa et al. [2006] and Nunes et al. [2008]. With this information appliance people can use a simple *stamp* metaphor to assign specific digital information (for instance a digital image) to a physical artefact as illustrated in Figure 7.10. This technique is implemented with an attached force sensor on the backside of the tangible controller: if the user *stamps* the controller onto a physical artefact (e. g., paper maps, photos, souvenirs) and the RFID reader detects an RFID tag of this artefact, then the currently selected digital information (e. g., photos, web links, videos, text) is associated with this physical artefact. The information appliance software is adding a corresponding entry in the shared dictionary that maps the received RFID tag and the digital information. This technique could be also used with the described *Augmented Map* appliance: the user can then easily add his/her own digital information to the augmented physical map.

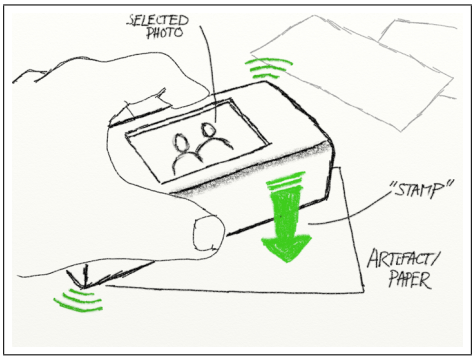


Figure 7.10: Create associations between the digital and physical world.

6 Development project: `AugmentedMap` [Marquardt, 2008].
7 Development project: `PhotoLines` [Marquardt, 2008].

The *Sensor Processing* appliance[8] implements a demonstration of integrating simple gestures into appliances. The software is observing values of two IR distance sensors that are mounted on top of a display. If a user in front of the screen waves his/her hand in front of the screen to the left or right side, the distance sensor is forwarding events to the appliance. The appliance software is then interpreting the values and time stamps, and if they pass a certain threshold, the appliance triggers a gesture event. As demonstration, this detected gesture is passed to the shared dictionary (by using the appliance high-level events) and also controls the images of a photo slide show (i. e., the commands *next image* and *previous image*). To test the detection of the gestures, the appliance includes test cases of the *Simulation Recorder* (cf. Subsection 6.3.2). Future work could include the integration of advanced sensor processing algorithms.

Other example appliances include a reimplementation of the *Lumi Touch* device [Chang et al., 2001][9]; an appliance that allows the activation or deactivation of digital outputs (e. g., power outlets) by sending text message from a mobile phone[10]; or the demonstration of navigating menus with an accelerometer or circular touch sensor[11].

In summary, these implemented prototypes illustrate the applicability of the toolkit for the development of different kinds of information appliances. Diverse physical hardware sensors and actuators—partially distributed—were utilised to build these appliances.

7.2 Discussion and Limitations

In the remainder of the chapter the toolkit's characteristics are discussed. This includes the critical discussion of the toolkit implementation as well as limitations of the toolkit design.

Klemmer et al. [2004] summarise important properties for the end user experience in software development (derived from empirical studies of the programmers' community) that they apply for the evaluation of their Papier-Mâché toolkit: *ease of use*, *facilitating reuse*, and *schemas yield similar code*. They evaluate their toolkit based on these properties; besides the common evaluation metrics for software engineering projects: *performance*, *reliability*, and *lines of code* [Klemmer et al., 2004]. Therefore, this subsection discusses the Shared Phidgets toolkit im-

8 Development project: `SensorProcessing` [Marquardt, 2008].
9 Development project: `LumiTouch` [Marquardt, 2008].
10 Development project: `MobileController` [Marquardt, 2008].
11 Development project: `Menu` [Marquardt, 2008].

plementation in consideration of these mentioned metrics and evaluation properties.

Ease of use

Providing a *low threshold* for the development is one of the most important requirements of the Shared Phidgets toolkit. With the introduced programming strategies, the library of programming building blocks, as well as the seamless integration into the IDE, the toolkit minimises the necessary tasks for developers. The appliance case studies in this chapter have illustrated the easy to use applicability of the toolkit; especially the use of the API library.

It is, however, important to note that even with a *low threshold* the development of appliances still requires fundamental programming knowledge. Users with no previous programming experience are not able to create custom applications as it would be possible with end user programming systems [Myers, 1986; Gross and Marquardt, 2007]. The toolkit also does not directly address the needs of interaction designers, which might prefer the building of prototypes in Adobe *Flash* or *Director*. Therefore, even if the toolkit minimises the development efforts in terms of the requirements for developers, there still remains the initial barrier of necessary fundamental programming skills.

The applicability of the toolkit has also been proven with the distribution of the toolkit to other developers. The Shared Phidgets toolkit has recently been applied for the development of physical user interfaces in an HCI course, and is also used by several academic and industrial research labs. It is planned, however, to further evaluate the application and use of the toolkit by other developers.

Facilitating Reuse

The encapsulation of the hardware access in the proxy objects builds the foundation of the reusable programming building blocks that are included in the developer library. Experienced developers can easily add additional proxy objects by using the extensible class framework introduced in Section 5.4. The high-level events of appliances are another mechanism to facilitate reuse. If an appliance is publishing high-level events to the shared dictionary, other appliances can simply register for these events to receive notifications of updated values.

Schemas Yield Similar Code

With this characteristic Klemmer et al. [2004] describe the advantages of similar code structures in development projects. It is desirable that the toolkit design yields to similar code patterns between programmers (but also for one program-

mer across different tasks) because this *"minimises design errors, facilitates collaboration, and makes maintaining the code of others easier"* [Klemmer et al., 2004].

So far the coding practices of developers using the Shared Phidgets toolkit have not yet been formally evaluated. The experiences, however, with the development of the appliance case studies have shown first development characteristics. These are for instance the frequently used event callbacks of the proxy objects, the only rarely used direct access to the shared data model, and the methods how appliances exchange information by using high-level events.

Performance, Scalability and Latency

The scalability of the development platforms is an important aspect in the development of ubiquitous computing applications [Helal, 2005; Abowd, 1999]. Ballagas et al. [2003] summarise in the evaluation of the iStuff toolkit that *"the issue of latency is inevitable in ubiquitous computing because of its distributed nature. Although latency can be minimised, it must be tolerated at some level in ubiquitous computing environments."* As it is important to estimate the performance of the Shared Phidgets runtime platform, this subsection discusses the scalability and latency issues.

To evaluate the performance of the Shared Phidgets toolkit, the toolkit network connection was tested with *round-trip latency*[12] measurement. In these tests[13], the platform was able to handle around 1000 events per second, with a packet size of 100 byte for each event, and an average round-trip latency of 35 ms. These results are comparable to the IBM TSpace server [Lehman et al., 2001] and the ECT [Greenhalgh et al., 2004]; both of these systems include a similar distributed data space architecture.

The Shared Phidgets toolkit runtime platform was tested with infrastructure installations that comprise around 30 hardware devices connected to multiple distributed clients and with a network connection to a single server instance. The *Connector* software was able to handle these installations without any performance issues, and for most prototyping implementations this number of around 30 connected hardware devices is likely to be sufficient (cf. Subsection 2.6.5). However, developers are not limited to this number of devices and can easily in-

12 *Round-trip latency*: measuring the period of time it takes for an event to be transmitted from the client to the server and back to the client. These latency tests for the Shared Phidgets toolkit can be executed with the `PerformanceTestUnit` development project.

13 Setup of the tests: two standard Intel 2.2 GHz PC, 1024 MB RAM, connected with a 802.11g wireless network connection. The test measurements however do not cover the delay caused by the hardware implementations; for instance delays of sensors or servo motors. These delays are dependent on the available hardware, and are therefore not considered in these networking tests.

tegrate more hardware devices. Two factors influence the maximum number of supported devices: the frequency of the sensor events (e. g., high data rate of an accelerometer vs. low data rate of an RFID reader), and the data size of the sent packages (e. g., large packet sizes of binary images vs. small packet sizes of events with integer values). Especially with frequently sent events of large binary data to the shared dictionary the latency of the transmitted events will increase. As mentioned earlier (cf. Subsection 4.2.2), for the case that it is necessary to overcome these scalability issues, future work could include the exchange of the underlying network system with peer-to-peer infrastructure that implements load-balancing algorithms [Bienkowski et al., 2005].

Reliability

The reliability aspects of the toolkit include the *robustness of the runtime platform* and the *transparency of the current status*. The *robustness of the runtime platform* has been tested with device infrastructures for prototypes that were running up to one week. Nonetheless, additional tests would be necessary to further evaluate the runtime robustness of the Shared Phidgets toolkit platform; especially if the toolkit is utilised to support evaluation of deployed prototypes. The available utilities of the toolkit can provide the developers a *transparent overview of the current status of the infrastructure*. At any time, they can monitor the available hardware; for instance to check the current values of sensors or to confirm that an RFID reader is connected and working correctly. Furthermore, with the appliance view in the geographical infrastructure visualisation developers can check the status and data flow of appliances at runtime.

Lines of code

The demonstrated information appliances presented in Section 7.1 have shown that it is possible to build information appliance prototypes with only around 20–50 lines of code. This code comprises the necessary code for the implementation of the application logic, whereas additional code is automatically provided by the *appliance template* (initialises network connection, provides methods for high-level events), the IDE visual designer (if classes from the developer library are instantiated by using the toolbox integration), and the Shared Phidgets toolkit *code framework generator add-in* (creates proxy object constructors, provides member initialisation, adds interface skins). Furthermore, the toolkit facilitates the transmission of events between appliances as well (by using high-level events) that can further minimise the necessary lines of code by subscribing for these events. Future work could include the extensions of the software building blocks in the developer library (cf. Section 8.1).

7.3 Chapter Summary

This chapter has described three case studies of prototype appliances built with the Shared Phidgets toolkit. These example case studies, as well as briefly described further appliances, are integrating various hardware sensors and actuators that are partially distributed at different locations. For each of these appliances the implementation was briefly described and the prototype design as well as the toolkit support discussed. Finally, at the end of the chapter, the toolkit design and limitations have been discussed.

CHAPTER 8

Conclusions and Future Work

The Shared Phidgets toolkit supports developers to build their visions of interactive distributed physical user interfaces. This thesis first motivated the research project, investigated the application area of physical user interfaces and the related research work, and derived the requirements for a prototyping toolkit. Based on these requirements, the Chapters 4–6 have introduced the concept and implementation of the Shared Phidgets toolkit runtime platform, developer library, and advanced development utilities. The evaluation of the toolkit included the development of appliance case studies and a discussion of the implemented toolkit.

In this last chapter, first the future work continuing the research of this thesis project is discussed. Second, the thesis contributions are summarised. Finally, the closing words conclude the thesis.

8.1 Future Work

Although the toolkit introduced with this thesis facilitates the development of distributed physical user interfaces, there are issues in the context of prototyping information appliances that can be investigated in further detail. The next section gives an overview of the research areas and topics that can extend the work presented in this thesis. Some of the mentioned research areas are based on the limitations of the toolkit already discussed in Section 7.2.

Further Evaluations

The presented case studies in Chapter 7 were only an initial evaluation of the toolkit and the developed appliances. Even though these case studies illustrate the application areas and the applicability of the toolkit and the utilities, it is important to evaluate the development process in further detail. This could include the comparative evaluation of different programming strategies, as well as the evaluation of occurring patterns in the development process when developing distributed interfaces.

It is also necessary to gain a deeper insight into the way users work and live with the embedded physical user interfaces. This can lead to a deeper understanding of how the developed technology has to be designed to merge seamlessly with the social practices of people [Dourish, 2001].

Library Extensions

The runtime architecture and developer library of the toolkit already support a wide range of diverse hardware devices as building blocks for new physical interfaces. Nonetheless, further hardware could be integrated by means of the runtime platform extensions (cf. plug-ins explained in Subsection 4.4.1). Feasible would be advanced identification devices (e. g., RFID readers with collision detection[1], bluetooth identification), new LC display technologies, and smaller wireless sensors.

Besides additional hardware, the library of software building blocks could be extended with additional components and user interface skins. This could include components to filter, aggregate, and interpret raw sensor data (similar to Context Widgets [Dey, 2000]); as well as advanced visualisations of sensor values.

Advanced Sensor Processing

When working with physical user interfaces that are built with a variety of sensors, it is also necessary to provide more powerful ways for the interpretation of the raw sensor data. This could be achieved with data mining algorithms that monitor the low-level sensor values and derive high level information from these values. For instance, these algorithms could be used to recognise gestures of users that are captured with distance and motion sensors.

The Shared Phidgets toolkit provides a good starting point for the integration of these algorithms. By simply adding the needed sensor components and register-

1 The currently used Phidget RFID reader does not provide a collision detection. This means that only one RFID tag can be detected at the same time. If multiple tags are in the range of the RFID antenna it is not guaranteed that any of these tags are detected.

ing for the events of the sensors, the algorithms could use this information to search for patterns in the received sensor values. The *Exemplar* system developed by Hartmann et al. [2006] for instance uses the *Dynamic Time Warping* algorithm[2] to detect pattern occurrences in a series of sensor values. A similar detection algorithm could be integrated as an appliance software unit of the Shared Phidgets toolkit, and publish the recognised gestures as high-level events to the shared data space. This would in turn raise notifications to all registered event listeners; for instance another appliance that is interpreting these high-level events to control an actuator.

End user Reconfiguration

The toolkit and utilities presented in this thesis address in particular the needs of application developers. However, the introduced concepts and utilities to facilitate the exploration of interactive environments could also support end users getting information about the embedded technology. Further research could introduce end user utilities based on the map visualisation presented in Section 6.2, or the RFID tagging mechanism introduced in Subsection 6.1.2. Furthermore, the toolkit's functionality for the reconfiguration of appliances could be used as a basis for tools that end users can apply for changing appliance configurations to their personal needs.

Utilities and Visualisation

The introduced infrastructure visualisation of the distributed hardware sensors, actuators, and appliances can be the foundation of advanced visualisation systems. For instance, the map could display the history of occurred events (e. g., as colour coded trails between the hardware components). Furthermore, the visualisation could include advanced interactive functionality to control the data flow between hardware devices and appliances (e. g., a drag-and-drop functionality to create new connections) and extend the functionality of the metadata regions (e. g., with conjunction and disjunction operations).

8.2 Thesis Contributions

The main objective of this thesis was to facilitate the development of distributed physical user interface prototypes. To achieve this objective, the concept of the

2 The *Dynamic Time Warping algorithm* was primarily applied in speech recognition engines, but is recently also utilised in gesture recognition engines of sensor-based interactive systems [Hartmann et al., 2006].

Shared Phidgets toolkit has been introduced. The contributions of this thesis are a flexible *runtime platform* that allows the remote access to distributed hardware, a *developer library* that facilitates the programming of information appliances, and *utilities* that let developers easily monitor and control the distributed hardware and appliances.

The implemented *runtime platform* facilitates the setup of infrastructures for the distributed access to physical sensor and actuator hardware. The *Connector* software maintains a shared data space to connect all client machines over the network. The implemented reference plug-ins automatically integrate locally attached hardware into the shared data space; for instance accelerometer sensors, servo motors, and graphic displays. Most importantly, once the software is installed it manages these tasks autonomously and is running in the background of the operating system. The class framework facilitates the development of plug-ins to integrate custom hardware into the infrastructure. With these characteristics, the runtime platform meets all the requirements described in Section 3.3.

To let developers build information appliances with physical user interfaces, the second contribution of the thesis is the *developer library*. With the object-oriented API of the proxy objects for the hardware access, as well as the seamless integration into the development environment (for instance with programming templates, integrated device browser, automatic code framework generator) the developer library provides a *low threshold* for developers with no experience of programming hardware. In addition, the introduced appliance model, metadata information, and the interface skins also facilitate the development of appliances. Case studies in Chapter 7 have proven the applicability of the developer library and shown that prototypes can be built with around 20–50 lines of code that developers have to implement. To furthermore provide a *high ceiling*, expert developers can access the shared data model directly (e. g., for the efficient access to collections of hardware devices) and can develop extensions of the library (e. g., custom proxy objects, filters, aggregators) by deriving new classes from the framework interfaces of abstract base classes.

Finally, the third contribution of the thesis is the development of a collection of *development utilities* that support the monitoring, control, and simulation of the distributed hardware devices. First, *the monitoring and control utilities* allow the access to the three abstraction levels of the toolkit: the shared data space level, the hardware device level, and the appliance level. Second, the *advanced spatial visualisation utility* shows the network of appliances as well as the hardware at their geographical locations. This utility therefore allows insights into the internal connections and event flow of appliances, supported by the details-on-demand user interface. Finally, the *integration of simulations* allows the testing and debugging of appliances. While the simulation interface components provide *Wizard of*

Oz simulations of the hardware, the recording mechanism allows the recording of complex sensor values or an event series in general, and the later testing of the appliance based on the recordings.

These three main parts of the toolkit work seamlessly together and complement each other. By using the Shared Phidgets toolkit developers can focus on the interaction design of the envisioned information appliances. The facilitation of the rapid prototyping process by means of the toolkit is the most important contribution of this thesis project.

8.3 Closing Words

I would like to conclude this thesis with a quotation from Mark Weiser that points out an inspiring motivation behind the research of ubiquitous computing:

> *"Most important, ubiquitous computers will help overcome the problem of information overload. There is more information available at our fingertips during a walk in the woods than in any computer system, yet people find a walk among trees relaxing and computers frustrating. Machines that fit the human environment instead of forcing humans to enter theirs will make using a computer as refreshing as taking a walk in the woods."* [Weiser, 1991]

This is a very important motivation behind the research of this thesis. I invite the readers of this thesis to try out the Shared Phidgets toolkit by themselves and to discover the creation of innovative and unconventional user interactions with physical interfaces.

`http://grouplab.cpsc.ucalgary.ca/cookbook/`

I hope that the Shared Phidgets toolkit will support many application developers with the exploration of physical user interfaces, in order that these innovations come up to Mark Weiser's vision of interfaces that *fit the human environment*.

References

ABOWD, G. D. 1996. Software Engineering and Programming Language Considerations for Ubiquitous Computing. *ACM Computing Surveys, Special Issue: Position Statements on Strategic Directions in Computing Research 28,* 4, 190.

ABOWD, G. D. 1999. Software Engineering Issues for Ubiquitous Computing. In *Proceedings of the 21st International Conference on Software Engineering - ICSE 1999 (Los Angeles, CA, USA)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 75–84.

ABOWD, G. D. AND MYNATT, E. D. 2000. Charting Past, Present, and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction 7,* 1, 29–58.

BABULAK, E. 2006. Automated Smart House 2015 via Ubiquitous Computing. In *Proceedings of the International Conference on Interactive Mobile and Computer Aided Learning - IMCL 2006 (Amman, Jordan)*.

BALLAGAS, R., MEMON, F., REINERS, R., AND BORCHERS, J. 2007. iStuff Mobile: Rapidly Prototyping New Mobile Phone Interfaces for Ubiquitous Computing. In *Proceedings of the 25th ACM Conference on Human Factors in Computing Systems - CHI 2007 (San Jose, CA, USA)*. ACM Press, New York, NY, USA.

BALLAGAS, R., RINGEL, M., STONE, M., AND BORCHERS, J. 2003. istuff: a physical user interface toolkit for ubiquitous computing environments. In *Proceedings of the 21th ACM Conference on Human Factors in Computing Systems - CHI 2003 (Fort Lauderdale, Florida, USA)*. ACM Press, New York, NY, USA, 537–544.

BALLAGAS, R., SZYBALSKI, A., AND FOX, A. 2004. Patch Panel: Enabling Control-flow Interoperability in Ubicomp Environments. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications - PerCom 2004 (Orlando, Florida, USA)*. 241.

BARRETT, R. AND MAGLIO, P. P. 1998. Informative Things: How to Attach Information to the Real World. In *Proceedings of the 11th Annual ACM Symposium*

on User Interface Software and Technology - UIST 1998 (San Francisco, CA, USA). ACM Press, New York, NY, USA, 81–88.

BEIGL, M. 1999. Point & Click - Interaction in Smart Environments. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing - HUC 1999 (Karlsruhe, Germany)*. Springer, 311–313.

BEIGL, M., SCHMIDT, A., LAUFF, M., AND GELLERSEN, H.-W. 1998. The UbicompBrowser. In *Proceedings of the 4th ERCIM Workshop on User Interfaces for All*.

BIENKOWSKI, M., KORZENIOWSKI, M., AND AUF DER HEIDE, F. M. 2005. Dynamic Load Balancing in Distributed Hash Tables. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems - IPTPS 2005*. 217–225.

BOYLE, M. AND GREENBERG, S. 2005. Rapidly Prototyping Multimedia Groupware. In *Proceedings of the 11th International Conference on Distributed Multimedia Systems - DMS 2005 (Banff, Canada)*. Knowledge Systems Institute.

BRUMITT, B., MEYERS, B., KRUMM, J., KERN, A., AND SHAFER, S. 2000. Easy Living: Technologies for Intelligent Environments. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing - HUC 2000 (Bristol, UK)*. 12–27.

BUXTON, W. A. S. 1995. Integrating the Periphery and Context: A New Model of Telematics. In *Proceedings of Graphics Interface - GI 1995 (Quebec, Canada)*. 239–246.

BUXTON, W. A. S. 1997. Living in Augmented Reality: Ubiquitous Media and Reactive Environments. In *Video Mediated Communication*, K. Finn, A. Sellen, and S. Wilber, Eds. Lawrence Erlbaum Associates, Inc., Hillsdale, N.J., USA, 363–384.

BUXTON, W. A. S. 2007. *Sketching User Experience : Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers, San Francisco, CA, USA.

CHANG, A., RESNER, B., KOERNER, B., WANG, X., AND ISHII, H. 2001. LumiTouch: An Emotional Communication Device. In *Extended Abstracts of the 19th ACM Conference on Human Factors in Computing Systems - CHI 2001 (Seattle, Washington, USA)*. ACM Press, New York, NY, USA, 313–314.

CONSOLVO, S., ROESSLER, P., AND SHELTON, B. E. 2004. The CareNet Display: Lessons Learned from an In Home Evaluation of an Ambient Display. In *Proceedings of the Sixth International Conference on Ubiquitous Computing - UbiComp 2004 (Nottingham, UK)*, N. Davies, E. D. Mynatt, and I. Siio, Eds. Lecture Notes in Computer Science, vol. 3205. Springer, Nottingham, UK, 1–17.

CONSOLVO, S. AND TOWLE, J. 2005. Evaluating an Ambient Display for the Home. In *Extended Abstracts of the 23th ACM Conference on Human Factors in Computing Systems - CHI 2005 (Portland, Oregon, USA)*. ACM Press, New York, NY, USA, 1304–1307.

CRAMPTON SMITH, G. 1995. *The Hand That Rocks the Cradle*. I.D.

DEY, A. K. 2000. Providing Architectural Support for Building Context-Aware Applications. Ph.D. thesis, Georgia Institute of Technology.

DOURISH, P. 2001. *Where the Action Is: The Foundations of Embodied Interaction*. The MIT Press.

DOW, S., MACINTYRE, B., LEE, J., OEZBEK, C., BOLTER, J. D., AND GANDY, M. 2005. Wizard of Oz Support throughout an Iterative Design Process. *IEEE Pervasive Computing 4,* 4, 18–26.

DUCHENEAUT, N., SMITH, T. F., BEGOLE, J. B., NEWMAN, M. W., AND BECKMANN, C. 2006. The Orbital Browser: Composing Ubicomp Services Using Only Rotation and Selection. In *Extended Abstracts of the 24th ACM Conference on Human Factors in Computing Systems - CHI 2006 (Montreal, Quebec, Canada)*. ACM Press, New York, NY, USA, 321–326.

EGGLESTONE, S. R., BOUCHER, A., GREENHALGH, C., HUMBLE, J., LAW, A., PENNINGTON, S., , AND RODDEN, T. 2006. Supporting Collaboration in the Deployment of Ubiquitous Computing Installations. In *Ubiquitous Systems Workshop (UbiSys) at the Eighth International Conference on Ubiquitous Computing - UbiComp 2006 (Orange County, CA, USA)*. Irvine, California, USA.

EGGLESTONE, S. R., HUMBLE, J., GREENHALGH, C., RODDEN, T., AND HAMPSHIRE, A. 2006. The Equator Component Toolkit: Managing Digital Information Flow in the Home. In *Adjunct Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST 2006 (Montreux, Switzerland)*. ACM Press, New York, NY, USA.

ELLIOT, K., NEUSTAEDTER, C., AND GREENBERG, S. 2007. StickySpots: Using Location to Embed Technology in the Social Practices of the Home. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction - TEI 2007 (Baton Rouge, LA, USA)*. ACM Press, New York, NY, USA, 79–86.

ELLIOT, K., WATSON, M., NEUSTAEDTER, C., AND GREENBERG, S. 2007. Location-Dependent Information Appliances for the Home. In *Proceedings Graphics Interface - GI 2007 (Montreal, Quebec, Canada)*.

FITZMAURICE, G. W. 1993. Situated Information Spaces and Spatially Aware Palmtop Computers. *Communications of the ACM 36,* 7, 39–49.

FITZMAURICE, G. W., ISHII, H., AND BUXTON, W. A. S. 1995. Bricks: Laying the Foundations for Graspable User Interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1995 (Denver, Colorado, USA)*. ACM Press, New York, NY, USA, 442–449.

FITZPATRICK, G., KAPLAN, S., MANSFIELD, T., DAVID, A., AND SEGALL, B. 2002. Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections. *Computer Supported Cooperative Work 11,* 3, 447–474.

GREENBERG, S. 2005. Collaborative Physical User Interfaces. In *Communication and Collaboration Support Systems,* K. Okada, T. Hoshi, and T. Inoue, Eds. IOS Press (Amsterdam, The Netherlands).

GREENBERG, S. 2007. Toolkits and Interface Creativity. *Multimedia Tools and Applications 32,* 2, 139–159.

GREENBERG, S. AND BOYLE, M. 2002. Customizable Physical Interfaces for Interacting with Conventional Applications. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology - UIST 2002 (Paris, France)*. ACM Press, New York, NY, USA, 31–40.

GREENBERG, S. AND FITCHETT, C. 2001. Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology - UIST 2001 (Orlando, Florida, USA)*. ACM Press, New York, NY, USA, 209–218.

GREENBERG, S. AND KUZUOKA, H. 2001. Using Digital but Physical Surrogates to Mediate Awareness, Communication and Privacy in Media Spaces. In *Personal Technologies*. Elsevier.

GREENBERG, S. AND ROSEMAN, M. 1999. Groupware Toolkits for Synchronous Work. In *Computer-Supported Cooperative Work (Trends in Software 7)*, M. Beaudouin-Lafon, Ed. Vol. 7. Wiley & Sons, Chapter 6, 135–168.

GREENHALGH, C., IZADI, S., MATHRICK, J., HUMBLE, J., AND TAYLOR, I. 2004. ECT: A Toolkit to Support Rapid Construction of Ubicomp Environments. In *Workshop on System Support for Ubiquitous Computing (UbiSys) at the Conference on Ubiquitous Computing - UbiComp 2004 (Nottingham, UK)*. Nottingham, UK.

GROSS, T. AND MARQUARDT, N. 2007. CollaborationBus: An Editor for the Easy Configuration of Ubiquitous Computing Environments. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing - PDP 2007 (Naples, Italy)*. Naples, Italy, 307–314.

HARRISON, B. L., FISHKIN, K. P., GUJAR, A., MOCHON, C., AND WANT, R. 1998. Squeeze Me, Hold Me, Tilt Me! An Exploration of Manipulative User Interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1998 (Los Angeles, California, USA)*. ACM Press, New York, NY, USA, 17–24.

HARTMANN, B., KLEMMER, S. R., BERNSTEIN, M., ABDULLA, L., BURR, B., ROBINSON-MOSHER, A., AND GEE, J. 2006. Reflective Physical Prototyping Through Integrated Design, Test, and Analysis. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST 2006 (Montreux, Switzerland)*. ACM Press, New York, NY, USA, 299–308.

HELAL, S. 2005. Programming Pervasive Spaces. *IEEE Pervasive Computing 04,* 1, 84–87.

HELAL, S., MANN, W., EL-ZABADANI, H., KING, J., KADDOURA, Y., AND JANSEN, E. 2005. The Gator Tech Smart House: A Programmable Pervasive Space. *Computer 38,* 3, 50–60.

HUDSON, S. E. AND MANKOFF, J. 2006. Rapid Construction of Functioning Physical Interfaces from Cardboard, Thumbtacks, Tin Foil and Masking Tape. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST 2006 (Montreux, Switzerland)*. ACM Press, New York, NY, USA, 289–298.

ISHII, H. AND ULLMER, B. 1997. Tangible Bits: Towards Seamless Interfaces Between People, Bits and Atoms. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1997 (Atlanta, Georgia, USA)*. ACM Press, New York, NY, USA, 234–241.

JOHANSON, B. AND FOX, A. 2002. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, Washington, DC, USA, 83–93.

KASAL, A., LIU, J., MILLER, J., NATH, S., ROUHANA, D., SANTANCHE, A., AND ZHAO, F. 2007. SenseWeb Project. `http://research.microsoft.com/nec/senseweb/`. Website. Website last visited on September 18, 2007.

KIDD, C. D., ORR, R., ABOWD, G. D., ATKESON, C. G., ESSA, I. A., MACINTYRE, B., MYNATT, E. D., STARNER, T., AND NEWSTETTER, W. 1999. The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In *Cooperative Buildings*. 191–198.

KIM, S. W., KIM, M. C., PARK, S. H., JIN, Y. K., AND CHOI, W. S. 2004. Gate Reminder: A Design Case of a Smart Reminder. In *Proceedings of the 5th ACM Con-*

ference on Designing Interactive Systems - DIS 2004 (Cambridge, Massachusetts, USA). ACM Press, New York, NY, USA, 81–90.

KLEMMER, S. R., LI, J., LIN, J., AND LANDAY, J. A. 2004. Papier-Mache: Toolkit Support for Tangible Input. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 2004 (Vienna, Austria)*. ACM Press, New York, NY, USA, 399–406.

LEACH, P., MEALLING, M., AND SALZ, R. 2005. A Universally Unique IDentifier (UUID) URN Namespace. `http://www.ietf.org/rfc/rfc4122.txt`. The Internet Engineering Task Force (IETF), Network Working Group, Request for Comments: 4122. Website last visited on February 10, 2008.

LEE, J. C., AVRAHAMI, D., HUDSON, S. E., FORLIZZI, J., DIETZ, P. H., AND LEIGH, D. 2004. The Calder Toolkit: Wired and Wireless Components for Rapidly Prototyping Interactive Devices. In *Proceedings of the 5th ACM Conference on Designing Interactive Systems - DIS 2004 (Cambridge, Massachusetts, USA)*. ACM Press, New York, NY, USA, 167–175.

LEHMAN, T. J., COZZI, A., XIONG, Y., GOTTSCHALK, J., VASUDEVAN, V., LANDIS, S., DAVIS, P., KHAVAR, B., AND BOWMAN, P. 2001. Hitting the Distributed Computing Sweet Spot with TSpaces. *Computer Networks: The International Journal of Computer and Telecommunications Networking 35,* 4, 457–472.

LI, Y., HONG, J. I., AND LANDAY, J. A. 2007. Design Challenges and Principles for Wizard of Oz Testing of Location-Enhanced Applications. *IEEE Pervasive Computing 6,* 2, 70–75.

LIU, L. AND KHOOSHABEH, P. 2003. Paper or Interactive?: A Study of Prototyping Techniques for Ubiquitous Computing Environments. In *Extended Abstracts of the 21st ACM Conference on Human Factors in Computing Systems - CHI 2003 (Fort Lauderdale, Florida, USA)*. ACM Press, New York, NY, USA, 1030–1031.

MARQUARDT, N. 2008. Shared Phidgets (Downloads, Tutorials, Examples). `http://grouplab.cpsc.ucalgary.ca/cookbook/index.php/Toolkits/SharedPhidgets3`. Developer Cookbook of the GroupLab at the University of Calgary. Website last visited on March 10, 2008.

MATTHEWS, T. 2005. Peripheral Display Toolkit: A Toolkit for Managing User Attention in Peripheral Displays. M.S. thesis, Computer Science Division, University of California, Berkeley.

MATTHEWS, T., DEY, A. K., MANKOFF, J., CARTER, S., AND RATTENBURY, T. 2004. A Toolkit for Managing User Attention in Peripheral Displays. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology - UIST 2004 (Santa Fe, New Mexico, USA)*. ACM Press, New York, NY, USA, 247–256.

MEIJER, E. AND GOUGH, J. 2000. Technical Overview of the Common Language Runtime. `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`. Microsoft. Website last visited on November 20, 2007.

MICROSOFT CORPORATION. 2007a. Microsoft Virtual Earth Interactive SDK. `http://dev.live.com/virtualearth/sdk/`. Website last visited on November 3, 2007.

MICROSOFT CORPORATION. 2007b. Virtual Earth Map Control 5.0. `http://msdn2.microsoft.com/en-us/library/bb429619.aspx`. Microsoft Developer Network. Website last visited on November 10, 2007.

MICROSOFT CORPORATION. 2007c. Virtual Earth Website - Windows Live Local. `http://local.live.com/`. Website last visited on November 14, 2007.

MICROSOFT CORPORATION. 2007d. Visual Studio 2005 Developer Center. `http://msdn.microsoft.com/vstudio/`. Microsoft Developer Network. Website last visited on September 10, 2007.

MICROSOFT RESEARCH. 2007. MapCruncher for Virtual Earth Website. `http://research.microsoft.com/mapcruncher/`. Microsoft Research Website. Website last visited on September 5, 2007.

MORRIS, M. R. 2004. Visualization for Casual Debugging and System Awareness in a Ubiquitous Computing Environment. In *Adjunct Proceedings of the Sixth International Conference on Ubiquitous Computing - UbiComp 2004 (Nottingham, UK)*.

MYERS, B. A. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the 4th ACM Conference on Human Factors in Computing Systems - CHI 1986 (Boston, Massachusetts, USA)*. ACM Press, New York, NY, USA, 59–66.

MYERS, B. A., HUDSON, S. E., AND PAUSCH, R. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction 7,* 1, 3–28.

MYNATT, E. D., BACK, M., WANT, R., BAER, M., AND ELLIS, J. B. 1998. Designing Audio Aura. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1998 (Los Angeles, California, USA)*. ACM Press, New York, NY, USA, 566–573.

MYNATT, E. D., ROWAN, J., JACOBS, A., AND CRAIGHILL, S. 2001. Digital Family Portraits: Supporting Peace of Mind for Extended Family Members. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 2001 (Seattle, Washington, USA)*. ACM Press, New York, NY, USA, 333–340.

NAGEL, K., KIDD, C. D., O'CONNELL, T., DEY, A. K., AND ABOWD, G. D. 2001. The Family Intercom: Developing a Context-Aware Audio Communication System. In *Proceedings of the Third International Conference on Ubiquitous Computing - UbiComp 2001 (Atlanta, Georgia, USA)*. Springer-Verlag, London, UK, 176–183.

NIELSEN, J. 1993. *Usability Engineering*. Morgan Kaufmann Publishers.

NORMAN, D. A. 1988. *The Psychology of Everyday Things*. Basic Books, New York, NY, USA.

NORMAN, D. A. 1999. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are The Solution*. MIT Press.

NORMAN, D. A. AND DRAPER, S. W. 1986. *User Centered System Design; New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA.

NUNES, M., GREENBERG, S., AND NEUSTAEDTER, C. 2008. Sharing Digital Photographs in the Home through Physical Mementos, Souvenirs, and Keepsakes. In *Proceedings of the ACM Conference on Designing Interactive Systems - DIS 2008 (Cape Town, South Africa)*.

PHIDGETS INC. 2008. Phidgets Product Website. `http://www.phidgets.com`. Website last visited on October 24, 2007.

PREECE, J., ROGERS, Y., AND SHARP, H. 2002. *Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA.

PRINZ, W. AND GROSS, T. 2001. Ubiquitous Awareness of Cooperative Activities in a Theatre of Work. In *Proceedings of Fachtagung Arbeitsplatzcomputer: Pervasive Ubiquitous Computing - APC 2001 (Munich, Germany)*, A. Bode and W. Karl, Eds. VDE Publisher, 135–144.

REILLY, D., RODGERS, M., ARGUE, R., NUNES, M., AND INKPEN, K. 2006. Marked-up Maps: Combining Paper Maps and Electronic Information Resources. *Personal Ubiquitous Computing 10,* 4, 215–226.

REITMAYR, G., EADE, E., AND DRUMMOND, T. 2005. Localisation and Interaction for Augmented Maps. In *Proceedings of the Fourth IEEE and ACM International Symposium on Mixed and Augmented Reality - ISMAR 2005 (Vienna, Austria)*. IEEE Computer Society, Washington, DC, USA, 120–129.

REKIMOTO, J. 1996. Tilting Operations for Small Screen Interfaces. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology - UIST 1996 (Seattle, Washington, USA)*. ACM Press, New York, NY, USA, 167–168.

ROSEMAN, M. 1993. Design of a Real-Time Groupware Toolkit. M.S. thesis, University of Calgary, Department of Computer Science.

RUDD, J., STERN, K., AND ISENSEE, S. 1996. Low vs. High-Fidelity Prototyping Debate. *ACM interactions 3,* 1, 76–85.

SALBER, D., DEY, A. K., AND ABOWD, G. D. 1999. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1999 (Pittsburgh, Pennsylvania, USA)*. ACM Press, New York, NY, USA, 434–441.

SANTANCHE, A., NATH, S., LIU, J., PRIYANTHA, B., AND ZHAO, F. 2006. SenseWeb: Browsing the Physical World in Real Time. In *Proceedings of the Fifth ACM/IEEE International Conference on Information Processing in Sensor Networks - IPSN 2006*. Nashville, TN.

SCHILIT, B. N., ADAMS, N., GOLD, R., TSO, M. M., AND WANT, R. 1993. The PARCTAB Mobile Computing System. In *Workshop on Workstation Operating Systems*. 34–39.

SELLEN, A., HARPER, R., EARDLEY, R., IZADI, S., REGAN, T., TAYLOR, A. S., AND WOOD, K. R. 2006. HomeNote: Supporting Situated Messaging in the Home. In *Proceedings of the 20th ACM Conference on Computer Supported Cooperative Work - CSCW 2006 (Banff, Alberta, Canada)*. ACM Press, New York, NY, USA, 383–392.

SHNEIDERMAN, B. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of IEEE Symposium on Visual Languages*. 336–343.

SOHN, T. AND DEY, A. 2003. iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications. In *Extended Abstracts of the 21st ACM Conference on Human Factors in Computing Systems - CHI 2003 (Fort Lauderdale, Florida, USA)*. ACM Press, New York, NY, USA, 974–975.

SUN MICROSYSTEMS, INC. 2007. Java Beans Architecture. `http://java.sun.com/products/javabeans/`. Sun Developer Network (SDN). Website last visited on October 30, 2007.

ULLMER, B. AND ISHII, H. 1997. The metaDESK: Models and Prototypes for Tangible User Interfaces. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology - UIST 1997 (Banff, Alberta, Canada)*. ACM Press, New York, NY, USA, 223–232.

ULLMER, B. AND ISHII, H. 2000. Emerging Frameworks for Tangible User Interfaces. *IBM Systems Journal 39,* 3-4, 915–931.

VILLAR, N. AND GELLERSEN, H. 2007. A Malleable Control Structure for Softwired User Interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction - TEI 2007 (Baton Rouge, LA, USA)*. ACM Press, New York, NY, USA, 49–56.

WANT, R., FISHKIN, K. P., GUJAR, A., AND HARRISON, B. L. 1999. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1999 (Pittsburgh, Pennsylvania, USA)*. ACM Press, New York, NY, USA, 370–377.

WANT, R., HOPPER, A., FALCAO, V., AND GIBBONS, J. 1992. The Active Badge Location System. *ACM Transactions on Information Systems 10,* 1, 91–102.

WEISER, M. 1991. The Computer for the 21st Century. *Scientific American 265,* 3 (September), 66–75.

WEISER, M. 1993. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM 36,* 7, 75–84.

WEISER, M. 1996. Ubiquitous Computing Website at XEROX PARC. `http://sandbox.xerox.com/ubicomp/`. Website last visited on February 2, 2008.

WEISER, M. AND BROWN, J. S. 1996. Designing Calm Technology. *PowerGrid Journal 1*, 1.

WEISER, M. AND BROWN, J. S. 1997. The Coming Age of Calm Technology. *Beyond Calculation: The Next Fifty Years 1*, 75–85.

WISNESKI, C., ISHII, H., DAHLEY, A., GORBET, M., BRAVE, S., ULLMER, B., AND YARIN, P. 1998. Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information. In *First International Workshop on Cooperative Buildings - Integrating Information, Organization, and Architecture - CoBuild 1998*. Darmstadt, Germany, 22+.

WORLD WIDE WEB CONSORTIUM (W3C). 1999. XML Path Language (XPath). `http://www.w3.org/TR/xpath`. W3C Recommendation. Website last visited on September 12, 2007.

YONEZAWA, T., SAKAKIBARA, H., NAKAZAWA, J., TAKASHIO, K., AND TOKUDA, H. 2006. Spot & Snap: A Bootstrap Interaction for DIY Smart Object Services. In *Adjunct Proceedings of the Eighth International Conference on Ubiquitous Computing - UbiComp 2006 (Orange County, CA, USA)*.

Acronyms

3D	Three-Dimensional
ACM	Association of Computing Machinery
API	Application Programming Interface
CHI	Computer-Human Interaction
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
DLL	Dynamic Linked Library
GIS	Geographic Information System
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HCI	Human-Computer Interaction
HTML	Hypertext Markup Language
HTTP	Hypertext Transmission Protocol
ID	Identification
IDE	Integrated Development Environment
I/O	Input / Output
IR	Infrared
LC	Liquid Crystal
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LOC	Lines of Code
MODEM	Modulator-Demodulator
MVC	Model-View-Controller
NMEA	National Marine Electronics Association
OO	Object-Oriented
OOP	Object-Oriented Programming

PC	Personal Computer
PDA	Personal Digital Assistant
PUI	Physical User Interface
RF	Radio Frequency
RFID	Radio Frequency Identification
SDK	Software Development Kit
SE	Software Engineering
SMS	Short Message Service
TCP/IP	Transmission Control Protocol/Internet Protocol
TUI	Tangible User Interface
UbiComp	Ubiquitous Computing
UI	User Interface
UML	Unified Modelling Language
URL	Uniform Resource Locator
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WOz	Wizard of Oz simulations
XML	Extended Markup Language

APPENDIX A

Development

A.1 System Requirements

This appendix section gives a brief introduction of the development requirements of the Shared Phidgets toolkit architecture.

The toolkit, the developer library, and all included utilities are written in Microsoft *C#.NET 2.0* (Version 2.0.50727). Nonetheless, developers can choose to use the developer library with any of the managed *.NET* programming languages supported by the *Common Language Runtime (CLR)* [Meijer and Gough, 2000]. These are *Visual Basic*, *J#*, *C++*, and *C#*. The source files of the Shared Phidgets toolkit are compiled with the *C#.NET 2.0* compiler. The toolkit can be used with the versions 3.0 and 3.5 of the *.NET* platform as well. A limited version of the developer library is available for the *.NET* 1.1 target platform. The source code examples of classes and interfaces in this thesis are written in *C#.NET 2.0*.

Microsoft *Visual Studio 2005* (Version 8.0.50727.42) is recommended as the Integrated Development Environment (IDE). The setup of the toolkit automatically integrates the developer library into the visual designer toolbox and the add-in into the Visual Studio IDE. The developer library can be used for development in Visual Studio 2008 as well, though. Nonetheless, the add-in is not yet compatible with Visual Studio 2008. The Shared Phidgets toolkit can also be referenced from any *.NET* project without using the IDE, but using the command line compiler directly.

The following list provides an overview of the requirements:

- **Operating System:** Microsoft Windows 2000, XP, or Vista.
- **Programming Languages:** supported are all programming languages of the Microsoft *.NET CLR*: *Visual Basic*, *J#*, *C++*, and *C#*. Supported versions of the *.NET* framework are 2.0, 3.0, and 3.5. Limited support for 1.1.
- **Development Environment:** target IDE is Visual Studio 2005 [Microsoft Corporation, 2007d] (full integration of the development tools). The Toolkit is also tested with Visual Studio 2008 (add-in not available), and a previous toolkit version is available for Visual Studio 2002 and 2003 (with *.NET 1.1*). The toolkit developer library can be used with any other IDE and the *.NET CLR* compiler directly.
- **Virtualisation:** tested with VMware Fusion (Version 1.0 and 1.1) and Parallels Desktop (Version 2.0) under Apple Mac OS X (Version 10.4.10). Limited support for Microsoft Virtual PC 2004 on Mac OS X; Virtual PC does not support all USB devices.

A.2 Development Projects

The Shared Phidgets toolkit consists of the following *C#.NET* development projects[1]:

- **SharedPhidgetsPlatform:** The connector tool and all plug-ins (Phidgets, Phidgets Remote, GPS Module, GSM Gateway, and Graphic LCD).
- **SharedPhidgetsLibrary:** This project includes the developer library with the proxy objects, interface skins, and all other implemented programming objects.
- **SharedPhidgetsUtilities:** Includes the implementations of the utilities (e. g., explorer, map visualisation).
- **SharedPhidgetsAddIn:** The implementation of the Visual Studio add-in. The setup integrates this add-in into Visual Studio 2005.
- **SharedPhidgetsUseCases:** Includes the example implementations and case studies explained in the thesis.
- **SharedPhidgetsTemplate:** The project template for Shared Phidgets projects. The setup integrates this template into Visual Studio 2005.

1 These projects are included on the `/src/` folder of the thesis DVD and are available for download at `http://grouplab.cpsc.ucalgary.ca/cookbook/`

- **SharedPhidgetsExamples:** Simple programming examples that illustrate the programming with the proxy objects and interface skins.
- **SharedPhidgetsWirelessDisplay:** Client software running on Windows Mobile devices to use these devices as wireless Graphic LCDs.
- **SharedPhidgetsSetups:** Development projects for the compilation of the two versions of the setup program.

APPENDIX B

Developer Library API

Name	Proxy Component	Interface Skin	Simulation	Connection	Management	Description
`Accelerometer`	■					Proxy component to access the Phidget Accelerometer hardware.
`AccelerometerSkin`		■				User control to observe the Phidget Accelerometer attributes, and to control the measurement sensitivity.
`AccelerometerTimeline`		■				Graph timeline representation of the numeric Phidget Accelerometer values.
`ApplianceManager`					■	Overview of all current running information appliances built with the Shared Phidgets toolkit. Provides controls to view interface skins, simulations, and observer windows.
`CircularTouchSkin`		■				User control for the Phidget CircularTouch device. This control wraps an `InterfaceKit` proxy component.
`ConnectionManager`				■		Component to handle the remote connection to the central `Connector` instance. Provides methods to specify the remote address, and to open or close the connection.
`ConnectionManagerSkin`				■		User control to enter the remote address and open/close a connection.

Table B.1: .NET components in the developer library (A-C).

Name	Proxy Component	Interface Skin	Simulation	Connection	Management	Description
`ConnectionManagerSkinTiny`				■		Very small user control to enter the remote address and open/close a connection. Opens a dialog window to specify the remote address.
`DeviceContainer`					■	Represents a single hardware device, and displays the device properties.
`DeviceManager`					■	Provides event handlers to get notifications of attached and detached hardware devices.
`DeviceManagerSkin`					■	User control with a list view of all currently attached hardware devices.
`DeviceSelection`					■	User control that provides various methods for the user to select a physical hardware device.
`DictionaryManager`					■	User control that provides a list view of all key/value pairs in the shared dictionary. Provides controls to search, edit, add, and delete entries.
`EditObject`					■	User control to edit value entries of the shared dictionary; (i. e., strings, boolean, bitmap, integers, doubles).
`Encoder`	■					Proxy component for the Phidget Encoder hardware; measures rotation.
`EncoderSkin`		■				User control for the Phidget Encoder hardware; displays rotation and status of the digital input.
`ExpandableGroupBox`					■	Extends the default .NET GroupBox with controls to collapse/expand the view.
`GPS`	■					Proxy component for the GPS signal receiver.
`GPSSkin`		■				User control that displays the current longitude and latitude of the GPS signal.
`GraphicLCD`	■					Proxy component for the colour graphic LC displays.
`GraphicLCDSkin`		■				User control for the colour graphic LCD. Provides methods to send test images, write text messages, draw sketches, and choose transition effects.

Table B.2: .NET components in the developer library (C-G).

Name	Proxy Component	Interface Skin	Simulation	Connection	Management	Description
`GraphicsMenu`					■	Provides methods to work with a selection menu on a graphic LCD.
`GraphicsText`					■	Provides methods to display text messages with images on a graphic LCD (with templates).
`GraphSkin`		■				Graph timeline visualisation for numeric sensor values.
`GSMGateway`	■					Proxy component for a GSM gateway; can be used to send and receive text messages from mobile phones.
`GSMGatewaySkin`		■				User control for the GSM gateway hardware. Users can send text messages and view all incoming messages.
`ImageCollection`					■	Container object for a collection of digital images.
`InterfaceKit`	■					Proxy component to access the Phidget InterfaceKit hardware. These I/O boards include digital inputs and outputs as well as analog sensor inputs.
`InterfaceKitSkin`		■				User controls to view the status of the input and outputs of the Phidget InterfaceKit.
`LED`	■					Proxy component to control the Phidget LED hardware; can control the brightness of up to 64 LEDs.
`LEDSkin`		■				User control for the Phidget LED hardware; allows the user to specify the brightness of the LEDs.
`LibraryInfo`					■	Displays version and release information of the Shared Phidgets toolkit library.
`MapControl`					■	User control that displays the geospatial map visualisation of the Shared Phidgets infrastructure.
`MotorControl`	■					Proxy component for the Phidget Motor Controller hardware.
`MotorControlSkin`		■				User control that allows the user to specify the acceleration and velocity of up to two connected motors.
`RFID`	■					Proxy component for the Phidget RFID reader.
`RFIDSkin`		■				User control that displays the incoming received RFID tags.
`SensorSkin`		■				Graph timeline visualisation for the sensor values of an analog sensor input of an InterfaceKit.
`Servo`	■					Proxy component for the Phidget Servo hardware; controls up to four servo motors.
`ServoSkin`		■				User control that allows users to specify the position of the Phidget servo motors.

Table B.3: .NET components in the developer library (G-S).

Name	Proxy Component	Interface Skin	Simulation	Connection	Management	Description
SimulationAccelerometer			■			User control to simulate an Phidget Accelerometer hardware device.
SimulationGraphicLCD			■			User control to simulate a graphic LC display.
SimulationGSM			■			User control to simulate a GSM gateway and mobile phones.
SimulationInterfaceKit			■			User control to simulate a Phidget InterfaceKit.
SimulationManager			■			User controls that allows the creation of multiple simulated devices.
SimulationRecording			■			User control to record/playback a time series of sensor and hardware events.
SimulationRFID			■			User control to simulate a Phidget RFID hardware.
TagManager					■	Facilitates the association of RFID tags to digital information (e.g., files, devices, names).
TagManagerSkin					■	User control that enables users to create associations between RFID tags and digital information.
TextLCD	■					Proxy component to access the Phidget Text LCD hardware.
TextLCDSkin		■				User control to specify the displayed text of the Phidget Text LCD.
UniversalSkin		■				Generic user control that displays the corresponding interface skin for a specified hardware (e.g., by setting the dictionary path).
UniversalSkinSmall		■				Smaller version of the user control that works similar to the UniversalSkin.
WeightSensor	■					Proxy component to access the Phidget Weight Sensor hardware.
WeightSensorSkin		■				User control that displays the current measured weight of the sensor.
XMLPersistence					■	Allows the persistent storage of shared dictionary entries in an XML file.

Table B.4: .NET components in the developer library (S-X).

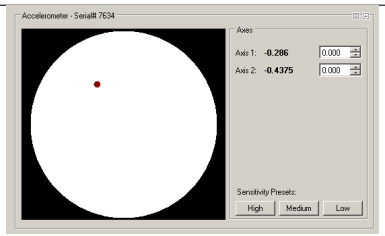
APPENDIX C

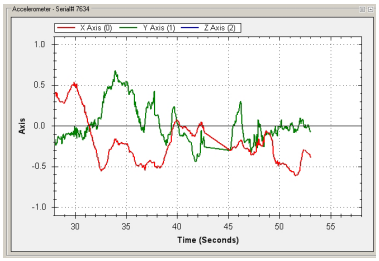
Implemented Hardware Devices

The following tables give an overview of the most commonly used hardware components that are implemented in the Shared Phidgets toolkit developer library. The information includes the corresponding API proxy objects in the *.NET* developer library and the important properties and event handlers of the objects. It also shows the available graphical interface skins for the hardware. These interface skins are included in the developer library as *.NET* user controls.

Accelerometer Hardware

Implementation of the *Phidget Accelerometer*: this sensor can measure the acceleration of movement, as well as tilting of the sensor between -90 and +90 degrees. Two versions are available: two and three axis accelerometer.
The `AccelerometerSkin` displays an overview of the current position of the axis, and the sensitivity of the measurement (including three preset configurations). The `AccelerometerTimeline` displays a graph representations of the last 30 seconds of the two or three accelerometer axis values.





Application:

- Detection of movement and simple gestures.
- Determine the orientation of a device (and orientation changes).
- Input controllers: navigating menus, select items, etc.
- Detect vibration or shaking of the sensor.

API Methods and Properties:

- `accelerometer.Axes[<index>].Acceleration`: The current value of the axis, as double value between -1.0 and 1.0.
- `accelerometer.Axes[<index>].Sensitivity`: The sensitivity of the axis, as double value between -1.0 and 1.0. The `AccelerationChange` event notifies subscribers if the delta change of the axis is higher than the sensitivity threshold.

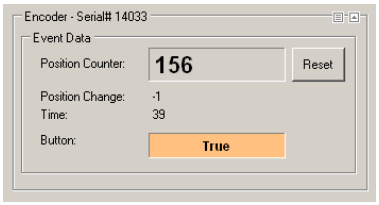
API Events:

- `accelerometer.AccelerationChange`: Notifies if the acceleration value of the axis has changed (depending on the current sensitivity value). This is a thread-safe event, and can be used for UI changes.
- `accelerometer.AccelerationChangeAsynchronous`: Notifies if the acceleration value of the axis has changed (depending on the current sensitivity value). This event is not thread-safe, but provides faster asynchronous event notifications.
- `accelerometer.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.1: Accelerometer hardware and API.

Encoder Hardware

Implementation of the *Phidget Encoder*: the encoder can measure the rotation (for instance of a dial or a motor). The encoder also includes a digital input that can be used as selection button. Extended versions of the encoder controlling board can connect to multiple encoders.
The `EncoderSkin` shows the current position of the encoder dial, and the status of the digital input button.



Application:

- Input: input control with continuous rotation (e. g., dial) and selection (by pressing the button).
- Measuring: the encoder can measure rotations (e. g., of an attached motor).

API Methods and Properties (Subset):

- `encoder.Encoders[<index>].Position`: Get the position of the encoder.
- `encoder.Encoders.Count`: Get the number of connected encoders.
- `encoder.Inputs[<index>].State`: Get the status of the digital input.
- `encoder.Encoders.Count`: Get the number of digital inputs.

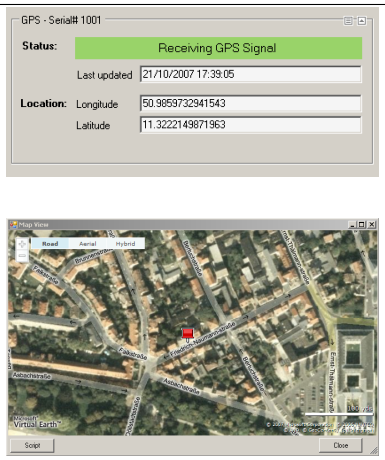
API Events (Subset):

- `encoder.PositionChange`: Event notification when the position of one of the encoders changes.
- `encoder.InputChange`: Event notification when the status of one of the digital inputs changes.
- `encoder.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.2: Encoder hardware and API.

GPS Hardware

Implementation of a *GPS Receiver*: the device provides the longitude and latitude coordinates of the current location. The implementation can be used with NMEA compatible devices (e. g., Microsoft Pharos GPS-360).
The `GPSSkin` shows the longitude and latitude coordinates of the current location, and the timestamp of the last update.



Application:

- Location-aware applications: build applications that can provide information dependent on the current location.
- Locate objects or people.
- Update the geographical location of the hardware devices (e. g., the location of situated displays in the environment).
- Associate information to a specific location.

API Methods and Properties (Subset):

- `gps.LastPosition.Longitude`: Get the longitude coordinate of the last position as double value.
- `gps.LastPosition.Latitude`: Get the latitude coordinate of the last position as double value.

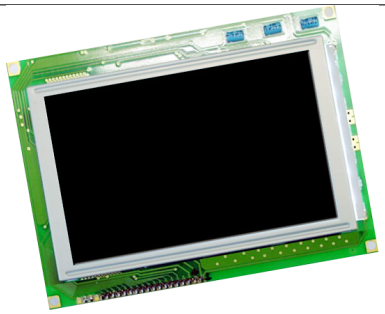
API Events (Subset):

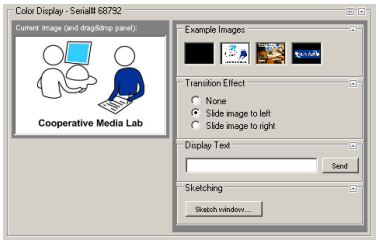
- `gps.GpsPositionChanged`: Event notification when the current location changes (new longitude and latitude coordinates).
- `gps.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or if an error occurred.

Table C.3: GPS hardware and API.

GraphicLCD Hardware

Implementation of the wired or wireless *graphic colour LCD*. In the current implementation these displays are based on Windows Mobile 5 hardware, and the displays are connected over the wireless network. The screen of the device can display images up to a resolution of 320x240 pixel (larger images are automatically scaled down to this resolution). A future release of the plug-in will support the *ezLCD screens* (http://www.ezlcd.com/).
The graphical skin displays the current image, and includes functions to send text messages, drawn notes, or image files to the display. The skin supports drag and drop of image files.





Application:

- Displays can be used for situated and ambient displays.
- Displaying text messages or notes.
- Displays for event notifications and reminders.
- Visual feedback for a local or remote located controller.
- Status display.

API Methods and Properties (Subset):

- `graphiclcd.Image`: Set the image to display (images that are larger than the screen resolution are scaled down).
- `graphiclcd.Text`: Displays a text message on the display.
- `graphiclcd.Transition`: Change the transition effect (is used when new images are displayed). The `GraphicLCD` class includes constant members for the transition settings: sliding image to the right or left side of the screen, or deactivate the transition.

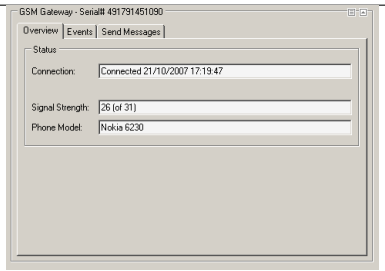
API Events:

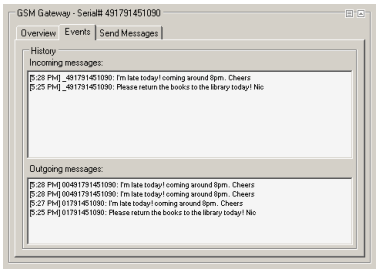
- `graphiclcd.DisplayedImageChange`: Notifies if the displayed image of the graphic LC display was changed.
- `graphiclcd.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.4: Graphic LCD hardware and API.

GSM Gateway Hardware

Implementation of a *GSM gateway*: by using the GSM gateway (e. g., via a connected GSM phone modem) it is possible to send and receive text messages from mobile phones (SMS).
The `GSMGatewaySkin` displays the current status of the connected GSM modem (e. g., phone number, connection status). It also displays the sent and received messages, and can be used to send text messages directly.





Application:

- Information: send users information about an important occurred event.
- Control: let users send text messages with commands to control the environment (e. g., *lights on* or *lights off)*.
- Location-based messaging: users can send messages and reminders to situated displays that are for instance at different locations inside of a building.

API Methods and Properties (Subset):

- `gsmgateway.SendMessage(string number, string message)`: Send a text message to the specified phone number.
- `gsmgateway.PhoneModel`: Get the model of the connected GSM modem.
- `gsmgateway.SignalStrength`: Get the signal strength of the GSM modem.

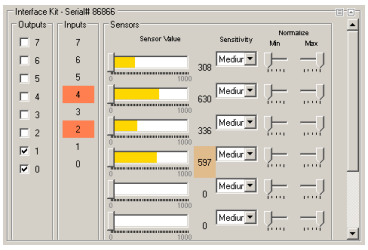
API Events (Subset):

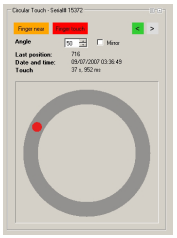
- `gsmgateway.ReceivedSms`: Event notification when new SMS messages are received.
- `gsmgateway.SentSms`: Event notification when new SMS messages are sent.
- `gsmgateway.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.5: GSM gateway hardware and API.

InterfaceKit and Sensor Hardware

Implementation of the *Phidget Interface Kit*: this generic I/O board has multiple digital inputs and outputs, as well as analog inputs (for sensors). There exist different hardware versions, with varying numbers of inputs and outputs.
The `InterfaceKitSkin` displays all inputs and outputs, and their current value (digital status or analog value).
The `SensorSkin` displays a graph plot of a single analog sensor of the InterfaceKit.
The `CircularTouchSkin` is a graphical interface for the circular touch sensor, a special form of the `InterfaceKit`.





Application:

- Digital input: connect physical switches, buttons, contact sensors, reed sensors, etc.
- Digital outputs: control lights, signals, relays (e. g., for controlling power outlets), etc.
- Analog sensors: measure temperature, movement, distance, force, pressure, rotation, etc.

API Methods and Properties (Subset):

- `interfacekit.Sensors[<index>].Value`: Get the current value of the analog sensor as integer value between 0 and 1000.
- `interfacekit.Inputs[<index>].State`: Get the status of the digital input as boolean value: true or false.
- `interfacekit.Outputs[<index>].State`: Get/set the status of the digital output as boolean value: true or false.

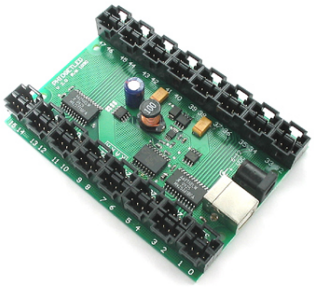
API Events (Subset):

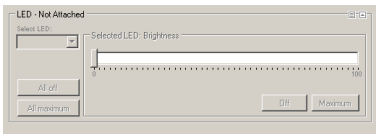
- `interfacekit.InputChange`: Notifications when the status of one of the digital inputs changes.
- `interfacekit.OutputChange`: Notifications when the status of one of the digital outputs changes.
- `interfacekit.SensorChange`: Notifications when the value of one of the connected sensors changes.
- `interfacekit.SensorChangeAsynchronous`: Notifications when the sensor value changes (event is not thread-safe).
- `interfacekit.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.6: InterfaceKit and sensors hardware and API.

LED Controller Hardware

Implementation of the *Phidget LED Controller*: with this controller board, up to 64 LEDs can be activated/deactivated. The brightness of each LED can be changed separately, as a value between 0 and 100 (in contrast to the InterfaceKit that can only activate/deactivate up to 8 LEDs).
The `LEDSkin` provides controls to change the brightness of each LED, and to activate/deactivate all LEDs at once.





Application:

- Illumination: use LEDs as light sources to illuminate objects.
- Signals: they can be used as signals and feedback indicators.

API Methods and Properties (Subset):

- `led.Lamps[<index>].Value`: Change the brightness value of one of the LEDs (integer value between 0 and 100).
- `led.Lamps[<index>].SetToMaximum()`: Set the brightness of the LED to the maximum level (brightness = 100).
- `led.Lamps[<index>].SetOff()`: Deactivate the LED (brightness = 0).
- `led.SetAllToMaximum()`: Set the brightness of all LEDs to the maximum level (brightness = 100).
- `led.SetAllOff()`: Deactivate all LEDs (brightness = 0).

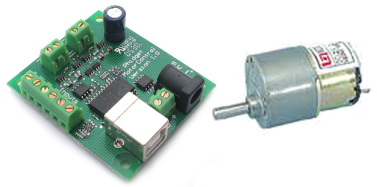
API Events (Subset):

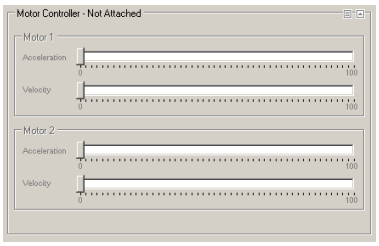
- `led.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.7: LED controller hardware and API.

Motor Controller Hardware

Implementation of the *Phidget Motor Controller*: two connected motors can be controlled by changing the acceleration and velocity.
The `MotorControlSkin` provides controls to change acceleration and velocity of the two motors from the GUI.





Application:

- Rotations: with a connected gearbox the motors can be used similar to servos, but with more control over the rotations (e. g., 360 degrees).
- Traction units: the motors can be used to move objects (e. g., with connected wheels).

API Methods and Properties (Subset):

- `motorcontrol.Motors.Count:` The number of available motors.
- `motorcontrol.Motors[<index>].Acceleration:` Set the acceleration of the motor.
- `motorcontrol.Motors[<index>].Velocity:` Set the velocity of the motor.
- `motorcontrol.Inputs.Count:` The number of available digital inputs.
- `motorcontrol.Inputs[<index>].Status:` Get the status (true/false) of the digital input of the motor controller.

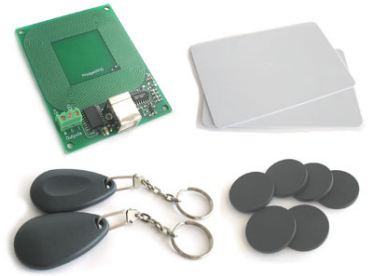
API Events (Subset):

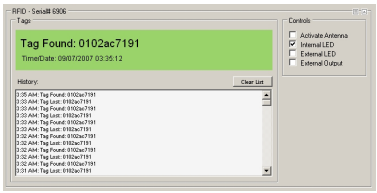
- `motorcontrol.InputChange:` Notifications when the status of an digital input changes.
- `motorcontrol.Attach/Detach/Error:` Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.8: Motor controller hardware and API.

RFID Reader Hardware

Implementation of the *Phidget RFID Reader*: the reader can identify RFID tags up to a distance of 10–15 cm. All RFID tags have a unique 16 digit hexadecimal identification. The reader has no collision detection; therefore, only one RFID tag can be identified at the same time.
The `RFIDSkin` displays the found RFID tags, and includes controls to activate/deactivate the antenna and the digital outputs (e. g., LED).





Application:

- Identify objects (e. g., paper, cards, boxes) by attaching RFID tags to them.
- Identification of people (e. g., with RFID keyfob).
- Identify the current location (e. g., with associations between RFID tags and a geographical location).

API Methods and Properties (Subset):

- `rfid.Antenna:` Activate or deactivate the antenna of the RFID reader.
- `rfid.LastTag:` The last RFID tag detected by the reader.
- `rfid.Output[<index>].State:` Get/set the status of the digital output as boolean value: true or false.

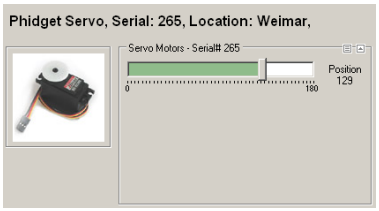
API Events (Subset):

- `rfid.Tag:` Event notification when an RFID tag is found.
- `rfid.TagLost:` Event notification when an RFID tag is lost.
- `rfid.Attach/Detach/Error:` Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.9: RFID reader hardware and API.

Servo Hardware

Implementation of the *Phidget Servo*: the position of the servo motors can be set to any value between 0 and 180 degrees. Two versions of the servo controller exists: a single servo controller, and a controller for up to four servo motors.
The `ServoSkin` shows the current position of the servo motors. A slider control can be used to change this position.



Application:

- Feedback display: indicate a value on a measuring scale.
- Move objects: the servos can be used to move objects (e. g., rotate a camera by 180 degrees).
- Force feedback: the servo can give feedback in a control device.
- Robotics: the servo motors are often used as engines for robots.

API Methods and Properties (Subset):

- `servo.Motors[<index>].Position`: Get/set the position of the specified servo motor.
- `servo.Motors.Count`: Get the number of controlled servo motors (one or four motors).

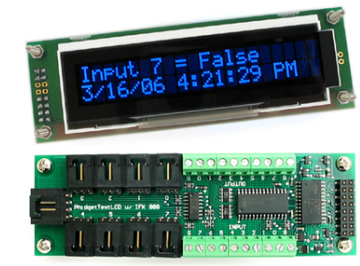
API Events (Subset):

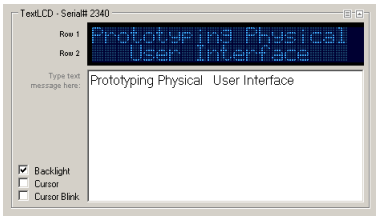
- `servo.PositionChange`: Event notification when the position of one of the servo motor changes.
- `servo.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.10: Servo hardware and API.

TextLCD Hardware

Implementation of the *Phidget Text LCD*: this LC display can show alphanumeric text messages (two rows, each with 20 characters).
The `TextLCDSkin` shows the current displayed message, and the user can change the text of the message. It also provides controls to activate/deactivate the backlight of the display.





Application:

- Situated Display: display messages or reminders.
- Feedback: provide feedback about the status of a device (e. g., display the current value of a sensor).

API Methods and Properties (Subset):

- `textlcd.Text`: Get/set the current message of the display.
- `textlcd.Backlight`: Activate/deactivate the backlight of the display.
- `textlcd.Cursor`: Activate/deactivate a cursor on the last changed text position.

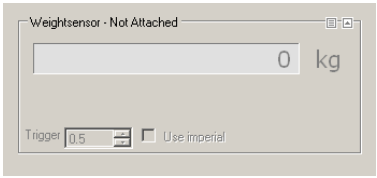
API Events (Subset):

- `textlcd.TextChanged`: Event notification when the displayed text message of the display is changed.
- `textlcd.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.11: Text LCD hardware and API.

Weight Sensor Hardware

Implementation of the *Phidget Weight Sensor*: this sensor allows the weight measurement of objects or persons.
The `WeightSensorSkin` provides the overview of the measured weight.



Application:

- Identify objects: use the weight sensor to identify objects based on their weight (and maybe in combination with computer vision).
- Keeping track of weight changes.

API Methods and Properties (Subset):

- `weightsensor.Weight`: The current measured weight (as double value).
- `weightsensor.WeightChangeTrigger`: The minimum delta difference of the weight value to trigger the event notification.

API Events (Subset):

- `weightsensor.WeightChange`: Event notifications when the sensors measures a weight difference (can be modified with the `WeightChangeTrigger`).
- `weightsensor.Attach/Detach/Error`: Default events of the `BaseComponent` class to send notifications if device is attached, detached, or an error occurred.

Table C.12: Weight sensor hardware and API.

APPENDIX D

Contents of the Thesis Project CD

The following content is included on the thesis project CD:

- `/bin/`
  Developer library and all executables of the toolkit. It is, however, recommended to use the Shared Phidgets setup to install the toolkit. This setup (located in the `/setup/` directory) installs the toolkit as well as all additional development tools.
- `/case_studies/`
  Case studies that illustrate the applicability of the toolkit. This includes various examples of distributed information appliances. These appliances are partially described in Chapter 7.
- `/setup/`
  The setup installs the toolkit and all development utilities. It automatically integrates the library into the Visual Studio IDE, adds the development template, and installs the Shared Phidgets IDE add-in.
- `/source/`
  The source code files of the Visual Studio 2005 development projects. All source code files of the project are written in *C# .NET*. Besides the source code files, the directories include the Visual Studio project files, images, binary resources, and XML files. An overview of these projects can be found in Appendix A.2.
- `/thesis_pdf/`
  The diploma thesis as Adobe PDF file (two versions: low and high resolution images).
- `/thesis_tex/`
  Source LaTeX files of the thesis chapters as well as the referenced figures as PNG files. This directory also includes the BibTeX reference library file.

Independence Statement

Herewith I declare that I have completed this work solely and with only the help of the mentioned references.

I authorise the Bauhaus-University Weimar and the University of Calgary to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Weimar, March 10, 2008

Nicolai Marquardt