Integrating Usability Models into Pervasive Application Development

Paul Holleis

München 2008

Integrating Usability Models into Pervasive Application Development

Paul Holleis

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

vorgelegt von
Paul Holleis
aus Bad Reichenhall

München, den 15.12.2008

Erstgutachter:	Prof. Dr. Albrecht Schmidt
Zweitgutachter:	Prof. Dr. Heinrich Hußmann
Externer Gutachter:	Prof. Dr. Antonio Krüger
Tag der mündlichen Prüfung:	19.01.2009

To my dad, who would have enjoyed reading this thesis.

Table of Contents

Abstract

This thesis describes novel processes in two important areas of human-computer interaction (HCI) and demonstrates ways to combine these in appropriate ways.

First, prototyping plays an essential role in the development of complex applications. This is especially true if a user-centred design process is followed. We describe and compare a set of existing toolkits and frameworks that support the development of prototypes in the area of pervasive computing. Based on these observations, we introduce the EIToolkit that allows the quick generation of mobile and pervasive applications, and approaches many issues found in previous works. Its application and use is demonstrated in several projects that base on the architecture and an implementation of the toolkit.

Second, we present novel results and extensions in user modelling, specifically for predicting time to completion of tasks. We extended established concepts such as the Keystroke-Level Model to novel types of interaction with mobile devices, e.g. using optical markers and gestures. The design, creation, as well as a validation of this model are presented in some detail in order to show its use and usefulness for making usability predictions.

The third part is concerned with the combination of both concepts, i.e. how to integrate user models into the design process of pervasive applications. We first examine current ways of developing and show generic approaches to this problem. This leads to a concrete implementation of such a solution. An innovative integrated development environment is provided that allows for quickly developing mobile applications, supports the automatic generation of user models, and helps in applying these models early in the design process. This can considerably ease the process of model creation and can replace some types of costly user studies.

Zusammenfassung

Diese Dissertation beschreibt neuartige Verfahren in zwei wichtigen Bereichen der Mensch-Maschine-Kommunikation und erläutert Wege, diese geeignet zu verknüpfen.

Zum einen spielt die Entwicklung von Prototypen insbesondere bei der Verwendung von benutzerzentrierten Entwicklungsverfahren eine besondere Rolle. Es werden daher auf der einen Seite eine ganze Reihe vorhandener Arbeiten vorgestellt und verglichen, die die Entwicklung prototypischer Anwendungen speziell im Bereich des Pervasive Computing unterstützen. Ein eigener Satz an Werkzeugen und Komponenten wird präsentiert, der viele der herausgearbeiteten Nachteile und Probleme solcher existierender Projekte aufgreift und entsprechende Lösungen anbietet. Mehrere Beispiele und eigene Arbeiten werden beschrieben, die auf dieser Architektur basieren und entwickelt wurden.

Auf der anderen Seite werden neue Forschungsergebnisse präsentiert, die Erweiterungen von Methoden in der Benutzermodellierung speziell im Bereich der Abschätzung von Interaktionszeiten beinhalten. Mit diesen in der Dissertation entwickelten Erweiterungen können etablierte Konzepte wie das Keystroke-Level Model auf aktuelle und neuartige Interaktionsmöglichkeiten mit mobilen Geräten angewandt werden. Der Entwurf, das Erstellen sowie eine Validierung der Ergebnisse dieser Erweiterungen werden detailliert dargestellt.

Ein dritter Teil beschäftigt sich mit Möglichkeiten die beiden beschriebenen Konzepte, zum einen Prototypenentwicklung im Pervasive Computing und zum anderen Benutzermodellierung, geeignet zu kombinieren. Vorhandene Ansätze werden untersucht und generische Integrationsmöglichkeiten beschrieben. Dies führt zu konkreten Implementierungen solcher Lösungen zur Integration in vorhandene Umgebungen, als auch in Form einer eigenen Applikation spezialisiert auf die Entwicklung von Programmen für mobile Geräte. Sie erlaubt das schnelle Erstellen von Prototypen, unterstützt das automatische Erstellen spezialisierter Benutzermodelle und ermöglicht den Einsatz dieser Modelle früh im Entwicklungsprozess. Dies erleichtert die Anwendung solcher Modelle und kann Aufwand und Kosten für entsprechende Benutzerstudien einsparen.

1 Introduction and Structure

The style of interacting with computer systems is about to change drastically. Clearly, the standard scenario where one person communicates with a single desktop computer through keyboard and mouse will remain important for the next couple of years. However, in many situations there is a shift towards a less clear boundary between computers and the world. There has been increasing attention in research, media and industry to distributed and mobile computing in the last years and there is strong evidence that this will manifest itself even stronger in the near future. The basic vision of pervasive computing is that devices and environments are getting smarter and the possibilities of input and output increase drastically. A variety of new applications have already become possible with the introduction of large public displays, radio frequency identification, powerful mobile devices, sensor networks, and a large number of available software components and services.

In this new space, methods like those for creating user interfaces or evaluating applications that worked well for the desktop setting have to be checked and often need to be revised. One widely used way of trying to create devices, applications, and environments that are really accepted by end-users is to take potential users into account from the very beginning of the development process (‘user-centred design’). This implies that, ideally, there should be input from users to the idea, design, as well as implementation of a system. Practically, this can be achieved in two ways (which are not mutually exclusive). The first is to divide the development process into several phases and for each phase build a prototype presented to a sample of potential users. The second possibility is to integrate models into the development process that represent important aspects of users. Since a model is per definition a simplification, this bears the risk of imprecise or incomplete data. On the other hand, it can be used whenever needed and reduces the need for cost intensive and time consuming user studies. We will show advantages and weaknesses of both ways and provide techniques and tools to support both of them.

Such models underlying the development of a system can provide characteristics, metrics, and properties that can be checked. If this can be done automatically, a number of errors and issues can be avoided. However, in practice, it is still important to generate prototypes of applications and devices in order to assess, communicate, and evaluate the ideas. Therefore, we provide a rich set of tools to create applications in the domain of pervasive computing. We also present ways of combining the power of prototyping with the expressiveness of user models. One of the prototyping environments we will describe is based on programming by demonstration and a user interaction time model. It thus allows quickly building functional prototypes and at the same time helps in identifying issues in task sequences and gives well grounded estimates of the time users will need to perform these tasks on the envisioned target platform.

1.1 Goals and Contributions

The key contributions of this thesis can be found in three areas.

Prototyping

- An introduction and detailed comparison of available hardware and software prototyping tools
- The design and implementation of the EIToolkit for rapid prototyping of pervasive applications
- Various examples and case studies developed based on that toolkit

User models

- An introduction to the application of user models for usability, e.g. the Keystroke-Level Model (KLM)
- Validated extensions to the Keystroke-Level Model for mobile interactions with the real world
- A demonstration of the use of graph theory methods for user interface and interaction design

Incorporating user models in the development process

- An extensible framework for building pervasive applications employing user models
- Easy to use interfaces to create prototypes and applications based on usability input from user models
- The design and implementation of MAKEIT, a prototyping framework for applications on mobile devices

1.2 Structure

After the introduction of important terms and an overview of pervasive computing, a set of related and own work is briefly presented that serves to show the design space of pervasive computing. Since one part of this thesis demonstrates new paths to design and implement pervasive and mobile applications, development processes of pervasive and mobile applications are concisely described in Chapter 2, Developing Pervasive Applications, including a brief overview of available methods, tools and frameworks to create tangible prototypes.

User models and their influences on user interface design play an important role in reaching the goals of this work. Therefore, existing cognitive and related user models are described in Chapter 3, User Models for UI Design. We concentrate on models that allow for making predictions with regard to interaction times of end-users and present important application areas of such models. One of those, the Keystroke-Level Model (KLM), is treated in more detail since it is used extensively in the following chapters. In order to be able to use the model in the new domain of pervasive computing, our work provides several extensions of the original concept to novel areas such as physical mobile interactions. The measurement and validation of new parameters for the KLM are described in conjunction with several examples. We built a variety of development tools on top of such models and can thus fully include their advantages early in the development process of pervasive applications.

One further specific contribution developed in this thesis is the EIToolkit, an open source toolkit designed to support the creation of pervasive applications. Its idea, architecture and implementation are presented in Chapter 4, Tools for Rapid Application Development. This section also contains a detailed set of requirements for such toolkits collected from a long list of related work and experiences. It influenced design decisions during the development of our tools and can serve to evaluate other toolkits or to pick a specific one for own purposes.

The following Chapter 5, Prototyping Using EIToolkit and User Models, describes another central theme of this thesis, i.e. the combination of user modelling approaches and prototyping tools. After a description of the concept, a series of (e.g. graphical) tools on top of the EIToolkit is demonstrated. These can be used to quickly build applications using a variety of virtual and physical input and output components. They are intended to lower the threshold of building such applications while still maintaining a high ceiling, i.e. they allow creating complex applications without requiring more effort than necessary. The second part of this chapter concentrates on existing and new approaches to integrate models like those described in Chapter 1 into existing and novel development tools. In order to substantiate these ideas, we present examples of the integration of a KLM module in two existing application development environments.

Chapter 6, Case Studies – Applications Based on the EIToolkit, then shows a selection of applications and scenarios implemented using the EIToolkit. It distinguishes between two main branches. First, we developed several components that enable the use of some technology or technique without having to learn the details of their use. One example is the Particle microcontroller system that allows developers to quickly build smart, distributed sensors. Second, device and application specific components are described that connect the toolkit to specific devices or applications like remotely controllable power sockets and music player software.

These theories are then used in conjunction with the EIToolkit in the following Chapter 7, Prototyping Mobile Device Applications. It provides a working implementation of a standalone, integrated development environment for the creation of mobile phone applications. The chapter begins with a description of current tools to create applications especially tailored for mobile devices. After a short excurse on how graph theory can be used in the domain of user interface design, the MAKEIT environment and its use are presented in detail. It builds on state graphs and uses programming by demonstration in conjunction with several of the methods and tools described in previous chapters to easily generate mobile applications using models such as the KLM and building on EIToolkit support. Besides the process of creating such applications itself, implications for application design and several example applications developed using this system are given.

The final Chapter 8, Summary and Future Work, reviews the content of the thesis, restates the set of contributions and ends with a treatment of future and open work. It shows where the work presented here offers a solid basis for other developers to add dedicated extensions for specific application needs and where the research community can help to extend the systems to reach a further level of generality of the processes which can only be achieved by wide application, acceptance, and development.

2 Developing Pervasive Applications

This chapter introduces the research area of pervasive computing, discusses related terms and fields and shows in which ways this thesis contributes to several of these aspects.

The two main parts of this chapter are concerned with a description of pervasive computing and related terms (2.1) and the components of typical application development processes within this field (2.2). In the course of those descriptions, we show in which areas the contents of this thesis are most profoundly located and to which specific aspects it contributes most.

2.1 Pervasive Computing

In order to introduce the topic, place our research in the design space of applications and development, and to describe important terms and concepts, we begin with a concise overview on the topic of pervasive computing.

2.1.1 Brief History and Overview

Pervasive computing and ubiquitous computing are two terms that have caught much attention over the last years. However, the terms are still not used coherently in the literature. In this work, we use pervasive and ubiquitous (or ‘ubicomp’) applications) interchangeably, although they are sometimes used to carry somewhat different meaning. Niemelä and Latvakoski, for example, describe *pervasive* computing as to concentrate on user interactions with mobile and wireless devices, creation and deployment of applications, and the use of *ubiquitous* services to enhance the user experience [Niemelä and Latvakoski 2004]. However, ultimately, they also decided to follow the general movement in research and conclude in general not to distinguish between those two terms. We refer to, e.g., [Satyanarayanan 2001], who gives an overview of the history, some current work, and proposed research necessary to further tackle requirements of pervasive computing scenarios, most of which still hold even if some years have passed since then.

In 1991, Mark Weiser first published his vision of bringing computing beyond the desktop [Weiser 1991]. He deliberately distinguished it from virtual reality which uses a different, virtual, artificial environment which maps in some sort to the real world. He coined the term ubiquitous computing during his work at Xerox PARC (Xerox Corporation's research centre in Palo Alto, USA). Major concepts of this vision include the seamless integration of possibly hundreds of computers in the real world, that there is no need for the user to focus on a (specific) computer, and that any surfaces can be turned into a display and show arbitrary information. Two important concepts are location and scale. This means that a system can use information about the location of its users to adapt information and output accordingly. On the other hand, different situations need different types of utilities. One of the consequences is that helpful technology must be able to scale. For mobile use, input and output footprint must be very small; as soon as several people are involved or the space changes, larger screens and collaborative input possibilities become interesting.

One category of ideas within ubiquitous computing has started with the name *calm computing* and has later been called *ambient computing*. This subsumes all those types of applications that try to minimise information overload and convey information in a way that do not need the primary focus of the users. Examples are known

objects that can be changed according to some input variable, e.g., connecting the height of water fountains to the stock market, or colours of buildings or objects to the weather forecast. All of those have in common that information can be registered peripherally without interrupting a main task.

An interesting dilemma that emerges from those ideas is that the (distributed) objects can become very much specialised. This means that they can potentially be used to efficiently solve a small set of dedicated problems. However, it also means that one would have to have many of those devices at hand at any time to be prepared for a variety of applications. This favours devices that incorporate much functionality as we see today with mobile phones and PDAs. Unfortunately, most of those all-in-one devices are less effective in use when applied to a specific problem. One example that relates to the scalability issue mentioned above is text input. A desktop keyboard enables rather comfortable and quick text input. Surely, it is less suitable for mobile use or in specific environments like the car.

Another issue is that, on the one hand, technology is supposed to disappear (see for example the Disappearing Computer Initiative[1] for a range of related projects) so as to be indistinguishable from the object and environment one is used to. On the other hand, users often need or want to have control over what happens. On occasion, feedback or input from the user is necessary. More often even, as soon as a person's mental model of a system does not fit to the system's behaviour, this person needs access to the system to control, change, or just understand what is happening. Thus, the system must either be visible in some way or there must at least be some method to gain access of its input and output capabilities as well its way of working. The term 'affordance' gains enormous weight here. It defines the way an object or application describes itself and the way it can be used without previous knowledge or learning phase. If a user interface element looks like a door handle for example, it is very likely that people will try to press it down to initiate an action. A pilot's instrument panel on the other hand has in general not a clear affordance and implies a steep learning curve. One of the aims in pervasive computing is to exploit the affordances of physical objects and transfer it to other possible applications.

One way to achieve that is to stick to these physical objects as handles into the computational world and introduce technology such as processors, sensors, actuators, and input / output capabilities directly into such objects. This enables implicit interactions and context aware applications to be built that do not or only slightly change the original behaviour. By adding a set of sensors to a camera, as we did for example in [Holleis, Kranz, and Schmidt 2005a], it is possible to support users in their interactions by automatically rotating pictures and recording environmental and location information for later classification. Other approaches to generate applications with high affordance have been to use virtual representations of physical objects that people know how to operate like files and folders, or buttons, tabs, and form fields. Of course, the indirect manipulation through mouse and keyboard drastically reduces the similarity effect. One attempt to remedy this situation is to use physical handles such as our display cube [Terrenghi et al. 2005] or an object with physical knobs and buttons in combination with larger displays such as in [Hilliges, Baur, and Butz 2007].

Pervasive computing draws from several topics in computer science and other research fields. This is one reason why several terms and notions have been created and emerged to describe the whole area as well as different subjects within the design space covered by corresponding applications. The next section gives an overview of many facets and ideas with which pervasive computing is concerned.

[1] The Disappearing Computer, an EU-funded initiative of the Future and Emerging Technologies (FET) activity of the Information society Technologies (IST) research program; project page: *http://www.disappearing-computer.net*

2.1.2 Applications and Related Terms

As is always the case with an area that is as complex and steadily changing, there exist many definitions and ways to describe pervasive computing. For the sake of this thesis, we adopt the one given by the editors of the Journal of Pervasive and Mobile Computing:

> "The goal of pervasive computing is to create ambient intelligence where network devices embedded in the environment provide unobtrusive connectivity and services all the time, thus improving human experience and quality of life without explicit awareness of the underlying communications and computing technologies. In this environment, the world around us […] is interconnected as pervasive network of intelligent devices that cooperatively and autonomously collect, process and transport information, in order to adapt to the associated context and activity."
> [Das, Conti, and Shirazi]

It catches the most important aspects of distributed devices and services, i.e. anywhere and anytime access to information. It also promotes the adaptation and use of contextual information to make people's tasks better, quicker, or more comfortable in a way that they only notice this support as little as possible. One of the best ways to put that last part into words is a famous statement of Mark Weiser about ubiquitous computing:

> "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]

From that perspective also stem the terms describing very similar concepts namely *disappearing computer* and *calm computing*. When the use of real world objects and tangible devices is emphasized, pervasive computing also sometimes appears using the names *physical computing* or *tangible media*. Another more recently coined term by Greenfield in [Greenfield 2006] is *everyware*. By stressing the fact that sensing, processing, and services are available anywhere and anytime, he tries to bring the many facets of pervasive computing into one coherent paradigm of interaction. This particular process of spreading components, intelligence, and services in the world has also led to the notion of *distributed computing* in conjunction with ubiquitous computing; see for example [Barbeau 2002] for a treatment of that connection.

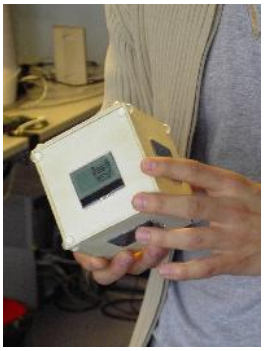
With all these different components that make up pervasive applications, it is difficult, if not impossible, to completely classify their design space since nearly all possible applications have some or can be augmented with some aspects of pervasive computing. In order to give an overview on what kind of applications have traditionally been built in the area of pervasive computing, the following presents a list of common categories and gives examples for each. This is not an exhaustive list in the respective area; the given samples denote some recent, prominent or very typical application areas.

The categories are: tangible computing, embedding information, ambient computing, context aware applications, wearable computing, augmented and virtual reality, computer supported cooperative work (CSCW), communicating information, mobile applications, and input in pervasive computing.

In our work, we contributed to several of those categories with various projects developed in our research group. A selection of these projects is mentioned in the following together with representative examples of works from other research groups. Whenever we directly reference our own work, we denote this by a footnote ([2]). Some of them provide additional information on some of the topics treated in detail in this thesis and will be elaborated in appropriate later sections. Some images from our projects are shown to the right of the descriptions.

Tangible Computing

Even if one of the paradigms followed with pervasive computing is that technologies should step into the background of any user interface or interaction, one line of research is concerned with making information and its manipulation visible and tangible. As an example, we built a cube with displays as its six sides, [Kranz, Schmidt, et al. 2005][2]. One of its many applications can be found in learning, where for example multiple-choice tests, image associations, or 3D projection tests can be performed [Terrenghi et al. 2005][2]. The cube allows grasping, playing with, and passing information (in this case the whole application) to others, opening new possibilities.



A cube with displays as its sides, [Terrenghi et al. 2005]

This area also includes many works that produce haptic feedback in addition to the most commonly used visual and auditory channels. A simple example is the vibration motor found in most modern phones to convey an almost inaudible signal [Sahami et al. 2008]. As an advanced example, consider the Phantom device, [Salisbury and Srinivasan 1997], which can simulate the forces that apply when touching objects. This can be used, e.g., for training of medical surgery.

The concept of tangible computing is also described by other terms like *passive real-world props*, *graspable*, *manipulative*, and *embodied interfaces* [Fishkin 2004].

Small wireless displays that can display information about each others' state, [Holleis, Kranz, and Schmidt 2005b]

Another very active research area is tabletop and surface computing. Applications range from games (see [Magerkurth et al. 2005] for an overview on computer augmented entertainment and tabletop games in particular), to music generation (e.g. the reacTable, [Jordà et al. 2007] and [Kaltenbrunner and Bencina 2007]), to computer supported cooperative work (CSCW, see below, and the thesis [Ringel Morris 2006] for a good overview on this topic).

Embedding Information

As described below, there exist many possibilities to sense the current context of the user or an application. However, this generates issues in areas such as privacy, accuracy, power consumption, etc.

Embedding information directly places information at the spot where it is of interest, see [Schmidt, Kranz, and Holleis 2004][2] and [Schmidt, Kranz, and Holleis 2005][2]. For example, information about the weather is displayed close to where the umbrella or the clothes are located [Matthews et al. 2004], or directly in the umbrella[3]. As we argue in [Holleis, Rukzio, et al. 2006b][2], it is not even absolutely necessary to have such displays fixed at a certain position. It can also be left to the users to move them around and put them in places suitable for their purpose. See also a more detailed treatment in Section 6.2.1.

Hanger for clothes with embedded sensors and several displays. [Schmidt, Kranz, and Holleis 2004]

Public, and especially situated displays, are currently very actively studied by many research groups. The papers in [O'Hara, Perry, and Churchill 2003] give insight into social, technical, and interactional aspects in this area.

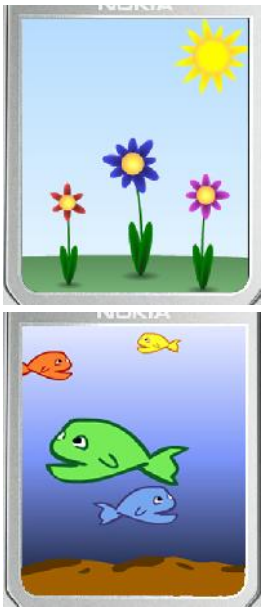
[2] One of the projects created within the DFG project Embedded Interaction by the author of this thesis and colleagues; home page of the research group: *http://www.hcilab.org*

[3] Ambient Devices, Ambient Umbrella; product page: *http://www.ambientdevices.com/products/umbrella.html*

Ambient Computing

Ambient devices build on the fact that in an arbitrary environment, a human concentrates on a small part of it at any one time. Besides this area in focus, there is a large area in the periphery where some information can be registered without being disturbed in one's main task. One can notice people entering a room, lights being switched on, etc. Some of those even only enter the mind unconsciously like types of background music. Ambient displays exploit this fact and show information supposed to be less important than the foreground task but interesting enough that a user wants to stay up-to-date. A widely known ambient device is the ambient orb, a spherical lamp that can change the colour of its light with soft transitions. It is, e.g., used in [Matthews et al. 2004] as an example of an ambient motion monitor that conveys a notion of remote activity. Extended versions are currently being developed. One of the first examples of an ambient device is the dangling string (created long before it was mentioned in [Weiser and Brown 1996]) which is a string dangling from the ceiling that adapts its speed of turning to indicate network flow.

Other appliances include applications like the ambient umbrella, or the use of screensavers. We implemented the latter concept on mobile phones to support people in remembering and getting reminded about their personal communication and location behaviour [Schmidt, Häkkilä, et al. 2006][2]. Several projects also combine ambient information displays with the possibility to access more detailed information through, e.g. a mobile device, see for example [Prante et al. 2004].



Mobile phone screensavers displaying information about communication behaviour, preserving privacy, [Schmidt, Häkkilä, et al. 2006]

Context Aware Applications

Applications in the area of embedding information get most of the context they need from the direct placement in the desired surroundings. Most ambient devices receive their data input from few specific channels to not overload their complexity. In contrast to that, context aware applications rely on their capability to retrieve context from the environment to adapt their appearance and behaviour. A simple example is described in [Holleis, Kranz, and Schmidt 2005a][2], where context information such as location and temperature is added to digital pictures the very moment the pictures were taken. The term context itself has received much attention and many definitions exist that differ in detail. Dey gives a short discussion on that topic and we use his definition here:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [Dey 2001]

Much work has been put into location based services [Steiniger, Neun, and Edwardes 2006] like a myriad of nearby restaurant recommendation finders, mobile tourist guides [Ballagas, Kratz, et al. 2007], or presence systems [Kranz, Holleis, and Schmidt 2006][2]. However, as stated in [Schmidt, Beigl, and Gellersen 1998], this is not the only type of context to sense. Besides relative or absolute location, they list many additional types of sensors that can sense low-level types of context, e.g., temperature, humidity, touch, movement, and orientation. This input can be used to infer higher-level context, e.g. emotions, mood, or activity (sitting / walking, reading / watching, running for the bus / fitness, see for example the special issue of the IEEE Pervasive Computing magazine [Davies, Siewiorek, and Sukthankar 2008], for additional recent work).

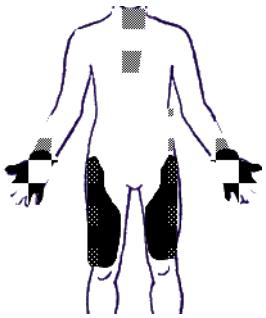
[Holleis, Kranz, and Schmidt 2005a]; sensor box and picture with corresponding sensor data; thickest line is compass data

Wearable Computing

Sensing context as described in the last item often needs special placement of a variety of sensors. Sometimes, a single sensor can be enough to get an idea of movements in a whole house, as for example described in [Patel, Reynolds, and Abowd 2008]. However, most of the time, the explicit placement of sensors is vital. Especially for activity recognition, the ability to have sensors directly on the body of the user can be of much help, see e.g. [Laerhoven and Gellersen 2004] for a description of detecting motion and pose of a person by using several sensor nodes distributed in that person's clothing. This is included in the research area of wearable computing. Already in 1998, much research has gone into the creation of fabric computer interfaces, see e.g. [Orth, Post, and Cooper 1998]. More recently, Linz, Kallmayer et al. show in [Linz et al. 2005] how to use flexible electronic modules and a way of connecting them with conductive yarn. This can be seen as an enabling basis for such work as [Holleis, Paasovaara, et al. 2008][2] which looked into the placement and usability of touch sensors on clothing. It could also make it easier to deploy technology such as the Lilypad Arduino, [Buechley, Eisenberg, et al. 2008], a prototyping platform for applications integrated into clothes.

Phone bag with touch controls integrated in the design. Prototype from [Holleis, Paasovaara, et al. 2008]

Wearable computing also touches other aspects like fashion design and retrieving energy from body movements to power embedded devices and sensors, see [Mateu and Moll 2005]. Besides input – be it explicit through touch or wearable keypads, or implicit by collecting and interpreting sensor data – various methods of data output have also been studied. Famous examples are video enhanced glasses or goggles, headphones, vibration motors etc. Many augmented reality applications as described next are based on such extended wearable accessories.



The picture of the body below indicates regions where people would expect touch input on clothing to be, see [Holleis, Paasovaara, et al. 2008]

Augmented and Virtual Reality

This subject is concerned with applications that either add to or completely replace how we perceive the real world. The first concept is known as augmented reality (AR), the second as virtual reality (VR). Virtual reality lets the user completely immerse into a virtual world which can either be realistically modelled using real places, persons, and textures, or abstract showing for example concepts or data structures. Possible applications range from entertainment [Benford, Magerkurth, and Ljungstrand 2005], to tourist information [Díez-Díaz, González Rodríguez, and Vidau 2007], to training and simulation scenarios, e.g. in the area of medical surgery (see [Neubauer 2005] for a broad overview as well as technical details on the subject). Many of those use head mounted displays and high-end computing machines to directly process and add information to the visual channel. Recently, however, mobile devices have caught up in terms of processing power and graphical capabilities. Applications such as an automatic, PDA-based translator for street signs seen through the built-in camera [Zhang et al. 2002], or mobile, collaborative games [Wagner et al. 2005] have become possible. The high mobility of the user in general imposes a non-trivial problem on the implementation of such applications since the discrepancy between the user or the device and the environment has to be measured very precisely and then computationally eliminated. Some generic work like the ARToolkit[4] can be used to simplify such processes. As closely as augmented reality and wearable computing are related, much research combines different areas like for instance AR and tangible computing, see [Bianchi et al. 2006] for an example.

[4] ARToolkit at HITLab, University of Washington; project page: *http://www.hitl.washington.edu/artoolkit/*

Computer Supported Cooperative Work (CSCW)

Computers have always been built to aid and support work done by their users. However, the rather fixed setting and limited input and output capabilities of the standard desktop systems have hindered the collaborative aspect that is found very important in small and larger teams. The importance and long history of the field of computer supported cooperative work can be seen in the dedicated CSCW conference series[5] that started as early as 1986 and its various related conferences and workshops. A variety of smart meeting rooms with support for video and audio conferencing and quick and easy transfer and sharing of information such as virtual and physical documents has been introduced over the years, see for example [Jaimes and Miyazaki 2005].

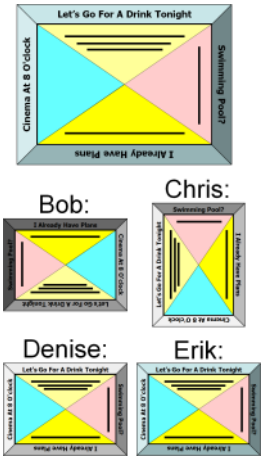
This supports collaboration between teams at remote locations. Still, even when team members are co-located, perhaps in the same room, it is not trivial to find the right infrastructural and interface means to combine physical and virtual information in a way that it really improves interaction styles. Tabletop surfaces suggest themselves as mediator for various types of scenarios where several people can work together on one common subject. A comprehensive overview of recent research and technological prototypes (even though it mostly focuses on work from the Stanford University) can be found in [Morris et al. 2006]. Of course, these approaches have strong links to augmented reality and tangible computing.

Communicating Information

This is a very broad area and one of those where it shows most prominently that pervasive computing includes experts and research from a great variety of fields. Projects dealing with communicational issues can range from very low-level networking protocols and hardware cable design to more high-level technologies like VOIP (voice over IP, i.e. internet telephony) and video conferencing to specialised applications that incorporate various input and output channels. The project we describe in [Holleis, Kranz, and Schmidt 2005b][2] incorporates several small, wirelessly connected, mobile displays without any visible input capabilities. A set of gestures can be used to communicate simple messages to the other displays, e.g. for remote voting: four possibilities are shown; to make a decision, the device is put on the table oriented in a specific way. Then, the position and amount of lines on the display show the votes of the other participants in that particular group. Another example concentrating on the output side is [Sahami et al. 2008] where we describe a mobile phone with several haptic actuators.

[Schmidt 2005]: information about connected alarm clocks is aggregated and displayed

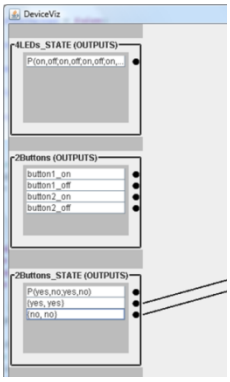
A contrasting approach that does not abstract information so much is shown in the Hug [Gemperle, DiSalvo, et al. 2003], which manifests the need for types of remote communication other than speech. It is a soft form that can comfortably be held in one's arms and, in addition to a phone which can provide an audio and possibly video connection, it uses an array of pressure and accelerometer sensors to convey to the remote party physical actions such as stroking or hugging (see also the newer HugShirt[6], which remotely sends a hug to another person's shirt through mobile phones connected with Bluetooth). In fact, much research goes into how to communicate one's state of mind, emotions and feelings to a remote person. A recent project for instance uses a small vibration motor attached to a normal ring to transfer the heartbeat of a friend in real-time, [Werner, Wettach, and Hornecker 2008]. In a similar project a few years earlier, we describe a standard alarm clock augmented with internet access [Schmidt 2005][2]. Through its connection to other alarm clocks, its functionality can adapt to the presence and state of remote clocks and persons like husband / wife or colleagues.



[Holleis, Kranz, and Schmidt 2005b]: a simple, tangible interface for voting over a distance

[5] CSCW Conference Series; CSCW'08 conference page: http://www.cscw2008.org
[6] Cute Circuit, HugShirt, 2006; product page: *http://www.cutecircuit.com/projects/wearables/thehugshirt/*

All those projects that focus on communicating information in a different way than by standard telecommunication means have in common that they often need to include expertise in hardware, software, HCI, and many more areas. Surely, most applications need to use some form of communication, be it between two people in the same building, between a larger group of people distributed all over the world, or also simply between the user and a computer. This implies that the problem of having to congregate diverse expertise arises for many pervasive computing applications. Fortunately, however, abstractions exist for many of those issues that can alleviate such problems. Many projects, for example, simply use existing infrastructure and communicate using HTTP, i.e. simple web access, and thus do not need to consider the choice and implementation of the concrete type of network communication (e.g. whether a device is connected by LAN, WLAN, Bluetooth, GPRS, etc.). In addition, toolkits exist – for instance our EIToolkit described in more detail in Chapter 4 – that relieve developers from the need to know hardware details of every possible sensor and also abstract from specific implementations.



DEVICEVIZ graphical, rule-based prototyping environment, see Section 5.2.2

Mobile Applications, Anywhere / Anytime Access

The success of mobile and pervasive computing applications in the last years has largely been driven by technological advances that made it possible to communicate wirelessly across wide distances, that enabled access to technology and information to a high percentage of the world's population, and that continuously reduce the amount of effort and money for such applications for both developers and users. As examples, a car is currently equipped with about 100 networked processors[7], dish washers employ processors and logics far beyond their real requirements, and modern mobile phones are more powerful than standard computers have been just a decade ago. According to analyst Informa Telecoms & Media, the number of mobile phone subscriptions has reached 3.3 billion, i.e. 50 % of the world's population in 2007[8] with 59 countries actually having more subscriptions than inhabitants. Over one billion phones have been sold in 2007[9] and many of these phones incorporate additional sensors and actuators such as GPS (Global Positioning System), accelerometers for orientation and gesture input, compasses, and cameras. This renders phones and other mobile devices like PDAs and PocketPCs a very interesting and powerful platform for pervasive computing applications.

[Holleis, Rukzio, Otto, et al. 2007]: paper prototypes and advanced phone interaction

As has been mentioned before in conjunction with the term 'everyware', pervasive computing builds heavily on the fact that many services can be accessed independent of time and place. Underlying infrastructure for the mobile web (using technology like GPRS, UMTS, protocols like WAP, services like i-mode, etc.) is now widely available. To overcome the small input and output areas of mobile devices, much research has been put into the use of public displays, see, e.g., [Tuulos, Scheible, and Nyholm 2007] of how to combine the personal, private screen of a phone with public displays. Interacting with non-private displays obviously has implications on privacy as information is potentially displayed that users do not want to share. Furthermore, interactions in front of a large display might especially attract interest by passers-by. In [Holleis, Rukzio, Otto, et al. 2007][2], we show that this is especially true if gesture input is used and can be exploited in a positive way for advertising purposes. Nevertheless, input to small or public devices is a difficult process. We will especially concentrate on mobile application development in Chapter 7.

[Holleis, Huhtala, and Häkkilä 2008]: capacitive sensors that sense touch and pressure add another input dimension to a phone keypad

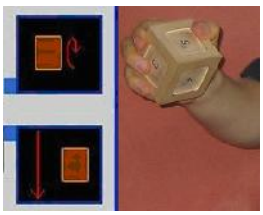
[7] Column at Embedded.Com "Motoring with Microprocessors": *http://www.embedded.com/columns/significantbits/13000166*
[8] News entry "Global Mobile Penetration Hits 50% Today", analysis of Informa Telecoms & Media: *http://www.telecoms.com/itmgcontent/tcoms/news/articles/20017483752.html*
[9] News entry "Cell Phone Sales Hit 1 Billion Mark", analysis from Gartner: *http://news.cnet.com/8301-10784_3-9881022-7.html*

Input in Pervasive Computing

Pervasive applications impose two specific problems on the way information can be fed into their systems. First, the user interface and the whole idea that there actually is an application running in the background should not be made explicit to the user. Thus it is difficult to provide non-obtrusive but still very much intuitive input modalities. Second, mobile users most often interact with their own mobile devices with tiny keypads or keyboards which makes text input very cumbersome. Besides various types of text input on small keyboards (see [Wigdor and Balakrishnan 2004] for an overview), speech [Starner 2002] and gestural input [Kranz, Freund, et al. 2006][2] has been analysed. This applies not only with respect to the mobile devices we carry but also in specific settings such as on the bike or in the car.

Images from a study with a cushion augmented with sensors [Holleis, Kranz, Winter, et al. 2006]

Even though the quality of speech recognition is on the increase, it might not be appropriate in all situations. Whenever the type of information that needs to be entered into a pervasive system does not need to be very elaborate, however, specialised methods can be used. For example, in [Holleis, Kranz, Winter, et al. 2006][2], body movement on a chair is used to control an application. In [Holleis, Paasovaara, et al. 2008][2], we analysed various types of touch input controls on clothing and wearable accessories, e.g. to control a music player. The device in [Kranz, Holleis, and Schmidt 2005][2] is another example of a gadget that only allows a very restricted type of input, in this case just the distance between the body and the device and an additional large button. This makes sense when a highly specialised scenario constrains the set of interactions that are possible, e.g., when wearing thick gloves. On the other hand, research is conducted to extend the input capabilities of existing appliances. As an example, we augmented a standard mobile phone keypad with a capacitive touch sensor on each key [Holleis, Huhtala, and Häkkilä 2008][2]. This means that a third dimension is available on the phone keypad which enables concepts like 3D movement, tooltips, and additional shortcuts in menus.



[Kranz, Freund, et al. 2006] detailing gestural input methods: a sensor cube and visualisation

A taxonomy of how problems of distributed computing have become more complicated and accompanied by a range of additional problems (such as adaptive applications and location dependency) with the emerging area of mobile computing can be found in [Satyanarayanan 2001]. A similar effect appears when introducing issues of pervasive computing (e.g. invisibility and smart spaces) to mobile computing.

2.2 Application Development Process

Pervasive applications have many requirements in that they need to incorporate many different areas of computing: distributed software, hardware support, heterogeneity of platforms, operating systems, devices, etc. Surely, other software programs also need to deal with certain of these aspects, e.g., specialised hardware (grid computing, robot control), different platforms and devices (web browsers, virtual machines), distributed systems (data warehouse, stock exchange), or user-focused applications (data visualisation, end-user programming). However, a majority of pervasive applications rely on combining a large set of those areas. This is one reason why framework, toolkit and development support are crucial to push development for pervasive applications. Besides providing input to several aspects of pervasive computing, one of the major contributions of this thesis is support for enabling and simplifying the creation of such a diversity of applications, e.g. by providing tools such as the EIToolkit (see Chapter 4) and the MAKEIT programming environment (see Chapter 7).

An important aspect for creating pervasive applications is derived from the fact that the success of many of such programs relies on their affordance, i.e., simply put, their ability to express their purpose and functionality without additional explanation or documentation. Many programs are not targeted at experts in a certain area but on a (fictive) average person who has access to the required technology like for example a mobile phone with web access. This implies that the user should play an important role at the beginning as well as during the whole development process.

A typical procedure that follows such a user-centred design (UCD) process involves several steps. An ISO standard[10] describes how to apply user-centred design methods in the development life-cycle of interactive applications. However, no exact actions are specified, partly because the scope of the standard is very generic and aimed at a broad range of applications. In general, and based on the standard, a process following UCD principles includes, first of all, initial studies about related work, interested stakeholders and target groups, user expectations and profiles, application conditions and scenarios, as well as business cases and system requirements. Next, there will be the task of creating design solutions. This can involve several stages from coarse (low-fidelity) prototypes made from paper to polished applications (or high-fidelity prototypes) running on a simulated or even the target platform (see also Section 2.2.1). These products will then be evaluated and results will be fed into potential redesigns as shown in Figure 1. After the initial requirements have been met, a first working system can be developed. This can also consist of several stages beginning with a horizontal implementation, e.g. providing the whole menu structure of a user interface (UI) but without most of the functionality and then later filling in the necessary semantics. As before, each stage should be tightly accompanied and followed by an evaluation (see also Section 2.2.4). The results of the evaluation are subsequently used to refine the implementation. This is most often in the form of small bug fixes, patches, or small changes. Sometimes, however, some more fundamental flaws in the design might be discovered and it might lead to the need of a redesign. In most of these stages, simulation techniques can be employed. They can help in identifying problems with technology (e.g. scaling or power consumption), resource requirements and performance, and also usability. The last aspect can be covered, e.g., by simulating devices or functionality (such as using the Wizard of Oz technique where a human simulates automatic behaviour behind the scenes or the EIToolkit which can be used to generate arbitrary system input), situations and scenarios (e.g. by deploying it to virtual platforms such as Second Life), or simulating users with user models as described in Chapter 1.

This iterative design is supposed to produce a version that fulfils all initially set requirements and can then finally be deployed (see also Section 2.2.3). There are many similarities to standard software engineering processes but also some aspects where the development of pervasive applications departs. This includes involving several areas like hardware engineering and a large focus on UI design in novel areas. We refer to [Kranz 2008] for a more detailed comparison of these approaches.

Although the UCD process presented here is rather generic and can (and should) contain methods from software engineering such as performance and stress tests, the user-centred aspect is most important to keep in mind while creating (prototypical) versions and planning evaluations. A tight contact with prospective users or their representatives found through the initial analysis is critical. If this is too expensive in terms of money and time, a possible solution can be to compare the design against previously found user profiles and personas. Another possibility, pushed in this thesis, is to employ user models in order to facilitate this process. Although real customer studies are preferred in some cases, such models can be used to cheaply and quickly replace controlled lab studies. We will further detail this approach in the following Chapter 1.

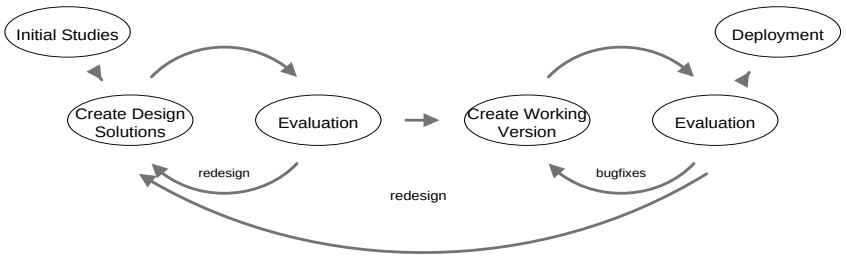


Figure 1: A common process based on which a user-centred design (UCD) development cycle can be applied. A critical part is to integrate the target users into each of the process parts.

In the following sections, we will briefly go into more detail about some of the steps in this development cycle and show where and how our approaches can improve the development of pervasive applications. We assume that the initial creative stages of finding ideas and looking for a business case, as well as the analysis of the state of the art, user expectations, application requirements, etc. have already been performed.

[10] ISO 13407:1999 Human-centred design processes for interactive systems; web page (access not free): *http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=21197*

2.2.1 Prototyping

An important aspect in the development cycle, especially when following a user-centred design process is to be able to see, try, and convey ideas. This is especially critical when people with different backgrounds are working together. Designers, for example, make use of a totally different vocabulary as computer programmers do. They also often think in different ways and roles. One possibility to alleviate this situation is to produce prototypes early and repetitively. These can be used to convey ideas to colleagues as well as to potential users, to get an idea of technical feasibility, and to get early feedback from users. Another potential advantage is that customers and users can be actively involved. Prototypes can also often be used in the target environment.

Prototypes are used to find out about aspects important for the success of an application or product. They can be used in different types of studies to get an idea of the affordance of a device, the practicality of interactions, the comprehensibility of tasks, and much more. This ranges from subjective opinions about non-functional prototypes to objective measurements such as task completion time and error rate for more advanced prototypes. They can also serve as physical instances for deciding between various designs and implementation possibilities.

Several types of prototypes exist, fitting to different stages and requirements. One criterion is fidelity. Low-fidelity prototypes remain sketchy and far from the product but are usually simple, quick, and cheap to produce. We favour high-fidelity prototypes since data gathered from studies can be more directly applied to the envisioned product. The mobile phone prototyping environment we propose in Chapter 7, for example, allows quickly generating applications that run on the target platform and thus convey the correct feeling of handling to users. Developers can still decide whether to produce vertical (completely implementing one or a few features) or horizontal prototypes (demonstrating the breadth of available functionality of a device or application but skipping the implementation of most of those).

Simple versions of a product can be created in software, e.g. using GUI-builders, rapid design builders like Adobe Flash, or toolkits such as those presented in Section 4.1.2, Software-focused Toolkits. Other types of prototypes include sketches, short videos, and paper prototyping.

2.2.1.1 Methods and Tools to Create (Tangible) Prototypes

One of the main contributions of this thesis is to provide a platform to easily generate prototypes of pervasive applications. However, there are a number of established ways to generate prototypes. Non-functional prototypes are often created using paper prototyping. With paper, a pair of scissors, some glue, and various pencils, one can create mock-ups of devices and user interfaces. Dynamic contents can then be simulated by exchanging the contents of a screen, replacing parts of it, or erasing, writing, or attaching additional text and graphics. The method is extremely quick, prototypes are easy to adjust, and for many applications it can convey a good enough impression of the desired product. It is mostly employed very early in the development process when the focus of the evaluation is on the idea and style of interaction but details such as look and feel can also be tested. It belongs to the category of throwaway prototyping since the prototypes do not help in implementing the system.

We used paper prototyping in several projects, for example to generate and manifest initial ideas of an Eclipse plug-in that helps developing pervasive applications. This is presented in Section 5.3.3. Another application was to find out very early about people's actions and reactions to using NFC interactions with a mobile phone. The results have been integrated into a framework for such interactions [Rukzio, Wetzstein, and Schmidt 2005] which we use in a mobile phone application development environment described in Chapter 7. This environment is also used to leverage such paper prototyping processes using images and storyboards in order to quickly generate working test applications. The results have also been proven valuable for the generation of user model parameters that allow us to apply interactional data (such as task completion time information) to prototypes without the need for real user studies (see the following chapter about user modelling). In addition, we used other prototyping techniques such as 3D printouts (e.g. [Kranz, Schmidt, et al. 2005]). This is still rather expensive. However, such printers are getting cheaper and it can already pay off for mid-sized groups to buy one of these. Often, a cheaper variant is to exploit and augment existing objects as we did in [Schmidt, Holleis, and Kranz 2004]. Although we do not currently provide own hardware, the tools we present in Chapters 4 to 7 especially ease the use of third-party hardware, in particular if they include some sort of 'smartness', such as toolkits like [Lee et al. 2004] and [Greenberg and Fitchett 2001].

Such toolkits can radically lower the threshold of developing applications for pervasive computing. Languages that base on a set of graphical UI widgets have helped to generate software user interfaces more quickly and ensured the adherence to certain standards and guidelines. Besides the two mentioned in the last paragraph, there exist a variety of hardware toolkits that try to achieve the same goals for physical interfaces. There has yet been no definite standard. However, those tools can tremendously accelerate development, especially for people without detailed technical knowledge. We present, compare, and evaluate several of them in Section 4.1, A Review of Existing Prototyping Toolkits. Of course, the software part is most often at least equally important and we treat software focused toolkits in the same section. The strength of our EIToolkit framework presented in Chapter 4 lies in its ability to abstract and combine software and hardware components and thus simplify the generation of pervasive applications.

2.2.1.2 Prototyping Issues

Obviously, prototypes play an important role for development, especially if a user-centred design process should be followed. It enables developers to present their ideas to colleagues as well as end-users in a more concrete way than would be possible by sketches alone. However, there are a couple of drawbacks that have to be taken into account. Some of those can be alleviated by using concepts presented in this thesis.

- **Prototypes can only test specific aspects**: some problems of deployed, real products like power consumption, bandwidth, cost, etc. are often not treated. Distributed, complex applications or very small or huge devices as well as those using hardware which is not yet available make it difficult to prototype.
- **High level of abstraction**: the level of abstraction required by users of low-fidelity prototypes can influence study results. Testers in general succeed differently in interpreting a rough prototype as a finished product.
- **Convey false impressions**: users can get false ideas with respect to the real state and performance of the project; e.g. a simulated speech interface will work better than most speech recognising software.
- **Applicability of results**: some results from the handling of a prototype cannot be directly transferred to the envisioned product. Emulating text recognition on a paper prototype feels and works differently than on a real touch screen with an automatic system.
- **Long way from prototype to product**: since many prototypes use communication through a central device (i.e. most often a PC is used to transform, adapt and forward communication between system components), these approaches are difficult to deploy, e.g. to another site or sell to the public.
- **Consequences of changes**: since prototypes can often be adapted quickly and easily, it may be difficult to see the implications on the existing system such as a back-end.
- **Prototyping tools target a specific audience**: prototyping tools and environments are often built to support a specific target group of developers. For example some replace low-level programming with scripting (but users nevertheless need programming abilities), some allow designers to concentrate on visuals (reducing the potential complexity of applications), or some use specific abstractions for young developers (which will often reduce efficiency).
- **Prototypes cost**: the effort in creating prototypes should not be underestimated. In contrast to evolutionary prototyping where a system is continuously refined, many of such rapidly built systems are throwaway prototypes and thus do not contribute directly to the progress of the product since they often neglect aspects like form factor, weight, etc. Difficulties arise as people have to use several different ways of developing, i.e. paper prototyping, hardware, and software, and with each of them have to use different types of design tools, programming languages etc..

However, only few of these issues imply that prototyping methods should not be employed. By allowing prototypes to be built directly on the target platform (e.g. on mobile phones [Holleis and Schmidt 2008], see Chapter 7), prototyping can achieve realistic levels. This can also simplify the transition from prototype to product. Complex algorithms such as image analysis can, for example, be simulated using the Wizard of Oz technique. Entirely novel devices or those that are hard to build can be simulated using augmented or virtual reality. By involving prototyping tightly in the development process, the impact of problems such as implementations diverging from the users' needs can be reduced and correct information about the state of the project can be communicated.

In general, tools can aid in keeping development costs down and also help in retrieving objective results. As we discuss in following chapters, a system can for instance be prototyped in software and – through the use of existing user models – metrics can be generated that are valid for the envisioned target (e.g. hardware) platform. Advantages of tools such as the MAKEIT environment (Chapter 7) are that prototypes can be generated on the fly, in fact even during meetings or user studies. To immediately counter issues and user feedback, it is possible to quickly change and alter prototypes. An important aspect is also that the threshold of creating prototypes is lowered. It is extremely valuable in a heterogeneous team that, for example, designers or even end-users can create simple prototypes to try things out, to store designs, and to explain ideas to colleagues. A combination of developing by simply demonstrating tasks and writing code as in Chapters 5 and 7 allows systems to be used across groups participating in the development. This combination also provides a low threshold and a high ceiling and has been one of the requirements of the EIToolkit suite introduced in Chapter 4. We also make use of standard tools such as drawing tablets, (scanned) images, and well-known development environments such as the Eclipse IDE to enable a broad range of developers to feel comfortable with their tools.

2.2.2 Implementation

One of the most difficult tasks in implementing a pervasive computing application is arguably to cope with the heterogeneity of the devices involved. Sometimes teams need to employ knowledge in microcontroller programming (assembler, C), mobile device (restricted C++ or Java), and platform independent software development including web-based services at the same time. They also have to tackle other issues of distributed and mobile systems like various data types, interrupted communications, and roaming users.

Even though most emphasis in this thesis is put on rapid prototyping and the quick generation of demonstrators, much of its work can also be applied to implementing final products. The combination of heterogeneous devices, the focus on reusability of components, and the support for existing applications, development environments, and toolkits (hardware as well as software), enables developers to make use of a broad basis on top of which interesting applications can be built.

It will not be an integral part to consider direct implementation specific issues. However, we will show several approaches to implement pervasive systems, focusing on rapid prototyping with graphical tools. We will also demonstrate the utility of different programming languages, the use of code generators and combinations of graphical tools with scripting or code writing.

2.2.3 Deployment

Deploying an application can happen in various levels and stages. The most general situation is that a system is made available for public access, will work independently and run unsupervised. Davies and Gellersen draw from experiences with the deployment of a variety of pervasive applications [Davies and Gellersen 2002]. They describe that the step from limited prototypes to polished products requires many aspects to be tackled. They give a list of technical, social and legal, as well as economic issues that makes deployment of pervasive applications difficult even if most standard problems concerned with distributed, ad-hoc systems such as name resolution, configuration or error management have already been taken care of. Several projects created and deployed in the past serve as illustration to collected challenges that must be treated with care to enable or facilitate the deployment of pervasive applications.

Although we do not go into much detail about deploying systems in this thesis, we adopted many of these challenges in the list of requirements that a development framework should comprise. Section 4.2, Toolkit Requirements for Pervasive Applications describes these prerequisites and also shows which of them are fulfilled by the current version of our EIToolkit.

It should also be noted that it is possible to deploy systems generated with similar tools to those presented here at least for collaborative and research purposes as has been successfully done with the iRoom and its components, [Borchers et al. 2002].

2.2.4 Evaluation

A major part of the development cycle, especially when following a user-centred design process is governed by evaluations. Most of the different types of prototypes imply specific types of evaluations. Additionally, various methods to evaluate a system have been introduced and proven valuable, depending on the specific stage in the development process, the desired results and the amount of time (and money) that is available.

Scholtz and Consolvo describe nine ubiquitous computing evaluation areas that they identified to be important for evaluating pervasive applications [Scholtz and Consolvo 2004]. These are abstract values for which an evaluator has to find ways of measuring. The authors argue that this process has to be project and implementation specific and cannot be generalised easily. It includes aspects concerning user attention, device invisibility, and application robustness. However, there is also a set of evaluation techniques that can be used to find out about many of those issues. In summary of the current situation in evaluating pervasive applications, there are still few results regarding the methodologies to use, what kinds of tools and standardised technology would be needed, and how to evaluate specific aspects like context awareness. [Neely et al. 2008] report on several workshops independently organised around the theme of evaluation requirements and systems in ubiquitous computing. They conclude that although there is strong interest in the subject, most methods used at the moment follow standard methods adopted from general HCI studies, which might not be the perfect solution. Still, much work has to be done to improve the situation. As one promising area, the combination of evaluations in the physical and the virtual world is mentioned. We are convinced that the models and integration strategies described in the following chapters can contribute to this research area in the sense stated in the paper.

Some types of evaluation methods, for example focus groups and brainstorming approaches like the Six Thinking Hats method [de Bono 1999], are applied very early in the development process where the focus is on finding ideas, creating concepts, and gathering requirements. Although many methods can be performed using early prototypes as well as final products, observational user studies, for instance, are best employed for functional applications close to finalisation. This ensures that people do not need to employ much imagination to fill in missing parts or to try to judge how they might evaluate a, say, different input method. The results will be more reliable if the user interface is as intended. Then, objective (time, errors) and subjective (pre- and post-study opinions) aspects can be measured. In between, i.e. for rough prototypes providing more or less functionality, expert and heuristic evaluations as well as cognitive walkthroughs are often chosen as a quick and inexpensive method. Domain experts are recruited to check the system, either by using it as an end-user would, by following certain tasks and trying to identify issues, or by comparing the system at hand against guidelines, usability principles, and other criteria.

Although we do not provide an entirely new evaluation system, we introduce methods based on user models to simulate user tests of a deployed product far before the system reaches a state mature enough to be evaluated by end-users. This is not meant as a complete substitute to real user tests, however. As we argue for example in [Schmidt, Terrenghi, and Holleis 2007], direct interaction with users in a participatory design process can bring much insight into users needs, especially if a prototype or product can be evaluated in a natural environment, e.g. the home of the users themselves. Thus, there are several **drawbacks** of using only models for evaluation, e.g.:

- **No direct observation**: observations and conversations with users that can sometimes lead to further insights are not available; on the other hand, this eliminates some subjectivity on the side of the evaluator
- **Learning effects**: it is difficult (but possible, see for example [Brown 1996]) to incorporate learning effects into models; however, it is not necessary to treat learning for first-use or routine tasks
- **Models simplify**: by definition, models might not capture all aspects of a system; however, they can indicate issues that require more attention
- **Modellers influence**: sometimes, aspects not thought of by the modeller do not appear in the evaluation, too; however, appropriate tools can help reducing this threat and even help end-users to become modellers themselves; additionally, the influence of a person designing a user study and questionnaires should also not be underestimated

On the **positive side**, several difficulties of evaluations with real end-users can be overcome by employing objective user models for evaluation, e.g.:

- **Easy specification of the target group**: for user studies, it may be difficult to concretely specify the target user group; models, in contrast, can be generic in some points and can often be quickly adapted for that
- **Gain of objectivity**: it is often hard to tell or find out why exactly an interface posed a problem for users; models can directly show where and what the problems were and often reveal their causes
- **Less cost of recruiting**: the cost of recruiting (enough) participants of the target group can be high; models can often be built quickly and reused in similar or other projects or project stages
- **Less cost in time**: an evaluation with users takes a considerable amount of time (usually about an hour per person in addition to the design and preparation of the study and the combination and interpretation of results); the time to apply a user models is negligible; the same holds for analysing gathered data
- **Less cost of changes**: changes such as in target group or interface / implementation are costly to incorporate, and some users simply do not like changes; most models can be easily adjusted in many aspects
- **Better control on learning effects**: learning can largely influence results: if the same person repeats a test, learning might have a higher impact than the changes in the system actually under observation, if another user is chosen, comparability is reduced; models behaviour does not change unintentionally and some learning effects can be incorporated

We will go into some more detail in the next chapters. In general, user models can be applied to incorporate simulated use of a system earlier, more tightly and cheaper in the development process. Different types of users can be generated in order to make tests with regard to specific target groups or system tolerance with the use of personas and user profiles. This can increase the speed of iterative development and reduce effort and cost for user studies. However, it would be optimistic to hope that an easy to use system with high affordance that directly targets the users' needs can be found without involving target users at all. Still, we can argue that a considerable set of issues can be found without the often subjective and expensive help of user studies which, alone or not done with appropriate rigour, also cannot guarantee a successful application.

3 User Models for UI Design

This chapter introduces the notion of user models and their application to human computer interaction. Specifically, we provide extensions to the Keystroke-Level Model (KLM) for physical mobile interactions.

After introducing the term and application of user models in the sense we use it (3.1), we delve into the area of cognitive user models and explain the concepts of specific models such as GOMS and KLM (3.2). This leads to a discussion on the properties and applicability of these models (3.3) before we detail an extension we developed for one of these models in the area of physical mobile interaction (3.4). We then present a summary of this work and point to additional extensions that can bring models like the KLM to further application areas (3.5).

3.1 User Models – Overview

In general, a model is a simplified version of an entity or method; simplified in the sense that certain details might be abstracted or ignored. Within the scope of the model, complex situations or processes can be illustrated and made easier to understand. It must be kept in mind that models normally imply a certain set of assumptions and setting, outside of which no guarantees can be made that the model still holds. When simulating complicated processes, or processes that are only partially understood, models are often used to simplify them. An example that we are going to treat in more detail here is how to model human cognitive and motor behaviour in order to simulate and make predictions on the interaction of users with pervasive systems.

It should be noted here that we mainly employ user models for predicting error rates and interaction times. In a broader sense, user models have been used, among others, to create dialogue systems [Wahlster and Kobsa 1989] or recommender and personalisation systems [Resnick and Varian 1997]. In contrast to personalisation, our goal is rather to abstract from individual differences (see, e.g., stereotypes in [Rich 1979]). We refer to the conference series on user modelling[11] and [Wahlster and Kobsa 1989] for a more detailed overview on the involved terms:

> "The aim of research in HCI is not necessarily to develop computer systems that construct a model of the user. Instead, knowledge about users' mental models should enable designers to build systems that can be more easily understood by users, or at least to predict learnability and performance."
> [Wahlster and Kobsa 1989]

[11] Conference on User Modeling, Adaptation, and Personalization; UMAP'09 web page: *http://umap09.fbk.eu*

Other examples specific to the domain of human computer interaction include assistive systems built in line with simulated user behaviour. Here, user models can improve context-sensitive help systems by making predictions on possible goals and subsequent desired steps of the users [Gong and Elkerton 1990]. User models can also replace users (acting as surrogate users). These models are built to include all information necessary to behave similar to real users and have been given the means to interact with a system. As such, different designs can be tested and behaviour studied or the models can be used as test engines for interfaces [St. Amant 2000]. Although most of the systems we describe here would allow using different types and different purposes of models, we concentrate on predictive models and use them throughout the prototyping and development process in order to predict task completion times on the target platforms.

We follow the understanding of S. MacKenzie and use the term 'descriptive models' for models that provide a basis for understanding, reflecting, and reasoning about certain facts and interactions [MacKenzie 2003]. They provide a conceptual framework that simplifies a, potentially real, system. We contrast that with the term 'predictive models' which is used to characterise models that estimate some metrics in advance, i.e. before an application or product has been implemented. This terminology is not uniformly used in the literature. For example, [Ritter and Young 2001] distinguish between descriptive and functional models. In their terminology, descriptive models incorporates both types of models described above while functional models focus on models that can be used as a replacement for users, i.e. to simulate how people would use an interface.

In the next section, we briefly describe the concept of descriptive models. They include, for instance, models that classify devices by a set of properties, e.g., the number of dimensions sensed. We briefly discuss two examples. The Three-state Model for graphical input devices gives a concise and useful characterisation of the way input is realised with graphical devices. As second example, Guiard's model of bimanual skill is outlined. These descriptive models can also be used to inspect an idea or a system and make statements about their probable characteristics. However, they are mostly used to reflect on a certain subject. We will then further focus on predictive models, which provide analytical metrics about certain characteristics of the modelled system. They are most often engineered with more mathematical rigour than descriptive models. We will briefly mention several examples, including such famous ones as Fitts' or the Hick-Hyman Law. Additionally, there are classes of such models that are based on state transitions or grammars. The focus on that chapter will be, however, on those models that make predictions about human performance, especially in terms of the time to completion of tasks. As such, the GOMS family of models and the Keystroke-Level Model will be treated in some detail.

The KLM will then be extended to be able to model advanced interactions with mobile phones such as gestures and visual markers. This will serve as a basis for the effort to combine such models with prototyping and development tools described in the next chapters.

3.1.1 Descriptive Models

Since we do not focus on descriptive models in this work, we only briefly touch on two examples in order to see differences to predictive models and their possible utility in designing applications.

The first example is Buxton's Three-state Model for graphical input devices [Buxton 1990]. As the name suggests, it is built around three states, namely *out-of-range*, *tracking*, and *dragging*, Figure 2. It can thus easily model such devices as the mouse which is normally in the tracking state and can be brought into dragging mode by pressing a button when the cursor is over an icon. A simple tablet input method, on the other hand, is normally in the out-of-range state and can be brought into tracking mode but needs additional means to bring it into the dragging state. A tablet with a stylus can exploit all three states. The model has been employed very often to characterise and evaluate new input techniques, e.g. Apple's TrackPoint touchpad, see [MacKenzie 2003]. However, although it has proven to manage combining several input technologies, it has weaknesses in detail and expressiveness. An interesting extension to the model is given in the ExperiScope project [Guimbretière, Dixon, and Hinckley 2007], which uses visualisations built on both, the Three-state Model and the Keystroke-Level Model. The latter is a predictive model that we will use extensively throughout the rest of this work and that will be introduced in the following section. They also extend Buxton's model with additional states to include novel technology such as multi-level buttons and pressure sensing techniques.

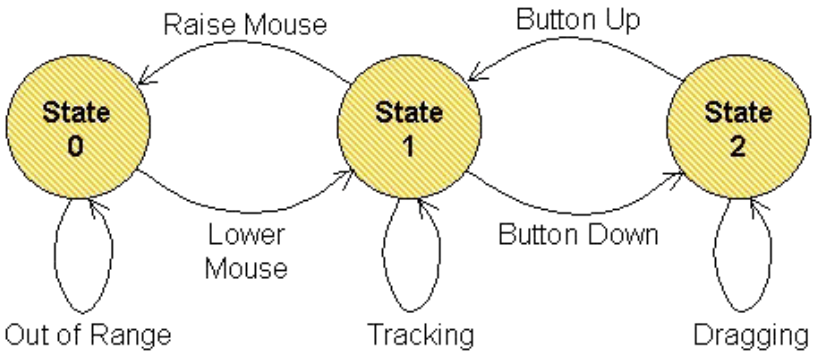


Figure 2: Three-state Model of Buxton [Buxton 1990]. Figure taken from [MacKenzie 2003].

The visualisation in the ExperiScope project also includes a separation between the dominant and non-dominant hand. Our second example, Guiard's Model of Bimanual Skill, elaborates exactly on this distinction between the different use of the preferred and the non-preferred hand in routine tasks [Guiard 1987]. Figure 3 illustrates and concretises the differences.

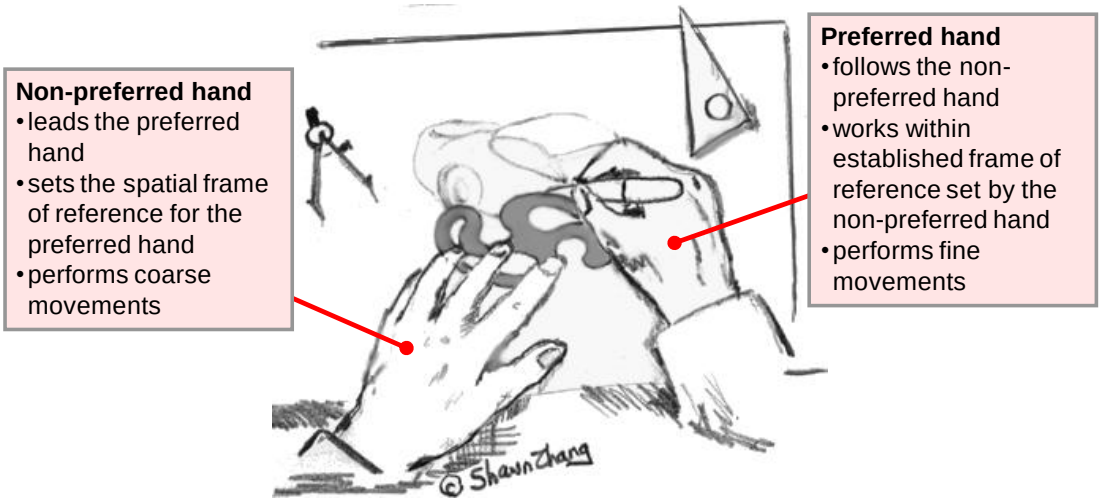


Figure 3: An illustration of Guiard's Model of Bimanual Skill[12].

There are manifold implications that can be drawn from that model for the design of user interfaces. As argued in [MacKenzie and Guiard 2001], the model clearly indicates that the task of scrolling should be done with the non-preferred hand. However, in most desktop settings this is implemented as to be done with keys (e.g. page up / down), the mouse (scrollbar), or the mouse wheel which are all actions normally initiated by the preferred hand. In applications such as treated in Chapter 7 about mobile device applications, the non-preferred hand should be used to hold the mobile device and perform tasks such as mode switching. Another application of the model suggests having the non-preferred hand guide the other one in tasks such as pointing in a direction or at an object or identifying rough directions for capturing visual tags.

3.1.2 Predictive Models

This section treats a set of models that allow making predictions about a system. Most of them are used to give some kind of performance estimate about the projected use of the modelled system. One of the most famous ones that we will only briefly introduce here incorporates results of studies done by Paul Fitts in 1954 and has later become known as Fitts' Law to acknowledge its importance and applicability. It gives timing information about pointing tasks and can be used to calculate the throughput of specific devices. We then briefly introduce more general approaches based on grammars and state graphs that we will reuse in work described later, before we focus on the area of cognitive models that will form the foundation of several extensions and applications provided in this thesis.

[12] Figure taken from "Models of Interaction – What are they?", slides for the course "Research in Advanced User Interfaces: Models, Methods, Measures", University of Tampere, Finland; course page: *http://www.cs.uta.fi/~scott/mmm/*

Fitts' Law

In 1949, C. E. Shannon provided an expression subsequently known as the Shannon-Hartley theorem that calculates an upper bound of the capacity *C* of a communication channel with given bandwidth *B* and signal-to-noise ratio *S/N* [Shannon 1949]. The formula for *C* is the first in Equation (1).

In his work, Fitts came up with an analogy for human targeting tasks involving arm movements [Fitts 1954]. The setup of the experiment backing the analogy required test persons to alternately tap on target areas of width *W* that were placed at distance *D* to each other. Fitts specified the formula to calculate the index of difficulty, *ID*, as a metric of the difficulty of a task as given in Equation (1) as ID_{Fitts} with the units of bits.

$$C = B\log_2\left(1+\frac{S}{N}\right);\quad ID_{Fitts} = \log_2\left(\frac{2D}{W}\right);\quad ID_{MacKenzie} = \log_2\left(1+\frac{D}{W}\right) \tag{1}$$

To improve the analogy to the Shannon-Hartley theorem above, a slightly different form of the calculation for *ID* is given in Equation (1) as $ID_{MacKenzie}$ in [MacKenzie 1989]. Besides being closer to the formulation on communication capacity, this has, according to research by the author, the advantage that the results correlate even closer to empirical values than Fitts' original version. Another positive effect is that the *ID* can no longer be negative. In ID_{Fitts}, the term *2D/W* can be smaller than one (and consequently the logarithm smaller than zero) for very close or large targets. Such a negative index of difficulty does not fit well to the model. The added '+1' in MacKenzie's equation ensures that, in those cases, the *ID* approaches zero instead.

Using $ID = ID_{MacKenzie}$, often referred to as the Shannon notation of Fitts' index of difficulty, the movement time *MT* necessary to hit a target at distance *D* and width *W* is expressed by Equation (2), where *MT* is linear in *ID*.

$$MT = a + b\log_2\left(1+\frac{D}{W}\right) = a + b\cdot ID \tag{2}$$

The parameters *a* and *b* are constants specific to the device used. The simplest method to find particular values for these parameters is to perform controlled tests with varying values for distance *D* and width *W* and to fit a line through the gathered data points, e.g. using linear regression.

Further detail on Fitts' analogy to information processing can be found in numerous documents. Its use and importance in human-computer interaction has probably been treated in most detail in [MacKenzie 1991]. In [MacKenzie 1992], the author also describes methods to correctly adjust the width according to the exact location of users' pointing actions and error rates. Fitts' Law is now incorporated as an ISO standard to evaluate pointing devices[13]. This shows that, although first conceived as a tool to measure and predict pointing times in one dimension, it holds remarkable well for higher dimensions and more complex pointing tasks. It still should not be taken as granted that the formula holds for arbitrary novel types of pointing interaction as some adjustments have to be made, see for example the treatment of a seemingly elementary task in two dimensions in [MacKenzie and Buxton 1992]. We will refer to Fitts' law again in Section 3.4.2, Model Parameters where text input on mobile phone keypads and pointing tasks using phones with built-in tag readers are studied.

Grammar based: Task Action Grammar

Grammars are often used to formalise the syntax of a language or a system. One of the advantages is that a grammar can concisely describe the rules along which all elements of a language are generated. For the purpose of describing interactive systems, the Task-Action-Grammar (TAG) has been defined in [Payne 1984] and [Payne and Green 1986]. It helps to formalise a mapping between the user's conceptual model of a system and the real application's behaviour. The TAG is a context-free grammar that formally describes how tasks can be solved taking sequences of appropriate actions. This helps in identifying the steps needed to complete a task and is especially useful in grouping tasks with similar functionality.

[13] ISO 9241-9:2000, Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs), part 9; web page (access not free): *http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=30030*

As an example consider the task to navigate in a document. We base it on the example given in [Green, Schiele, and Payne 1988] and update it to the behaviour in current word processors. We also show how different types models can work together by applying the descriptive Key-Action-Model developed as a simple example in [MacKenzie 2003] using the distinction of ‘symbol keys’, ‘executive keys’ and ‘modifier keys’.

```
task[unit, direction] → modifier_key[unit] + executive_key[direction]
modifier_key [unit = letter] → ""
modifier_key [unit = word] → "CTRL"
executive_key [direction = forward] → "cursor right"
executive_key [direction = back] → "cursor left"
```

Written in this way, one can clearly see the separation of choosing the granularity of the action and the direction. One could also have a slightly different set of rules:

```
task[unit, direction] → modifier_key [unit] + executive_key [direction]
modifier_key [unit = line] → ""
modifier_key [unit = paragraph] → "CTRL"
executive_key [direction = forward] → "cursor down"
executive_key [direction = back] → "cursor up"
```

However, it is not sensible to simply combine these two example rule sets as this would result in a not deterministic model (see, e.g., the ‘executive_key [direction = forward]’ rule). In this example, this can be easily solved by adding ‘down’ and ‘up’ to the possible values of ‘direction’. However, it is still often easier to add such functionality to the modelled system than to update the model.

By attaching time predictions to the terminal symbols, one can make predictions about the performance for the entire task. An example of the use of TAG as an evaluation tool to identify learnability problems in an interface without having to perform any user testing is given in [Brown 1996]. Already in 1988, Green et al. used the TAG in a study about command languages where they compared various formal modelling techniques and design guidelines in the context of learnability [Green, Schiele, and Payne 1988]. In their comparison to actual results, they found the predictions of the TAG most accurate. However, they also showed that there is the need for several extensions to the TAG technique whenever it is not only used to analyse the consistency of command languages. Together with the fact that the concept of grammars is not easily grasped by people without background in computer science or language processing, this approach seems less attractive from a prototyping point of view. Sometimes, however, behaviour is already defined using grammars, e.g. in Backus-Naur form anyway. In such cases, the application of TAG might be advantageous.

State transition based

One way to see a device or an application is as a system reacting to events by transitioning from one state into another. We use this approach in Chapter 7, which describes, among other things, an interface for developers to create a state transition based model of an application using a visual approach and a specific graph data structure. Statecharts have long been looked at to provide a graphical simplification of larger systems, see for example its introduction in [Harel 1987].

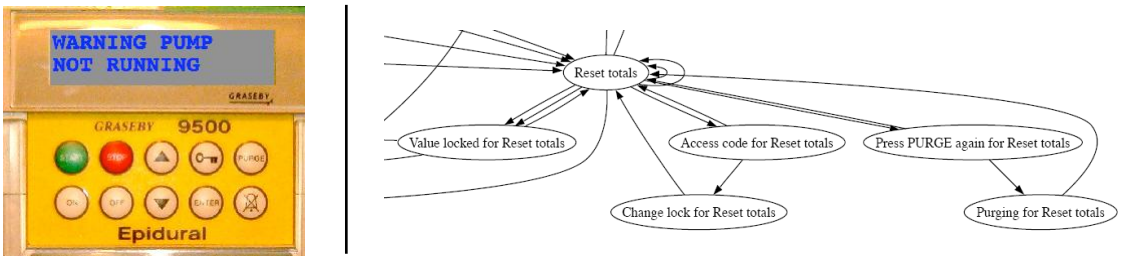


Figure 4: User interface (*left*) and excerpt of a state graph of a syringe pump (*right*) from [Thimbleby and Gow 2007]. The device which has a display and 10 buttons generates a graph with 54 states and 157 arcs.

A formal construct to treat these structures are for instance finite state machines (FSM). Several researchers argue that FSMs are not appropriate to model real interactive systems due to the simplifications they induce and the large number of states necessary to describe systems of even moderate size, e.g. [Palanque and Paternò 1998]. However, it often is exactly this power of simplifying complex behaviour that makes it valuable to understand and assess fundamental issues and properties. Thimbleby argues along similar lines in [Thimbleby, Cairns, and Jones 2001], although he targets a different implementation using Markov models which offers the advantage of being better scalable than FSMs.

We do not go into more detail about state transition based systems here but will revisit the approach several times, e.g. when treating the d.tools system that heavily bases on a statechart to create its application logic [Hartmann, Klemmer, et al. 2006] and postpone a detailed discussion to Chapter 7.

3.2 Cognitive User Models

In contrast to the approaches described in the last section, we now briefly mention models that concentrate less on describing the semantics of a device or application but target a formalisation of the way people think and act. These models can be summarised using the term *cognitive user models*. We briefly introduce the general concept and a few formalisms used to describe them before delving deeper into a specific family of such models that we are using throughout the rest of this work.

Model Human Processor
Most of these models are based on a certain type of understanding of how humans interact with each other and with systems. We introduce the Model Human Processor (MHP) since the models we describe afterwards draw heavily from its concepts.

The starting point of all these attempts to describe how humans interact is the human information processing model. In its extended version of [Barber 1988], it describes that any external input sequentially runs through four stages: encoding, comparison, response selection, and response execution. These stages are themselves all influenced by the two factors attention and memory. This means that all four stages are executed differently, with different speed, efficiency, priority, error proneness, etc. depending on whether or how much the person is focused on the task at hand and what and how much information is stored and accessed during each step of processing.

Memory plays an especially important role in information processing. Already in 1968, Atkinson and Shiffrin proposed three different types of memory: sensory, working (or short-term), and long-term memory [Atkinson and Shiffrin 1968]. Based on such approaches, the Model Human Processor was developed in [Card, Newell, and Moran 1983]. Its three interactive systems each consist of a processor and interact with different types of memory. The perceptual processor produces information that is stored in visual and auditory storage; the cognitive processor outputs into working memory and has access to both, working and long-term memory; and finally the motor processor coordinates actions.

Of course, this is a very coarse view on human abilities and it does not take into account any connections to the environment or collaborative actions. However, it proved to be very valuable for tasks where one person interacts with one system, e.g. a computer. Other models have also been developed that look into the relations of information processing in the presence of several people. Such a distributed cognitive model has, e.g., been presented by [Hutchins 1991]. We refer to the extensive literature provided at the ACT-R project page[14] for recent developments.

In order to make use of most of the models and architectures described in the following, tasks have to be defined that should be evaluated (benchmark tasks). There are several formalisms and modalities to describe such tasks. An overview of those can be found in [Paternò 2002].

[14] ACT-R cognitive architecture at Carnegie Mellon University; project page: *http://act-r.psy.cmu.edu/publications/index.php*

Cognitive Architectures

For the rest of this work, we will focus on two models, namely GOMS and the Keystroke-Level Model (KLM). These have the advantage of being abstract and simple enough to be employed and understood with only little introduction. They are models to formalise and describe human behaviour. Frameworks which implement such models of human cognition are called cognitive architectures. These try to simulate the human brain and possibly human methods to retrieve input and manipulate the environment. Although, ultimately, all those architectures base on similar assumptions, they vary in approach, detail, and in focus. [Ritter 2004] provides an introduction and many pointers to reviews and reports in that area.

We briefly mention three of the architectures that are in use nowadays. In order to keep the focus on our main approach followed in this chapter, we will not go into much detail. However, we want to acknowledge their existence since it might be interesting as future work to combine their power (which cannot be fully described here) with approaches given in this thesis.

The SOAR system (short for **s**tates, **o**perators, **a**nd **r**easoning) views all human processes as steps within the task of solving a problem [Laird, Newell, and Rosenbloom 1987][15]. As often done in logics processing, it builds on a knowledge base. A main goal (task) is split into sub goals that are solved independently and whose results are passed into the knowledge base for reuse. This is represented as a goal stack in a model of working memory. Production memory stores all the knowledge necessary to execute tasks. Deciding between several possible production rules in one state is modelled as a sub goal and based on preferences from other rules. A central concept in SOAR is to use all information available to choose and execute actions. Thus, it cannot properly model uncertainties, irrational decisions, or forgetting. Decisions are always optimal with respect to the knowledge available.

The ACT-R system (short for **a**daptive **c**ontrol of **t**hought, **r**ational) is also based on production rules and chooses optimal strategies [Anderson 1993]. However, this optimality is based on gain which means that it will choose the method that incurs the least cost while having the highest probability to achieve the current goal. This implies that, in contrast to SOAR, not all knowledge is necessarily used, e.g. if it would be too expensive to retrieve this knowledge, and thus can cope with inaccurate information. In ACT-R, cost is implemented using time and, similar to other parameters attached to rules, is adjusted through learning processes during the runtime of the model. The focus on time also renders ACT-R superior to SOAR with respect to time predictions.

The EPIC (short for **e**xecutive-**p**rocess **i**nteractive **c**ontrol) system can be viewed as to add ‘eyes’ and ‘hands’ to a cognitive architecture [Meyer and Kieras 1997][16]. It provides various processors in charge for input and output such as a visual and auditory processors as well as one for manual motor processing. This enables the system to directly interact with an implementation or a simulation of a user interface (provided it is implemented in a certain way to be able to interface with the architecture). Similar properties have been added to other architectures, see for example [Salvucci, Zuber, et al. 2005] for ACT-R in which extensions are described enabling the ACT-R system to observe and control a driving simulator. A further difference to the previously mentioned architectures is that there is no decision making aspect within the system. All processors (e.g. cognitive and motor control) can work and also all production rules can fire in parallel. Only if two steps use the same resource, a process to decide which of them takes precedence has to be provided by the modeller. This offer of parallelism provides flexibility but makes implementing constraints between different processes more complex. A drawback of EPIC is that the model does not offer means to incorporate learning although it is generally seen important to have a tight coupling between information processing and learning.

The following two sections go into some detail about two predictive models that also base on the Model Human Processor and are important to follow the reasoning in the next chapters of this work.

[15] This is the initial publication of the SOAR system; the architecture has been continuously refined, see the project page: *http://sitemaker.umich.edu/soar/home*

[16] This is the initial publication of the EPIC system; the architecture has been continuously refined, see the project page: *http://ai.eecs.umich.edu/people/kieras/epic.html*

3.2.1 GOMS: Goals, Operators, Methods, Selection Rules

In 1983, i.e. 25 years before this writing, Card, Moran, and Newell presented, among other things, a model to describe tasks and how users are proceeding to do these tasks in their book 'The Psychology of Human-Computer-Interaction' [Card, Newell, and Moran 1983]. Since the model is composed of four components, namely **g**oals, **o**perators, **me**thods, and **s**election rules, they named it GOMS. Despite its age, it is still one of the most widely cited and used models of its kind.

Analysing a task of a user who is interacting with a computer system, the first element to be perceived is the purpose of the task, i.e. what users want to achieve. All the models treated here assume that this target is clear, at least to the users themselves. In the GOMS language, this is called the *goal*. In all interactive systems, there exists a certain set of steps that can be performed, such as typing a command or selecting with the mouse. The GOMS model calls these single units of interaction *operators*. To achieve the aforementioned goal, a sequence of operators can be used. Often, there exist several such sequences, i.e. *methods*, to achieve the same goal. In this event, users can choose between different methods. Which method they use in a particular moment can depend on several factors like available or preferred input method, complexity of the task, etc. To adequately model such choices, GOMS uses *selection rules*. These specify the conditions under which a specific method is employed.

Goals: In most variants of GOMS, this is a rather verbal description of what a user wants to accomplish. Examples include 'move the cursor one word forward', 'copy a word / file / disk from place A to place B', 'open file F', 'order product P online', 'write a paper about X'. As can be seen, the complexity and abstractness of such a goal can vary greatly. It can be as simple and quick to solve as moving a cursor but can also be complex and involving many sub-goals as when writing a paper. It should be noted, however, that GOMS is best employed in that broad area in between those extremes. For too straightforward and quick tasks, it is often too much effort in contrast to the achieved gain. On the other side of the continuum, with very complex tasks involving much of human's creativity, problem solving, or communication skills, it can be difficult to model all the details. However, it is easily possible and often of great help to have a model of all the sub-tasks that need to be accomplished. In the example of writing a paper, a model might not be able to correctly describe all the aspects of extracting the important contents, summarizing the results, identifying parts that still need to be done, etc. It could be used, though, to create or analyse an interface built to support a user in performing this task, i.e. write a paper from scratch.

In written models, goals are preceded by the 'Goal:' keyword.

Operators: Most systems provide the user with a variety of different ways to interact with it. For the GOMS model, these are split into small units. Depending on the abstraction of the whole model, this can mean pressing a key or combining several steps as in 'enter address' or 'open file'. It is in the responsibility of the modeller to choose this level but it needs to be taken care that it remains consistent throughout the model; an operator should not be split somewhere else in the model.

In most cases, an operator describes direct user input by entering commands with a keyboard, selecting menu items with a mouse, or more advanced actions like performing gestures, using speech, or using tag based interaction. Other systems might support implicit interaction where the system draws from knowledge of the user and the environment. Actions not intentionally meant to be manipulating the system like entering a room can thus also be used as input. These play a superior role in the Keystroke-Level Model described further below.

In written models, operators are often given a name but otherwise simply listed one per line.

Methods: Different sequences of operators that lead to the same goal are subsumed into a method. It is important to capture such methods because different users might have different ways of doing something. Also, this construct serves to enable the comparison of different methods using the model.

In written models, each method is assigned a name or short description. A list of methods is enclosed by brackets and introduced by the 'select' keyword.

Selection Rules: Having different methods for one goal means that people have to choose between those methods. The reasons to select a specific path can be manifold. Some users might only know about one of these, others generally prefer mouse over keyboard, etc. In many cases, the choice will also depend on the specifics of the task, e.g. selecting a word might be done differently than selecting a whole page. The need to specify such rules not only makes the model applicable more generally, i.e. to more situations and types of users, it also helps the designer to think about possibilities and potential extensions, affordances, hints, or missing shortcuts.

In written models, selection rules are specified separately for each 'select' statement. Each rule is preceded by the 'Rule *n*:' keyword with *n* being the number of the rule.

As an example we use the goal to move one word (*n* characters) to another place (the end of the text) in the MS Word text editor. It was done for Office 2003 or older but most of it still holds for newer versions (with some changes in menu names). We assume that the cursor is at the beginning of the word to move. One approach is to separate different methods first and then specify each of them with appropriate sub-goals, see Figure 5.

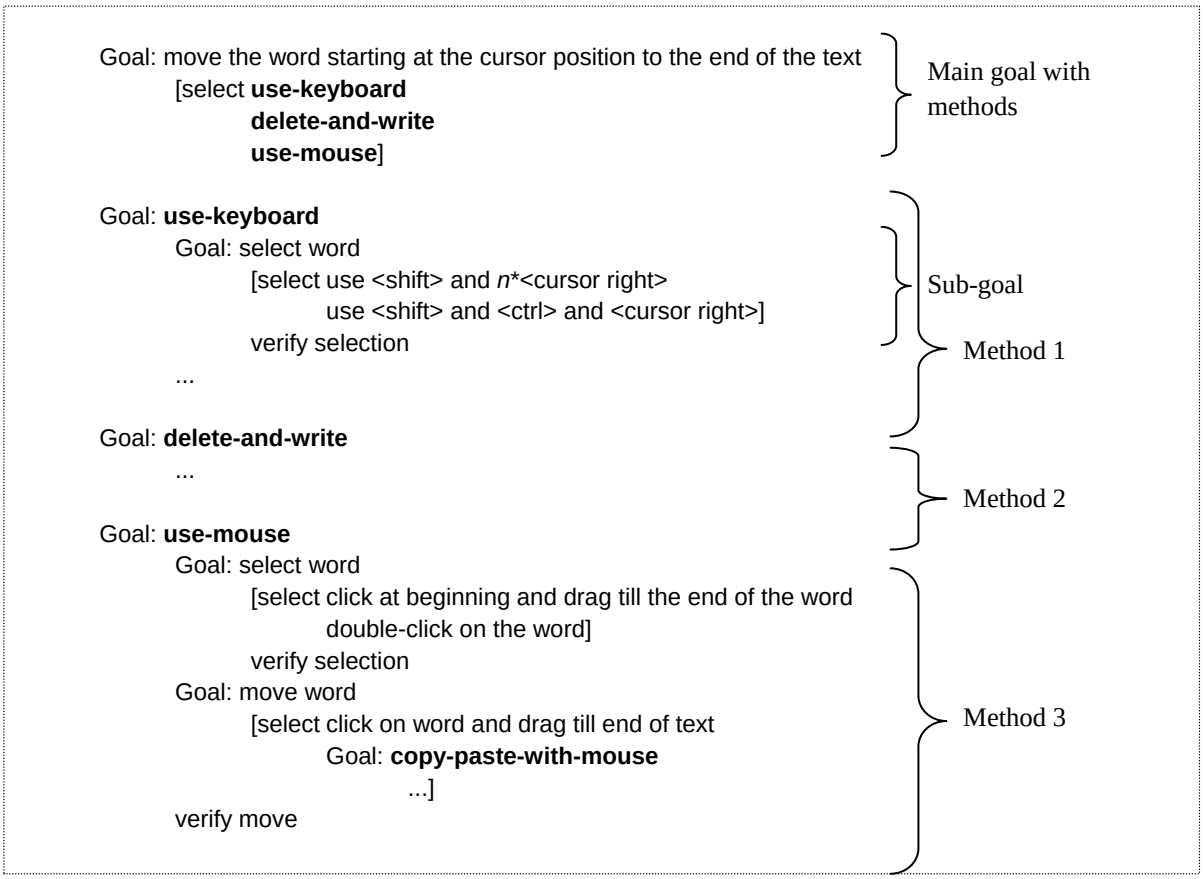


Figure 5: GOMS model of a moving a word in a text editor. Three different methods are described (only the use-keyboard method is fully described). Selection rules are not printed.

There are several more variants that differ in detail, e.g., a menu item can be activated with the mouse or with the keyboard or a combination of the two. Even without such distinctions, there are an impressive number of methods of how to achieve that simple goal. The reason why this has been implemented in such diverse ways can be found when trying to set up the selection rules. The main reasons why people choose a specific method are a personal preference between keyboard and mouse and whether people know and remember keyboard shortcuts or not. One possibility to satisfy both groups that prefer mouse or keyboard, respectively, as well as novices who more often use the (context) menu and advanced users who prefer the quick keyboard shortcuts is to offer all these methods in parallel. For systems that target expert users, the decision might be different. One important aspect to note here is that in building the model before, during or after the design of such a user interface, the designer gets a framework to think about, e.g., who are the users, what are prerequisites inherent to the system, what is the initial learning effort and what steps are necessary to help the user get more efficient.

Today, GOMS is considered to be a whole family of cognitive modelling approaches. Besides the Keystroke-Level Model described afterwards and the original GOMS, i.e. the one described in [Card, Newell, and Moran 1983], at least two more variants have been developed and are in active use, NGOMSL and CPM-GOMS. We do not employ those additional two in this work and therefore mostly point the reader to the corresponding, quite extensive existing literature. John and Kieras, for example, give a good overview and comparison of the four models in [John and Kieras 1994].

NGOMSL, short for **n**atural GOMS **l**anguage, uses a structured natural language notation, [Kieras 1988]. It bases on a cognitive architecture called cognitive complexity theory (CCT). Working memory plays an integral part in this architecture (see also the description of the Model Human Processor, above). In fact, working memory triggers rules specified in very much analogous form as in the original GOMS but using a specific format prescribed by the CCT. Besides manipulating the working memory itself, these rules can also trigger external operators, e.g. those on a keystroke level.

Two main differences to standard GOMS lie in the inclusion of access to working and long time memory and the different treatment of cognitive and perceptual operators. While the first adds a new way to evaluate an interface directly with respect to the load placed on memory, the last is more due to different underlying systems without considerably changing the result (the original GOMS mostly captures time for cognitive processes in a 'verify' operator, NGOMSL distributes this into every single step, and the CCT architecture incorporates one for a sequence of actions). One of the most interesting applications of NGOMSL is that it can be used to predict the time to learn a specific interface. This power is derived from the empirical result that the learning effort is roughly linear in the number of NGOMSL statements and long time memory chunks that need to be learned; see [Gong 1993]. By removing duplicate entries, knowledge transfer is simulated. The constants involved in this linearity have to be measured. If existing values are used, strict care has to be taken that the same methodology and style has been used in both, the empirical measurement of those values and the setup of the model at hand.

It should be stressed that this model cannot be used to predict learning or execution times for unknown operators. The method to predict learning time assumes that the user is already familiar with how each of the operators' actions is to be executed.

One of the limitations of those models – namely that all actions that lead to a goal are modelled to be executed in sequence without support of parallelism – has been sought to be overcome by a variation called CPM-GOMS [John 1990]. It uses the mechanism of the **c**ritical **p**ath **m**ethod (CPM) often employed to plan the timeline of projects. All single items necessary to reach a goal are defined and dependencies between them are specified, i.e. if an item *A* needs to be executed before another item *B*, then *B* depends on *A*. It explicitly allows parallel activities. These relationships can then be visualised in a PERT chart[17]. The longest path in this chart is called the critical path since it defines the minimum duration of the whole process. The disadvantage of this type of model is that, to be able to have cognitive, perceptual, and motor operators interleave each other, it must be very detailed and very low-level. In fact, modelling starts with an original GOMS model and refines its methods to the level of processing done by the cognitive, perceptual, and motor processors used in the MHP architecture. An implication is that, for tasks that do not offer strong evidence that several operations are running in parallel, the CPM-GOMS model might be too much of an effort.

A simplification that some models use to simulate parallelism is to set the times of some operators to be executed in parallel to zero. Available templates for some activities have been provided to simplify the creation of such models [John and Gray 1994]. However, the modeller has to keep in mind that the assumption of experienced users underlying all the GOMS models is taken to the extreme by this model as it models all processes to be executed as fast as the MHP allows. The assumed times to locate something on a screen and to verify the results of an action are slower than a person new to an interface would probably need. For many specialised applications where some tasks are repeated quite often, these assumptions are very likely to hold, however.

[17] PERT (program evaluation and review technique) Chart; a tool to visualise the flow of activities. A common application is to emphasise the critical path on which all activities can be found that would delay the whole process if any one of them were delayed; see for example the NetMBA documentation at *http://www.netmba.com/operations/project/pert/*

3.2.2 KLM: Keystroke-Level Model

In the following, the Keystroke-Level Model, KLM is introduced. In some sense, it is the simplest of the GOMS family. It has the advantage of being very quick to understand even by people without background in cognitive science. The reason is that the level of detail in which such a model has to be set up is clearly defined. On the one hand, it is on a level low enough for people to be able to easily follow and understand it; on the other hand, and in contrast to the CPM-GOMS, it abstracts from the details of the Model Human Processor. Nevertheless, its predictions have proven to be sound and valuable.

The KLM is especially useful when there are one or several clearly defined and describable sequences of actions, e.g. to achieve some task. It concentrates on the operators defined by the GOMS concept and does not treat different methods and thus also does not need any selection rules. One specific goal is modelled using a sequence of operators at the keystroke-level. That means that operators typically include small units of actions that have a direct impact on the interface. Common examples are mouse clicks, key presses, and hand movements. The time spent with cognitive and other processes not directly modelled in one of the operators, is subsumed into one generic 'mental' operator. It models such aspects as searching for the position of an item or verifying results. It is obvious that it can be sensible to use the original GOMS formulation to more abstractly analyse a system and then use the KLM to concretely describe a critical segment.

Three years before the publication of the book introducing GOMS, the Keystroke-Level Model was introduced [Card, Moran, and Newell 1980]. It has originally been used to analyse simple interactions with a computer-mouse-keyboard setting. Much work has been put into extending the use of KLM to a variety of other areas; an example of our own work in this area is described in Section 3.4 KLM Extensions for Advanced Mobile Phone Interactions and we list an extensive sample of other works below. The original formulation includes six operators with one more added later for button presses, as detailed in [Kieras 1993] (this paper also provides a good introduction to KLM), see Table 1.

Table 1: Original Keystroke-Level Model operators for the desktop setting.

Operator	Description	Associated Time
K	Keystroke, typing one letter, number, etc. or function key like 'CTRL', 'SHIFT'	Expert typist (90 wpm): 0.12 sec Average skilled typist (55 wpm): 0.20 sec Average non-secretarial typist (40 wpm): 0 .28 sec Worst typist (unfamiliar with keyboard): 1.2 sec
H	'Homing', moving the hand between mouse and keyboard	0.4 sec
B / BB	Pressing / clicking a mouse button	0.1 sec / 2*0.1 sec
P	Pointing with the mouse to a target	0.8 to 1.5 sec with an average of 1.1 sec Can also use Fitts' Law
D(n_D, l_D)	Drawing n_D straight line segments of length l_D	$0.9*n_D + 0.16*l_D$
M	Subsumed time for mental acts; sometimes used as 'look-at'	1.35 sec (1.2 sec according to [Olson and Olson 1995])
R(t) or ***W(t)***	System response (or 'work') time, time during which the user cannot act	Dependent on the system, to be determined on a system-by-system basis

The keystroke operator models typing one character. Often, a sequence of consecutive characters is either written in the form of '***K[hallo]***' or another operator name '***T(n)***' for typing *n* characters is introduced. The time is simply the number of characters multiplied by the time associated with one ***K*** since no difference is made between different keys. For specific keyboards or special characters, this might have to be revised. However, for short texts, an average value can usually be found.

There are only few aspects that have to be treated with care when applying the Keystroke-Level Model. Most of the operators are clearly self-explanatory and easy to apply. The Mental Act operator, ***M***, though, can pose some difficulties. Kieras concisely summarises the use of the ***M*** operator:

> "[The Mental Act operator] is based on the fact that when reasonably experienced users are engaged in routine operation of a computer, there are pauses in the stream of actions that are about a second long and that are associated with routine acts such as remembering a filename or finding something on the screen."
> [Kieras 1993]

The operator is not meant to model cognitively hard operations such as looking for a solution for a problem or making calculations. Some model generators break that operator down into a visual part such as a 'look-at' operator in order to describe searching a specific element on screen or validate an entered value and a separate 'think' operator. We do not need that level of detail and stick to one operator that subsumes both actions.

Despite its rather clear description, placing these ***M*** operators is often difficult when creating a KLM. We give a list of heuristics that help in deciding where to insert an ***M*** below. However, it should be remembered that, in most cases, the result of a KLM is a single time value. Thus, it does not matter where or in which order the operators are placed. It is much more important to think about the number of ***M***s in an action sequence. Simply put, a pause should be allowed before starting a new sub-sequence of actions such as typing a text, or whenever some remembered information such as a file or menu name needs to be accessed.

The second aspect that needs attention from modellers is the expert-user assumption underlying all GOMS approaches: the prediction holds best if the modelled users know what they are doing, i.e. they do not have to think or search for possible ways to solve a task. The models are best suited for routine tasks that are executed often. We provide several reasons why this assumption does not hamper the application of these models to manifold application areas below in Section 3.3. [Baskin and John 1998] compare KLM with CPM-GOMS in an experiment. They conclude that, with increasing experience (they observed one user repeating a task 500 times), CPM-GOMS can outperform KLM with respect to prediction precision. The main reason for this was identified as being the amount and duration of mental act operators placed in the KLM. With extensive practice, the user needed less time than predicted by these operators. However, this can be seen as an extreme case of routine. We do not go into detail about how to render this operator dynamic by including learning effects and proceed by giving some examples to illustrate the use of KLM. For text input, Isokoski and MacKenzie suggest to add a learning rate function to model the change of text entry speed from first sight to asymptotically expert use, [Isokoski and MacKenzie 2003].

A simple example of a KLM shows the difference between different implementations and input methods for a common task in the Microsoft Word text editor. As task, we chose to add some space after a paragraph. In versions before Word 2007, the necessary steps were to open the 'Format' menu entry, find the 'Paragraph' item and change a setting in the appearing dialogue. The corresponding KLM can be found in Table 2 for using mainly the keyboard (5.73 seconds) or the mouse (8.21 seconds), respectively. In Word 2007, a new ribbon replaces the menu and directly shows icons to access most functionality. It contains a shortcut icon offering a drop down menu with an entry for adding space to a paragraph. The accompanying KLM is found in Table 3 for the mouse (7.65 seconds) and in Table 4 for the keyboard (7.22 seconds). This chooses a standard amount of space to be added. Remarkably enough, most of these predictions are of a very similar magnitude indicating that there has been little improvement (for that particular task). For completeness, Table 5 shows the KLM that applies if a user happens to use the quite complex keyboard shortcut (2.47 seconds).

When comparing the models for the mouse interaction of the two Word versions, one can see that the steps of interaction are very much the same (except that, in one case, the size of the space has to be entered explicitly). The expert user assumption removes differences that might arise with novice users such as problems in identifying the correct icon. The model as given in the tables assumes that the distances and the sizes of the targets (i.e. menu headings and names, icons, etc.) are approximately the same. Sometimes, this has to be modelled more precisely, e.g. using Fitts' Law described earlier. Word 2007, for example, displays a context menu directly besides the current cursor position after selecting some portion of text, shown in Figure 6. A KLM using the standard 1.10 seconds for pointing would not be able to extract its advantage to the same icons placed in the menu bar. Only if the distances that the mouse has to be moved are taken into account the time saved by this approach can be correctly predicted.

Table 2: KLM to add a 6pt space after a paragraph in Microsoft Word *prior to version 2007* using keyboard shortcuts or the mouse to call a dialogue via the menu.

Description	Operator (keyboard)	Time allocated	Operator (mouse)	Time allocated
Locate menu 'Format'	***M***	1.35	***M***	1.35
Press ALT-o or mouse click	***K,K***	2*0.28	***P,B***	1.10+0.10
Locate entry 'Paragraph'	***M***	1.35	***M***	1.35
Press 'p' or mouse click	***K***	0.28	***P,B***	1.10+0.10
Locate item in dialogue	***M***	1.35	***M***	1.35
Point to item	***K,K***	0.28	***P,B***	1.10+0.10
Enter a 6 for a 6pt space	***K***	0.28	***K***	0.28
Close the dialogue (ENTER)	***K***	0.28	***K***	0.28
	Sum (keyboard):	**5.73 sec.**	**Sum (mouse):**	**8.21 sec.**

Table 3: KLM to add some space after a paragraph in Microsoft Word 2007 using the *mouse*. It is assumed that the 'Home' section in the ribbon is not active. Otherwise, it would be 2.55 seconds faster.

Description	Operator	Time
Locate ribbon heading 'Home	***M***	1.35
Click on heading	***P,B***	1.10+0.10
Locate icon 'Line Spacing'	***M***	1.35
Click on icon	***P,B***	1.10+0.10
Locate 'Add Space After Paragraph'	***M***	1.35
Click on entry	***P,B***	1.10+0.10
		Sum: 7.65 sec.

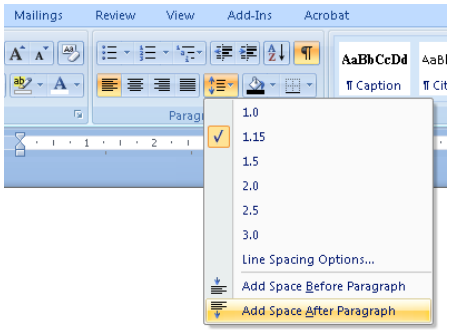


Table 4: KLM to add a space after a paragraph in Microsoft Word 2007 using *keyboard shortcuts*.

Description	Operator	Time
Prepare for shortcut	***M***	1.35
Press ALT	***K***	0.28
Locate key for 'Home'	***M***	1.35
Press 'h'	***K***	0.28
Wait for visualisation	***R(t)***	0.70
Locate key for 'Line Spacing'	***M***	1.35
Press 'k'	***K***	0.28
Locate key for 'Add space'	***M***	1.35
Press 'a'	***K***	0.28
		Sum: 7.22 sec.

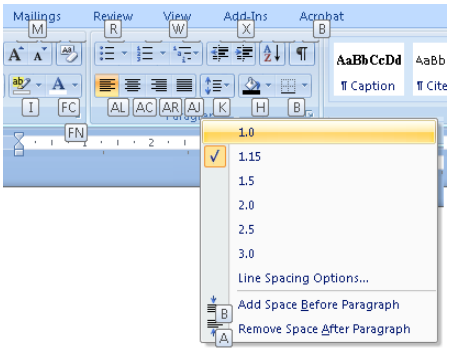


Table 5: KLM to add a space after a paragraph in Microsoft Word 2007 using a *known keyboard shortcut*.

Description	Operator	Time
Prepare for shortcut	*M*	1.35
Press ALT-h-k-a	4**K*	4*0.28
		Sum: 2.47 sec.

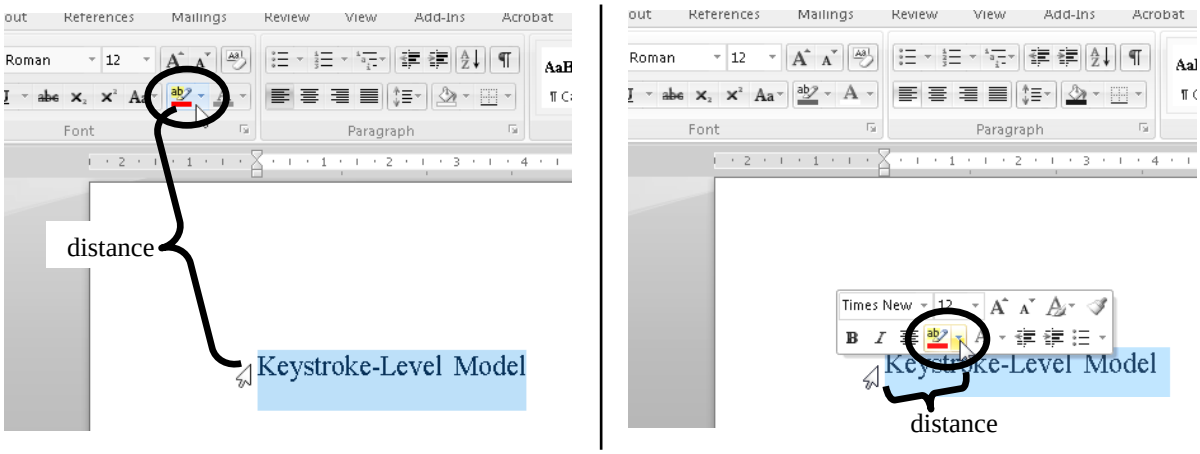


Figure 6: Mouse path for using the text highlighter in Word 2007.
***Left*: using the ribbon. *Right*: using the new context menu that appears close to the mouse cursor.**
The target size is the same but the distance is considerably different.

It is not difficult to create Keystroke-Level Models for existing applications if all interactions are known and operators have been defined for all of them. The definition of new operators or the adaptation of existing ones to new domains proves to be more difficult and time intensive. Section 3.4 below describes an example for such a setting. In general, the steps that need to be taken to apply the KLM to a given design are straightforward. We reproduce a list from [Kieras 1993] with some additions:

1. Choose one or more representative task scenarios.
2. Have the design specified to the point that keystroke-level actions can be listed for the specific task scenarios.
3. For each task scenario, figure out the best way to do the task, or the way that you assume users will do it.
4. List the keystroke-level actions and the corresponding physical operators involved in doing the task.
5. If necessary, include operators for when the user must wait for the system to respond, ***R(t)***.
6. Insert *mental operators*, ***M***, for when user has to stop and think. See the guidelines given in [Card, Moran, and Newell 1980] and more detailed heuristics in [Kieras 1993]
7. Look up the execution time for each operator. If these are parameterised or exist in different flavours (e.g. system response time or text input) then find the values appropriate to your application.
8. Add up the execution times for the operators.
9. The total of the operator times is the estimated time to complete the task.

Adapted from [Kieras 1993]

Many non-trivial examples of Keystroke-Level Models can be found in the literature. We will see another more complex example in Section 3.4.4 using the model extended for interactions with mobile phones.

3.3 Discussion and Applications of the GOMS Family of Models

The following list of pros and cons of modelling applications with GOMS models has been extracted from various sources including personal communications, publications (most prominently [John, Why GOMS? 1995] by Bonnie E. John and an included sidebar by Jacob Nielson) and through personal experiences with that subject. The different versions of GOMS are perhaps best treated in a comparison found in [John and Kieras 1994]. We concentrate on an analysis that tries to generalise over these particular differences. The analysis is also based on some of the issues and advantages mentioned in Section 2.2.4 for the application of user models in general. The following treatment is specialised to variants of GOMS and KLM.

Weaknesses with Respect to Application and Results

- **Just spending time is not modelled**: GOMS models build on sequences of actions that lead to a specific state or goal; merely killing time as identified by [Nielsen 2000] to be a potential 'killer' application is difficult to model; however, this is a distinct niche which is separated easily from routine tasks.
- **Difficult to target specific users**: a user study can sample exactly the target group whereas this is more difficult with models; however, models can often be easily specialised in several aspects. Also, GOMS assumes expert, near error-free interactions which are often not mirroring reality; however, there are approaches to counter this, e.g. using typing speeds of less experienced persons, or explicit modelling of novice users as shown in [Pavlovych and Stürzlinger 2004] for mobile text entry.
- **No real users**: the modeller has to predict the way(s) people will follow to accomplish a certain task; this might not be the way users will choose and if a specific action sequence is predetermined, this can potentially influence the results; however, the 'methods' parameter in a GOMS analysis can take care of several possibilities. Even if the sequence of actions taken has been correctly identified, for entirely new systems (i.e. without a direct predecessor), actual usage motivations and ways of use can diverge from the task analysis; however, if identified, these can be incorporated into the model. Also, comments from users in a study can be informative and reveal opinions, restraints, suggestions etc.; however, the more objective model analysis can often give more precise explanations of why an interface is inferior to another one. In the end, it is always sensible to have a user study with target end-users accompany tests with user models.
- **Difficult to model novel interactions**: as stated above, previously unknown operators or those not studied in depth are difficult to incorporate without time consuming pre-studies, see for example [Holleis, Otto, et al. 2007]; however, if done with rigour, it can subsequently be used without the need to re-run studies.
- **Various variable parameters**: some operations like system response times can be difficult to predict, especially in distributed systems (which also includes, e.g., simple web browsing); however, such parameters can often be ignored if they are fixed and occur the same number of times in different designs.
- **Users like to have impact**: it can have a positive advertisement effect when users see that the can have (or think that they have) a real impact on the development of an application; however, suggestions that are not taken into account and false impressions (e.g. a prototype looks finished but it takes long time till the release of the product) can have an even stronger negative impact.

Strengths with Respect to Learning, Results, and Efficiency

Good treatment of learning effects

- **Measurement of learnability**: GOMS/KLM can predict the time needed to learn to perform tasks.
- **Independence of sequences**: it can reduce bias due to the order in which users are presented with interfaces.
- **Measurement of knowledge requirements**: it can show how much prior knowledge users need to have; this can lead to a redesign to give novice users better guidance, e.g. repeat helpful information like an invoice address entered on a no longer visible page.

Good results

- **Gives reasons**: GOMS/KLM explains why something is judged to be slower or more difficult; user studies often only show *what* is negative.
- **Helps in decision making**: it provides cues for deciding between two or more different interfaces.
- **Identifies bottlenecks**: it can find components that should be re-implemented using a better (e.g. faster) method; we describe an example in Section 3.4.4.
- **Provides illustrative figures**: it can give predictions that can directly be converted into cost which make it well suitable for reporting to management.
- **Combines various views**: it can bring together requirements and descriptions from a variety of views like those from end-users, domain experts, designers, etc.
- **Treats feasibility and cognitive load**: it can indicate how much a user needs to remember between steps and can thus be used to predict the frequency of errors and also whether target users can use it.

Less cost in money and time

- **Quick to apply**: in Gong and Kieras' analysis, it took only 12 % of the whole project time to apply such an analysis, and only half of the time of executing user-surveys and a formal user-study, [Gong 1993].
- **Quick to prepare**: GOMS/KLM can predict long-term, skilled user performance; in contrast to many types of studies like think-aloud or cognitive walkthrough, there is no need to train people.
- **Helpful to design**: it can help developers to focus on understanding the problem and the application domain, [Gong and Kieras 1994].
- **Cheap to apply**: it alleviates some issues with user studies which are cumbersome, expensive, take a long time, and are subjective.
- **Easy to repeat**: it can be run any number of times, on a number of different tasks.
- **Quick to analyse**: its results can directly be extracted; retrieving information from, e.g., user studies, and transcribing it often takes considerably longer than gathering the data itself.
- **Precise to interpret**: it shows also small changes; sometimes even a difference in the range of seconds can be very important, especially if a task is done or repeated several thousands of times as in the case of telephone operator workstations [Gray, John, and Atwood 1992].
- **Easy to convey**: its results are easy to understand and quick to explain; they can be used to pass information to colleagues, managers and customers who do not necessarily have a background in cognitive modelling.

Discussion

A strong argument in favour of such models is that, in the last 25 years, a variety of projects have emerged that confirmed the validity of the GOMS model in various domains. Extensions and variants have been introduced to broaden its applicability into areas including some that had not even been known at the time of its introduction. The models can be compared to and developed along the assumed conceptual model of the users. This is important, among other things, to find discrepancies in the way developers and users think.

Sometimes it is seen as a drawback of such models that they assume nearly error-free expert user interaction. [Baskin and John 1998] describe the impact of the expert-assumption on models built with KLM and GOMS. In the end, KLM revealed remarkably precise prediction results in several projects (e.g. [Teo and John 2006]). Even in cases where experimental studies indicated that estimates were in fact considerably off the actual measured values, the estimated difference between two examined designs still proved to be a strong basis for making a choice between them (e.g. [Gong and Elkerton 1990], [Koester and Levine 1994], [Dunlop and Crossan 2000], and [Myung 2004]).

It should be stressed that time to completion of a task is only one aspect of a promising application. Still, it is an important criterion for a large set of applications ranging from small games to reservation systems, from sub-tasks to larger systems, from support to search systems. This is especially true for applications designed as side tasks or that exploit people's precious and short amount of spare time, e.g., between two tasks or while waiting for a meeting or the bus. These fundamentally rely on quick and hassle-free interactions. In addition to games and entertainment, mobile phones are increasingly used to enhance productivity and throughput in various fields like security or ticket sale. This is one of the platforms on which people do often not fully concentrate for a longer period of time. In general, experience has shown that it is essential to assess designs and applications early in the development phase. The phone company NYNEX probably saved millions of dollars [Gray, John, and Atwood 1992] because the Keystroke-Level Model was used to find out that the interaction performance of a newly designed workstation would have been slower to use than the existing system. As mentioned below, this also illustrates that saving a few seconds can make a huge difference as soon as a task is repeated a large number of times or by a large number of people.

The GOMS/KLM approach has been developed as a rather general method to describe and analyse a wide variety of applications. However, in the beginning, it was mainly targeted at small, static interactions of one user with a simple system. In the years after its introduction, it has been adopted by many researchers and application designers alike and has been extended in several dimensions. In the following, we briefly list a few areas in which it has successfully been applied in order to demonstrate the power and generality of the approach.

Application Areas

An important advantage of the user model analysis is that it can be applied to ideas and designs. There is no need to have the system implemented as long as the single interaction steps are known [Haunold and Kuhn 1994]. This tremendously helps in the design process because such tests can be run extremely early. An additional advantage is that, by creating the test environment (i.e. identifying important tasks, analysing the necessary steps, etc.), an improved understanding of the system under development can be achieved. Although distinctively more complex models would be necessary to directly support creative tasks, GOMS/KLM can also be applied in this area. It should be noted that the mental operator provided by the KLM is not suitable to model such creative actions. However, at least the routine parts of creative tasks, e.g. the use of dialogues or toolboxes can be evaluated even if the main task is mostly creative. In general, it is not necessary that a system is modelled on the whole. The models can also be applied to sub-systems or independent parts of a system. It is also easily possible to use different methods for different parts or to combine it with other usability methods.

A totally different area is that of surrogate users. The main aspect that renders this so different is that the task is mostly not directly set by the one who is using an interface but by, e.g. a customer. A telephone support person who tries to help a customer is such an example (see [Lawrence, Atwood, and Dews 1994] for an application of CPM-GOMS). The person is using a specific system to retrieve data, look up manuals etc. However, the goals that have to be achieved using that interface are set by the customer for whom the person acts as a surrogate user.

There has been some discussion to what types of systems these techniques can be employed. Historically, it has been used for rather static settings where, for example, a person creates and edits a manuscript in a text editor. However, it has been shown to work well for user-paced, passive systems from CAD systems [John and Kieras 1996] to flight-management computers in commercial airplanes [Irving, Polson, and Irving 1994]. It has also proven to be valid for single-user, active systems like radar monitoring [Santoro, Kieras, and Campbell 2000] or video games [John, Vera, and Newell 1993]. Recently, research also began to bring the analysis of coordinated work (see [Min et al. 1999]) to the realm of collaboration between several people with combined tasks [Kieras and Santoro 2004]. However, most advances break group tasks down into single tasks and define communication operators to coordinate them. This still does not seem to capture the full power of collaborative settings and leaves much work to do in this area. Another recent work to narrow this gap and extends KLM by analysing a collaborative game in detail is described in [Ferreira and Antunes 2006].

A further general area of application can be found in assistance systems. For example, context-sensitive and task-oriented help can be based upon such models, see e.g. [Pangoli and Paternó 1995]. Task-based documentation, both for developers and users, can profit largely from GOMS specifications.

As an additional, novel application area, we propose cross-platform evaluation as ongoing work in [Holleis, Kern, and Schmidt 2007]. It is easy to enhance any type of prototype like a paper or interactive HTML / Flash prototype to generate a KLM of a given task sequence. Our model estimates execution time of those tasks on, e.g., a mobile phone without the need to have a single line of code actually running on a phone. This can be generalised to various platforms.

There are possibly hundreds of specific application areas to which GOMS/KLM has been applied. Sometimes, extensions or variants have been specifically developed in order to take advantage of the full power of KLM and others, e.g. [Holleis, Otto, et al. 2007]. Only a small portion of those has been published to a larger audience. With the intention of giving a glimpse of the spectrum that these cover, we briefly indicate a few sample applications that can act as references for further research: menu selection [Lane et al. 1993], manual map digitising [Haunold and Kuhn 1994], email organisation [Bälter 2000], predictive text input to mobile phones [How and Kan 2005], in the driving context [Salvucci, Zuber, et al. 2005], [Pettitt, Burnett, and Karbassioun 2006], and specific interface designs [Hinckley et al. 2006].

A novel, recent extension and application of the KLM approach can be found in [Luo and Siewiorek 2007]. A KLM for mobile devices is created using storyboards [John and Salvucci 2005]. User actions are correlated with system actions. The latter are connected to energy consumption derived from interaction benchmarks on the target platform. This information is then enough to predict the time and energy consumption of defined tasks.

In the remainder of this chapter, we focus on time / performance predictions for the evolving domain of mobile phone interactions building upon KLM. This choice is motivated by the large number of publications in the human computer interaction community using KLM in a variety of emerging application domains. Many projects in cognitive modelling such as ACT-R rely on such data in ongoing research areas like in-vehicle interfaces. We adopt and define a set of operators giving sound and study-based estimates of performance measures for each of them. Developers of mobile applications, which possibly include wireless identification tags (RFID) and smart objects, can then describe tasks as a sequence of these operators and predict user interaction times without even needing to create prototypes. Table 7 shows an annotated excerpt of a model resulting from the new mobile phone KLM developed in this section (the full model is listed in Table 16).

3.4 KLM Extensions for Advanced Mobile Phone Interactions

Mobile phones have become a computing and communication platform that provides services going far beyond traditional phone calls and text messaging. Still, nearly the only aspect that has already received attention from researchers from a modelling point of view is text input. We first introduce the notion of physical mobile interactions and then provide a number of additions and changes to the known set of KLM operators in order to be able to cope with the new set of interactions. These are motivated, explained, and justified before a set of user studies is presented to attach time values to each of the operators. We then show a validation study where we designed a system, identified a task and modelled it with the extended KLM. The comparison with real user data indicates a very good fit of the predicted values. The remainder of this section is closely based on the paper [Holleis, Otto, et al. 2007].

3.4.1 Physical Mobile Interactions

Mobile phones can be used to manipulate virtual data, e.g. received messages, contact details, and information from the World Wide Web. Additionally, today's devices allow interactions with the real world through several sensors and systems. We first introduce such novel interaction methods before we go into more detail about modelling them in various novel extensions of the KLM.

Mobile Phone Interactions

Mobile phones have mainly been used to make phone calls, send text messages, and sometimes as calendar. However, other uses are becoming more and more popular. Taking pictures, surfing the web, storing data, and playing music as well as videos are some of them. Additionally, researchers started to use it as universal remote control (e.g., [Myers 2002]) and suggest further interactions with the world (see Figure 7). This adds several new interaction styles that have not yet been treated by any interaction model. In [Rukzio, Leichtenstern, et al. 2006], we define *physical mobile interactions* as being interactions between a user, a mobile device, and a smart object in the real world. The user interacts with the mobile device and the mobile device interacts with the smart object. This allows the implementation of systems envisioned, e.g., in [Kindberg et al. 2002], or [Rukzio, Paolucci, et al. 2006] and allows bridging the physical and virtual worlds using devices that many people carry with them, see also [Want, Fishkin, et al. 1999].

Although many people still think that mobile phone applications include mostly games and entertainment, phones are increasingly used to enhance productivity. In Japan, for example, it is common to buy tickets for public transport with the phone. Security personnel can use a mobile device that can read tags to quickly log the places they have checked.

Up to now there is no user performance model available for physical mobile interactions. We studied general preferences of people to use a specific physical mobile interaction method (touching, pointing and scanning) in [Rukzio, Leichtenstern, et al. 2006]. In this study, we also show that performance in terms of time is an issue for users. However, no quantitative performance numbers have been measured and only individual opinions of subjects are given. Ballagas et al. describe, explore and categorise a multitude of interaction types with mobile phones but do not give any timings or comparisons in that respect [Ballagas, Borchers, et al. 2006].

Figure 7: A sample of physical mobile phone interactions: using tags and taking pictures (for example of visual markers).

Interaction Types and Technology
Besides number entry and menu selection, there are several ways physical mobile interactions can be implemented. [Ailisto et al. 2003] and [Ballagas, Borchers, et al. 2006] give an overview of current technology for physical selection (visual patterns, electromagnetic and infrared methods) and compare them according to several characteristics like transfer rate and operating range. In [Välkkynen, Korhonen, et al. 2003] and [Välkkynen and Tuomisto 2005], as well as in [Rukzio, Leichtenstern, et al. 2006], projects are described using prototypical implementations of three basic physical selection techniques, *touching* (using RFID [Want, Fishkin, et al. 1999], Near Field Communication [Rukzio, Leichtenstern, et al. 2006], or proximity sensors [Välkkynen, Korhonen, et al. 2003]), *pointing* (visual codes like Semacodes and QR Codes [Rekimoto and Nagao 1995] and [Rohs and Gfeller 2004], laser pointer and light sensors [Välkkynen, Korhonen, et al. 2003], IrDA (Deutsche Post: Mobilepoint), or object recognition [Föckler et al. 2005]), and *scanning* (WLAN, GPS, Bluetooth [Rukzio, Leichtenstern, et al. 2006]).

Another interaction method we investigated is performing gestures. The underlying technology can be based on tracking the phone by an external camera, using the phone's built-in camera [Ballagas, Rohs, and Sheridan 2005], or reading sensors (found, e.g., in the Nokia N96 or the Samsung SGH-E760 and S4000 phones).

We keep the revised KLM as general as possible to be able model most of these types of actions and we give accurate estimates for some special cases.

3.4.2 Model Parameters

In this section we show differences and similarities between the KLM used for desktop interaction and the new KLM for mobile phone interactions. As presented before, the original KLM defines six operators and assigns time values to each of them: *Keystroke* (***K***, key and button presses), *Pointing* (***P***, mouse movements), *Drawing* (***D*(n_D,l_D)**, straight lines drawings with the mouse), *Homing* (***H***, hand movement between keyboard and mouse), *Mental Act* (***M***, pauses needed for reflection, choice, etc), and *System Response Time* (***R(t)***, user waits for the system). Kieras additionally lists *Button Press / Button Lift* (***B***, for pressing or releasing a mouse button, ***BB*** for a mouse click) as a standard operator [Kieras 1993].

Some operators have to be added to describe interactions that do not exist in the standard desktop metaphor. Others have to be examined closely to see whether the original timing specifications are still applicable or new values have to be derived. Others again are not applicable to the phone setting at all. The execution time of a task in the new model is then given by Equation (3) where $OP = \{\boldsymbol{A}, \boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}, \boldsymbol{I}, \boldsymbol{K}, \boldsymbol{M}, \boldsymbol{P}, \boldsymbol{R}, \boldsymbol{S_{Micro}}, \boldsymbol{S_{Macro}}\}$ is the set of available operators and n_{op}, d_{op}, D_{op} are the numbers of occurrences of an operator *op* in the model without distraction, with slight, and with strong distraction, respectively. Distractions are modelled with the ***X*** operator.

$$T_{execute} = \sum_{op \in OP} (n_{op} + d_{op} \cdot X_{slight} + D_{op} \cdot X_{strong}) \cdot op \qquad (3)$$

We first briefly describe which operators we adopted, which we needed to adapt for the new setting, and which we newly introduced. After that, we present several user studies in which we measured average timings for each of them. A table summarising the operators and the allocated times can be found in Table 6 on page 48.

3.4.2.1 New Operators

We added in total seven operators to the standard KLM. Three of them are concerned with how users shift their focus of attention and potentially get distracted due to the mobility of the device. Another three cope with novel types of interaction with the new platform and one needed to be introduced to model the initiation of interactions with a mobile phone.

Macro Attention Shift (S_{Macro})

One major difference from desktop interaction to using a phone is that the attention of users may have to be split between the phone and the real world surrounding them (Figure 8).

Figure 8: Attention shift (S_{Macro}) between the mobile phone and objects in the real world.

Thus, a *Macro Attention Shift* operator models the time needed to shift the focus between the contents on the screen of the mobile device to an object (e.g., a poster) in the real world and vice versa. The original KLM does not need to consider this case since it assumes that the whole interaction session takes place on one single screen.

Micro Attention Shift (S_{Micro})

S_{Micro} models the time needed to look from the display to the key regions and vice versa (Figure 9).

Although this can also happen in the desktop setting, this has not been mentioned in the original KLM. A possible explanation is the expert user assumption: users were not expected to need to look at the keyboard at all and therefore this time was incorporated into the *Keystroke* operator. This is different on mobile phones since the mapping of the keys is considerably more complex. Even experienced users tend to spend some time to confirm their input. Thus, the *Micro Attention Shift* operator allows a much more fine grained control over user interaction. It can also model uncertainty when, e.g., entering critical data like credit card numbers.

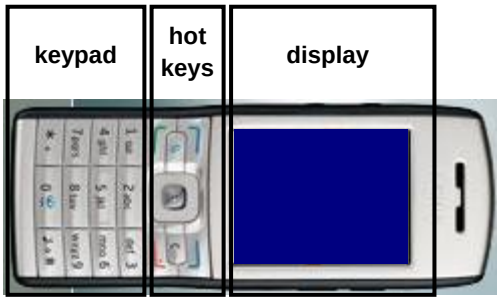


Figure 9: Regions of a standard mobile phone: keypad, hotkeys, and display. The S_{Micro} operator measures eye movements between those regions.

Distraction (X)

Since interactions with mobile phones take place in the real world, people are likely to be distracted from their main task by approaching people, passing cars, conversations, etc. This is accounted for by the *Distraction* operator. In contrast to all other operators, distraction is modelled as a multiplicative factor modifying the times of other operators. We distinguish between slight X_{slight} and strong X_{strong} distraction.

Action (*A(t)*)
This general operator models the time needed to execute a certain complex action with the phone that cannot sensibly be subdivided into smaller tasks and modelled with a combination of other operators. Possible actions include touching RFID tags, or focus to take a picture of a marker or other objects. The time for this operator highly depends on the type of action and, similar to the response time operator ***R(t)*** must be input to the model (indicated by the ***(t)*** notation).

Gesture (*G*)
This operator models the time needed when using a system that recognises gestures such as rotating or shaking the device, or drawing numbers in the air.

Finger Movement (*F*)
This operator models the time needed for a user to move a finger from one place (especially a key or button) to another one on the device. It will in most models be subsumed in the *Keystroke* operator but allows designers a more fine-grained modelling, e.g., to adjust for predicted repeated key presses.

Initial Act (*I*)
In the KLM for the desktop, it is generally assumed that users are already sitting in front of their keyboard, mouse and monitor, ready to initiate the next task. The phone introduces a completely different setting since people have to carry out some preparations (e.g., locating it in a bag) before being able to use it in most circumstances. The value depends on whether the interaction was initiated by the user or externally, e.g., by an incoming call.

3.4.2.2 Adapted Operators

We slightly changed three of the original operators to fit our needs. First, the keystroke operator needed to be adapted to the different keypad. Second, we generalised the mouse pointing operation to pointing with the mobile phone to a target in the real world (such as to an RFID tag). And third, we interpreted the homing operator as a specific movement with the phone switching from listening to reading or writing.

Keystroke or Button Press (*K*)
[Card, Moran, and Newell 1980] originally defined the *Keystroke* operator ***K*** to be the average time needed to push a button. It is to be measured by dividing the time needed for a longer sequence of button presses by the number of these presses. Even though KLM is targeted at expert users, immediate corrections of incorrectly pressed buttons, e.g. by hitting backspace, have explicitly been allowed and incorporated.

There are four factors influencing the value of ***K*** in our setting. Distances between buttons are much smaller on a phone than on a standard keyboard thus removing the need for head and larger eye movements and indicates a smaller value for ***K***. However, buttons are in general harder to spot and to press and people use only one or two fingers to type (in contrast to up to ten fingers for keyboard input). Finally, all but the most experienced users check and validate their input at some points needing some *Micro Attention Shift*s. The last three aspects suggest a higher value.

For text input, we concentrate on multi-tap which, based on figures presented from industry in a panel at MobileHCI 2006, is still used by about every second user. Predictive text entry methods are still not available in all languages and many names of people and places as well as colloquial terms still need to be entered with multi-tap. In addition, multi-tap proves useful for comparisons with previous research. Still, variants like T9 can be – and have been, see the references in the appropriate section below – easily modelled using KLM.

Pointing (*P*)
Pointing has originally been defined to model the time used to move a cursor to a target area using the mouse. This is in general not applicable for mobile phone applications except in rare applications in which a cursor can be controlled using the joystick or special buttons. Such interactions can be modelled using appropriate *Keystroke*s since they are not based on Fitts' Law as is the original interpretation of ***P***.

For larger screens or handheld devices using stylus input, we refer to [Luo and John 2005] who updated the values for *Pointing* for touch and stylus use. In our context, this models the time needed to move the phone from one place to another possibly to perform some *Action* at that point (which itself is not included in the pointing action and has to be represented by an ***A*** following the ***P***). This operation is similarly based on Fitts' Law as the original operator.

Homing (*H*)
In the original KLM, this modelled the movement of the hand from the keyboard to the mouse or back. For mobile phone interactions, this is not relevant. However, the action of moving the phone from a position where one can read the screen to one's ear or back is an analogous motion and similarly important. Therefore, we use the *Homing* operator whenever the user changes from listening and speaking to reading the screen or vice versa. In this setting ***H*** can be expected to be somewhat smaller but close to *Pointing* ***P***.

3.4.2.3 Unchanged Operators

The two operators for mental breaks and system response time could be adopted without changes.

Mental Act (M)
This operator can be adopted as defined and existing usage guidelines like from [Kieras 1993] can be used. There is no reason why a new setting should change the value of this parameter. It has been derived as a general value for short pauses to think about the next action, to find something on screen, or to validate an input. Since we use new operators, we give additional guidelines in a later section on the *Mental Act* operator.

Most studies adopted the original choice of ***M*** = 1.35 seconds for their applications (e.g., [Dunlop and Crossan 2000], [Haunold and Kuhn 1994], or [James and Reischel 2001]). [Mori, Matsunobe, and Yamaoka 2003] and [Myung 2004] propose a smaller value of 0.38 and 0.57 seconds, respectively. However, these values were taken from much specialised applications. The latter one, for example, examined Korean text input only. For general settings, a higher average value can be assumed. Larger values than the original value are reported in [Manes, Green, and Hunter 1996] which mainly result from studying an explicit scenario (a car navigation system) and users who do not have the routine of expert users assumed by KLM. Current cognitive architectures like ACT-R confirm the original value. We also found no evidence to justify a change.

Response Time (*R(t)* or *W(t)*)
The *Response Time* operator models the time the system needs to react to user input as long as it blocks the user from executing further actions. It can be adopted as defined and must be input to the model (thus the ***(t)*** notation) since it is highly variable and dependent on the application.

3.4.2.4 Not Applicable Operators

Two operations tightly connected to the use of a computer mouse are not used in the mobile phone setting.

Mouse Button Press (*B*)
The *Button* operator ***B*** models the action of pressing or releasing a mouse button. It has no direct equivalent in the world of mobile devices.

Drawing ($D(n_D,l_D)$)
The *Drawing* operator ***D*** models manual drawings of n_D straight line segments with a total length of l_D (measured in cm) with a mouse. This is not applicable in our setting with mobile phones. However, this might find an application with possible future additions to advanced mobile device interactions. In some settings where the device is used, for example, to perform some drag and drop operations, this might be a fitting way of modelling such applications.

3.4.3 User Studies for Time Measurements

To be able to use the model in practice and to predict the time required for certain complex tasks based on the model, the duration of a single application of an operator must be known. In total, we performed seven studies to acquire the data to estimate execution times. We recruited volunteers of various backgrounds (about 50 % students) on a study by study basis (between nine and 19 participants per study). We did not see any differences caused by gender with, altogether, 41 % female participants. If not explicitly stated otherwise, we present the median of the measured values and list 1st and 3rd quartiles in Table 6.

Before conducting each of the studies, questionnaires were given to the users to clarify their experience with mobile phones in general and more specifically with the mobile phone interaction technique under observation. We also aimed to adhere to the expert user assumption by running one or several training sessions with each user. Participants had to repeat the same or similar tasks until they (and we) were confident that they would make only minimal errors. Erroneous trials were discarded. With the exception of one study that used a stationary eye tracker, all studies were executed in various, every day, non-laboratory situations.

3.4.3.1 Special Operators

Initial Act (*I*), Homing (*H*)

Without preparation or creating a specific situation, we observed people in everyday settings, receiving and answering phone calls, and specifically asked them to pull out their mobile phone and execute a phone call. We videotaped all these actions and extracted timing information from 11 people, aged 25-54 with an average of 34.6 years, 4 female, all used to standard mobile phone interactions, see Figure 10.

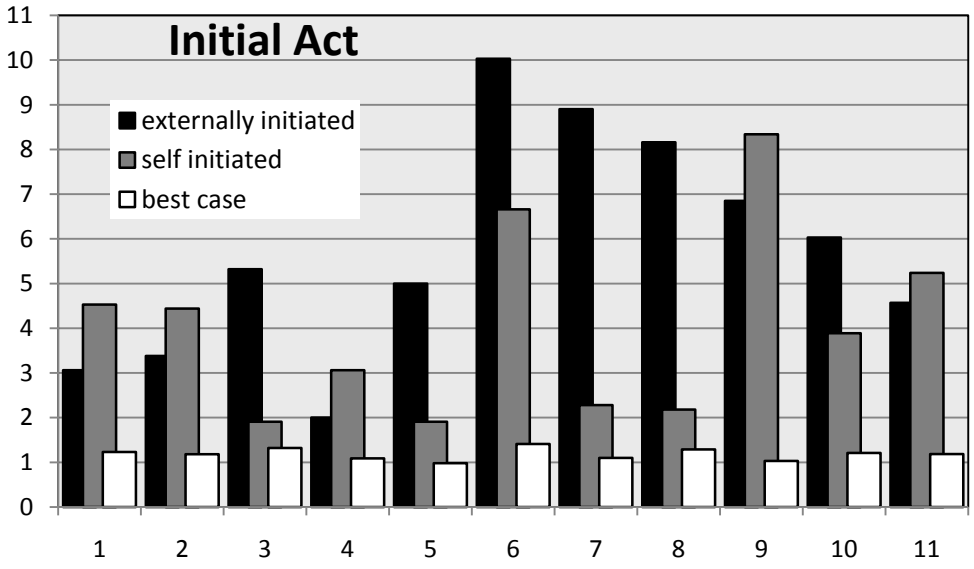


Figure 10: Average time in seconds of 11 persons to prepare for a phone interaction. We distinguish between externally and self initiated situations as well as a 'best case'.

We found that the value for the *Initial Act* operator strongly depends on whether the interaction was initiated by the users themselves (leading to a median value of ***I*** = 3.89 seconds) or externally, e.g., by an incoming call (median ***I*** = 5.32 seconds). Those values are highly diverse, however, since people have very different ways to store the phone (for example in a trouser pocket, a handbag, or in a pocket attached to the belt). A best-case study in which the phone was placed on a table in front of the users who initiate the action themselves or expect a call gives a median of ***I*** = 1.18 seconds. Thus, if no assumptions can be made, we suggest an average value of ***I*** = 4.61 seconds. For repeated or expected interaction the ***I*** = 1.18 seconds estimate should be used.

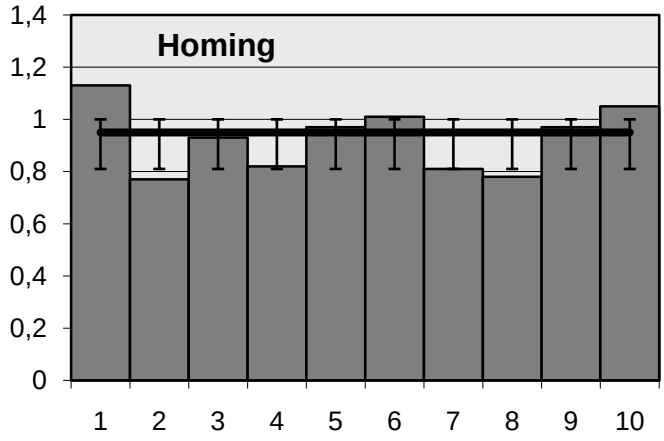


Figure 11: Average time in seconds of 10 persons to change between a position with the phone at the ear and one looking at the screen and vice versa. The median is 0.95 seconds; bars indicate range between 1st and 3rd quartiles.

In the same study we measured times needed to switch from a phone position where the screen can be read to one close to the ear and back, see Figure 11. As this is very similar to changing from keyboard to mouse and vice versa, we use the *Homing* operator to describe this action. The times of all people under observation were very similar and we extracted a median of ***H*** = 0.95 seconds. As expected, this is only slightly smaller than the found value of *Pointing* ***P*** = 1.00 second described below. To model the fact that people have to refocus on the phone's screen and continue their interrupted action, we strongly suggest that a *Mental Act* operator be placed after a *Homing* away from the ear as specified in the heuristics given in the section about the *Mental Act* operator.

Pointing (*P*), Action (*A(t)*)

To measure execution times for *Pointing* and *Action*, we needed an application where such interactions occur quite often. In some countries like Japan, visual markers and near field communication (NFC) are already very wide-spread technologies in the public see, e.g., [Boyd 2005], and [Fowler 2005]. This is not yet the case in Europe. Therefore, in conjunction with other projects run in our lab (e.g., [Rukzio, Schmidt, and Hussmann 2004]), we prepared a movie poster acting as user interface for several interaction methods. Users can select and use different services by, e.g., touching NFC tags or taking pictures of visual markers. We asked users to follow the brief instructions on the poster and let them buy tickets for their favourite movies in a theatre close to them. From the videotaped footage we were able to extract timing measurements regarding the movement (***P***) and alignment (***A(t)***) of the phone to the NFC tag, and the approach (***P***) and focus (***A(t)***) of the phone to take a picture of a marker.

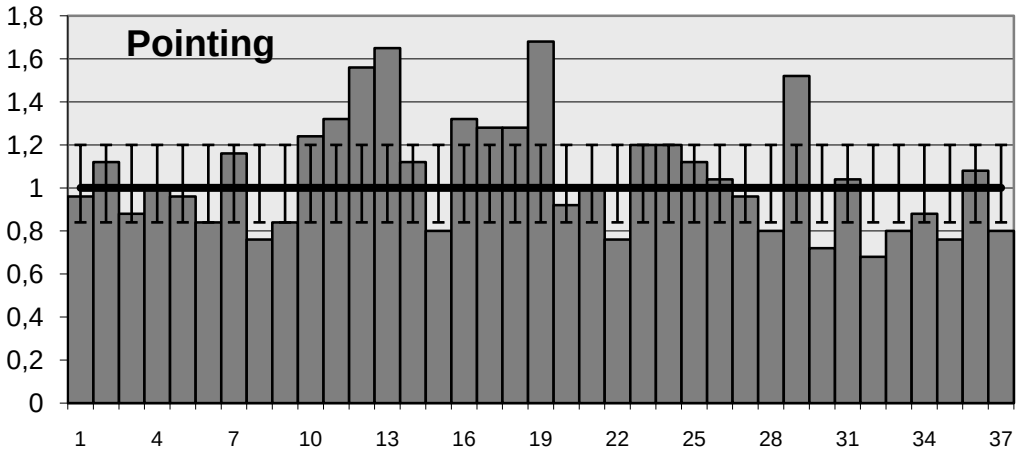


Figure 12: Average time in seconds for 37 pointing operations with NFC tags. The median is 1.00 second; bars indicate range between 1st and 3rd quartiles.

The user study was carried out with 9 persons, aged 22-46, with an average of 28.6 years, 2 female. From a set of 64 error free videotaped actions, we deduced *Pointing* $\boldsymbol{P}$ = 1.00 second. The 37 NFC interactions (Figure 12) showed that aiming at the NFC tag itself did not need any separate action besides the phone movement and we define $\boldsymbol{A_{NFC}}$ = 0. The remaining 27 photographs of visual markers led to a value of $\boldsymbol{A_{picture}}$ = 1.23 seconds for correct positioning and focusing (note that this does not include *Pointing*).

Note also that the time needed by the system to recognise the tag or interpret the marker and initiate the appropriate action is not included in the *Pointing* or *Action* operator but must be modelled by with the *Response Time* operator ***R(t)***.

Macro Attention Shift (S_{Macro})

Using a careful frame-by-frame manual analysis of the video tapes from the study presented in the last section, we counted the number and determined the duration of head and eye movements that indicate an attention switch from the phone to the poster and vice versa. We extracted a total of 121 attention shifts, see Figure 13. The times of the shifts in one direction do not differ significantly at all from those in the other direction ($t = .57$, $p > .56$). Thus, we propose a common value of $\boldsymbol{S_{Macro}}$ = 0.36 seconds.

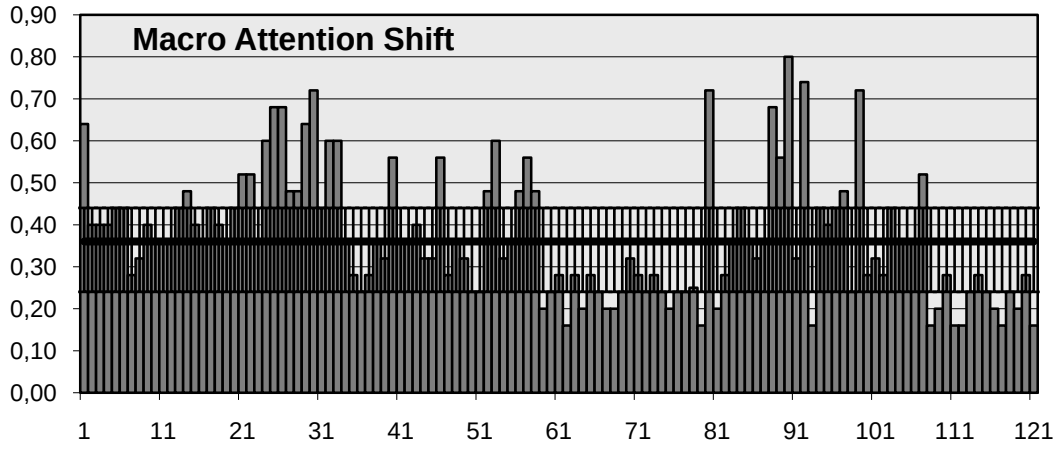


Figure 13: Average time in seconds for a change in focus between phone and real world. The median of 121 observations is 0.37 seconds; bars indicate range between 1st and 3rd quartiles.

Micro Attention Shift (S_{Micro})

Most current mobile phones suggest a separation into three regions: display, hot keys, and keypad (Figure 9). Finding out when and to which section people looked proved to be infeasible with conventional videotaping. Therefore we used an eye gaze tracker from Eye Response Technologies that samples images with a sufficient rate of 60Hz. The participants had to run three pre-set tasks that included writing a short text message (mainly text input), changing the ring-tone (mainly menu navigation), and setting the time of the alarm-clock (menu navigation and number input).

Figure 14: Eye gaze positions during a task, overlaid over the phone in use. *Left*: write a text message. *Right*: set alarm time.

All 10 people (aged 24-34 with an average of 27.5 years, 6 female) were allowed to use their own mobile phone and we ran several sessions to ensure error free interaction. We then automatically calculated the number and time of gaze position changes between the regions from the logged data. Figure 14 shows data overlaid on some phones. We counted more than 1500 shifts between the three regions and found the following median values: display ↔ hotkeys 0.12 seconds, display ↔ keypad 0.14 seconds, and keypad ↔ hotkeys 0.04 seconds. If no distinction should or can be made between the single sections of the phone, we suggest using the median of all values of S_{Micro} = 0.14 seconds. In Figure 15, one can see that the eye tracker does not deliver continuous values. Also, some values are below its accuracy and have not been taken into account for calculating the final values.

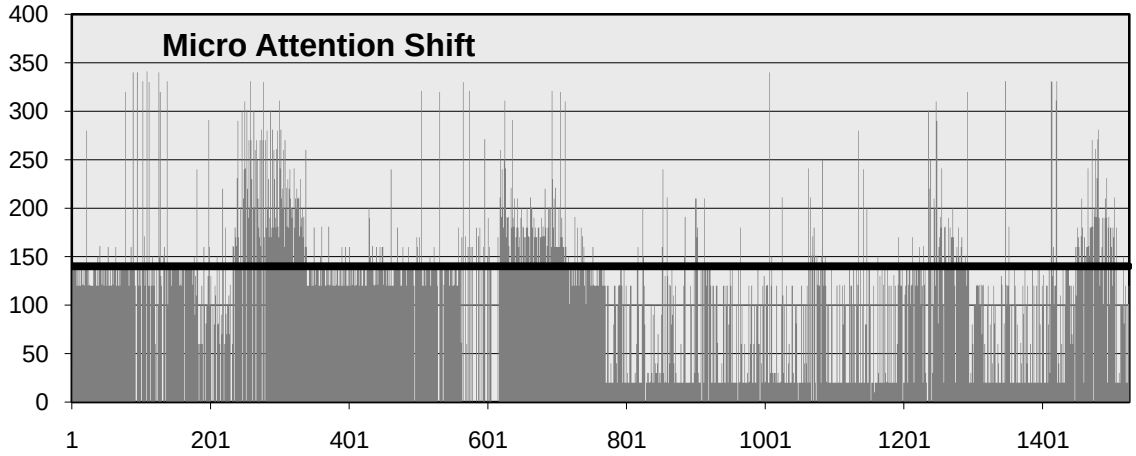


Figure 15: Average time in milliseconds for a change in focus between different regions on a phone. The median of 1509 observations is 0.14 seconds.

Gesture (*G*)

To measure gesture input, we used a Samsung SGH-E760 phone with built-in acceleration sensors and a few games and standard applications that can be controlled using simple gestures (Figure 16). The times are consistent across different gestures. However, if new gestures like those introduced with the iPhone should be modelled, it some study to ensure applicability would be advised.

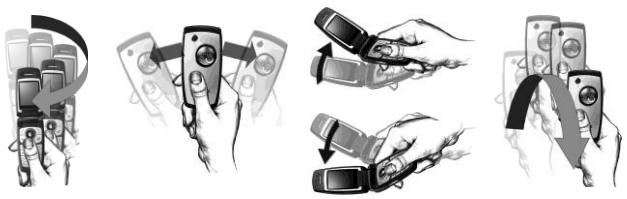


Figure 16: The Samsung SGH-E760 phone and some of its recognised gestures.

The times were extracted from videos of 6 different types of gestures, each done by 10 people (aged 23-33 with an average of 26.3 years, 5 female), see Figure 17 and we fixed an average value of G = 0.80 seconds.

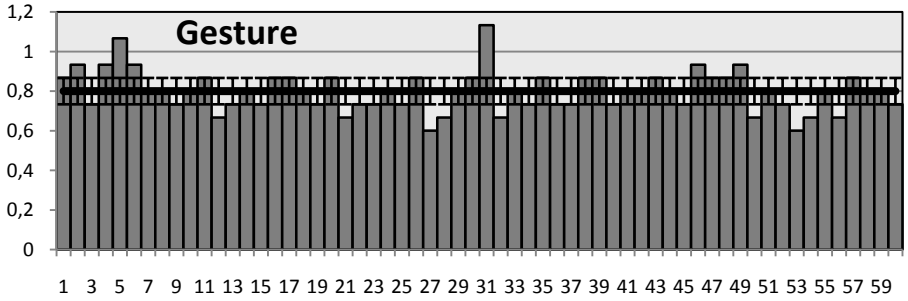


Figure 17: Average time in seconds for one of six gestures with a mobile phone by 10 persons. The median is 0.80 seconds; bars indicate the range between 1st and 3rd quartiles.

Keystroke (*K*)

Keystrokes were measured with a small program that logs timestamps of pressing and releasing keys into a file on the mobile phone (using the File API on a Nokia N90). We invited 19 people, aged 25-40 with an average of 27.8 years, 9 female. Each person entered two mobile phone numbers of their own choice. All of them used the wide-spread one-hand thumb entry method. During the study we observed that no errors were made.

The Nokia N90 phone features a standard 12 button keypad. The median of all interaction times is 4.63 seconds and the time per keystroke was calculated to be ***K*** = 0.39 seconds, see Figure 18. For the five most experienced users we got a value of ***K*** = 0.33 seconds per keystroke.

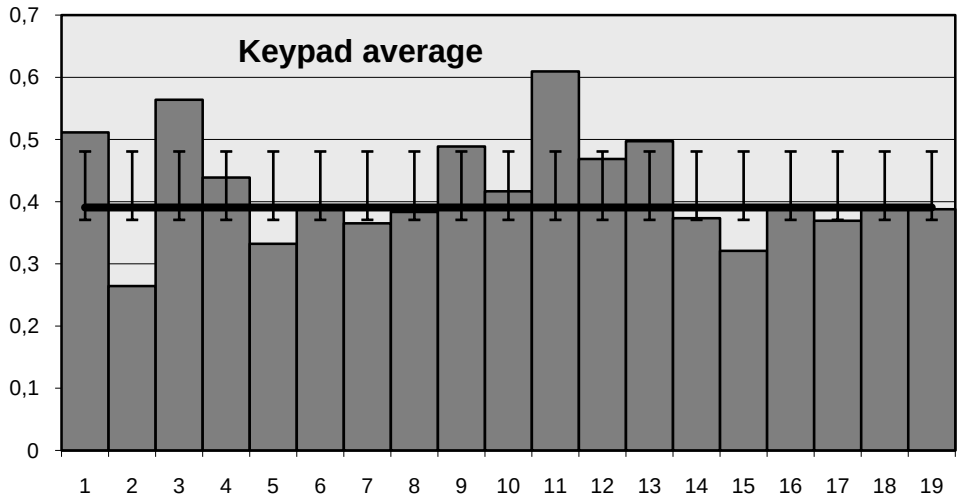


Figure 18: Average time in seconds for a keystroke on a mobile phone keypad of 19 persons. The median is 0.39 seconds; bars indicate range between 1st and 3rd quartiles.

Another interesting value we estimated is the mere physical action of pressing and releasing a key. It was measured by the key logger to be 0.13 seconds (the most experienced users were only 10 milliseconds faster, i.e. needed 0.12 seconds) and will be used to calculate *Finger Movement* time in the next section.

We measured standard keypad input separately from the hotkeys, although we did not take additional special hotkeys into account that can be found on several phone models on the side or top of the phone. For the hotkeys of the Nokia N90 phone (4 buttons and a 5 button joystick), ***K*** = 0.16 seconds were measured as a median. The smaller value can be easily explained with the smaller distance between buttons and the larger average size as well as the more direct and known semantic mapping of the buttons. See the modelling example on page 160 for a remark about how to model the time necessary to switch between the standard keypad and hotkeys.

These findings are close to those of related research. [Mori, Matsunobe, and Yamaoka 2003] for example also mentions 0.39 seconds. The original KLM suggests values between 0.08 and 1.20 with 0.28 seconds for a user with average routine on a standard sized desktop QWERTY keyboard. An average value for typing random characters is also given. This better resembles text input on mobile phones. The suggested value of 0.50 seconds again comes quite close to our estimate.

These values are meant for individual button presses or number input only. Several projects already verified and improved the Keystroke-Level Model of more complex variants of text entry. The results are quite diverse: [Dunlop and Crossan 2000] predict a value of 2.01 and 1.84 seconds on average for multi-tap and predictive text input, respectively. [How and Kan 2005] specify 1.32 and 1.00 for the same techniques, assuming an average SMS length of 60 characters. [Silfverberg, MacKenzie, and Korhonen 2000] also examined multi-tap and predictive text input and give values of 0.57 and 0.30 seconds for optimal expert use. The comparatively very small values result from only modelling the pointing component with the help of Fitts' Law, neglecting the time needed to find and actually press the buttons as well as verification. [Pavlovych and Stürzlinger 2004] calculate values ranging from 2.04 to 1.58 seconds for different input methods and suggest how those times should be adjusted for different states of routine.

These results might indicate that KLM does not work too well in this respect. However, [James and Reischel 2001] show that, although the predicted times can differ from actual performance times, relative relations between different designs prove to be correct and significant. Because of the rich set of publications in that area, we did conduct detailed studies for text entry. Some values were taken from the study used to find values for the *Distraction* operator described later.

Finger Movement (*F*)
From our observations during the tests for the *Keystroke* parameter we can report that most users verified less than every second number they typed. This means that the average total time needed to enter an 11 digit number was actually composed of 11 physical key presses, on average 4 *Micro Attention Shift*s, and 10 *Finger Movement*s. Since we know the values of the other operators, we can calculate ***F*** to be 0.24 seconds for all but the quickest users. According to our experience, full experts tend to only check their typing once during writing. Modelling that behaviour for the five quickest users results in the median value of ***F*** = 0.22. To additionally verify those assumptions, we ran an extra 10 tests using a mobile phone with a blinded display considerable reducing the use of *Micro Attention Shift*s. The upshot of this study was a median of ***F*** = 0.23 seconds. These results from the 323 keystrokes performed in the tests make it a very stable parameter. Figure 19 shows movement paths of three sample phone numbers types in the tests.

The value is also very close to what others like Silfverberg et al. found who measured 0.27 seconds with the thumb and 0.31 seconds using the slower two-handed index finger input [Silfverberg, MacKenzie, and Korhonen 2000]. In [Mori, Matsunobe, and Yamaoka 2003], the authors specify 0.19 seconds for ***F***.

When movements occur in the hotkey region *only* (as is the case for menu navigation sequences and when starting an application), ***F*** is smaller. Our studies indicate that the time drops to ***F*** = 0.16 seconds on average. Depending on the interaction, designers can choose which value fits better or use an average according to an assumed ratio of key uses.

Figure 19: Approximate finger movements (*F*) occurred while typing three different phone numbers on the Nokia N90 keypad.

3.4.3.2 General Parameters

Some parameters cannot be measured in a single specific setting. The system response time, for example, differs strongly depending on the phone model, the application running on it, and the action invoked. Also, the influence of mental preparation and the appropriate placement of the *Mental Act* operator has always been a complex issue in KLM models. Kieras gives several suggestions and heuristics specifying where and in what quantity the operator should be placed that also apply to the model as used in this work, [Kieras 1993]. Another parameter that belongs to the same category is the *Distraction* operator ***D*** that we newly introduced. It has not been treated in previous research on task models but we found that it has a considerable impact on time performance and there is a whole set of applications especially in the area of mobile interactions that are considerably influenced by distractive and disruptive factors.

Response Time (*R(t)* or *W(t)*)
As already discussed, information on system response times is supposed to be input to the model since these are highly diverse. We can, however, support the assumption of Silfverberg et al. that key presses in general have immediate feedback [Silfverberg, MacKenzie, and Korhonen 2000]. Menu browsing and selection was also running in negligible amounts of time on all phones we investigated. Starting applications needed anything between 0 and 6 seconds. For our setting, we only explicitly give values for the special cases when tags are detected (NFC) or pictures taken.

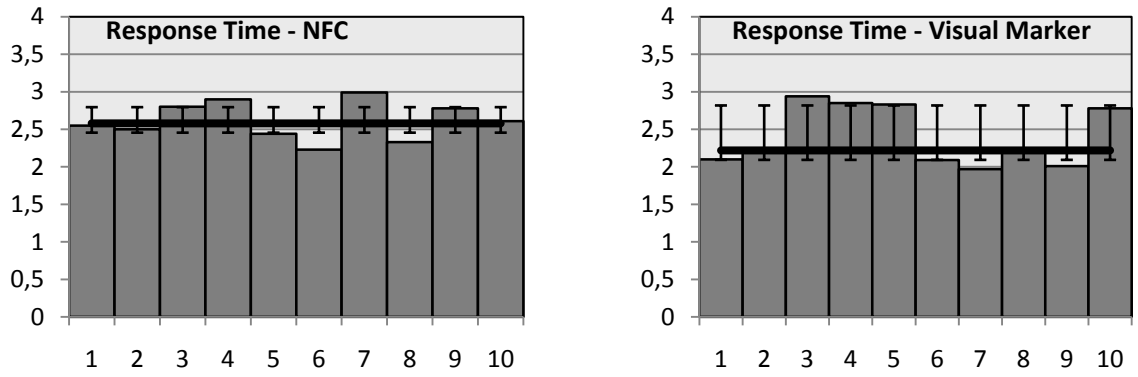


Figure 20: Measured response times in seconds for NFC and visual marker recognition with a Nokia 3220 and N71, respectively. Median values are 2.58 and 2.22 seconds; bars indicate range between 1st and 3rd quartile.

Our 10 tests with a Nokia 3220 with a built-in NFC reader showed that the processing of a tag takes on average R_{NFC} = 2.58 seconds. It should be noted that this time has significantly decreased and on the new Nokia 6131 it takes roughly 1 second. Using a Nokia N71, measurements for visual marker processing resulted in R_{marker} = 2.22 seconds, see Figure 20.

Distraction (*X*)

Interactions in the real world have to take into account various events that divert the concentration on the task at hand. This includes approaching people, passing cars, traffic lights, conversations, etc. Situations in which it is known that such distractions occur frequently (like at a bus stop) can be modelled with the *Distraction* operator.

People cope with such situations in different ways. They use their peripheral view, make quick glances, or introduce pauses. Initial tests showed that the behaviour also depends on the type of task. Thus, distraction cannot be easily modelled as certain specific actions. Through our tests we found that it is more appropriate to model distraction as a multiplicative factor rather than an additive operator.

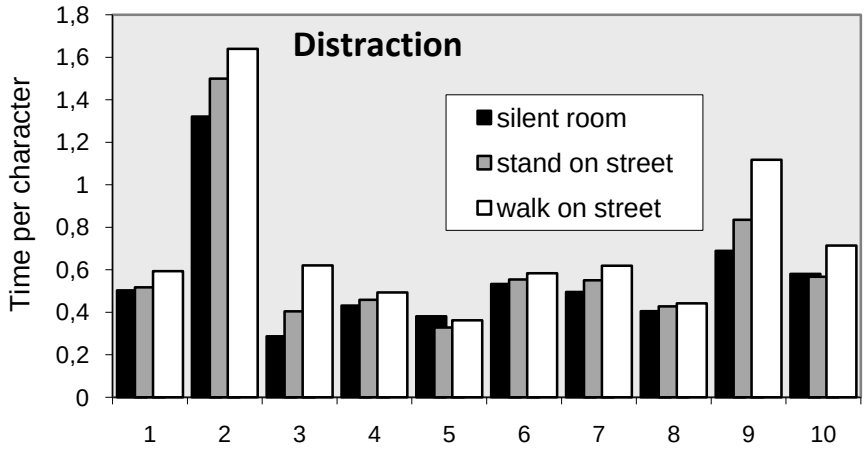


Figure 21: Average time in seconds to type one character on a mobile phone of 10 persons in three settings with different amount of distraction. Minor distraction adds about 6 %, major distraction about 21 % to the time.

Although the type and consequences of distractions can be manifold, several studies of distraction and multi-tasking, e.g., [Salvucci 2002], proved feasible in cognitive modelling. We give a simplified and rough but nevertheless justified idea how a task is probably slowed down by common side activities.

We ran three experiments, each with the same 10 people, aged 24-33, with an average of 26.7 years, 3 female. Subjects had to write a short message (about 90 characters) on their own phone in 3 different settings: a silent

room (experiment 1), standing on a busy street (experiment 2) and walking along and crossing that street (experiment 3). To obviate the possibility of sequence effects, we varied the order in which the participants conducted those trials. They also freely chose the contents of the message to write.

Typing speed of the participants varied considerably (0.76 to 3.49 characters per second in experiment 1). However, relative changes in the performance of each user are quite consistent. Experiments 2 and 3 revealed in an analysis of variance that the expected increase in time demand for distracted tasks is relevant well beyond the 5 % level ($t = 2.23$, $p < .03$ and $t = 3.28$, $p < .01$). The results suggest to add $\boldsymbol{X_{slight}} = 6$ % of the modelled time to the whole interaction if there is an anticipated slight distraction and $\boldsymbol{X_{strong}} = 21$ % for distractions that force persons to deviate from their task in a more rigorous or regular fashion. Figure 21 illustrates the data.

Mental Act (*M*)
As said earlier, we found no reason to change the value of ***M*** = 1.35 seconds from the original KLM. However, since we added and slightly changed the interpretation of some operators, we update the heuristics in [Kieras 1993] to place ***M***'s. The general principle remained the same: use Rule 0 to place ***M***'s and then cycle through Rules 1 to 5 for each ***M*** to see whether it should be deleted.

Rule 0 Place ***M***'s in front of all ***K***'s, ***H***'s, $\boldsymbol{S_{Macro}}$'s and ***G***'s.
Rule 1 If an operator following an ***M*** is anticipated in the operator before ***M***, delete the ***M*** (e.g., ***PMK*** becomes ***PK***).
Rule 2 If a string of ***MK***s belongs to a cognitive unit (e.g., writing a known number), then delete all ***M***'s but the first.
Rule 3 If a ***K*** is a redundant terminator (e.g., the selection key for entering submenus), then delete the ***M*** in front of it.
Rule 4 Delete the ***M*** in front of a ***H*** which describes the movement from the reading to the listening position.
Rule 5 If unsure, emphasise more the number than the placement of the occurrences of the ***M*** operator.

3.4.3.3 Parameter Values Overview

Table 6 shows the results of the studies. The median of each operator is given in the '*Time*' column. If applicable, the other two columns contain the 1st and 3rd quartiles, respectively. If not specified otherwise, all times are specified in seconds. The names of operators we newly introduced in this work have been set in bold.

Table 6: Overview of the proposed times for all operators. Newly defined operators are set in bold.

Operator		Time	Qu. 1	Qu. 3
A*, *Action	picture / marker	**1.23**	0.61	1.44
	NFC	**0.00**	-	-
	in general	*variable, input to model*		
B*, *Mouse Button Press		*not applicable*		
D*, *Mouse Drawing		*not applicable*		
F*, *Finger Movement		**0.23**	0.20	0.29
G*, *Gestures		**0.80**	0.73	0.87
H*, *Homing		**0.95**	0.81	1.00
I*, *Initial Act	external trigger	**5.32**	3.98	7.51
	self triggered	**3.89**	2.23	4.89
	optimal setting	**1.18**	1.10	1.26
	no assumptions	**4.61**	-	-
K*, *Keystroke	keypad average	**0.39**	0.37	0.48
	keypad quick	**0.33**	0.32	0.37
	hotkey	**0.16**	0.15	0.20

Operator		Time	Qu. 1	Qu. 3
M*, *Mental Act		**1.35**	-	-
P*, *Pointing		**1.00**	0.84	1.20
R*, *System Response Time	NFC	**2.58**	2.46	2.80
	visual marker	**2.22**	2.09	2.82
	in general	*variable, input to model*		
$\boldsymbol{S_{Macro}}$***, Macro Attention Shift***		**0.36**	0.28	0.44
$\boldsymbol{S_{Micro}}$***, Micro Attention Shift***	keypad ↔ display	**0.14**	0.14	0.19
	hotkey ↔ display	**0.12**	0.02	0.14
	keypad ↔ hotkey	**0.04**	0.02	0.12
	in general	**0.14**	0.10	0.16
X*, *Distraction	slight	**6 %**	3 %	13 %
	strong	**21 %**	11 %	25 %

3.4.4 Evaluation of the Extended KLM

Generating such values for specific operators has a long tradition in cognitive modelling. However, the mere setup of a study and the measurement of times are of limited value since it is not clear whether the new operators behave as expected when applied in the KLM context. The assumption of this model is that a task can be split into unit actions and the total time to perform the task can be predicted by the sum of the durations of all the unit actions. A conventional way of evaluating and adding credibility to such values is to apply the model to some real world tasks and compare the predictions to actual user data. Examples can be found in various works on the topic, e.g. [James and Reischel 2001] or [Teo and John 2006].

Therefore, in order to validate the values we found for the mobile phone KLM, we set up a scenario in our lab and modelled it using our new parameters. We then ran a user study and measured the actual times that each of the participants needed to perform certain tasks and then compared the predictions to the observed timing data.

Scenario

The scenario was based on a ticket service for public transportation in Munich, Germany. The tasks included the download of a service from a poster augmented with NFC tags (see [Broll et al. 2007]). The interaction device was a Nokia 3220 with built-in NFC reader. People had to order a ticket to a pre-defined target for three persons and write a text message to a friend about the expected time of arrival. In order to illustrate the applicability of the modelling approach, two different ways of accomplishing the main task were implemented: using a form input on the phone's web browser, and using the NFC capabilities of the phone, see Figure 22.

The participants were either routinely working with the used technology or trained before the study. None of them had taken part in any of the earlier studies. We also alternated the order in which participants had to use the two different types of technology to minimise any learning or other distracting effects

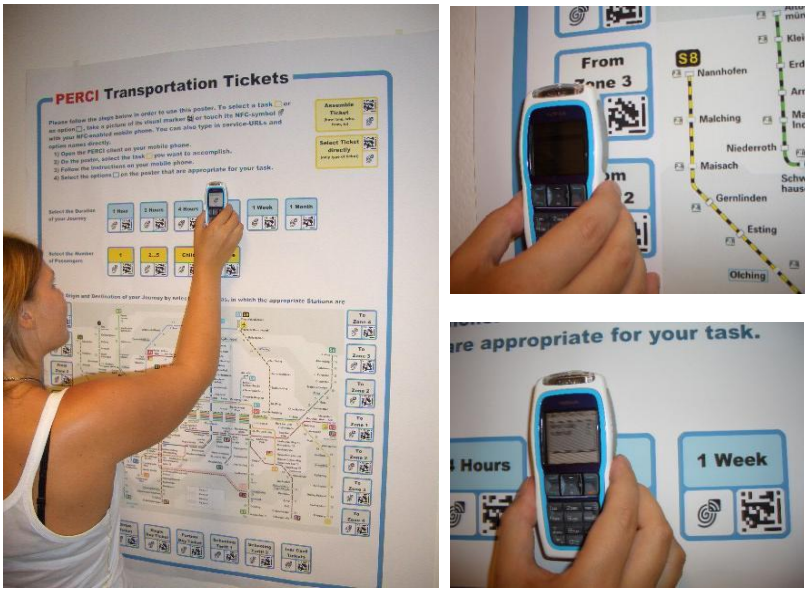


Figure 22: Setting for the evaluation of the phone KLM. Participants had to order a transportation ticket using, e.g., NFC interaction.

KLM Prediction

The KLM predicts 147.48 seconds for the NFC version of the scenario with a total of 135 operators. Table 7 shows some excerpts (the full model is listed in Table 16 in the Appendix). The model of the other variant of the scenario using direct input and browsing uses 110 operators and predicts 122.77 seconds (the full model can be found in the Appendix in Table 17). Distractions were neither observed nor modelled. The prediction was calculated before any user tests were made based on a detailed analysis of the scenario and the heuristics given in an earlier section.

Table 7: Selected sequences from the proposed mobile phone Keystroke-Level Model of a scenario based on NFC tags. See Table 16 in the Appendix for the full model.

Description	Operator	Allocated Time
Pick up the mobile phone	***I***	1.18 sec.
Enter main menu	***M, K***[Hotkey]	1.35 sec. + 0.16 sec.
Go to 'Programs'	***M,*** 3***K***[Hotkey]	1.35 sec. + 3*0.16 sec.
Select 'Programs'	***K***[Hotkey]	0.16 sec.
Go to 'Collection'	***M, K***[Hotkey]	1.35 sec. + 0.16 sec.
Select 'Collection'	***K***[Hotkey]	0.16 sec.
Select 'Choose program'	***K***[Hotkey]	0.16 sec.
Open application	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for program to open	***R***	4.63 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Attention shift from mobile phone to poster	***M***, S_{Macro}	1.35 sec., 0.36 sec.
Movement to tag	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.
Attention shift from poster to mobile phone	***M***, S_{Macro}	1.35 sec., 0.36 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Download ticket service	***K***[Hotkey]	0.16 sec.
…	…	…
	Sum:	**147.48 sec. ~ 2:27 min.**

Empirical Validation
Both alternatives of the scenario were completed by 9 people, aged 23-34 with an average of 27.6 years, 3 female. The times needed for the first version ranged from 106 to 133 seconds with an average of 117 seconds (σ = 9.40). The values are remarkably close to the predicted value of roughly 123 seconds. The upshots of the second, NFC version of the scenario are similar in magnitude: the average duration of the task was 137 seconds (times ranged between 120 and 162 seconds, σ = 13.1) which is also quite close to the KLM estimate of 147 seconds.

This means that the deviations of the KLM predictions to each data sample are in the good range of -15 % to +18 %. The measured averages actually deviate only 5 % and 8 %, respectively. Even more important, the speed loss of 16.7 % of the NFC implementation predicted by the KLM is confirmed by the study with a measured average increase in task completion time of 14.4 %.

3.4.5 Discussion and Related Work

It is evident that average users handle complex interaction styles differently and with different speed. It can also be hard to get into a routine for tasks that are new to a user even after several repetitions. This may render the expert user assumption difficult to support. The complexity of the interactions adds to this problem. However, our experience and evaluation show that for a set of interaction methods known to its users through at least sporadic use, estimates given by the mobile phone KLM are very good indeed. Especially when target users are early adopters or professional workers, it is very likely that these learn and adapt quickly and reach a state of experience that can be modelled close enough to make sound predictions.

We presented models of two different implementations of a real world scenario that also indicate that well grounded design decisions can be reached purely based on the model predictions. Nevertheless, we strongly encourage other researchers to expand our initial set of studies and examine our results through additional measurements and scenarios.

The collection of introduced operators is necessary to apply KLM to interactions such as those described in this work. According to the experiences in our lab and after reviewing relevant publications, we conclude that this set also suffices to capture the interactions possible with mobile phones at the current state of the art. Of course, time will inevitably bring different and additional types of interactions in the future for which new operators might have to be defined. Others might need adjustments when new or radically more time consuming variants are introduced (like complex gestures). It can also happen that some interactions become considerably easier (for example by getting rid of the need to accurately aim and focus for visual marker recognition). Nevertheless, after having conducted our tests, we are very positive that those changes are easy to integrate into the model and predictions can be made that lead to an early and valuable basis for design decisions.

As has been mentioned before and stated by several research papers, e.g. [Pavlovych and Stürzlinger 2004], the model can be very valuable even if the absolute times are not as accurate as in our evaluation. As we have seen, interesting results can be achieved. For example, it was quite surprising for us that the NFC version was indeed slower than the one using the phone's web browser. This can give a clue for deciding between two different designs. Of course, it should never be forgotten that there are other characteristics that influence users' preferences and might even have higher priority than time to completion. The NFC technique might be judged as having better affordance, being quicker to learn, offering a more novel and interesting method, being subjectively quicker, or simply as being more fun. These naturally are aspects that have to be taken into account when designing an application. We see it as further work to look into how such aspects could be modelled and potentially be integrated into a KLM. One example of such an integration is [Luo and Siewiorek 2007] where a prediction of energy consumption on mobile devices has been combined with a standard KLM.

Assuming that time to completion of tasks should play a central role in interface design, it should also be stressed that the model can provide insight into where exactly time is lost within a task. In the case of the NFC interaction, one of those parts was identified to be the response time until a tag was read and feedback provided.

Up to now, most research on performance measures for phone users has been limited to the input of text for short messages: an initial work by Dunlop and Crossan shows KLM operator sequences for three different text entry methods (traditional, predictive, and word completion) [Dunlop and Crossan 2000]. However, the authors adopted the original operator values used for desktop interaction which proved to be imprecise in this new environment. This is improved by How and Kan whose presented model is more fine-grained [How and Kan 2005]. They define 13 operators that more directly map onto the phone keyboard interface according to the different input methods (multi-tab etc.). New times are gathered from videotaped sessions with a small set of subjects and a message typing task. However, this unnecessarily complicates the modelling process. In a complementary approach pursued in 2004, non-perfect users are considered using the cognitive load operator to model input verification [Pavlovych and Stürzlinger 2004]. Although we do not focus on text messages, our model supports this view using *Micro Attention Shifts*. There is also work reporting on time measurements for keystrokes and the *Mental Act* operator for text input in different languages (see for example [Myung 2004] for the Korean language).

In addition to text input, Mori et al. studied how the time values of the original KLM operators apply to mobile phone menu navigation and conclude that the operator values fit quite well and suggest only minor modifications [Mori, Matsunobe, and Yamaoka 2003].

3.5 Further KLM Extensions

In Table 8, we present a composition of KLM operators concerning text input commonly used on small devices. One can see that there is sometimes considerable variance within the values given for the same method. This shows that care has to be taken to account for the assumptions set by the authors.

Table 8: Text entry speed on small device keypads for T9, multitap, Less-Tap, and some specific settings. Values are given as words per minute (WPM).

Text Entry Method		WPM	Reference
T9		25.0	[Dunlop and Crossan 1999]
T9	index finger (thumb):	45.7 (40.6)	[Silfverberg, MacKenzie, and Korhonen 2000]
T9, empirical	expert: novice:	20.4 9.09	[James and Reischel 2001] (they also give different values according to type of messages used, e.g. chat vs. newspaper)
T9, model		17.6	
T9	expert: novice:	4.80 3.80	[Cockburn and Siresena 2003]
T9	novice:	7.58	[Pavlovych and Stürzlinger 2004] [18]
Multitap		18.4	[Dunlop and Crossan 1999]
Multitap		14.9	[Dunlop and Crossan 2000]
Multitap with timeout kill button	index finger (thumb): (thumb):	22.5 (20.8) 27.2 (27.5)	[Silfverberg, MacKenzie, and Korhonen 2000]
Multitap, model		14.9	[James and Reischel 2001] (they also give different values according to type of messages used, e.g. chat vs. newspaper)
Multitap, empirical	expert: novice:	7.93 7.98	
Multitap	expert: novice:	8.20 5.00	[Cockburn and Siresena 2003]
Multitap		7.15	[Pavlovych and Stürzlinger 2003]
Multitap	novice	5.87	[Pavlovych and Stürzlinger 2004] [18]
Multitap (hotkeys or number input)	expert: novice:	30.4 25.7	[Holleis, Otto, et al. 2007]
Less-Tap		23.5	[Dunlop and Crossan 1999]
Less-Tap	index finger (thumb):	25.0 (22.2)	[Silfverberg, MacKenzie, and Korhonen 2000]
Less-Tap		7.82	[Pavlovych and Stürzlinger 2003]
Less-Tap	novice:	6.53	[Pavlovych and Stürzlinger 2004]
Especially designed Fastap keyboard	novice: expert:	7.10 8.50	[Cockburn and Siresena 2003]
Unistrokes		51.5	[Isokoski 2001] (values are calculated without taking inter-character times into account)
Roman hand printing		33.5	
Graffiti		35.5	
MDITIM (Minimal Device Independent Text Input Method)		31.1	
2 thumbs on small QWERTY keyboards		60.7	[MacKenzie and Soukoreff 2002]
Graffiti		17.3	[Fleetwood et al. 2002]

[18] Note that [Pavlovych and Stürzlinger 2004] reproduces slightly different values than given in the original text of Silfverberg et al. [Silfverberg, MacKenzie, and Korhonen 2000] for multitap and Less-Tap and swapped their predictions for use with index finger and thumb in the description.

Text input has been studied for a long time and rather extensively since it is one of the most difficult, tedious, and time consuming tasks on small devices. Thus, the keystroke operator has been adapted for specific input methods (also see for example the *Say* operator in Table 9) and specific keyboards. Examples for the latter include the keyboard found on the Ali-Scout navigation system with 22 buttons in two rows modelled in [Manes, Green, and Hunter 1996] or the 16 button input cursor device of a digitising pad in [Haunold and Kuhn 1994]. Substantial research has also gone into different layouts of keyboards, especially for soft keyboards. An overview can be found in [MacKenzie, Zhang, and Soukoreff 1999].

There is to our knowledge no published research yet that includes new mobile interaction techniques in its predictive user model. A beginning is indicated in [Luo and John 2005] and its follow-up [Teo and John 2006] where the authors show that the method can be soundly applied to handheld devices using stylus-based interfaces. They also present a tool they developed to automatically generate KLM models from storyboard descriptions. This tool, called the '*CogTool*' has been introduced in [John and Salvucci 2005]. The research group states in future work that they plan to apply their research and tool support to novel interfaces like speech input or gesture control.

A great advantage of the modularity of Keystroke-Level Models is that results from third parties can easily be reused. Therefore, efforts to measure and validate new operations can be valuable for many other research groups and industry applications. Obviously, one has to carefully inspect the setting and assumptions made during the measurement process. Sometimes, specific operators get adjusted after some time. An example is work by [Clarkson et al. 2007] who refine and validate assumptions underlying a previously published version of a model of two-thumb text entry. It should also be taken into consideration that there is a trade-off between having many, detailed operators and few slightly less exact operators. The more parameters a model has, the more complex it gets to model, understand, and subsequently interpret it.

Table 9 lists some results retrieved from selected publications where the authors measured the time necessary for specific operations outside the realm of text entry. Note that such a list can hardly be exhaustive. Especially since by far not all efforts in this direction have been published or are accessible somewhere. There are possibly hundreds of applications that have been modelled in industrial settings and whose modellers opted to derive new operators and made measurements for them. However, not all of them had the means or interest to make those additions available to the public. There is, for example, a body of work in the area of workplace evaluation that use similar models for their evaluations. However, it is difficult to find or get access to such values.

Nevertheless, there are still novel areas where active research also shows much interest in models such as the KLM. We presented in [Holleis, Kern, and Schmidt 2007] the concept of combining prototyping tools for tangible interfaces with user modelling aspects. We proposed mobile phone and in-car interfaces as specific application areas. There is also much work being done in the latter area of vehicular interfaces. Salvucci for example concentrates on the distracting aspects of interfaces for car drivers in [Salvucci 2002]. He and colleagues also developed an extension to the ACT-R cognitive architecture called Distract-R which helps in modelling and testing the impact of interfaces to driver distraction [Salvucci, Zuber, et al. 2005]. In a similar effort, Pettitt and co-authors extended the KLM and aim for replacing the cumbersome and time-consuming occlusion method for measuring the visual demand of in-car user interfaces, [Pettitt, Burnett, and Stevens 2007].

In the table, we did not include slight adaptations of existing parameters for very specific settings. For example, there are several projects in which deviating values for *Pointing* ***P*** and *Drawing* ***D*** were used, either based on Fitts' Law or measured empirically, since specific designs and interactions allow more specific predictions, e.g. [Haunold and Kuhn 1994]. In [Hinckley et al. 2006], for example, the authors present a setting where most mouse targets lie very close to each other and thus question the applicability of Fitts' Law or most average values proposed elsewhere in different settings. Other specialisations include tailored pointing operators such as using cursor control keys to control a large screen in a very early interactive display system [Kankaanpaa 1988].

There is little research present in the area of pervasive computing besides text input on small, mobile devices. As detailed in a later chapter, we see our work as a basis for further research to enable designers of a great variety of pervasive applications to use models such as the KLM early in the design process.

Table 9: Some operators with which the original KLM has been extended gathered from literature. If not specified otherwise, values are given in seconds.

Operator	Description	Value	Reference
Illumination	Adjustment factor to be added for low light conditions	10-15 %	[Manes, Green, and Hunter 1996] (they additionally give times for specific classes of buttons)
Age	Adjustment factor to be added for age middle: older:	45 % 119 %	
Search Constant, SC	Time to find an item for which a user cannot currently recall the name and location: M + SC * #items / 2	0.20	[Bälter 2000]
Scroll Constant, SCr	Time needed to scroll if some items are not visible: (1 - (#visibleItems) / (#items)) * SCr	2.60	
Query Formulation	Time to formulate a search query for emails	5.00	
Search	Time to search the display for an item	0.55	[Kankaanpaa 1988]
K_{pen}	Press a button on the pen input device	0.25	
Eye	Eye movement general	0.23	[Card, Newell, and Moran 1983]
$Perceive_{binary}$	Perceive a simple, binary signal	0.10	
$Perceive_{complex}$	Perceive a complex visual signal (word or code)	0.34	[John and Newell 1989]
Say, unpractised	Say a syllable in an unpractised sentence	0.17	[John 1990]
Say, practised	Say a syllable in a highly practised sentence	0.13	
Listen	Time to recognise a pause as a turn in a conversation	0.65	
Reach-far (Rf)	Hand movement from steering wheel to in-car device and v.v. (‘reach-far’)	0.45	[P. Green 2003]

4 Tools for Rapid Application Development

This chapter introduces the EIToolkit, a framework that simplifies the connection of various types of software and hardware applications, sensors, and actuators. 50 existing frameworks in this area are introduced and compared leading to 46 general requirements for such toolkits.

This chapter begins with a description and comparison of various types of existing prototyping toolkits (4.1). Based on these and existing literature, a list of general requirements for such toolkits is assembled (4.2) and the EIToolkit and a number of existing approaches are evaluated against that list. After a presentation of design considerations for the EIToolkit (4.3), we give a concise overview of its architecture and concentrate on some specific aspects and issues of its implementation (4.4).

4.1 A Review of Existing Prototyping Toolkits

This section summarises several products and projects that pursue similar aims as our EIToolkit, namely providing means to ease the combination of software and hardware to form 'intelligent' applications. We point out the strengths and weaknesses of these approaches and show where our toolkit builds on the strengths of those approaches and where it offers solutions to shortcomings revealed during this evaluation.

Most of the approaches discussed in this section will be compared and contrasted in Table 10 against a list of requirements for such toolkits defined and elaborated upon in the next section. These are cited also with their Arabic number found in Table 11 on page 82 in order to provide a more concise format and direct lookup.

The toolkits and frameworks presented here are split into three categories. *Hardware-focused toolkits* concentrate on physical building blocks and tangible components, while *software-centred projects* are more concerned with expanding current development environments and APIs. The last type subsumes all projects trying to *combine the physical and digital world* and propose solutions for development support in this area.

4.1.1 Hardware-focused Toolkits

Hardware components and products are often harder to develop than pieces of software. It requires physical skills ranging from the ability to work with crude tools such as saws, drilling machines, and hammers, to very fine tools such as soldering irons, needles, and pincers. Most importantly, however, a difficulty arises from the fact that many of the standard software prototyping approaches are not applicable to hardware: existing software code can be used and run, example code can be copied and adapted, software objects can easily be cloned, worked on collaboratively, sent and distributed, etc. Some researchers have started to provide developers with ready-made components to simplify use, reuse, adaptation, and combination of hardware. The EIToolkit does not directly include custom hardware components but focuses on integrating existing approaches within its model.

General Purpose

The beginning of tangible computing was marked by Ishii and Ullmer's work on tangible interfaces [Ishii and Ullmer 1997]. They introduced 'phicons' to represent physical icons. However, in this early work, most of these phicons were complex and highly specialized and the authors did not provide toolkit support for such interfaces. One way of trying to tackle the problem of a high threshold in creating hardware artefacts is to provide a base component that relieves the user from having to cope with various issues concerning power supply, connecting external parts, reading and processing sensor values, and transferring data between components or to a central component such as a PC. There are a variety of small yet very powerful processor boards available. Some of those even run complete versions of a Linux operating system[19] . This simplifies the use and programming of those components. However, the price that one has to pay for this power and comfort are short battery recharge cycles, impractical size, and considerable cost. Other examples in that direction include the Motes [Nachman et al. 2005] that are used by several research institutes and run the TinyOS operating system.

Particle Computer System
Less powerful but more energy saving, smaller, and cheaper are the Particle Computers [Decker et al. 2005]. The small boards (about 35x48x10mm) consist of a PIC microcontroller, 512kB flash memory, a ball switch to detect movement, some small lights, and an RF transceiver for sending data wirelessly to other components (sensor networks) or special receivers using a custom protocol. A connector collects most usable pins from the processor and enables attaching additional boards. One of those was explicitly built after an analysis of the most often used types of sensors in pervasive applications, namely movement, light, force, temperature, audio, humidity, and proximity [Beigl et al. 2004]. This makes the approach quite extensible, if only for those who know how to design and build custom printed circuit boards. In our experience however, the RF module exhibits problems even in close range (within a few meters), and makes the connection to other platforms such as mobile phones difficult. This is one of the reasons why we, after using them successfully for several projects, switched to a custom platform with Bluetooth transceiver. There is some support for high-level programming, e.g. using the standard parts and sensors without the need to reprogram code running on the microcontroller.

Arduino and Wiring
Similarly, Arduino and Wiring boards (see Section 6.2.2, Connection to Third Party Platforms and Components) are not powerful enough to run a complete operating system. However, the purpose of these prototyping platforms is mostly to gather sensor data, perform straightforward processing, and control simple actuators. Thus they have been used in many prototyping projects such as the Lilypad Arduino wearable control [Buechley, Eisenberg, et al. 2008]. One of the disadvantages is that programming the devices is still a rather low-level task.

Specialized Components

Several projects concentrate on the problem that those generic platforms often need to be physically extended and programmed to fulfil their task.

Phidgets
Arguably the most prominent example of a toolkit providing support for specific hardware needs is the Phidgets project [Greenberg and Fitchett 2001], [17]. Their physical widgets offer special-purpose, integrated components like a servo controller that are attached to a PC using USB. The advantage of this approach is that the hardware parts do not need to be programmed for most applications and thus the logics can be implemented in software on the host PC. Several APIs in different languages are available with the aim that developers can use their favourite language and use hardware components by simply reading from or assigning values to virtual versions of the physical objects. Phidgets are commercially available but are not targeted at hardware extensibility. Additionally, drawbacks include that they do not provide wireless components and cannot directly be combined with other toolkits.

[19] A long list of boards that support embedded Linux: *http://www.linuxdevices.com/articles/AT8498487406.html*

Calder Toolkit
Similar in principle is the Calder toolkit which provides several input and output components such as RFID (radio frequency identification) readers and tags, buttons, knobs, joysticks, and lights [Lee et al. 2004], [31]. These are connected either by short cables or by a wireless link to hub components that include a microcontroller and a USB connection to a PC. Different to Phidgets, the authors emphasize the use for designers who want to augment their physical designs with functionality thus bringing forward small devices with wireless capabilities. From a design perspective, this is close to the approach of the BOXES system described below but provides more functionality with the cost of slightly larger and less flexible modes of attachment. At a software level, the Calder components register themselves as USB Human Interface Devices and can be accessed using C code or wrappers around them. This is identical to the way Phidgets are used and does not provide means for non-programmers to create more than the simplest applications.

Voodoo-I/O
Such an example is Voodoo-I/O [Villar and Gellersen 2007], [Block, Villar, and Gellersen 2008]. It is still general in the sense that one basic component is connected to a PC by USB. However, it is specialized to simple controls on (currently) flat surfaces offering a large set of specific input and output modules. A special substrate, about 10 mm thick with a built-in conductive layer can be cut into any shape and is used as a basis for prototyping the looks and form of the desired device. A set of small controls such as buttons, knobs, sliders, and lights can then be attached at any location by simply firmly attaching them into the surface. A 1-wire bus system transfers information about the appearance, disappearance, and state of mobile components such as phones, sensors, or devices that are switched on or off, to a central controller that forwards these messages to a host PC. The foremost advantages are that there is no restriction in placing the components and that they can be repositioned at runtime, allowing a designer or user the opportunity to quickly try out different designs without having to make any changes to the running application. Different software controls are provided. By mapping inputs to simulated keystrokes, the majority of existing software applications can easily be controlled by physical representations of their inputs. Although the system is specialised to the supported components, not targeted at distributed applications, and does not support further interaction models, it can tremendously improve the rapid prototyping of simple hardware devices. There is potential in extending the system to 3D and distributed models when the technology becomes easier to fabricate and cheaper to get.

Crickets
The Crickets offer an embedded processor unit with bidirectional infrared communication and connections for two analogue inputs and outputs [Martin, Mikhak, and Silverman 2000], [36]. Additionally, several manufactured sensor and actuator boards such as a MIDI music synthesizer, numeric displays, or distance and sound sensors can be attached through a bus system making it extensible and simple to use. The Crickets also try to lower the threshold of building applications by running a virtual machine on their microcontroller and enabling the use of a simple dialect of the Logo programming language. There is no support for debugging. However, a dynamic mode lets users program in a simulated interactive style.

Lilypad Arduino
There are several additional projects that provide hardware components but are conceptually very similar to the ones described above with slightly different emphasis. Buechley, for example, introduces a sewable version of an Arduino microcontroller hardware board that is extensible in a way such that different sensors and actuators can simply be attached [Buechley, Eisenberg, et al. 2008]. The main focus of this project is to enable novices to work with basic electronics and to make such wearable controls aesthetically pleasing.

X10
Devices such as X10[20], for example, are built to reduce the infrastructural requirements for setting up a system with several nodes. These modules can communicate through an existing standard power network. Advances based on those as well as the iStuff toolkit are described below since they integrate hardware and software development much tighter than the previous, more clearly hardware-focused toolkits. An advantage of this toolkit is that there are many manufactured elements available such as motion sensors, lamps, and switches.

[20] X10 home automation components; product page: *http://www.x10.com/automation/*

4.1.2 Software-focused Toolkits

Many software solutions try to combine several types of technology, platforms, and different programming languages into one coherent development system. Examples include the Microsoft .NET suite that allows developing applications for platforms like Windows XP, Windows Vista, or Windows Mobile in languages like C++, C#, and Visual Basic, and enables combining those through various common technologies. Focusing on projects that in some way connect software development with external components, we list several application areas together with examples of tools used in this area.

We separate the systems into the following categories: *generic approaches*, *augmented reality and wearable computing*, *robotics*, *phones and other mobile devices*, *simulation and collaboration*, *tabletop interfaces*, *active spaces*, *intelligent environments*, and *end-user focused* approaches.

Generic Approaches

Plan B

A problem common to all infrastructures and middleware systems is that several prerequisites have to be fulfilled by the platforms it runs on. Plan B tries to minimise these requirements by using the file system concept present in nearly all operating systems on most higher-power computational units [Ballesteros et al. 2007], [5]. It provides file discovery services and also allows specifying some graphical user interfaces with file hierarchies. One of the advantages of using files is that users can inspect those and write custom data to them at any time. It is slightly more complicated to implement event-based services such as notifications or streaming devices such as a mouse, but it is possible. People who know how to look for specific data in the Plan B directory structure and can work with scripts and Unix-style pipes and bash-programming will find it easy to create simple applications like home automation systems. However, the initial threshold is quite high. Since most processing takes place while writing to and reading from files, it is difficult to isolate, find, and treat errors. It offers some support for adapting to resources that become unavailable (e.g. devices that are switched off or leave the radio range) and can easily be extended with additional services but makes quick prototypes and replacing (processing) parts more demanding.

Equip Component Toolkit

From the large Equator Project[21] emerged a toolkit called the Equip Component Toolkit [Greenhalgh et al. 2004], [18]. It is a highly component-based system that seeks to implement as few restrictions on applications as possible. Similar to the EIToolkit and also the iStuff toolkit described below, it employs an event-based tuple-space approach. To emphasise their claim for generality, they extend fixed tuple types to a subset of CORBA to allow complex, hierarchical types and matching of data between applications. This alleviates the restrictions of some other component-based systems such as P2PComp [Ferscha et al. 2004] and PCOM [Becker et al. 2004] that need to adhere to specific (identical) protocols for inter-component communication. While requiring considerable effort to setup an initial infrastructure, ‘inspectable properties’ and built-in protocol translators simplify the extension and observation of running systems. A flow-based graphical editor can aid in generating simple applications. However, the huge amount of visual components automatically generated from the CORBA descriptions complicates the first entry into the system. Besides iStuff, this is probably the framework that is most related to the EIToolkit. They share the focus on extensibility, support for external devices, portability, and diverse access for users including creating and observing applications. In contrast to the EIToolkit, it does not directly support control of existing applications, has no facilities to operate stream- or state-based, does not offer component simulation, and does not integrate into existing prototyping tools.

[21] The Equator six-year Interdisciplinary Research Collaboration (IRC); project page: *http://www.equator.ac.uk/*

PowerInteraction
A similar intent has recently been shown by the PowerInteraction[22] toolset also known as VID-Framework [Lorenz, Eisenhauer, and Zimmermann 2008]. The goal of the project is to provide a generic architecture to simplify the connection of available services and devices. It focuses on data input and provides mechanisms to abstract from physical hardware, software and input metaphors using the concept of virtual input devices. The implementation of the component-based framework uses standardised web services technologies to connect available services. It is actively developed in Java and Java ME but is not restricted to these languages. By using concrete implementations of the generic architecture, the framework enables a flexible combination of a variety of input and output sources and thus supports the modelling, rapid prototyping, and developing of dynamic and mobile distributed applications.

One.World
Another toolkit attempting to facilitate the development and deployment of pervasive applications is One.World [Grimm 2004], [19]. It builds on three requirements enabling and supporting roaming users, dynamic composition as well as combination of devices, and information exchange. The latter two are in common with the EIToolkit as are the use of events and self-describing tuples. A distinguishing feature is the use of environments (in other projects also named sessions) that bundle applications and data. Thus, One.World can, like the EIToolkit, store data persistently and, by nesting environments, other applications can observe and make use of the information passed in nested environments. However, it does not support replaying logged information or simulated components. Even though it allows easily extending the system and is based on the Java Runtime, it cannot directly command existing applications, has limited support for external devices and protocols, and does not integrate easily in design processes or other prototyping tools. It does, however, provide some mechanisms for application specific, automatic adaptation.

JSense, TASK, SNACK
The lack of support for debugging sensor applications and simulating components has been picked up by JSense [Santini et al. 2006], [48]. The central concept is a hardware abstraction layer that allows a user to directly access sensors and actuators without necessarily knowing their implementation details. It is aimed to provide information gathering and processing together in a centralised development environment using a high-level programming language such as Java. The implementation of these concepts allows near platform independence and a largely extensible system like the EIToolkit. However, it requires each sensor node to run a version of the abstraction layer, thus limiting the number of supported devices. The use of different programming languages is not easily possible and history, replay, or simulation of data is not taken into consideration. Other directly related sensor network development applications are the Tiny Application Sensor Kit (TASK) [Buonadonna et al. 2004] and the Extensible Sensing System EES (see the sidebar in [Szewczyk et al. 2004]). Their use is limited, however, since the former cannot directly be extended and both are targeted towards users without programming abilities, thus reducing the complexity of possible applications. The Sensor Network Construction Kit (SNACK) is an example of the opposite approach which only gives access to the computational nodes through a proprietary component composition language [Greenstein, Kohler, and Estrin 2004].

IrisNet
Such issues are attempted to be mitigated by Intel's IrisNet framework [Nath et al. 2002], [42]. It tries to deliver a system for world-wide sensing systems by making sensors available through a combination of web services with standardised databases and XML-based queries. It provides resource discovery and distribution of data. The latter allows shifting computational resources and bandwidth according to current needs. General assumptions of the IrisNet are that most processing can be done close to the sensing systems, that only high-level events need to be sent across the network, and that data can be pulled if of interest (e.g. a camera-based movement detector would not send its video stream; an interested application can query whether a movement has occurred and ask for an image at that point in time). While the EIToolkit also has the capabilities of accessing distributed and stored data, it is not optimised for general web-based access. IrisNet, on the other hand, cannot directly simulate sensor input and has a high threshold requiring a high competence in programming (a developer of a minimal service needs to write a browser front end, a 'senselet', and an XML schema).

[22] PowerInteraction, project page: *http://www.fit.fraunhofer.de/projects/mobiles-wissen/power-interaction.html*

Context Toolkit
Not explicitly web-based but also relying on the use of XML and HTTP, the Context Toolkit builds on a strict definition of context and implements so-called context widgets [Dey, Salber, and Abowd 2001], [13]. These are related to UI widgets with the distinction that generated events are asynchronous and can be modified on their way to a client. Data from external sensors and actuators can easily be included. As in the EIToolkit, components are reusable and distributed applications can easily be created. The implementation using Java ensures that it runs on most platforms and the lightweight protocol makes it easy to use with most devices. Device discovery mechanisms and simple to use interfaces provide a low initial threshold. However, knowledge about how to write call-back functions is necessary to work with the system. The existence of aggregators and interpreters in conjunction with the chance to use several programming languages such as Java, C++, Visual Basic, and Python allow creating rich applications. In contrast to the EIToolkit, simulation and replaying of logged data is not possible as are state- and query-based applications. The most prominent and lasting contribution is the concept of context widgets and various ways of combining inputs to context and processing it for services and applications.

.NET Micro Framework
Lastly, there are some programming environments that concentrate on developing applications on small devices and sensor nodes. The .NET Micro Framework (see for example [Thompson 2007]) brings the power of .NET programming to resource restricted devices (although there are still requirements that many sensor nodes do not fulfil, e.g., a minimum of 256kB RAM and 512kB Flash/ROM; it mostly supports ARM processors). The support here is much more on the side of single, local applications than the distributed ones using heterogeneous devices of pervasive computing.

Augmented Reality and Wearable Computing

One of the intersections of software applications and the physical world is augmented reality (AR). In these applications, information is projected over a view of the real world. Examples range from repair or construction work assistance with enhanced goggles to mobile tourist guides where data is included in the video stream of the camera. All of these applications have in common that some artefacts in the world have to be recognized. This can mean exact identification (e.g. face recognition), type identification (e.g. a product), or location / orientation (e.g. for manipulating virtual data with a tangible control). The ARToolkit (see page 8) is a very popular library that enables its users to create AR applications without the need of detailed knowledge in vision recognition and image processing. It uses small markers to detect the exact viewpoint of the user / camera. Several other libraries simplifying the use of AR technologies have been written. However, they are often still cumbersome to use and the combination and replacement of several of them is still a problem.

Developers Augmented Reality Toolkit (DART)
One approach to overcome those issues is the Developers Augmented Reality Toolkit (DART) [MacIntyre et al. 2004], [35]. Foremost, it is a framework that integrates into a designer's tool, namely Adobe Director and also uses its programming language Lingo. It supports rapid prototyping by using Wizard of Oz techniques for complex algorithms and integrates into the whole design cycle by helping designers to go from simple storyboards to 3D contents and 'animatics' (sequenced 2D storyboards with synchronised audio). As in the EIToolkit, capturing and using captured data is easily possible. Most important is that underlying sensing technology can be exchanged or updated without changing the application. It is one of the few tools which offer readymade solutions for a set of small but frequently arising problems by providing a suite of Lingo scripts. Drawback of the architecture are that only PC-based applications can be written, existing applications cannot be controlled, and it is strongly based on Lingo as well as DART specific protocols. It is, however, with some effort possible to connect external components other than cameras.

DWARF
There are also systems for augmented reality that try to enable developers to use existing components to create applications and to simply connect several services that implement commonly used tasks in augmented reality applications. The DWARF system, for example, is aimed at leveraging such libraries as the ARToolkit and provides modules bundling software and hardware pieces that belong together [Bauer et al. 2001]. A layered

architecture ensures that low-level hardware and software such as trackers and data storage are encapsulated and abstracted. The used middleware allows easily combining services and the architecture enables developers to write and integrate new components. Existing components can also mostly be configured using XML descriptions. The whole system is an extensible framework dedicated to a variety of different user profiles with a large focus on augmented reality applications.

Papier-Mâché
Another AR toolkit that additionally manages to combine sensors such as RFID with marker detection is Papier-Mâché [Klemmer, Li, et al. 2004], [30]. It generalises inputs by encapsulating detected objects into 'phobs'. Each of the three supported input types, i.e. RFID, barcode, and marker detection generates phobs with the IDs of objects that appear, disappear, or get modified. Each phob, however, can potentially carry additional information such as the picture that triggered the event of the visual system. This abstraction helps a developer in exchanging one input method with another, e.g., a system developed using vision can be deployed on an RFID system. The threshold of creating such applications is lowered by a simple monitoring environment that shows the state of each sensing technology and all currently detected objects. There is a simple graphical way of putting applications together. Still, most applications need to be developed using Java programming albeit very simple constructs are mostly enough. The EIToolkit does not provide explicit AR support. However, external vision systems can easily be connected whereas it is difficult to extend the Papier-Mâché system which is also limited in the number and types of protocols it uses. It is also not built for distributed applications and there is no further support for deploying the system.

A Construction Kit for Electronic Textiles
One common application area of augmented reality is wearable computing. This includes a variety of display technologies integrated into glasses as well as mobile devices. Wearable computing applications also use sensors with relatively low data rates such as temperature, accelerometers, tilt, motion, and bend sensors. Even though there is much work going on in this area, there are only few toolkits to aid in their development. Buechley describes hardware that allows the quick composition of individual components such as simple sensors (temperature, light, pressure) and actuators (lights, vibration motors). The kit supports integrating the microcontroller and other components into clothing by attaching conductive yarn to connect and power different parts [Buechley 2006]. Available components are still quite large and the programming of the microcontroller represents an obstacle for many inexperienced developers and hinders collaboration with designers.

Robotics

Vision technology and image processing algorithms needed for AR applications are also central for the development of various types of robots. Without going into much detail about application areas, conceptual and mechanical problems, two environments for prototyping and developing robotics applications shall be mentioned that are closely related to the EIToolkit and similar frameworks.

LEGO Mindstorms NXT
The LEGO Mindstorms NXT robotics kit[23] features a central processing unit based on an ARM7 microcontroller. It is able to control up to four sensors and three motors. Available sensors include light, touch, distance, movement (accelerometer), and direction (compass). The firmware and sensor specifications are openly available and thus the creation of custom sensors is possible though not directly supported. The system also provides a basic visual programming environment targeted at end-users. Several blocks can be put on a stage and are interpreted sequentially. Each block represents basic actions such as motor control or sound output as well as triggers based on attached sensors. In this way, simple programs in an if-then style can be quickly generated and downloaded on the processor. Complex programs can only be realized through the connection of other third-party programming languages. The abstractions found in the NXT systems hinder the kit's use for physical interfaces beyond robotics applications. However, through the easy combination with standard Lego parts, hardware prototypes can be built and adapted very easily. The combination with the provided software allows deployment on the (only supported) target platform but limits its use for debugging and runtime access.

[23] Lego Mindstorms NXT robotics kit; product page: *http://mindstorms.lego.com*

Microsoft Robotics Studio
Microsoft Robotics Studio[24] pursues similar goals but is more open in its applicability. In fact, it can also be used to create programs to be run with the LEGO Mindstorms NXT hardware. In contrast to that, however, it does not provide any particular hardware system and is targeted to the use of third-party robotics kits such as from fischertechnik[25]. A similar visual programming language is used to make the entry to robotics programming easier. However, the possibilities are much stronger including variables and more complex structures. Still, more challenging applications have to be written in other languages like those found in Microsoft Visual Studio (the visual constructs can be used to generate C# code). Although some concepts of the methodology used in the Robotics Studio could potentially be used in other fields, it concentrates on robotic applications. This is visible in both the hardware platforms it supports and the simulation environment in which applications can be tested. Similar to the EIToolkit it favours a loosely coupled infrastructure (by using decentralised software services), extensibility by third parties, and several reusable components providing useful functionality. On the other hand, it does not focus on discovery, state- or stream-based applications, does not leverage data processing (e.g. sensor input and context information), has little support for direct debugging (such as storing and replaying), and does not integrate into development cycles or other development tools.

Phones and other Mobile Devices

The deployment of applications in environments different than standard PCs is one of the core concepts of pervasive computing. Besides robots and wearable computers, mobile phones and PDAs are emerging platforms due to their large distribution, acceptance, and increasing power regarding processor and graphics capabilities. The creators of mobile phone hardware have slowly started to enable third-party programs on their hardware and various development environments and programming languages are now available for such platforms.

Carbide, EclipseME, NetBeans Mobility Pack
Nokia's Carbide IDE[26] (integrated development environment) started as a multi-language development tool for mobile phone programming. After having quit the development of the Java version Carbide.j, development is now supported mainly for C++. As with the other IDEs described next, it hides many activities necessary to generate a deployable phone program like compiling, pre-verifying, bundling, obfuscating, etc. and relieves the developer from being concerned with these steps. It is based on the Eclipse framework and can therefore exploit its plug-in system. In addition to coding, the IDE provides a visual interface for building graphical user interfaces targeted at mobile phone displays. Similar to this approach, plug-ins to the known IDEs NetBeans and Eclipse (the corresponding plug-in is EclipseME[27]) have been created in order to harness the power of existing development tools. They make use of many of the built-in features such as code highlighting, auto-completion, syntax checking, and compilation scripts. In addition, the NetBeans Mobility Pack plug-in[28] uses a graphical editor to design the screens of an application and a simple mapping of input widgets like buttons to transitions between screens. An emulator is also provided in which an application can be run. Its set of features is limited (e.g. no Bluetooth functionality) but it can be used to test whether the user interface works as expected. However, there are still differences with respect to the layout of user interface widgets that depend on the actual phone model and version of the operating system and are not modelled correctly. Programs developed using these frameworks use a subset of the Java language called Java Micro Edition, Java ME (formerly J2ME)[29] and is based on Sun's Java Wireless Toolkit for CLDC[30] (formerly J2ME Wireless Toolkit, J2ME WTK). Many phone models provide a Java runtime machine on which such programs can be run.

[24] Microsoft Robotics Studio; project page: *http://msdn2.microsoft.com/en-us/robotics/default.aspx*
[25] fischertechnik Robo Interface; product page: *http://www.fischertechnik.com/html/computing-robot-kits.html*
[26] Nokia's Carbide IDE; development page: *http://www.forum.nokia.com/main/resources/tools_and_sdks/carbide/*
[27] EclipseME, emerged from the mobile tools for the Java platform project; project page: *http://www.eclipseme.org/*
[28] NetBeans Mobility Pack, Java ME plug-in; development page: *http://mobility.netbeans.org/*
[29] Sun's Java Micro Edition, Java ME; developer page: *http://java.sun.com/javame/*
[30] Sun's Java Wireless Toolkit for the Connected Limited Device Configuration (CLDC); developer page: *http://java.sun.com/products/sjwtoolkit/*

Windows Mobile Platform, Maestro

A different approach has been chosen by Microsoft who advertises the Windows Mobile[31] platform on a set of more powerful devices. It is a restricted version of a Microsoft Windows operating system on which many of the programming paradigms for developing applications for Windows can be used. Thus, most functionality of the .NET framework can be employed. Maestro[32] is a commercial tool for prototyping mobile phone applications. It uses a hierarchical state machine, software simulation, and graphical widgets to help constructing graphical user interfaces. It is a powerful tool chain implemented in Java that can be used to create complex applications. However, its openness is restricted by the commercial licences and the effort of setting up the environment and the threshold to begin developing is quite high. The framework can be extended to develop software for embedded devices other than phones but it is tailored for Java ME devices that support MIDP[33].

iStuff Mobile

All these projects (and there are many more, e.g., for the iPhone) show that there is demand for such development tools. Most of them are limited in the sense that they are closed frameworks that do not work well in conjunction with other tools. There is no direct support for the integration of third-party applications, external hardware and prototyping is restricted by using limited emulators. iStuff Mobile tries to tackle these issues by combining mobile phone programs with a variety of hardware toolkits using a graphical composition language. It builds on the iStuff framework (see page 69) and extends the central event repository (called the event heap) to events from a mobile phone. A background application running on the phone is able to collect user input such as button clicks and relay this information via Bluetooth to a PC running iStuff. Reversely, events generated within the framework and targeted to the phone are captured by the background application and either directly executed or forwarded to the current foreground application. Within the event heap, data from other external devices, applications, and sensors can be made available. Thus, phone events and external events can be combined. By attaching sensors to the phone, this setup eases the process to create context aware mobile applications or to extend the phone's functionality with additional interaction methods such as gestures. iStuff Mobile also provides a way of simplifying development of these applications itself. By integrating the event heap structure into Quartz Composer[34], a graphical scripting language from Apple, a simple 'cable patching' method can be used to specify flow graphs and conditions to create basic application logic. In order to raise the ceiling of possible programs, JavaScript components are introduced to enable dynamic and more complex data processing. Obviously, the indirect connection between phone and sensors has the advantage that no direct hardware manipulations have to be made and sensors can easily be exchanged. On the other hand, a PC must be present as controller and a certain time lag is introduced. In contrast to the EIToolkit, simulating events and devices and distributed applications are not envisioned. There is no direct support for data processing, the use of Quartz Composer limits platform independence, and information about appearance and disappearance of parts is not available. However, the tight coupling of physical and virtual components and the opportunity to quickly create sensor enhanced phone prototypes make it a powerful tool in the hands of developers and designers alike.

Simulation and Collaboration

Mobile platforms are continuously getting more powerful and hardware is generally becoming cheaper. However, there are always reasons why an envisioned application cannot (yet) be developed on the target platform. It might not yet be available (if only in a certain country), still be too expensive, there might be delivery problems or licensing issues, devices might be in use by others, or there might simply be a lack of expertise to develop on that very device. Often, it is not clear at the beginning of a project what device, which configuration, and what infrastructure should be chosen. Since it can be very expensive with regards to both time and money to try out several alternative simulations can be very useful.

[31] Microsoft Windows Mobile platform; product page: *http://www.microsoft.com/windowsmobile/*
[32] Cybelius Maestro development suite; product page: *http://www.cybelius.com/products/default.htm*
[33] Java Specification Request 118, Mobile Information Device Profile (MIDP); specification page: *http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html*
[34] Apple's Quartz Composer software; main documentation: *http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer/*

UbiWise

One platform to test several options for pervasive computing applications is described in UbiWise [Barton and Vijayraghavan 2002], [7]. A 3D model of the environment is generated and devices can either be distributed or carried by people placed in this environment. The Quake III Arena game engine[35] is used as modelling and graphics engine for which several easy to use editors exist. A 3D first person view of the rooms and buildings of the virtual world is available which can show virtual objects such as dynamic picture frames. Another window shows the users' personal devices carried with them as well as close-ups of all other available devices and allows control of these using mouse and keyboard. For simulation purposes, parts of the graphical representation of a device can be assigned to act to mouse clicks (just as in [Hudson and Mankoff 2006], [23]) and output regions can be defined, e.g. for the use of a web browser. In this way, the interactions possible with a device like a camera can easily be simulated. UbiWise focuses, like the EIToolkit, on providing help in debugging applications by tracing devices and interactions as well as recording and playback of logged data. The simulation framework allows studying aspects like user interfaces, protocols , wireless transmission and location issues, and exploring and testing scenarios. Of course, the use of a 3D virtual environment and specific software makes using and extending the system more complex than in some other approaches.

Yamamoto

Yamamoto is a similar tool to visualise and simulate settings and scenarios for location based services [Stahl and Haupert 2006]. It uses a handcrafted modelling and visualisation tool to create maps and plans of buildings as well as their environment. It focuses on a simple and easy to use interface. As such, the creation of 3D models is for example facilitated by tools to help tracing the outline of objects from imagery such as scanned plans. All items are geo-referenced such that, e.g., several fine-grained models can be combined into a common world. Activation zones can be defined that trigger specific location based services. Sensors such as RFID and IR beacons can be arbitrarily placed and their ranges visualised as circle segments. The world can then be traversed in 2D or a 3D rendering. The underlying data model uses an event heap implementation. To that, various types of sensors can potentially be connected through a generic HTTP connection. More specifically, virtual displays can be added showing real-time content of external displays through a VNC[36] connection. Although the 3D modelling system of Yamamoto is designed in an extensible way, it currently is specialised on navigation. As such, a route finding algorithm is included that can help to design and test applications for pedestrian navigation.

FUSE Platform

Two important aspects for pervasive applications built on top of existing frameworks are the opportunity to allow users and devices (temporarily) leaving an environment, changing input and output devices, and moving between different ways of being connected to the system as well as having people work together (e.g. for computer supported collaborative work, CSCW). The FUSE platform achieves these goals by supporting roaming of users and their devices and maintaining session management responsible for creating, joining, and modifying sessions that keep track of users, services, and data [Izadi et al. 2004], [25]. In contrast to the EIToolkit, FUSE directly tackles connectivity issues as well as mobile and asynchronously working participants. During disconnections, messages to the client are stored such that appropriate ones can be repeated upon reappearance, i.e. the client is informed about important changes during its absence. The notion of 'migratable objects' enables clients to move data towards a session such that it remains accessible to others during phases of disconnection. FUSE is implemented using JNI[37] and supports standard protocols such as WAP and HTTP to include many different platforms and devices. Like the EIToolkit, it supports cooperation with other tools like the MASSIVE-3 system, a collaborative virtual 3D environment [Greenhalgh, Purbrick, and Snowdon 2000]. It is largely extensible through the use of community facilitators and protocol adaptors, allows a variety of platforms, protocols and programming languages, supports information logging, history, and replay, and provides different levels of abstractions. Its weaknesses lie in a considerable overhead for deploying the framework and lack of support for making errors and low-level events visible and traceable and for simulating components. The EIToolkit offers a few additional features such as state-based computing, control of off-the-shelf applications and specialised protocols for time critical and stream-based projects.

[35] Id Software, Quake III Arena; developer page: *http://www.idsoftware.com*
[36] Virtual Network Connection (VNC), remote display system; project page: *http://www.hep.phy.cam.ac.uk/vnc_docs/*
[37] Java Native Interface, JNI; project page: *http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html*

Tabletop interfaces

DiamondSpin, DiamondTouch
Another approach to connecting the physical with the virtual world is to use tabletop interfaces. They also suggest themselves for collaborative purposes since several people can view content and interact with it at the same time. Categorizing tabletop computing into software-focused toolkits does not mean that there is not major effort put into the development of the hardware. However, from a third-party point of view, only the development of APIs and other functionality makes such technology available for a wider range of applications and projects. Currently, Microsoft is trying to bring augmented tables into the public with their Surface project[38]. However, up to now there is no software developer kit (SDK) available to broadly make use of this technology. The DiamondSpin project [Shen et al. 2004], on the other hand, provides a full Java toolkit for collaborative interfaces on large interactive surfaces, specialized for people sitting around a common table. It features the full set of Java Swing's graphical user interfaces but makes them arbitrarily rotatable on differently shaped tabletops. Document visualization techniques, Fisheye transformations, and menu bars are supported as is the differentiation between shared and private areas on one display. Besides general interactional issues like the different viewing angle of participants, several problems regarding input arise. Besides technical challenges such as reliably detecting multiple fingers on the display – which is being solved – some problems arise because multiple people interact at the same time. First, the system should recognize who is currently manipulating an object. The DiamondTouch technology has a solution for up to four seated persons using conductive seat pads and capacitive sensing [Dietz and Leigh 2001]. Second, it should be possible to employ the other more conventional types of input such as mouse and keyboard.

SDGToolkit
Tse and Greenberg suggest a toolkit for single display groupware (SDG) that allows several mice and keyboards (or input pointing devices in general) to be attached to a display system [Tse and Greenberg 2004]. In order to make identification of mouse pointer easier for the users, the pointers and their coordinate space are rotated in accordance to the user's viewing angle. For developers, inputs generate standard mouse and keyboard events as in most interface toolkits but different users are distinguished by an additional identifier in the event parameters.

Synlab API
Although such systems are targeted at general collaboration between people, applications building on those toolkits are restricted to the tabletop platform. From a research perspective, combinations with other areas like ambient devices, intelligent rooms, and tangible interaction is of special interest. An attempt to combine tabletop interfaces with additional modalities such as tangible object that potentially have their own input interfaces is the Synlab API [Mazalek 2006]. It abstracts from the actual sensing method in use (e.g. computer vision, electromagnetic tags, or acoustic measurement) and allows additional events to be generated ranging from RFID tagged objects to physical manipulations of displays such as tilting or spinning. An application for such a combination (in this case tracking of visual markers on physical objects) can be found in the reacTable application [Jordà et al. 2007], a tangible music synthesizer application using a set of quite powerful widgets to control sound generation, filters, and modifiers. Many of the toolkits based on such multi-touch solutions base on the Open Sound Control (OSC) protocol, see for example the TUIO protocol [Kaltenbrunner and Bencina 2007] used by the reacTable, and can thus easily be integrated into the EIToolkit.

Active Spaces, Intelligent Environments

Tabletop interfaces are often used in conjunction with other large display technology and input methods like multi-touch and gestures. This research area appears with different names such as intelligent environments and active or smart spaces. The common idea is to augment one or several rooms, buildings, or even remote locations with input and output technology such that interactions can use the whole environment. Different to the idea of pervasive computing in general, these applications are still confined to certain areas and often applied in specific fields like augmented workplaces or aware homes. Scenarios envision that information such as a calendar stored on a person's mobile phone follows from the bath mirror to the kitchen wall, editing and combining pictures and

[38] Microsoft Surface, tabletop hardware and software; project page: *http://www.microsoft.com/surface*

documents takes place on a central table making use of large displays in the surroundings, simple gestures can be used to push images to a picture frame or printer, or simply send and store it somewhere. Sometimes, such environments are proactive and context aware such that they can adapt to the user's needs and provide information (only) when and where it is needed. Underlying principles include the collaboration of several types of technology, high affordance of the interaction methods, implicit and explicit input, and the use of various modalities for interactions. The EIToolkit supports several of those interaction models, although of course the concrete implementation of the devices and coordination modules has to be done on top of it. Besides the iStuff (see below) toolkit that exhibits a tight integration of physical and virtual prototyping and is therefore described below in the section about combining hardware and software development, three frameworks developed specifically for such active spaces shall be highlighted.

SpeakEasy
First, the SpeakEasy project [Newman, Izadi, et al. 2002], [43] is a recombinant framework that seeks to enable the ad hoc connection of services and targets user interface generation for mobile devices. The underlying concept of connecting different applications is based on mobile code, i.e. code that can be downloaded to and then executed on a different device. This includes 'typehandlers' that can convert data between different formats and 'session objects' that are shared between applications in order to initiate and control their connection. This renders the system very flexible and extensible even during runtime. The EIToolkit has better support for directly controlling existing applications. However, dynamic type conversions and connections are far less formalized and less strictly defined. Even though SpeakEasy contributes a lot with respect to user interface generation, it does not directly provide visualizations of available services, interfaces, or data flow.

Obje Perception Framework (OPF)
The Obje Perception Framework (OPF) builds on Obje, an infrastructure derived from SpeakEasy [Van Kleek et al. 2006], [52]. The OPF adds support for context acquisition and processing. In comparison with tools that mostly concentrate on context inference such as Exemplar ([Hartmann, Abdulla, et al. 2007]), OPF also concentrates on system features such as interoperability, mobility, and distributed applications. Nevertheless, it uses concepts from systems such as the Context Toolkit ([Dey, Salber, and Abowd 2001]), e.g. preceptors (context widgets) and aggregators (combiners and interpreters). A prominent feature of OPF is that, once a query for a specific context (specified using tuples and variables) is available, the system first tries to answer it with existing resources using a discovery service and, in case of failure, automatically and dynamically uses a pipeline system to instantiate available aggregators. By holding references to objects in use, aggregators can also be released when not needed, thus minimising communication. This is targeted at one of the system's main goals, namely to control and reduce energy consumption. This is further achieved by, e.g., allowing aggregators to push code towards the sensing preceptors. The system is implemented in Python and various auxiliary libraries and projects are available. An advantage with respect to projects using tuple spaces such as iStuff described below, is that no centralised instance is necessary, making it easier to deploy the whole system.

Gaia
The last project in this category is called Gaia, [Román et al. 2002], [47]. It has similarities with the OPF system but focuses less on energy awareness and the construction of context inferring pipelines. It is set up as an operating system and thus provides (abstractions of) several features of common operating systems such as program execution facilities, I/O and file operations, and resource allocation. It explicitly allows external sensors to gather information about location, temperature, weather and the like through context providers. As in InterPlay ([Messer et al. 2006], [39]) described below, tuples with a structure drawn from English language are used to store context information. A generalised model-view-controller (MVC) pattern allows, among other things, more generic views of input and output devices and one input device to control several applications. As a framework, it uses a generic mapping system between the requirements of applications to available services but relies heavily on script-based rules and actions. A security model and automatic service discovery are listed as future work. In comparison with the EIToolkit, it presents a very much extensible and dynamic structure. It lacks, however, integration of graphical, existing prototyping, and design tools, and does not provide direct support of control of existing programs, deployment, and different data protocols. It is open in some sense, though, as for example [Ranganathan, Al-Muhtadi, and Campbell 2004] describes an approach using the context predicate system of Gaia to allow coping with imprecise data using various approaches like fuzzy logics and

Bayesian networks. In [Ranganathan, Chetan, et al. 2005], Ranaganathan et al. have further developed the Gaia infrastructure to provide a programming model. It provides virtual active spaces with sets of operators (such as 'in' to describe that a person is in a certain location, independent from available locating methods) and uses semantic matching on top of ontologies to automatically discover services and execute queries and programs. By restricting themselves to a specific ontology, combinations with other active spaces become more difficult for developers and integrators and the many automatic decisions reduce the amount of control and understanding end-users have of the system.

End-user Focused

Lastly, one branch of research should be mentioned that could be applied to most of the projects described above. After a system has been developed and deployed, it is rare in pervasive computing scenarios that the system performs optimally over time without any changes. Users are mobile and environments and requirements can be highly dynamic. Besides the ideas described above to automatically adapt to certain changes (e.g. by using the mapping system in Gaia), it is also possible to include the user in the adaptation process. Of course, the threshold has to be sufficiently low to avoid long training times and frustration.

Customizable Pervasive Applications, VisualRDK
Weis et al. show a tool with which professional developers implement applications for smart homes. Just before deploying it, a customisation step is introduced that allows end-users to adapt and combine different components with each other, [Weis, Handte, et al. 2006], [53]. By using a graphical programming language, users connect events from one component to input commands of another component e.g. to control the volume of a media player with a mobile device. The graphical language has been subsequently extended by the same authors and now enables further end-user programming capabilities [Weis, Knoll, et al. 2007]. This is similar to the tools we describe in Section 5.2, Graphical, State-based Application Development. The composition language uses concepts like 'component contracts' and 'signals & slots' to achieve compatibility between components. In comparison with the EIToolkit, the underlying PCOM system [Becker et al. 2004] also allows quick extensions, adaptations, and alterations of developed systems. It has, though, little support for debugging and data processing techniques. The plug-in mechanism allows the control of existing applications but still provides a rather closed setup which does not integrate too easily into the development cycle of pervasive applications.

Interplay
While VisualRDK is not targeted at the average end-user but at a person with basic knowledge of programming who is very much interested in technology and in configuring applications, Interplay builds on an interface for users who think task-based and not technology-based [Messer et al. 2006], [39]. Their approach is to let users compose pseudo-English sentences to express their desires. The components of these sentences specify what should be done ('verb') with which object ('subject') at what location ('target device'). When a user specifies to play (verb) a certain movie (subject) on a specific TV set (target device), a 'Task Orchestration Layer' collects, prioritises, and converts this sentence according to existing rules, device capabilities, and location information. The 'Seamless Device Integration' layer is then responsible to, among other things, support a single directory where information about devices, content, and users is stored and maintains sessions (elsewhere also called environments). This layer relies on external middleware solutions such as Universal Plug and Play (UPnP[39]) that can discover and connect different components. The prototypical implementation is based on Java and uses the Web Ontology Language (OWL[40]) and the Resource Description Framework (RDF[41]) to describe and serialize device and task descriptions. Rules are processed using the Java Expert System Shell (JESS[42]) rule engine. The (not optimized) prototype has considerable feedback times in the range of seconds which is slow for an interactive system. Its formal basis and layered architecture leaves much room for extensions and potential for dynamic changes of the system. However, from a toolkit point of view, it is still a closed infrastructure and does not support debugging, simulation, error recovery, or combination with other higher-level frameworks.

[39] Universal Plug and Play, UPnP; UPnP Forum: *http://www.upnp.org*
[40] Web Ontology Language, OWL; project page: *http://www.w3.org/TR/owl-features*
[41] Resource Description Framework, RDF; project page: *http://www.w3.org/RDF*
[42] Java Expert System Shell, JESS; project page: *http://herzberg.ca.sandia.gov/jess*

Jigsaw
Another approach to give end-users the power of combining sensors, events, and services is presented as Jigsaw in [24] and a subsequent publication that focuses less on the underlying system but more on scenarios and user views [Rodden, Crabtree, et al. 2004]. A jigsaw metaphor is introduced to have end-users connect inputs (physical to digital transformers), processing (digital transformers), and outputs (digital to physical transformers) in the shape of puzzle pieces. The metaphor helps to choose the correct pieces for each function and to find matching parts. However, only simple, linear control flow can be created and the usability depends much on the graphics of the icons on the puzzle pieces. An interesting add-on is the manifestation of the digital jigsaw pieces with paper pieces that can tangibly be connected.

SiteView
The idea of using tangible objects to control an active space has also been adopted in the SiteView prototype [Beckmann and Dey 2003]. However, the link between the physical world and digital manipulators is much more eminent. The main interaction space is a floor plan of the surroundings where the user can place physical objects (detected using integrated RFID tags) to express rules and actions. Each of those objects, called 'interactors', either specify conditions using time or weather information, or actions such as light or temperature control. The described prototype suffers from a very small space of possible rules but shows potential in extending the system. The authors conclude that the system is easily usable to create simple rules but that there might be space to customise the tools for different user groups such that more expert users could override physical constraints in combining constraints (deliberately introduced to keep the complexity down).

4.1.3 Toolkits Tightly Combining Hardware and Software

BOXES
A way of tackling the difficulties in building custom hardware different to those described above has been pursued by the creators of the BOXES system [Hudson and Mankoff 2006], [23]. They use existing, tangible objects, possibly ones that have been created by using specific formable shapes or foam (often used by designers to create rough shape models) and attach electrodes to it to simulate physical objects with input components. The electrodes are connected to a small sensor board that uses a touch sensing circuit to detect when a finger gets in contact with the electrode. The board generates events and sends them to the PC using a USB connection (that could potentially be replaced by a small wireless Bluetooth sender). The combination with software is realised by a tool that further supports the rapid prototyping of applications often without requiring explicit knowledge in programming at all. The tool is capable of simulating key presses and mouse clicks at arbitrary locations. Developers map the touch of an electrode to a series of mouse and keyboard events and can thus use the features of available programs for their designs. This system is similar to the approach of the EIToolkit and allows a large diversity of actions. Still, for applications that depend on previous states (such as a simple toggle button) or more complex actions, custom programs need to be written. Compared to the EIToolkit, the system integrates very well into the way designers prototype but does not include further external sensors (a potential connection to the Calder Toolkit is only mentioned), does not support actuators, has rather closed protocols, and a strict focus on local, PC-based application development.

Real World Interfaces
While the X10 protocol (see page 57) and devices (wired and also wireless options are available[43]) are very much hardware related, the Real World Interfaces project attempts to leverage it from a purely device-centric view to a toolkit to create real world interfaces [McCrickard, Bussert, and Wrighton 2003], [38]. Several layers and abstractions have been developed to remove some limits of the protocol such as blocking of the communication bus until an addressed component replies. The combination with a variety of interfaces to languages such as C and C++ and especially interface toolkits like Amulet and Tcl/Tk render the system much easier to use for simple and complex applications. Still, very limited support for different hardware and software, for discovery, higher data rates, debugging and error control as well as integration into other toolkits or development processes make it less suitable for general use.

[43] X10 home automation, wireless components; product page: *http://www.x10wirelesshome.com/*

iStuff

A more generic approach, although also with a slight emphasis on augmented environments, is pursued by the iStuff system [Ballagas, Ringel, et al. 2003], [4]. It tends to concentrate on the software side but also provides several hardware components. Besides supporting X10 devices mentioned before, provided input devices are buttons, sliders, and other augmented physical objects like a mouse, a pen (for handwriting recognition), and a microphone (for speech recognition). On the output side, simple buzzers, speakers, and light controls are offered. On the software side, there are many similarities with the EIToolkit system. iStuff builds on the iROS application model which in turn uses an event heap data structure that passes, stores, and provides information as tuples. This decouples applications from underlying services and from each other in time as well as in space allowing for observation, change, and on-the-fly remapping of data flow. The latter can be done by using the PatchPanel software [Ballagas, Szybalski, and Fox 2004] which allows mapping tuples from input to output. It also makes data accessible in a whole subnet and thus enables distributed applications. The iStuff infrastructure has permeated to different application and research areas and by now also includes iCrafter, a framework for describing, discovering, combining, and automatically generating graphical user interfaces tailored for the use on several devices with different requirements and capabilities. From a hardware point of view, in comparison with devices that are provided by, e.g., [Lee et al. 2004], [31] or [Hudson and Mankoff 2006], [23], iStuff hardware is mostly too large to be used in integrated devices for information embedding. However, the toolkit system is largely extensible, is compatible with several platforms and programming languages, and the type of applications that can be created is nearly unrestricted. One of the most prominent strengths of the system is the inherent support for dynamically adapting and changing a running system. Nevertheless, even though this leverages rapid prototyping, there is hardly any integration or combination with other prototyping or designer's tools. In contrast to the EIToolkit, support for understanding, debugging, and simulating parts of the system is missing.

d.tools, Exemplar

iStuff does not provide any means to directly see the relationship between physical artefacts and their virtual counterparts. In contrast, d.tools features a direct connection between hardware and software components [Hartmann, Klemmer, et al. 2006], [22]. The appearance and disappearance of a component is visually apparent and changes in the hardware are immediately shown in the virtual representation. Vice versa, Wizard of Oz style development is possible. The d.tools system has been designed to integrate the whole cycle of iteratively designing, testing, and analysing applications. The hardware parts consist of a microcontroller board connected to a PC using USB. Sensors compatible with the standard I²C protocol[44] can be attached to such interface boards which also support the direct connection of simple buttons and sensors that output data by adjusting their resistance or voltage. Since information passing is done in the open OSC protocol, other devices and applications that use OSC can be incorporated as well. In contrast to other toolkits like the EIToolkit or Calder [Lee et al. 2004], [31], only wired components are currently provided. d.tools lets users build applications in a simple way. First, a 2D graphical representation of the envisioned product is drawn and icons for sensors and actuators present in the hardware prototype are placed into the representation. Next, the planned semantics of the device is described by creating a graph of the different states of the device and specifying the actions between such states. Issues with such graphical, state-based approaches include the large number of states necessary to describe even small systems, that all actions are sequential, and that the possible complexity of applications is limited. Therefore, several methods are applied in order to allow more complex applications. Parallelism can be achieved by allowing several states to be active at the same time. Java code can be attached to states to generate behaviour otherwise difficult or impossible. Finally, testing prototypes is supported by recording video data of the interaction with a device, correlating video scenes with events (i.e. actions). An add-on called Exemplar helps in processing available sensor data [Hartmann, Abdulla, et al. 2007], [21]. It allows for demonstrating actions using hardware sensors and provides a software visualisation for selecting, filtering, pattern matching, and generalising such input to create events based on future, similar input. Shortcomings of d.tools are the need for a central device such as a PC (this is shared with the EIToolkit), the inability to control existing, off-the-shelf components, and the lack of exploitation of the state-based approach. In contrast to the EIToolkit, the framework incorporates iterative design and directly supports analysing and testing applications. However, the dependence on its implementation in Java, the integration into the Eclipse IDE, and the restriction to use the graphical environment with code attachments limits its practical use.

[44] I²C protocol, Philips Semiconductors, now developed by NXP; project page: *http://www.semiconductors.philips.com/i2c*

4.2 Toolkit Requirements for Pervasive Applications

In this subsection, we present important requirements that a system used for prototyping and developing pervasive applications should support. Although we cannot claim that it is exhaustive (especially since a specific project might have some very specific requirements in addition to the ones mentioned below), it surely represents a large quantity of the issues concerned with such approaches. The list results from experiences in our own research group and the discussion and collaboration with colleagues and researchers of other institutions. Several meetings like the EIToolkit workshop[45] or DIPSO'07 [Kawsar et al. 2007] workshops, and more domain specific ones like Classroom of the Future workshop [Mäkitalo-Siegl et al. 2007] also contributed to this compilation. Most importantly, requirements were derived from a detailed literature study and the review of the toolkits presented in the last section.

Again, whenever we cite a publication that was used as a source for this list or was evaluated against this list, we also cite its Arabic number as listed in Table 11 on page 82. This provides a more concise way to write and a quicker way to access the publication.

There are several more aspects that are listed in, e.g. [Banavar et al. 2000] that can be important to develop pervasive applications. Some of them are specifically concerned with specialised application domains like augmented environments [Román et al. 2002], [47]. We also examined these projects but focused on those requirements we found of particular, general importance and those that have been directly mentioned as requirements by at least three independent sources. Thus, for each of the requirements, a list of references is given that explicitly argue for the importance of that requirement. Note that this does not necessarily mean that the respective project is successfully implementing it. Reversely, projects that implement a particular requirement but do not explicitly specify and rectify it are not listed. For example, several toolkits support creating on-screen PC programs but not all argue that this is important for people to create and debug distributed, mobile applications. Thus, we tried to distinguish between the requirements of those systems and the features that a particular system actually offers (the latter is afterwards presented in Table 10 for a selection of projects). Several papers talk about evaluating pervasive systems, e.g. [Neely et al. 2008] where the authors report about the discussions in several workshops on the topic of evaluation methods, or address general requirements, e.g. [Want and Pering 2005], which lists power management, discovery, user interface adaptation, and location-aware computing. Another recent publication, [da Costa, Yamin, and Geyer 2008], elaborates on challenges for pervasive applications derived from previous publications and own experiences of the authors. They bring forward ten abstract categories and provide very generic approaches and types of technology that can help to tackle these (e.g. use virtual machines and interoperable protocols to achieve heterogeneity). Most of them directly map to requirements listed on the following pages such as context awareness and management, mobility, dependability, heterogeneity, and spontaneous interoperation. Others such as invisibility or transparent user interaction are more abstract and difficult to be considered in underlying development tools. However, these challenges are not directly targeted at the systems with which pervasive applications are built, but rather to the systems themselves. Still, this should of course have an impact on development environments. Therefore, we try to incorporate those at least indirectly. Intelligent power management, for example, can be supported by being able to shift computation in a system. We directly include discovery and see that location- or more general context aware computing is one implication of the ability to use several external sensor systems.

Edwards and Grinter provide seven challenges for ubiquitous computing applications in the home, [Edwards and Grinter 2001]. Besides problems of heterogeneity of devices, extensibility over time, reliability, and potential mismatches between the mental models of its users and the actual behaviour and implementation of the system, most identified issues touch social, historical, maintenance, and design issues as well as missing domain knowledge. These aspects require much further work in order to see how these can be supported by design processes and development tools. A very similar reasoning can be applied to the 14 challenges of [Henricksen, Indulska, and Rakotonirainy 2001]. They distinguish four groups, namely support for devices, for software components, for users, and for user interfaces. They state that pervasive computing development tools mostly support the second and third group of challenges.

[45] EIToolkit Workshop. December 07-09, 2005; web page: *https://wiki.medien.ifi.lmu.de/view/HCILab/EIToolkitWorkshop*

We categorise our list of gathered requirements into four areas.

a) **Support hardware, software and development paradigms**: different properties such a system needs to support an appropriate variety of hardware, software and development paradigms. This includes devices, protocols as well as different ways of development.

b) **Creating applications**: the developer should be aided in creating simple as well as complex programs quickly and easily. The system should give support for several different approaches.

c) **Debugging and changing applications**: during and after the process of creating applications, a paramount property of development support tools is to provide help and guidance in debugging and exchanging parts of an application.

d) **Integrating (into) the development process**: the set of provided tools should fit into the development process currently applied. This can be realised in two ways. Either existing programs should be able to integrate the new tools or the new tool chain incorporates the current development process.

For all of the requirements in the following list, we mention whether and how our EIToolkit fulfils each of those.

a) Support Hardware, Software, and Paradigms

A development toolkit needs to provide support for various devices, protocols and programs. It should abstract from their respective capabilities and encapsulate them into an interface that is as simple and general as possible but provides access to functionality as detailed as necessary. In addition to such components, certain types of development processes, paradigms and programming languages should be acknowledged. A third item in this sense includes infrastructural issues such as portability and other more general properties of a system like security aspects.

Supported components

- *Control off-the-shelf and proprietary programs*: [3, 11, 16, 21, 22, 25, 26, 27, 29, 53]
  Commercial off the shelf (COTS) devices and programs that are not open-source rarely offer a well defined, open API to control them. Some motivations for using such devices can be found in descriptions of a project done in our group to connect a standard Nokia 770 internet tablet to Particle sensor nodes. Such modifications lie outside the capabilities of a software toolkit. However, software programs that do not provide an accessible interface can often be controlled at least in some aspects.
  The EIToolkit supplies, e.g., a component to emulate keystrokes and send them to an arbitrary application.

- *Allow external hardware sensors / actuators / devices*: [3, 4, 20, 22, 28, 29, 31, 32, 40, 41, 42]
  One of the most important features for pervasive applications is support for a variety of sensors such as cameras, movement sensors, light sensors as well as actuators like displays, motors, and lights.
  The EIToolkit uses several protocols and communication methods to reach as many external components as possible. A mechanism to easily exchange and expand the mode of communication assures that it is open for future innovations.

- *Support different protocols and devices in general*: [7, 8, 11, 20, 25, 28, 37, 45]
  Besides lower-level communication protocols (UDP, TCP, Bluetooth, ...), a toolkit should enable applications using higher-level protocols. If not natively supported, it should be comparatively easy to connect a novel device.
  The EIToolkit supports protocols like OSC and RTP that build on other lower-level protocols like UDP and TCP, respectively. This broadens the available base of components that are directly supported. The stub concept helps in keeping the necessary additional layer between the toolkit and any new devices / protocols as minimal as possible.

- *Support the communication between devices, applications and people*: [4, 7, 18, 20, 25, 34, 35, 40, 42]
  Passing data is central in many applications, especially if sensors and actuators are used. The exchange of information should be made simple, visible, and controllable. This includes devices, applications as well as users, and other people involved in the system.
  The EIToolkit offers a central communication area through which most messages are passed. Components interested in all or specific messages can register themselves and are notified appropriately. The messages can be visualised, logged and replayed. It does not directly support more high-level and abstract ways of enhancing person to person communication as, e.g., in [Hilliges, Terrenghi, et al. 2007] in which the authors describe a system for enhancing communication between people taking part in a collaborative setting focused on creativity and brainstorming.

- *Support wired as well as wireless components*: [4, 31]
  The way to communicate with external components should not depend on a specific mode. For some projects, it is more important to have a reliable power source while for some others unrestricted movement is more critical.
  The EIToolkit supports several protocols, e.g., serial line (wired) and Bluetooth (wireless).

Infrastructural aspects and general properties

- *Support for deploying the system*: [1, 12, 15, 53]
  Developed systems often run on simulated devices and / or on one machine instead of on distributed nodes. Deployment generates several new issues that need to be tackled, see, e.g. [Davies and Gellersen 2002] or the special issue of IEEE Pervasive Computing on Real-World Deployments [Fox et al. 2006].
  A possible approach is described in [Andersson 2000].
  The EIToolkit does not offer help in this respect other than the separation of components by design.

- *Reusable components*: [4, 10, 13, 16, 18, 27, 31, 42, 52]
  To be able to support the rapid construction of applications, parts of a toolkit that can be used in different situations should be encapsulated into reusable components. These components should also remain exchangeable throughout the development process. This includes sensors, actuators as well as whole devices and possibly applications.
  The EIToolkit is mainly based on message passing. The sender of particular messages can be exchanged as long as the messages remain the same. With regard to such events, the toolkit enables components to define and describe their interfaces. As we describe, e.g., in [Holleis and Schmidt 2005], components can then easily be exchanged according to their interface and interface inheritance.

- *Portability across multiple platforms*: [4, 22, 23, 27, 28, 42, 45]
  This relates to several previous items. Applications should be able to run – with appropriate efforts – on different platforms. Designers often use the Mac environment while more technology-oriented groups may prefer Linux variants. A toolkit should be available on several of those and the toolkit should enable communication across all of those, especially in sight of a cooperation of different roles.
  The EIToolkit does not depend on a specific platform and supports message passing over protocols that are platform and programming environment independent. It encapsulates the communication facilities so that these can be exchanged or augmented with specific platform dependent ones. Components can be written and adapted to run on arbitrary platforms that support some form of external communication.

- *Build 'local' as well as distributed applications*: [8, 28, 41, 53]
  Applications that run locally should also be able to run in a distributed manner, i.e. the single components can be separated. This implies some sort of treatment of time synchronisation issues.
  Since the EIToolkit is based on passing messages between components over UDP or similar protocols, applications are not limited to run on a single machine. For time critical applications, it relies on time-stamps and the capabilities of OSC, see the next item.

- *Allow high data rates and time critical applications*: [21, 35, 45, 47, 52]
  First, this aspect addresses the issue of applications requiring that events are generated and data is passed within certain time bounds. This strongly depends on the devices, load of the network, amount of data sent, as well as other infrastructural limitations. Second, it refers to applications that need to generate or react on events in a specific timely order or at specific absolute or relative points in time, e.g. those generating music. Packets sent through the EIToolkit include timestamps. In addition, as described in more detail on page 84, the OSC protocol provides some mechanisms to time messages. There is no dedicated control about particular delivery times. For high data rates, the EIToolkit supports different protocols, e.g. for streaming.

- *Operate packet-based as well as stream-based*: [21, 35, 41, 52]
  Although packet-based messages sent through, e.g., UDP or TCP are appropriate for events, several types of applications (using cameras, music, etc.) require stream-based data exchange.
  The EIToolkit supports both, packet-based and stream-based communication.

- *Use standardised but flexible protocols*: [4, 8, 18, 29]
  There must either be a compromise between a fixed standard protocol and an open format, or it must be possible to switch between several formats. There are obvious advantages of using standard protocols followed by most of the toolkits referenced in this section. However, projects like [Greenhalgh et al. 2004], [18] that use powerful systems (CORBA, in this case) suffer from problems such as complex structures and difficulties in observing and simulating messages. Others such as [Johanson and Fox 2002] that use a custom format harden the combination with other components.
  The EIToolkit suggests and implements a simple, general packet format but leaves the formatting of the specific contents to the data provider and custom `PacketFormatters` can always be added for specific needs.

- *Inform about appearance and disappearance*: [6, 8, 13, 22, 25, 27, 29, 41, 42, 45, 52]
  Discovery of components can be an important factor for an application. It should also be made known when a component is no longer available.
  Developers of components for the EIToolkit should make their components known when they appear. They should provide information on request from a central component. The information can then be used to check whether a component is 'still alive'. The developer is supported but not forced to adhere to this guideline.

- *Ability to shift or distribute computation load in the system*: [10, 37, 41]
  In a heterogeneous system, some devices are more powerful than others. It should be possible to move critical or heavy processing to those machines.
  The EIToolkit does not provide automatic dynamic scheduling or load shift algorithms in its basic version. However, the developer can decide which stubs to run where and have them carry out the main calculations while others merely relay data.

There are several classical technical criteria such as performance or scalability that are doubtlessly important for any infrastructure. However, as described in [Edwards, Bellotti, et al. 2003], we focus more on the "value for end-users" than on core "technical workability". In [Bass and Kates 2001], the authors also elaborate on several dozens of connections between software architecture and usability. Still, two aspects that are mentioned as requirements for such infrastructures in the literature reviewed shall be explicitly mentioned: first, **security** and **privacy** [29, 34, 37, 40]. This aspect covers many topics including access control and authentication. Most security related issues relate to the stability of the system and to guard private information from external observers. In general, several applications using common data should be separated such that one cannot know about the other if this is not desired. The EIToolkit is designed as an open system and does not directly employ security related tools or standards. Of course, the communication system can easily be enhanced with cryptographic methods to shield delicate information. Second, **robustness** [11, 28, 29, 34] incorporates other issues like single point of failure, network disruptions, and dependency between components but is closely linked to the criterion to be stable with respect to changes and errors, stated in c) below.

b) Creating Applications

The main task of a development toolkit is to assist its users in creating applications with all the subsystems and components it supports. Developers should be made aware of components that are not directly supported and guide them to a possible solution, i.e. help to connect those to the toolkit.

Low threshold

- *Ease of use*: [31] and almost all others
  Writing applications should be easy and quick to learn; existing programs should be easy to read for others and comprehensible to novice users. A main aspect is that it should be simple to express certain algorithms. However, some toolkits may target a specific group or experts in some area; e.g. the authors of [Weis, Handte, et al. 2006], [53] state that they try to make it possible that "technically interested persons can develop customizations that can be used by others as well".
  For the EIToolkit, the message passing system is easy enough to understand for developers. People without much background in technology or programming can rely on add-on tools such as those described in Section 5.2, Graphical, State-based Application Development.

- *Simple things must be simple*: [4, 14, 17, 22, 32, 35, 40]
  The system should allow simple applications to be developed quickly, simply and with as little programming knowledge as possible.
  In the EIToolkit, connecting components with each other is just a matter of converting messages.

- *Hide implementation details*: [20, 22, 35, 43, 52]
  A main function of a toolkit is to hide often difficult implementation details, while exposing functionality through a well-defined interface.
  The EIToolkit hides details like discovery and communication protocols in components that communicate through a standard channel. The details of the messages, though, are left to the developer of the component.

- *Abstract and package*: [6, 8, 20, 28, 48, 52]
  Express standard and non-standard input and output controls through well designed interfaces and packages and combine those with the same or similar user interface.
  The EIToolkit allows but does not enforce the combination of similar components.

- *Separate services from applications*: [25, 29, 35, 39, 40]
  Application development should be separated from those parts that offer access to sensors, actuators, and other internal or external components.
  The EIToolkit deliberately separates a hierarchy of stubs from the one of applications.

- *Support or integrate visualisations*: [3, 16, 22, 28, 42, 52, 53]
  A not too complex visualisation of components and their relationships can provide a quick entry point for beginners and an overview of the whole and parts of the system.
  Note that this does not necessarily mean support for a visual programming language although this is a possibility, e.g. statechart-based systems like presented in Chapter 7.
  The component and message-based architecture of the EIToolkit is particularly suitable for graphical visualisations. This can also imply supporting the movement of user interfaces to devices with specific requirements or visualising data flow.

- *Allow additional tools on top*: [3, 6, 8, 17, 28, 51]
  A toolkit should offer means to extend its way of development. For example, it should enable tools on top that lower the required amount of programming hence enabling a different view on developing applications.
  The EIToolkit allows this, see for example Section 5.2, Graphical, State-based Application Development.

- *Scheme to link physical and virtual components*: [22, 35, 44, 52]
  To render the control and programming of external components easier, it helps to have some identification scheme to link external parts with software components, i.e. to simplify identification and use if, for example, some sensor is attached.
  The EIToolkit supports the specification of unique IDs for components but has no way of checking their validity or uniqueness.

- *Don't impose a specific architecture / paradigm*: [35]
  Let developers the freedom to choose an architecture / paradigm of their choice.
  The EIToolkit has a focus on event-based processing but is free in the development of components and applications, including programming paradigms and languages.

High ceiling

- *Allow complex applications to be built*: [4, 22, 35, 42] and almost all others
  Although there is often no need to have a Turing-complete language, a nearly arbitrary set of applications and algorithms should be possible to build.
  The EIToolkit allows the full power of most available programming languages to be used.

- *Support data processing*: [20, 21, 28, 37]
  An integral part in many applications in pervasive computing is the access and processing of sensor data. Transformations such as filtering, smoothing and aggregating data should be made available.
  The EIToolkit does not contain an extensive library of such transformations. However, it is easy to connect external libraries to the system and it supports the hierarchical application of such filters.

- *Provide solutions to common tasks*: [2, 21, 35, 52]
  In systems using heterogeneous, distributed components, several identical or at least similar tasks have to be solved in many projects. Common examples are creating and parsing messages, combining them, processing sensor data, and accessing external libraries. Wizards, samples, pre-packaged processing libraries and access to a variety of systems and protocols are possible ways to support this.
  The EIToolkit provides a set of components for many standard problems as described in the next chapter.

- *Support the re-use of fragments*: [20, 28, 40, 41]
  Often, software parts can be written that are applicable in a more general way than for one specific purpose. It should be possible to write and re-use such components or patterns for similar problems.
  The EIToolkit is based on independent components and as such well-suited for this approach.

Ways of development

- *Support hierarchical data processing*: [11, 13, 20, 43, 54]
  As has been stated before, data exchange and processing is often a critical part of an application. Raw sensor data, e.g., almost always needs special treatment before it can be used sensibly in applications. Most developers use a model as suggested in [Gellersen and Beigl 2002] to generate more high-level events from low-level data. In a hierarchical process, data is processed generating higher-level 'cues' that abstract from raw data. A pressure sensor mounted on a chair, e.g., can provide continuous pressure data. A next level generates 'occupied' / 'not occupied' events using a threshold algorithm. Another, possibly application dependent, level on top of that can, for instance, combine that information with a building entry identification component and subsequently supply information about 'person A is sitting at a desk'. iCrafter [Ponnekanti et al. 2001] built on top of [Ballagas, Ringel, et al. 2003], [4] is another example that uses such higher-level components.
  The EIToolkit offers support for such kind of hierarchical data processing and at the same time gives access to all layers. This is especially useful for generating a sense of the current context of an activity.

- *Allow different programming environments and languages*: [18, 26, 27, 28, 47]
  It is important to support a multitude of different development environments, platforms and languages to ensure that a broad enough user base can potentially use the system. Such widespread support does not need to be natively built into a system. However, the system must be open in that sense.
  The EIToolkit provides direct implementations in C++, Java, and C#. Since the central design is based on decoupled components communicating through a common communication area accessible through standard connections and protocols, it is possible to connect almost any device or application that has a method of communication. For example, all programs and languages that allow browsing to a web page are supported.

- *Support event-based development*: [3, 13, 28, 29, 41, 42, 51, 52]
  Many applications follow an event-driven model, reacting on pushed information. Input components generate an event in a specific interval or whenever a change occurs. This in general keeps the used data rate low enough and generating applications simple and understandable.
  The EIToolkit is based on a publish-subscribe or observer model. This means that the standard way of interacting with components is to generate and listen to events.

- *Support pulling for data (as opposed to, potentially event-driven, pushing)*: [13, 25, 41, 43, 54]
  For some reasons such as the use of devices that need to keep their power consumption very low or if continuous sending of data would be excessive, a pull model that allows applications to query for information exactly in the moment they need it can be more appropriate. A toolkit should support both pull and push models to cater for all specific application needs.
  Even though the EIToolkit uses an observer model, each component stub can decide to adjust its behaviour by listening to control or request messages from other applications. This means that it is possible that a stub's output is controlled by others. Problems arising by conflicting requests (e.g., 'send every second' versus 'send every minute') have to be handled by each component themselves. It is conceivable, though, to write a component that receives such control messages and finds solutions for such problems and forwards adjusted messages. There is a stub that provides data when requested by HTTP requests and one that writes messages into a database for asynchronous querying.

- *Support state-based development*: [22, 27, 32, 33]
  Often, specifications of applications concentrate on the result of some event. To a lesser extent it is important how this is achieved. The more detailed knowledge is necessary to understand a process, the more details should be left to the back-end. One way of interpreting this is to think of a state-based approach. If the desired result is known, the component only needs to be brought into that specific state.
  The EIToolkit directly supports a state-based approach. It models the visible, interface related and potentially internal state of components. As described in Section 5.2.1, the system can be used to bring a component into a required state without needing to know the necessary sequence of events beforehand. Tools such as the mobile device development IDE described in Chapter 7 use a state-based approach to allow for simple programming by demonstration.

- *Create on-screen PC programs*: [28, 32]
  Although many applications in the domain of pervasive computing focus on distributed systems that depart from the known desktop computer setting, it is important to be able to create programs that run on a standard PC. This assists in keeping the focus on the involved components, helps in developing and debugging, and it simplifies communication, demonstration, collaboration, and distribution of software. This should be complemented with support for deploying the final system on the target hardware and software platforms.
  The EIToolkit focuses on developing applications using a variety of distributed components. At the same time, applications running on a single PC are of course possible and encouraged.

c) Debugging and Changing Applications

During and after the development phase, it is desirable – and often necessary – that programs can easily be fixed and changed. One part in this process is concerned with debugging and correcting bugs. This means that the logics of the program as well as the data and communication flow should be well observable. Another part includes all those cases where programs have to be adapted to fit a different set of controls, operating system, or even change to fit different requirements.

Observing and debugging

- *Storage* [13, 25, 26, 28, 42], *history* [13, 25], *logging* [22, 26], *and replay* [32, 35]:
  Data produced and required by each component can be distributed or shared, permanent or volatile, etc. To take advantage of tests, logging of messages and data is extremely important. This is especially true for long-term tests or those with deployed, possibly external setups. To understand situations that led to a certain state or to use a realistic emulation of external components, it should be possible to replay created logs. For several types of application, e.g. those based on learning, accessing data from the past can essential.
  The EIToolkit does not impose any restrictions with regard to storage. Any component can use resources within its capabilities and provide data, for example as an RTP stream, to other components. There are generic components that can persistently store and replay messages. The EIToolkit offers components that can log and later resend all or a specific subset of messages. Another component can be used to write (a selection of) messages into a separate database that can be used as an alternative data source.

- *Allow remote control and observation*: [4, 13, 16, 22, 26, 29, 52]
  Maintaining a running system is much easier if it can be done remotely. Debugging code, e.g. by setting breakpoints, stepping through the execution, inspecting the current value of variables, etc., helps to find and fix bugs. An important aspect is to be able to observe all messages passed in the system.
  The EIToolkit relies on the extensive set of debugging tools included in IDEs such as Visual Studio or Eclipse. In addition, it can visualise all communicated data and allows remote observation and influence.

- *Simulate components*: [7, 22, 32, 51, 52]
  Simulating components is often necessary, e.g. if the actual components are not available or do not run in some environment. Sometimes, a specific output needs to be generated, potentially the same output several times. Many components like sensors hardly ever produce the same data twice during a certain time span.
  The EIToolkit is optimised in a way that actual components like input and output devices can be replaced with software components without changing the rest of the system.

- *Find errors and issues in the idea, design and architecture of an application*: [22]
  The development steps executed before any implementation is started could be part of a toolkit.
  The EIToolkit has only very little support for such high-level decisions apart from offering the opportunity of trying out several ideas or device variants with often very little effort.

- *Make errors and exceptions accessible and potentially offer solutions*: [47]
  Predictable errors (such as a non-existent device) as well as unexpected ones can occur during development and runtime. If possible, these should be made obvious and help in treating those should be suggested.
  The EIToolkit mostly relies on the exception handling of the programming languages. For inter-component communication, the standard messaging mechanism with messages of special types can be used.

- *Provide feedback about what is happening behind the scenes*: [13]
  The power of abstracting functionality is that users do not need to know details. But nevertheless, it can sometimes be necessary to know about implementation details, e.g. for timing issues. It should be possible to get as much information about abstractions in use as possible.
  The EIToolkit is based on independent components. Currently, only guidelines exist specifying that they should be descriptive with respect to their internals. We are in the process of developing further ways for stubs to provide information about their capabilities, interface, and physical structure.

Changing and updating

- *Allow incrementally extending the system*: [2] and almost all others
  Applications should be able to adapt and change according to needs of their users and the environment. The authors of [Rodden and Benford 2003], [46], for example, argue that the dynamics of buildings should influence the development of domestic applications. In the same sense, it should be possible to extend development tools with technology, techniques, and other novel achievements.
  The component-based architecture of the EIToolkit fits perfectly to this requirement.

- *Support quick alterations*: [22, 31, 34, 42, 51, 52]
  Trying out different designs or implementations should be quick and easy.
  Since the EIToolkit is based on independent components, these can be replaced with ease.

- *Allow easily and quickly exchanging input, output and application logic*: [4, 6, 11, 13, 22, 28, 36, 40, 42, 52]
  It should be possible to treat input, output, and application logic differently.
  Each component of the EIToolkit should be designed in order to fulfil exactly one of these three tasks. This makes input, output, and processing parts independent.

- *Be stable with respect to changes and errors*: [6, 10, 25, 26, 27, 29, 36, 42, 43, 47]
  Applications should need only minor changes if parts are exchanged or show faulty behaviour.
  The component concept of the EIToolkit ensures that parts offering the same interface can be exchanged without interruption. Only little configuration, e.g. reformatting messages is necessary in most other cases.

d) Integrating (into) the Development Process

The usability of a set of tools or development environments is sometimes reduced because it does not fit into the development process of the users. There are two ways of how to combine a new tool to an existing development approach. Either the tools allow integrating the process into their own system or the tools can be integrated into that process improving parts of it. The d.tools project [Hartmann, Klemmer, et al. 2006], [22] is an example of the first approach. It uses a design-test-analysis cycle model and users can use their own methods to execute the three parts. However, it also prescribes a lot of the process which reduces the freedom of development.

The EIToolkit is guided by the second approach and provides assistance in several steps in the development process, most notably the generation of low and high-fidelity prototypes. Through the connection to existing tools like Adobe Flash and functions like logging and replay, it also supports the analysis and study of existing or created applications.

- *Support the different phases of the application design*: [22, 31, 32, 35]
  The process of developing an application should all be supported by a toolkit. Each accomplishment should be transferred and reusable in the following phase. This includes phases like drafting the concept, implementing prototypes, testing a design, iterating over it and deploying the application.
  The EIToolkit as such is most probably too low-level to be useful for all phases. However, it can serve as a basis for appropriate tools, see, e.g., Section 5.2, Graphical, State-based Application Development. It is especially valuable for quick prototyping and testing of variants.

- *Support the use of design tools*: [22, 31, 35, 42]
  Well-known design tools such as Adobe Director should be directly supported.
  The EIToolkit does not integrate directly into Adobe Director or other design tools.

- *Connect to existing prototyping and programming tools*: [4, 17, 22]
  Tools often used for prototyping like Adobe Flash should be supported. This also includes procedures normally not supported by computers such as paper-prototyping. [Hudson and Mankoff 2006] [23] shows how paper prototyping methods could be employed for such purposes.
  The EIToolkit has been integrated into the Eclipse IDE, the d.tools project [Hartmann, Klemmer, et al. 2006], [22] and Flash. It can be used from within programming languages such as Visual Basic that support one of the toolkit's protocols (that includes all programming languages that can access web pages).

There are a couple of further features that often appear in requirements lists. Some of them are mostly consequences of the requirements listed above and thus not listed as a separate item.

Scalability [26, 29] is an issue on several layers. At a low level, data passing mechanisms must not clobber the network. That can be assured by having exchangeable protocols (that can be optimised and distribute the load) and event- and stream-based messaging. At a higher level, hierarchical data processing methods, filters, and visualisations can ensure that developers are not overburdened with masses of data. **Context** [13, 29, 50], [Yau et al. 2002] can be treated as a special case of processing sensor data. It can help tremendously having a hierarchical method to filter and aggregate data to higher level events as suggested in [Schmidt, Beigl, and Gellersen 1998]. **Programming by demonstration** [21, 32, 48] (see Chapter 7, Prototyping Mobile Device Applications) is another programming paradigm that can be used on top of a toolkit's features. **Configurability / end-user programming** [9, 24, 33, 49, 53] basically combines two requirements: first, the system must provide support in deploying a system; second, simple visual tools on top of the system must be available to enable users with no experience in programming and little knowledge about technical details to change and configure a system according to their needs. **Multiple users** [2, 4, 27] should normally be able to use a system simultaneously. Besides aspects that touch privacy (e.g. defining access rights) and security (e.g. implementing access rights), the distributed nature and the choice of protocols used for data transfer should support multiple users. The sensor and sensor interpretation layers are responsible for the identification of users.

In Table 10, we now show whether the described requirements and features are supported, partly supported / under development, or not supported in the current version of the EIToolkit and several other related toolkits.

There is an Excel sheet available online that contains this data[46]. It is not particularly surprising that the EIToolkit is found at the top of this list as it was designed taking most of these requirements into account. More value of the table can arguably be derived from the online version. It is possible to specify weights for each of the items and therefore check for the most appropriate toolkit(s) for more specific settings and prioritised requirements. It also contains brief remarks giving an indication about why specific rankings have been given and thus provides pointers into the relevant publication for further information on particular topics.

The table uses three different symbols to indicate the relationship of each toolkit with the given requirement: '■' means the requirement is fulfilled, '▪' means it is only partly fulfilled or can be fulfilled with little effort, and '×' signifies that this particular requirement is not fulfilled. Numerically, we rate the toolkits by assigning values '1', '0', and '-1', respectively and add them. In Table 10, we used weights equal to '1' for all requirements. Thus, since the list comprises 46 items, a project can be rated with a value ranging from -46 to +46. A value of zero therefore indicates that the number of positive and negative entries is the same.

[46] Comparison of current hardware and software toolkits for pervasive application development; Excel sheet available at *http://www.paul-holleis.de/files/diss/toolkits_requirements.xlsx*

Table 10: Categorisation of the EIToolkit and several related projects with regard to the criteria detailed in Section 4.2, Toolkit Requirements for Pervasive Applications. (■ : requirement fulfilled ▪ : requirement only partly fulfilled / could be fulfilled with little effort × : requirement not fulfilled)

	EIToolkit	d.tools [22]	FUSE Platform [25]	ECT Toolkit [18]	Context Toolkit [13]	iROS / iStuff / iCrafter [4]	OPF [52]	Plan B [5]	Calder Toolkit [31]	BOXES [23]	iStuff Mobile [3]	UbiWise [7]	SpeakEasy [43]	DART [35]	Gaia [47]	IrisNet [42]	JSense [48]	Robotic Studio[24]	One.World [19]	VisualRDK [53]	Papier-Mâché [30]	MetaCricket [36]	Interplay [39]	Real World Interfaces [38]
a) Support Hardware, Software and Paradigms																								
Supported Components																								
Control off-the-shelf and proprietary programs	■	▪	▪	▪	▪	■	×	▪	×	■	■	▪	▪	×	▪	▪	▪	×	×	■	▪	×	▪	×
Allow external hardware sensors / actuators / devices	■	■	■	■	■	■	▪	■	■	■	■	▪	■	■	■	■	■	■	▪	■	▪	■	■	■
Support different protocols and devices in general	■	▪	■	▪	▪	▪	▪	■	×	■	▪	▪	■	×	×	×	×	■	×	▪	×	×	▪	×
Support the communication between devices, applications and people	▪	■	■	×	■	▪	▪	▪	■	▪	▪	■	■	×	■	×	×	▪	▪	×	■	×	×	▪
Wired as well as wireless components	■	×	■	■	▪	■	■	■	■	▪	■	▪	▪	▪	▪	▪	■	▪	×	▪	×	■	▪	×
Infrastructural aspects and general properties																								
Support for deploying the system	×	×	▪	▪	■	■	■	■	×	×	×	×	×	×	▪	▪	×	■	×	×	×	■	▪	■
Reusable components	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	▪	■	■	■	■	■	■	×	■
Portability across multiple platforms	■	■	■	■	■	■	■	■	■	■	×	■	■	▪	▪	■	■	■	■	▪	■	▪	■	■
Build 'local' as well as distributed applications	■	■	■	■	■	×	■	■	×	×	×	■	■	×	■	■	■	▪	■	■	×	■	×	▪
Allow high data rates and time critical applications	■	▪	▪	▪	×	▪	×	×	▪	×	×	■	▪	▪	▪	■	×	▪	▪	×	▪	×	▪	×
Operate packet-based as well as stream-based	▪	▪	×	×	×	▪	×	▪	×	▪	▪	×	▪	▪	▪	■	×	×	▪	×	×	×	▪	×
Use standardised but flexible protocols	■	▪	■	■	■	■	▪	■	×	×	■	×	■	×	▪	■	▪	▪	×	■	▪	×	×	×
Inform about appearance and disappearance	▪	■	▪	▪	■	×	■	■	■	▪	×	■	■	▪	▪	▪	▪	×	▪	■	▪	▪	▪	×
Ability to shift or distribute computation load in the system	▪	×	■	×	×	×	■	■	▪	×	×	×	▪	×	▪	■	×	×	×	×	×	▪	×	×
b) Creating Applications																								
Low threshold																								
Ease of use	▪	■	▪	▪	■	▪	▪	×	■	■	■	■	▪	■	▪	×	▪	■	×	▪	▪	▪	×	▪
Simple things must be simple	■	■	▪	▪	▪	■	▪	▪	■	■	■	▪	×	■	▪	×	■	■	▪	×	■	■	■	■
Hide implementation details	▪	■	■	■	■	▪	■	■	■	▪	▪	■	■	■	■	▪	■	■	■	■	■	■	■	■
Abstract and package	▪	×	▪	■	■	▪	▪	×	■	×	▪	×	▪	▪	▪	■	▪	▪	▪	■	■	×	×	×
Separate services from applications	■	■	■	■	▪	▪	■	×	■	■	■	■	■	▪	■	■	■	■	■	■	■	■	■	■
Support or integrate visualisations	■	■	■	■	×	■	▪	▪	▪	■	■	■	×	■	×	×	▪	■	▪	■	■	×	▪	×
Allow additional tools on top	■	▪	▪	■	×	■	■	■	▪	■	■	×	▪	×	▪	▪	■	×	■	■	×	▪	×	■
Scheme to link physical and virtual components	×	■	×	▪	×	×	×	▪	▪	■	■	▪	×	■	×	×	×	×	×	×	▪	▪	×	×
Don't impose a specific architecture / paradigm	▪	×	×	×	▪	×	▪	×	×	×	×	×	▪	×	×	×	▪	×	×	×	×	■	×	×

	EIToolkit	d.tools	FUSE Platform	ECT Toolkit	Context Toolkit	iROS / iStuff / iCrafter	OPF	Plan B	Calder Toolkit	BOXES	iStuff Mobile	UbiWise	SpeakEasy	DART	Gaia	IrisNet	JSense	Robotic Studio	One.World	VisualRDK	Papier-Mâché	MetaCricket	Interplay	Real World Interfaces
High Ceiling																								
Allow complex applications to be built	■	▪	■	■	■	■	▪	■	■	×	▪	■	■	■	■	■	■	■	■	■	■	▪	▪	▪
Support data processing	▪	■	×	▪	■	×	■	×	×	×	×	×	×	×	×	▪	×	×	×	×	▪	×	×	×
Provide solutions to common tasks	■	×	×	▪	▪	×	▪	▪	×	×	×	×	×	■	×	×	×	■	▪	×	×	×	×	×
Support the re-use of fragments	■	■	■	■	■	■	■	■	■	■	■	■	■	■	▪	■	■	■	■	■	■	■	■	■
Ways of development																								
Support hierarchical data processing	■	×	■	▪	■	■	■	×	×	×	■	×	▪	■	■	■	▪	×	■	▪	×	×	▪	×
Allow different programming environments and languages	■	×	■	■	■	■	×	■	■	■	■	▪	▪	×	■	■	▪	■	▪	×	×	×	■	■
Support event-based development	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	×	■	■	■	■	■	■	■	■
Support pulling for data (as opposed to, potentially event-driven, pushing)	▪	×	▪	■	×	▪	■	■	×	×	×	×	■	×	■	■	■	▪	▪	×	×	■	×	▪
Support state-based development	■	■	×	■	×	×	×	×	×	×	×	×	×	×	▪	×	×	×	×	×	×	×	▪	▪
Create on-screen PC programs	■	▪	■	▪	■	▪	■	■	■	■	×	■	■	■	■	■	■	■	■	▪	■	×	■	■
c) Debugging and Changing Applications																								
Observing and debugging																								
Storage, history, logging, and replay	■	■	■	▪	▪	▪	▪	▪	▪	×	▪	■	×	■	×	▪	×	×	▪	×	×	×	×	×
Allow remote control and observation	■	■	■	▪	×	▪	▪	■	×	▪	■	■	▪	×	■	■	×	■	■	×	■	×	▪	×
Simulate components	■	■	▪	×	×	▪	×	■	×	×	×	■	×	■	×	×	■	×	×	■	■	×	×	×
Find errors and issues in the idea, design and architecture of an application	×	■	×	▪	▪	×	×	×	▪	▪	×	■	×	■	×	×	×	×	×	▪	▪	×	×	×
Make errors and exceptions accessible and potentially offer solutions	×	▪	×	×	×	×	×	×	×	▪	×	×	×	×	■	×	×	×	▪	×	×	×	×	×
Provide feedback about what is happening behind the scenes	▪	×	×	×	×	×	×	×	×	×	×	×	×	×	■	×	×	▪	×	×	■	×	×	×
Changing and updating																								
Allow incrementally extending the system	■	■	■	■	■	■	■	■	■	■	■	■	■	▪	■	■	■	■	■	■	×	■	■	■
Support quick alterations	■	■	■	■	■	■	■	×	■	■	■	■	■	■	▪	■	■	▪	■	■	■	■	■	■
Allow easily and quickly exchanging input, output and application logic	■	■	■	■	■	■	■	×	■	■	■	■	■	■	▪	■	▪	▪	■	■	■	■	■	▪
Be stable with respect to changes and errors	■	×	▪	▪	▪	■	▪	▪	▪	▪	■	×	■	×	×	■	×	×	■	■	×	×	▪	×
d) Integrating (into) the Development Process																								
Support the different phases of the application design	▪	■	×	▪	×	▪	×	×	▪	■	▪	▪	×	■	×	×	▪	×	×	×	▪	▪	×	×
Connect to existing prototyping and programming tools	▪	■	▪	×	▪	×	×	×	■	■	▪	×	×	▪	×	×	×	×	×	×	×	×	×	×
Support the use of design tools	×	×	×	×	×	×	×	×	■	■	▪	×	×	■	×	×	×	×	×	×	×	×	×	×
Sum (46 items, each {-1, 0, 1}, i.e. possible range [-46, ... +46])	**23**	**13**	**12**	**11**	**7**	**6**	**6**	**5**	**5**	**5**	**4**	**4**	**4**	**3**	**3**	**2**	**0**	**0**	**-1**	**-1**	**-2**	**-8**	**-9**	**-11**

Table 11: References used to support the list of requirements for prototyping toolkits.

1. **Andersson, J.** "A Deployment System for Pervasive Computing." *ICSM'00.* 2000. 262-270.
2. **Ballagas, R. A.** "Bringing Iterative Design To Ubiquitous Computing: Interaction Techniques, Toolkits, and Evaluation Methods." PhD thesis, RWTH Aachen University, 2007.
3. **Ballagas, R. A., Memon, F., Reiners, R., and Borchers, J.** "iStuff Mobile: Rapidly Prototyping New Mobile Phone Interfaces for Ubiquitous Computing." *CHI'07.* 2007. 1107-1116.
4. **Ballagas, R. A., Ringel, M., Stone, M., and Borchers, J.** "iStuff: a Physical User Interface Toolkit for Ubiquitous Computing Environments." *CHI'03.* 2003. 537-544. See also [Ballagas 2007].
5. **Ballesteros, F. J., Soriano, E., Guardiola, G., and Leal, K.** "Plan B: Using Files Instead of Middleware." *IEEE Pervasive Computing* 6, no. 3. 2007. 58-65.
6. **Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J., and Zukowski, D.** "Challenges: an Application Model for Pervasive Computing." *MobiCom'00.* 2000. 266-274.
7. **Barton, J. J. and Vijayraghavan, V.** "UBIWISE: A Ubiquitous Wireless Infrastructure Simulation Environment." Technical Report HPL-2002-303, Mobile and Media Systems Laboratory, HP Laboratories Palo Alto, 2002.
8. **Becker, C. and Schiele, G.** "Middleware and Application Adaptation Requirements and their Support in Pervasive Computing." *ICDCS'03.* 2003. 98-103.
9. **Beckmann, C. and Dey, A. K.** "SiteView: Tangibly Programming Active Environments with Predictive Visualization." Technical Report IRB-TR-03-025, Intel Research, 2003.
10. **Bohn, J.** "Prototypical Implementation of Location-aware Services Based on a Middleware Architecture for Super-distributed RFID Tag Infrastructures." *Personal Ubiquitous Computing* 12, no. 2. 2008. 155-166.
11. **Borchers, J., Ringel, M., Tyler, J., and Fox, A.** "Stanford Interactive Workspaces: a Framework for Physical and Graphical User Interface Prototyping." *IEEE Personal Communications* 9, no. 6. 2002. 64-69.
12. **Davies, N. and Gellersen, H.-W.** "Beyond Prototypes: Challenges in Deploying Ubiquitous Systems." *IEEE Pervasive Computing* 1, no. 1. 2002. 26-35.
13. **Dey, A. K., Salber, D., and Abowd, G. D.** "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context Aware Applications." '*Context-Toolkit*'. *Human-Computer Interaction* 16. 2001.
14. **Edwards, W. K., Bellotti, V., Dey, A. K., and Newman, M. W.** "Stuck in the Middle: The Challenges of User-centered Design and Evaluation for Infrastructure." *CHI'03.* 2003. 297-304.
15. **Fox, A., Davies, N., de Lara, E., Spasojevic, M., and Griswold, W.** "Real-World Deployments." *IEEE Pervasive Computing* 5, no. 3. 2006. 21-56.
16. **Gilleade, K., G., Sheridan J., and Allanson, J.** "Liquid: Designing a Universal Sensor Interface for Ubiquitous Computing." Technical report Comp-001-2003, Computing Department, Lancaster University, 2003.
17. **Greenberg, S. and Fitchett, C.** "Phidgets: Easy Development of Physical Interfaces Through Physical Widgets." *UIST'01.* 2001. 209-218.
18. **Greenhalgh, H. J., Izadi, S., Mathrick, J., and Taylor., I.** "ECT: a Toolkit to Support Rapid Construction of Ubicomp Environments." *UbiSys'04.* 2004.
19. **Grimm, R.** "One.World: Experiences with a Pervasive Computing Infrastructure." *IEEE Pervasive Computing* 3, no. 3. 2004. 22-30. See also [Grimm, Davis, et al. 2004].
20. **Hacklinger, F.** "Java/A - Taking Components into Java." *IASSE'04.* 2004.
21. **Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S. R.** "Authoring Sensor-based Interactions by Demonstration with Direct Manipulation and Pattern Recognition." *CHI'07.* 2007. 145-154.
22. **Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., et al.** "Reflective Physical Prototyping through Integrated Design, Test, and Analysis." '*d.tools*'. *UIST'06.* 2006. 299-308.
23. **Hudson, S. E. and Mankoff, J.** "Rapid Construction of Functioning Physical Interfaces from Cardboard, Thumbtacks, Tin Foil and Masking Tape." '*BOXES*'. *UIST'06.* 2006. 289-298.
24. **Humble, J., Crabtree, A., Hemmings, T., Akesson, K., Koleva, B, Rodden, T., et al.** "Playing with the Bits: User-configuration of Ubiquitous Domestic Environments." *Ubicomp'03.* 2003. 256-263.
25. **Izadi, S., Coutinho, P., Rodden, T., and Smith, G.** "The FUSE Platform: Supporting Ubiquitous Collaboration Within Diverse Mobile Environments." *Automated Software Engineering* 9, no. 2. 2004. 167-186.
26. **Johanson, B. and Fox, A.** "The Event Heap: a Coordination Infrastructure for Interactive Workspaces." *WMCSA'06.* 2002. 83-93.
27. **Johanson, B., Fox, A., and Winograd, T.** "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms." *IEEE Pervasive Computing* 1, no. 2. 2002.

28. **Johanson, B., Ponnekanti, S., Kiciman, E., Sengupta, C., and Fox, A.** "System Support for Interactive Workspaces." Unpublished Report, http://graphics.stanford.edu/papers/iwork-sosp18/, Stanford Computer Graphics Lab, Stanford University, 2001.
29. **Kindberg, T. and Fox, A.** "System Software For Ubiquitous Computing." *IEEE Pervasive Computing* 1, no. 1. 2002. 70-81.
30. **Klemmer, S. R., Li, J., Lin, J., and Landay, J. A.** "Papier-Mâché: Toolkit Support for Tangible Input." *CHI'04.* 2004. 399-406. See also [Klemmer 2004].
31. **Lee, J. C., Avraham, D., Hudson, S. E., Forlizzi, J., Dietz, P. H., and Leigh, D.** "The Calder Toolkit: Wired and Wireless Components for Rapidly Prototyping Interactive Devices." *DIS'04.* 2004.
32. **Li, Y. and Landay, J. A.** "Into the Wild: Low-cost Ubicomp Prototype Testing." *Computer* 41, no. 6. 2008. 94-97.
33. **Li, Y., Hong, J., and Landay, J. A.** "Topiary: A Tool for Prototyping Location-Enhanced Applications." *UIST'04.* 2004.
34. **Ma, J., Yang, L. T., Apduhan, B. O., Huang, R., Barolli, L., Takizawa, M., et al.** "A Walkthrough from Smart Spaces to Smart Hyperspaces: towards a Smart World with Ubiquitous Intelligence." *ICPADS'05.* 2005. 370-376.
35. **MacIntyre, B., Gandy, M., Dow, S., and Bolter, J. D.** "DART: a Toolkit for Rapid Design Exploration of Augmented Reality Experiences." *UIST'04.* 2004. 197-206.
36. **Martin, F., Mikhak, B., and Silverman, B.** "MetaCricket: a Designer's Kit for Making Computational Devices." *IBM Systems Journal* 39, no. 3-4. 2000. 795-815.
37. **Mayrhofer, R.** "Towards an Open Source Toolkit for Ubiquitous Device Authentication." *PerCom'07.* 2007. 247-254.
38. **McCrickard, D. S., Bussert, D., and Wrighton, D.** "A Toolkit for the Construction of Real World Interfaces." '*Real World Interfaces*'. *ACMSE'03.* 2003. 118-123.
39. **Messer, A., Kunjithapatham, A., Sheshagiri, M., Song, H., Kumar, P., Nguyen, P., et al.** "InterPlay: a Middleware for Seamless Device Integration and Task Orchestration in a Networked Home." *PerCom'06.* 2006.
40. **Michiels, S., Horré, W., Joosen, W., and Verbaeten, P.** "DAViM: a Dynamically Adaptable Virtual Machine for Sensor Networks." *MidSens'06.* 2006. 7-12.
41. **Modahl, M., Agarwalla, B., Saponas, S., Abowd, G., and Ramachandran, U.** "UbiqStack: a Taxonomy for a Ubiquitous Computing Software Stack." *Personal and Ubiquitous Computing* 10, no. 1. 2006. 21−27.
42. **Nath, S., Ke, Y., Gibbons, P. P., Karp, B., and Seshan, S.** "IrisNet: an Architecture for Enabling Sensor-enriched Internet Services." Technical Report IRP-TR-02-10, Intel Research Pittsburgh, 2002.
43. **Newman, M. W., Izadi, S., Edwards, W. K., Sedivy, J. Z., and Smith, T. F.** "User Interfaces When and Where They are Needed: an Infrastructure for Recombinant Computing." '*SpeakEasy*'. *UIST'02.* 2002.
44. **Ng, K. H., Koleva, B., and Benford, S.** "The Iterative Development of a Tangible Pin-board to Symmetrically Link Physical and Digital Documents." *Personal Ubiquitous Computing* 11, no. 3. 2007. 145-155. See also [Villar and Gellersen 2007].
45. **Niemelä, E. and Latvakoski, J.** "Survey of Requirements and Solutions for Ubiquitous Software." *MUM'04.* 2004. 71-78.
46. **Rodden, T. and Benford, S.** "The Evolution of Buildings and Implications for the Design of Ubiquitous Domestic Environments." *CHI'03.* 2003. 9-16.
47. **Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K.** "A Middleware Infrastructure for Active Spaces." '*Gaia*'. *IEEE Pervasive Computing* 1, no. 4. 2002. 74-83.
48. **Santini, S., Adelmann, R., Langheinrich, M., Schätti, G., and Fluck, S.** "JSense - Prototyping Sensor-Based, Location-Aware Applications in Java." *USC'06.* 2006. 300-315.
49. **Sohn, T. and Dey, A. K.** "iCAP: an Informal Tool for Interactive Prototyping of Context Aware Applications." '*Context Toolkit*'. *CHI'03.* 2003. 974-975.
50. **Sohn, T. and Dey, A. K.** "iCAP: Rapid Prototyping of Context Aware Applications." '*Context Toolkit*'. *CHI'04.* 2004. 103-129.
51. **Terada, T., Tsukamoto, M., Hayakawa, K., Kishino, Y., Kashitani, A., and Nishio, S.** "Ubiquitous Chip: a Rule-based I/O Control Device for Ubiquitous Computing." *Pervasive'04.* 2004. 238-253.
52. **Van Kleek, M., Kunze, K., Partridge, K., and Begole, J.** "OPF: A Distributed Context-Sensing Framework for Ubiquitous Computing Environments." *Ubicomp'06.* 2006. 82-97.
53. **Weis, T., Handte, M., Knoll, M., and Becker, C.** "Customizable Pervasive Applications." *PerCom'06.* 2006. 239-244. See also [Weis, Knoll, et al. 2007], '*VisualRDK*'.
54. **Yang, H., Jansen, E., and Helal, S.** "A Comparison of Two Programming Models for Pervasive Computing." *SAINT'06.* 2006. 134-137.

4.3 EIToolkit – Design Decisions

In the last two chapters, we strongly argued for tools that support prototyping applications and either build on or take into account models such as the Keystroke-Level Model. This chapter presents, based on the analysis in the previous section, the design and functionality of the EIToolkit[47] which will serve as the basis of such tools. It has a component-based architecture that allows for easily combining heterogeneous devices and programs. Since the beginning of the toolkit development in 2004, several other research projects have built frameworks and toolsets to ease the creation of tangible or pervasive applications as has been described in the last section.

4.3.1 Envisioned Application Scenarios

The following three scenarios demonstrate the aims of the toolkit from a user's point of view. They show three different ideas, from simple to complex, and ways of implementing these using the EIToolkit.

Angelina's Automatic Music Interrupter (Figure 23)
Angelina built a small hardware board with an attached pressure sensor. The board continuously reads the sensor value and uses a Bluetooth sender to communicate changes in the pressure data. She mounts the sensor on her chair and can thus detect whether she is sitting on the chair or not. Angelina likes to listen to music or an audio book while working (she obviously has an easy job). She now simply wants playback to be stopped whenever she leaves her desk and resumed when she returns without her needing to explicitly execute any actions such as bringing the multimedia player into the foreground and hitting any buttons.

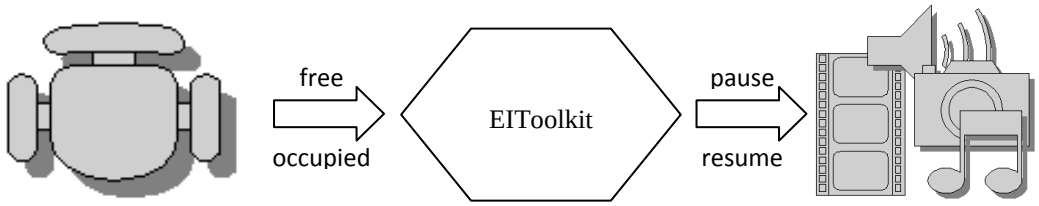


Figure 23: Make simple things simple: music playback is paused while the seat is not occupied.

The sensor board communicates through the serial protocol over Bluetooth. Thus, a toolkit component makes all messages available to the toolkit with no effort. Also, the Winamp multimedia player is directly supported (alternatively, MS Windows multimedia keys can be used to control other applications such as the Windows Media Player). All Angelina needs to do is to create an application that sends an event containing the message 'pause' to the Winamp wrapper whenever the pressure sensor indicates that the person stood up, and a 'resume' event in the opposite case. To simplify this, Angelina can easily select the necessary events and connect them to the appropriate actions in a graphical tool.

Brad's Context Aware System Development (Figure 24)
Brad is developing an application supposed to react proactively to a user's actions without explicit input, i.e. the system uses sensors to get the state of the environment and the user and then deduces appropriate actions. For sensing, he uses infrastructure from his customer that he can access only locally and at specific times (since it is in use by others the rest of the time). They employ specific microcontroller boards that connect to a PC using OSC[48] which is a protocol primarily intended to transmit audio data but has often been employed ti communicate sensor data in general. He therefore records sensor data of one day for later analysis. In his own office, he can replay the sensor data, use components to filter and aggregate data, as well as simulate specific events by hand. Thus, he can develop applications without needing to continuously access the infrastructure.

[47] Embedded Interaction Toolkit, EIToolkit; project page: *http://www.eitoolkit.de*
[48] Open Sound Control, OSC; project page: *http://www.cnmat.berkeley.edu/OpenSoundControl/*

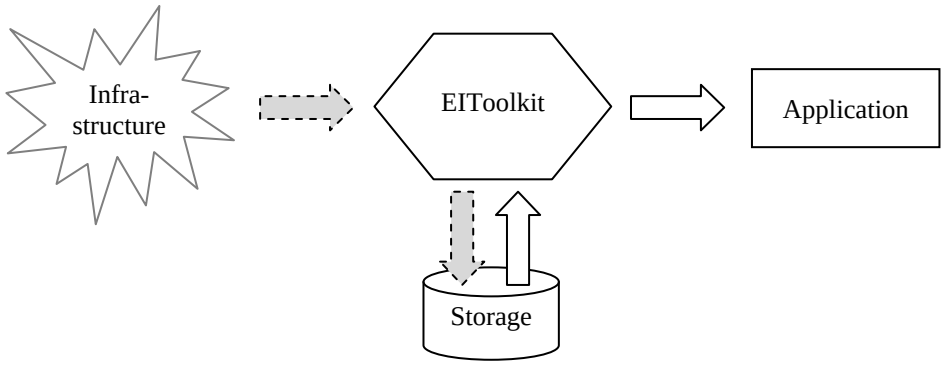


Figure 24: When connected to some external resources, a toolkit component can be used to log all or certain messages. These can later be replayed to further develop an application *offline*.

Brad can use the toolkit's OSC wrapper to connect the sensor data to the EIToolkit. A logging component stores all messages occurring in a specific time interval. If he is not interested in all generated messages, he can easily filter out specific messages. He can then replay these messages any time later. Whenever he needs to test a specific scenario that he has not yet recorded, he can use a further component to generate and simulate custom messages and events. For his project, he writes simple transducers that filter (e.g. only temperature values larger than some value), enhance (e.g. apply a running average), or combine data (e.g. switching the light off and closing the door indicates leaving the room).

Celine's Ideas for Ambient Email Visualisation (Figure 25)
Celine wants to visualise the amount of new emails. Ultimately, she wants to use her small water spring as ambient device, i.e. more water means more emails. During development, she does not yet have control over the well and uses a Flash software visualisation. She also plans to potentially replace the spring later with another device like a light to avoid her office being flooded coming back from a holiday without reading emails.

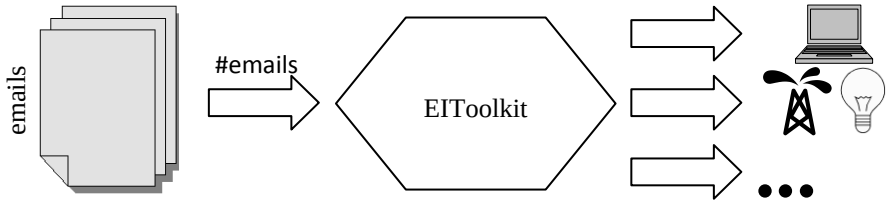


Figure 25: The design of the toolkit makes it easy to use several visualisations at the same time or exchange them during runtime.

Celine writes software to retrieve the number of unread emails and sends it to the toolkit whenever a change occurs. She uses the component that makes toolkit data available on a TCP port to access it from Flash and writes a visualisation. The moment she finishes the control of the spring and connects it to the toolkit, her envisioned application is working. Software and hardware visualisation are running at the same time. She can even remove existing or add further visualisations at any time without the need to change or restart anything.

4.3.2 Requirements Identification

Before and during the design of the EIToolkit, we collected requirements based on the list presented in the previous section that it should fulfil. Since the resources available for the implementation were restricted, some limitations had to be accepted as well.

To enable the toolkit to support a number of projects over the years, we strongly built on a component-based approach. Thus, we were able to add features that we found out to be important later and enlarge the support for additional components that emerged over time. The architecture ensures backward compatibility to existing applications.

The following list shows the most important **requirements, features, and properties** the toolkit is built upon, based on scenarios such as those presented in the last section.

- **Component-based architecture**: besides a central code base for core functionality, most code should be encapsulated into smaller components. These components should be separated in a way that they can be developed independently and be easily exchanged.
- **Thin connection layers**: it should only be required to write a very small additional layer of software to connect the toolkit to specific hardware and software.
- **Programming abstractions**: writing an application that builds on various components should mean not more than combining these components. No additional knowledge about the concrete implementation or (proprietary) communication protocols should be necessary.
- **Open and replaceable main protocol**: communication should be based on a well-known, open, widely used communication protocol such as UDP or TCP/IP. This should be exchangeable and other forms of communication should be possible.
- **Support of extension protocols**: further protocols such as the Real-time Transport Protocol[49] (RTP), OSC, or MIDI should be supported to enable cooperation with other applications and especially streaming of data.
- **Logging**: to facilitate development and debugging, it should be possible to log all data to screen and into a file in a simple but comprehensive format. Filters should be available to reduce the amount of logged data.
- **Simulation and replay**: to emulate devices that are not available or to simulate the output of components in specific events, it should be possible to generate messages and replay information using stored logging data.
- **Interface descriptions**: in order to get more information about a component, components can optionally describe their interface for automatic detection, connection, and integration. This should be done in a comprehensible and standardised format.
- **Simple component replacement**: it should be possible to easily replace components with the same or similar functionality with each other. This ensures, for example, that components can be updated and prototyping can be done in software before replacing it with a hardware device. Interface information can potentially be used to automatically exchange components according to their interface (i.e. if they inherit the same interface), see for example our approach in [Holleis and Schmidt 2005]. It should also be transparent for an application developer whether a wrapper around some software or a hardware component is used.
- **Layered context processing**: it should be possible to apply sensor fusion and filtering techniques (in separate components and several layers), see for example the concept of cues described in [Schmidt, Beigl, and Gellersen 1998].
- **Easy hardware extension**: several hardware platforms should directly be supported (such as Particle Computers). Others should be connected using specific protocols such as serial connections or OSC.
- **Independence**: the implementation of the toolkit should support various platforms, operating systems, and programming languages as well as paradigms.

To keep the toolkit manageable, easy and quick to use, and to keep the implementation effort reasonable, the following **restrictions** have been specified in advance:

- **No direct high-level guarantees**: there will be no direct guarantees or control on quality of service, reliability, or speed. This makes the system much easier to use in the majority of cases where such direct control is not necessary. Often, such properties can be added later and in the appropriate components, when desired, see for example [Mayrhofer 2007] for an approach regarding authentication. There will also be no inherent support to ensure security and privacy of communication. Measures such as encryption, making data anonymous or filtering data can be implemented by each or special components.
- **Restricted evaluation support**: there will be no direct support for finding errors and issues in the idea, design and architecture of an application. Although tools for debugging, logging, and simulating will be provided, most core toolkit elements are on a lower level than necessary for that purpose. However, components on top of it can be developed, e.g. the more high-level development tools described in Section 5.2 about graphical application development or the user models developed in following sections.

[49] Real-time Transport Protocol, RTP, RFC 3550; definition of the standard: *http://tools.ietf.org/html/rfc3550*

- **Simple base protocol**: there will be a simple and human-readable (but potentially inefficient and restricted) protocol for passing messages. However this can be exchanged for more sophisticated protocols later.
- **Simple interface descriptions**: the expressive power of descriptions of the capabilities of components will be simple and not necessarily based on more complex standards. A transition to standard interface description languages (IDLs) is planned for a later increment in the development process. Further types of component descriptions, i.e. a 3D model of physical devices such as the orientation of accelerometer sensors, will not be implemented immediately but will already be integrated in the design.
- **No direct support for deployment**: there will be no direct support for deploying the system. Currently, the idea is to route most communication traffic through a central point or restrict it to a common subnet. Even if, for example, two mobile devices communicate only with each other and do not need any intermediary, this path is taken. It remains future work to enable the deployment of such applications which are completely separated from the toolkit's infrastructure. The decision was taken since developing, debugging, distributing software and especially collaboratively developing it is considerably easier this way.
- **Focus on stand-alone tools**: even though integration in the design process of pervasive applications is a main goal and we show how to integrate the toolkit with other projects and development environments (see Sections 5.3.2 and 5.3.3), the integration into design tools like Adobe Director is not an element of the core functionality but can be added later with appropriate extensions.

The following sections provide insight into how these requirements have been implemented in the design and architecture of the system.

4.4 Architecture and Implementation

The basic idea of the EIToolkit is to represent resources by proxy components (called 'stubs') in the system. A resource in this sense can be anything that takes some input or provides some output. This includes software from small pieces like simple visualisations to complex applications like a spreadsheet program. It also explicitly allows hardware components like sensors connected to a microcontroller, displays of various types, and compound devices like PCs, cars, or infrastructure in general. Figure 26 illustrates the architecture.

The role of stubs is close to that of a proxy or driver communicating with a specific device. They encapsulate all the information needed to communicate with the resource they represent. On the toolkit side, all stubs pass and receive messages through a common, specific protocol. This design ensures that components are separated from each other, thus being easily exchangeable and allowing applications to be as independent from the implementation of specific components as possible. Of course, application developers need to know about the semantics and at least some details about how a resource works. However, the toolkit relieves the developer from having to learn, know, and implement specific details for any resources.

It should be noted at this point that introducing new resources to the toolkit does not necessarily imply that a new stub has to be written. As detailed in Section 6.2 about technology enabling applications, there exist stubs that can be used to connect all possible applications based on a specific technology. One stub for example can read from serial line, i.e. a COM port. Any device based on, e.g., the Smart-Its microcontroller platform [Gellersen, Kortuem, et al. 2004] which is connected to a PC with a serial cable can thus communicate with the toolkit. This stub at the same time also enables communication with mobile devices that use the serial line over Bluetooth protocol to exchange data. Another example is the control of MS Windows applications. A stub can emulate key presses on the Windows operating system and thus prototypically control nearly all running applications. Nevertheless, it is sometimes advantageous to implement a specific stub for a specific application or device. This can then support all the features of the resource in the best possible way and run without side effects to other applications (as incurred by, e.g., the stub that generates keystrokes). In fact, there is no reason why these general and specific stubs should not be available at the same time.

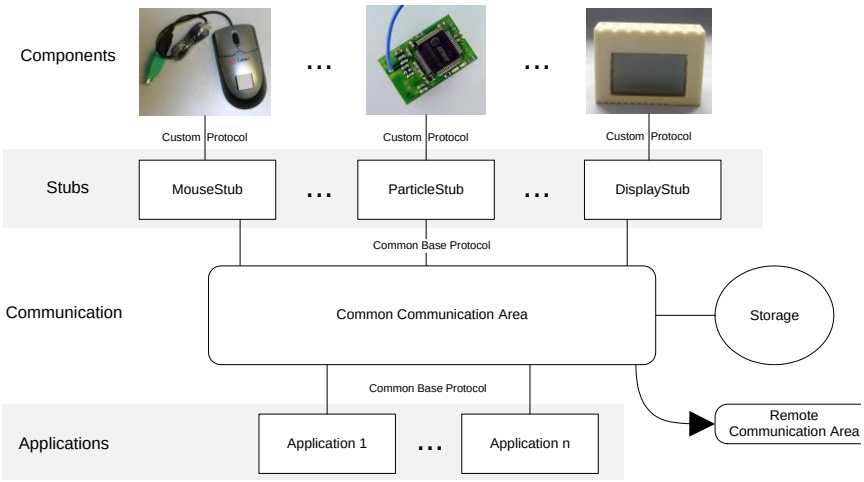


Figure 26: Overall architecture and idea of the EIToolkit system. Components are connected to a common communication area using small and lightweight stubs. Applications can then connect the data from different components simply by filtering, changing and relaying messages from one set of component to another. Messages can be stored, retrieved, and exchanged between remote sites.

Applications can then use the data sent through the common communication area and potentially query the interfaces provided by these stubs. Their core functionality is then to detect events that are important to them, process them, and generate commands to be executed by other stubs. They implement the desired semantics using the input and output capabilities provided through the available stubs without the need to know about implementation specific details of these stubs.

One potential drawback of this approach can be that a number of programs have to be started separately. Often, one stub has to be started for generating events, one for executing commands, and one application component connecting those. Besides the effort to start these, most stubs provide some sort of graphical user interface or at least feedback, e.g., through a console window. These are important for debugging and observation but may clutter the application's user interface. We suggest having, if possible, a central component that acts as a command centre of available stubs and applications. Through it, stubs can be initiated and possibly brought into a window-less state where no visual feedback is provided. If necessary, their interfaces can be revoked.

One more automated approach that we implemented is a component that maintains a user-generated mapping of applications to stubs. In that way, whenever an application is started, the corresponding stubs are initiated automatically. A simple garbage collector type of daemon is then also able to shutdown stubs that are no longer needed. This means, however, that there is a central point in the infrastructure that has global knowledge of all available applications, all running stubs, and their desired connections. Besides some technical issues, this implies that the user of this system needs to maintain and update some configuration. We show possible approaches to end-user adaptation and programming later in this chapter. A slightly simpler idea is to have all messages specify their target or each stub by sending a specific message at the beginning to indicate the desired target. However, this complicates writing generic stubs and limits the runtime re-configurability of the system. In [Holleis and Schmidt 2005], we propose a hierarchical system of real world objects. Together with the interface descriptions that stubs can provide, this would at least allow replacing objects if they represent an equal or a specialisation of a required interface.

Application Implementation

As an example, consider Angelina's scenario above (page 84). A simple pressure sensor is mounted on her chair, connected to a controller board that sends events via Bluetooth using serial line communication. The SerialStub receives these events and forwards them to the EIToolkit. On the output side, the WinampStub is running which translates EIToolkit messages to control events for the Winamp music player.

The only part that needs to be written is an application listening to events from the sensor and producing corresponding commands for the music player. In this simple setting, this merely means to send a ‘pause’ command whenever the sensor board sends a new event (independently whether it is a ‘pressure:on’ or ‘pressure:off” event). This setting is illustrated in Figure 27. If the sensor board sent an analogue value for the pressure sensor, the application would need to additionally add a threshold.

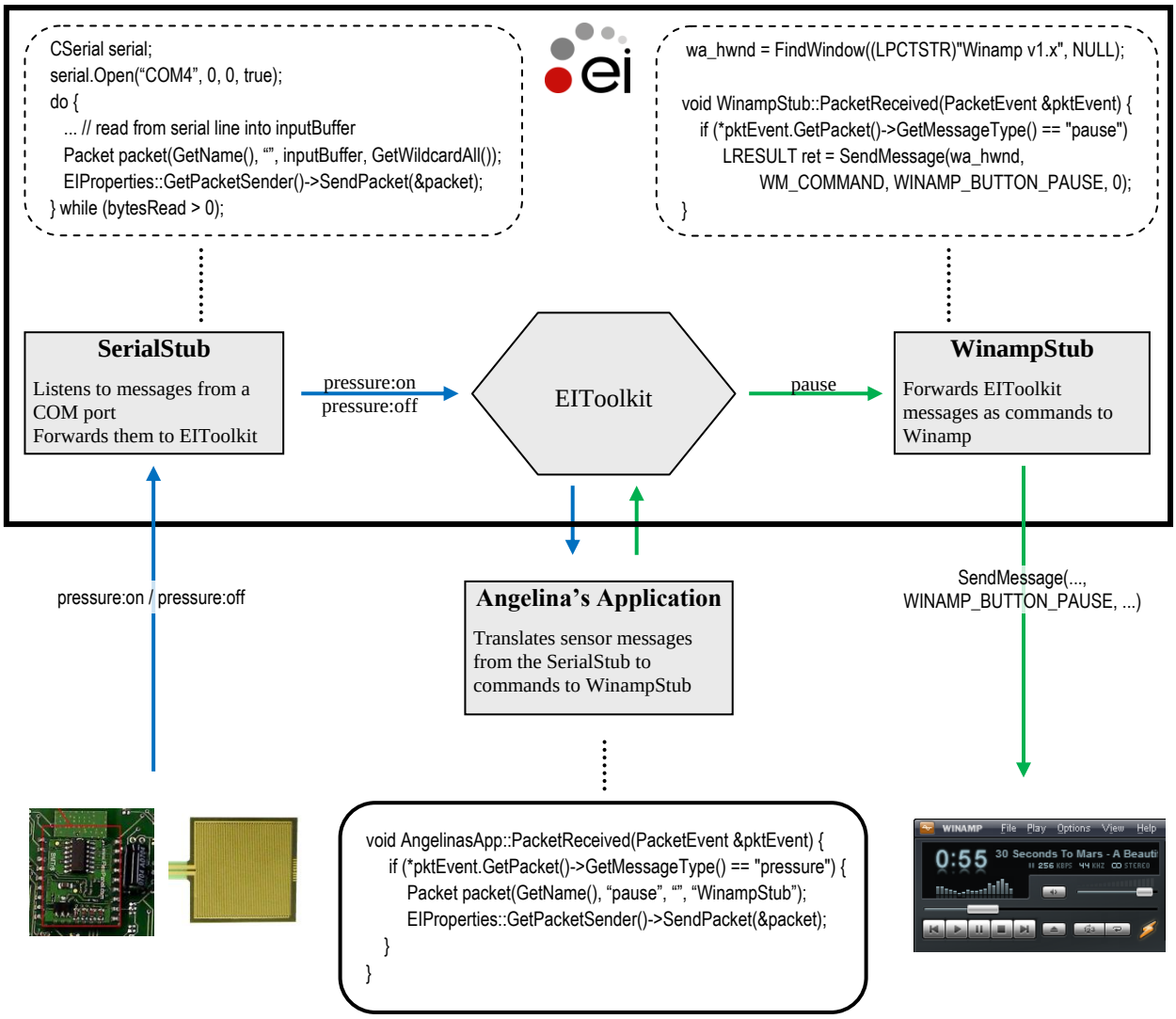


Figure 27: Illustration of the implementation of Angelina’s scenario described on page 84. Only the application at the bottom of the figure needs to be implemented. The rounded boxes show the entire custom code necessary. Alternatively, the graphical tools shown in Section 5.2 could have been used.

The main structure builds on four parts supporting the quick development through a set of libraries and other utility classes:

- **Creating stubs**: templates and wizards (i.e. scripts) exist to create stubs in several programming languages; basic functionality is provided and additional settings (such as whether the stubs send and / or receive messages) can be configured.
- **Communicating with components**: libraries exist that simplify and standardise the use of communication listeners and senders in general
- **Enabling specific technologies**: libraries exist that encapsulate the code necessary to communicate with specific technologies (such as specific stubs or applications, or using specific protocols)
- **Expanding the use**: there exists a library collecting available stubs and applications; even though this has not yet been thoroughly implemented, it is thought to have this part connected to an (online) database that can dynamically use components contributed from various sources.

Architecture Implementation

Table 12 gives an overview of the most important interfaces and classes in conjunction with their most prominent methods. A detailed description can be found in the online documentation and in the source code. This brief list shall only introduce a few terms and give an idea of some of the architectural aspects. One main consideration was to keep the core parts clean and simple to ease its understanding and use by others.

Table 12: Main classes and interfaces of the EIToolkit system with their most important methods.

Classes and Methods	Function
Stub *Start, Stop, IsRunning, GetName*	Stubs connect the toolkit and external devices or applications. Most functionality is specific to the application.
Application *Start, Stop, IsRunning, GetName*	Applications specify the components that communicate between the different stubs and implement the application logics.
Packet *Get- / SetSenderID, -ReceiverID, -MessageType, -Message* PacketEvent *Get- / SetPacket, -Timestamp*	A Packet is the smallest unit of information. It contains information about the sender, the desired receiver as well as message content, and a type identifier describing the message. A PacketEvent adds further information like a timestamp when the packet has been sent / received.
PacketFormatter *WritePacket, ParsePacket* *GetWildcardAll*	A PacketFormatter can serialise and parse serialised messages. The class SimplePacketFormatter is a reference implementation using a simple colon separated string for a packet's elements.
PacketSender *AddPacketSenderImpl, SendPacket* PacketSenderImpl *SendPacket*	The PacketSender manages a list of PacketSenderImpl classes which are responsible for sending messages using a specific protocol. An implementation of a PacketSenderImpl is the UDPClient.
PacketObserver *PacketReceived* PacketReceiver *AddPacketObserver, NotifyObservers* *AddPacketReceiverImpl* PacketReceiverImpl *SetPacketReceiver, NotifyPacketReceiver*	Stubs and applications implement the interface PacketObserver if they need to receive messages and are registered at the PacketReceiver class. A PacketReceiver manages a list of PacketReceiverImpl classes which are responsible for receiving messages using a specific protocol. A reference implementation of a PacketReceiverImpl is the UDPServer.
EIProperties *UseStandards, StopStandards* *Get- / SetPacketFormatter,* *-PacketReceiver, -PacketSender*	The EIProperties class acts as a central access point for toolkit components. It offers access to the PacketReceiver, PacketSender, and PacketFormatter classes (implemented as singletons).

In the following, we briefly describe three aspects of the system that are important for later sections and offer some insight into restrictions and power of the current implementation.

Sending and Receiving Data

By default, all data sent and received in the toolkit is in form of packets. This decision was made since it maps directly to many standard packet-based communication protocols such as UDP. In addition, it suits well to bundle data additional to the payload useful for routing, early filtering, etc. Nevertheless, continuously sending data is also important in many applications. Therefore, we included support for streaming of data such as sensor output and video transmission (as explained below). Since packed-based protocols are not suitable for such data communication, we employ separate channels and use the standard protocol to exchange information about this separate channel.

Packets accordingly contain information about its sender, a characterisation of the type of the message, the message itself, and, optionally, the desired receiver. A message type can distinguish descriptive control messages from those containing actual content or it can further describe the content. Specifying an explicit receiver is treated as a suggestion that can speed up processing. A component thus can choose to process only those packets

sent especially to this component. Since such a mechanism would anyway not be enough to ensure a secure connection between exactly two parties, a component is also explicitly allowed to also listen and accept packets sent to others or to broadcast to all entities. This allows applications to be built that observe the communication of other components, e.g. a general logging and replaying application.

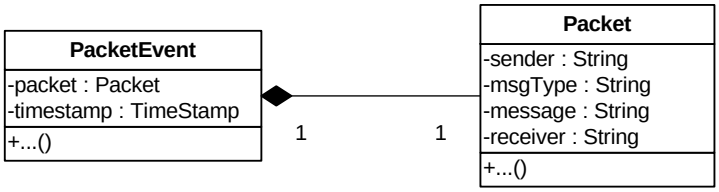


Figure 28: UML diagram of the PacketEvent and Packet classes. A packet encapsulates information about who sends what to whom. Additionally, the message can be described with a message type field. A PacketEvent holds a timestamp of the packet.

Implementation Details
Whenever a packet is sent or received, a PacketEvent is built that additionally includes a time stamp recording the time when it was sent or received, respectively, see Figure 28. This can be used by time-critical applications.

Following the delegation pattern, a singleton instance of a general PacketSender class is used within the toolkit to send packets, see Figure 29. Its implementation is designed to keep the actual way of sending (protocol, etc.) as independent as possible from the caller's implementation. The central properties mechanism registers one or several instances of the interface PacketSenderImpl at the PacketSender. If another component running in the toolkit needs an additional sender implementation, it can add it during runtime. The standard implementation uses a UDPClient for broadcasting packets over a specific UDP port. Other implementations can, e.g., use a dedicated TCP connection.

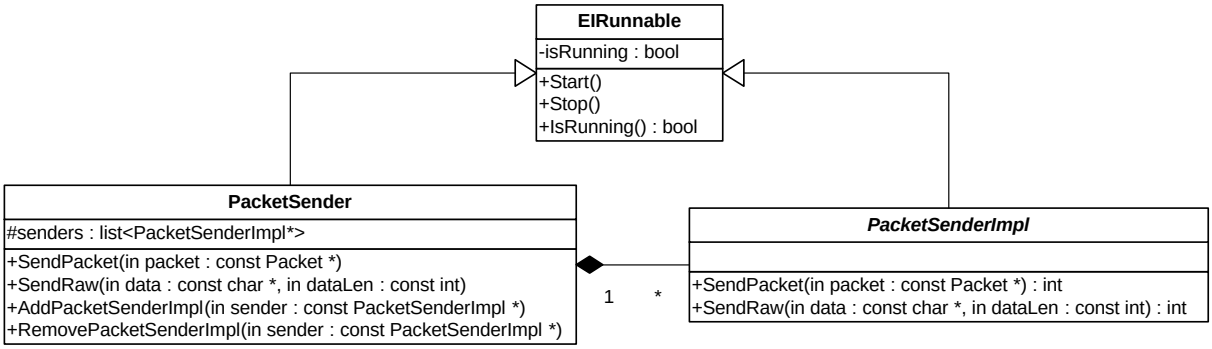


Figure 29: UML diagram of the classes responsible to send packets. PacketSender is a singleton that delegates actual sending of data to instances of PacketSenderImpl.

The receiving side is implemented in a similar way, see Figure 30. A singleton PacketReceiver manages two sets. One consists of one or more classes that implement the interface PacketReceiverImpl. These are responsible to listen to messages sent using a specific protocol (e.g. most probably generated by one of the PacketSenderImpl classes). The standard implementation listens to UDP packets at a specific port. The second list managed by PacketReceiver is a set of observers (interface PacketObserver) that are notified when one of the PacketReceiverImpl classes receives a packet. The PacketReceiver itself implements PacketObserver and is registered as the only observer in all receiver implementations, i.e. all receiver implementations notify this class.

In this way, only the two classes PacketSender and PacketReceiver are used by stubs or applications. All other details are hidden behind this concept. The internals of which protocols are used can be exchanged or adapted without any influence with respect to the users' point of view.

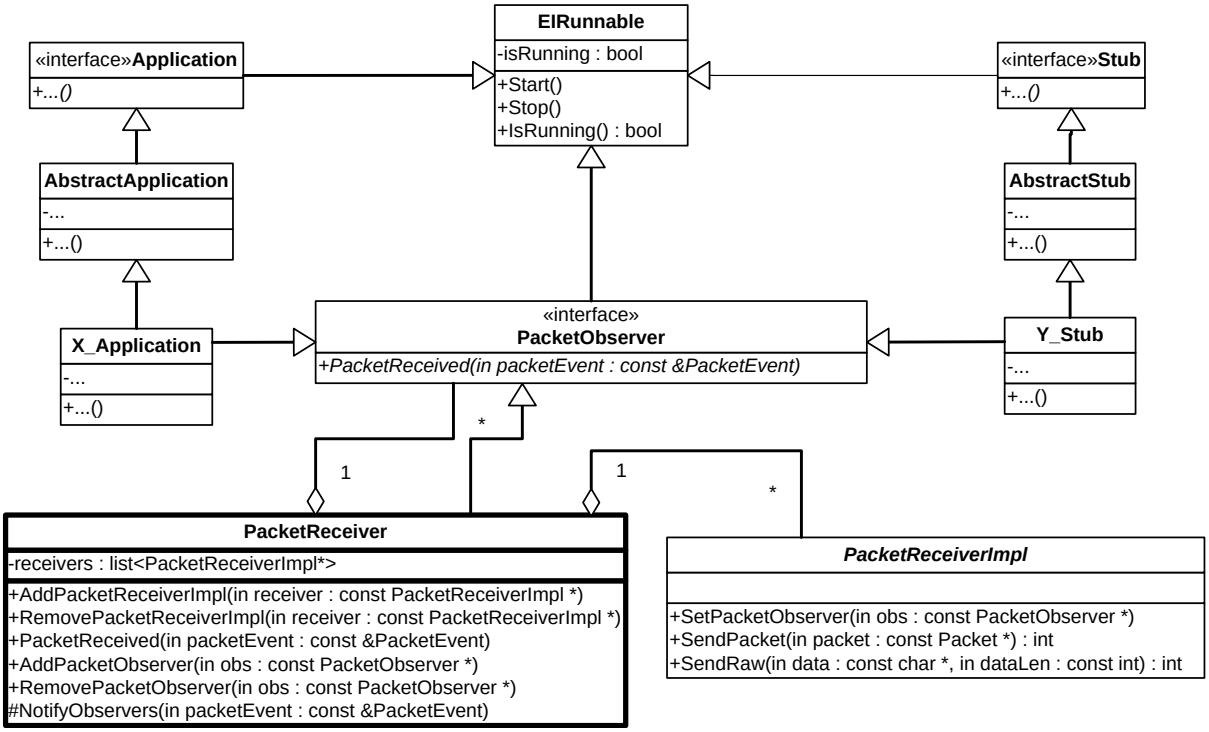


Figure 30: UML diagram of one part of the EIToolkit architecture. The central component in this diagram is the PacketReceiver class. It uses instances of PacketReceiverImpl to delegate packet reception and notifies all registered PacketObserver instances (such as stubs or applications) of incoming packets.

The connection-less type of communication ensures that any party can send data without waiting for any subscriber to appear. On the other hand, in the current implementation, applications cannot access messages that have been sent before they registered themselves. One way is to use the storage mechanism of the toolkit which saves messages to a file. However, the replaying mechanism resends all messages. This can generate undesired results if other applications also use these messages. A better solution is to – as done for example in [Ballagas, Ringel, et al. 2003] – attach a tuplespace implementation. It stores data packets and allows applications to query for specific packets, read, and remove packets from the space. We are currently adding such functionality to the toolkit which is sometimes necessary, for example, to enable services that become unavailable for a certain period to catch up with information that it missed during that time.

Streaming

The packet based messaging system is not ideal for streaming a lot of data in real time. For such purposes, the messaging service can be used to make information available about where such streaming data is available.

One implementation we made available is based on RTP. It uses the open implementation called JRTPLIB[50]. To setup a stream and read it, controlled by the messaging system, you only need to use the two stubs RTPStreamerStub and RTPListenStub. In the following, the C++ implementation is briefly described:

- RTPStreamInfoEI: It encapsulates all information necessary to describe a stream (except its ID). If you want to provide a stream, e.g., you have to call SetDestinationIp and SetDestinationPort.
- RTPStreamerStub: This is a stub used to provide information about streaming data. The stream is identified by a unique ID that must be specified. The stub offers information about its stream with a specific message (SendRTPInfo()). Parameters are specified through the RTPStreamInfoEI class described above.
- RTPListenStub: This is a stub used to receive streaming data from a stream with a specific ID. The stub gets the information about the stream by requesting it (SendStreamRequest()) and waiting for the stream information message which is parsed into an RTPStreamInfoEI instance described above.

[50]Java RTP implementation, JRTPLIB; project page: *http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Jrtplib*

Component Descriptions

One of the goals of the toolkit is the simple use of its components. An approach taken to simplify the automatic and manual use of the toolkit's stubs is to allow each stub to provide information about itself. This information is split into two parts, interface description and state description.

Interface description: All capabilities of stubs fall into two categories. First, everything that the stub produces (e.g. if a stub encapsulates the keyboard, it might send a message whenever a key is pressed), is called an *event* sent by the stub. Second, a stub offers input in the sense that other components in the toolkit can send a command to it which the stub then executes. These inputs are called *operations*. An interface description thus incorporates a set of events and a set of operations. Concerning the implementation, the descriptions of an event and an operation are both encapsulated into a class to keep the description reusable and extensible. To keep the initial system simple, the name of the event or operation is used. It can later be expanded to use interface description languages (IDLs) such as CORBA[51].

Each stub can choose to inherit from DescribingPacketObserver instead of the standard PacketObserver interface. This declares that the stub offers the opportunity to be queried for its interface. This query – as is its answer – is passed using the same packet communication mechanism as always.

Allowing to transmit information about the interface of a stub enables applications that help a developer to implement an application. As shown in Section 5.2, tools can be written that make use of the toolkit and present the user with visualisation of available stubs and their capabilities. One simple application example is to let some outputs (events) be directly and graphically connected to some inputs (operations). In this fashion, simple applications like the one presented in Angelina's scenario on page 84 (pause playback when someone leaves the desk) can be built very quickly and efficiently.

Additionally, automatic documentation can be built and the replacement of components can be supported. The latter means that there can be a hierarchy of interfaces and thus it can automatically be deduced whether a component can be replaced by another one (or by several). Although this approach has not yet been completely exploited, it is clear that it offers vast possibilities. We published some ideas in that direction in [Holleis and Schmidt 2005].

State description: Besides the capabilities (interface) of an object, its current state is of interest as well. As will be outlined later in Sections 5.2.1 and 7.2, one way to think of software development is to bring devices and applications into a certain state at a specific time or situation. A state in this sense is the set of values of all external, i.e. visible, and internal parts of a system that are potentially changed during runtime. This includes external components such as switches, displays, or lights with possible states being simply on / off, or including parameters such as intensity or frequency. Internally, there are the states of variables and physical as well as volatile data storage. As has been argued in, e.g., [Thimbleby and Gow 2007], from a user's point of view, only the external components are visible and thus count for the user's mental model of the system. However, it is up to the implementer of a stub to decide whether to include internal states in the description or not.

The system to store a state is currently kept rather straightforward and is subject to enhancements. For each variable that contributes to the state of the whole stub, a StateDesc object is created which contains a name and a description of the range of possible values in a standard mathematical form, e.g., limited ranges such as '{on, off}' or '[0, 1, ..., 255]', or unlimited ranges such as '[0, ... ['. A stub is then associated with an object of type CombinedStateDesc which combines these single states by creating the cross product of all these, denoted by 'P{stateDesc1, statedesc2, ...}'. This technique has some limitations. The inability to express restrictions such as when the possible values of one variable depends on the value of another one. However, it is able to express a superset of the possible states. Users are required to have some knowledge about the stub to be able to interpret and make use of that information. They will then be able to choose between those states of interest, e.g., to use them only if the system is powered on. The main feature of such StateDesc and CombinedStateDesc classes is that they can be compared with each other and that wildcards can be used in this process to denote currently unimportant variables (such as internal ones).

[51] Common Object Request Broker Architecture, CORBA, project page: *http://www.corba.org*

Portability

For a toolkit like the EIToolkit, it is especially important that it can run on different types of machines as well as operating systems and that it can also be used in conjunction with many programming languages. The use of cross-platform languages like .NET and Java helps achieving this goal. In addition, the separation of applications into several stubs that communicate using standard protocols removes the necessity that all components are written in the same language.

Furthermore, the toolkit's core parts have been implemented natively in three different programming languages namely C++, Java, and C#. There exists technology like SWIG[52] that can be used to make implementations written in one programming language available to another one. Additionally, one of the key features of the .NET technology is the possibility to wrap and reuse code written in different languages. This can still be used with the given implementation. However, providing software in more than one language has several advantages:

- **Language features**: the unique features and strengths of each language can be exploited.
- **Language support**: there might exist better support for certain devices, protocols, or hardware in some programming languages. Java, e.g., does not directly support serial line communication for the Microsoft Windows operating systems any more.
- **Broader code basis**: often, users of implemented functionality want to use external libraries. The selection of possible components increases which also means that, if such external code does not work as expected, it can be replaced more easily.
- **Lower threshold**: for users with deep knowledge in only one programming language, it is much easier to accept, understand, and use code when it is written in that particular language. People are more likely to find and to try pieces of code if it is written in their favourite language.
- **Simplified debugging**: the possibility to debug code it simplified if all elements are written in the same language and do not need any intermediaries or are wrapped in some libraries. Also, the same development environments can be used.

Disadvantages of the pursued approach mainly include some overhead for system developers since they have to write code several times. However, there is always the fallback solution of using implementations in a different language. Also, it can introduce different ways of thinking and implementing specific tasks which can help to come up with solutions that are more efficient, more robust, less error prone, etc.

As noted before, the decision to separate components and use, e.g., UDP to communicate between them allows employing any programming language capable of using this protocol. For all those languages or platforms that do not allow direct access to this protocol, wrappers can be written to enable them to use their specific capabilities. Flash for example does not currently provide means to send or receive data directly via UDP. However, as do most languages, it supports access to TCP/IP. A simple TCP-to-UDP 'converter' that connects to a Flash application using TCP and reroutes the sent information to the toolkit's UDP port and vice versa enables Flash programs to use all functionality of the EIToolkit and its connected components. We used this approach for example in [Holleis, Kern, and Schmidt 2007] where an interactive Flash visualisation shows the interior user interface of a car communicates with a Keystroke-Level Modelling component through the EIToolkit.

[52] Simplified Wrapper and Interface Generator, SWIG; project page: *http://www.swig.org*

5 Prototyping Using EIToolkit and User Models

This chapter introduces ways of prototyping using the EIToolkit described in the last chapter. The main focus is on generally lowering the threshold of creating prototypes and at the same time taking into account usability metrics like the time a user needs to complete a task.

After having introduced the EIToolkit and various advantages of user models for developing applications in the last chapters, we now bring these concepts into practise. First, we introduce the concept of a state graph and its potentials for usability purposes (5.1). Next, we illustrate some graphical approaches to development using state-graphs and the EIToolkit as underlying system (5.2). After that, explicit support for the combination of user models and prototyping tools is examined, concentrating on the demonstration of how to integrate our approach into existing tools (5.3). This also demonstrates the possibility of state-based development and sets the basis of the MAKEIT development environment for mobile devices described in the following chapter.

5.1 Describing Application Semantics with State Graphs

Applications and their behaviour can be described in many ways. For the purposes at hand, it proves suitable to use graph theory in which a state graph represents application logic. Simply put, a state graph is a description of the possible states of an application and the transitions that bring the application from one state into another. Since the notion of such graphs bases on a formal framework, we briefly introduce a few graph theoretical terms.

5.1.1 Graph Theoretical Foundations

Whenever we use terms and concepts from graph theory, we refer to a fairly standard set of definitions that can be found in many standard textbooks such as [Aho, Hopcroft, and Ullman 1983]. However, in order to avoid possible misunderstandings, we define the most important terms in the appendix on page 175. This includes terms like neighbourhood, paths, and connected components.

In general, we write $G = (S, A)$ to refer to a graph G consisting of a set of nodes S and edges A. Additionally, a widely used convention is to have a function $l_S: S \rightarrow L_S$ associating labels from a set of labels L_S to each node. In analogy to that, edges can be labelled with a function $l_A: A \rightarrow L_A$. For our purposes, this is not enough, however, and we map an entire hierarchy of attributes to each graph element (node and edge), see Definition 1. It is beneficial to have the attributes in form of a hierarchy since it maps well to a tree-based data structure. It can also be used to group attributes with similar function, e.g. graphic attributes for drawing, or semantic attributes for specific purposes. In our scenario, nodes will contain information about the state of the application they represent and edges contain information about the actions that lead from one state to another. The labelling functions are implemented as an attribute mapping.

Definition 1 (Attribute Mapping)
In a graph $G = (S, A)$ there is an **attribute mapping** function $att: (S \cup A) \times String \rightarrow Object$ that associates a graph element $e \in S \cup A$ with an arbitrary piece of data. The object is further described by a string which can be used to implement a hierarchy of attributes.
As a special case, if the meaning of an edge attribute is the cost of traversing it (e.g. its length), then $att(a, X)$ can be written as $||a||_X$ or, if it is clear what attribute identifier X is meant, $||a||$ for an edge $a \in A$.

An important notion in graph theory is a path connecting two or more nodes. In general, a *shortest* path in a graph is the path between two nodes that uses the smallest possible number of edges. For a graph whose edges are mapped to attributes, the term 'shortest' can be intended otherwise. If, for example, a graph describes a network of roads and each edge has its length as attribute (often called its 'weight'), a shortest path should be the one with the smallest sum of lengths, which is not necessarily the one with the smallest number of edges. To distinguish between those cases, we define a specialisation of a shortest path in the following Definition 2.

Definition 2 (Length of a Path and Shortest Path with Respect to (Attribute) X)
A path $p(s_a, s_b)$ in a graph $G = (S, A)$, $s_a, s_b \in S$ which uses edges $a_0, a_1, ..., a_n$ is a **shortest path with respect to (attribute) X**, if and only if

- $\forall a \in A: \exists o_i \in Object: att(a, X) = o_i$, i.e. X describes an attribute for edges and each edge has such an attribute associated with it,
- $\{att(a, X) \mid a \in A\} = M$ forms at least a magma, i.e. has a closed binary relation $+: M \times M \rightarrow M$ (i.e. 'attributes can be summed up'),
- $\{att(a, X) \mid a \in A\}$ has a total order $\leq$ (i.e. 'all attributes can be compared with a $\leq$ relation'), and
- for every path $p' \in \{p(s_a, s_b)\}$ which uses edges $a'_0, a'_1, ..., a'_n$ it holds that $att(a_0, X) + att(a_1, X) + ... + att(a_n, X) \leq att(a'_0, X) + att(a'_1, X) + ... + att(a'_n, X)$.

The **length of a path with respect to (attribute) X** then is $||p(s_a, s_b)||_X = att(a_0, X) + att(a_1, X) + ... + att(a_n, X) = ||a_0||_X + ||a_1||_X + ... + ||a_n||_X$. If it is clear to which attribute is referred, we simply write $||p(s_a, s_b)||$.

It follows that $\forall a \in A : att(a, X) = 1 \Rightarrow ||p(s_a, s_b)||_X = |p(s_a, s_b)|$.

The second condition in the definition of the shortest path ensures that sums of elements as used in the fourth condition can also be compared with other sums or elements using the relation from the third condition. Examples for shortest paths include all graphs that define a weight function for their edges like distance or cost; one such class of graphs describes network flow where each edge is assigned a certain capacity. We will focus on examples where the weight describes the amount of time necessary to traverse an edge.

5.1.2 Definition of the State Graph

The following Definition 3 defines a state graph as a structure to describe the behaviour of an application.

Definition 3 (State Graph, Transition, Action-ID Function)
A **state graph** is a directed graph $G = (S, A)$. The states of the application that is currently designed represent the set of nodes S. There is an edge $a = (s_1, s_2) \in A$ between two nodes s_1 and s_2 if and only if an action has been defined that lets the application switch from s_1 to s_2. An edge is also called an action or **transition** since it describes the transition from its source to its target state. Each edge is associated with an action ID by a special attribute mapping function, the **action-ID function** $action: A \rightarrow Actions$ with *Actions* being the set of action IDs.
One node can be the source or target of several actions. However, the graph must fulfil the following constraints:

- **Disambiguation Property**: $\forall (s_a, s_b), (s_a, s_c) \in A : action((s_a, s_b)) \neq action((s_a, s_c))$, i.e. all actions with the same state as source must be pairwise disjoint. This means that from one state there cannot be transitions fired by the same action to two different states. Otherwise it would be non-deterministic as it would not be clear which strategy should be employed to choose the transition that should be used when the according action is executed. This also implies that no two edges between two states have the same action, eliminating redundancy.
- **Start State Property**: There is a distinguished state called the start state s_S that is a source, i.e. not the target of any transition. This represents the state the application is in right before its use.
- **Reachability Property**: $\forall s \in S : \exists p(s_S, s)$, i.e. all states can be reached from the start state by a sequence of transitions. Note that this is not the same as saying that every node must have an incoming edge (simply imagine a graph with two connected components). This also implies that there is exactly one start state s_S.

We will see several examples of state graphs in the following sections and also in Chapter 7. An important aspect in the system we will describe below is that these properties are ensured *by construction* and thus cannot be violated.

5.1.3 Advantages of Using State Graphs in User Interaction Design

Thimbleby and Gow describe several aspects that can be derived from an underlying graph model [Thimbleby and Gow 2007]. Definition 4 introduces five terms that prove to be useful for further characterisations of graphs for that purpose.

Definition 4 (Eccentricity, Radius, Centre, Diameter, Periphery)
In a graph $G = (S, A)$, the **eccentricity *ecc*(s)** of a node $s \in S$ is the longest shortest path $p(s, s')$ with $s' \in S$.
The **radius** of a graph is its smallest eccentricity, i.e. $rad(G) = min(\{ecc(s) \mid s \in S\})$. The **centre** of a graph is the set of all nodes with eccentricity equal to the radius.
The **diameter** of a graph is its largest eccentricity, i.e. $dia(G) = max(\{ecc(s) \mid s \in S\})$. The **periphery** of a graph is the set of all nodes with eccentricity equal to the diameter.

Note that those definitions use the notion of lengths of paths. If this is replaced by the length with respect to an attribute as defined above, we can derive the same definitions with respect to this attribute, e.g. ***ecc*(s, X)**.

Some properties of state graphs important with respect to usability are listed in Table 13 and Table 14 together with the sufficient characteristics of a state graph $G = (S, A)$, with $s, s_a, s_b, s_{off} \in S$, and $a_0, a_1, ..., a_n \in A$ that can be used to verify or check this property. Those in Table 13 are adapted from [Thimbleby and Gow 2007], re-written in a more concise and formal way. We also added several novel properties shown in Table 14 that are of interest from a usability point of view such as a possibility to predict user goals.

Table 13: Properties of applications or devices that directly map to properties of the corresponding state graph.

Property of the Application / Device	Property of the Corresponding State Graph
There are no dead-ends and no unreachable states.	G is strongly connected.
There is a state s from which one can get everywhere but not back, e.g. for a non-reusable device like a fire extinguisher.	G is connected but not strongly connected, $G \setminus \{s\}$ is strongly connected
The set of states from which all tasks are cheapest.	Nodes in the centre
The set of states in which all most expensive tasks start and end.	Nodes in the periphery
The expected average cost of a task.	The characteristic (average) path length, i.e. $\left(\sum_{p\ shortest\ Path} \|\|p\|\|\right)/\|\{p: p\ shortest\ Path\}\|$. This does, per definition, not include trivial paths of length 0.
Cost to undo an action.	$\|\|p(s_b, s_a)\|\|$ for an action (s_a, s_b). Average undo cost: $\left(\sum_{(s_a,s_b)\in A} \|\|p(s_a, s_b)\|\|\right)/\|A\|$
Cost to undo an 'overrun' action, i.e. accidently executing an action twice (e.g. double instead of single click; often happens when feedback is not present or slow)	$\|\|p(s_c, s_b)\|\|$ for a path $p(s_a, s_c)$ using actions $a_1 = (s_a, s_b)$ and $a_2 = (s_b, s_c)$ with $action(a_1) = action(a_2)$
Cost to 'reboot' an application and return to the previous state.	$\|\|p(s_b, s_{off})\|\| + \|\|p(s_{off}, s_a)\|\|$ for an action (s_a, s_b)
Set of actions essentially to know to be able to use all functionality.	Smallest set of edges that disconnect G (also called minimum cut).
States good to know in order to use the device.	Very strongly connected states, i.e. $\{s \mid deg(s) > n\}$ for some n ('hub nodes').
Cheapest sequence of actions to use all actions. Important, e.g., for testing purposes.	Solution to the Chinese postman tour problem (see for example [Aho, Hopcroft, and Ullman 1983]).
Cheapest sequence of actions to visit all states, i.e. related to the minimal knowledge to be able to control the application perfectly.	Solution to the travelling salesman problem (see for example [Aho, Hopcroft, and Ullman 1983]).

Table 14: More properties that directly map to properties of the corresponding state graph, see Table 13.

Property of the Application / Device	Property of the Corresponding State Graph
Predicted goal of a user interaction.	If $\|\|a_n\|\|_{(a_0,a_1,\ldots,a_{n-1})}$ is the conditional probability that an action a_n is executed after having executed the sequence a_0, a_1, ..., a_{n-1}, the most probable goal starting from a state s_a can be computed by $\{s \mid \|p(s_a,s)\| = min\{\|p(s_a,s_b)\|, s_b \in S\}\}$. Those probabilities can be approximated by using the relative frequencies of logging data. A simplification could use the probability that an action is executed instead of the conditional probability.
Set of paths that justify introducing shortcuts (i.e. those with high cost).	$\{p \mid p\ Path, \|p\|_X > n\}$ for some threshold n in the magma formed by $\{att(a,X) \mid a \in A\}$ (see Definition 2). This can also be done or combined with the previous measures.
Necessary steps to get to a specific state s (e.g. how to get to the list of messages).	$\{p(s_S, s)\}$ where s_S is the start state or the current state.
A measure of how many ways there exist to reach a goal.	$\left(\sum_{s_a \neq s_b \in S} \|\{p(s_a,s_b) \mid p\ acyclic\}\|\right) / \left(\|S\|(\|S\|-1)\right)$ gives the average number of paths between two arbitrary nodes s_a, and s_b. The condition (p acyclic) is necessary to exclude the infinite number of paths containing cycles.
Cheapest way(s) to achieve something.	Find the shortest path(s).

As in Definition 4, the notion of 'cost' (and thus 'cheap' and 'expensive') has been chosen to allow for different metrics in the underlying actual cost of edge traversal. If the length of a path is defined by the number of its edges, then a cheapest path uses the least possible number of edges. The length could also be defined with respect to a certain attribute leading to a minimisation of the sum of all edge weights it uses. As such, a cheapest path can mean the most efficient, quickest, easiest, the one which generates most fun, is used most often, etc.

5.2 Graphical, State-based Application Development

There are two ways of creating applications on top of state graphs. The first is to develop an application and then generate a state graph from the program description or from executing representative tasks on the application. The second is to generate the application semantics in conjunction with the state graph. We will concentrate on the latter approach in the rest of this as well as in the following chapter. More precisely, we will show how to create the state graph in parallel to application development or even how to directly build the state graph in order to define application behaviour as done in the following section.

Two of the requirements identified in the previous chapter are 'simple applications should be simple to create' and 'visual tools should be easy to put on top of the toolkit'. In order to demonstrate ways to accomplish these requirements with the EIToolkit, we developed several add-ons to the system. A most simple one is a wizard producing a sample project (Visual Studio, C++) specialised to parameters like name, support for receiving and / or sending messages, etc. The next four subsections show more sophisticated approaches concentrating on graphical ways of developing applications. The first two are stand-alone applications, while the others illustrate how existing tools can be augmented with such functionality.

5.2.1 Example 1: Output-state-based Development

Simply put, one goal in writing (pervasive) applications is to make something happen, under certain circumstances. Slightly rephrased, an application or device should execute a specific function or be brought into some state under specific conditions. This view applies especially well to simple devices that use input and output facilities with discrete states such as buttons or lights. Additionally, even continuous inputs are often used in a discrete way by applying thresholds or defining intervals, e.g. triggering an event when the temperature drops below 30 degrees (see for example [Hartmann, Abdulla, et al. 2007]). Proceeding along this way of thinking, we present a visual approach exploiting several features of the EIToolkit and focus on system states. Our OSBAD system (working title, being the abbreviation of '**o**utput **s**tate **b**ased **a**pplication **d**evelopment') directs the user along a series of steps until a set of rules has been specified that define the desired behaviour.

The simple process involves the following steps:

1. **Specify output**
   a. Specify the components that should be brought into a specific state and / or
   b. Specify the operations that should be executed
2. **Specify inputs**
   a. Identify the components whose state (changes) are of interest and / or
   b. Identify the events that are of interest
3. **Define dependencies**: which output components depend on which input components
4. **Optionally define rules**: specific input conditions that trigger the output

Example scenario
David lives in a smart home where most of the electronic devices are connected and remotely controllable. As a simple application, he wants the lights to go on when he enters his house. He indicates four main lights and decides to trigger them after having stepped through the garden door as well as the front door. His detection system informs him when he is on his premises and in his house. After leaving both, house and garden, the lights should be switched off.

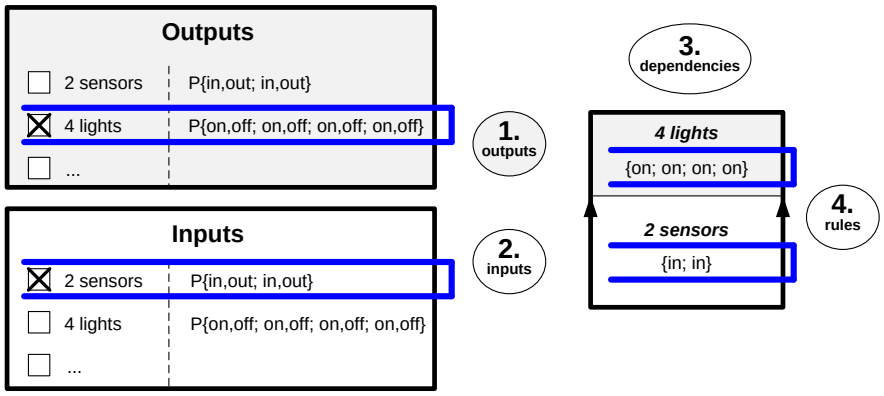


Figure 31: Conceptual interface to generate rules focused on the output of an application. Here, four lights are to be switched on whenever the two sensors are in the correct state.

Figure 31 illustrates the steps mentioned above. David would choose the four lights as desired outputs and the two sensors as inputs. In the final step, he would specify that the four lights should be switched on, i.e. the application controlling them should be in state {*on*; *on*; *on*; *on*}, whenever the two sensors indicate that he passed both, his garden and the front door, i.e. this application enters state {*in*; *in*}.

To simulate this scenario, consider one input device with two buttons that can either be pressed, '*yes*', or not, '*no*', thus simulating the detection system, and one output device with four lights that can each be switched '*on*' and '*off*'. The top left part of Figure 32 (which shows the user interface of the OSBAD prototype) lists both components ('*2Buttons*' and '*4LEDs*'). The 'range' column indicates the state space of the components. The *4LEDs* component, for instance, can be in any of the 16 states possible with four independent lights with binary states each denoted by the power set 'P{*on,off; on,off; on,off; on,off*}'. This is chosen on the output side. The input condition will depend on the state of the *2Buttons* device (see the selection in the top right part of Figure 32). In the visualisation, it is also possible to choose operations (on the output side) or events (on the input side), i.e. commands that should be executed when a certain event occurs. These two approaches, state-based and action-based, can also be mixed, e.g. a command is executed when a device is in a specific state.

The dependency of input and output components is visualized in the bottom left part of the figure. Two concrete instances of a condition have been created to the right of it. They specify that, if both buttons are pressed, i.e. *2Buttons* is in state '{*yes*, *yes*}', all lamps are switched on, i.e. *4LEDs* should be brought into the state '{*on*, *on*, *on*, *on*}' and, if no button is pressed, the lights are switched off. In all other cases (one button toggled, the other not), nothing happens. This is exactly the semantics described in the scenario.

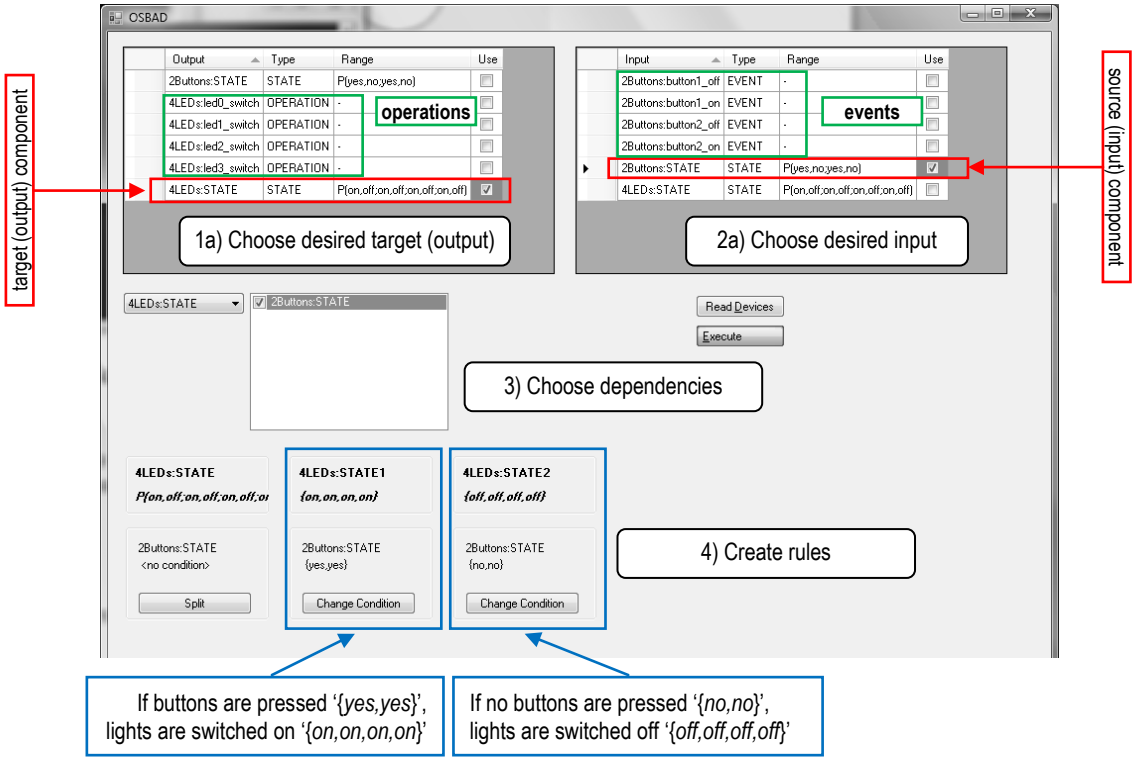


Figure 32: The main screen of the OSBAD application development prototype. If the two buttons (component '*2Buttons*') are pressed (state '{*yes, yes*}') the four LEDs (component '*4LEDs*') will be switched on (state '{*on, on, on, on*}'). Similarly, the second condition specifies that in the moment when no button is pressed, all LEDs will be switched off.

The EIToolkit, on which OSBAD is based, takes care of the automatic detection of available components, the categorization into input and output components (using the interface description capabilities of stubs) as well as the execution of the rules built using the OSBAD system. Whenever the '*Execute*' button is toggled, all specified rules are active. Rules can also be (de-)activated, removed, added and changed during runtime. Using two simple EIToolkit components, the given scenario can easily be emulated.

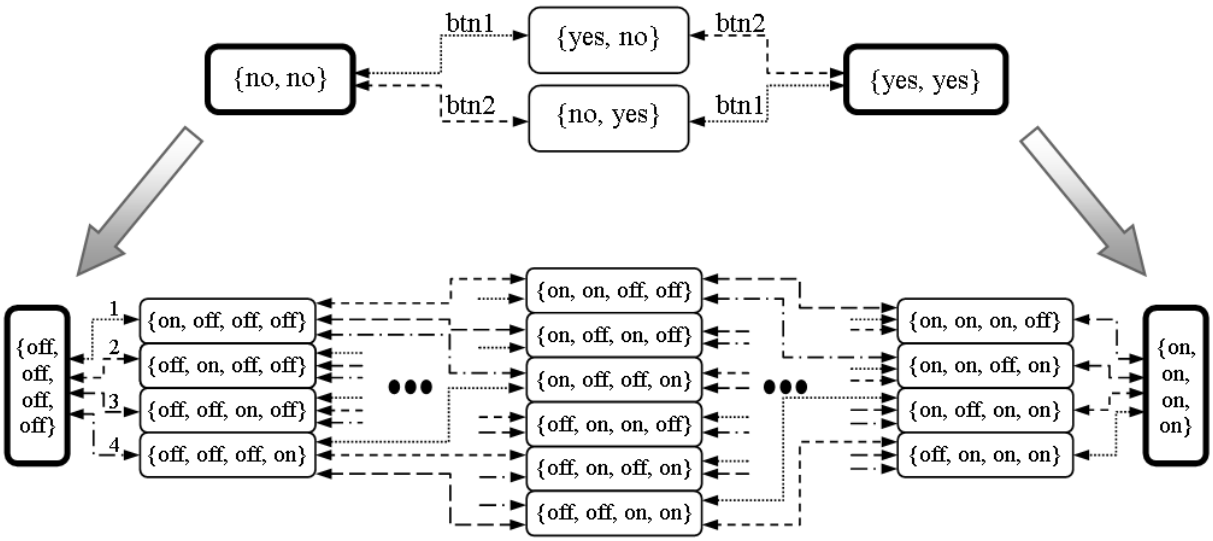


Figure 33: A visualisation of the state graphs for the components '*2Buttons*' (*top*) and '*4LEDs*' (*bottom*) used in Figure 32.

The application behaviour is again illustrated in Figure 33 (the example will also be reused to demonstrate another technique in the following section). A device with four LEDs can be in 16 different states. Whenever the two buttons are in one of the highlighted states, the four lights are brought into the highlighted states of the *4LEDs* component, respectively. Note that there are potentially 4! = 24 different shortest paths to get, e.g., from

state ‘{*off, off, off, off*}’ to ‘{*on, on, on, on*}’ provided there exist methods to independently toggle each light. However, there are also infinitely many paths that do not lead to the desired state (since the graph contains cycles). The system must be able to find one of those correct paths automatically.

One way to solve this problem is to know the state graph in advance and apply standard algorithms for finding paths within graphs. Although there are a lot of advantages when the state graph is known (see Section 5.1.3 for a more detailed treatment), it is cumbersome and error prone for application developers if they have to specify it independently of application development. If, for some reason, the state graph of the system is known, the desired path(s) can be easily calculated using any graph search algorithms like breadth-first search. This includes finding the shortest path or the best path (e.g. using Dijkstra’s well-known algorithm, see for example [Aho, Hopcroft, and Ullman 1983]) in the event that transitions are labelled with some cost function. In the following Chapter 7, we will present a project that bases application development on state graphs which can then be used to make such calculations.

We now briefly present a heuristic to find such a path without knowing the state graph in advance. To be able to easily switch between various heuristics, our implementation uses a stub component to encapsulate it. The StateFinderStub uses a heuristic based on the assumption that every action can be undone by executing it a second time directly after the first execution. This assumption captures of course only a small portion of all possible applications. It implies that, e.g., there are no counters or methods with side-effects (making it useful in combination with pure functional languages such as Haskell). The same approach can of course be used whenever a dedicated undo function is implemented in the system and made public to the StateFinderStub.

The stub executes a modified depth-first search in the state graph (without actual knowledge of the state graph). The necessary backtracking operation is enabled by the undo assumption. This may even include backtracking beyond the state at which the search was initiated since one of the tested actions might undo the action from a potential previous state to this start state. This means that the desired end state will always be reached, i.e. the search will be successful, if there exists a path in the undirected graph underlying the directed state graph. The heuristic tries to speed up the process of finding the target state by following a gradient process. A simple metric is used to define the distance of the current state to the target state. The system then first pursues those edges where this distance does not grow. Since the state graph may contain cycles, the algorithm marks visited edges in order to avoid unnecessarily executing actions. In fact, most real devices have cycles, one that occurs most often being from any state to the ‘*off*’-state and back.

As is the case with most heuristics, this approach can also slow down the search if the distance function is not monotonic along paths from the source to the target. In fact, for a state graph with m edges, a simple greedy algorithm would need at most $2m-2$ steps (each edge but the last two traversed two times, first to execute an action, then to undo it). It can easily be shown that the heuristic could need up to $3m-4$. However, it is conjectured that the heuristic nevertheless makes sense since many paths in state graphs will follow decreasing state distances and the asymptotic complexity still remains in $O(m)$, i.e. $O(n^2)$ with n being the number of nodes in the graph as the number of edges can be quadratic in n.

The assumption that every action can be undone by repeating the action assures and implies that there are no two states that lead to the same state with the same action, otherwise the undo would be ambiguous. The assumption at least holds for two-state actuators like simple lights, some display implementations and internal Boolean states. It is also possible to get rid of this assumption and allow any instance of a state graph. However, this complicates the path finding algorithm. It cannot, for instance, be guaranteed anymore that a solution is found if the graph is not strongly connected.

An informal evaluation of the OSBAD prototype with five people with profound technical knowledge but no background in visual programming showed that the user interface – though clearly structured according to the interaction process – needs some training time. Also, having first to define the output side follows a goal oriented process but can be counter-intuitive for people who think more along the lines of causes and actions. This led to a different view described in the following section.

5.2.2 Example 2: Trigger-action-based Development

The OSBAD system in the previous section focused, in both its visualisation and its task flow, on the target (output) state of the system. This makes sense since often the 'results' of an application are most important. However, the mental model of users developing a system often more closely follows an if-then approach. This most possibly results from the wide-spread use of imperative programming languages and might be different for other groups. People without programming skills, for example, often prefer to demonstrate what they want the system to do [Lieberman, Paternò, and Wulf 2006]. This model of programming by demonstration is also different from the output-state-based approach. We provide such a system in Chapter 7.

Our DEVICEVIZ prototype presented now offers very similar power to the previous system but stresses the flow of action from triggering states and events to their respective causes. It also allows the specification of a more direct connection between triggers and the available methods as well as the injection of custom code into the created system.

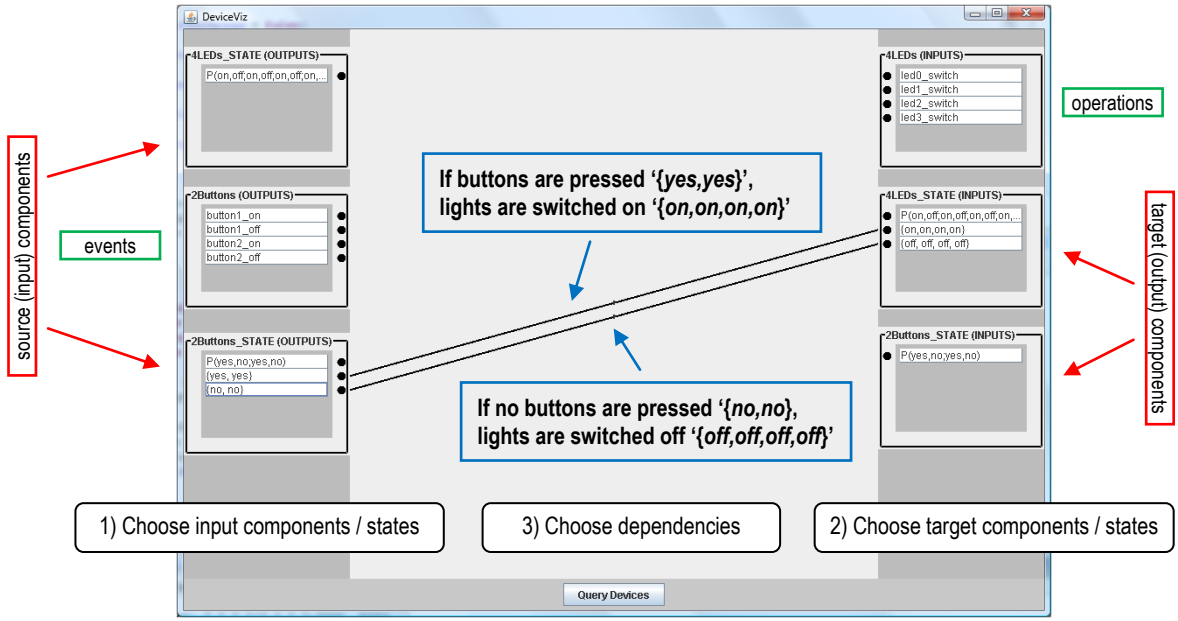


Figure 34: The main screen of the DEVICEVIZ prototyping environment. Components that produce output (events) are arranged on the left hand side. Those that accept input (commands) are shown on the right. This figure shows the same application as developed in the OSBAD environment in Figure 32. When the two buttons are both pressed / not pressed, the four LEDs are switched on / off, respectively.

Figure 34 shows that the input and the output side are both given the same emphasis. Following the reading direction from left to right common in the US and Europe, components that provide some output, i.e. fire events, are placed on the left hand side; those that accept input or can be triggered, e.g. commands, are placed to the right. Obviously, the direction of flow can trivially be adjusted to a reading direction from right to left. A vertical layout would need slightly more effort since the connection points would also have to be altered. Since states of components can act both as sources of action and targets, a state box appears for each available component on both sides. Initially, it shows a general description of the state space, e.g. the power set '*P*(*yes,no; yes,no*)' for the component featuring two buttons as explained above. The user can click on this description and generate an arbitrary number (up to the cardinality of the state space, i.e. |*P*(...)|) of specific state instances like '{*yes, yes*}'. Placeholders for parts of a state that should not be restricted can also be used in that pattern matching process. As in the previous tool, it is also incorporated to add simple tests like '> 10'. However, since the tool is not targeted at experts, it is not implemented as a full logic processor and also currently does not allow calculations or references to other data. In the example shown in Figure 34, the two states indicating that the two buttons are both pressed or released are connected to two output states of the component with four lights, thus creating the same application as in the description of the previous system. The connection can be drawn by a simple drag-n-drop operation with the mouse. A rubber band and snapping mechanism helps in defining application behaviour.

As mentioned before, DEVICEVIZ also allows quick rules to be set that do not include states. Figure 35 indicates how the method '*led0_switch*' can be triggered whenever the first button is pressed or released. The code box that can be flipped open in the centre of the screen gives a rather high level view of what this particular connection means. This box enables developers to insert additional code logic that is out of the scope of the graphical development tool. This is, for example, necessary when using external knowledge or invoking complex calculations. When generating the execution rules, the additional code can be compiled and used in the rules at runtime. It would also be possible to apply different kinds of programming paradigms in these boxes instead of text based coding or to use a fully fledged code editor like the one found in Eclipse.

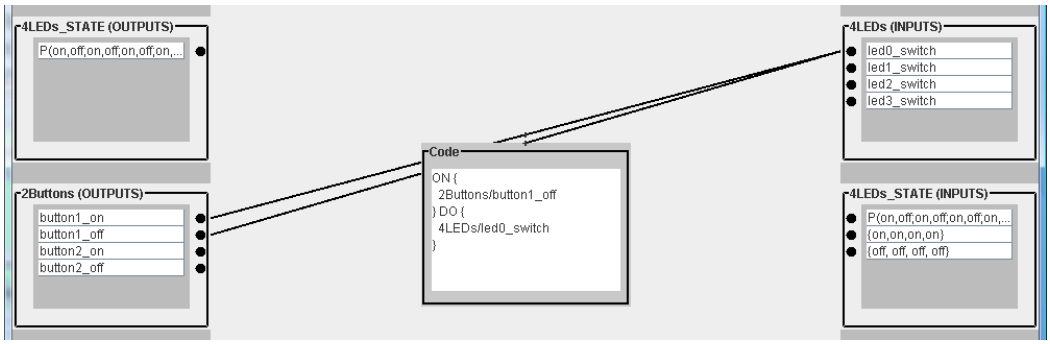


Figure 35: Using '*button1*' to toggle '*LED1*'. The pseudo code of the lower connection is shown with precondition ('*2Buttons/button1_off*') and consequence ('*4LEDs/led0_switch*').

All such visualisations that strive to offer the user a simplified view and a handle on the available system struggle with a possible reduction in expressive power. Although the scripting code component gives additional freedom to advanced users, several aspects still cannot be covered easily. Some questions include:

- How is data transferred from one component to another (e.g. the ambient light level to adapt a light source)?
- How are wildcards and patterns visualised and used?
- How can visual clarity be ensured for more complex applications?
- What if multiple inputs need to be taken into account for one action?

The first two issues are difficult to solve for people with no previous knowledge or understanding of the concept of variables. It can partly be overcome by using colour-coded shapes or icons to represent the same data at different places. Our suggestion for slightly advanced users is to employ a notation standard in many pattern matching languages, i.e. to use '\1', '\2', etc. in order to specify the pieces of data used as input or matched against. Considering the third issue, visual clutter, we recognise that this is a problem common to all graphical development tools. Currently, the components can be rearranged by hand but we could also employ graph drawing algorithms used for bipartite graphs. If the number of available components grows, one has to introduce means to help finding a specific one, e.g. through a search box or dialog (which could also use patterns to search for certain interfaces). We explicitly provide a solution for the fourth issue mentioned above, i.e. the fact that several components on the input side may need to be used for some actions. A similar problem is that a specific action may only be triggered by an event if the system is in a specific state. A simple example is that any button of a device only has an impact if the device is currently switched on. We deal with such issues by introducing an additional component. The box in the centre of Figure 36 allows an arbitrary number of input connections and offers ways to connect these, e.g. using a Boolean 'and', meaning that both preconditions have to be satisfied.

Advantages of such approaches are that the application semantics can be quickly grasped, its behaviour can be adapted during runtime with a small number of mouse clicks, and it offers a visual stage that eases the threshold to begin developing and reduces the amount of programming necessary. Another very helpful feature (mentioned as a key requirement in the last chapter) is that there can be a tight coupling between the application and the development tool as demonstrated, for example, in the d.tools system [Hartmann, Klemmer, et al. 2006]. By manipulating generators of data and events, e.g. occluding a hardware light sensor, the user interface can present the data and, more importantly, highlight which component is currently active and which event or rule would be fired. This largely eases development by demonstration and exploration as well as incremental development.

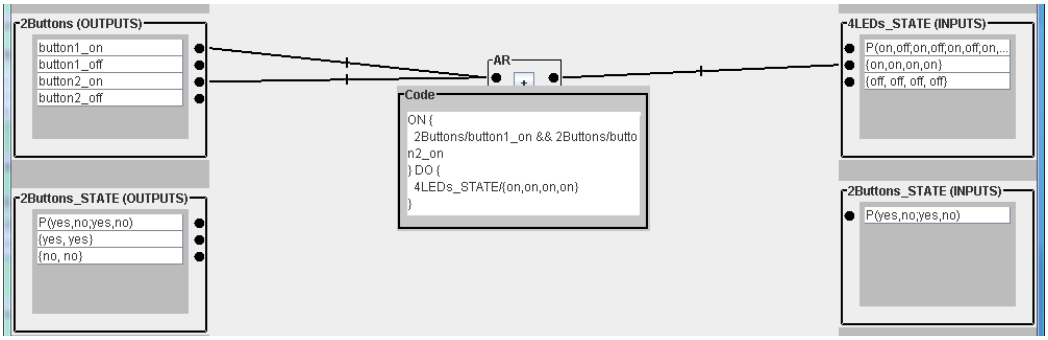


Figure 36: An additional component allows combining several input events with Boolean or comparison operators. In this example it specifies that '*button1*' and '*button2*' both need to be pressed to trigger the specified action.

5.3 Combining Models and State-based Prototyping Tools

Creating user models and developing applications are both non trivial tasks. There are many people that are experts in one of these domains. Only few of them would judge themselves to be experts in both areas. An additional problem in using models such as those mentioned in previous chapters for developing and prototyping applications is that the initial threshold to be able to build accurate models is quite high.

> "A common criticism of formal methods, and indeed of many other usability engineering methods, is that they depend for their success to an excessive degree on craft knowledge […]. That is, there are specialized skills tacit in many approaches, and other practitioners therefore find them harder to deploy with good results than their proponents."
> [Thimbleby, Cairns, and Jones 2001]

We have seen in the last chapters that there are a number of tools that can help in creating pervasive applications. An integral part of the current chapter is to provide methods to combine the two worlds of cognitive modelling and prototyping of pervasive applications. Both tasks can profit from a description of the other one: modelling tools can exploit knowledge about an application's internal structure and description of its user interface; prototyping tools can use information from modelling processes to automatically adapt applications according to identified issues and provide the user with concrete suggestions about solving problems identified by the models.

After introducing some new approaches to make modelling easier, we go into more detail about core components for a framework we built that can combine the creation of prototypical applications with the generation of models for those applications and show how these can be interconnected. It uses a variety of concepts introduced in the last chapters, most prominently the Keystroke-Level Model and its extensions (Chapter 1) and the EIToolkit and its areas of application (Chapters 4 and 5). The following Chapter 6 applies much of this research and goes into detail of a development environment specialised in the creation of mobile phone applications.

5.3.1 KLM Component for Combining Prototyping with User Modelling

It is possible to automatically generate a Keystroke-Level Model for a specific task expressed as a series of actions. Since such a generator is of use independent of an application, we exploited the EIToolkit architecture and built a KLM stub for that purpose. It mainly consists of a database of mappings from actions to time values, each optionally with a set of options. In our implementation, we chose XML to describe these mappings.

As can be seen on the left hand side in Figure 37, such a specification consists of some ID describing the action (<*prop*> tag) and timing information (<*klm*> tag). This simple case is displayed for example in the '*Take Picture*' action which represents the response time needed on average by a phone after taking a picture. The value of 1.23 seconds is simply recorded. A more advanced entry can be found, e.g., with the action of pointing the phone somewhere, for example in order to touch an NFC tag or to take a picture. Three options are provided, one being an overall estimate of 1.00 seconds. Another option allows specifying a user-defined value while the third one employs Fitts' Law. The parameters for the last option specify the constants of the pointing device in

use as well as the actual distance to the target and the target size. The text within the <*option*> tag specifies the formula to actually calculate the time necessary to execute this action. The right side of Figure 37 shows the basic graphical interface of the stub by presenting all available options to the user and potentially allows adding new or changing existing entries.

```
...
 <prop name="Distraction">
  <klm>
   <options>
    <option opt1="slight">6 %</option>
    <option opt1="strong">21 %</option>
   </options>
  </klm>
 </prop>

 <prop name="Point Somewhere">
  <klm>
   <options>
    <option opt1="estimate">1.00</option>
    <option opt1="custom" opt2="$0">$0</option>
    <option opt1="fitts' law" opt2 = "$0" opt3="$1" opt4="$2"
            opt5="$3>$0+$1*lb($2/$3+1) "</option>
   </options>
  </klm>
 </prop>

 <prop name="Take Picture">
  <klm>1.23</klm>
 </prop>
...
```

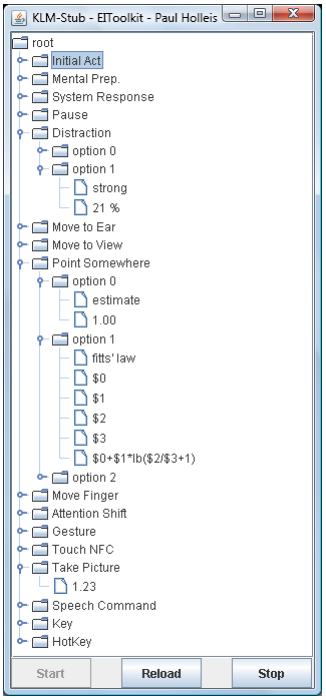


Figure 37: Left: excerpt of an XML file specifying existing KLM operators. '$1', '$2', ... placeholders can be used as variables to specify required input from the modeller. Modelling pointing with Fitts' Law, for example, requires four parameters, see Section 3.1.2. Right: visualisation of the KLM stub showing the available operators.

Some time values depend strongly on the type and instance of the interaction device. There are different values assigned for, e.g., the homing operator in the original KLM describing the hand movement from keyboard to mouse and vice versa, and the same operator used in a mobile setting to describe different interaction modes with a mobile phone. There are two possibilities to incorporate this in the XML file. Either this is taken care of by choosing designated options, or by defining different sets of these mappings in one single XML file. The last approach has the advantage of needing to specify the overall setting just once. Obviously, some parameters that stay the same (like the mental act operator) then appear several times unless some kind of reference scheme or hierarchical structure is devised – an option that would render the file more complex to understand and edit.

The mode of operating the stub is then simply as follows. First, initiate the task by sending a '*start_interaction*' command and specify the type of KLM to be used. Next, action descriptions can be sent. For each of those, the stub adds the appropriate operator if available in the chosen mapping and returns the associated time value, i.e. sends a message back to the sender of the action. By emitting the '*end_interaction*' command, the task is specified to be finished and the stub can send the sum of the times of the single operators.

The KLM stub can be used by any party that uses or is compatible with the EIToolkit's ways of communication. It does, however, in its current state of development not provide much more functionality than the controlled mapping between actions and KLM operators. It would be easy, though, to incorporate more services like automatically applying the common heuristics to place mental operators, to time and add pauses in a live demonstration of tasks as mental or system response operators, or to add more graphical visualisations of tasks and their model. In the following two sections, as well as in Chapter 7, we show how this stub can be integrated into novel and existing development environments in order to exploit the advantages of early modelling task interaction times.

5.3.2 Example 1: Integration into the d.tools Environment

The d.tools design, test, and analysis environment offers a statechart-based visual design tool [Hartmann, Klemmer, et al. 2006]. In this section, we describe how to integrate Keystroke-Level Models into its design process. At this point, it is helpful to know that d.tools manages the communication between the software part and its hardware using OSC messages over UDP. This makes the direct connection to the EIToolkit very simple. With slight modifications, the discovery and description mechanisms of the two can be combined. Whenever a user initiates a transition in the state diagram, an OSC message is generated that can be used to control any hardware or software connected to the EIToolkit in the same way as it influences any d.tools component.

The d.tools system offers an approach similar to our mobile phone development environment presented in the following Chapter 7. It is based on a visual, statechart-based prototyping model as can be seen in Figure 38.

Creating the model

The first step in using d.tools is designing the look and interface parts of the device currently built. All compatible physical user interface items like buttons, lamps, or microphone have a visual equivalent. Already in this step, user modelling tools could be employed. For example, Fitts' Law could determine the time needed to move from one button to the next; or some models could be used to find out which elements can potentially be operated simultaneously (see, e.g. Section 3.1). However, without a more detailed knowledge of the semantics and use of the application, most existing tools build on actual interaction sequences or logs and are therefore of limited value. Fitts' Law, for example, can only be applied if both, the target and the source are known, i.e. where a user has positioned the pointing device before clicking on the target. A more fitting approach in this case would be to offer help in implementing guidelines for user interfaces which are available for different systems, for instance for Windows Vista[53], the Apple user interface[54], or general web pages, e.g. [Lynch and Horton 2004]. Similar approaches for tangible interfaces are still a matter of research.

The device that is being designed with d.tools in Figure 38 on the next page is simple but offers enough possibilities to show the idea and features of the system. It mainly consists of two buttons and two differently coloured lights. These visual elements can have direct equivalents in hardware, i.e. two push buttons and two LEDs. A simple statechart has been created that consists of six states. Four of them are different in their appearance using the two lights as indicators (all off, all on, and two states with exactly one on). The example shows that it is sometimes necessary to clone the same visual appearances into several states. Possible reasons include different desired behaviour depending on the sequence of actions that led to this state, or different internal states that are not reflected by the indicators. There are several paths that take the device from the state with both lights switched off (top left) to the state with only the left light switched on (in the centre). One of these paths is highlighted in the illustration. The sequence of button presses for this path is '*1-1-1*'. Another possible path would use the button sequence '*1-2*'. Since the graph contains cycles, there are an infinite number of possible paths, e.g. by repeating the sequence of actions '*1-1-2-2-2*'.

[53] Microsoft Windows Vista, user experience guidelines: *http://msdn2.microsoft.com/en-us/library/aa511258.aspx*
[54] Apple, user experience guidelines: *http://developer.apple.com/documentation/UserExperience/index.html*

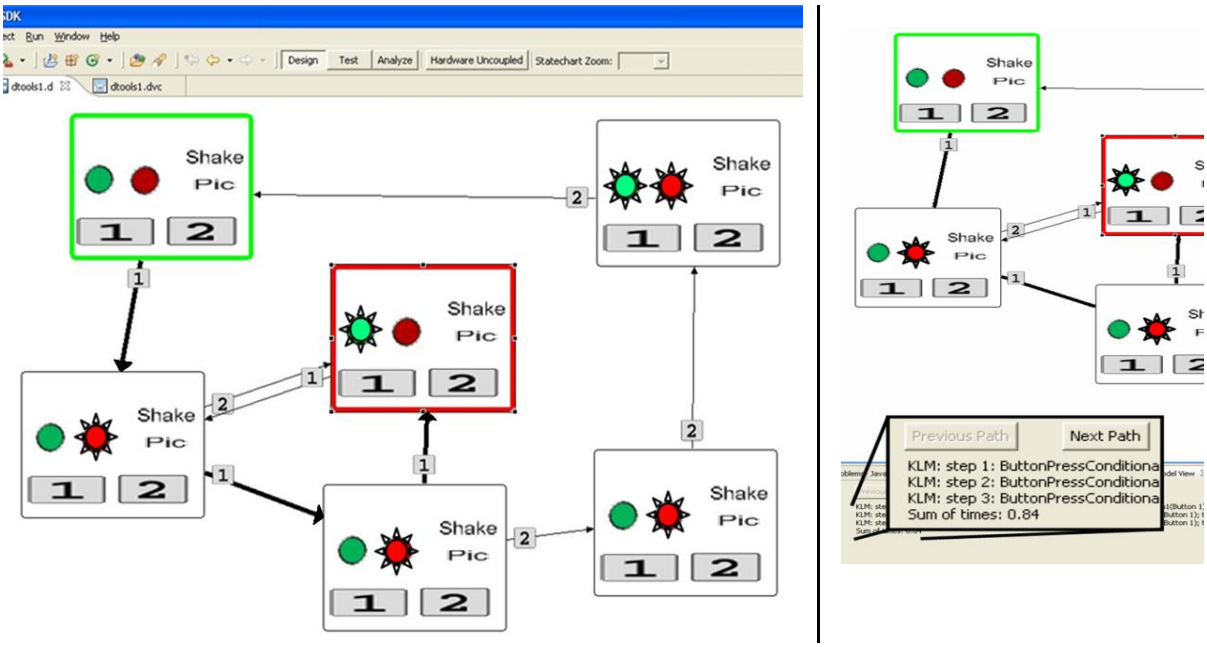


Figure 38: *Left*: d.tools visualisation of the states and transitions of a simple application involving two buttons and two lights with the start (top left) and target (centre) states selected. A path between them is highlighted (button sequence '*1-1-1*'). *Right*: The bottom pane shows a series of KLM operators and an estimate of the time necessary to execute the selected action sequence on the target device.

An interesting aspect to note here is that each action from the user is clearly defined by the interaction object in use. For example, if a transition from one state to the next is triggered by pressing a button, this action defines the necessary user action, i.e. pointing to the button and pressing it. Still, this information may not be enough to apply the models described in the previous chapters. To distinguish between, e.g., a button on a touch screen and a physical push button, additional information may be necessary. This can either be done by using special widgets while designing the device or by editing properties of the widget. As we have seen in the description of the KLM stub in the last section, it often suffices to have a set of conditions once at the beginning. A setup could be described, e.g. by specifying that the device in use is a mobile phone with a touch screen or a desktop application to be controlled with a mouse.

User model descriptions can then be added automatically to the application description. Whenever a transaction from one state to another is defined, an appropriate description of the user model, e.g. KLM time information, can be added. If a button press is necessary, a KLM can augment the transaction with information about the time needed to press this button.

Using the model

As shown in Figure 38, the extension of the d.tools environment allows selecting a start and a target state. This initiates the calculation of all possible, acyclic paths between the start and the target state. One of those is immediately highlighted. In a different view, the model properties for this particular path are shown. This view also offers the possibility to browse through the set of paths. The KLM parameters seen in the screen give an indication of the time each of the transaction will need (within the assumptions detailed in Section 3.2.2). This can then be used to calculate a total amount of time necessary to execute the given sequence of actions.

Some of the model's operators need more information than is encoded in the transaction alone and often even more than in the triple '(*current state*, *transaction*, *next state*)'. If pointing time should be included in the estimate, e.g. using Fitts' Law, some information about the previous actions that led to the current state is necessary. In the example, the previous position of the finger or device used for pointing must be known to be able to employ Fitts' formula. Since, at this point, the whole path from the start state to the target state is known, this information can be extracted (except for the start state, where some assumptions about the state of the users have to be made in any case, e.g. whether they have their hands on the keyboard or the mouse).

One of the contributions of the d.tools project, the support for analysing the use of the system within the development environment, also offers opportunities for our approach, i.e. enhancing development with user modelling support. In the same way as the design can be adjusted using results from the analysis, the predictions done by the user models can also profit from actual usage data: although the added modelling is supposed to describe average (expert) behaviour well, a few or even a single trial with an independent user can reveal errors in the model, missing operators originating from the automatic generation of the model, or a divergence of actual data because of novel technology or wrong assumptions. A more detailed treatment of how to combine application development with predictive user models will be described in the following Chapter 7. Contrary to the d.tools approach, this concentrates on the development of applications for mobile devices.

Implementation

Since the demonstration of an action sequence in d.tools already sends descriptive commands of the transitions used, there is little to implement to such information. The pane shown at the bottom right part of Figure 38 can be implemented independently of the d.tools system. However, it was some effort to add certain user interface components: the original statechart visualisation did not allow selecting two states and or assigning specific meaning to them as was necessary in order to specify the start and target state of a particular task. However, because of the modular and object oriented way d.tools is implemented on top of existing graph modelling tools, these additions proved to be quickly possible after having worked through the custom code.

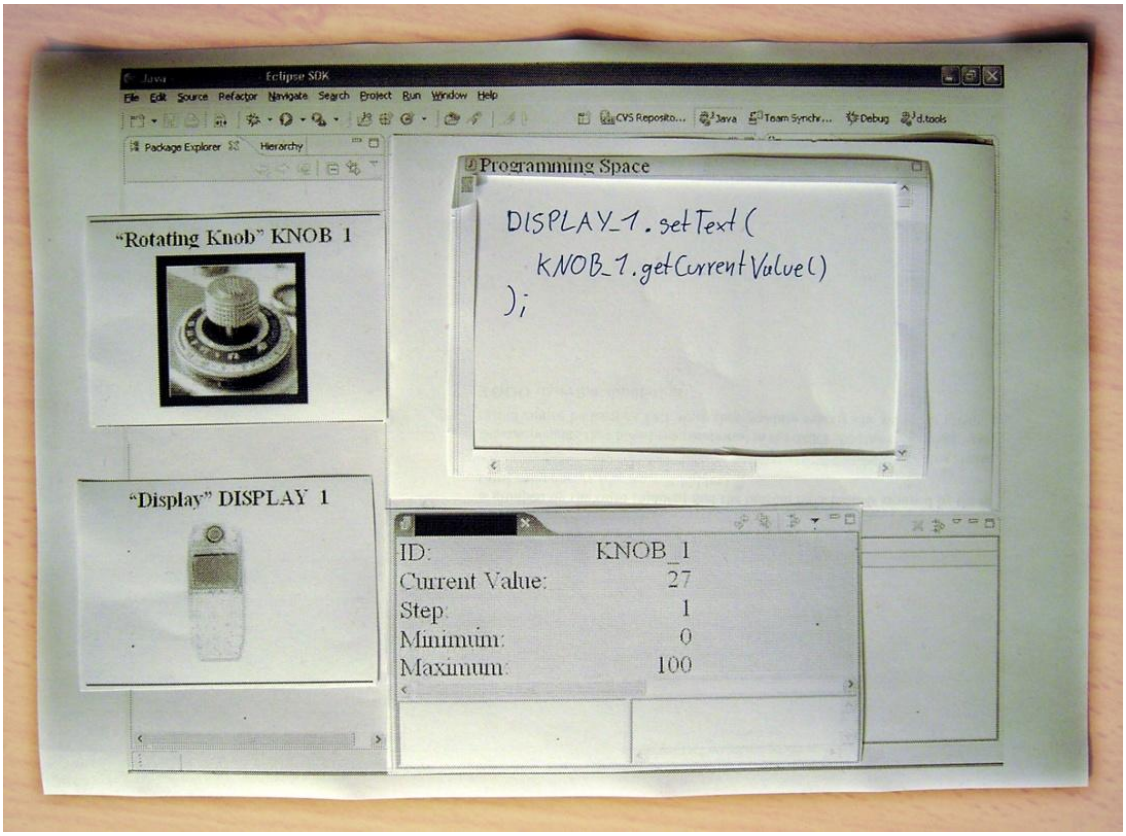


Figure 39: Initial paper prototype of a design of an Eclipse plug-in. Whenever a device or component comes in range, it is displayed and can be used in a way as simple as possible.

5.3.3 Example 2: Integration into the Eclipse Environment

As second example of integrating EIToolkit functionality into existing tools and development environments, we considered the Eclipse IDE. Figure 39 shows one of the initial paper prototypes used to convey the idea, explore the possibilities, and see how to best incorporate the additional views into the existing Eclipse workspace. An initial arrangement of windows (called a 'perspective' in Eclipse) has been chosen. All other parts can be exchanged and are made from small pieces of paper with appropriate text and pictures.

The user interface consists of three main parts:

- One shows a list of available components (in Figure 39, one can see a knob and a multi-purpose display with generic names '*KNOB 1*' and '*DISPLAY 1*' on the left hand side). These are displayed with a name and possibly an icon or image of the represented object.
- Another part provides detailed information about the properties of a selected object (the element '*KNOB 1*' with, e.g., minimum and maximum value is visible at the bottom of Figure 39).
- The third part, the programming space, shows an editor for programming the system. In most cases this would just be the standard code editor provided by the Eclipse framework. Depending on the programming language used, this could of course also be some kind of visual editor. Using the provided editors for, e.g., Java or C++ means that a vast amount of auxiliary information can be employed. This includes function and variable completion, syntax highlighting, automatic compilation, and more.

As an example, consider the setting shown on the left side in Figure 40. A Particle microcontroller uses a light sensor as input and is connected to a light that it can control. The very moment the Particle is switched on, the EIToolkit registers its existence through the Particle stub and makes information about it available to the extended Eclipse system.

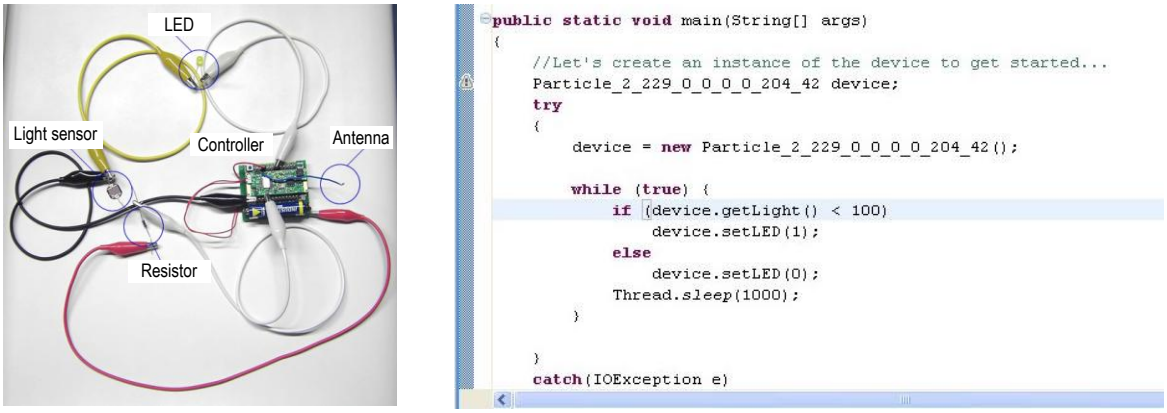


Figure 40: *Left*: a physical test setting using a Particle computer, a light sensor and an LED. *Right*: an excerpt of corresponding code created in Eclipse. The light sensor value is read and, if it reflects a dark environment, the LED is switched on.

This means that a Java class incorporating the information about this Particle is automatically generated (called, e.g., *Particle_2_229_0_0_0_0_204_42*, using its ID to generate a unique class name). The code on the right hand side in Figure 40 shows all that is necessary to implement an application that switches the light on whenever it gets dark, i.e. the value of the light sensor decreases below a certain value (100). The methods used, '*getLight'* and '*setLED'*, can be easily found through the standard auto completion feature of Eclipse since these have been made available in the respective Java proxy class. Figure 41 shows this feature in another automatically generated class encapsulating access to the application Skype. When accessing the Skype component, Eclipse provides the methods available for it. Among those common to all Java objects, one can find methods to login / logout and to set the current state.

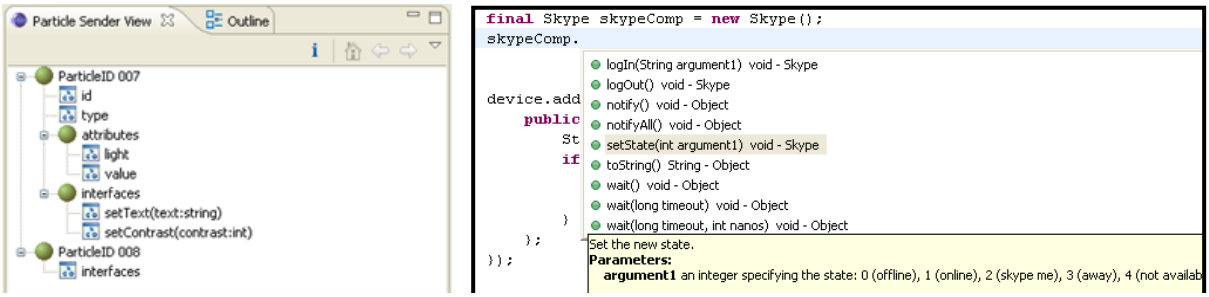


Figure 41: *Left*: Sample implementation of an Eclipse viewer that shows available objects and their interface. *Right*: The Eclipse auto-completion feature when using a stub component. In this example, the Skype stub offers a method called '*setState*' to change the current state of the user.

These generated classes internally use the EIToolkit to relay procedure calls and implement data transfer between the software and the microcontroller system. Although not fully implemented in the prototype, the pull based method shown in this example can also be implemented by registering a listener to a method of interest, possibly giving details about the parameters of interest, and getting notified (pushed) with appropriate events.

The advantage of the presented system is that the implementation is done completely in an environment and programming language that is independent of the hardware implementation. Thus, developers can use their preferred IDE and still make use of advanced technology outside the range of components normally included in such an environment. A disadvantage of all such systems is that processing is split into two parts meaning that data has to be transferred from one platform to the other. In case that sensor data has to be processed in real time and that it is sampled very quickly, e.g., data from accelerometers for gesture detection, this can become a bottleneck. However, it still can serve as a first step in developing an application. A refined algorithm can, e.g., later be implemented on the microcontroller itself while the rest of the application can remain unchanged.

To further help alleviating the problem of having different platforms for this particular setting, a solution has been built for the combination of MS Visual Studio and Particles. We ported the code base running on the Particle computers in a way that it can be compiled in Visual Studio.NET. Much of the internal functionality has been stripped and empty function stubs are used. Primitive data types are translated correctly which helps avoiding a prime source for errors. Not all internal functions are implemented, support for restricting the use of memory is only rudimentary available, and performance will differ greatly on machines, e.g. time slices have not been implemented (in fact, at the time of this writing, developers of the Particles at TecO were starting to work on a more sophisticated emulator platform which might also support some of those constraints mentioned here.). Still, it can serve as a convenient testing environment of algorithms. Again, auto-completion, syntax highlighting, online context-sensitive help, etc. are readily at hand. Code written that way can very often be used exactly as it is on the target Particle. Besides the easier and more comfortable way of editing and debugging, the approach also offers the possibility to simply plug in different functions that emulate external sensors. One can, for example, use the Particle in the target scenario, make some gestures, and record the data. This sensor data can then be used and analysed with a PC application and recognition algorithms written for it. The algorithm can then also be quickly tested against other arbitrary signals. The functions for sending data via the built-in RF module have been altered to directly emulate the output of the receiving bridges. That means that any program supposed to remotely talk to a Particle can also be tested against the emulation without any changes.

Some of the results of the projects described in this chapter have been used in conjunction with other approaches to create a graphical environment specialised for the development of mobile phone applications. It also makes use of the KLM facilities described in this section but combines the development, modelling, and evaluation part more tightly. Thus, in the next chapter, we concentrate on mobile applications and the combination and implementation of many of the concepts treated in the last chapters. It reuses the concept of state graphs and graphical programming and builds on the KLM component introduced above, additionally employing programming by demonstration.

6 Case Studies – Applications Based on the EIToolkit

This chapter demonstrates the concrete use of the EIToolkit in various projects. It shows a set of technologies that are supported in the current state of the system and provides a sample of applications that can be built on top of it.

In the course of our research, several components and applications have been developed that take advantage of the capabilities of the EIToolkit. Some of these have been written to support various devices like a remote controlled power plug (6.1). Others enable the use of some technology in general, e.g. a stub supporting a specific microcontroller platform (6.2). We provide a generic, versatile, and massively configurable visualisation system of input and output components used to observe and control all devices and applications connected to the toolkit (6.3). We then describe the use and advantages of the system in a specific application domain, i.e. wearable computing (6.4).

6.1 Device Specific Applications

In the following, we briefly list a sample of devices and applications that have already been connected to the toolkit. This provides some idea about the variety of options open to a user of the toolkit. Since these stubs are very specific, a short description of each shall suffice. Source code and documentation are available[55].

- **IP power socket:** the PM211-MIP[56], a power socket with two sockets that can be independently switched on and off. The socket is connected to the internet by LAN. A toolkit stub produces – in combination with a username and a password – HTTP get requests that can control the sockets. Other Infratec power sockets (like the PM4-IP that offers four independent sockets) are equally supported.
- **Joystick input devices**: using an open source Joystick library[57], events (position of axes, state of buttons) from various joysticks, steering wheels and joypads are generated and made available over the toolkit. This also works if several joysticks are attached to a single PC.
- **MS PowerPoint presentation control**: events are generated that control a slideshow in a MS PowerPoint presentation.
- **Winamp**[58] **media player**: the media playback controls (start, stop, forward, etc.) can be remotely controlled using the Winamp API[59]
- **Steerable projector**: Dave Molyneaux from Lancaster University is working on steerable projector solutions in combination with tangible user interfaces and employs the toolkit for his purposes[60].

[55] EIToolkit documentation; project page: *https://wiki.medien.ifi.lmu.de/HCILab/EIToolkitDocumentation*
[56] Infratec PM211-MIP Socket, (c.f. Conrad Electronic); product page: *http://www.conrad.de/goto.php?artikel=999171*
[57] Joystick Driver for Java, open source; project page: *http://sourceforge.net/projects/javajoystick*
[58] Winamp Multimedia Player; product page; *http://www.winamp.com*
[59] Winamp Multimedia Player SDK API; developer page: *http://www.winamp.com/development/sdk*
[60] Embedded Interactive Systems, Computing Department, University of Lancaster; group page: *http://eis.comp.lancs.ac.uk*

- **MIDI** (Musical Instrument Digital Interface): a stub has been developed that translated ASCII codes of letters into music notes and plays them over the MIDI device of a PC using the Windows Multimedia API.
- **Small wireless displays**: a stub that allows sending text and images to a wireless display as well as retrieving its 3D orientation; see Section 6.2.1 and [Holleis, Rukzio, et al. 2006b].
- **Laser pointer detection module**: a sensor module detects pointing on an object with a laser pointer modulated with a specific frequency. See [Rukzio, Leichtenstern, et al. 2006] for a use case scenario.
- **Skype**[61] **status**: allows remotely setting the current Skype status [Kranz, Holleis, and Schmidt 2006].
- **Wiimote**[62] **control**: the remote controller of Nintendo's Wii, also known as 'Wiimote', can be accessed including buttons, accelerometer, position data, lights and the rumble feature (a vibration motor).
- **Nabaztag**[63] **control**: the device Nabaztag is used to display information such as financial data in an ambient, unobtrusive, and fun way. Using the Violet API[64], an EIToolkit component has been developed that can control the rabbit's functions including lights, ear movements, and text to speech engine.

It should be mentioned at this point that writing such a stub only involves handling the communication with the device or application it should support. The stub controlling the status of the Skype application, for example, uses only code generated by the EIToolkit stub wizard, code that the Skype API[65] needs (mostly for initially connecting to the Skype application), and 24 lines of custom code. The custom code merely translates incoming messages like 'online' into commands that the API understands, e.g. '#cmd11 SET USERSTATUS ONLINE'.

6.2 Technology Enabling Applications

Conceptually more interesting than device specific connections are those that make a whole set of applications and devices available to toolkit users. In the following we present several of those approaches.

6.2.1 Example Projects Using the Particle Microcontroller Platform

The Particle microcontroller platform [Decker et al. 2005] is a rather generic platform to create applications using one or many sensor and actuator nodes. Its main advantage is the small footprint of its components. At the time of this writing, there exist five different nodes with different characteristics and intended use: Particles (low power radio frequency (RF) communication), µPart (micro sender-only node), cPart and Part-c (small, low-power RF), Blueticle (Bluetooth communication), zPart (supports ZigBee[66]). They use different versions of the PIC microcontroller family[67]. There also exist several add-on boards which offer a variety of sensors and actuators. The 'spart' sensor module, for example, provides 3D acceleration sensing, sensors for force, pressure, ambient light, infrared light, temperature and sound. More details can be found on the project page[68].

Programming Particles can be done in assembler or C. We used Particles in multiple projects, e.g. [Holleis, Kranz, Winter, et al. 2006], [Holleis, Kranz, and Schmidt 2005a], and [Holleis, Rukzio, et al. 2006b]. In order to support the development of applications using factory-made Particle systems, components developed by third-party groups, as well as custom-made add-on boards, it is important to get access to the communication system. For self-made sensor and actuator boards, it is still necessary to write or adapt code on the microcontroller. We provided some detail about how to enable systems such as the Particles to simplify application development with the help of integrated development environments like Eclipse in Section 5.3.3.

The ParticleGeneral stub has been written to generally connect the Particle system to the EIToolkit. It listens to all messages sent by Particles using the libparticle project[68]. The data is then repackaged and sent to the toolkit. Reversely, any message sent directly to the ParticleGeneral stub is transferred to the Particle communication system. The message type of the toolkit packet is used to specify the receiver ID and the Particle packet type.

[61] Skype phone and chat application; product page: *http://www.skype.com*
[62] Nintendo Wii game console and Wii remote control; product page: *http://www.nintendo.com/wii*
[63] Nabaztag ambient rabbit; product page: *http://www.nabaztag.com*
[64] Violet's API for the control of registered Nabaztag devices; developer page: *http://doc.nabaztag.com/api/*
[65] Skype API, Skype developer zone; developer page: *https://developer.skype.com*
[66] ZigBee Alliance; the protocol is often seen as the low-power successor of Bluetooth; project page: *http://www.zigbee.org/*
[67] Microchip PIC Microcontrollers; product page: *http://www.microchip.com/*
[68] TecO Particle communications library; developer page: *http://particle.teco.edu/software/libparticle/index.html*

The ParticleGeneral stub thus enables EIToolkit applications to communicate with Particle nodes and encapsulates all the necessary communication implementation details. Whenever a receiver ID is given, an acknowledged sending process is used to reduce the risk of packet loss. However, this behaviour can be easily switched off if desired.

6.2.1.1 Wireless Display Controller

For several projects (e.g. [Holleis, Rukzio, et al. 2006b], [Holleis, Kranz, and Schmidt 2005b], [Schmidt, Terrenghi, and Holleis 2007]), we built a small wireless display, see Figure 46.

Hardware implementation

Its main components are a Particle 2/29 base board and a custom-made add-on board. The add-on board can be attached to the Particle using a standard connector. The display is a Batron BT96040[69] chip on glass monochrome display with a resolution of 96x40 pixels. This is enough to show icons and small images as well as five lines of text with 16 characters each with a fixed font where each letter uses 5x8 pixel and one pixel space between two consecutive letters in a row.

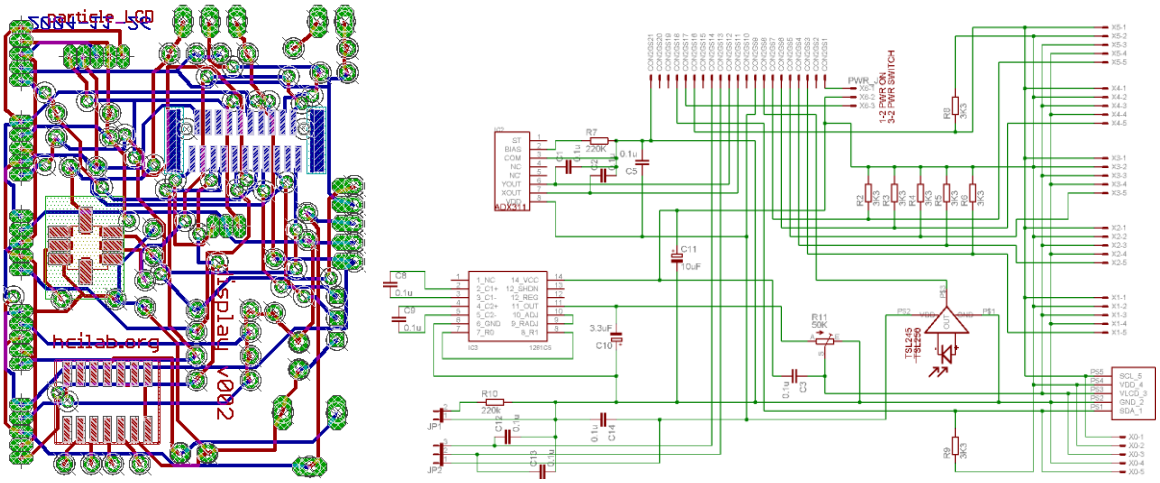


Figure 42: Lard layout (*left*) and schematic (*right*) of the add-on board designed using the CadSoft EAGLE[70] layout editor. In the schematic, one can see the 21 pin connector to a Particle on top, the connectors for the displays on the right and the acceleration sensor and the 12V generator on the left.

Seven of the nine resistors seen in Figure 42 are necessary pull-up resistors for the internal I²C communication from the PIC to the display. The other two are required by the ADXL311JE[71] accelerometer sensor to ensure proper and stable operation. The majority of the capacitors are used to stabilise power supply and reduce jitter on sensor output lines. The LTC1261CS[72] chip producing a negative voltage of 12V is necessary internally for the Batron display.

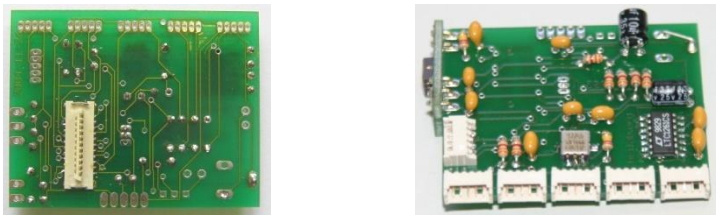


Figure 43: *Left*: On the back of the board, a 21 pin connector connects the board to a Particle node. *Right*: On the top left side, a small connector board containing a second ADXL311JE acceleration sensor is soldered orthogonally to the main board to achieve a 3D orientation measurement.

[69] Batron BT96040 display; product page: *http://www.data-modul.com/de/products/passive_displays/graphic_displays/bthq-96040av1-fstf-12-i2c-cog.html*

[70] CadSoft Online, EAGLE Layout Editor; product page: *http://www.cadsoft.de/*

[71] Analog Devices, ADXL311 accelerometer; product page: *http://www.analog.com/en/prod/0,,764_800_ADXL311,00.html*

[72] Linear Technology LTC1261 negative voltage generator; product page: *http://www.linear.com/pc/productDetail.jsp?navId=P1160*

To support applications relying on a long running time without changing batteries, the display power can be switched on and off by the microcontroller. Up to six displays can be controlled using one single board. As shown in Figure 44, one display can also directly be soldered on the board to minimise required space. A 21-pin connector (Figure 43 on the left hand side) connects a Particle base node with the sensor board, see Figure 44. The node provides power, controls the displays using I²C commands and reads and sends the analogue accelerometer values to a base station connected to a PC or LAN.

Figure 44: *Left*: these two images show the final setup with one display added in the front and a Particle with a AAA battery placed at the back. *Right*: example content of the screen.

Software implementation

To control the display, we built a small stub for the EIToolkit. This Batron display stub is an example of a device specific stub designed on top of the technology enabling Particle stub. It supports a list of commands (such as write some text, display an image) reacting on corresponding EIToolkit packet messages. The message type is used to specify the operation name as well as its parameters – except when one of the parameters is some text. Text is always specified in the packet message body. The character used to separate the name of the operation as well as the parameters is, by convention, the underscore '_'.

Message type: <*operation name*>_<*param*$_0$>_...<*param*$_n$>
Message content: <*text_param*>

As an example, 'hello' can be written when the following packet is sent to the stub:

Message type: *text_20_10*
Message content: *hello*

6.2.1.2 Applications and Studies

The desire to use such displays in longer term studies meant that we needed to give some of those to people's homes for some period of time (see [Schmidt, Terrenghi, and Holleis 2007] for detailed reasoning). For that, an electronically and electrically working version is not enough. We had to have the display placed in different orientations on a surface. We also wanted to increase robustness and, especially, increase the acceptability of the display with regard to persons without a particular background or interest in electronics or technology. This was possible by designing and creating a robust housing for the display, Particle and battery, see Figure 45.

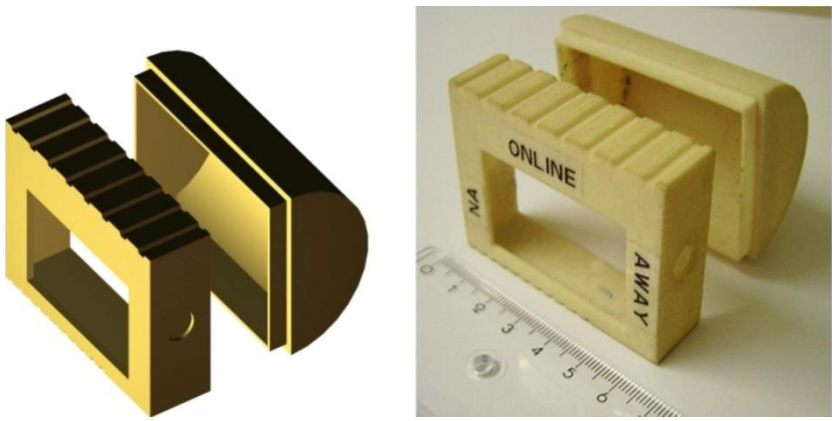


Figure 45: *Left*: 3D model of the display housing.
***Right*: materialised using a 3D printer (the ruler's unit is centimetre).**

Although, currently, few persons have direct access to a printing machine that creates tangible objects from 3D models, the value and potential of such technologies should not be underestimated in the process of prototyping applications. The casing we designed has several different tangible textures on its sides to provide a haptic sensation which side currently is on top. We also experimented with a rounded back. This allows two different modes of interaction, depending on the centre of gravity within the housing: when the centre of gravity is close to the display, the housing can be placed on each of its two long sides and it will not move. Moving, e.g., the batteries to the back, these two states are merged into one: the display tumbles over and rolls on its back. Depending on the application, this can be a desired behaviour.

In our description, categorisation and study of presence and instant messaging systems [Kranz, Holleis, and Schmidt 2006], we used this display design as a tangible input component to set the user's state in messaging applications like Skype (e.g. 'online' or 'away'). The right image in Figure 45 shows one possible labelling. The state in the application can be set using the Skype API. The orientation of the display, i.e. on which side it is put, or which side is on top, respectively, determines the current state. This setting has three main advantages over the standard mode built into messenger systems like Skype: the current state is visualised physically, the visibility on the desk makes forgetting to set one's state less probable, and changing one's state is much faster. Especially changing from any state to 'offline' can be done with a flick of the hand in the moment while leaving the desk; the display is just thrown on its round back, see the leftmost picture in Figure 46. Although there can be made no direct link between states like 'away', 'invisible', 'not available' and 'online' to the four orientations in the centre of Figure 46, the participants in an informal study confirmed our assumption of a strong affordance between setting the state to 'do not disturb' and placing the device with its display facing down. Unfortunately, a planned long-term study had to be cancelled because of too short battery life time.

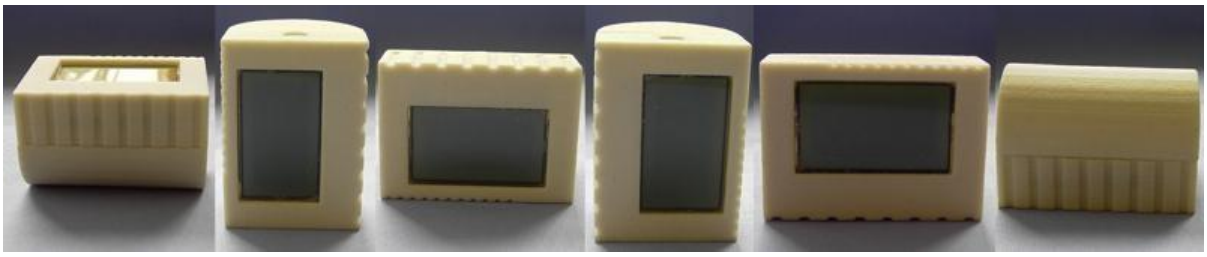


Figure 46: The six possible discrete states of the wireless display in a housing designed and fabricated especially for this board. These states can easily be recognized using the built-in accelerometers.

There are six different positions in which the display can rest (shown in Figure 46). Shifting the centre of gravity to the back as described above, this number of possible states can be reduced to four. Integrating a small motor that shifts a small weight could also be used to remotely change the centre of gravity and, e.g., remotely and physically switch from the 'online' state to an 'offline' state by letting the display tumble on its round back.

Environment-based Messaging

We used the display and the sensor board (often with slight variations) for various projects. In one application that also demonstrates the utility of the EIToolkit, we investigated the use of situated messaging [Holleis, Rukzio, et al. 2006a]. The key idea is to enhance current communication possibilities. Almost all of the messages that are sent using email, phones, mail, etc. are designed to reach one or several persons. In contrast to that, we propose to direct notes to a specific location rather than persons. This places the message directly into a context where it might make much more sense than if it were sent to one or several people. As simple examples, consider 'Don't eat the cake' posted to a display close to the fridge or requests like 'Can anyone please feed my fish?' with possible answers 'with what?', 'done' or 'already died'.

Often, messages are not of interest to people who are travelling elsewhere, especially when they are concerning coordinative or informative messages related to households or joint residences. A display at the house entrance can then take on the role of a reminder. Although it is still more appropriate to send important and time critical messages to the corresponding person, the system would save broadcasting a request to several persons if just the first person returning home would see it and could react on it.

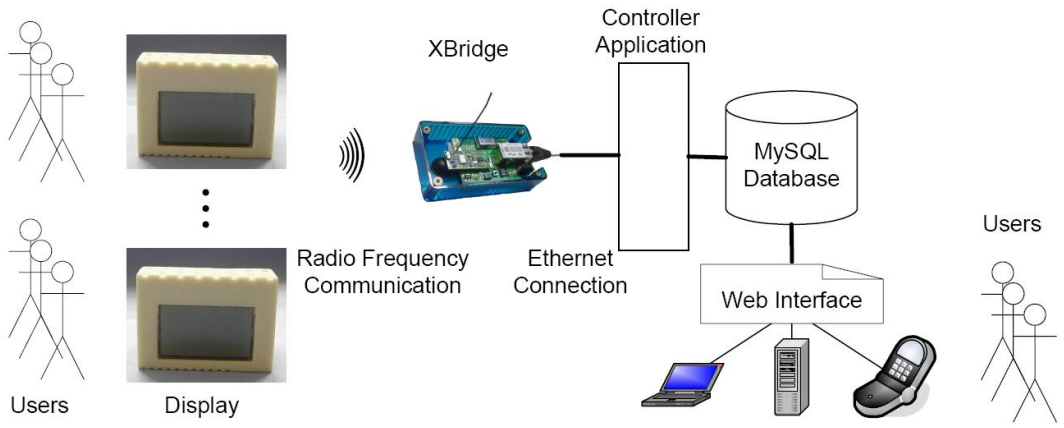


Figure 47: Abstract architecture of the system that uses small situated wireless displays as end points of communication. The 'controller application' acts as a mediator to update changes in the database of messages. These messages can be viewed and answered using the displays themselves. The creation of messages is mainly done using a web browser based interface. (Taken from [Holleis, Rukzio, et al. 2006a].)

Figure 47 shows the general architecture of the system. Large parts of the necessary software are actually taken on by the EIToolkit. Only the web interface and database connection had to be implemented independently.

One characteristic that differs from standard messaging systems is that the sender has no idea if or when the message is read. Without having undertaken specific studies, general discussions show that people seem to have a rather optimistic opinion with regard to the addressee receiving and reading an SMS message without much delay. Getting a notification when a message has been delivered to the partner's handset or the possibility to request an answer for important messages adds in this understanding. Whenever a message is sent to a place, there are fewer ways of finding out whether the message has reached someone (or even the 'right' person). For this reason, we added a rich set of input methods to the display without inadequately increasing the size. We thus discarded suggestions to add a small keyboard or keypad. A touch screen and a set of buttons also seemed to be inappropriate in some situations, e.g. wearing gloves when entering the house, having dirty or wet fingers in the bathroom or kitchen, etc. In the end we opted for simple gesture based input. Besides the orientation of the display as shown in Figure 46, additional ways of input have been devised:

- **Display number of unread messages**: the standard position with the display facing to the front shows the current number of unread messages.
- **Read a message**: this is meant to be an implicit gesture; picking the display up and holding it in some angle shows the first unread message; the active angle range has been determined by measuring the angle of the display with respect to the ground floor when people held it in the hand to read the contents of the screen
- **Proceed to the next message**: tilting the display towards the body as if emptying the contents through the display switches to the next unread message.
- **Answer a message**: there is no way of writing a verbatim answer. We use a version where the sender specifies up to three possible answers. By tilting the display to the left, to the right or turning it upside down, one of those can be chosen, see Figure 48.

We were aware of some issues in the current design. For example, there was no way of leaving a message unread, e.g. if it was intended for someone else. However, at this stage we were more interested in the general acceptability and intuitiveness of the interface. Thus, based on the prototype described in the previous section we conducted a small user study with 8 students of media informatics aged between 21 and 25. The goal was to evaluate the overall idea of environment-based messaging and the specific concept of using gestures to interact with small displays in this context. At the beginning, we thus explained a specific scenario of such a system in a flat-sharing community. We said that every person in the flat has a small display in a private room and, in addition to that, there are displays in all public rooms like kitchen, lobby, or bathroom. The study took place in two parts. In every phase there was a predefined sequence of messages provided by the display and the students had to set predefined answers.

Figure 48: Wireless display used for situated messaging. *Top left*: a question with three possible answers. *Top right and bottom left*: a user chooses between answers by tilting in the direction of the arrows. *Bottom right*: a selection has been made. Note that the display is always visible and facing the user.

First, the testers used the system without any knowledge about the provided functionalities and supported gestures. The goal of the first phase was to see how intuitive the gestures and the provided functionalities are. It turned out that most testers had considerable problems to figure out the provided functionality, especially with the intended gestures for interaction with the display: for example, some testers moved the display on the table because they thought that the arrows (see the picture in the top left of Figure 48) indicate a direction and not a rotation which had been our intention. Furthermore, their gestures were often performed too fast or short for our implementation and they were not able to set answers. Figure 49 shows some impressions.

Figure 49: Some issues from the study about gesture input. The person on the left felt uncomfortable with the gestures; the other person held the display without being able to see the text on the display any more. (Taken from [Holleis, Rukzio, et al. 2006a].)

In the second phase, we started by explaining what features were possible and which gestures could be used. We were interested in seeing how quick people could learn the gestures we devised. In this phase, everyone managed to answer the given questions correctly as planned. Thus, we generally conclude that the set of gestures we chose were not intuitive in the sense that people did not guess them at first try. However, it requires only little effort and a short time to actually learn how to use them. An informal discussion revealed some subjective opinions with respect to environment-based messaging. In general, participants of the study liked the idea but had difficulties in seeing the advantages in comparison with standard SMS sending. Still, people understood the idea of sending messages to places instead of to people and valued the concept. Additionally, we found an additional requirement to include SMS and larger displays for sending and receiving messages using the provided system.

One important outcome of the project was that, using the EIToolkit, we were able to split the implementation part such that the web interface and database connection could be implemented by a person without any knowledge about the technical details of the display or the implementation of the gesture recognition algorithm. It was also possible to adapt and refine the gesture recognition algorithms without incurring any changes in the user interface or back-end. We refer to [Holleis, Rukzio, et al. 2006b] for further project specific results.

Mobile Selection Techniques

To furthermore explore the possibilities of interacting with mobile devices in a certain environment, we conducted experiments comparing different input methods using mobile devices. In [Rukzio, Leichtenstern, et al. 2006], we studied the differences of three selection methods. Using a mobile phone, users were asked to select devices such as TV sets, stereos, or other home appliances either by touching them, pointing to them, or by scanning the environment and selecting one from a list. Touching was implemented using RFID tags with the PMIF framework [Rukzio, Wetzstein, and Schmidt 2005] and the scanning technique uses a simple Bluetooth query to retrieve a list of available devices. The pointing method was more difficult to implement. Users should be able to select a remote device merely by pointing with their mobile device towards this device. There are several ways to realise this. One is to use markers on the objects and image processing algorithms using the built-in phone camera. Together with the use of infrared communication (IrDA), it has the disadvantages of not providing direct feedback to the user and of incurring a counterintuitive way of pointing with a phone. As a remedy, we added a small laser pointer module to a phone and built detection modules attachable to any object (Figure 50). The detection modules trigger an event whenever the laser beam is detected and send a unique ID to a central server using Bluetooth. In order to distinguish between a hit by the laser beam and sudden changes in ambient light or occlusion of light by passing people, the laser was modulated with a certain frequency. This made it possible to nearly completely rule out any misdetection. Ma and Paradiso already introduced this concept and used different frequencies for different pointers and could even transfer some data during the pointing task [Ma and Paradiso 2002]. Our detection modules were connected to the main application using EIToolkit technology. This allows, e.g., to easily exchange the modules with any other technology without incurring any changes in the back-end. For further details and results, we refer to [Rukzio, Leichtenstern, et al. 2006].

Figure 50: Prototype of phone with attached laser pointer (*left*) and a light detection module (*right*).

6.2.2 Connection to Third Party Platforms and Components

Besides the various links to external hardware (e.g. the Particle platform), software (e.g. through the Skype or Winamp APIs), and communication protocols (e.g. HTTP) that have been mentioned before, some additional connections are worth mentioning to demonstrate the openness and compatibility of the toolkit.

The following is an excerpt of the list of technologies that are currently supported by the toolkit. For more details, we refer to the source code and documentation that are available for each of those in the toolkit.

- **Serial line communication**: sending and receiving data via serial line enables not only the communication with several prototyping platforms as described below but also the connection to several manufactured devices like set-top boxes or sensors.
- **Bluetooth communication**: enables communication with microcontroller nodes equipped with Bluetooth transceivers as well as with devices with built-in Bluetooth such as most current computers or phones.
- **Open Sound Control (OSC) and other protocols**: enables the use of various devices and programs based on these protocols, e.g. the graphical multimedia processing program Max/MSP[73], the IDE d.tools [Hartmann, Klemmer, et al. 2006] and many more (see a list on the OSC project page[74]).
- **Streaming of data**: see also Section 4.4; enables the use and creation of multimedia applications that continuously need to receive or send larger amounts of data since UDP messages are not suitable for streaming data. Support and sample applications have been implemented that use UDP control messages to negotiate a protocol and information about ports etc. and then allow streaming over TCP or RTP.
- **Keyboard event emulation**: a powerful concept to control nearly arbitrary applications (even those without a dedicated, open API) and also to switch between applications; obviously, allowing that kind of control is connected with high security and privacy risks.

One early, small-scale, embedded prototyping platform was conceived within the Disappearing Computer Initiative. The Smart-Its project aims at augmenting everyday objects with "sensing, perception, computation, and communication" [Gellersen, Kortuem, et al. 2004]. The nodes communicate either directly or via a radio module with each other or a PC over serial line. An EIToolkit stub can be used to handle the translation of incoming data through one of the PC's COM ports into EIToolkit messages and vice versa.

Other platforms that have gained more attention in the past years are Wiring[75] and Arduino[76]. Arduino boards are based on ATmega168 processors and offer several digital and analogue inputs and outputs. They are available in different form factors, supporting either USB, serial, or Bluetooth communication capabilities. They feature a custom made programming language and a graphical development environment based on Processing[77]. The target group includes hobbyists, artists, and designers. There are some projects that help users with only little background in electronics and hardware design to simplify and speed up the learning process. Fritzing, e.g., is a project from the University of Potsdam, Germany, that offers a graphical user interface for electronic design automation (EDA). A virtual Arduino board can be connected with a virtual representation of a breadboard (a board that allows simple adding and wiring electronic components). The software is then supposed to support the user in the process from the first designs until a finished printed circuit board (PCB).

Wiring and Arduino can both interface with each other or a PC using OSC over serial line communication. An EIToolkit stub can thus be used to mediate between the boards OSC messages and the toolkit's UDP messages. The UNIX style component referencing mechanism of OSC neatly fits into the message type concept employed by the EIToolkit. A more detailed description of how to connect to those platforms using OSC can also be found on the d.tools project page[78].

[73] Cycling '74, Max/MSP development environment; project page: *http://www.cycling74.com/products/maxmsp*
[74] Open Sound Control, OSC; application areas page: *http://opensoundcontrol.org/osc-application-areas*
[75] Wiring, hardware prototyping platform; project page: *http://wiring.org.co*
[76] Arduino, hardware prototyping platform; project page: *http://www.arduino.cc*
[77] Processing, programming language; project page: *http://www.processing.org*
[78] d.tools, prototyping environment; project page: *http://hci.stanford.edu/dtools*

The following list is a sample of current rapid prototyping platforms that support people without extensive technological background in creating functional physical prototypes:

- **Gainer** [Kobayashi et al. 2006]: the hardware consists of a Cypress microcontroller, a USB-to-UART bridge and an I/O module board. The connection to development applications such as Flash, Max/MSP, and Processing is realised through serial line. As such, the EIToolkit has a built-in stub for communication.
- **Phidgets** [Greenberg and Fitchett 2001]: a commercially available plug and play platform based on configurable devices communicating by USB. On the PC side, application support is offered for .NET, Visual Basic, LabView, Java, Delphi, C, C++ and Python. The d.tools project group implemented an OSC wrapper for Phidgets which makes it available to the EIToolkit, see [Hartmann, Klemmer, et al. 2006].
- **DaKa**[79] ('DatenKäschtli'): a recent, similar project by the Zürich University of the Arts that offers a low-cost, low-level USB module with several input and output pins. It also uses serial data communication and provides development interfaces for several languages such as C/C++ and Python. For that, a custom EIToolkit stub would have to be written.
- **CREATE USB Interface**[80]: another project providing a USB controller module. This particular board is based on a Microchip PIC18F4550 microcontroller. There exist wrappers to access its input and output capabilities from the EIToolkit, e.g., using OSC. One interesting aspect that distinguishes it from other works is the prototyping area built into the standard modules.
- **MAKE Controller Kit**[81]: an open source board based on an Atmel ARM7 SAM7X processor. A pluggable application board offers additional application-level features such as various networking interfaces (LAN, CAN (Controller Area Network), USB, etc.) and additional internal storage. The main board offers access to most low-level signals and allows attaching a variety of sensors and actuators. Applications can be developed in .NET, Processing, Java, Max, PureData[82], Adobe Flash, or C/C++. Connections to a PC and the EIToolkit can also be made using serial line communication.
- **Scratch Sensor Board**[83]: the Scratch programming language is a free graphical programming environment targeted at young people developed by the Lifelong Kindergarten group at the MIT Media Lab [Monroy-Hernández and Resnick 2008]. The Scratch Sensor Board has recently been developed to extend the possible applications to resources like microphones, light sensors, sliders, etc. It additionally has four generic pins that can connect the board to other external sensors using alligator clips. The board also communicates using the serial line protocol and can thus easily be connected to the EIToolkit.
- **Linux operating system based boards**: in addition to the specific boards presented above, many devices are available that run some version of a UNIX / Linux operating system. An example is the Fox Board LX832[84] running Linux. It offers two USB and an Ethernet interface in addition to four dozens general purpose I/O lines usable for communication using I²C, serial or parallel ports or sensor and actuator control. Depending on the model, EIToolkit compatibility is ensured through serial line access or can be easily created by, e.g., running a webserver on the module.
- **Voodoo-I/O (Pin'n'Play)**: as has been described earlier (see page 57), Voodoo-I/O is a platform developed by the Embedded Interactive Systems group at the Lancaster University Computing Department [Villar and Gellersen 2007]. The general idea is to enable users to completely customise a physical user interface, even during run-time. An interface is decomposed in its most basic components, i.e. sliders, knobs, buttons, displays, etc. Those parts are available as very small independent devices and are simply attached to a special material offering both power and communication facilities. Thus, interface items can be placed, grouped, and moved wherever desired. An EIToolkit stub has been created that packs information sent by the Voodoo-I/O input components over a USB connection into an EIToolkit message and can send commands to output components. As a sample application the events generated by a set of buttons were made available as keystrokes on a PC. A user can thus place buttons at arbitrary positions on a projected display equipped with the Voodoo-I/O substrate and control, e.g., the visible area in Google Earth[85].

[79] DaKa, hardware prototyping platform; project page: *http://interaction.zhdk.ch/projects/daka*
[80] CREATE USB Interface (CUI); project page: *http://www.create.ucsb.edu/~dano/CUI*
[81] MAKE Controller Kit; project page: *http://makezine.com/controller*
[82] PureData (Pd), dataflow programming environment; project page: *http://puredata.info*
[83] Scratch Board, hardware prototyping platform; project page: *http://scratch.mit.edu/pages/scratchboard*
[84] Fox Board LX832, see ACME Systems product page: *http://www.acmesystems.it/?id=4*
[85] Google Earth, maps and satellite images; product page: *http://maps.google.com*

6.3 Data Visualisation Tool with Exchangeable Components

Often, a first step in developing applications depending on sensor input and actuator output is to understand the values that are being generated. The correct use of touch or accelerometer sensors, for example, may not be obvious from the beginning. Furthermore, one component can be slightly different to another component even if they are from the same type. This can be due to differences in the manufacturing process, different ways of accessing and interpreting the values, updated firmware, or external influences like temperature and humidity.

Some applications are even little more than displays of some sort that show information from different sources, e.g. an umbrella stand with an integrated light issuing a reminder or warning depending on the current weather forecast [Schmidt, Kranz, and Holleis 2005], or picture frames that present information about other people's opinions [Holleis, Kranz, and Schmidt 2005b]. They completely rely on a well chosen portrayal of gathered information. However, the development of applications that go further than that also needs a good understanding of what information is available. To achieve that, a versatile tool for visualising a variety of dynamic values is of much value. On the other hand, producing a stream of values, be it as input from the end-user or as a simulation of existing or envisioned sensor input, is also often necessary in order to test and build a working application.

DATAVIS[86] is a set of components that builds on the EIToolkit communication structure. The main visual application area serves as a simple container in which an arbitrary selection of components can be inserted. Each component serves either as a virtual sensor and visualises one piece or a stream of data (e.g. a bar graph) or acts as an actuator and provides a visual handle to produce one or a sequence of values (e.g. a slider).

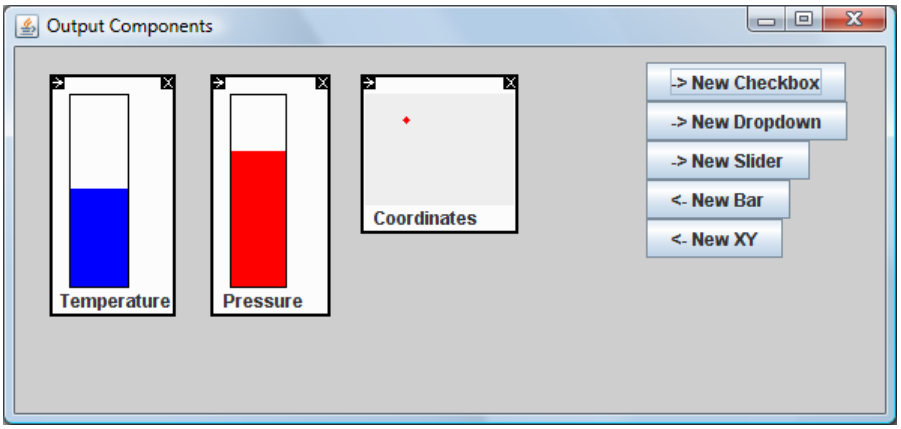


Figure 51: Sample components visualizing different types of parameters. The bars on the left are useful for single discrete or continuous parameters. The visualisation in the centre shows the relationship between two parameters, e.g. x and y coordinates. Scales can be omitted for qualitative analyses.

Figure 51 shows a sample of possible output components. They are used to visualise information transferred using the EIToolkit. An arbitrary number of components (of the same type or of different types) can be displayed at the same time. They can be arranged and grouped to one's liking. To connect a component to data available in the toolkit, it is normally enough to specify the message type of messages passed through the EIToolkit. If a component accepts more than one message type (for example displaying a 2D point for two streams of 1D coordinates), a context menu allows quickly setting the specific inputs. All available properties can quickly be altered through context menus including, for instance, the colour of the data points, potential minimum and maximum values, and display of the current value as text.

The current visualisation concentrates on a manual method where the user chooses the messages of interest and the accompanying type of visualisation. Another possible approach would be one that we applied in the concept for the attribute inspector in the Gravisto graph visualisation toolkit [Bachmaier et al. 2004]. The inspector presents a visual representation of the properties (called attributes) of graph elements. These properties are organised in a hierarchical structure. A node can, for example, have a label described by a label attribute. This is comprised of a string, possibly an icon as well as information about positioning. The position itself consists of several attributes like relative coordinates. Such a relative coordinate is then finally represented by two numbers

[86] DataVis, data visualization components for the EIToolkit; project page: *https://wiki.medien.ifi.lmu.de/HCILab/DataVis*

which are stored as primitive types (i.e. double values). In the same way, each EIToolkit message could be associated with a specific type. When data of some type *A* is passed through the messaging system, all available types are searched whether they are suitable to represent this data. If none is found, the type hierarchy is used to decompose *A* into its subcomponents and the procedure continued recursively. In this way, the most suitable representation of the data structure can be found automatically. However, the automated process implies issues like information overflow, rapidly changing visualisation structure, and reduced control over the representations.

One important point became clear after using the DATAVIS system for several smaller projects (mainly involving a range of simple sensors like light or touch sensors [Holleis, Huhtala, and Häkkilä 2008], but also more complex ones like steering wheels in ongoing work about in-car user interfaces [Kern et al. 2008] or multi-touch systems based on, e.g., [Döring and Beckhaus 2007]): the diversity of possibilities to pass even one and the same value complicates the strive for a simple connection between data and visualisation. Those problems are well-known and manifold solutions have been proposed starting from defining a standard protocol to sending accompanying type information (see for example the OSC protocol or more complex data transmission and message passing descriptions like CORBA). However, even in settings where multiple types of data have to be transmitted (e.g. information about content, position, recipient, and security breaches of an intelligent transport box [Müller, Holleis, and Schmidt 2007]), the structure and content of messages is rather well defined. Therefore, one possible approach of tackling such format issues is to let users choose between several ways of interpreting the data as well as a small scripting window to manually extract the information from within more complex messages like encrypted data.

Currently we use the flexibility of the EIToolkit to dynamically insert stubs at any point in time and simply write another stub that translates the data from one representation to one that the current visualisation components understand. In order to make the visualisation tool more flexible, we added support for regular expressions to define input and output.

Note that the separation into input and output components is not necessarily meant to be on a component level. Most components that allow the user to specify some value, e.g. a slider or a checkbox, are also suitable to produce a message containing the same value. As another example, the coordinate component in Figure 51 that shows two input values as *x* and *y* coordinate can also be used to accept mouse input and produce the appropriate coordinate events.

Technically, the system is realised by using interfaces named InputComponent and OutputComponent for each type of component. Thus, components that exhibit both functionalities can simply implement both interfaces. The DATAVIS application then uses the reflection API of Java to locate and load available instances of those interfaces. For each of those, a button is generated that lets the user add instances of this component to the main interface. The button also gives a hint whether the component is used as input, output, or both. Components can be placed and moved freely within the window.

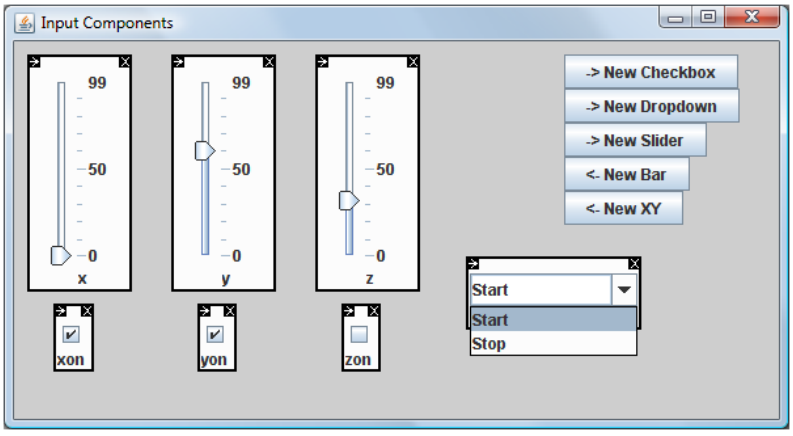


Figure 52: Generic components used for data input (sliders, checkboxes, and a combo box).

To quickly emulate sensors or generate events, output components can be used to produce data in a visual way. Figure 52 shows sliders, checkboxes, and a drop down box where one can select from various values or custom values, all of which can generate and send EIToolkit messages. The example in the figure shows a simulation of a 3D acceleration sensor where each of the axes can be switched on or off with checkboxes and its values set using sliders. A combo box element is used to send dedicated start and stop events to the processing application.

Input and output components can be added, removed, freely positioned, and adopted through properties at any time during compile or runtime. Since all components extend the standard Java Swing graphical panels, they can be used in any existing or new application based on Java Swing. In the spirit of the EIToolkit's several basic programming platforms, future work includes providing similar tools in other languages.

6.4 Wearable Computing

One specific area in which such a toolkit and the tools described before can be widely employed is wearable computing. It is in fact particularly suitable for this approach since it makes use of one or more controller units that process and communicate data from a multitude of different sensors and actuators. Especially in the early stages of trying and designing wearable controls, these components often need to be moved to different places, exchanged with other components (e.g., newer versions), and simulated (e.g. when an expensive control is not yet available or a particular use-case scenario difficult to provide).

In two projects in close collaboration with the User Experience Group of J. Häkkilä at Nokia Research, Finland we looked into where on clothing and wearable accessories areas can be identified to be used as input location with a special focus on touch input and user acceptance (Section 6.4.2). We also extended input methods on mobile devices with an extra dimension by integrating touch sensors into a phone keypad (Section 6.4.3). Parts of this section are closely based on [Holleis, Paasovaara, et al. 2008] and [Holleis, Huhtala, and Häkkilä 2008].

6.4.1 Related Work within Wearable Computing

Although electronics can be integrated into textiles, it is still difficult for garments to have all the characteristics of usual clothes as well as a network of sensors, input and output features, wireless connection, and power to run for a satisfactory period of time. However, there have been enormous advances in textile fabrication processes as well as in the understanding of integrating conductive yarn. [Marculescu et al. 2003] and [Post et al. 2000] give in-depth overviews about the subject of smart textiles. They cover early work like the Georgia Tech Wearable Motherboard described in [Park, Mackenzie, and Jayaraman 2002], as well as current work like the wearable digital MP3 player [Jung, Lauterbach, and Weber 2002]. The authors point to a number of challenges and visions as well as technological advances that drive research in the wearable computing community.

In the following, we deliberately leave out a more detailed treatment of the huge range of wearable devices that are either carried or in some other way obtrusively alter the way people wear clothes. We refer to overviews such as [Rhodes and Mase 2006] and the review [Randell 2005] for pointers to such projects and products, and mention those only if they have a distinct relation to our work.

A general overview and collection of issues, end-user needs and appropriate technology has been assembled through several years of experience with designers, technologists and industrial collaborators by [McCann, Hurford, and Martin 2005]. Most research done in the field of smart clothing can be split into four categories. Much effort has gone into a technology driven approach that will enable systems and applications to be built. This includes research in the miniaturization of devices and power supplies, wireless communication methods as well as the integration of conductive wires into available clothing. Other researchers and product designers have focused on providing ideas and solutions for very specific areas of applications and usage settings. This includes specific environments as well as particular tasks. In recent years, development support of wearable systems and smart textiles has received increased attention. This includes actual hardware components as well as software frameworks that aid in the connection and interaction of such components. Only few results can be found about actual studies that consider the usability and the acceptance of such wearable controls in general, see section 'Studies with Wearables' below. The remainder of this section treats each category in more detail and gives further pointers to corresponding research projects.

Technology Driven Projects

A huge step towards the goal of including electronic components like sensors, actuators and computing platforms into clothing is achieved when these elements can be integrated into the sewing process. The second important quality of such components must be to seamlessly integrate into the comfort and original qualities of the garment. One way of encasing electronic circuits in a way that they are protected from power and data lines, do not depart from the design of the clothing, and still enable the normal use like folding, is described in [Hannikainen, Mikkonen, and Vanhala 2005]. Button shaped casings are used to hide electronics, provide connections that can be sewn, and preserve a cloth-like look as buttons often appear for fastening pockets or even for mere design purposes.

[Linz et al. 2005], show how to use flexible electronic modules and a way of connecting them with conductive yarn. The system allows using common fabrication processes and does not change the flexibility and feel of the textiles used. Although the requirements put on the yarn (e.g. that it can be sewn by a machine and must be conductive on the outside) place some restrictions on the material and their use, the reliability tests done by the authors show promising results. Metal wires are rather inflexible, however, and can reduce the wearablitity of the garment. Several companies now offer conductive yarn for little money. SparkFun offers one, e.g., which has a resistance of about 270 Ohms per metre[87]. The paper ‘Conductive Fabric Garment for a Cable-Free Body Area Network’ describes a setup of a cable-free body area network which is based on conductive fabrics that are supposed to behave like normal fabrics [Wade and Asada 2007]. The authors provide a 2-wire bus transmitting power and data to sensor nodes. These can be located anywhere on a piece of garment and can take power from a single battery. Details about the underlying two wire bus system can be found in [Wade and Asada 2004]. Since one of the central issues in such conductive circuits is power loss because of the wiring, the placement of nodes will in practice be restricted. Wade and Asada set up a model to predict the resistance of large parts of a body shaped piece of textile. This can give hints as to where nodes should be placed most effectively.

A closer presentation and examination of different types of yarn that can be used for such purposes as well as their properties regarding knitting, yarn control, relaxation, and geometry of knitted structures, is found in [Power and Dias 2003]. These projects provide enabling technology for creating smart textiles, especially for mass production scenarios. They can all be applied to several parts of our setups. [Orth, Post, and Cooper 1998] show what technology was already available in 1998 with respect to such fabric computing interfaces.

Projects Targeted to Specific Tasks or Settings

There are many more projects for specific application areas that use some instance of wearable computing than can be recounted here. However, the vast majority of those use larger additional devices like wrist keyboards or focus much more on the output or implicit input side. This includes a variety of head-up displays, headsets, watches or other body-worn displays well-known for several museum and tourist guide scenarios. Implicit input is implemented by motion trackers and systems using health monitoring sensors. Explicit and direct input of users has mostly been studied for various methods of text input. One of the famous examples is the Twiddler system which is a one-handed chording keyboard that has been demonstrated to outperform the multi-tap standard for mobile text entry [Lyons, Plaisted, and Starner 2004] (but requires a longer learning time).

Concentrating on current mobile phones, there are currently two ways of data input: using buttons or a touch screen. The first approach provides tactile feedback, the second usually not. Some effort has been made in combining these by adding tactile feedback to touch screens (e.g. [Poupyrev and Maruyama 2003] who use a small vibrating actuator placed behind small displays), but the affordance and feeling of a button can only be simulated this way. To overcome the limitations of small keypads, several ideas have been followed including speech input and auditory UIs [Brewster 2002], but their usage can be limited, e.g. because of social situations. Gesture input has also been suggested, but such methods suffer from problems in recognition as well as in social acceptability [Ronkainen et al. 2007]. Obviously, attachable or fold-out keyboards could be used but these negatively affect size and mobility.

[87] SparkFun conductive thread; product page: *http://www.sparkfun.com/commerce/product_info.php?products_id=8544*

One approach that enhances conventional keypad input is to add a touch sensitive layer on top of the buttons. In this way, touches and slight presses on the buttons can be detected while keeping the normal look and feel of the buttons. This enables additional functionality and interaction types to be added to standard applications. In [Rekimoto, Ishizawa, et al. 2003], the authors first presented a working prototype of a keypad augmented with capacitive sensors and introduce the notion of 'previewable user interfaces'. Similar to tooltips on PCs, touching a button can give information about what would happen if this button was pressed. The authors provide several application ideas; however, no evaluations with users have been reported. Moreover, there is no working version with a real mobile phone. [Clarkson et al. 2006] add a layer of piezoresistive force sensors, one below each phone key. This enables them to continuously measure force exerted on each key. It therefore adds a continuous dimension to each key and enables applications such as smooth zooming into images proportional to the force on a button. The authors present, among others, an application for text input that uses a technique similar to that presented in [Zeleznik, Miller, and Forsberg 2001] simulating a tri-state button. A disadvantage of the force sensors is that, below a certain amount of pressure on a button, the sensors do not register the touch. They also cannot distinguish between pressure in a pocket from the touch of fingers.

PreSenseII further develops this concept by using a touch pad and adding force sensors below it [Rekimoto and Schwesig 2006]. Thus, a continuous space in three dimensions can be achieved. However, this looses the affordance and feeling of buttons and tactile feedback has to be simulated with a piezo-actuator. Again, no formal studies have been undertaken to get feedback from users. Although the touchpad clearly gives more freedom and possibilities, we argue in favour of the affordance and clear separation of buttons and see an advantage in leaving the current user interface unchanged to ease the process for users of getting used to such a new way of interaction (see Section 6.4.3).

Other modern data input methods that can be used in conjunction with mobile phones but also in different settings include speech recognition and data gloves equipped with sensors to retrieve hand position and detect gestures (see, e.g., the glove of [Liu, Liu, and Jia 2006] which is used specifically for text input). Both approaches have advantages but also, among others, the disadvantages that they often lack social acceptance, need heavy processing and power, are still unreliable, and impose not negligible amounts of training on the user.

[Rantanen et al. 2000] focus on the design for a specific application domain and give an example of the design of a wearable interactive system for a particular type of environment. They introduce a vest and a combined input and output device (appropriately called the YoYo interface) which is especially designed for an arctic environment requiring, e.g., the use of gloves during the interaction.

A slightly different approach is taken by B. H. Thomas who demonstrates the e-SUIT in [B. H. Thomas 2002]. He shows how to augment a standard business-type suit with input and output features in a way to minimize the social weight, i.e. the "measure of the degradation of social interaction that occurs between the user and other people caused by the use of that item of technology". This means that the location of buttons and other controls is not controlled by aspects such as ease of finding, accessing or using them, but more from a social acceptability point of view. However, as stated in the paper, issues of look and feel, location, etc. are open research questions. The placement of interaction items like buttons has been based on the author's judgment. We extend this work by presenting study-based results for the acceptance, location and type of controls to be used.

Development Support

To bring the development of wearable systems forward and to enable more people to implement their own ideas of smart textiles, it is vital to provide frameworks and toolkits. These can then be used to reduce the complexity of building such systems. In order to rapidly construct textiles using simple input, output, and processing units, one can employ the construction kit for electronic textiles recently published in [Buechley 2006] and [Buechley, Eisenberg, et al. 2008]. It offers temperature, light and pressure sensors, LEDs and small vibration motors, as well as a programmable microcontroller. The components are constructed in a way that helps connecting them to conductive wires. It is as such similar to the tools that we used ourselves to create the settings in our studies. The toolkit's rather bulky and standardized components do not seem to offer easy customization possibilities of their appearance, however. The toolkit includes a pressure sensitive button but does not use any touch sensing

components. Obviously, the highest obstacle in creating a running application is still the programming of the microcontroller to process all the data and implement the logics. However, it won't take long until such assistive systems like the d.tools project [Hartmann, Klemmer, et al. 2006] include hardware support for the special requirements of wearable computing. Other projects, e.g. [Witt, Nicolai, and Kenn 2007], focus more on a model-driven approach and abstract descriptions to enable the semi-automatic generation of different user interfaces for display devices like PDAs or head-up displays.

Studies with Wearables

Some interesting projects have emerged that can assist those wanting to develop or evaluate ideas and systems in the wearable domain. [Knight, Deen-Williams, et al. 2006] present a methodology to evaluate wearable systems. Among other things, they concentrate on physiological and biomechanical aspects as well as the comfort of wearing or carrying such a system. The last aspect is analyzed in more detail in [Knight and Baber 2005] who also give tool support in assessing the comfort of large wearable computers. More focused on the development process, [Hurford, Martin, and Larsen 2006] provide the interesting result from a survey that the user-centred design approach is still only rarely employed in the field of wearable computing.

Of value for the work described here are also especially publications that concentrate on studying the location of controls on the human body. The well-known analysis from Gemperle et al. shows details about where on the human body solid and flexible forms can be attached best [Gemperle, Kasabach, et al. 1998]. This study focuses on places that are most stable during motions. The authors of [Thomas et al. 1999] briefly describe a study to find out where people would want to place a small touch pad during different activities like sitting and standing. The results do not seem to be entirely conclusive but suggest, e.g., that the front of the thigh is a good position for placing such an input device. The main driving factors of the study to find optimal placements of optical input devices on the human body described in the technical report [Mayol, Tordoff, and Murray 2001] are the possible occlusion by the wearer, a clear sight to the main workspace of the user, and movements of body parts that can influence the retrieved image. Since we do not use cameras for input and focus on public, every day use in our application scenarios, aspects like reachability and social acceptability are more important in this context.

Even though the EIToolkit currently has the disadvantage of routing all messages and events through a common infrastructure, we argue that it can be well applied in this area of research. It offers the possibility of using all components available for the toolkit for no additional cost in terms of implementation effort. It also allows easily exchanging sensor and actuator technology without influencing the application itself. This supports quickly exploring a larger amount of design possibilities in a shorter time.

6.4.2 Touch Input on Clothing

One issue that arises in the domain of wearable computing is command input. Many applications in that area are supposed to augment or support specific activities. A general guideline for such projects is to seamlessly integrate technology such that the main activity is not disturbed. This means that feedback and output from, but especially input to the system, has to be integrated in a user friendly and unobtrusive way. One proposed solution is speech input technology developed to offer hands-free interactions. However, besides accuracy issues, the useful situations are often limited in practice, e.g. due to social situations where silent use is often preferred.

Thus we looked into manual input and decided to concentrate on touch controls that can be integrated well into existing as well as specifically designed clothing [Holleis, Paasovaara, et al. 2008].

Advances in computing technology have led to computing devices in different forms to be present in our everyday life. Mobility is a phenomenon that has taken giant steps during the recent decade e.g. due to miniaturization, improvements in energy consumption and developments in communication infrastructure. The use of mobile computing devices such as mobile phones, mobile music players, and PDAs have become part of common practices, and overall omnipresence of the technology is gradually approaching the vision of Ubiquitous Computing [Weiser 1991] by becoming a truly integrated part of our society and personal life.

Despite of the high adoption rate with mobile computing, input technologies have not yet evolved to an optimal level in the means of usability. Almost without exceptions, the manual control of a mobile device happens with buttons integrated into the gadget. This results in a large number of small buttons placed in a small physical area. Such input devices can sometimes be difficult to use especially if they are placed, e.g., in a pocket or handbag. To overcome the problem, remote controllers and headsets with input buttons have been introduced. However, these need to be remembered by the users to be taken along and their physical shape significantly influences the comfort of carrying them. Additionally, if the control has to be held in one hand and controlled by the other, this leaves no hand free for other activities. Other suggested controls are integrated, e.g., in headphones or cables. This means, however, that the location of the controls is defined by the device and not by usability aspects.

Wearable computing offers an interesting approach for integrating new input methods to mobile computing technology and hence shows potential in mobile HCI. The term 'wearable computing' generally refers to a small computer attached to its user in some way other than holding it. The main characteristics of wearable computers are that they are always accessible by the user and that the user can continue various activities while using them. Wearable computing offers large areas available for placing input controls and can embed controls into users' normal clothing. It can utilize smart textiles which constitute an underlying technology for wearable computers (e.g. power and data lines integrated into clothing). An ultimate goal of wearable computing is that all technology is completely and seamlessly integrated into clothing or accessories like bags or standard glasses.

Research on wearable computing has so far concentrated on demonstrating new concepts and applications. Systematic studies on the performance, preferences, expectations and acceptability of such technology have been rare. However, studying these aspects is important for creating usable and successful products, and can offer valuable insights for future designers of wearable computing applications. As has been described in [Rantanen et al. 2000], besides the technological and material aspects, a piece of intelligent clothing also has to provide the esthetical and functional properties expected from clothing in general.

In this section we present different wearable prototypes controlling mobile technology that we developed in order to study the user experience with that technology. We present results of an extensive, two-phase user study concentrating on usability and acceptability issues with the technology, and discuss about general guidelines and lessons learned with designing wearable input technology.

6.4.2.1 Developed Prototypes

As has been mentioned in the previous section, there are a few projects that attempt to theoretically describe users' views on different aspects on wearable computing. Since, however, people's knowledge about the possibilities and opportunities that wearable computing offers is still restricted, we strongly argue in favour of giving people demonstrators and prototypes at hand. This can significantly increase the precision and validity of people's responses and is necessary to effectively find out about users' opinions, fears and acceptance of such ideas. We thus report on the set of prototypical devices and garment we built. These can be seen as an enabler for testing, demonstrating, and studying such applications.

Base Technology

The four prototypes developed for these studies all rely on the same type of technology although slightly different sensors and controller boards can be used for different packaging restrictions.

The underlying principle of the touch sensors we used is capacitive sensing. This technique has been in use for a long time already, one of the notable early uses being the musical instrument invented by L. Theremin in 1919. Since then it has already been used in touch sensitive tablets, e.g. [Lee, Buxton, and Smith 1985] and nowadays the controls of modern stoves and washing machines are often equipped with touch controls based on capacitive sensing. Without going into too much detail, the method relies on generating an electric field the strength of which is measured. When an object such as a finger interferes with the electric field, the capacitance measured at the receiver changes and can be used to detect proximity or touch.

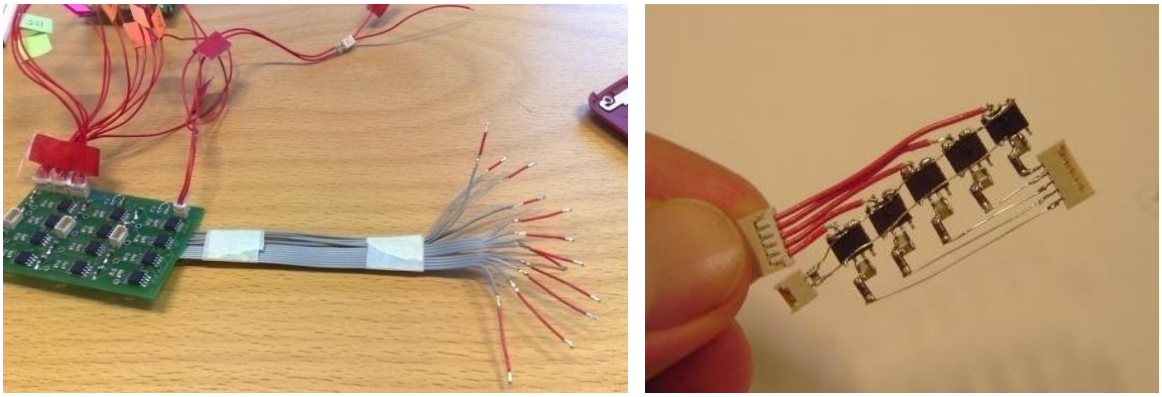


Figure 53: Two versions of capacitive sensing circuits. *Left*: A larger board with twelve sensors. *Right*: Flexible soldering of five sensors with a thickness of only 1.5 mm.

The general hardware for our prototypes and studies in the area of wearable computing consists of three parts. The main part is a custom-made circuit board housing sensor chips to detect touch on its connected electrodes. Different layouts and versions have been built. One uses twelve QT110[88] chips that can each be used to detect touch or proximity at one electrode. The rather big board shown in Figure 53 on the left is well suitable for applications where a small base size is not essential. It is easy to connect and the sensitivity can be adjusted to one's needs. Another version, shown in Figure 53 on the right, has been optimised towards a small size and is not built on a rigid PCB to allow it to be flexible in order to ease its integration into clothing. It uses only five chips and can therefore only detect touch on five electrodes. However, several of these can easily be used in parallel. A third version, shown in Figure 54, uses a single chip from Analog Devices, AD7142[89], that offers support for up to 14 sensors. It needs to query the sensor inputs sequentially but still manages to provide a complete update each 36ms. Besides a small footprint, it has built-in algorithms for automatic environmental compensation and adaptive sensitivity levels. A disadvantage for prototyping purposes is that soldering is very hard without professional equipment.

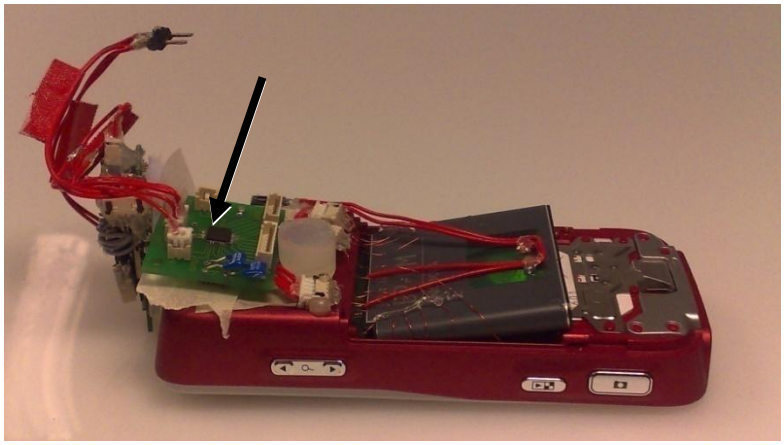


Figure 54: Analog Devices' AD7142 chip with up to 14 capacitive sensor inputs. Here, it is displayed attached to a phone and powered from the phone's battery for the project described in Section 6.4.2.

The second part of the hardware is responsible for communication. We use a Linkmatik 2.0 Bluetooth transceiver[90] (Figure 55, right). It can be controlled by a simple serial protocol on the local side and also provides a serial line over Bluetooth which makes it very easy to connect to external devices like a PC. The EIToolkit's stub for serial line communication (see Section 6.2) can be used for this purpose. Program logic, i.e. initialisation

[88] Quantum Research Group, QProx company page: *http://www.qprox.com/*
QT110 Family QTouch Sensor IC, product page: *http://www.qprox.com/products/page-16035/qt110.html*
[89] Analog Devices AD7142 programmable controller for capacitance touch sensors; product page: *http://www.analog.com/en/prod/0,,760_1077_AD7142,00.html*
[90] RF Solutions Ltd, Linkmatik 2.0 Bluetooth Transceiver
product page: *http://www.rfsolutions.co.uk/acatalog/LinkMatik_2.0.html*
data sheet: *http://www.flexipanel.com/Docs/LinkMatik%202.0%20DS379.pdf*

of the sensors, interpretation of their output, and generation of events using the Bluetooth transceiver is done by a PIC 18F2550 microcontroller (Figure 55, left). It is a low-power controller that offers support for several protocols like USB or I²C. This renders it a valuable fit for applications in the wearable domain where power consumption is a big issue and many different sensors and actuators are potentially in use. Sensor values are read at a frequency of about 25 Hz. Whenever a change in the values is registered, an event is generated and sent through the attached Bluetooth module.

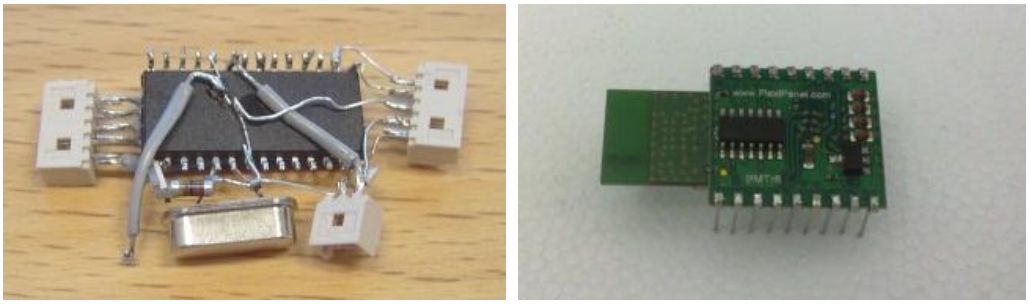


Figure 55: *Left*: Microcontroller with connections for power and 2x5 input / output connections. *Right*: Linkmatik 2.0 Bluetooth module. Both measure about 30x25mm.

The third part consists of the electrodes used to lead the touch sensitive areas to the desired locations. Characteristics of those electrodes such as shape, colour, and size can be determined on a project to project basis. Some possible implementations are introduced later and can be seen in Figure 56.

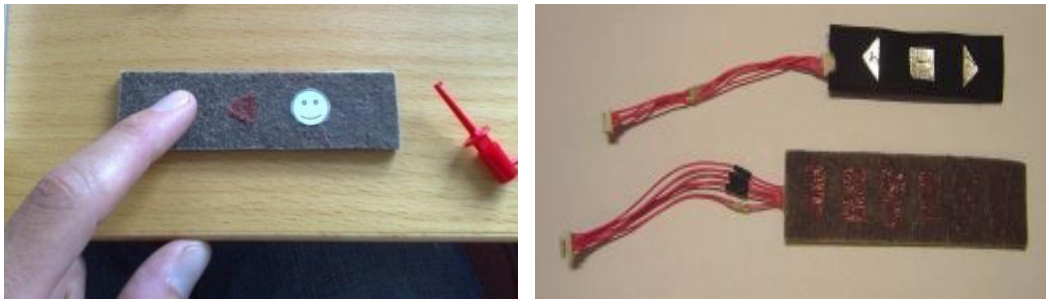


Figure 56: Possible implementations of touch buttons; the conductive yarn can be sawn in specific shapes, thin layers can be placed on top, or conductive tape can be used to enhance sensitivity (top right).

This platform enables designers to quickly add touch controls to nearly arbitrary clothing, accessories and other devices. The only need is to integrate areas with conductive material (which is available as strings, sheets, etc.) and attach them to the platform. The microcontroller and sensors can be powered using a variety of batteries ranging from about 3 to 12 Volt. Our prototypes either use an external small battery pack or are powered directly from a mobile phone's battery. An abstract overview of the system is shown in Figure 57. Any device that supports the serial communication over Bluetooth protocol can then wirelessly connect to the sensor board. We implemented applications that react on touch input running on Series 60 phones, Nokia N800 internet tablets and PCs using Java ME and Python for Series 60, Python for Unix systems, and Java / C++ in conjunction with EIToolkit functionality, respectively.

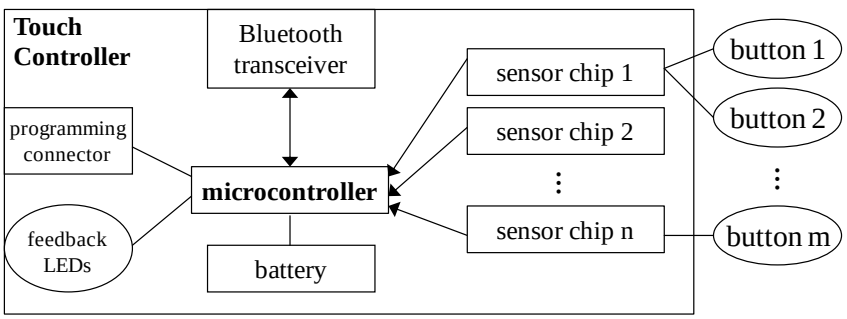


Figure 57: Abstract overview of the architecture of the whole system.

For quickly implementing and testing prototypes, we used the EIToolkit infrastructure to generate a versatile platform to which different applications could be connected. An EIToolkit stub receives events from the Bluetooth transceiver and makes them available to EIToolkit applications. Another stub can then be used to transform these messages according to, e.g., the layout of the buttons or the purpose of the application. These messages are then suitable for stubs like the one controlling the Winamp multimedia player or the one emulating key presses that are used to control applications like specific music players, the Windows Media Player, or digital TV receiver applications. This setting allows simply and on-the-fly remapping between touch input and effect. This proved to be not only important for initial application development but also vital during user studies. Some people for instance wanted to have volume up and down functions swapped with respect to others.

The whole system consisting of microcontroller, Bluetooth module, some additional required components, a few lights for status display and the AD7142 sensor is not larger than a small mobile phone battery, about 2x4x1cm in size. As a side note, the overall price of one such system is dominated by the Bluetooth module. The one we commonly use costs about 110€ as a single item. There are much cheaper modules that, for example, do not support the unnecessarily large range of up to 100 meters as does the Linkmatik module. Most of those are, however, more difficult to use in a non-automated process. Upcoming ZigBee modules promise a small footprint, good bandwidth, and reduced power consumption. All other items (microcontroller, sensors, etc.) add up to approximately 10€. This price is only valid for developers who want to use one or only a few of those systems – the price tag will of course be significantly less with higher volumes.

We now briefly present four different types of clothing or wearable accessories that we augmented with touch controls based on the technology presented above. Afterwards, we go into some detail about two studies we performed using those components. The prototypes were backed by the EIToolkit which enormously facilitated conducting the study since we were able to adapt application behaviour quickly and on the fly.

Phone Bags
We created two sample versions of touch controls on off-the-shelf mobile phone bags, one is shown in Figure 58. The touch sensitive areas were made from thin conductive wires. The areas were intended to be integral parts of the existing design or shaped in ornamental forms themselves so as to constitute an interesting design that does not immediately suggest its button functionality.

Building on the fact that many people use their mobile as a music player, we added five buttons for the common controls to start / pause a song, skip forward, skip backward, turn the volume up, and turn it down.

Figure 58: *Left*: One of the phone bags with five touch areas seamlessly integrated in the existing design. *Right*: a smaller phone bag with just two touch areas, ornamental and blackened, thus barely visible.

The demonstration implementation was mainly targeted to control the music player function of a phone placed inside the bag. A Java ME application directly receives Bluetooth messages and controls the playback of available music files. Since we built all the application with the EIToolkit and conformed to some simple messaging standards, it is also possible to use it – without modifications of any sort – to control an N800 internet tablet or a home cinema application running on a PC (as the one we describe in Study II below).

Helmet

As a second example of a wearable accessory, a standard bicycle helmet has been equipped with two buttons realized as two larger touch sensitive areas. We undertook a quick set of trials to find out the optimal location and shape of these areas. It was chosen to be on the left and right side of the helmet such that it is easy to find them. This tackles the problem that one cannot see where the controls are while wearing the helmet. Since the touch controls do not need to completely cover the whole area, it was easy to integrate them into the existing design by using conductive foil to imitate the manufacturer's name and logo as well as some ornaments like lines or arrows, see Figure 59. The small controller and battery were attached to the helmet in a place supposed to not alter the safety features of the helmet.

In this prototype, we also experimented with a small lamp attached to the front of the helmet that could provide some feedback to the user, e.g., blink when a control was touched.

Figure 59: An off-the-shelf bicycle helmet equipped with touch sensitive areas on both sides ('GIRO' logo and the arrow). A lamp in the front can give feedback.

Gloves

We built a versatile test implementation of gloves with controls on the back of the palm that can be controlled using the index finger of the other hand. The necessary modifications of the gloves are in fact very small and can be implemented very cheaply. As can be seen in Figure 60, the index finger of the control hand has been extended with a small patch of conductive yawn to enhance the capacity flow to the sensor. The more expensive electronic parts (e.g. Bluetooth module) are meant to be attached separately. Since especially working gloves are replaced in shorter intervals, this ensures that the additional cost is kept to a minimum. In the test prototype, different sets and layouts of controls can be attached to the glove using a small plug and Velcro tape. In a final version, this could of course also be embroidered into the glove.

Figure 60: The prototypes used to demonstrate the idea of adding touch input on gloves. Some configurable buttons are placed on the left palm. The right index finger is used to initiate commands.

Apron

For a larger study of direct input on clothing, we built an apron equipped with three different implementations of a set of touch button (Figure 61) and a small pocket for the electronics and, possibly, a phone. One of the major issues in finding out about how and what people would want and accept, is the interplay between functionality and fashion. To find out more about this, we use three different button styles: visible button-like shapes, ornamental buttons, and nearly invisible buttons.

In Figure 63, it can be seen how wearing the apron looks like. Sure enough, some male participants had the impression of wearing a skirt but got quickly used to it, especially since we placed some part of the studies in a kitchen scenario.

There are three obvious reasons for choosing an apron. First of all, it is a stand-alone piece of clothing that can be employed, besides as a standard apron, as a remote control. More importantly however, it served as a simulation tool for controls embedded into a skirt, trouser, or pair of shorts. Such an approach drastically eased the study process. Several versions can be tested relieving participants from having to change clothes in the beginning and during the course of a study. In contrast to other attachments, the apron still conveys the feeling that it is closely attached to the body, moves with the motions of the wearer's body, and also otherwise behaves like a normal piece of garment. Second, this additionally enables the use of only one piece of technologically enhanced garment that fits all sizes and allows using one and the same sensor, processing and communication platform for different layouts and types of controls. And third, it fulfils an important prerequisite of our study that people have the freedom of slightly adjusting and relocating the set of controls on the body. This possibility of quickly and unconsciously rearranging controls gives much better results about optimal positions than just having users try to imagine them.

Figure 61: The apron (*top*) and close-up of the 3 different designs of the touch buttons (*bottom*).

6.4.2.2 Setup of the User Studies

In order to evaluate the wearable inputs, two user studies were arranged. Before each study, a pilot with one participant was held in order to refine the questions and to time the sessions to have an appropriate length.

Participants

The two user studies were done during July and August 2007 in Finland. The first study included 10 participants recruited as test participants in an ad hoc fashion. For the second user study, 8 participants were pre-recruited for a one hour test session each. In the beginning of the second study, the questions and tasks of the first study were repeated increasing the number of participants for the first study to 18.

In the tests, there were an equal number of male and female participants, with ages ranging from 16 to 30 years. Young people were chosen as a target group for the study as such types of technology typically enter the market first at this group. This appeared to be true as all of the test participants had mobile phones and 15 out of the 18 also owned MP3-players. To get an idea of how 'wearable' their current habits to use mobile technology was, we also asked where they stored their mobile devices at the time of the test. Only one participant was carrying an MP3 player at the time of the interview, but all had their mobile phone with them. All female participants carried their phone in a bag but half of them responded to also have it sometimes in their trouser pocket and one person in her jacket pocket. With regard to male participants, 7 of 9 had the phone in their trouser pocket correlating with their handedness, and 2 carried the phone mostly in a bag.

Study I

The first user study was held in a public place, a city café, and consisted of a semi-structured interview during which the participant could try out all three prototypes (Figure 62). The gathered material was based on the direct answers to the questions as well as on observations. Three researchers were involved in the interviewing situation – one interacting with the user, and the other ones observing the users and preparing the prototypes. Each test session took approximately 20 minutes.

In the beginning, ten background questions were asked. After that, the bag, helmet, and gloves were shown to the participant, in this order. With each prototype, the impressions and opinions of the participant about these were gathered. In the end, the participants were asked which of the ideas were their favourite and the reasons for that. Finally, they were asked about the benefits and obstacles they saw with wearable computing and to demonstrate where (on the body) they would like to place such input controls. To facilitate the question about placing the wearable input, we introduced the users with the idea of a near-to-eye display with the possibility to play computer games or watch videos unnoticed by others, thus needing a suitable set of wearable controls.

Figure 62: Study setups in a public shopping mall for Study I and a usability lab for Study II.

Study II

The second study took approximately one hour per participant and consisted of two parts. The first part of Study II was held in a semi-public restaurant and had an identical setup with Study I.

The second part of the Study II was arranged in a usability lab immediately after the first part (Figure 62). The user was asked to perform tasks with wearable controls integrated into an apron, see Figure 63, where three sets of controls had the same function but a different look and feel (i.e. visible buttons, ornaments, and nearly invisible buttons). The tasks were set up to resemble two different environments, namely a train setting and a home kitchen scenario.

The train setting simulated a small semi-public display integrated in the front seat of a train compartment. The user was asked to control a front seat display with apron buttons while sitting. Some tasks were:

- Find a certain TV channel.
- Turn the volume up and down.
- Stand up (‘in order to get coffee’) and perform some tasks.
- Perform some tasks with a coffee cup in hand.

This was repeated with different wearable controls (Figure 63). The order in which the controls were used was changed with each participant.

In the kitchen scenario, home cinema devices (stereo and TV) were controlled using the controls on the apron. In this scenario, the user was standing in the kitchen, ‘preparing’ a meal and should control first the TV and afterwards the stereo running in the background. Here, the tasks included navigating TV channels and changing the volume. After that, users were asked to control the music player with the same wearable controls. Here, they were additionally asked how they would switch between devices with wearable inputs and about their opinions related to settings, controls and tasks.

Figure 63: The apron worn by a user and adjusted such that the visible set of controls is located on the upper thigh as used in Study II.

6.4.2.3 Results, Lessons Learned, and Guidelines

In the studies we present here, the participants often had to rate a specific tool, type of interaction or give their opinion. We used an answer scheme according to a Likert scale which translates into numbers between 1 (very negative) through 3 (neutral / do not know) to 5 (very positive). For such quantitative data, we present mode values. Besides the subjective values of the participants, the observers also rated the interactions according to their own observations. There was a high correlation between the ratings (> 75 %), the main differences being that observers more often neglected difficulties at the first attempts of a user with a specific technique. This also indicates that users were not influenced too much by the 'presenter-bias', i.e. possibly trying to be polite to the presenters. Given citations from non-English speaking participants have been translated to English.

Wearable Accessories
All participants from the studies carried a mobile phone with them at that moment. Common storage places were bags and pockets suggesting that it can indeed be of additional value to them to be able to control some functionality without needing to take the phone in the hands.

The social acceptability is a large issue for the design of wearable devices and controls. As one of the initial questions, we asked where people would be prepared to use wearable controls in public. Only controls on a trouser, wrist band or separate bag received acceptable values (median values of 4 or more). Locations on the upper body like shirt or scarf were generally rejected (median values of 2 or less).

Several participants who tried the devices mentioned that, because of a personal taste, they do not like dangling or attachable things. Those then suggested integrating the controls into clothing like a belt or trouser. This additionally motivated us to initiate the second study described next.

Wearable Controls
Participants in this study could put on the apron and control media playing in the infrastructure as described above. Generally speaking, the users were very positive about the applications and the ease of controlling; especially that they were not bound to a specific location in the room, that a line of sight to any of the devices was not necessary, and that there was actually no sign of any control or computing infrastructure. When asked about how to switch from controlling one device to another, all but one suggested using another special button on the clothing. This indicates that, when seeing how this interaction technique works, they do not think of physically walking to a device. However, if the choice of available devices grows, solutions to select the desired device like those we presented in [Rukzio, Leichtenstern, et al. 2006] are clearly superior than having one button to cycle from one to the next.

Main Findings

In the course of the studies as well as during the analysis of the results, several issues have been identified that should be taken into account by application developers in this area. This section goes into some detail about the four most important guidelines that we found. Afterwards, we more concisely mention several others in a lessons-learned section.

There are No Clear Expectations on Layout and Meaning
Even for simple interfaces involving only very few controls, there seems to be no clear expectation of the arrangement, layout and meaning of the controls. For the media player control (start / pause, next, previous, volume up, volume down) for example, we had 5 different interpretations of one and the same button layout (swapping, e.g., next and previous or volume up and down). Hence one could argue this should be customizable and that idea also appealed to the participants. We implemented this version for the N800 scenario. However, an interesting observation was that all of those who initially expected different behaviour than they found, accepted the given mapping and got used to the specific arrangement after at most 2 wrong attempts. This means that the need for configuration is maybe less important if the next guideline is followed.

Locating and Identifying Controls must be Quick and Easy
For a successful design, it is essential that users can find the controls and see the functionality assigned to them visually and tactile. A tactile way of finding a control is important for all applications that should possibly be operated during other activities ('blind'). In our sample applications, for example, purely visual controls might be enough for the setting using the public display in the train seat. The other music player applications, however, are often in use while doing sports, cooking, etc. Not much surprisingly, the first reactions on the three control designs (visible, ornamental, invisible) were according to this statement:

> "Big buttons are ugly. It would be even better if the metallic ones were part of a larger ornament. The invisible ones 'look' of course the best."

However, during the experiment, the observed and subjective grades given for the four tasks were lower for the invisible controls than for the visible or ornamental ones (however, statistically significant only with respect to the visible ones: a 1-tailed, paired t-test gives $p < .03$). As can be seen in Figure 64, people found it harder to use without looking (significant differences for all three button styles: $p_{visible} < .001$, $p_{ornamental} < .01$, $p_{invisible} < .0001$). Nevertheless, only the invisible ones made the scores degrade in a completely unacceptable way.

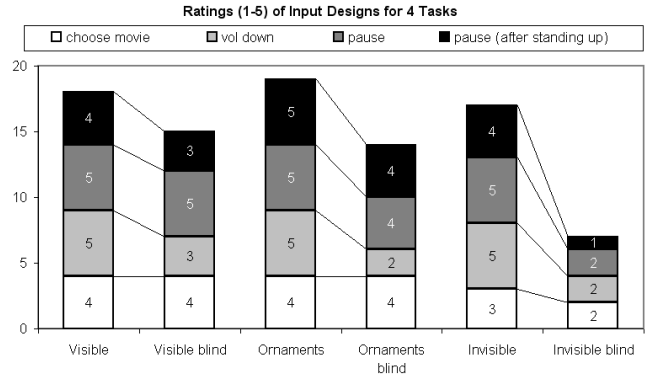


Figure 64: Subjective ratings (mode) according to a Likert scale from 1 (negative), to 3 (neutral) to 5 (positive). Four tasks have been evaluated with three different designs; each of those was also done allowing tactile search only ('blind').

This data suggests using controls that are embedded in the design, are visible and tangible as well as look and feel different for different functionality is the best choice.

Ensure One-handed Interaction
Besides the appearance of controls, care must be taken to ensure that one handed interaction is possible as this is expected from a wearable control. This should not be underestimated. In fact, several existing designs can be found that use, e.g., the sleeve to embed a keypad or keyboard. This means that both arms and hands cannot serve any other purpose at the moment of input. Avoiding this is especially important for use during specific activities like cycling, motor biking, running, skating, or working. A designer can thus also decide not to follow this guideline if a thorough investigation of the usage of patterns of the target group reveals it is unnecessary. For any generic application, though, it is vital.

Provide Immediate Feedback
Even a short delay between input and action can be extremely critical. We observed that people do not yet know or understand the way of operating touch controls. Although all our touch control designs have a haptic impression, all people initially used those expecting some tactile, 'button-like click' feedback. Whenever there was no immediate consequence after a touch, people tried to push the electrodes harder instead of releasing and touching it again. In most controls we implemented a dwell time which required people to stay on a control for a short amount of time to tackle false sensor readings and to avoid accidentally initiated commands (see also the next section). According to observations and comments, this accounted for more than a third of the problems the users initially had in controlling the applications. The physical separation of controls and system and the several points of failure require a minimal delay between input and action (see, however, the guideline 'Need to Tackle the Fear of Accidentally Initiated Commands' below about ways to avoid accidental uses).

Lessons Learned

This section briefly describes more results that we were able to draw from the studies described above.

Even Minimal Enhancements Can Convince the User
A very positive and motivating finding was that, even if several scenarios arguably offered only a small increase in the ease of handling, such approaches are very much appreciated. For the designs used as remote control for a phone, more than half of the participants explicitly valued the indirect access:

> "That is useful. No need to take phone out while walking or cycling. I don't see any bad sides."

Additionally, the demonstration with the augmented helmet showed that people also recognize the value of having an additional motivation to using such safety devices.

Preserve the Original Functionality
For users it is paramount that the original functionality and way of use must not be compromised. This also includes safety concerns, e.g., when embedding hardware in a helmet. It especially applies to devices that have a distinct set of functions to fulfil like gloves.

Need to Tackle the Fear of Accidentally Initiated Commands
As mentioned above, we saw a timing dilemma in our experiments. On one side, users want to be sure that they don't operate a button when touching it by accident. On the other hand, longer required dwell time when deliberately touching often led to frustration as there is no immediate reaction. Mechanisms that provide immediate response and also have a key lock function need to be developed. Possible solutions include using double taps (as suggested in [Ronkainen et al. 2007]) or operate the sensors only when the palm touches a surface slightly above the controls.

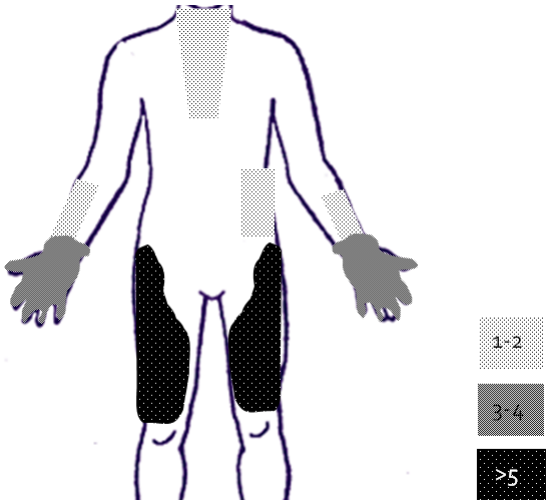


Figure 65: Preferences of people where wearable touch controls are acceptable. The darker the colour, the more often this place was indicated. The thigh was mentioned most often (standing postures only).

Optimal Position of Controls Influenced by Posture
The optimal position of controls with regard to the body shows clear trends over the whole population of participants (see Figure 65). However, body posture has an impact on optimal positions and hence the location of controls which are operated while standing may have a different slightly optimal location while sitting. Although we did not concentrate on further exploring this yet, we can say that all but one of the users raised the controls on the thighs when standing by about six inches. Several persons also wanted the button functions to be different while standing. We implement that using a built-in accelerometer and refer to, e.g., [Lombriser et al. 2007], [Mattmann and Tröster 2006] for projects and links to more advanced recognition methods.

It should be noted here that all of the participants stated that they judged *detachable* controls to make sense. The two reasons given were that the location could be controlled and it could be used for different clothing and can be reused even if, e.g., the gloves break. However, the cost of embedding the touch electrodes (and possibly sensors) is already minimal. Only more expensive parts like the wireless connection module must be replaceable. A possible solution for the location problem could be replicating sets of controls at different positions.

More General Remarks

For clothes as well as accessories, users put fashion forward as a main concern. This provides a good opportunity to improve the expressiveness of clothing (e.g. a technical style) but is a serious risk as people might not like the product merely because of its appearance and not because of its function or functionality. Including fashion and clothing designers for a commercial project is therefore mandatory.

Besides fashion design, one of the issues we confirmed in the interviews is that there exists a variety of specific interface needs, in particular with regards to the gloves (e.g. for different work environments).

Additionally, an often expressed issue is that people do not want to be concerned with another device to charge. Long battery life and ease of handling, storing and charging is critical. We propose to employ a simple plug mechanism to connect and draw power from the phone, e.g., implemented in the phone bags or a pocket (we saw that more than 60 % of the participants kept it in a trouser pocket). We refer to [Toney, Thomas, and Marais 2006] for a way of simplifying the management (which includes recharging) of smart clothing with enhanced clothes hangers.

Generally speaking, integration of controls into garment enables possibilities for interactions during activities like working or cycling that were difficult or not possible. This shows the utility of wearable controls but at the same time introduces new risks to these activities.

6.4.3 Touch Input on Mobile Phone Keypads

Command and text input on mobile devices is the focus of numerous research projects. Their small size heavily restricts the user interface design space. Whereas screen resolution has increased, thus allowing sophisticated GUIs, the input mechanism has remained rather unchanged through the history of mobile phones. The next section presents findings from a possible extension of small-sized keyboards focusing on the evaluation of several applications based on this idea: we added touch sensors to the keypad of a normal mobile phone, thus adding an additional dimension to the input possible with such a keypad.

Technology and Prototype

This project also demonstrates a design process supported by our EIToolkit approach. Although the final application was supposed to run on a phone independently of any infrastructure, the toolkit proved helpful during the iterative development process. As one of the first steps, for example, we implemented a simple PC application visualising the output of the touch sensors, see Figure 66. It is very easy to write and test applications in this way as they do not need to be compiled for, transferred to, and installed on a phone.

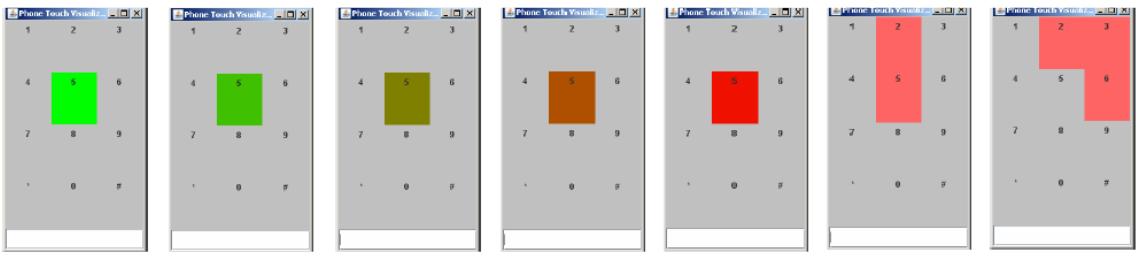


Figure 66: PC application to visualise touch states on a phone's keypad. More pressure is visualised with darker colour / red. When a finger exerts too much pressure, other keys are touched as well.

We used Flash Lite[91] to develop applications for the user studies. This has the advantage that graphics and animations can be taken from the applications developed using the EIToolkit. It also enables the straightforward initial testing and evaluation of the applications and quick alteration and replacement of background themes.

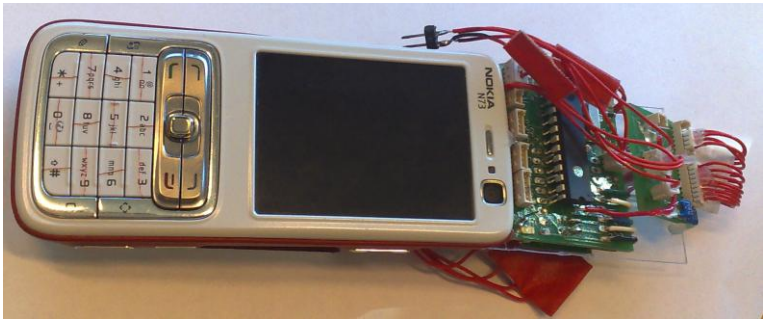


Figure 67: Nokia N73 with touch electrodes on the keypad, Microcontroller, Bluetooth transmitter and AD7142 touch sensor board (far right) attached to the top.

Our prototype (see Figure 67) uses the AD7142 touch sensor chip based on capacitive sensing. It can detect touch on up to 14 electrodes (nearly) concurrently. Upon detection, an event is sent that contains information about the keys that have been touched as well as a value indicating the amount of exerted pressure.

In order to place the electrodes on the phone keypad, a Nokia N73 phone was disassembled and thin isolated metal wires were placed around the keys. Figure 68 shows how the wires were routed in a way they do not touch and still cover a large part of each key. The wiring is nearly invisible and does not change the feeling and functionality of the keys at all. Since we had to place the additional sensor electronics on top of the phone in order not to change its way of holding and handling, it was a challenge to route the wires through the phone's body without collisions and without letting the internal components influence the sensitive measurement.

Figure 68: Layout of the electrode wires on the keypad of the N73 and its use in the test applications.

A problem with Flash Lite in its current version is that it does not support serial connections to Bluetooth devices. Inspired from the Flyer project[92] (which relies on Python for S60 and requires a full installation of Python and the Python Script Shell), we developed two EIToolkit stubs as Java ME MIDlets that on one side connect to a Bluetooth module and on the other can forward data to Flash Lite using the XMLSocket connection available in Flash Player 3.0. Thus, touch events from the sensor platform received by the Java ME MIDlet are forwarded to the Flash Lite applications where they trigger a specific action like showing a pop-up menu.

End of 2007, Nokia released its Nokia N81 8GB with a touch sensitive hotkey ring that can be used to scroll through menus (also planned to be on later phones such as the N96 but was removed in the production models). We deliberately did not use the hotkeys or joystick region of the phone to attach the touch sensors. First of all, the area is very small such that touch with fingers often leads to wrong detections. Second, we also planned to enhance text and number input. Lastly, there are many applications that use the keypad as main control or shortcuts since hotkeys are already dedicated to menu and options navigation.

[91] Adobe Flash Lite runtime for mobile devices; product page: *http://www.adobe.com/products/flashlite*
[92] Flyer open source Python framework for Flash Lite; among other features, it uses XMLSockets to communicate data from Python to Flash applications; project page: *http://code.google.com/p/flyer/*

User Interface

On top of our touch keypad, we developed three distinct applications building upon a simulation of the graphical user interface of Series 60 phones. The applications present the phonebook and the image gallery. The basic functionality is the same as in the standard applications except the number keys are used to scroll and select.

Info screen application
The first application was used to introduce the touch-press feature. It is possible to browse names available in the phone book by pressing up and down and to look at the respective phone number by pressing the middle button. When merely touching the middle button, a window showing recent calls to this contact popped up (see Figure 69). This info screen disappears after a moment when the finger is released from the button.

Pictures application
The second application also uses the phonebook, but shows a collection of images related to the selected contact in the pop-up (see Figure 69, right). During the time the pop-up is shown, all available images can be scrolled by touching the up and down keys. This pop-up also automatically disappears after a short while without touch.

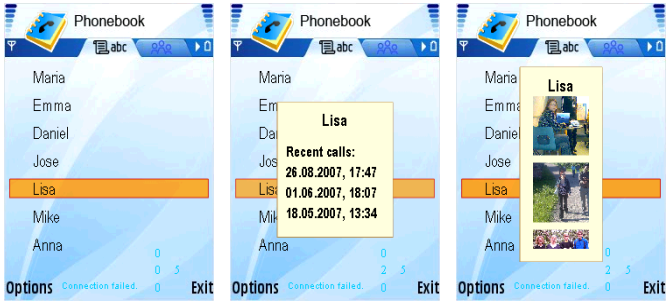


Figure 69: Phonebook application with two types of pop-up info screens that appear when touching the select key.

Gallery application
The third application features a thumbnail gallery in which users can browse images and enter a full screen browsing mode by pressing buttons. In full screen mode, it is possible to zoom into and out of the image by simply touching the up and down keys. The sequence is illustrated in Figure 70.

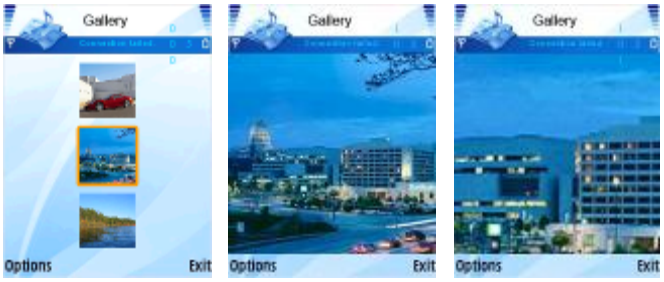


Figure 70: Image gallery application. While clicking selects, touching zooms in and out of the currently selected image.

User Study

The main focus of the study was to find out whether people understood the basic concept of touchable buttons, how they judged the user experience of this kind of interaction and whether they could find benefits from such an extension. We recruited 10 people for the tests, 4 female, all of them Finnish, between 20 and 50 years old, most of them having a technical background. All of them were right-handed, familiar with the S60 look-and-feel, used the phonebook daily, and the image gallery at least weekly. Besides telling the users that the buttons were touch sensitive, we initially did not give any details in order to see how intuitive the prototype user interface was.

In the tests, we had two different kinds of use for this technique. One was to use the touch sensitive buttons as enabling technology for pop-up windows showing additional information about menu items. The other one was to use it to implement shortcuts to commands such as zooming. The given tasks were:

- Select 'Lisa' in the two phonebook applications and touch the middle key.
- Scroll the images in the (second) phonebook application and exit the view.
- Select the image of the city in the gallery and find out how to zoom.

The user tests took place in a usability lab furnished like a standard living room. A small video camera mounted on the phone was continuously recording finger movements and screen content. After the tests, people filled in a questionnaire where they could judge and comment on different usability metrics.

Study Outcome

Only 2 of the 10 participants had difficulties using the first application – mainly because there was a too long delay before the window popped up and users didn't keep their fingers long enough in the same position. We reduced the delay from 0.7 to 0.5 seconds for the other participants which clearly improved the situation. Despite of the delay, people perceived a clear difference between touch and press actions.

The usefulness of the technique was rated high, see Figure 71, and participants found both pop-ups and shortcuts practical. The participants felt that touching can increase input speed and in general liked it. Our application made zooming easier and faster than in existing galleries with the S60 look-and-feel, and the interaction techniques were seen as a way to reduce the use of menus. One participant mentioned that the input technique could also improve possibilities to use the phone with one hand only, as today, for example, the copy-paste function in S60 phones needs to be performed with two hands.

As application specific feedback, users mostly liked the idea of pop-up screens, but typically wanted to see some other information than recent calls, e.g. phone number, address, a picture, or application help. The opinion was split on the way the pop-up window was closed: one half liked that it disappeared automatically, the other half wanted to manually control when the screen would go back to standard mode. The image gallery application turned out to be the most popular one. Eight of ten users found out the zoom function without help, and after finding it, they all liked it. Here, dealing with the delay was somewhat problematic again.

People were positive towards the interaction in general, but also saw potential issues. The concerns with the input method were mainly related to learnability and memorability. Participants were worried that they had more things to remember because they had to learn where touch is enabled and what functions the buttons had. Also, anxiety to learn the correct sensitivity for a proper use of touch-press was mentioned. However, some people got quite comfortable with the touch buttons and even tried whether scrolling movements would also work.

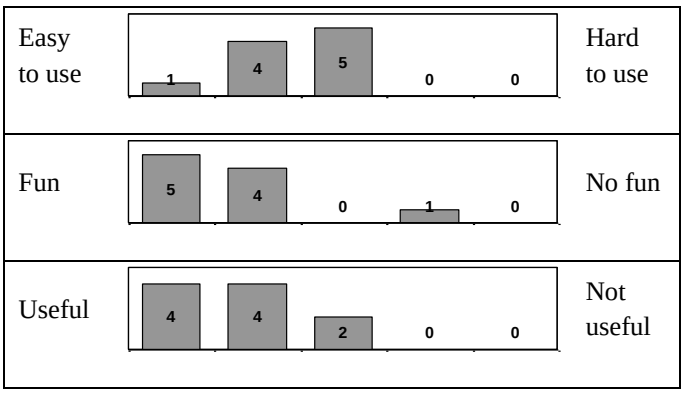


Figure 71: Feedback from the user study. Possible values ranged from 'completely agree' to 'neutral' to 'completely disagree'.

People are very reluctant in accepting erroneous or slow device behaviour. To have such touch functionality in a deployed product, the sensing system must therefore work extremely well. Achieving such correct behaviour is especially challenging when sensor placement and sensitivity should be optimized for a mass market. People are also reluctant to do calibrations. Based on our observations, we recommend people can switch the feature off.

As Figure 72 illustrates, care must be taken to have enough spacing between buttons. When pressing one button, the one below is often also touched. This effect is especially strong if the keypad is used with the thumb and increases the farther away from the centre of the hand the finger are since they are then held more horizontally.

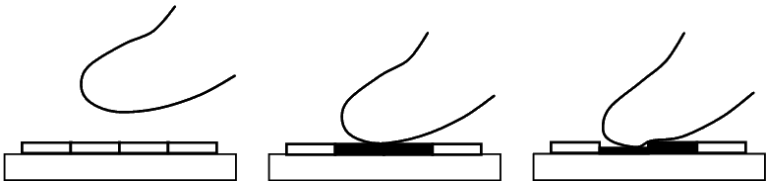


Figure 72: When pressing a key, a finger often touches a lower key at the same time.

There is a necessity of a small delay (we found a value of around 0.5 seconds to be sensible) between touch and triggered action since otherwise the action would always occur when pressing a button (without the intention of triggering it by touching). Since people do not like to wait, such touching interactions should mainly be used to display additional, non-critical information or to enable functionality with a large benefit (e.g. shortcuts). Often, people are hesitating before they initiate an action, and offering help or information at such times seems to have large potential. However, we also observed that people often move or keep their fingers on a button without a particular intention or just to reflect about something, potentially resulting in erroneous actions. Also, based on our observations, wide and varying use of touch enabled button functionality can lead to confusion and can be hard to learn and remember. Thus, UI designers must be very careful in using touch for such purposes since otherwise it might quickly become more annoying than helpful.

The EIToolkit functionality helped in designing, testing, and creating the described applications. In the following chapter, we show how the development of such applications can be even more simplified and accelerated using a specifically designed development environment for mobile phone applications. We also present more examples of applications realised using the touch features of out prototype.

7 Prototyping Mobile Device Applications

This chapter introduces a development process and environment to quickly generate prototypical applications on mobile devices. It provides a low threshold and a high ceiling and enables people with various backgrounds to create applications. It makes use of the EIToolkit, user models, and many of the processes described in previous chapters.

First, we give an introduction to the concept of mobile phone programming, the design goals of the MAKEIT environment, and a succinct treatment of related work (7.1). We then show how mobile phone applications are developed, describe how our framework enables simple prototyping, and detail the role of programming by demonstration in the design and creation of applications (7.2). After treating some implementation specific issues (7.3), we describe how a separate code generation is used to develop for different target platforms (7.4) and list interactions and features currently supported by MAKEIT (7.5). A short evaluation and discussion of the system concludes the chapter (7.6).

Parts of this chapter have been published as [Holleis and Schmidt 2008].

7.1 Introduction and Related Work

Mobile phones have become a ubiquitous computing platform outnumbering desktop computers. A large portion of current mobile phones offer means for third parties to develop custom software for them. Most notably many of them run Java ME, Symbian OS, or the Windows Mobile platform. Modern phones provide rich ways for interaction, reaching from speakers, microphones, and keyboards, to gestures, cameras, and colour touch screens. Additionally, more and more such devices include additional sensors, e.g. for acceleration (e.g. Samsung SGH-E760, Nokia 5500 / N96, Apple's iPhone). Interaction with physical objects using barcodes is a common feature in many phones and some devices can read smart labels (e.g. NFC reader in the Nokia 6212). Furthermore, phones can be extended with external sensors connected via Bluetooth, e.g., GPS, step counting and ECG (electrocardiograph). Thus, current mobile devices and phones provide a promising platform for many pervasive computing applications.

These basic technical capabilities enable developers and interaction designers to create novel interactive experiences using mobile phones in domains such as data access via physical artefacts, context aware applications and mobile health applications. Although APIs exist that allow accessing sensor values, it is still often a challenge to create sophisticated user interfaces that exploit all these capabilities. In comparison to conventional interaction techniques, there is yet little established knowledge about how to build compelling applications using these new means. Hence developments often rely on trial and error which can be costly. In most cases, novel experiences require functional prototypes to be built and evaluated. We believe that prototyping and tool support is essential to make this process efficient. Development environments support the implementation on source code level and to some extent the design of the interaction flow (e.g. in the NetBeans Visual Editor). Some of them have been mentioned in Section 4.1 about general software and hardware toolkits. There is, however, a lack of tools supporting prototyping interactive mobile applications that make use of advanced interaction techniques using internal and external sensors.

To support development, a user-centred design process is often employed based on paper prototypes. It is commonly agreed that at least partially working prototypes are essential in order to efficiently develop interactive applications, to assess new interaction concepts, and to convey ideas and interfaces to end-users as well as developers. Including target users in this phase can be especially important for pervasive systems. Various arguments and examples can be found in a special issue on rapid prototyping in the IEEE Pervasive Computing magazine [Davies, Landay, et al. 2005].

With our system, we address the gap between low-fidelity paper prototyping and fully working implementations. The MAKEIT framework (short for '**m**obile **a**pplications **k**it **e**stimating **i**nteraction **t**imes') is used to create functional, high-fidelity prototypes of applications that support advanced interaction techniques and run on mobile devices. In particular, we focus on the need to easily create and change applications while at the same time providing assistance in keeping estimated end-user interaction times low. We thus build on the concept of programming by demonstration in order to keep the initial threshold low and follow a similar process to paper prototyping by building a state graph supported and maintained by the system.

In order to help the developer evaluate novel applications, we integrate testing with user models as described in the last chapters. A goal is to enable people who are not experts in user modelling to make use those in order to derive usability measures. To our knowledge, this is the first project that combines an environment for quick prototyping of mobile device applications with the direct support for the integrated creation and application of predictive user models. The contributions of this project include:

- An integrated development environment for high-fidelity prototyping of mobile phone applications using programming by demonstration.
- A connection to other development tools such as the EIToolkit which enables the simple use of advanced interaction techniques within the developed application.
- An underlying model based on state graphs that can validate parts of the application logic, detect flaws in the navigational structure, and suggest alternatives.
- An integrated model that can estimate task completion times early in the design process without needing to deploy a prototype on the actual target hardware platform.

Concepts and projects related to the MAKEIT system fall into four categories. First, several rapid prototyping environments have been suggested. Second, programming by demonstration has been employed by some projects, mostly for specific areas or end-users. Third, there has been much research in the area of user models and some applications can be used to generate user models. Finally, very little support is available to combine application development tools with the generation and exploitation of user models.

Rapid Prototyping Environments

The first category subsumes all kinds of rapid prototyping and authoring frameworks as well as tools that can be used to quickly create prototypes of mobile or pervasive applications. A comprehensive review of this area has been given in Section 4.1. Some of those also follow a state-based approach but, in comparison to MAKEIT, they all are restricted in one or several of the following aspects.

- They are strictly based on a set of available components like text boxes and lists and do not allow quickly adding free drawings and designs.
- They do not directly support advanced interaction methods such as gestures or using RFID tags.
- They have not been built to integrate non-functional properties like KLM parameters.
- They do not use underlying models that can be exploited for consistency checks or usability analysis.
- The built prototype depends on the presence of a PC as a common gateway and data store.
- They do not leverage a single, well-known development environment for these multiple purposes.

Programming by Demonstration

A strong concept in the area of end-user application development is programming by demonstration. The idea is to remove the need for explicit textual or graphical programming. By simply performing an action, the computer system is supposed to understand the intentions of the user and can then use this interaction to trigger some action. In practice, however, this is not so simple to realise. One of the hard questions is how to abstract from the demonstrated actions (and how to know whether abstraction is necessary and in what magnitude). If an example interaction is opening a door, there must be some means to specify whether this is equal to *any* door to be opened, the same door to be opened by different angles, etc.

The idea itself is rather old, see for example [Halbert 1984] or [Nardi 1993] for overviews. Specifically in the area of prototyping for pervasive computing, the paradigm of programming by example or by demonstration has been directly followed, e.g., by Topiary [Li, Hong, and Landay 2004], HP Mediascape [Hull, Clayton, and Melamed 2004], and DENIM [Newman, Lin, et al. 2003]. They allow specifying triggers of actions, which are comparable to the actions used in the MAKEIT environment. Topiary and Mediascape concentrate on location-based applications where certain places and regions are indicated on a map. Subsequently, actions can be triggered based on movements related to these regions. The DENIM project shows similarities to the approach presented here, letting the designer create transitions between states. The integration of conditionals, i.e. actions that depend on the properties of a state is planned; this would reduce the number of states visible at the same time as do corresponding approaches in MAKEIT. The system, however, requires its user to learn several types of gestures, is designed for web page generation, is not open and easily extensible for external components, and does not integrate well with later steps in the application development process. It will be interesting to see how a planned, more powerful visual programming language will influence the capabilities and usability of the system.

In [Hartmann, Abdulla, et al. 2007], the authors describe the Exemplar system with which users can demonstrate specific events based on continuous sensor data. The variety of interaction reaches from simple thresholds to more complex pattern recognition algorithms. Their approach to ease the automatic adaptation of the demonstrated actions is to keep the users in the loop while iterating and refining the provided examples by graphically adjusting parameters. Another example of programming by example relevant to our work can be found in the description of the CogTool [John and Salvucci 2005]. It is a storyboard-based system that can generate simple Keystroke-Level Models by demonstrating a task on a mock-up of a mobile device. This is similar to part of the development process suggested in the environment described in this chapter.

It should be noted that programming by demonstration is different to visual programming in general (as employed, e.g., by [Monroy-Hernández and Resnick 2008] in the Scratch language). iCap, for example, is a visual language that uses condition-consequence (if-then) rules and relations between people, places and things to define the semantics of an application [Sohn and Dey 2004]. Even though the threshold to creating application can be much lower than when using textual programming, more understanding and knowledge about concepts, rules, and syntax is required than with programming by demonstrating action sequences.

User Modelling Tools

There are many strategies and methodologies for usability evaluations of user interfaces. These reach from think-aloud (users tell what they are thinking and why they are doing certain actions) to log file analysis to questionnaires and surveys. Ivory and Hearst give an extensive overview of 132 usability evaluation methods [Ivory and Hearst 2001]. One analytical approach involves user models such as GOMS and KLM. Ivory and Hearst's work can serve as a reference for those methods and also shows a description of several tools that are able to generate user models. We explicitly mention several tools that had a high impact or are more widely in use today in order to compare them to our own system. Some approaches mentioned in Section 3.2 in the area of cognitive user modelling such as ACT-R, EPIC, and SOAR are related but are mainly used to describe and formalise such models. In this section, we concentrate on those tools that aid developers without deep knowledge in cognitive modelling in some way to employ user models.

GLEAN helps in automating the process to derive usability metrics from GOMS models [Kieras, Wood, et al. 1995]. A user interface developer has to provide a GOMS model and a set of benchmark tasks. In addition, the system needs an abstract description of the user interface to enable the system to interact with it, e.g. to know when a hand movement between mouse and keyboard occurs or where objects are in order to find and modify them. GLEAN has the advantage that it directly uses cognitive concepts such as a simplified model of working and long time memory to interact with a user interface. However, a property that complicates the generation of predictions for non-experts is the language that has to be used to describe the GOMS model and the tasks. GOMSL, a machine readable version of the natural GOMS language (NGOMSL) prescribes the structure and syntax of goals, methods, and operator descriptions. In addition, the need for manually describing a representation of the interface makes the system rather cumbersome to use. Still, GLEAN has been demonstrated to model differences between interfaces in a similar way as hand-made models but needs only a fraction of modelling time.

A similar approach is followed by Apex [Matessa, Remington, and Vera 2003]. It differs in implementation detail and in that it supports CPM-GOMS modelling which enables parallelising certain actions. It provides a framework defining a formal specification and a procedural description language with Lisp syntax to specify the methods to perform a sequence of steps. In order to let steps that only use one of the cognitive, perceptual, or motor resources (CPM) run in parallel, templates are defined to bundle, order, and prioritise unit operators. Apex is quite powerful in automating part of the process to generate a GOMS model and offers a visualisation as a PERT chart. However, the initial specification of the methods and tasks still requires that a user has much knowledge about the underlying system and can express procedures in a programming language.

In order to further simplify the generation, modification, and execution of GOMS models, Quick GOMS has been introduced in [Beard et al. 1997]. It features a tree-like visualisation of the hierarchical structure of a GOMS task. The graph can directly be manipulated and properties of each component is visualised. Using this hierarchy, interesting elements of the tasks can be expressed. For example, relative probabilities can be set that specify what method will be used how often with respect to other methods for the same task. Execution time predictions are given for the unit operators and are propagated to their parent. This means that the predictions can easily be seen for each sub-goal at any time. Quick GOMS allows for much easier and quicker alterations in a user interface than possible in GLEAN and Apex. However, there is no connection to the real user interface being developed and it cannot be tested whether the model conforms to the interface or whether some errors have been introduced. We refer to [Baumeister, John, and Byrne 2000] who provide a more detailed comparison between GLEAN, Quick GOMS, and other tools to generate GOMS models.

A well-known project called CRITIQUE attempted to deal with some of the issues pointed out with these tools [Hudson, John, et al. 1999]. A similarity with our MAKEIT tool for mobile device programming is that users can demonstrate the tasks they want to have modelled. For this, CRITIQUE draws from a specific user interface toolkit named subArctic in order to automatically generate a GOMS model. This process requires a selection of heuristics to be applied and the authors use a set of patterns to match events to operators and filter unimportant events such as minimal mouse movements within a double click. A comparison of an example model to real user data revealed that some assumptions, e.g. on the experience of the users, had to be reconsidered. One of the strengths of the CRITIQUE system is that small changes within the model can easily be done since a different method can quickly be demonstrated or additional operators be added. The tight connection to the subArctic interface toolkit complicates the combination with other systems or applications and we judge the effort of needing to program in C++ or learning a new language for user interface descriptions as too high for being accepted by a broad audience of non-experts.

Besides these rather complex projects, there are some approaches that mainly consider ways to visualise KLM and GOMS models and action sequences. Since we concentrate more on the generation and interpretation of these models, we refer to a recent project called ExperiScope [Guimbretière, Dixon, and Hinckley 2007] for an example and overview of that part. It provides many features such as clustering of patterns and combines KLM with Buxton's Three-state Model mentioned in Section 3.1. Many other systems, notations, and visualisations describe models of tasks and applications. We refer to [Mori, Paterno, and Santoro 2002] for an overview of those methods and projects including, for example, a treatment of UML to model such tasks.

Prototyping Tools Integrating User Models

Prototyping tools that explicitly incorporate user models are hardly available. Some applications incorporate user traces into the process of developing a user interface prototype. SUEDE [Klemmer, Sinha, et al. 2000] and WebQuilt [Hong et al. 2001], e.g., record user test data for speech and web UIs, respectively. The major difference to our system is that we do not rely on actual user data but use validated interaction models. This drastically reduces time and cost for reaching decisions regarding projected user interaction times.

There is only one project where a prototyping tool is suitably connected to user modelling: John et al. introduce the idea of the CogTool application in [John, Prevas, et al. 2004] and refine it in [John and Salvucci 2005]. CogTool offers the possibility to create quick designs of applications based on storyboards encoded as HTML. Users then record the execution of tasks by interacting with these storyboards with the mouse. From these action sequences, a model is generated and described in the ACT-Simple language, a construct similar to KLM [Salvucci and Lee 2003]. Some interaction types are automatically translated, i.e. mouse actions are converted into appropriate actions for the target platform, e.g., a mouse click to a tap with a stylus. The system has subsequently been used to integrate the modelling and simulation of user interface interaction as secondary tasks while driving [John, Salvucci, Centgraf, et al. 2004]. A slightly more specialised and fully integrated version is described in a follow-up [Salvucci, Zuber, et al. 2005]. The whole process of designing a mock-up interface, demonstrating tasks, specifying scenarios, running simulations, and displaying results has been packaged into a single application. Similar to the MAKEIT environment, CogTool provides a visual tool to define user models. However, MAKEIT additionally focuses on providing support for the actual implementation by generating source code, incorporates more non-functional parameters, and supports a wider range of interactions and extensions.

7.2 Creating Prototypes of Mobile Phone Applications

This section describes the architecture and interface of the MAKEIT development environment that allows quickly and simply prototyping applications for mobile devices. A common screen-based interaction process is reflected in the way it helps designing applications. A state graph data structure represents the possible flow of actions in a program. By creating such a state graph, the designer lays out the functionality supported by the application, the possible sequences of user actions, and the resulting visual behaviour of the mobile device.

Furthermore, the system is able to semi-automatically adorn transitions between states with additional non-functional parameters, such as KLM operators. The framework can then retrieve predictions of the interaction time of any possible (i.e. defined) sequence of actions by a potential user. These predictions are based on a modelled, deployed version of the application running on a real phone. The system is designed to support a variety of interaction techniques as listed below. Some common ones are directly integrated, whereas others can be customized and easily added. For some of those interactions, a detailed discussion can be found in our paper about physical mobile interactions [Rukzio, Leichtenstern, et al. 2006]. Example interactions currently used are:

- **Media Capture**: capturing audio and video, and storing or potentially analysing it is used in many applications, e.g. [Holleis and Schmidt 2007].
- **Visual Markers**: using the camera in the phone, marker-based interactions can be supported; this includes simple recognition of barcodes but also advanced augmented reality applications, e.g. [Rohs 2005].
- **Proximity**: based on proximity, actions can be triggered or application behaviour can be changed; one example is scanning for Bluetooth devices, e.g. [Mahato et al. 2008].
- **Gestures**: accelerometers built into phones offer many opportunities for interaction based on movements and gestures, e.g. [Kranz, Freund, et al. 2006].
- **RFID / NFC**: to capture the identity of a tagged object, RFID and NFC tags provide easy means; to implement physical mobile interactions, the identifier can be linked to further content, e.g. [Boyd 2005].
- **Location**: GPS or cell IDs are widely used to retrieve information about the user's location enabling location based services and interactive applications; e.g. [Hull, Clayton, and Melamed 2004].
- **External sensors**: ECG and oxygen saturation are examples of physiologic parameters that can be sensed for applications reacting to body signals, e.g. [Nuria and Flores-Mangas 2006]; other sensors can e.g. retrieve orientation [Holleis, Kranz, Winter, et al. 2006] or touch [Holleis, Huhtala, and Häkkilä 2008].

7.2.1 Generating the Application Behaviour

The overall concept of MAKEIT is similar to that of paper prototyping where typical steps are to start with a picture of a mobile phone with an empty screen and then to simulate pressing some key which results in preparing another picture and drawing content into the new screen. Next, the user is allowed to, e.g., touch an NFC tag and another screen is prepared. This process is repeated until all important states have been prepared. This is exactly how one can work with MAKEIT. However, it eliminates the difficulty of keeping track of the stack of pictures and the mapping between screens and actions.

One part of the user interface presented to the developer comprises an image of a modern mobile phone featuring the standard set of keys and an empty display (see Figure 73). All keys can be pressed using the mouse to emit events to the framework running behind the visualisation. Next to the phone are several buttons that can be used to simulate advanced interactions with the phone. Examples include taking a picture or touching an RFID tag. Since not all of those actions are supported by all phone models and new types of interactions are added as we speak, this list of buttons is automatically generated from an XML properties file which can easily be extended and potentially depend on the features of the phone model in use. Using the action buttons and the controls provided by the mobile phone, the developer can simulate actions with a simple click. This triggers a dialog in which the interface designer or developer can specify the new contents of the display. It can be a simple string or a URL / filename of a web page or image. Simple drawings can also be made in place, which is especially useful for people working with graphic tablets and directly mimics paper prototyping.

By repeatedly linking actions to visual elements, a linear sequence of screens can be created which represents the execution of a task in an application; however the majority of applications are more complex requiring richer application logic. This motivates the use of a state graph as defined in Definition 3 on page 96.

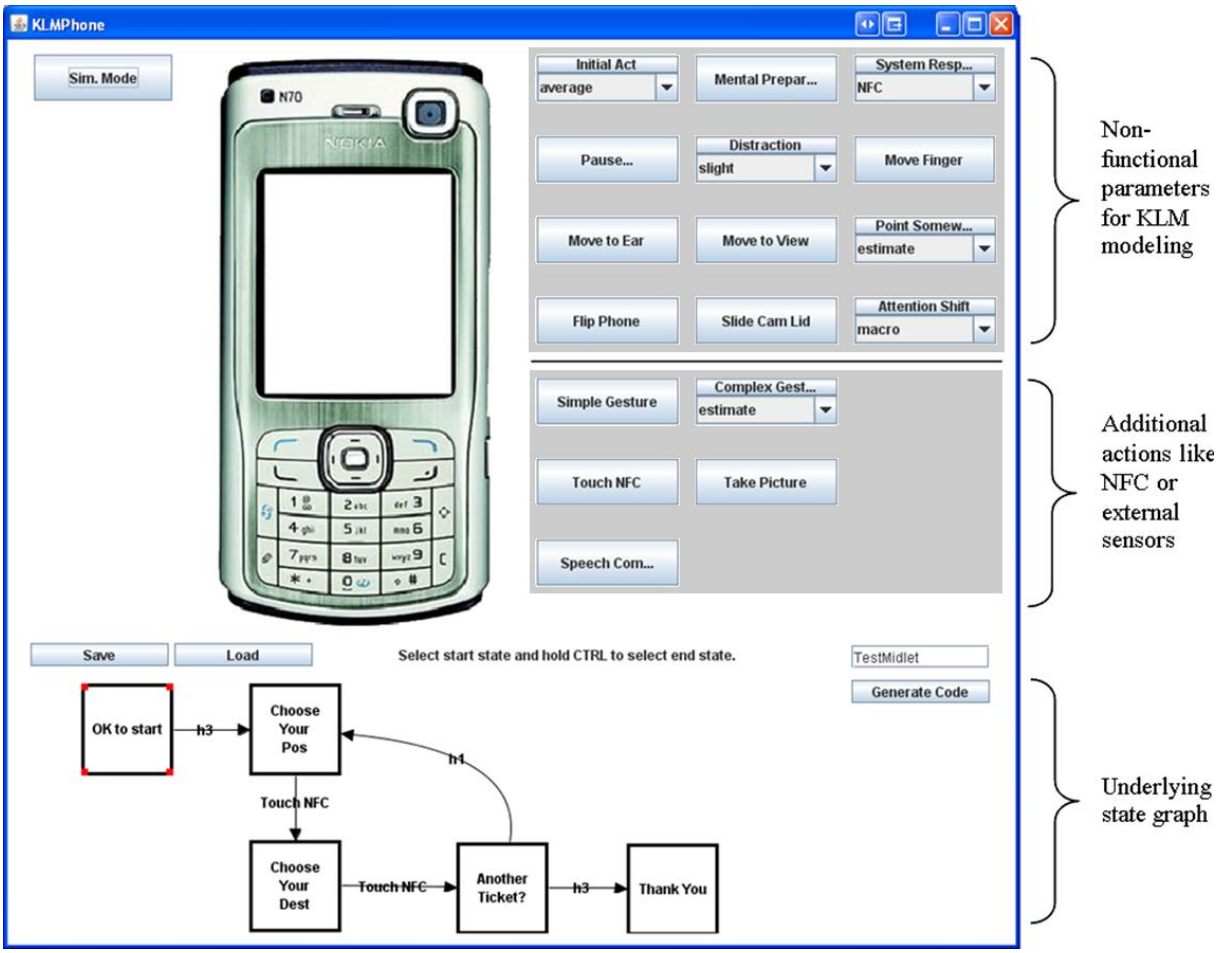


Figure 73: The keys in the simulated phone and additional interaction techniques can be chosen and the content of its display is updated accordingly.

7.2.1.1 Programming by Demonstration

The MAKEIT environment promotes the following steps to be executed in order to generate an application taking the expressiveness of user models into account during this process:

- Specify the contents of the mobile phone screens (type text / use an image / draw).
- Specify actions (press a key / activate a complex action like a gesture, RFID reading, ...).
  - ⇨ This builds a **state graph**
- Switch to demonstration mode and demonstrate a sequence of actions.
  - ⇨ This specifies a **path in the state graph**
  - ⇨ Use the model to check and secure specific properties.
- Automatically generate mobile phone code
  - ⇨ This generates a **MIDlet** and NetBeans project files.
- Adapt for final program.
- Optionally run a user study.

The arrows on the right indicate a potentially iterative process. After demonstrating certain tasks and analysing the built model, certain properties of the application have potentially been found that call for adaptations in the initial application design. Similarly, a user study run with the built application can still lead to necessary changes that can easily be incorporated into the state graph.

In the following, we go into further detail on the description and the actual use of the development environment.

7.2.1.2 Building the State Graph

MAKEIT provides a visualisation of the set of possible states as well as the transitions triggered by actions. A further part of the user interface presents the state graph described in Section 5.1.2. Initially, this comprises only the start state showing an empty phone screen.

The moment an action is triggered, a new node is created in the state graph and an edge is added between the current node and the new node. The edge is labelled with the name of the action (Figure 74). A dialog then prompts the developer for the content of the new screen. The new node is automatically selected, indicated by coloured dots in the corners of the rectangle representing the current screen contents of the mobile phone. After specifying the content of the new screen, the next action will continue the sequence and generate another node. This can be used to quickly create a vertical prototype that allows executing defined functionality in detail whereas not all functions that the application will provide when finished are supported.

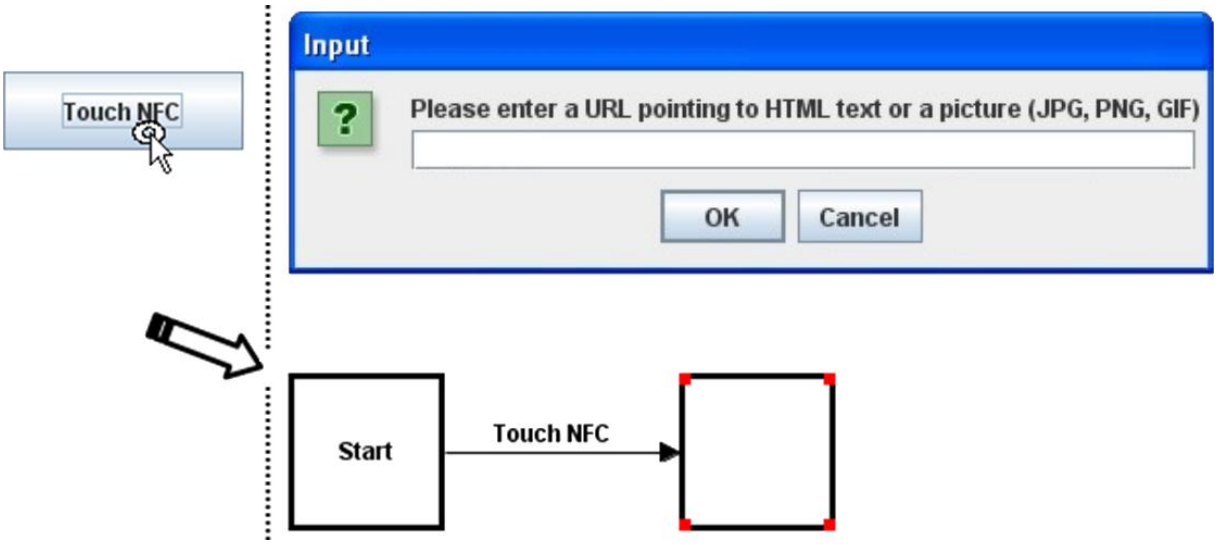


Figure 74: When triggering the 'Touch NFC' action, a new state is generated and a transition from the start state is added labelled with the action's name.

The creation of the state transitions is not restricted to a linear sequence. When a node of the state graph is selected with the mouse, the defined contents will be updated on the virtual phone's screen and the application is brought into this state. Demonstrating an action can be done in whatever state the application has been set to. This adds the possibility of leaving a state through different actions. One possible application is to implement different ways to reach the same goal, e.g., press a key, make a gesture or touch a tag. The left part in Figure 75 shows the application that the key '*8*' is used to browse through a list and another one, '*5*', to activate the selected item.

Adding edges to nodes, i.e. transitions to states, is only limited by the number of different actions allowed for the present state. Following the Disambiguation Property (two edges with the same source node must have different associated actions), transactions that already exist for a specific state cannot create a new edge. Instead, if such an action occurs, the existing transition is fired and the system changes the current state to the target of the edge. Such inputs from the user do not change the state graph. In this way, any sequence of tasks that has already been designed can be easily walked through and tested. This highly adds to the utility since people often go back to the beginning to recap the task at hand.

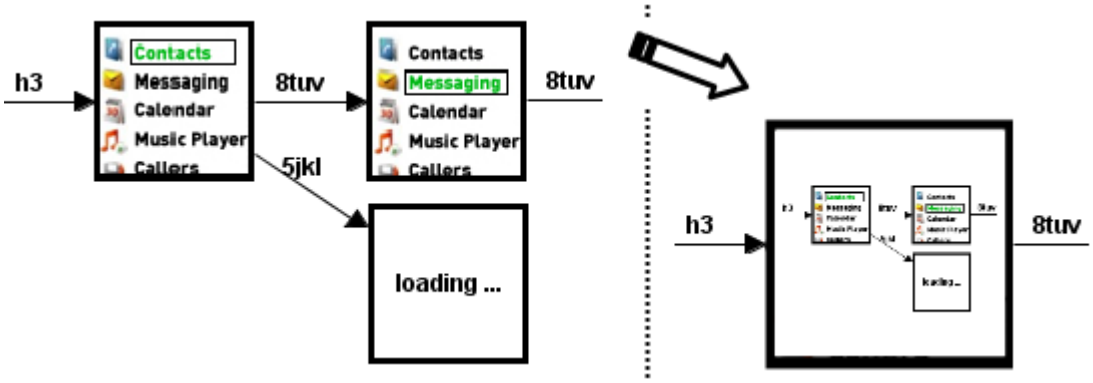


Figure 75: Reducing the number of visible states by condensing several nodes.

Merging States

One of the potential problems with state graphs is that the number of states can grow rapidly. The maximum number of states succeeding a node is only bounded by the number of different actions allowed for this node. However, in our analysis, we found that most applications, besides screens with highly dynamic content that are much better implemented in code anyway, do not need many screens. In addition, there are several possibilities to reduce the number of states. One is to **condense** several nodes into a super-node, as is often done to visualise and work with large hierarchical graphs (see Figure 75). We successfully used this approach for example in a graphical system for describing graph queries [Holleis and Brandenburg 2004].

A visually as well as semantically clear approach is based on the observation that applications often return to the same state after different sequences of interactions. Situations in which this occurs afford the merging of equal states. In the case of the visualisation chosen for this project, this means that it must be possible to combine two nodes (as shown in Figure 76). We define a **merging** operation in Definition 5 as follows:

Definition 5 (Merge Operation)
Merging two nodes $S_1 \in S$ and $S_2 \in S$ in a graph $G = (S, A)$, i.e. ***merge*(S_1, S_2)**, means

- for all nodes $X \in S$ such that an edge $a = (S_1, X)$ exists, add an edge $a' = (S_2, X)$ and copy the properties of a to a'; all edges a of the form $a = (X, S_1)$ are treated analogously
- delete S_1 (and all adjacent edges, i.e. those that have S_1 as source or target node) from G

Merging is only defined if no edges are added which would conflict with the Disambiguation or the Start State Property.

By definition, the Reachability Property is not affected by any merge operation.

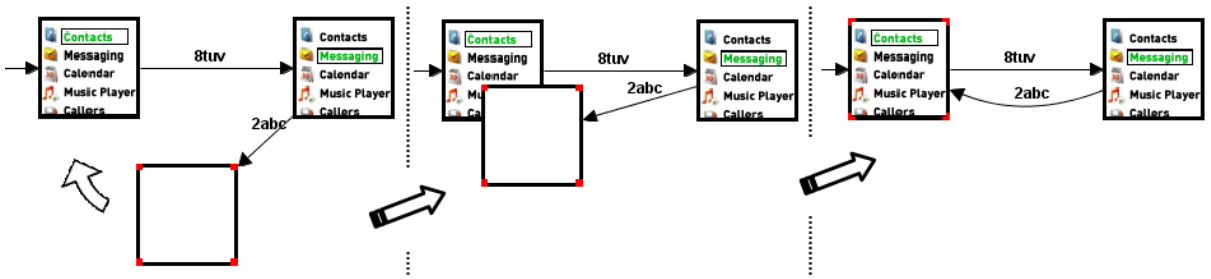


Figure 76: Merging two states by simply moving one node (empty) over another.

Merging states can introduce cycles to the graph which theoretically drastically complicates the automatic calculation of a visually pleasing layout of the state graph (it can, e.g., break planarity, i.e. the property that a graph can be drawn without any edge crossing). However, in our experience, most graphs seem to be fairly easy to layout since most cycles are very short. By moving the nodes in the view, the graph can also be manually adjusted anytime. Figure 76 shows how merging can quickly be performed in the user interface by simply dragging one node on top the other node.

Example

The example in Figure 77 demonstrates that the merging feature is absolutely essential for many situations like the aforementioned use of a list of items. Scrolling up and down through a list would repeatedly generate the same states. The figure shows a list that can be scrolled by pressing the number keys '*2*' and '*8*'. The '*5*' key selects the current item and switches to a state that handles the selected option in the list. This selection method can easily be replaced by, e.g., a gesture without inducing any other change in the graph. This example also illustrates that a node can be the target of several edges as the according state can be reached in several ways. It also keeps the number of states low by having only one state ('loading …') that is responsible for displaying a reaction to the selected option. One could also split the node in a way such that pressing the selection key will lead to a different state for each menu entry. Any combination of the two approaches is also possible.

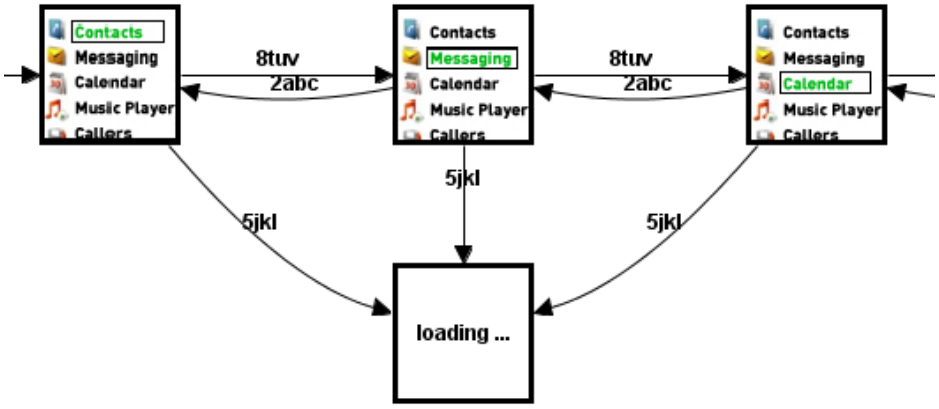


Figure 77: Designing list scrolling. When an item is selected (key '*5*'), the same state is reached, regardless of the previous state. Coding could then be employed to show a dynamic screen.

Another example for having several transitions to one state is an exit or error state. Applications may have a dedicated exit state, for example an 'off' state. Anytime an error occurs, an error state can be reached which offers fallback solutions. The approach can in general not be used, however, for a generic message state (presenting, e.g. a message like 'This action is not yet supported') since in most cases the application flow should return to the state that initially triggered the message. This would contradict the Disambiguation Property.

Design Space

We emphasise at this point that neither the state graph nor the tools to create it claim to be a full-fledged visual programming language. We leave difficult tasks to the places where it can be done best: the source code of the mobile application. The design space of the applications that can be created by using this mechanism only is clearly limited. For example, information cannot directly be passed from one state to the next, and it is not known which steps have led to a certain state. Although features like that could be added by using a richer data model, the simplicity of the chosen approach suffices to quickly start with and concretely test ideas and different interface and interaction designs. In [Holleis, Huhtala, and Häkkilä 2008], we added touch sensors to the standard keypad of a mobile phone. Using the MAKEIT framework, we were able to quickly develop and test several variations of a contact list showing preview information when the selection button is touched or an image gallery with zooming by touching. The contact list application with a list of four names, e.g., needs only 4·2 = 8 states. More details can be found below in Section 7.5.3.

7.2.2 Analysing Tasks during Application Creation

One of the important aspects in designing applications is to see and understand if and in what ways a task can be executed with a proposed design. During the design of the flow of an application, i.e. the creation of the state graph, a path finding algorithm can be employed. Selecting a start state s_a and a target state s_b, an algorithm finds all possible paths $p(s_a, s_b)$. Remember that a path is defined as a sequence of directed edges that connects one node with another. In this case, we limit the notion of paths to simple paths that do not contain a node or edge more than once. This avoids cycles and bounds the number and length of all paths by the number of nodes and edges in the graph. Note that a path does not necessarily exist between two arbitrary nodes. On the contrary, sinks, i.e. nodes that are not the source of any edge in the graph (for example states that indicate that a device cannot be reused) can only be the target node in a path, but no paths will start from those. However, the Reachability Property of the graph dictates that there will always be at least one path $p(s_S, s)$ from the start node s_S to any other node s in the graph.

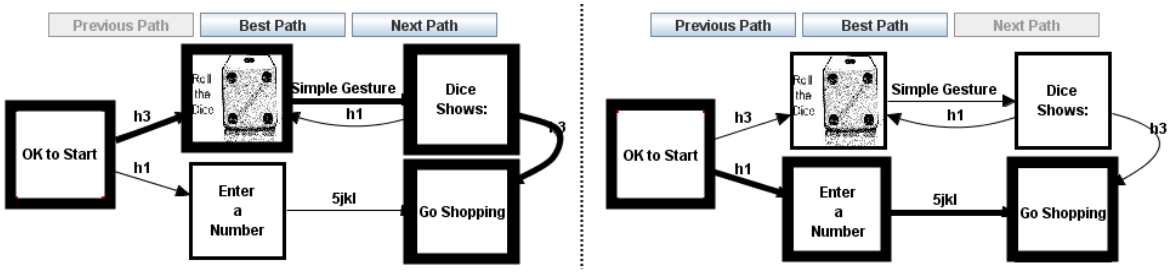


Figure 78: Two paths from the node 'OK to Start' on the left and the 'Go Shopping' node on the right.

In the graph visualisation, a path is shown by highlighting its edges as well as the traversed nodes of the path (Figure 78). There are potentially several paths between two states which can all be browsed and highlighted. The paths can additionally be used to provide an analysis of non-functional properties. This (and why there appears a '*Best Path*' button in Figure 78) is explained in the following sections.

7.2.2.1 Adding Non-Functional Properties

Non-functional properties are all characteristics not directly concerned with the semantics of an element. In the case of the transitions in the state graph, this means attributes of an action like the time necessary to execute it, the effort needed, the pleasure generated, or the privacy affected by it. In the following, we concentrate on interaction time characteristics and build on knowledge about the Keystroke-Level Model introduced previously.

We have already seen that, by triggering actions defined in the state graph, a task can be sequentially walked through and the state of the mobile phone is updated accordingly. To be able to additionally incorporate actions necessary to use the operator model of KLM, this part is elaborated in the user interface. After a version of the application has been defined using the state graph, the user can switch to simulation mode. The user interface is then extended with several additional actions. These KLM actions can also be easily configured and new elements can be added whenever new types of interaction are added in future phones using a property file.

In simulation mode, all actions can then be executed as defined in the state graph. Furthermore, additional KLM operators can be added at any time, in any state, in any order. Most of these actions have been introduced in one form or the other in the part treating the mobile phone KLM we developed in Section 3.4. Table 15 gives a quick overview over the meaning of some of the standard operators, see [Holleis, Otto, et al. 2007]. The general idea of those operations is that additional information about how a task is executed can be gathered and stored. The mentioned actions mostly concentrate on interaction times.

Table 15: Some non-functional operations supported in simulation mode.

Operator	Value	Details
Initial Act (average, self initiated, …)	4.61 sec.	Time necessary to retrieve and look at the phone
Mental Preparation	1.35 sec.	Time to mentally prepare for the next action
System Response	variable	The time the system needs for computations
Pause	variable	Interrupt for some amount of time
Distraction (slight, strong)	6 %, 21 %	Actions done while being distracted are slowed down on average by some factor
Move to Ear / Move to View	0.95 sec.	Time needed to move the phone between a state looking at the screen and one close to the ear
Point Somewhere (average)	1.00 sec.	Time needed to move the phone to a specific point (e.g. to touch a tag there)

As a simple example scenario, consider a poster that displays some products and advertises a URL. The task is simply to browse to this given website. A designer thinks about implementing one or more of the following three options: enter the URL by hand, take a picture of a marker on the poster or use the phone's NFC capabilities to retrieve the URL from a tag embedded in the poster. A simple state graph that is generated in less than two minutes is shown in Figure 79, left. Since one exit state has been attached to all three interaction methods, selecting the start and the end state will list all three interaction paths.

As next step, the details for each path can be demonstrated: in simulation mode, a separate window shows the action sequence of the currently highlighted path. In the example, from the start state, the hotkey 'h3' is pressed and the system prompts for the URL. The act of touching an NFC tag incorporates four steps: a unit of mental preparation is set to account for the time needed to prepare oneself for the interaction; in the rather coarse modelling of the KLM, this also includes the vague focusing on the target tag (action 'Mental Preparation'). Next, the movement of the phone is done (action 'Point Somewhere'). After the actually reading the tag ('Touch NFC'), the system needs some time to process that tag ('System Response (NFC)'), see Figure 79, right. There are heuristics that can be used to add operators not specified as actions (such as mental acts). However, the developer can always add such operators while stepping through the actions of the current path.

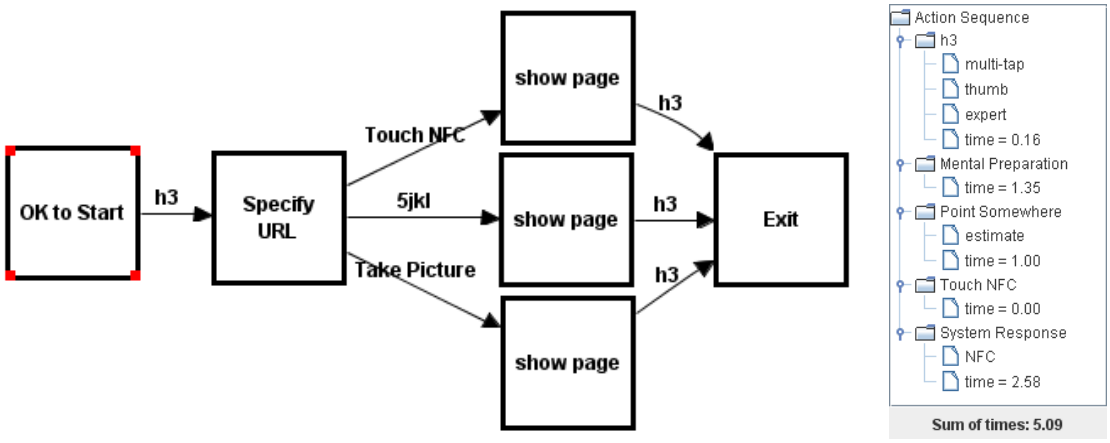


Figure 79: *Left*: Three different ways of specifying a URL: using an NFC tag, entering the URL with the keypad, and detecting a visual marker using the phone's camera. *Right*: Actions for the NFC interaction from the graph shown on the left.

7.2.2.2 Analysing the Augmented Path

The times for the described actions of the NFC interaction in the example are: hotkey (0.16 seconds), mental preparation (1.35), pointing (1.00), touching after pointing (0.00), system response time (2.58). This results in a total interaction time of roughly five seconds. The analysis of the other two interaction techniques results in 9.9 seconds (for an URL of 25 characters) and roughly six seconds (for a visual marker). In general, each path between start and end state can be associated with a usability measure such as the time that executing this path would take in real life. The system can then find the 'best' path which will be the interaction method that takes the least amount of time. In this example, the algorithm would suggest the NFC interaction. It should be noted at this point that several of the operations like reading NFC tags always result in the same sequence of KLM operators. Those additional non-functional actions can automatically be retrieved and saved. Missing steps, e.g., an anticipated period of mental preparation can thus be easily added to the transition in question.

As a further example, we recently used the mobile phone KLM to model different ways of interacting with more complex physical posters, e.g. for buying public transportation tickets. A graphical browser-based phone application was tested against one that used NFC tags embedded in the poster. Surprisingly, the model predicted that the text input variant would be considerably faster (two minutes instead of close to three minutes). We ran several tests with different users and found the model to be remarkably correct. Interestingly though, all users had the false subjective impression that they had been faster with the NFC version which points into the possible direction of adding such subjective opinions to the modelling system as well.

It is also important to see that such action sequences, augmented with interaction information, can not only be used to compare one method to another. A representation of the modelled sequence of actions is extremely useful to find bottleneck parts of the interaction sequence, i.e. those that are responsible for long interaction times. In the scenario under consideration, one of the problems identified was the time lost with checking the feedback of the phone after each single reading of a tag. A proposed solution is that detailed feedback is only given after a series of interactions. This can easily be changed in the state graph of the application by removing the intermediate feedback states and adding a later feedback state.

7.3 Implementation

Implementing applications using many different programs, hardware and software platforms, communication protocols, and programming languages is, in general, difficult. To counter that, we started the open source project EIToolkit which has been described in Chapter 4. To briefly recapitulate, it is a component-based architecture in which each component is represented by a proxy-like object called a stub. These stubs translate messages between a general communication area to the specific protocol of the devices and vice versa. Any component can then register to listen to messages directly addressed to it or broadcast to all. This enables exchanging components on the fly. The system also allows changing the protocol of the messages on a per component basis. The toolkit currently supports a simple proprietary format over UDP or TCP as well as OSC and RTP. The last two are widely used protocols for audio and multimedia systems and streams. Several microcontroller platforms can be connected through existing stubs as well as over a serial connection. Sample stubs are available, e.g. for media players or direct MIDI output. See Chapter 5 for more details.

Independently of the MAKEIT application, we implemented the KLM semantics in an EIToolkit application as described in Section 5.3.1. Thus, we have a component that is practically platform-independent and can be used remotely. Specific control messages choose the type of KLM, e.g. specific to mobile phones or to another set of controls. After that, queries are sent to the stub presenting information of an action. For a key press of a hotkey on a mobile phone, a sample message might contain the ID '*HotKey*' and parameters '*1 thumb*', '*expert'*. The KLM stub browses its known operators and, if available, sends an answer containing a time value back to the sender of the query, e.g. the MAKEIT system. This toolkit integration proves to be quite useful since the implemented KLM can easily be updated and extended without changing the applications that make use of it. It also allows applications to employ the KLM without direct knowledge about its implementation and manner of operation. A further advantage is that a KLM analysis can often be attached to applications (provided they present some information about their use) and interactions can be analysed without the need to especially design the application for that purpose.

Data Structure of the State Graph

For the implementation of the state graph and its visualisation, we reused most code from our graph visualisation toolkit called Gravisto [Bachmaier et al. 2004]. The data structure provided by Gravisto has been adopted without changes. Beside the basic features of graphs consisting of nodes and edges, the toolkit provides a mechanism to attach arbitrary data to any of the graph elements (nodes and edges) present in a graph. This data is stored in the form of hierarchically structured attributes of primitive as well as composed types. This structure is extremely helpful when several pieces of data have to be managed with graph and matches well with our attribute mapping function (see Definition 1 on page 96). The graph elements are used to store the states and contents of the display as well as information about the transitions between states and detailed information about timing and other model parameters.

The creation and manipulation tools of Gravisto were adapted to ensure the concordance with the state graph properties and to enable additional features like merging states. This was easily possible due to the modularity of the code structure. Some visual features have been added to correctly display state images. Gravisto also enables storing a generated state graph in a file in the standard graph data format GraphML[93] which is based on XML and supports custom attributes. A saved state graph can then be loaded without data loss and the connection to the mobile phone visualisation in the user interface is immediately updated.

Figure 80 shows part of a serialisation of a state graph to GraphML. Most details regarding graphical properties such as node shape, size, and position have been omitted. The code generator described in the next section creates a canvas (i.e. screen) for each of the states and a command for each of the transitions. Depending on the type of transition, the trigger of each command is implemented differently. Keystrokes, for example, are implemented using a special call-back function of the Canvas class used to display screens.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml http://graphml.graphdrawing.org/xmlns/graphml/graphml-
structure-1.0rc.xsd">

<key attr.name=".directed" attr.type="boolean" for="graph" id="graph_directed"/>
<key attr.name=".content" attr.type="string" for="node" id="node_content"/>
<key attr.name=".transition" attr.type="string" for="edge" id="edge_transition"/>

<graph edgedefault="directed" id="G">
  <data key="graph_directed">true</data>

  <node id="n0">
    <data key="node_content">OK to start</data>
  </node>

  <node id="n1">
    <data key="node_content">Specify URL</data>
  </node>

  <node id="n2">
    <data key="node_content">show page</data>
  </node>

  ...

  <edge id="e0" source="n0" target="n1">
    <data key="edge_tranisiton">h3</data>
  </edge>

  <edge id="e1" source="n1" target="n2">
    <data key="edge_transition">Touch NFC</data>
  </edge>

  ...

</graph>
</graphml>
```

attributes

states

transitions

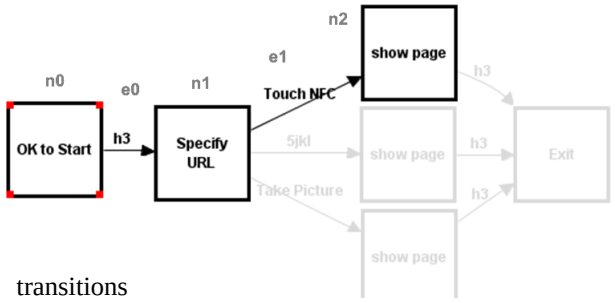


Figure 80: Annotated excerpt of the GraphML description of the state graph displayed in Figure 79.

[93] GraphML, file format for graphs; project page, *http://graphml.graphdrawing.org*

7.4 Mobile Device Source Code Generation

The whole semantics of the application under development is stored in the state graph. Nodes contain the data for states and the contents of the screens that will be displayed on the device. Edges represent actions from one state to another and additionally store information about non-functional parameters associated with transitions. A dedicated framework component transforms the state graph into a MIDlet, i.e. a program for the Java ME virtual machine which can be compiled, moved to, and run on many modern phones. Since the created application may need to be complemented with code changes, e.g. for dynamic screen contents, project files are generated that can be opened in the NetBeans development environment. The program can then be extended, compiled, downloaded to a phone, and tested there. Alternatively, the files can easily be imported into other development environments such as Eclipse. The manifold features of an integrated development environment such as syntax highlighting, choosing the target platform and debugging can thus be exploited. Of course, this also eases making quick alterations and additions to the code itself as is currently necessary for implementing the dynamic content of a screen.

In this process, the state graph is converted into a set of conditional statements. If an event named *a* occurs in a state *S*, the state *T* is loaded if and only if there is a transition (*S*, *T*) labelled with the action *a*. This is a common and easily understandable way to program such applications. In a mobile phone application, each screen is represented by an object. We use a custom sub-class of the Java ME class `Canvas` to write code that can load and draw images as well as render text to the screen. It is a low-level implementation of a screen and can also receive key events from the phone's keyboard.

It should be noted that it is the responsibility of the phone's operating system to choose how exactly to display hotkey actions. If, for example, several hotkey actions are defined, those can be each associated with a different physical key or they could all be packed as items in an options list opened through a single physical key. This might not be the look and feel intended by the programmer using our state graph approach. However, this is a restriction given by the current mobile phone programming model and not by the code generation module. To the best of our knowledge, all existing emulators suffer from similar problems.

getimage.template:

```
public Image get_$name$() {
    if ($name$ == null) {
        // Insert pre-init code here
        try {
            $name$ =
                    Image.createImage("$relpath$");
        } catch (java.io.IOException exception) {
            exception.printStackTrace();
            return null;
        }
        // Insert post-init code here
    }
    return $name$;
}
```

getimage.template, instance:

```
public Image get_image0() {
    if (image0 == null) {
        // Insert pre-init code here
        try {
            image0 =
                    Image.createImage ("/hello/ab.jpg");
        } catch (java.io.IOException exception) {
            exception.printStackTrace();
            return null;
        }
        // Insert post-init code here
    }
    return image0;
}
```

Figure 81: *Left*: template for the methods that create an image from a file. *Right*: instance of this template with *$name$* = 'image0' and *$relpath$* = '/hello/ab.jpg'.

Code generation is fully generic and uses templates for all methods. One template, for example, is used to generate function for initialising the application. This is the place to add code such as to start Bluetooth connectivity. Figure 81 shows a template and an instance of the method responsible for loading an image. All those templates are used by the code generation module to assemble the MIDlet file (which itself is a template including several sub templates). By adapting and extending these templates, the generated code can be quickly updated to novel functionality. It is also possible to integrate the generated code into existing MIDlets.

As can be seen in the example template, the default comments (e.g. '*// Insert pre-init code here*') that the wizard of the NetBeans Mobility Pack generates have been left intact. Together with the generation of corresponding project files, this means that the graphical designer component of this particular IDE can also load the generated files and all graphical and code-based advantages of that environment can be used.

In order to follow the general goal of a component-based architecture, the implementation of the code generation is also bundled into a separate component. It can be used as a standalone program that, given a state graph as input, generates a mobile phone application from it. This also implies that potentially any other application that produces a state graph can use this implementation to generate a mobile phone application. Of course, certain conventions for the encoding of the description of states and transitions into the attributes of nodes and edges have to be followed.

Interaction Patterns

Some parts in the state graph may appear several times when generating applications. Some of them are even independent of the type of application. One example is a scroll list as is depicted in Figure 77. In order to take advantage of such sub graphs, we added the possibility to define and use patterns of such recurring graphs which is a common technique in model-based development. There is only one obligatory element that has to be specified in order to create such a pattern: a graph has to be (programmatically) created that represents the pattern. In the example, this could consist of two nodes n_0 and n_1 with specific labels and two directed edges (n_0, n_1) and (n_1, n_0) as shown in the dialogue in Figure 82. An interaction pattern can depend on a series of options defined for this particular pattern. The scroll list example offers the possibility to set the number of consecutive items within the list. There are various possibilities to graphically represent such a list. In Figure 82, simple text with HTML mark-up is used. The list could also consist of several images which enables the use of complex graphical menus (see Figure 77 for such an example).

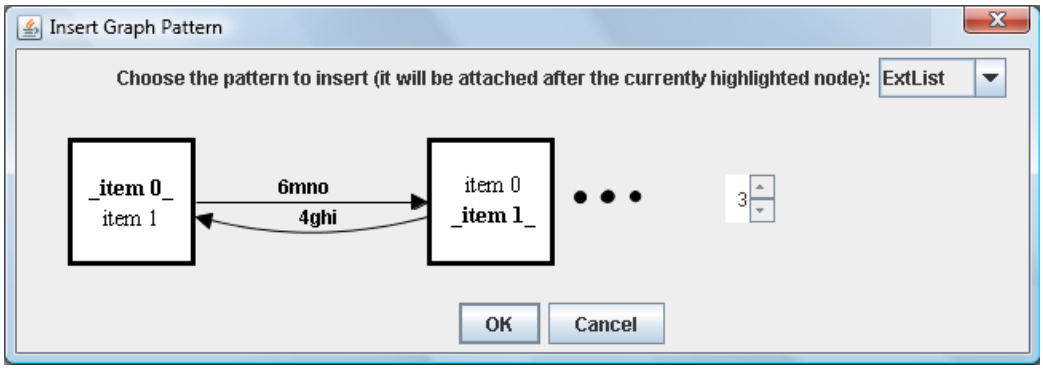


Figure 82: Dialog to create a scroll list with *n* entries. The currently selected item is highlighted using underscores ('_') and bold text specified with HTML.

As hinted at in the previous paragraph, the current version of the tool still incurs some effort in programming. However, this process could easily be made considerably simpler by allowing the selection of a subgraph as a pattern. Still, integrating options such as the size of the graph or a specific numbering of nodes is difficult to implement in such a manner.

These patterns enable the generation of a library of application parts that can then simply be added to an empty graph or combined with an existing graph. This supports code reuse, can help to save screen space, and accelerate the development process.

7.5 Capabilities and Examples

7.5.1 Supported Interactions and Features

The framework is highly extensible as will be detailed below. Currently, we directly support the following types / classes of interactions:

- **User modelling**: KLM operators are placed according to corresponding actions. The user can add or modify suggested operators and thus fine-tune the model.
- **Buttons / keys**: still the most important way of interacting with a mobile device is through its keys and special buttons (hotkeys). All buttons to which the Java ME API provides access are supported by MAKEIT.
- **RFID / NFC**: it should be noted that the framework is very open to extensions as we show in further detail below. As one example, we integrated the PMIF framework described in [Rukzio, Wetzstein, and Schmidt 2005]. This allows for interactions with RFID and NFC tags as well as visual markers. If a transaction with some RFID tag has been defined, for instance, code is generated that waits for and acts on the reading of a tag with the specified ID.
- **Bluetooth**: the current implementation of the framework also supports advanced interactions using external Bluetooth sensors and devices. A transaction using this communication channel is created by specifying the event sent through Bluetooth to the phone. This allows arbitrary sensors and actuators to be used in conjunction with the phone's functionality.
- **EIToolkit**: the development environment allows integrating EIToolkit events into the prototyped application. This means that all types of interactions enabled by that the connection to the EIToolkit can directly be used (see Chapter 5 for examples). This connection is not directly available on the mobile platform, though.

7.5.2 Sample Applications

In order to get a better impression for what kind of applications can be prototyped with the framework, we present some examples that have been implemented with MAKEIT.

NFC poster interaction

RFID and near field communication (NFC) currently become heavily used in areas such as supply chain management or mobile payment. As a sample application, Figure 83 shows the state graph of an interface that lets a user choose a ticket based on the number of people and zones for which it should be valid. The application uses a poster with embedded NFC tags which is used by simply clicking on the appropriate virtual buttons with their NFC enabled phone.

This example graph assumes a correct sequence of selecting the parameters. However, for example in order to evaluate different approaches, more transitions could simply be added. On the other hand, this would imply that an intermediate error state would have to be added to signal that not all parameters have been set before choosing the payment button.

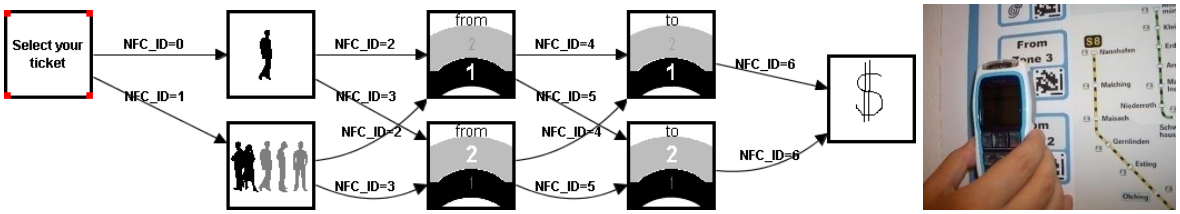


Figure 83: *Left*: state graph for an application where users can buy a public transportation ticket using NFC tags on a poster. *Right*: fragment of the poster with NFC tags attached to its back.

In this example, one can see the different ways of generating the content of the different screens in the phone application. The first state on the left uses simple text while the states in the second column show some images taken from the web. Next, images quickly drawn in a graphics program have been imported, and the final state features a dollar sign sketched directly in a dialog within the development environment.

Touch browser

Browsing web pages and text or viewing larger images on small devices is not straightforward to do and several strategies have already been implemented. In Section 6.4.3, Touch Input on Mobile Phone Keypads, we introduced the setup and a study about touch enabled keypads on mobile phones. Figure 84 presents part of the state graph of a prototypical image viewer application for small devices that offers a preview feature. On the phone, the keys '*2*' and '*8*' are used to move a cursor upwards and downwards, respectively. In the figure, one can easily see that merely touching one of those keys brings forward the content of that side. The moment the user stops touching the key, the view jumps back to the previous view. Only by pressing the respective key, the view permanently changes in the selected direction. Thus, touching a key provides a preview of the action that pressing it will perform. One could of course also choose appropriate hotkeys for these actions; the interface of MAKEIT allows changing such settings with ease.

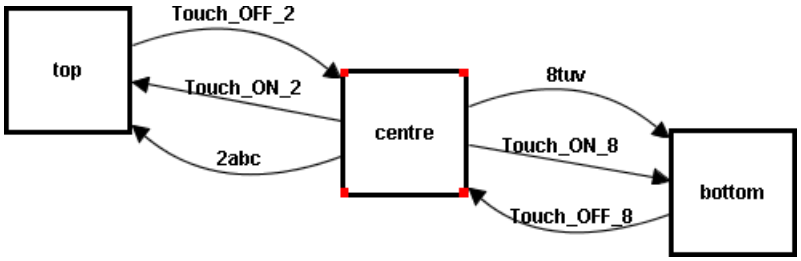


Figure 84: Prototype of a touch based browser or image viewer. Touching shows a preview of the part of the content that a click on the same button will bring up.

Touch Input on mobile phone keypads (see Section 6.4.3)

Another idea to use the touch enabled phone keypad was to enhance the standard phone contacts list with additional functionality. Whenever a specific contact is selected and the selection key is touched, additional information about that contact is displayed (such as a picture or the time or amount of the last conversations with that contact). Only when the selection key is pressed, the entry gets selected and standard information is shown. The development of this application took some time, especially because of the necessary combination of various tools such as Python or Java ME and Flash Lite. Using MAKEIT in conjunction with the EIToolkit, the same application can be prototyped within minutes. Figure 85 shows the state graph that encapsulates the full application logic for the first three entries of the list.

Although the state graph representation blurs details, one can easily see that the top row of three boxes models browsing a list with the '*2*' and the '*8*' key. *Touching* the '*5*' key pops up a small window displaying some information as can be seen in the phone screen on the right in the figure. *Pressing* the '*5*' key in any state selects the entry and shows full name, telephone number, etc as expected. One could argue that pressing a key always involves touching it first and that the transitions between the list selection states and the selected entry states could be left out. However, since there is a slight delay in recognizing touch (in fact, we deliberately introduced one for this scenario), this means that the original application behaviour did not change. Only after a slight hesitation or delay while touching the key, the popup is shown.

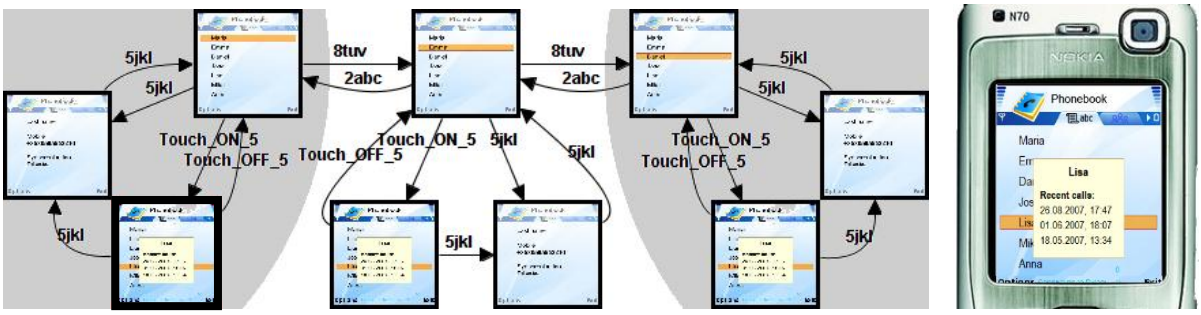


Figure 85: *Left*: state graph of phonebook application enhanced with a touch key based preview. *Right*: screen in the phone emulator. The currently selected state (bold frame) is always visible.

Text entry acceleration

Another application for touch sensors on mobile phone keys that we currently evaluate can speed up multi-tap text input. Figure 86 shows a state graph that allows entering the word '*deep*' using multi-tap. When a letter appears on the same key as the preceding one, there are currently two ways of entering this letter (as opposed to changing the one last typed): either one waits a short time (approximately one second on most phones), or one hits a specific hotkey. In the graph, the first is implemented using a special timer transition with one second as value. The second is specified using the hotkey '*h3*'.

In order to see how much this method could speed up text entry, we determine all involved actions. The studies in [Holleis, Otto, et al. 2007] did not look into switching between the number keypad and the hotkey panel during typing. We conjecture that it adds two finger movements between the keypad and the hotkey area adding another 0.62 seconds (2·0.23 + 0.16 seconds). In a quick, informal trial on a Nokia N95 phone which has its hotkeys on a separate, slightly elevated panel, we found that people typed a given sentence of 30 characters with 27.0 words per minute if there were no consecutive letters on the same key, and with 21.5 words per minute for text with, on average, every sixth letter needing a measure of disambiguation. Calculating the overhead incurred by the hotkey results in a value of 0.57 seconds which corresponds very well to our prediction of 0.62 seconds.

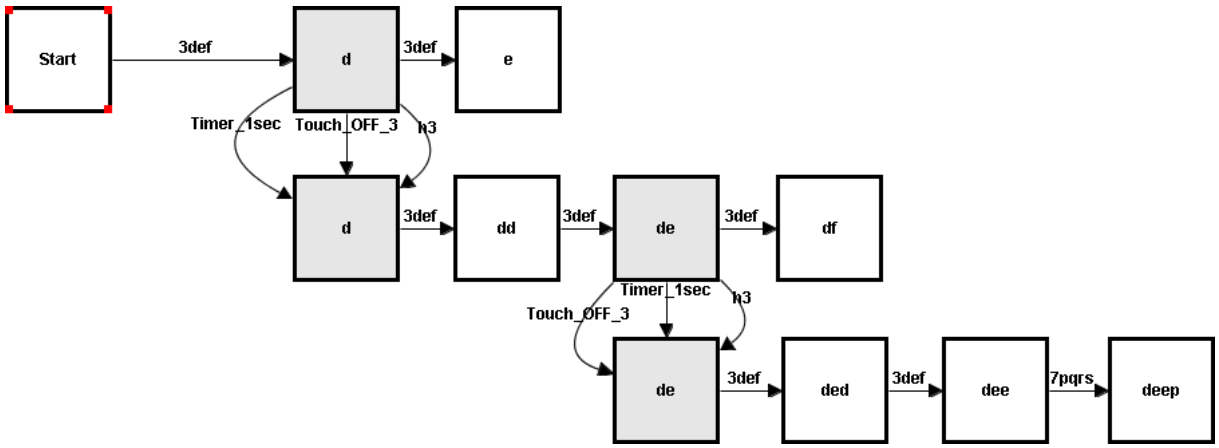


Figure 86: State graph showing three different methods in multi-tap text entry (transitions between the grey states) to enter letters which are on the same key as the one before: waiting for a timeout, pressing a dedicated hotkey, or releasing the current key. The word entered is '*deep*'.

In the state graph, we implemented a third way of disambiguating letter input. If one wants to change the letter just typed, the finger stays on the current button and simply presses it again. On the other hand, when a new letter should be added, a user briefly lifts the finger before pressing the button again. This decreases the slow-down per use from approximately 0.6 seconds (or one second when using the timeout method) down to 0.31 seconds. This value results from adding 0.08 seconds for lifting the finger (assumed to be half the time for a physical keypress) and 0.23 seconds for touching, i.e. the value specified for an arbitrary finger movement. We can show that, in English, approximately every 10th character is on the same button as its predecessor: with a random selection of various articles from the Financial Times UK with a total of roughly 10000 letters, we calculated the relative frequency that two consecutive letters are on the same phone key to be 10.1 %; using the 700 English sample phrases in the set provided in [MacKenzie and Soukoreff 2003] resulted in 9.63 %; in another corpus with more than 10.000 real English text messages provided in [How and Kan 2005], this frequency was 11.4 %. Even though these values are relatively low, text entry speed for the example above could be increased from 21.5 to 23.8 words per minute (or even 26.8 if people had used the timeout method) with this new method.

The graph in Figure 86 shows that it would be very time consuming to demonstrate the entire semantics of multi-tap text entry. However, by quickly demonstrating the different ways of typing one or two words, an application can very quickly be generated that allows demonstrating and evaluating the idea. It also creates a code structure which is easy to generalize to arbitrary letter input.

7.5.3 Extensibility of MAKEIT

For actions that are not yet directly integrated into the prototype and are added afterwards, stubs are generated that leave room for the developer to add the concrete code that implements the action. As mentioned above, the code generation component uses several template files that contain method stubs and code excerpts. If necessary, these templates can be adapted and extended to work for new interaction techniques. In the following, we briefly present the process to add new functionality to the MAKEIT system. As an example, we use the touch enabled buttons that we implemented for the studies in [Holleis, Huhtala, and Häkkilä 2008]. Events from the touch sensors are sent by Bluetooth to an application running on the phone. Section 6.4.3 presented more details.

There are three steps necessary to add such novel features to the MAKEIT system. First, add code that captures the new event, i.e. registers when a button is touched, to the '*MIDlet.template*' file. In the following code excerpt, an auxiliary class TouchConnection is used that opens a Bluetooth connection.

```
public class $name$ extends MIDlet
        implements CommandListener, TouchEventListener, KeyListener {
    TouchConnection touch;
    ...
    private void initialize() {
        touch = new TouchConnection(this);
        touch.start();
    ...
}
```

Second, add a new action element ('*Touch*') to the '*transitions.xml*' file:

```
<transition mode="plan" name="Touch">
    <options>
        <option>ON_0</option>
        <option>ON_1</option>
        ...
        <option>OFF_*</option>
    </options>
</transition>
```

This second step will add a button with a dropdown list to the interface of MAKEIT which can then be used to create transitions between states according to the specified option. The code generator module uses, besides the direct edits in the MIDlet template, the information from the XML file. In this example, using the specified name '*Touch*', it will automatically create an '*onTouch(String option)*' method called when such an event is captured and an interface '*TouchEventListener*' that contains the signature of this method. This enables developers to easily generate the code that actually captures and passes the event. In this case, the implementation of the code generation is particularly easy since it is nearly identical to existing code treating standard keypresses.

An optional final step enables the modelling component to integrate predictions about the new technique. The KLM stub presented earlier also builds upon an XML file which describes the timings associated with each action. The following code excerpt shows how such values can be added.

```
<prop name="Touch">
    <klm>
        <options>
            <option opt1="ON_0">0.23</option>
            ...
            <option opt1="OFF_*">0.08</option>
        </options>
    </klm>
</prop>
```

7.6 Discussion and Summary

7.6.1 Initial Evaluation

The prototype has not yet undergone a full formal evaluation. However, we got initial feedback on the prototyping and analysis process by demonstrating the system to and carrying out interviews with four experts working in different areas of developing and evaluating pervasive computing applications as well as students of a user interface master's class.

We saw that the main user interface quickly engages people to try to interact with it. Even without initial explanations, all were capable of grasping the idea that they could generate application logic on the fly. It became obvious that it was not clear in the beginning that more than linear sequences of actions could be created. After finding out or being told that it is possible to interact with the state graph itself, most people intuitively began to merge states by moving nodes. Demonstrating and refining the corresponding KLM model proved to be more difficult and indicated some necessary adjustments in the user interface. However, this was in part due to the missing familiarity of the participants with user modelling in general. All participants saw the advantage of graphically programming by demonstration and that arbitrary screen content could be used. The more programming oriented users initially asked for more complex visual control structures but also agreed that shifting complex aspects to the source code makes sense. They appreciated the possibility to quickly create prototypes and provide the starting point for more advanced applications without being hindered to implement whatever they want. The users agreed that the environment helps people concentrate on what they can do best: designers can quickly test ideas and create interaction sequences while developers can focus on implementing. The fact that screens could also be drawn in some separate graphics program was especially valued by the design oriented people since, when creating paper prototypes, they often assemble images and text using their own tools. Similarly, developers positively mentioned that they could use their own tools to write their code.

7.6.2 Strengths of MAKEIT

With MAKEIT, we address the gap between low-fidelity paper prototyping and high-fidelity implementations of mobile phone applications. The framework is used to quickly create functional prototypes for mobile devices supporting advanced types of interaction. In particular, it focuses on the need to easily create prototypes and help in evaluating and deciding between different interaction designs using a predictive user model.

The development process is based on state graphs eliminating the need to remember the order of paper material. The advantage of paper prototypes to be able to quickly react to unforeseen events during user studies remains. The required setup time can be slightly longer than for paper prototyping but considerably shorter than for implementing. Using state graphs, several types of errors like unreachable states can be avoided by design. The state graph also provides an easy way to understand and visualise application logic. We presented several approaches that counter issues when working with state graphs such as condensing and merging of states.

One of the most prevalent features of the MAKEIT development environment is its ability to allow programming by demonstration. This enables many users to create programs that otherwise would have difficulties in developing applications due to a lack of programming knowledge. The paradigm of programming by demonstration can easily be brought beyond standard interactions such as using the phone's keypad and display. We showed that the incorporation of advanced mobile phone interactions, including reading NFC tags, taking pictures, and reacting to external sensors is equally easy within the graphical user interface by using simulations. The application with all actions can be simulated within the tool. Additionally, a code module is provided that automatically generates a MIDlet, i.e. a program that can be run on an appropriate mobile phone.

MAKEIT employs a mechanism to create Keystroke-Level Models (KLM). While demonstrating a task, the necessary KLM operators are automatically generated and stored within the state graph description of the application. This model can then easily be used to make predictions about user performance for specific tasks, find bottlenecks within the application behaviour, and compare various ways of performing a task in terms of completion time. By simply selecting a state, the environment can provide all possible sequences of actions that lead to this state and provide corresponding KLM information. We argued that by integrating such a model as early in the process as possible, several issues and later changes can be successfully avoided.

8 Summary and Future Work

In this thesis, we described our results in research in the field of human-computer-interaction focusing on support for rapid prototyping and predictive user modelling. We also provided extensive means to combine these and introduced systems that enable quickly generating pervasive computing applications using the power of user models to predict certain usability factors early in the design process without the need for costly user studies. Key contributions of this thesis are the following:

- Introduction and extensions of user models for advanced mobile interactions.
- Provision of powerful tool support for the development of pervasive applications (EIToolkit).
- Design and evaluation of innovative interaction techniques and applications within pervasive computing.
- Design and implementation of innovative combinations of prototyping and model generation (MAKEIT).

8.1 Summary of the Contributions

Pervasive Computing and User Models

In order to set the scene of the work presented in this thesis, we provided an overview of pervasive computing and associated application areas and terms. This also served to demonstrate the breadth of our own research in this area by identifying and associating corresponding own projects and results within these categories, e.g. tangible computing ([Terrenghi et al. 2005]), wearable computing ([Holleis, Paasovaara, et al. 2008]), and mobile computing ([Holleis, Huhtala, and Häkkilä 2008]). We reused some of these projects in later chapters as proof of concept and examples for more general approaches such as prototyping and modelling methods.

As an introduction to one of the main topics in this thesis, we provided a concise description of methods to create and apply prototypes for pervasive applications. We included a discussion on the pros and cons of prototyping in this particular area and showed its importance in helping to assess and improve the usability and applicability of a product. We treated two ways of evaluating a prototype or a finished application or device in more detail. One employing user studies, the other relying on models that predict and explain certain behaviour and performance of its users. A discussion on advantages and issues in employing such models for testing ideas, prototypes, and applications was followed by a detailed treatment of user models in the area of human computer interaction.

We gave an in-depth introduction into the definition, use, and variations of user models including descriptive models (such as Buxton's Three-state Model and Guiard's model of bimanual skill) and predictive models (such as Fitts' Law, task action grammars, and state transition based systems). Much of the work presented here is based on a specific model of the latter category, the Keystroke-Level Model (KLM). In order to focus on and understand its use and features, we provided an introduction and treatment of cognitive user models including the Model Human Processor, a selection of cognitive architectures, and a variety of instances from the GOMS family of models very much related to the KLM.

Specifically, we gave the definitions, showed ways of applying, and provided examples of GOMS and KLM. A profound discussion on pros and cons and application areas of these models has been retrieved and collected from an extensive list of published research, projects, and personal experiences. This provides an introduction as well as a list of arguments which other researchers and practitioners can rely upon to check whether such an approach could be useful for them. Such user models are a simplification and formalisation of human cognition and try to predict human behaviour based on their model assumptions. They have been applied, refined, and validated in many studies and projects in the last 25 years. Still, new types of interactions require constant updates in the expressive power of these models. We considerably contributed to this process by extending the original KLM for advanced interactions with mobile phones. Such interaction types have emerged recently and we actively participated in their development and evaluation, [Rukzio, Leichtenstern, et al. 2006]. Therefore, we provided an introduction to advanced mobile interactions and an extensive review of related work for this area.

Such interactions include the handling of visual markers, gestures, and RFID tags, with which no models such as the KLM have dealt before. Also, since the settings in which applications are used that employ such novel features differ considerably in terms of mobility and type of interaction, aspects such as distraction and distribution of attention had to be taken into account as well. We described our novel results that identify, measure, and model advanced mobile phone interactions.

Finding, isolating, and measuring KLM operators is described in detail since it is important to understand potential underlying assumptions for their application [Holleis, Otto, et al. 2007]. It involved several types of user studies in different environments. In that process, we identified and introduced seven new KLM operators, adapted three operators from the original formulation of KLM, and were able to confirm the use of two original operators in the new setting. Novel types of interaction have been described including operators for shifting the attention between different parts on the phone or between the phone and the environment. Other important aspects involved novel types of interaction like gestures, visual tag recognition, and the use of NFC tags. In order to model distraction, we introduced a novel multiplicative operator that is adjustable for various settings.

To facilitate reproducing the results, we described the setup and execution of an extensive set of studies to measure interaction times and to assign performance values to the identified operators. We employed various techniques such as frame-by-frame video analysis, mobile phone applications, and eye tracking to extract these values. We believe this also provides an important contribution in itself and conjecture that these techniques can also be reused in different settings and for further types of future interactions.

We did not focus on text input for mobile devices in these studies but provide a concise overview of the set of existing work and operators handling various types of text input such as predictive methods like T9.

Our resulting Keystroke-Level Model for advanced mobile interactions has been validated in a study and is available for free. In this study, we compared data predicted by our model with data observed within that study. We examined two different types of interaction for mobile phones, one using a mobile web browser and one in combination with a poster enhanced with near field communication tags. The results showed that the deviations of the KLM predictions were small. Most importantly, the relative difference in task completion time between the two methods of interaction has been predicted exceptionally well (a model prediction of 30 % compared to an observed average speed loss of 31 %). Additionally, we compared some of our operator values to related approaches and thus further supported the validity of our results.

Throughout this work, we provided examples of the application of such models to real world scenarios. We showed that this represents an objective means to retrieve usability information from an application (or a model of an application) and can be used to design, understand, and streamline programs. We also described selected further extensions contributed by other researchers and identified additional research areas such as tangible computing and car interfaces where we are in the process of providing insight into how to user model these types of interactions.

Prototyping and Toolkit Support

Developing pervasive applications is still a difficult task. Even though the research area has shifted into focus in the last years, development support is still scarce compared to conventional software development. Our contribution is a powerful and extensible framework for the rapid and easy development of prototypes and applications. We developed the EIToolkit to facilitate the processes necessary to create programs in the broad area of pervasive computing [Holleis and Schmidt 2008]. A detailed requirements analysis led to the description of the architecture and main implementation aspects of the toolkit. Some specific aspects uncommon in related approaches have been selected and were treated in more detail such as the support for simulating data input and simultaneous support for packet and stream-based data communication. The EIToolkit consists of a communication framework and set of tools that greatly reduce the effort of creating pervasive computing applications. It offers abstractions, simplifications, and common interfaces that ease the process of combining various kinds of technologies, devices, and applications such as custom-built sensors and actuators, as well as existing programs. It features simple access to all compatible components through several programming

languages. In order to further reduce the threshold of building applications with the toolkit, we presented a variety of tools on top of the EIToolkit. These enable persons without technical background to quickly assemble programs using, for example, graphical tools for generating program semantics and rules. We conjecture that this improves the dialog between, e.g., designers and programmers in a team since it facilitates expressing and conveying ideas, designs, as well as data flow within an envisioned system.

In order to evaluate the framework, we derived a comprehensive set of 46 requirements for such toolkits from an extensive literature review, various professional meetings, workshops, and discussions on the topic, as well as personal experiences. We split them into four main categories and several subcategories and list them together with a concise explanation and gave a list of references for each for easy additional review. This provides a novel amalgamation of knowledge that previously only has been available in a distributed or unpublished form.

For the purpose of placing the EIToolkit within the design space of applications supporting the development of pervasive applications, an extensive set of existing toolkits were gathered and presented. In total, 50 toolkits and frameworks were categorised into hardware-focused toolkits, software-focused toolkits, and those tightly combining hardware and software. They were analysed and compared with respect to the EIToolkit idea, design, and implementation. A subset of 24 projects that proved to have similar goals as the EIToolkit and for which it was possible to gather enough information were collected and tested against the criteria of the requirements list. This can help developers decide which toolkit is most appropriate to their needs according to the concrete set of requirements that they judge to be most important. A dynamic evaluation sheet is available online to directly retrieve the ratings of the evaluated toolkits according to weights placed on specific requirements.

To demonstrate the applicability and power of the toolkit, we presented several scenarios and applications that have been built using the EIToolkit, e.g. [Holleis, Kranz, Winter, et al. 2006], [Schmidt, Terrenghi, and Holleis 2007], and [Rukzio, Leichtenstern, et al. 2006]. These fall into two categories. First, we presented components supporting applications using specific devices or applications. This includes, for example, controllable power sockets, Skype, and Winamp on the output side, and devices such as the Wiimote control on the input side. For the second category, components were shown that leverage the use of a whole set of applications since they enable some kind of technology or protocol such as communication over serial line or Bluetooth.

As a further example for the use of the EIToolkit, we described the hardware and software design, creation, and implementation of small wireless displays and illustrated their use in a project about environment-based messaging. We showed that there is still substantial effort needed to build such custom hardware but that the EIToolkit considerably reduces the amount of necessary knowledge and skills to build applications on top of that. The actual study also revealed interesting results on the subject at hand. Sending messages to places instead of people is promising. However, it also showed that the design of gesture-based interfaces requires much consideration and well-planned user studies in order to find out what gestures people can easily do and how to increase the affordance of such interfaces. We also found that providing several ways of entering data into a system (e.g. mobile application, web browser, text messaging) is necessary to accommodate to users' needs.

In another project concerned with mobile interactions, we extracted insight about users' attitudes with respect to different ways of selecting remote devices. We compared the interaction types scanning (showing a list of Bluetooth enabled devices), touching (using NFC technology), and pointing. The pointing interaction was quickly implemented using the EIToolkit in conjunction with a prototyping board equipped with simple light sensors, [Rukzio, Leichtenstern, et al. 2006].

We further demonstrated the application of the EIToolkit and our expertise in two projects using capacitive sensing within the field of wearable computing. We showed the design and development of several general purpose wireless hardware boards used for sensing proximity and touch by human fingers focusing on minimisation and flexibility of the involved electronics.

The first project concentrated on touch input on clothing and wearable accessories in general [Holleis, Paasovaara, et al. 2008]. Besides providing detailed insight into related work within this subject, categorised into technology, specific applications, development support, and user studies, we introduced four different prototypes. They were based on the same technology but optimised for specific scenarios. Controls on phone

bags or directly into clothing focused on replicating part of a phone's interface while controls placed on gloves or a bicycle helmet were specialised around concrete activities. Two larger studies examined the acceptability and potential use of such technologies and applications. They resulted in ten guidelines, sorted in order of importance to potential users. These guidelines will help designers of similar applications to avoid general issues and to refine their applications. In addition, we were the first to study possible locations where on the body people expect and would accept touch controls and we were able to give a map of preferred locations. We also prepared the means in terms of design, software, and hardware for further studies aimed at clarifying opinions of users with regard to the concrete layout and arrangement of controls placed on clothing.

Another set of results emerged from studies performed with a mobile phone with a key pad augmented with touch sensors [Holleis, Huhtala, and Häkkilä 2008]. Besides providing implementations of software components that facilitate creating applications combining sensors with mobile phones and existing development environments such as Flash Lite, we showed a range of existing applications and provided an additional set of novel ideas such as overlaying images or other information on a contact entry. A study revealed the usefulness of the created programs and several aspects and guidelines that have to be taken into account for such applications.

In addition to various novel devices, projects, protocols, and applications used in conjunction with the EIToolkit, we listed several currently widely used hardware prototyping platforms and demonstrated ways to make them available to users of the EIToolkit. Similar to our approach of supporting various programming languages, this enlarges the number of potential users and eases the adoption of the toolkit.

Since simplifying the development of applications is a central goal of the EIToolkit, we added a platform to visualise and simulate sensor events and other data that need to be communicated. DATAVIS is an extensible system that encapsulates components for visual input and output of numerical and textual data. It provides a simple way of visualising data flow in an application providing a variety of components such as bar graphs for output and sliders or dropdown boxes for input. This tremendously eases initial experiments with new sensors or actuators, debugging running applications, as well as simulating data (a feature provided by only very few other approaches). It is available for all applications using the EIToolkit communication structure.

One way of keeping the initial threshold of creating applications low is to not have people learn new programming paradigms, languages or environments. Therefore, we also demonstrated how the toolkit integrates into existing frameworks and integrated development environments such as the Eclipse IDE. Besides enabling easy entry into application development, an important aspect of frameworks such as the EIToolkit is that they provide a high ceiling, i.e. allow complex applications to be built. This trade-off between low threshold and high ceiling is often problematic with such approaches. We managed to have a broad spectrum in this respect by providing simple to use tools to quickly generate straightforward applications, but at the same time always allowing the developer to work with the program's code in order to define more complex application logic.

In order to further develop user friendliness and to simplify development, we presented two sample tools built on top of the EIToolkit. They provide graphical environments that simplify creating programs using two different paradigms. One builds on the idea that a main goal of developing an application often is to bring a system into a certain state, possibly depending on some event or other state. The system guides the developer along these lines. The second environment is similar in that it lets the developer graphically specify application logic. The underlying principle, however, is rather based on events and consequences. We provided a highly configurable and scriptable cable-patching interface for quick but powerful rapid prototyping of simple applications.

Prototyping Applications with User Modelling Support

We demonstrated several ways of how to combine the first two aspects, namely predictive user models and prototyping support. Drawing from concepts of user-centred design, we showed that there are good reasons for incorporating user studies early in the design process. However, such studies often require functional prototypes and can involve high costs in terms of money and, especially, time. We adopted the reasoning of the cognitive modelling community that employing user models can save considerable amount of effort compared to user studies and generate similarly valid results.

Existing work that combines prototyping tools with predictive models is still scarce. We highlighted recent results and provided our own powerful solutions that integrate user models into application development. One approach allows incorporating our ideas into existing development environments. A configurable and extensible component of the EIToolkit encapsulates all available KLM data [Holleis, Kern, and Schmidt 2007]. This makes it available to a large variety of applications. Simply sending descriptions of actions generates a model that can then be used to retrieve task completion time of tasks using these actions. The module has been used in several projects. In accordance with all EIToolkit components, it can also easily be exchanged with other model generating stubs (an example would be a potential connection to cognitive architectures such as ACT-R).

As two example showcases for the integration of KLM semantics into existing IDEs without built-in modelling support, we picked d.tools and the Eclipse IDE. We identified integration points and provided implementations demonstrating how these systems can make use of the power of user modelling. Within this context, we also showed how the features of such IDEs can be leveraged to speed up and ease the prototyping of pervasive applications by automating discovery processes and interface generation with the help of the EIToolkit.

Approaches like our prototypical graphical environments introduced above base on the concept of state graphs that describe application behaviour. We provide an extensive discussion on advantages of this concept and extended ideas from related work with novel properties that can help usability research based on simple graph properties. As an example, it is possible to automatically find the cheapest way to achieve a goal using varying interpretations of the notion of ‘cost’. We also showed that the same approach can be used to make predictions about a user’s potential goal after having performed some sequence of initial actions.

Building on such state graphs, we created a standalone development environment called MAKEIT that incorporates many of the features and capabilities discussed above [Holleis and Schmidt 2008]. It is specialised on the development of applications running on mobile phones. The use of a state transition system means that many of the advantages of knowing the underlying state graph of an application that we introduced before can directly be exploited. This has the huge advantage that it is possible to integrate some structural properties into the generation process of the application. Thus, it is, for example, ensured by construction that all screens that are designed can actually be reached. This, as we argued formally, offers the opportunity to automatically check and retrieve properties of the application such as how a specific goal can be achieved. The graph-based semantics of the application is stored in a structure with an attribute hierarchy based on an extensible graph data solution and can be serialised to and loaded from an international standard file format, GraphML.

The MAKEIT system is a prototyping environment targeted at keeping the initial threshold of developing applications very low. To achieve that, we employed the concept of programming by demonstration. This means that developers can use a simulated on-screen mobile device and simply demonstrate action sequences on it. The project targeted the gap between quick but low-fidelity prototyping methods, such as paper prototyping, and high-fidelity implementations. The visual appearance of states on the device, i.e. the content of the screen on the device, can be quickly sketched. We introduced several methods to specify these contents directly within the application. These can consist of text, images, or can be quickly drawn within the application. This offers similar advantages as paper prototyping but alleviates disadvantages such as having to keep track of the potentially complex chain of screens and to manually replace the screens while executing the application. It also ensures that people can concentrate on the idea, look, and behaviour of programs.

With MAKEIT, tasks can easily be demonstrated and a corresponding application that perfectly serves the purpose of a vertical prototype is generated automatically. In conjunction with an easy integration of graphics, this ensures that people without programming knowledge can also engage in creating and demonstrating mobile applications. An application can be run within the environment in parallel to the development process or it can be compiled into a Java ME MIDlet and run on a mobile phone. A further way to facilitate and speed up the development is achieved by providing recurring patterns as templates. A scrollable list, for example, can thus be quickly added from a template library and only the items on it need to be adapted. We also provided several suggestions to keep the state graph clear and small by, e.g., using hierarchical grouping and merging of states.

The proposed process enables designers and developers to quickly create demonstrators. These can then be used to convey initial ideas, to test the feasibility of concepts, and as prototypes for additional user studies.

A further aspect that renders the environment novel is the tight integration of a modelling tool. The KLM component described above is used in conjunction with the state graph to automatically create a Keystroke-Level Model of the applications under construction. By simply indicating source and target state, all possible ways to achieve the target goal are calculated and time predictions are given. Thus, one can directly see whether and how the given task can be completed in acceptable time. The integration of KLM semantics directly within the development system provides the chance to exploit the power of task completion time predictions already in the design phase. Since the demonstration of actions that describe the application logics also involved automatically adding KLM operators, time estimates of the various actions and methods can be displayed continuously and on the fly. Besides choosing a best method (in terms of task completion time), this can help developers find and figure out bottlenecks and time consuming interactions for which remedies can be found and easily integrated in such early stages.

The described system incorporates standard types of interaction with mobile devices such as using the keypad and hotkeys. Additionally, advanced mobile interactions are available such as taking pictures or touching RFID / NFC tags. This includes both, the implementation of applications using such interactions as well as the generation of the prediction models. We described that it also supports the connection to the EIToolkit which means that a large set of sensors, actuators, and applications can be used from the created application. Furthermore, the environment was designed to be very extensible since it is likely that new types of interactions or new sensors and actuators will be introduced. We illustrated the few and simple steps necessary to integrate new actions in both, the application development interface and the model generator. MAKEIT is the first approach combining the concept of developing advanced mobile interactions by demonstration with the on the fly generation of predictive models and an interface to take advantage of these features. We provided several examples of applications that have been prototyped with the system.

In conclusion, we introduced an approach to support prototyping and developing pervasive applications, extended a well-known predictive model to novel types of interaction important for pervasive computing, and demonstrated ways of combining those to enable the use of such user models during initial stages in development. We thus provided important insight and progress into research about how to simplify the development of usable pervasive computing applications and prepared the stage for further extensions of both, the prototyping and the modelling part. Other researchers are enabled to exploit and reuse many of the approaches presented here, and additional aspects such as novel types of interactions or subjective measurements will be easy to identify and to add to the framework in the future.

Main Publications

Selected applications and projects have been published in the following full papers / articles:

The cube as tangible object in the domain of learning	[Terrenghi, Kranz, Holleis, et al. 2005] in the *Journal of Personal and Ubiquitous Computing*
Mobile selection techniques	[Rukzio, Leichtenstern, et al. 2006] at *Ubicomp'06*
Embedding sensors in everyday objects	[Holleis, Kranz, Winter, et al. 2006] in the *Journal of Virtual Reality and Broadcasting*
Guidelines for domestic pervasive computing applications	[Schmidt, Terrenghi, and Holleis 2007] in the *Journal of Pervasive and Mobile Computing*
Touch input on mobile phones	[Holleis, Huhtala, and Häkkilä 2008] at *TEI'08*
Touch interfaces for wearable computing	[Holleis, Paasovaara, et al. 2008] at *MobileHCI'08*

Key concepts described in this thesis have been published in the following full papers:

KLM for advanced mobile phone interactions	[Holleis, Otto, et al. 2007] at *CHI'07*
MAKEIT development environment	[Holleis and Schmidt 2008] at *Pervasive'08*

8.2 Outlook and Future Work

Obviously, several of the projects introduced in this work are prototypes that can profit from enhancements and further iterations in order to optimise the design, functionality, or applicability for user studies. For example, the EIToolkit support of the Particle microcontroller platform could benefit from, at the one end, generalisations such that further platforms can be easily exchanged for it, and, at the other end, specialisations in the form of optimisations and a library of examples and applications. A similar argumentation holds for the displays in the project about environment based messaging. Even though the EIToolkit ensures that, e.g. through message rerouting and interface coherence, parts of an application can be swapped with other components, additional support for semantic exchangeability could be added. Another example is the provided library of data visualisation components in the EIToolkit framework. The set of modules is open to extensions to more dynamic ones like graphs or charts that can also display properties like minimum and maximum values, running averages, patterns, etc. It is also planned to incorporate third-party developments and existing data visualisation toolkits.

As a representative of several projects described in this thesis, we illustrate ongoing and possible future work in the area of input in wearable computing.

Interactions with Wearables

One focus of our work has been to create, use, and refine tools to quickly generate and evaluate objects and applications in the area of wearable computing, e.g. using touch controls. We built several working prototypes and evaluated them in several user studies. A comprehensive set of guidelines have been extracted from the results. Clearly, these prototypes can be improved and iterated such that they fulfil the whole range of requirements and guidelines that we identified in the process.

Also, we are currently in the process of extending some of the studies to more participants and more detailed settings and questions. For example, we try to further classify and identify user expectations and user acceptance for specific places on the body where potential user interfaces could be placed. By providing concrete scenarios and various setups, we seek to extract even more detailed results and guidelines for future products.

Two additional areas which we deliberately left out in our previous studies are personal layout preferences and end-user configurability. We plan to work with professional designers to create different designs of basic sets of controls. For that, we already started to build a set of button-like wires that can be overlaid on designs and connected to the sensor platform within a few minutes and therefore tested on the fly (see Figure 87).

Figure 87: Prototyping material for quickly testing control interface layouts: several strips of conductive wires with ornamental buttons can quickly be connected to the sensing system.

Moreover, as shown in Figure 88, we reserved an area on one part of our prototyping apron that can be used to learn about the different ideas of people about their personal optimized layout of specific controls. As expressed in one of the main findings in the project, people often have their own ideas about how an interface should look like. This even applies to a very simple five button layout. As example, in one of the studies, only about 60 % of the users felt that the volume *up* button in a placement on the thigh should actually point *upwards*.

Further research is proposed to be done in the area of combinations of various technologies and interfaces. Especially in view of a desired simple use by end-users, interesting aspects appear when allowing the users to transfer or combine interfaces on, e.g., a phone with input methods embedded in clothing and output technology such as augmented reality integrated in one’s glasses.

Figure 88: Several easily exchangeable layouts for the media player control scenario.

User Models

The various interactions available in wearable computing constitute one example that show how highly dynamic research in predictive user models is. Whenever a model such as KLM is to be applied to a new area of interaction, all operators and possible actions have to be examined. Sometimes general parameters can often be reused (e.g. gestures or Fitts' Law to estimate arm movement times for interactions with touch controls on the body). Still, it has to be carefully weighed whether an existing operator can used or a novel one has to be introduced – which requires a lot of work for studies.

However, not only novel types of interaction such as the advent of small touch screens and gesture interactions, like those featured by the iPhone result in a required update of the KLM. It is an ongoing task to update operator times. For example due to technological advances, operator times might have to be adjusted from time to time. One example that applies to the models introduced in this thesis is for example the response time of NFC interaction. New phones such as the Nokia 6212 have become considerably faster than the 3220 model. Thus, the system response time for this particular interaction needs an adjustment from 2.2 seconds to roughly 1.0 second.

This shows that there are reasons to parameterise KLM operators with respect to involved technology. Additionally, there are many areas in which continuous adaptation is necessary. We already identified several research areas that pose additional restrictions and call for a revaluation of existing time values. In tangible computing and in-car interfaces, we are in the process of providing insight into how to model these types of interactions with respect to task completion time predictions.

A possible attempt to generalise the applicability of models such as the KLM is to create a connection to cognitive architectures such as ACT-R. This approach is followed by the CogTool project that generates KLM-like structures in a language called ACT-Simple [John and Salvucci 2005]. This has the advantage that adjustments in the underlying model execution engine can be done without changing the model itself. However, the transformation into the architecture represents a breach for the user because it is difficult to understand what is going on behind the scenes. This points to additional work to add visualisations of the applied models. This has partly been achieved in the MAKEIT environment and related work such as ExperiScope [Guimbretière, Dixon, and Hinckley 2007] but still needs more attention in order to achieve a broader acceptance and use.

EIToolkit for Rapid Prototyping

Future work for the EIToolkit generally includes supporting the requirements identified for prototyping toolkits that it and existing add-ons do not yet satisfy. Especially interesting will be the integration of design tools such as Adobe Flash into application development. One way to achieve that would be to use graphical tools such as the MAKEIT project and embed Flash capability into its process, e.g. when specifying the contents of the current screen or the transitions between them. Another option would be to, reversely, integrate EIToolkit functionality into the Flash environment; an option we explored by augmenting the Eclipse environment.

Moreover, support for deployment of applications on mobile devices without requiring infrastructure is necessary. The EIToolkit is not tightly integrated for direct use with mobile applications as has been described in Chapter 7. The connection is currently only available in emulations such as MAKEIT. A related issue is that, even if two devices only needed to communicate directly with each other, the current approach would still require a central component such as a PC through which the traffic is routed. This could be alleviated by running an EIToolkit component directly on the mobile devices that can transparently relay the messages through appropriate channels such as a WLAN or GPRS connection.

As has been mentioned, the interface and capabilities description facilities in the current version of the EIToolkit are still rudimentary. In addition to exploring more powerful concepts such as CORBA for transmitting interface descriptions, possible enhancements will also include, for example, semantic information (e.g. update rates, constraints, etc.) and physical properties (e.g. size, sensor orientations, possible movements, etc.).

In order to further lower the threshold of creating programs, an additional library of components to be used as basis for the development of applications should be made available. Such items could, e.g., help to translate messages from one type to another. They could also include mechanisms for input processing such as filtering, combining, or changing sensor values. Many algorithms known from data mining or artificial intelligence could be provided as components, e.g. neural networks for pattern recognition. It would also be possible to integrate those in the described graphical development processes.

Regarding the combination of EIToolkit and user modelling, we provided a simple KLM component that keeps a list of KLM operators and can be controlled externally to query and define a model for task sequences. In order to generalise this concept, the component would need to be extended with a global database of model parameters and should be further customisable. This would allow the continuous improvements and extensions to such models mentioned before to be integrated easily and made broadly available.

Prototyping Applications with User Modelling Support

Potential improvements of the introduced prototypes for low-threshold application development include mostly craftsman's work about integrating coding capabilities, polishing user interfaces, adding wizards and help, and visualising component properties. However, as has been mentioned, there are also more general questions that will have to be treated such as problems concerned with scaling. As soon as there are hundreds or more components available in the system, discovery, presentation, and selection of appropriate devices and components becomes an issue. Concepts from data visualisation, search, and weighting mechanisms need to be devised and employed.

From a research perspective also very interesting is how to add various specialised implementations for different target users. Similar to the Scratch language targeted at young children [Monroy-Hernández and Resnick 2008], it is conceivable to have various notions and views on the same project from a designer's, a modeller's, and a programmer's point of view.

In these aspects, the MAKEIT framework has the advantage to have been designed in a very extensible way. With regard to its capabilities, we currently plan to support or extend the support for several types of interactions in addition to the mentioned set of features:

- **Bluetooth**: currently, events that can be specified for input via a Bluetooth connection are distinguished by string matching. This implies that the prototyping framework currently does not allow defining variables within an event that could be further processed. This concept is powerful enough for many applications. Gesture recognition, for example, is often done directly on the controller that reads the sensor values. Data to be sent then consists only of the name or an ID of the gesture that has been recognised. However, in order to support more powerful applications, some additions for data processing have to be integrated.
- **Touch screens**: the concepts implemented in MAKEIT fit well for interactions with touch screens. Stylus or finger-based input can easily be demonstrated and simulated with the mouse. The modelling system can then use KLM operators for these interaction methods, e.g. taken from [Luo and John 2005].

- **EIToolkit**: mobile phones have not yet been directly connected to the EIToolkit which means that, in the current prototype, this connection will not be available on the target platform. The extension of the EIToolkit to more directly support developing programs for mobile devices is therefore part of ongoing research. Currently, the EIToolkit communicates with MAKEIT through a Bluetooth connection. Although this already enables, for instance, the use of a phone as remote control or the integration of external sensors and actuators to mobile phone applications, a more direct integration would be favourable. We are currently looking into collaboration with the creators of the iStuff Mobile toolkit [Ballagas, Memon, et al. 2007] in order to make use of their more direct connection of a mobile phone with computer-based infrastructure and combine it with our concept of modelling and programming by demonstration.
- **RFID / NFC**: although acting on the proximity of such tags (using a supported phone model) can be done using the PMIF framework described in [Rukzio, Wetzstein, and Schmidt 2005], the mechanism is currently limited to reading a simple ID. Such tags can store data in the order of thousands of bytes which could potentially be used for application logic. We argue that such processes are better implemented in code than using a graphical environment. However, a better support for rich-content tags would be advantageous.
- **Devices / skins**: future iterations of the prototype should take into account differences in hardware and software access, e.g. concerning the use of specific libraries or the amount and placement of buttons. For the latter issue, a potential approach is to detect the shape of the phone and the placement of its buttons from a picture as suggested by [St. Amant, Horton, and Ritter 2004]. In general, support for the use of specific buttons can be an issue. For example, not all available buttons can be used since the Java ME API does not give access to all of them (for example the power button). Currently, the implementation supports a specific phone model's layout and S60 phones for the code base only. However, by adjusting the code generation module, it is possible to extend the compatibility to systems such as Apple's iPhone or Google's Android platform without the need to introduce changes in the user interface.
- **User modelling**: the KLM part is mostly based on user input. Currently, the placement of the mental act operator *M* does not employ the full set of heuristics available. Additional intelligence could also be used in a way such that the system tracks operator placement and learns specific sequences after some time. Additionally, if user performance data was available for a specific action sequence, automatic improvements to the model could be suggested.

Most of these identified shortcomings can be implemented with moderate research effort. There are, however, two aspects that are inherent in the approach and goal of the system and require further investigation.

First, highly dynamic applications are difficult to create. One approach would be to introduce animation style sequences and use a tool similar to the Adobe Flash development environment to generate more advanced transitions. However, most dynamic transitions depend on data from previous transitions. This could be tackled by storing such information in the graph. However, issues with data consistency, data types, addressing and computing of information would render the interface significantly more complex for the user. The advantages of increasing the power of creating applications have to be compared with the increase in complexity and gradient of the learning curve. As we argued before, we argue for shifting such effort into the programming domain where such transitions can be implemented more easily.

The second problem with this approach is again one of scalability. The number of states in a state graph of even a moderately sized application can be high and render it difficult to keep an overview on the application's semantics, especially if the state graph is generated automatically. If there are many transactions, it can be difficult to find a pleasing layout. We provided some means to reduce this number of states using merging and condensing of states. Other than that, the only way to reduce the complexity is to split the application in several parts, e.g. one for each entry in a menu, and combine them afterwards. As long as the main application area of the development environment is the rapid prototyping of smaller applications, these issues should have minor impact. Nevertheless, advances in graph layout algorithms and additional ways of condensing the visualisation would help to increase the ease of describing more complex algorithms. One way of research might go in the direction of three dimensional representations.

In addition, further research on solutions for supporting state-based application development is possible. This includes the expressive power and automatic generation of state descriptions as well as formalisations and algorithms for bringing an application into a specific state. Whenever (state) graphs are used underlying the development of applications, they can be employed for usability purposes. However, besides the mentioned approaches, this possibility is still not fully exploited and warrants additional research.

Another area of possible improvement is to further evaluate and improve the concept and user interface with a larger user study and concentrate on simplifying the inclusion of standard controls and widgets in the phone's screen like text input fields and scroll lists. Approaches like those seen in the upcoming Adobe Flash Catalyst project[94] (previously marketed as Adobe Thermo) aim at this direction by automatically converting drawings of, e.g., a text area to a functional text box. A second option is used by [St. Amant, Horton, and Ritter 2004] who apply models (using GOMS and ACT-R) to menu interaction on mobile phones and present a system to semi-automatically extract buttons from an image using shape detection algorithms.

Under active development is the extension of the concepts for mobile device development implemented in MAKEIT to all types of pervasive applications. It is definitely possible to use a similar interface for other areas within pervasive computing. Reviewing the categories introduced in Chapter 2, it is clear that it would be difficult to include inherently different metaphors such as virtual reality applications. However, concepts like ambient computing, context aware applications, wearable and tangible computing are potential candidates. In [Holleis, Kern, and Schmidt 2007], we for example indicated opportunities for that last category. In wearable computing, instead of the mobile phone, a certain piece of clothing could be displayed. The graphical interface would then need to be adapted in a way that specific types of sensors and actuators could be attached and altered. The same principle would apply for smart environments. The environment would have a representation on screen and available sensors could be used as sources for events creating state graph transitions, while actuators could be manipulated according to the respective states. [Hull, Clayton, and Melamed 2004] show approaches in that direction for location-based applications. The separation and possible coexistence of multiple sensors and actuators has already been integrated into the design of the system but has not yet been fully exploited.

Our systems laid the foundations for manifold potential combinations with research approaches that fit to specific parts of our system. As an example, pattern matching or data mining methods can be applied to further expand the power of programming by demonstration in conjunction with analogue sensors and more complex context information. Besides the suggested solutions, our work provides the ground for others to contribute to the further generalisation of the approaches which is only fully possible by broad approval, use, and continuous further development.

[94] Adobe Flash Catalyst, interaction design tool; project page: *http://labs.adobe.com/technologies/flashcatalyst/*

Appendix

Graph Theoretical Terms

We base our formalisations that use terms and concepts from graph theory mostly on standard and recognised definitions. In order to avoid misinterpretations, we give definitions of the terms we use in the following.

Definition 6 (Undirected Graph $\hat{G}$, Node, Edge)
An **undirected graph** $\hat{G}$ = (S, A) is a finite non-empty set S of objects called **nodes** (also called vertices) together with a (possibly empty) set A of two-element sets {s1, s2} of distinct nodes s1, s2 ∈ S called (undirected) **edges**.

In the literature, the most prominent notation is $G = (N, E)$ with N being the set of nodes and E the set of edges. However, since the specific graph used in this work uses states (S) and actions (A) instead, we directly use these letters for consistency.

Definition 7 (Directed Graph $\vec{G}$, Digraph, Arc)
A **directed graph** $\vec{G} = (S, A)$, also called a digraph, is a finite non-empty set S of objects called nodes (also called vertices) together with a (possibly empty) set A of ordered pairs (s_1, s_2) of distinct nodes $s_1, s_2 \in S$ called (directed) edges (or **arcs**).

Definition 8 (Graph, Graph Element)
A **graph** is either a directed or an undirected graph. The set of edges E of a graph can also contain directed and undirected edges at the same time.
The union of the set of nodes and the set of edges is called the set of **graph elements**.

Note that these definitions do not allow multiple edges since, for a graph $G = (S, A)$, A is a set and does not allow duplicates.

Definition 9 (Simple Graph, Self-loop)
An edge is called a **self-loop** (also reflexive or trivial) if it is of the form (s, s). A graph $G = (S, A)$ without self-loops is referred to as **simple**.

Note that since the edges of undirected graphs are two-element sets, there exist no self-loops in undirected graphs.

Definition 10 (Out-Edge, In-Edge, Source Node, Target Node)
In a graph $G = (S, A)$, an edge $a = (s_1, s_2) \in A$ is an **out-edge** (also called outgoing edge) of node $s_1 \in S$. It is an **in-edge** (also called incoming edge) of node $s_2 \in S$.
For an edge $a = (s_1, s_2) \in A$, s_1 is the **source node** and s_2 the **target node** of a.

For undirected graphs, these terms are undefined.

Definition 11 (Incident Node / Edge)
In a digraph $\vec{G} = (S, A)$, $s \in S$ is an **incident node** to an edge $a \in A$ if a is an out-edge or an in-edge of n.
In an undirected graph $\hat{G} = (S, A)$, $s \in S$ is an **incident node** to an edge $a \in A$ if a node $s' \in S$ exists such that $a = \{s, s'\}$.
In a graph $G = (S, A)$, $s \in S$ is an **incident node** to an edge $a \in A$ if one of the previous two conditions holds for n.
In a graph $G = (S, A)$, $a \in A$ is an **incident edge** to a node $s \in S$ if s is incident to edge a.

In directed graphs, another definition sometimes used is that an edge **leaves** its source and is **incident** to its target node.

Definition 12 (Out-Neighbour, In-Neighbour, Neighbour)
In a digraph $\vec{G} = (S, A)$, a node $s \in S$ is an **out-neighbour** of a node $s' \in S$, if there exists an edge $a = (s', s) \in A$.
A node $s \in S$ is an **in-neighbour** of a node $s' \in S$ if there exists an edge $a = (s, s') \in A$.
In a graph $G = (S, A)$, a node $s \in S$ is a **neighbour** of a node $s' \in S$, if there exists an edge $a \in A$ for which holds $a = (s, s')$ or $a = (s', s)$ or $a = \{s, s'\}$.

Definition 13 (In-Degree, Out-Degree)
In a digraph $\vec{G} = (S, A)$, the **in-degree** and **out-degree** of a node $s \in S$, $indeg(s)$ and $outdeg(s)$, are the number of in-edges and out-edges of s, respectively.
In a graph $G = (S, A)$, the **in-degree** and **out-degree** of a node $s \in S$ are the number of in-edges and out-edges of s, respectively, plus the number of undirected edges incident to s.

For undirected graphs, the terms in-degree and out-degree are undefined; instead the term degree is used:

Definition 14 (Degree, Source, Sink)
In a graph $G = (S, A)$, the **degree** of a node $s \in S$ is the number of edges incident to s.
In a digraph $\vec{G} = (S, A)$, a **source** is a node with an in-degree of zero. A node with an out-degree of zero is called a **sink**.

Thus, for a digraph, the degree is the sum of in-and out-degree. This does not hold for an undirected graph, since the undirected edges would be counted twice.
Obviously source and sink nodes are not defined for undirected graphs.

Definition 15 (Path, Path Length, (Strongly) Connected, Cycle, Acyclic Graph)
In a graph $G = (S, A)$, a **directed path** of **length** n is a sequence of n directed edges where an edge has the same target node as the source node of the subsequent edge. In other words, a path $p(s_a, s_b)$, of length n from a node $s_a \in S$ to a node $s_b \in S$ is defined as a sequence of n edges that connects s_a with s_b, i.e. a path $p(s_a, s_b)$ exists, if and only if

- there is an edge $a = (s_a, s_b) \in A$, i.e. path of length $n = 1$, or
- there exists an $n \in \{2, 3, \ldots\}$, edges $a_0, a_1, \ldots, a_{n-1} \in A$ and nodes $s_0, s_1, \ldots, s_{n-2} \in S$ with $a_0 = (s_a, s_0)$, $a_1 = (s_0, s_1)$, $\ldots$, $a_{n-1} = (s_{n-2}, s_b)$.

An **undirected path** is defined analogously but uses only undirected edges. A **path** in general may use directed and undirected edges. Directed edges must always be traversed from their source to their target node. Note that there is potentially more than one path between two nodes. We use $\{p(s_a, s_b)\}$ to refer to all distinct paths between $s_a \in S$ to a node $s_b \in S$.
Two nodes $s_1, s_2 \in S$ are **connected** if $s_1 = s_2$ or there exists a path $p(s_1, s_2)$ between them. They are **strongly connected** if the path uses only directed edges (from source to target).
A **cycle** is either a directed path where the source node of the first edge coincides with the target node of the last edge, i.e. a path $p(s, s)$ with $s \in S$, or an undirected path where each node appears exactly twice. A graph that does not contain any cycle is said to be **acyclic**.

Note that there can be several paths that connect two nodes. If the graph contains cycles, there can even be infinitely many of them. Paths play an important role in graph theory. For the state graphs introduced below, they describe, e.g., whether, how, and in how many steps users can reach their goal.

Definition 16 (Underlying Undirected Graph, Subgraph)
The graph G that is created by replacing all directed edges of a directed graph $\vec{G}$ with undirected edges is called the **underlying undirected graph** of graph $\vec{G}$.
A graph $G = (S, A)$, whose node and edge sets are subsets of another graph $G' = (S', A')$, i.e. $S \subset S'$ and $A \subset A'$, is called a **subgraph** of the other graph.

Definition 17 ((Strongly) Connected Graph, (Strongly) Connected Component)
A graph $G = (S, A)$, is **connected** if every pair of nodes in the graph is connected. A maximally connected subgraph of G is a **connected component**.
The description '**strongly**' applies if only directed edges are used.

This implies that each node and each edge belongs to exactly one connected component.

Definition 18 (Shortest Path)
A path $p(s_a, s_b)$ with length l in a graph $G = (S, A)$ with node $s_a, s_b \in S$, is a **shortest path**, if and only if for every path $p' \in \{p(s_a, s_b)\}$ with length $|p'| = l'$ it holds that $l \leq l'$.

Full Keystroke Level Models of the Evaluation Scenario

The following tables present the complete model of the interactions described in Section 3.4.4. Table 16 shows the version using NFC (with the Nokia 3220 NFC phone) while Table 17 shows the version using a standard browser on the same phone.

Table 16: KLM used to evaluate the KLM extensions for advanced mobile phone interactions, see Section 3.4.4. It uses the NFC reader of the Nokia 3220 phone and an NFC enabled poster. Continued on next page.

Description	Operators	Time
Pick up the mobile phone	***I***[optimal]	1.18 sec.
Enter main menu	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Go to 'Programs'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Programs'	***K***[Hotkey]	0.16 sec.
Go to 'Collection'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Collection'	***K***[Hotkey]	0.16 sec.
Select 'Choose program'	***K***[Hotkey]	0.16 sec.
Open application	***K***[Hotkey]	0.16 sec.
Wait for program to open	***R***	4.63 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Choose 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose 'Direct input'	***K***[Hotkey]	0.16 sec.
Go to 'Transportation'	***M***, 2***K***[Hotkey]	1.35 sec., 2*0.16 sec.
Select 'Transportation'	***K***[Hotkey]	0.16 sec.
Choose 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose 'Confirm'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Click 'Next'	***K***[Hotkey]	0.16 sec.
Confirm network access	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for service download	***R***[adv]	15.88 sec.
Go to duration: 1 day	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose duration: 1 day	***K***[Hotkey]	0.16 sec.
Go to start zone: zone2	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose start zone: zone2	***K***[Hotkey]	0.16 sec.
Go to end zone: zone4	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose end zone: zone4	***K***[Hotkey]	0.16 sec.
Go to passenger number:1 person	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Choose passenger number: 1 person	***K***[Hotkey]	0.16 sec.
Select 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.

Description	Operators	Time
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Select 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Card number'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Enter four numbers	***M,*** 4***K***[Keypad]	1.35 sec., 4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4x 0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Select 'OK'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Go to 'payment method': VISA	***M***, 3***K***[Hotkey]	1.35 sec., 3*0.16 sec.
Select 'payment method': VISA	***K***[Hotkey]	0.16 sec.
Select 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Click 'Main'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Click 'Exit'	***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait till program shuts down	***R***	4.63 sec.
Exit to main screen	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Time for the whole interaction		**122.77 sec. ~ 2:03 min.**

Table 17: KLM used to evaluate the KLM extensions for advanced mobile phone interactions, see Section 3.4.4. It uses the web browser on the Nokia 3220 phone.
Continued on next page.

Description	Operators	Time
Pick up the mobile phone	***I***	1.18 sec.
Enter main menu	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Go to 'Programs'	***M***, 3***K***[Hotkey]	1.35 sec., 3*0.16 sec.
Select 'Programs'	***K***[Hotkey]	0.16 sec.
Go to 'Collection'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Collection'	***K***[Hotkey]	0.16 sec.
Select 'Choose program'	***K***[Hotkey]	0.16 sec.
Open application	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for program to open	***R***	4.63 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Attention shift from mobile phone to poster	***M***, S_{Macro}	1.35 sec., 0.36 sec.
Movement to tag	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.
Attention shift from poster to mobile phone	***M***, S_{Macro}	1.35 sec., 0.36 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Download ticket service	***K***[Hotkey]	0.16 sec.
Confirm network access	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Download service	***R***	15.88 sec.
Attention shift from mobile phone to poster	***M***, ***A***[macro]	1.35 sec., 0.36 sec.
Movement to tag 'From'	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.
Find next tag	***M***	1.35 sec.
Movement to tag 'To'	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.
Find next tag	***M***	1.35 sec.
Movement to tag 'Duration'	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.
Find next tag	***M***	1.35 sec.
Movement to tag 'Number of passengers'	***P***	1.00 sec.
Action to accomplish NFC interaction	***A***[NFC]	0.00 sec.
Process tag	***R***[NFC]	2.58 sec.

Description	Operators	Time
Movement from poster to the body	***P***	1.00 sec.
Attention shift from poster to mobile phone	***M***, S_{Macro}	1.35 sec., 0.36 sec.
Scroll down and validate the four inputs	18***K***[Hotkey], 4***M***	18*0.16 sec., 4*1.35 sec.
Choose 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send '	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Choose 'single ticket'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'single ticket'	***K***[Hotkey]	0.16 sec.
Choose 'Options'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Click to enter credit card number	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Enter four numbers	***M,*** 4***K***[Keypad]	1.35 sec., 4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Enter four numbers	4***K***[Keypad]	4*0.36 sec.
Attention shift from keypad to display and back	S_{Micro}	0.14 sec.
Press 'OK'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Scroll down	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'VISA'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select Options	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Select 'Send'	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait for server connection	***R***	10.94 sec.
Read instructions	***M***	1.35 sec.
Scroll down to read further	***K***[Hotkey]	0.16 sec.
Read instructions	***M***	1.35 sec.
Exit program	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Wait till program shuts down	***R***	4.63 sec.
Exit to main screen	***M***, ***K***[Hotkey]	1.35 sec., 0.16 sec.
Time for the whole Interaction		**147.48 sec. ~ 2:27 min.**

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. *Data Structures and Algorithms.* Addison-Wesley, 1983.

Ailisto, H., Korhonen, I., Plomp, J., Pohjanheimo, L., and Strömmer, E. "Realising Physical Selection for Mobile Devices." *PI'03.* 2003. 38-41.

Anderson, J. R. *Rules of the Mind.* Lawrence Erlbaum Associates, 1993.

Andersson, J. "A Deployment System for Pervasive Computing." *ICSM'00.* 2000. 262-270.

Atkinson, R. C. and Shiffrin, R. M. "Human Memory: a Proposed System and its Control Processes." In *The Psychology of Learning and Motivation: Advances in Research and Theory*. 1968.

Bachmaier, C., Brandenburg, F. J., Forster, M., Holleis, P., and Raitner, M. "Gravisto: Graph Visualization Toolkit." *GD'04.* 2004.

Ballagas, R. "Bringing Iterative Design To Ubiquitous Computing: Interaction Techniques, Toolkits, and Evaluation Methods." PhD thesis, RWTH Aachen University, 2007.

Ballagas, R., Borchers, J., Rohs, M., and Sheridan, J. G. "The Smart Phone: a Ubiquitous Input Device." *IEEE Pervasive Computing* 5, no. 1. 2006. 70-77.

Ballagas, R., Kratz, S. G., Borchers, J., Yu, E., Walz, S. P., Fuhr, C. O., et al. "REXplorer: a Mobile, Pervasive Spell-casting Game for Tourists." *CHI'07.* 2007. 1929-1934.

Ballagas, R., Memon, F., Reiners, R., and Borchers, J. "iStuff Mobile: Rapidly Prototyping New Mobile Phone Interfaces for Ubiquitous Computing." *CHI'07.* 2007. 1107-1116.

Ballagas, R., Ringel, M., Stone, M., and Borchers, J. "iStuff: a Physical User Interface Toolkit for Ubiquitous Computing Environments." *CHI'03.* 2003. 537-544.

Ballagas, R., Rohs, M., and Sheridan, J. G. "Sweep and Point and Shoot: Phonecam-based Interactions for Large Public Displays." *CHI'05 Extended Abstracts.* 2005. 1200-1203.

Ballagas, R., Szybalski, A., and Fox, A. "Patch Panel: Enabling Control-flow Interoperability in Ubicomp Environments." *PerCom'04.* 2004. 241-252.

Ballesteros, F. J., Soriano, E., Guardiola, G., and Leal, K. "Plan B: Using Files Instead of Middleware." *IEEE Pervasive Computing* 6, no. 3. 2007. 58-65.

Bälter, O. "Keystroke Level Analysis of Email Message Organization." *CHI'00.* 2000. 105-112.

Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J., and Zukowski, D. "Challenges: an Application Model for Pervasive Computing." *MobiCom'00.* 2000. 266-274.

Barbeau, M. "Mobile, Distributed, and Pervasive Computing." In *Handbook of Wireless Networks and Mobile Computing*, 581-600. 2002.

Barber, P. *Applied Cognitive Psychology: An Information Processing Framework.* Methuen, 1988.

Barton, J. J. and Vijayraghavan, V. "UBIWISE: A Ubiquitous Wireless Infrastructure Simulation Environment." Technical Report HPL-2002-303, Mobile and Media Systems Laboratory, HP Laboratories Palo Alto, 2002.

Baskin, J. D. and John, B. E. "Comparison of GOMS Analysis Methods." *Conference Summary CHI'98.* 1998. 261-262.

Bass, L., John, B. E. and Kates, J. "Achieving Usability Through Software Architecture." Technical report CMU/SEI-2001-TR-005, Software Engineering Institute, Carnegie Mellon University, 2001.

Bauer, M., Bruegge, B., Klinker, G., MacWilliams, A., Reicher, T., Riss, S., et al. "Design of a Component-based Augmented Reality Framework." *ISAR'00.* 2001. 45-54.

Baumeister, L. K., John, B. E., and Byrne, M. D. "A Comparison of Tools for Building GOMS Models." *CHI'00.* 2000. 502-509.

Beard, D. V., Entrikin, S., Conroy, P., Wingert, N. C., Schou, C. D., Smith, D. K., et al. "Quick GOMS: a Visual Software Engineering Tool for Simple Rapid Time-motion Modeling." *interactions* 4, no. 3. 1997. 31-36.

Becker, C., Handte, M., Schiele, G., and Rothermel, K. "PCOM - a Component System for Pervasive Computing." *PerCom'04.* 2004. 67-76.

Beckmann, C. and Dey, A. K. "SiteView: Tangibly Programming Active Environments with Predictive Visualization." Technical Report IRB-TR-03-025, Intel Research, 2003.

Beigl, M., Krohn, A., Zimmer, T., and Decker, C. "Typical Sensors needed in Ubiquitous and Pervasive Computing." *INSS'04.* 2004. 153-158.

Benford, S., Magerkurth, C., and Ljungstrand, P. "Bridging the Physical and Digital in Pervasive Gaming." *Communications of the ACM* 48, no. 3. 2005. 54-57.

Bianchi, G., Knoerlein, B., Székely, G., and Harders, M. "High Precision Augmented Reality Haptics." *Eurohaptics'06.* 2006. 169-177.

Block, F., Villar, N., and Gellersen, H.-W. "A Malleable Physical Interface for Copying, Pasting, and Organizing Digital Clips." *TEI'08.* 2008. 117-120.

Borchers, J., Ringel, M., Tyler, J., and Fox, A. "Stanford Interactive Workspaces: a Framework for Physical and Graphical User Interface Prototyping." *IEEE Personal Communications* 9, no. 6. 2002. 64-69.

Boyd, J. "Here Comes the Wallet Phone." *IEEE Spectrum* 42, no. 11. 2005. 12-14.

Brewster, S. "Overcoming the Lack of Screens Space on Mobile Computers." *Personal and Ubiquitous Computing* 6, no. 3. 2002. 188-205.

Broll, G., Haarländer, M., Paolucci, M., Wagner, M., Rukzio, E., and Schmidt, A. "Collect&Drop: A Technique for Multi-Tag Interaction with Real World Objects and Information." *AMI'07.* 2007. 175-191.

Brown, J. "Evaluation of the Task-Action Grammar Method for Assessing Learnability in User Interface Software." *OZCHI'96.* 1996. 308-309.

Buechley, L. "A Construction Kit for Electronic Textiles." *ISWC'06.* 2006. 28-32.

Buechley, L., Eisenberg, M., Catchen, J., and Crockett, A. "The LilyPad Arduino: Using Computational Textiles to Investigate Engagement, Aesthetics, and Diversity in Computer Science Education." *CHI'08.* 2008. 423-432.

Buonadonna, P., Gay, D., Hellerstein, J. M., Hong, W., and Madden, S. "TASK: Sensor Network in a Box." *EWSN'05.* 2004. 133-144.

Buxton, W. "A Three-State Model of Graphical Input." *INTERACT'90.* 1990. 449-456.

Card, S. K., Moran, T. P., and Newell, A. "The Keystroke-Level Model for User Performance Time with Interactive Systems." *Communications of the ACM* 23, no. 7. 1980. 396-410.

Card, S. K., Newell, A., and Moran, T. P. *The Psychology of Human-Computer Interaction.* Lawrence Erbaum Associates, 1983.

Clarkson, E. C., Patel, S. N., Pierce, J. S., and Abowd, G. D. "Exploring Continous Pressure Input for Mobile Phones." Technical Report GIT-GVU-06-20, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 2006.

Clarkson, E., Lyons, K., Clawson, J., and Starner, T. "Revisiting and Validating a Model of Two-thumb Text Entry." *CHI'07.* 2007. 163-166.

Cockburn, A. and Siresena, A. "Evaluating Mobile Text Entry with the Fastap Keypad." *People and Computers* 2, no. 17. 2003. 77-80.

da Costa, C. A., Yamin, A. C., and Geyer, C. F. R. "Toward a General Software Infrastructure for Ubiquitous Computing." *IEEE Pervasive Computing* 7, no. 1. 2008. 64-73.

Das, S. K., Conti, M., and Shirazi, B. *Pervasive and Mobile Computing Journal* [Elsevier].

Davies, N. and Gellersen, H.-W. "Beyond Prototypes: Challenges in Deploying Ubiquitous Systems." *IEEE Pervasive Computing* 1, no. 1. 2002. 26-35.

Davies, N., Landay, J., Hudson, S., and Schmidt, A. "Rapid Prototyping in Ubiquitous Computing." *IEEE Pervasive Computing* 4, no. 4. 2005. 15-17.

Davies, N., Siewiorek, D. P, and Sukthankar, R. "Activity-based Computing." *IEEE Pervasive Computing* 7, no. 2. 2008. 20-21.

de Bono, E. *Six Thinking Hats.* 1999.

Decker, C., Krohn, A., Beigl, M., and Zimmer, T. "The Particle Computer System." *SPOTS Track, IPSN'05.* 2005. 443-448.

Dey, A. K. "Understanding and Using Context." *Personal and Ubiquitous Computing* 5, no. 1. 2001. 4-7.

Dey, A. K., Salber, D., and Abowd, G. D. "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications." *Human-Computer Interaction* 16. 2001.

Dietz, P. and Leigh, D. "DiamondTouch: A Multi-user Touch Technology." *UIST'01.* 2001.

Díez-Díaz, F., González Rodríguez, M., and Vidau, A. "An Accesible and Collaborative Tourist Guide Based on Augmented Reality and Mobile Devices." *HCI'07.* 2007. 353-362.

Döring, T. and Beckhaus, S. "The Card Box at Hand: Exploring the Potentials of a Paper-based Tangible Interface for Education and Research in Art History." *TEI'07.* 2007. 87-90.

Dunlop, M. D. and Crossan, A. "Dictionary Based Text Entry Method for Mobile Phones." *Workshop on Human Computer Interaction with Mobile Devices, Interact'99.* 1999. 5-7.

Dunlop, M. D. and Crossan, A. "Predictive Text Entry Methods for Mobile Phones." *Personal Technologies* 4, no. 2-3. 2000.

Edwards, W. K. and Grinter, R. E. "At Home with Ubiquitous Computing: Seven Challenges." *Ubicomp'01.* 2001. 256-272.

Edwards, W. K., Bellotti, V., Dey, A. K., and Newman, M. W. "Stuck in the Middle: The Challenges of User-centered Design and Evaluation for Infrastructure." *CHI'03.* 2003. 297-304.

Ferreira, A. and Antunes, P. "Quantitative Evaluation of Workspace Collaboration." *CSCWD'06.* 2006. 1-6.

Ferscha, A., Hechinger, M., Mayrhofer, R., and Oberhauser, R. "A Light-weight Component Model for Peer-to-Peer Applications." *ICDCSW'04.* 2004.

Fishkin, K. P. "A Taxonomy for and Analysis of Tangible Interfaces." *Personal and Ubiquitous Computing* 8, no. 5. 2004. 347-358.

Fitts, P. M. "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement." *Journal of Experimental Psychology* 47. 1954. 381-391.

Fleetwood, M. D., Byrne, M. D., Centgraf, P., Dudziak, K. Q., Lin, B., and Mogilev, D. "An Evaluation of Text Entry in Palm OS: Graffiti and the Virtual Keyboard." *Human Factors and Ergonomics Society, 46th Annual Meeting.* 2002.

Föckler, P., Zeidler, T., Brombach, B., Bruns, E., and Bimber, O. "PhoneGuide: Museum Guidance Supported by On-device Object Recognition on Mobile Phones." *MUM'05.* 2005.

Fowler, G. A. "QR Codes: In Japan, Billboards Take Code-crazy Ads to New Heights." *Wall Street Journal 10.10*, 2005.

Fox, A., Davies, N., de Lara, E., Spasojevic, M., and Griswold, W. "Real-World Deployments." *IEEE Pervasive Computing* 5, no. 3. 2006. 21-56.

Gellersen, H.-W. and Beigl, M. "Multi-sensor Context-awareness in Mobile Devices and Smart Artefacts." *Mobile Networks and Applications* 7, no. 5. 2002. 341-351.

Gellersen, H.-W., Kortuem, G., Beigl, M., and Schmidt, A. "Physical Prototyping with Smart-Its." *IEEE Pervasive Computing* 3, no. 3. 2004. 74-82.

Gemperle, F., DiSalvo, C., Forlizzi, J., and Yonkers, W. "The Hug: a New Form for Communication." *DUX'03.* 2003. 1-4.

Gemperle, F., Kasabach, C., Stivoric, J., Bauer, M., and Martin, R. "Design for Wearability." *ISWC'98.* 1998. 116.

Gong, R. J. and Elkerton, J. "Designing Minimal Documentation Using a GOMS Model: a Usability Evaluation of an Engineering Approach." *CHI'90.* 1990. 99-107.

Gong, R. J. and Kieras, D. E. "A Validation of the GOMS Model Methodology in the Development of a Specialized, Commercial Software Application." *CHI'94.* 1994. 351-357.

Gong, R. J. "Validating and Refining the GOMS Model Methodology for Software User Interface Design and Evaluation." PhD Thesis. GAX94-09693, University of Michigan, 1993.

Gow, J. and Thimbleby, H. "MAUI: An Interface Design Tool Based On Matrix Algebra." *CADUI'04.* 2004. 81-94.

Gray, W. D., John, B. E., and Atwood, M. E. "The Precis of Project Ernestine or an Overview of a Validation of GOMS." *CHI'92.* 1992. 307-312.

Green, P. "Motor Vehicle Driver Interfaces." In *Human-computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*, 844-860. 2003.

Green, T. R. G., Schiele, F., and Payne, S. J. "Formalisable Models of User Knowledge in Human-Computer Interaction." In *Working with Computers: Theory Versus Outcome*, 3-46. 1988.

Greenberg, S. and Fitchett, C. "Phidgets: Easy Development of Physical Interfaces Through Physical Widgets." *UIST'01.* 2001. 209-218.

Greenfield, A. *Everyware: The Dawning Age of Ubiquitous Computing.* New Riders Publishing, 2006.

Greenhalgh, C., Purbrick, J., and Snowdon, D. "Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring." *CVE'00.* 2000. 119-127.

Greenhalgh, H. J., Izadi, S., Mathrick, J., and Taylor., I. "ECT: a Toolkit to Support Rapid Construction of Ubicomp Environments." *UbiSys'04.* 2004.

Greenstein, B., Kohler, E., and Estrin, D. "A Sensor Network Application Construction Kit (SNACK)." *SenSys'04.* 2004. 69-80.

Grimm, R. "One.World: Experiences with a Pervasive Computing Infrastructure." *IEEE Pervasive Computing* 3, no. 3. 2004. 22-30.

Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., et al. "System Support for Pervasive Applications." *ACM Transactions on Computer Systems* 22, no. 4. 2004. 421-486.

Guiard, Y. "Asymmetric Division of Labor in Human Skilled Bimanual Action: the Kinematic Chain as a Model." *Journal of Motor Behavior* 19. 1987. 486-517.

Guimbretière, F., Dixon, M., and Hinckley, K. "ExperiScope: an Analysis Tool for Interaction Data." *CHI'07.* 2007. 1333-1342.

Halbert, D. C. "Programming by Example." PhD thesis, University of California at Berkeley, 1984.

Hannikainen, J., Mikkonen, J., and Vanhala, J. "Button Component Encasing for Wearable Technology Applications." *ISWC'05.* 2005. 204-205.

Harel, D. "Statecharts: a Visual Formalism for Complex Systems." *Science of Computer Programming* 8, no. 3. 1987. 231-274.

Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S. R. "Authoring Sensor-based Interactions by Demonstration with Direct Manipulation and Pattern Recognition." *CHI'07.* 2007. 145-154.

Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., et al. "Reflective Physical Prototyping through Integrated Design, Test, and Analysis." *UIST'06.* Montreux, Switzerland: ACM, 2006. 299-308.

Haunold, P. and Kuhn, W. "A Keystroke Level Analysis of a Graphics Application: Manual Map Digitizing." *CHI'94.* 1994. 337-343.

Henricksen, K., Indulska, J., and Rakotonirainy, A. "Infrastructure for Pervasive Computing: Challenges." *Workshop on Pervasive Computing and Information Logistics, Informatik'01.* 2001.

Hilliges, O., Baur, D., and Butz, A. "Photohelix: Browsing, Sorting and Sharing Digital Photo Collections." *IEEE Tabletop Workshop.* 2007.

Hilliges, O., Terrenghi, L., Boring, S., Kim, D., Richter, H., and Butz, A. "Designing for Collaborative Creative Problem Solving." *CC2007.* ACM, 2007. 137-146.

Hinckley, K., Guimbretière, F., Baudisch, P., Sarin, R., Agrawala, M., and Cutrell, E. "The Springboard: Multiple Modes in one Spring-loaded Control." *CHI'06.* 2006. 181-190.

Holleis, P. and Brandenburg, F. J. "QUOGGLES: Query On Graphs - a Graphical Largely Extensible System." *GD'04.* 2004. 465-470.

Holleis, P. and Schmidt, A. "MakeIt: Integrate User Interaction Times in the Design Process of Mobile Applications." *Pervasive'08.* 2008.

Holleis, P. and Schmidt, A. "New Learning Experiences with New Technologies." *Workshop Classroom of the Future, Alpine CSCL Rendez-vous 2007.* 2007.

Holleis, P. and Schmidt, A. "Towards Real-World Object Orientation." *Workshop UbiPhysics, UbiComp'05.* 2005.

Holleis, P., Huhtala, J., and Häkkilä, J. "Studying Applications for Touch-enabled Mobile Phone Keypads." *TEI'08.* 2008. 15-18.

Holleis, P., Kern, D., and Schmidt, A. "Integrating User Performance Time Models in the Design of Tangible UIs." *CHI'07 Extended Abstracts.* 2007. 2423-2428.

Holleis, P., Kranz, M., and Schmidt, A. "Adding Context Information to Digital Photos." *IWSAWC'05.* 2005a. 536-542.

Holleis, P., Kranz, M., and Schmidt, A. "Displayed Connectivity." *Adjunct Proceedings Ubicomp'05.* 2005b.

Holleis, P., Kranz, M., Winter, A., and Schmidt, A. "Playing with the Real World." *Journal of Virtual Reality and Broadcasting* 3, no. 1. 2006.

Holleis, P., Otto, F., Hussmann, H., and Schmidt, A. "Keystroke-level Model for Advanced Mobile Phone Interaction." *CHI'07.* 2007. 1505-1514.

Holleis, P., Paasovaara, S., Puikonen, A., Häkkilä, J., and Schmidt, A. "Evaluating Capacitive Touch Input on Clothes." *MobileHCI'08.* 2008.

Holleis, P., Rukzio, E., Kraus, T., and Schmidt, A. "Environment Based Messaging." *Advances in Pervasive Computing, Pervasive'06.* 2006a.

Holleis, P., Rukzio, E., Kraus, T., and Schmidt, A. "Interacting with Tangible Displays." *Advances in Pervasive Computing, Pervasive'06.* 2006b.

Holleis, P., Rukzio, E., Otto, F., and Schmidt, A. "Privacy and Curiosity in Mobile Interactions with Public Displays." *MSI'07.* 2007.

Hong, J. I., Heer, J., Waterson, S., and Landay, J. A. "WebQuilt: A Proxy-based Approach to Remote Web Usability Testing." *ACM Transactions on Information Systems* 19, no. 3. 2001. 263-385.

How, Y. and Kan, M. Y. "Optimizing Predictive Text Entry for Short Message Service on Mobile Phones." *HCII'05.* 2005.

Hudson, S. E. and Mankoff, J. "Rapid Construction of Functioning Physical Interfaces from Cardboard, Thumbtacks, Tin Foil and Masking Tape." *UIST'06.* 2006. 289-298.

Hudson, S. E., John, B. E., Knudsen, K., and Byrne, M. D. "A Tool for Creating Predictive Performance Models from User Interface Demonstrations." *UIST'99.* 1999. 93-102.

Hull, R., Clayton, B., and Melamed, T. "Rapid Authoring of Mediascapes." *Ubicomp'04.* 2004. 125-142.

Hurford, R., Martin, A., and Larsen, P. "Designing Wearables." *ISWC'06.* 2006. 133-134.

Hutchins, E. "Organizing Work by Adaptation." Vol. 2, in *Organization Science*, 14-39. 1991.

Irving, S., Polson, P., and Irving, J. E. "A GOMS Analysis of the Advanced Automated Cockpit." *CHI'94.* 1994. 344-350.

Ishii, H. and Ullmer, B. "Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms." *CHI'97.* 1997. 234-241.

Isokoski, P. and MacKenzie, I. S. "Combined Model for Text Entry Rate Development." *CHI'03.* 2003. 752-753.

Isokoski, P. "Model for Unistroke Writing Time." *CHI'01.* 2001. 357-364.

Ivory, M. Y. and Hearst, M. A. "The State of the Art in Automating Usability Evaluation of User Interfaces." *ACM Computing Surveys* 33, no. 4. 2001. 470-516.

Izadi, S., Coutinho, P., Rodden, T., and Smith, G. "The FUSE Platform: Supporting Ubiquitous Collaboration Within Diverse Mobile Environments." *Automated Software Engineering* 9, no. 2. 2004. 167-186.

Jaimes, A. and Miyazaki, J. "Building a Smart Meeting Room: from Infrastructure to the Video Gap." *MobiDE'05.* 2005. 1173-1173.

James, C. L. and Reischel, K. M. "Text Input for Mobile Devices: Comparing Model Prediction to Actual Performance." *CHI'01.* 2001. 365-371.

Jameson, A. and Krüger, A. "Preface to the Special Issue on User Modeling in Ubiquitous Computing." *User Modeling and User-Adapted Interaction* 15, no. 3-4. 2005. 193-195.

Johanson, B. and Fox, A. "The Event Heap: a Coordination Infrastructure for Interactive Workspaces." *WMCSA'06.* 2002. 83-93.

John, B. E. and Gray, W. D. "GOMS Analysis for Parallel Activities." *Conference Companion CHI'94.* 1994. 395-396.

John, B. E. and Kieras, D. E. "The GOMS Family of Analysis Techniques: Tools for Design and Evaluation." Technical Report CMU-CS-94-181, School of Computer Science, Carnegie Mellon University, 1994.

John, B. E. and Kieras, D. E. "Using GOMS for User Interface Design and Evaluation: Which Technique?" *ACM Transactions on Computer-Human Interaction* 3, no. 4. 1996. 287-319.

John, B. E. and Newell, A. "Cumulating the Science of HCI: from S-R Compatibility to Transcription Typing." *CHI'89.* 1989. 109-114.

John, B. E. and Salvucci, D. D. "Multi-Purpose Prototypes for Assessing User Interfaces in Pervasive Computing Systems." *IEEE Pervasive Computing* 4, no. 4. 2005. 27-34.

John, B. E. and Vera, A. H. "A GOMS Analysis of a Graphic Machine-paced, Highly Interactive Task." *CHI'92.* 1992. 251-258.

John, B. E. "Extensions of GOMS Analyses to Expert Performance Requiring Perception of Dynamic Visual and Auditory Information." *CHI'90.* 1990. 107-115.

John, B. E. "Why GOMS?" *Interactions* 2, no. 4. 1995. 80-89.

John, B. E., Prevas, K., Salvucci, D. D., and Koedinger, K. "Predictive Human Performance Modeling Made Easy." *CHI'04.* 2004. 455-462.

John, B. E., Salvucci, D. D., Centgraf, P., and Prevas, K. "Integrating Models and Tools in the Context of Driving and In-vehicle Devices." *ICCM'04.* 2004. 130-135.

John, B. E., Vera, A. H., and Newell, A. "Toward Real-time GOMS: a Model of Expert Behavior in a Highly Interactive Task." *Behaviour and Information Technology* 13, no. 4. 1993. 255-267.

Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. "The reacTable: Exploring the Synergy Between Live Music Performance and Tabletop Tangible Interfaces." *TEI'07.* 2007. 139-146.

Jung, S., Lauterbach, C., and Weber, W. "A Digital Music Player Tailored for Smart Textiles: First Results." *Avantex Symposium.* 2002.

Kaltenbrunner, M. and Bencina, R. "reacTIVision: a Computer-vision Framework for Table-based Tangible Interaction." *TEI'07.* 2007. 69-74.

Kankaanpaa, A. I. "FIDS - A flat-panel Interactive Display System." *IEEE Computer Graphics and Applications* 8, no. 2. 1988. 71-82.

Kawsar, F., Kortuem, G., Terada, T., Fujinam, K., Nakazawa, J., and Pirttikangas, S. *Workshop DIPSO'07, Ubicomp'07.* 2007.

Kern, D., Müller, M., Schneegaß, S., Wolejko-Wolejszo, L., and Schmidt, A. "CARS – Configurable Automotive Research Simulator." *AUIIA'08.* 2008.

Kieras, D. E. and Santoro, T. "Computational GOMS Modeling of a Complex Team Task: Lessons Learned." *CHI'04.* 2004. 97-104.

Kieras, D. E. "Towards a Practical GOMS Model Methodology for User Interface Design." In *Handbook of Human-Computer Interaction*, 135-158. 1988.

Kieras, D. E. "Using the Keystroke-Level Model to Estimate Execution Times." Unpublished Report, http://www.pitt.edu/~cmlewis/KSM.pdf, University of Michigan, 1993.

Kieras, D. E., Wood, S. D., Abotel, K., and Hornof, A. "GLEAN: a Computer-based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs." *UIST'95.* 1995.

Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., et al. "People, Places, Things: Web Presence for the Real World." *Mobile Networks and Applications* 7, no. 5. 2002. 365-376.

Klemmer, S. R. "Tangible User Interface Input: Tools and Techniques." PhD Thesis, Computer Science Division, University of California at Berkeley, 2004.

Klemmer, S. R., Li, J., Lin, J., and Landay, J. A. "Papier-Mâché: Toolkit Support for Tangible Input." *CHI'04.* 2004. 399-406.

Klemmer, S. R., Sinha, A. K., Chen, J., Landay, J. A., Aboobaker, N., and Wang, A. "SUEDE: A Wizard of Oz Prototyping Tool for Speech User Interfaces." *UIST'00.* 2000. 1-10.

Knight, J. F. and Baber, C. "A Tool to Assess the Comfort of Wearable Computers." *Human Factors* 47, no. 1. 2005. 77-91.

Knight, J. F., Deen-Williams, D., Arvanitis, T. N., Baber, C., Sotiriou, S., Anastopoulou, S., et al. "Assessing the Wearability of Wearable Computers." *ISWC'06.* 2006. 75-82.

Kobayashi, S., Endo, T., Harada, K., and Oishi, S. "GAINER: a Reconfigurable I/O Module and Software Libraries for Education." *NIME'06.* 2006. 346-351.

Koester, H. H. and Levine, S. P. "Validation of a Keystroke-Level Model for a Text Entry System Used by People with Disabilities." *Assets'94.* 1994. 115-122.

Kranz, M. "Engineering Perceptive User Interfaces." PhD Thesis, University of Munich, Department of Media Informatics, 2008.

Kranz, M., Freund, S., Holleis, P., Schmidt, A., and Arndt, H. "Developing Gestural Input." *IWSAWC'06.* 2006. 63-68.

Kranz, M., Holleis, P., and Schmidt, A. "DistScroll - A New One-Handed Interaction Device." *IWSAWC'05.* 2005. 499-505.

Kranz, M., Holleis, P., and Schmidt, A. "Ubiquitous Presence Systems." *SAC'06.* 2006. 1902-1909.

Kranz, M., Schmidt, D., Holleis, P., and Schmidt, A. "A Display Cube as Tangible User Interface." *Adjunct Proceedings Ubicomp'05.* 2005.

Laerhoven, K. V. and Gellersen, H. "Spine versus Porcupine: a Study in Distributed Wearable Activity Recognition." *ISWC'04.* 2004. 142-149.

Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: an Architecture for General Intelligence." *Artificial Intelligence* 33. 1987. 1-64.

Lane, D. M., Napier, H. A., Batsell, R. R., and Naman, J. "Predicting the Skilled Use of Hierarchical Menus with the Keystroke-level Model." *Human-Computer Interaction* 8. 1993. 185-192.

Lawrence, D., Atwood, M. E., and Dews, S. "Surrogate Users: Mediating Between Social and Technical Interaction." *CHI'94.* 1994. 399-404.

Lee, J. C., Avraham, D., Hudson, S. E., Forlizzi, J., Dietz, P. H., and Leigh, D. "The Calder Toolkit: Wired and Wireless Components for Rapidly Prototyping Interactive Devices." *DIS'04.* 2004.

Lee, S., Buxton, W., and Smith, K. C. "A Multi-Touch Three Dimensional Touch-Sensitive Tablet." *CHI'85.* 1985. 21-25.

Li, Y., Hong, J., and Landay, J. A. "Topiary: A Tool for Prototyping Location-Enhanced Applications." *UIST'04.* 2004.

Lieberman, H., Paternò, F., and Wulf, V. *End-User Development.* Springer, 2006.

Linz, T., Kallmayer, C., Aschenbrenner, R., and Reichl, H. "Embroidering Electrical Interconnects with Conductive Yarn for the Integration of Flexible Electronic Modules into Fabric." *ISWC'05.* 2005. 86-91.

Liu, Y., Liu, X., and Jia, Y. "Hand-gesture Based Text Input for Wearable Computers." *ICVS'06.* 2006.

Lombriser, C., Bharatula, N. B., Roggen, D., and Tröster, G. "On-Body Activity Recognition in a Dynamic Sensor Network." *BodyNets'07.* 2007.

Lorenz, A., Eisenhauer, M., and Zimmermann, A. "Elaborating a Framework for Open Human Computer Interaction with Ambient Services." *PERMID'08.* 2008.

Luo, L. and John, B. E. "Predicting Task Execution Time on Handheld Devices Using the Keystroke-Level Model." *CHI'05.* 2005. 1605-1608.

Luo, L. and Siewiorek, D. P. "KLEM: a Method for Predicting User Interaction Time and System Energy Consumption during Application Design." *ISWC'07.* 2007. 69-76.

Lynch, P. J. and Horton, S. *Web Style Guide: Basic Design Principles for Creating Web Sites.* Yale University Press, 2004.

Lyons, K., Plaisted, D., and Starner, T. "Expert Chording Text Entry on the Twiddler One-handed Keyboard." *ISWC'04.* 2004. 94-101.

Ma, H. and Paradiso, J. A. "The FindIT Flashlight: Responsive Tagging Based on Optically Triggered Microprocessor Wakeup." *Ubicomp'02.* 2002. 655-662.

MacIntyre, B., Gandy, M., Dow, S., and Bolter, J. D. "DART: a Toolkit for Rapid Design Exploration of Augmented Reality Experiences." *UIST'04.* 2004. 197-206.

MacKenzie, I. S. "A Note on the Information-theoretic Basis for Fitts' Law." *Journal of Motor Behavior* 21. 1989. 323-330.

MacKenzie, I. S. and Buxton, W. "Extending Fitts' Law to Two Dimensional Tasks." *CHI'92.* 1992.

MacKenzie, I. S. and Guiard, Y. "The Two-handed Desktop Interface: Are We there yet?" *CHI'01.* 2001. 351-352.

MacKenzie, I. S. and Soukoreff, R. W. "A Model of Two-thumb Text Entry." *GI'02.* 2002. 117-124.

MacKenzie, I. S. and Soukoreff, R. W. "Phrase Sets for Evaluating Text Entry Techniques." *CHI'03 Extended Abstracts.* 2003. 754-755.

MacKenzie, I. S. "Fitts' Law as a Performance Model in Human-Computer Interaction." PhD thesis, Department of Education, University of Toronto, 1991.

MacKenzie, I. S. "Fitts' Law as a Research and Design Tool in Human-computer Interaction." *Human-Computer Interaction* 7. 1992. 91-139.

MacKenzie, I. S. "Motor Behaviour Models for Human-computer Interaction." In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, 27-54. 2003.

MacKenzie, I. S., Zhang, S. X., and Soukoreff, R. W. "Text Entry Using Soft Keyboards." *Behaviour and Information Technology* 18, no. 4. 1999. 235-244.

Magerkurth, C., Cheok, A. D., Mandryk, R. L., and Nilsen, T. "Pervasive Games: Bringing Computer Entertainment Back to the Real World." *Computers in Entertainment* 3, no. 3. 2005. 11-29.

Mahato, H., Kern, D., Holleis, P., and Schmidt, A. "Implicit Personalization of Public Environments Using Bluetooth." *CHI'08 Extended Abstracts.* 2008. 3093-3098.

Mäkitalo-Siegl, K., Kaplan, F., Zottmann, J., Dillenbourg, P., and Fischer, F. *Workshop Classroom of the Future, Alpine CSCL Rendez-vous 2007.* 2007.

Manes, D., Green, P., and Hunter, D. "Prediction of Destination Entry and Retrieval Times Using Keystroke-Level Models." Technical Report UMTRI-96-37, Transportation Research Institute, The University of Michigan, 1996.

Marculescu, D., Marculescu, R., Zamora, N. H., Stanley-Marbell, P., Khosla, P. K., Park, S., et al. "Electronic Textiles: A Platform for Pervasive Computing." *Proceedings of the IEEE,* 2003.

Martin, F., Mikhak, B., and Silverman, B. "MetaCricket: a Designer's Kit for Making Computational Devices." *IBM Systems Journal* 39, no. 3-4. 2000. 795-815.

Matessa, M., Remington, R., and Vera, A. "How Apex Automates CPM-GOMS." *ICCM'03.* 2003. 93-98.

Mateu, L. and Moll, F. "Review of Energy Harvesting Techniques and Applications." *SPIE Microtechnologies for the New Millenium.* 2005. 359-373.

Matthews, T, Gellersen, H.-W., Van Laerhoven, K., and Dey, A. K. "Augmenting Collections of Everyday Objects: a Case Study of Clothes Hangers as an Information Display." *Pervasive'04.* 2004. 340-344.

Matthews, T., Dey, A. K., Mankoff, J., Carter, S., and Rattenbury, T. "A Toolkit for Managing User Attention in Peripheral Displays." *UIST'04.* 2004. 247-256.

Mattmann, C. and Tröster, G. "Design Concept of Clothing Recognizing Back Postures." *ISSS-MDBS'06.* 2006.

Mayol, W. W., Tordoff, B., and Murray, D. W. "On the Positioning of Wearable Optical Devices." Technical Report OUEL224101, Oxford University, 2001.

Mayrhofer, R. "Towards an Open Source Toolkit for Ubiquitous Device Authentication." *PerCom'07.* 2007. 247-254.

Mazalek, A. "Tangible Toolkits: Integrating Application Development accross Diverse Multi-User and Tangible Interaction Platforms." *Workshop Let's Get Physical, DCC'06.* 2006.

McCann, J., Hurford, R., and Martin, A. "A Design Process for the Development of Innovative Smart Clothing that Addresses End-User Needs from Technical, Functional, Aesthetic and Cultural View Points." *ISWC'05.* 2005. 70-77.

McCrickard, D. S., Bussert, D., and Wrighton, D. "A Toolkit for the Construction of Real World Interfaces." *ACMSE'03.* 2003. 118-123.

Messer, A., Kunjithapatham, A., Sheshagiri, M., Song, H., Kumar, P., Nguyen, P., et al. "InterPlay: a Middleware for Seamless Device Integration and Task Orchestration in a Networked Home." *PerCom'06.* 2006.

Meyer, D. E. and Kieras, D. E. "A Computational Theory of Executive Cognitive Processes and Multiple-task Performance." *Psychological Review* 104, no. 1. 1997. 3-65.

Min, D., Koo, S., Chung, Y., and Kim, B. "Distributed GOMS: an Extension of GOMS to Group Task." *IEEE Conference on Systems, Man, and Cybernetics.* 1999. 720-725.

Monroy-Hernández, A. and Resnick, M. "Empowering Kids to Create and Share Programmable Media." *interactions* 15, no. 2. 2008. 104-109.

Mori, G., Paterno, F., and Santoro, C. "CTTE: Support for Developing and Analyzing Task Models for Interactive System Design." *IEEE Transactions on Software Engineering* 28, no. 8. 2002. 797-813.

Mori, R., Matsunobe, T., and Yamaoka, T. "A Task Operation Prediction Time Computation Based on GOMS-KLM Improved for the Cellular Phone and the Verification of that Validity." *ADC'03.* 2003.

Morris, M. R., Cassanego, A., Paepcke, A., Winograd, T., Winograd, T., Piper, A. M., et al. "Mediating Group Dynamics through Tabletop Interface Design." *IEEE Computer Graphics and Applications* 26, no. 5. 2006. 65-73.

Müller, M., Holleis, P., and Schmidt, A. "The Smart Transport Container - Integration of Web Services with Augmented Real World Objects." *Video Proceedings Pervasive'07.* 2007.

Myers, B. A. "Mobile Devices for Control." *MobileHCI'02.* 2002. 1-8.

Myung, R. "Keystroke-Level Analysis of Korean Text Entry Methods on Mobile Phones." *International Journal of Human-Computer Studies* 60, no. 5-6. 2004. 545-563.

Nachman, L., Kling, R., Adler, R., Huang, J., and Hummel, V. "The Intel® Mote platform: a Bluetooth-based Sensor Network for Industrial Monitoring." *IPSN'05.* 2005.

Nardi, B. A. *A Small Matter of Programming: Perspectives.* The MIT Press, 1993.

Nath, S., Ke, Y., Gibbons, P. P., Karp, B., and Seshan, S. "IrisNet: an Architecture for Enabling Sensor-enriched Internet Services." Technical Report IRP-TR-02-10, Intel Research Pittsburgh, 2002.

Neely, S., Stevenson, G., Kray, C., Mulder, I., Connelly, K., and Siek, K. A. "Evaluating Pervasive and Ubiquitous Systems." *IEEE Pervasive Computing* 7, no. 3. 2008. 85-88.

Neubauer, A. "Virtual Endoscopy for Preoperative Planning and Training of Endonasal Transsphenoidal Pituitary Surgery." PhD Thesis, VRVis and ICGA TU Wien, 2005.

Newman, M. W., Izadi, S., Edwards, W. K., Sedivy, J. Z., and Smith, T. F. "User Interfaces When and Where They are Needed: an Infrastructure for Recombinant Computing." *UIST'02.* 2002.

Newman, M. W., Lin, J., Hong, J. I., and Landay, J. A. "DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice." *Human-Computer Interaction* 18, no. 3. 2003. 259-324.

Nielsen, J. "Killing Time is the Killer Application." 2000. http://www.thefeature.com/article?articleid=8183.

Niemelä, E. and Latvakoski, J. "Survey of Requirements and Solutions for Ubiquitous Software." *MUM'04.* 2004. 71-78.

Nuria, O. and Flores-Mangas, F. "MPTrain: a Mobile Music and Physiology Based Personal Trainer." *MobileHCI'06.* 2006.

O'Hara, K., Perry, M., and Churchill, E. *Public and Situated Displays: Social and Interactional Aspects of Shared Display Technologies.* Springer, 2003.

Olson, J. R. and Olson, G. M. "The Growth of Cognitive Modeling in Human-computer Interaction since GOMS." In *Human-Computer Interaction: Toward the Year 2000*. 1995.

Orth, M., Post, R., and Cooper, E. "Fabric Computing Interfaces." *CHI'98.* 1998. 331-332.

Palanque, P. and Paternò, F. *Formal Methods in Human-Computer Interaction.* Springer, 1998.

Pangoli, S. and Paternó, F. "Automatic Generation of Task-oriented Help." *UIST'95.* 1995. 181-187.

Park, S., Mackenzie, K., and Jayaraman, S. "The Wearable Motherboard: a Framework for Personalized Mobile Information Processing (PMIP)." *DAC'02.* 2002. 170-174.

Patel, S. N., Reynolds, M. S., and Abowd, G. D. "Detecting Human Movement by Differential Air Pressure Sensing in HVAC System Ductwork: An Exploration in Infrastructure Mediated Sensing." *Pervasive'08.* 2008.

Paternò, F. "Tools for Task Modelling: Where we are, Where we are headed." *Tamodia'02.* 2002.

Pavlovych, A. and Stürzlinger, W. "Less-Tap: a Fast and Easy-to-learn Text Input Technique for Phones." *GI'03.* 2003. 97-104.

Pavlovych, A. and Stürzlinger, W. "Model for Non-expert Text Entry Speed on 12-button Phone Keypads." *CHI'04.* 2004. 351-358.

Payne, S. J. and Green, T. R. G. "Task-Action Grammars: a Model of the Mental Representation of Task Languages." *Human-Computer Interaction* 2, no. 2. 1986. 93-133.

Payne, S. J. "Task Action Grammar." *INTERACT'84.* 1984. 139-144.

Pettitt, M., Burnett, G., and Karbassioun, D. "Applying the Keystroke Level Model in a Driving Context." *Ergonomics Society Annual Meeting.* 2006.

Pettitt, M., Burnett, G., and Stevens, A. "An Extended Keystroke Level Model (KLM) for Predicting the Visual Demand of In-vehicle Information Systems." *CHI'07.* 2007.

Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., and Winograd, T. "ICrafter: a Service Framework for Ubiquitous Computing Environments." *Ubicomp'01.* 2001. 56-75.

Post, E. R., Orth, M., Russo, P. R., and Gershenfeld, N. "E-broidery: Design and Fabrication of Textile-Based Computing." *IBM Systems Journal* 39, no. 3-4. 2000. 840-860.

Poupyrev, I. and Maruyama, S. "Tactile Interfaces for Small Touch Screens." *UIST'03.* 2003. 217-220.

Power, E. J. and Dias, T. "Knitting of Electroconductive Yarns." *Eurowearable'03.* 2003. 55-60.

Prante, T., Stenzel, R., Röcker, C., Streitz, N., and Magerkurth, C. "Ambient Agoras: InfoRiver, SIAM, Hello.Wall." *CHI'04 Extended Abstracts.* 2004. 763-764.

Randell, C. "Wearable Computing: A Review." Technical Report CSTR-06-004, University of Bristol, 2005.

Ranganathan, A., Al-Muhtadi, J., and Campbell, R. H. "Reasoning about Uncertain Contexts in Pervasive Computing Environments." *IEEE Pervasive Computing* 3, no. 2. 2004. 62-70.

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., and Mickunas, M. D. "Olympus: a High-Level Programming Model for Pervasive Computing Environments." *PerCom'05.* 2005. 7-16.

Rantanen, J., Karinsalo, T., Mäkinen, M., Talvenmaa, P., Tasanen, M., Vanhala, J., et al. "Smart Clothing for the Arctic Environment." *ISWC'00.* 2000. 15-23.

Rekimoto, J. and Nagao, K. "The World Through the Computer: Computer Augmented Interaction with Real World Environments." *UIST'95.* 1995. 29-36.

Rekimoto, J. and Schwesig, C. "PreSenseII: Bi-directional Touch and Pressure Sensing Interactions with Tactile Feedback." *CHI'06 Extended Abstracts.* 2006. 1253-1258.

Rekimoto, J., Ishizawa, T., Schwesig, C., and Oba, H. "PreSense: Interaction Techniques for Finger Sensing Input Devices." *UIST'03.* 2003. 203-212.

Resnick, P. and Varian, H. R. "Recommender Systems." *Communications of the ACM* 40, no. 3. 1997. 56-58.

Rhodes, B. and Mase, K. "Wearables in 2005." *IEEE Pervasive Computing* 5, no. 1. 2006. 92-95.

Rich, E. "User Modeling via Stereotypes." *Cognitive Science* 3, no. 4. 1979. 329-354.

Ringel Morris, M. "Supporting Effective Interaction with Tabletop Groupware." PhD Thesis, Stanford HCI Group, Stanford University, 2006.

Ritter, F. E. and Young, R. M. "Embodied Models as Simulated Users." *International Journal of Human Computer Studies* 55, no. 1. 2001. 1-14.

Ritter, F. E. "Choosing and Getting Started with a Cognitive Architecture to Test and Use Human-machine Interfaces." *MMI interaktiv* 7. 2004. 17-39.

Rodden, T. and Benford, S. "The Evolution of Buildings and Implications for the Design of Ubiquitous Domestic Environments." *CHI'03.* 2003. 9-16.

Rodden, T., Crabtree, A., Hemmings, T., Koleva, B., Humble, J., Åkesson, K.-P., et al. "Configuring the Ubiquitous Home." *COOP'04.* 2004.

Rohs, M. and Gfeller, B. "Using Camera-equipped Mobile Phones for Interacting with Real-world Objects." *Advances in Pervasive Computing, Pervasive'04.* 2004. 265-271.

Rohs, M. "Marker-based Interaction Techniques for Camera-phones." *MU3I.* 2005.

Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. "A Middleware Infrastructure for Active Spaces." *IEEE Pervasive Computing* 1, no. 4. 2002. 74-83.

Ronkainen, S., Häkkilä, J., Kaleva, S., Colley, A., and Linjama, J. "Tap Input as an Embedded Interaction Method for Mobile Devices." *TEI'07.* 2007. 263-270.

Rukzio, E., Leichtenstern, K., Callaghan, V., Holleis, P., and Schmidt, A. "An Experimental Comparison of Physical Mobile Interaction Techniques: Touching, Pointing and Scanning." *Ubicomp'06.* 2006. 87-104.

Rukzio, E., Paolucci, M., Wagner, M., Berndt, H. H., Hamard, J., and Schmidt, A. "Mobile Service Interaction with the Web of Things." *ICT'06.* 2006.

Rukzio, E., Schmidt, A., and Hussmann, H. "Physical Posters as Gateways to Context-aware Services for Mobile Devices." *WMCSA'04.* 2004.

Rukzio, E., Wetzstein, S., and Schmidt, A. "A Framework for Mobile Interactions with the Physical World." *WPMC'05.* 2005.

Sahami, A., Holleis, P., Schmidt, S., and Häkkilä, J. "Rich Tactile Output on Mobile Devices." *AMI'08.* 2008.

Salisbury, J. K. and Srinivasan, M. A. "Phantom-based Haptic Interaction with Virtual Objects." *IEEE Computer Graphics and Applications* 17, no. 5. 1997. 6-10.

Salvucci, D. D. and Lee, F. J. "Simple Cognitive Modeling in a Complex Cognitive Architecture." *CHI'03.* 2003. 265-272.

Salvucci, D. D. "Modeling Driver Distraction from Cognitive Tasks." *CogSci'02.* 2002. 792-797.

Salvucci, D. D. "Predicting the Effects of In-car Interface Use on Driver Performance: an Integrated Model Approach." *International Journal of Human-Computer Studies* 55, no. 1. 2001. 85-107.

Salvucci, D. D., Zuber, M., Beregovaia, E., and Markley, D. "Distract-R: Rapid Prototyping and Evaluation of In-vehicle Interfaces." *CHI'05.* 2005. 581-589.

Santini, S., Adelmann, R., Langheinrich, M., Schätti, G., and Fluck, S. "JSense - Prototyping Sensor-based, Location-aware Applications in Java." *UCS'06.* 2006. 300-315.

Santoro, T. P., Kieras, D. E., and Campbell, G. E. "GOMS Modeling Application to Watchstation Design Using the GLEAN Tool." *I/ITSEC'00.* 2000. 964-973.

Satyanarayanan, M. "Pervasive Computing: Vision and Challenges." *IEEE Personal Communications* 8, no. 4. 2001. 10-17.

Schmidt, A. "Network Alarm Clock." *Personal and Ubiquitous Computing* 10, no. 2-3. 2005. 191-192.

Schmidt, A., Beigl, M., and Gellersen, H.-W. "There is More to Context than Location." *IMC'98.* 1998.

Schmidt, A., Häkkilä, J., Atterer, R., Rukzio, E., and Holleis, P. "Utilizing Mobile Phones as Ambient Information Displays." *CHI'06 Extended Abstracts.* 2006. 1295-1300.

Schmidt, A., Holleis, P., and Kranz, M. "Sensor Virrig - A Balance Cushion as Controller." *Workshop Playing with Sensors, UbiComp'04.* 2004.

Schmidt, A., Kranz, M., and Holleis, P. "Embedding Information." *Workshop on Ubiquitous Display Environments at Ubicomp'04.* 2004.

Schmidt, A., Kranz, M., and Holleis, P. "Interacting with the Ubiquitous Computer: Towards Embedding Interaction." *sOc-EUSAI'05.* 2005. 147-152.

Schmidt, A., Terrenghi, L., and Holleis, P. "Methods and Guidelines for the Design and Development of Domestic Ubiquitous Computing Applications." *Journal of Pervasive and Mobile Computing* 3, no. 6. 2007. 721-738.

Scholtz, J. and Consolvo, S. "Toward a Framework for Evaluating Ubiquitous Computing Applications." *IEEE Pervasive Computing* 3, no. 2. 2004. 82-88.

Shannon, C. E. "Communication in the Presence of Noise." *Proceedings of the IRE,* 1949: 10-21.

Shen, C., Vernier, F. D., Forlines, C., and Ringel, M. "DiamondSpin: an Extensible Toolkit for Around-the-Table Interaction." *CHI'04.* 2004. 167-174.

Silfverberg, M., MacKenzie, I. S., and Korhonen, P. "Predicting Text Entry Speed on Mobile Phones." *CHI'00.* 2000. 9-16.

Sohn, T. and Dey, A. K. "iCAP: Rapid Prototyping of Context-aware Applications." *CHI'04.* 2004. 103-129.

St. Amant, R. "Interface Agents as Surrogate Users." *Intelligence* 11, no. 2. 2000. 28-38.

St. Amant, R., Horton, T. E., and Ritter, F. E. "Model-based Evaluation of Cell Phone Menu Interaction." *CHI'04.* 2004. 343-350.

Stahl, C. and Haupert, J. "Taking Location Modelling to New Levels: a Map Modelling Toolkit for Intelligent Environments." *LoCA'06.* 2006. 74-85.

Starner, T. E. "The Role of Speech Input in Wearable Computing." *IEEE Pervasive Computing* 1, no. 3. 2002. 89-93.

Steiniger, S., Neun, M., and Edwardes, A. "Lecture Notes: Foundations of Location Based Services." Department of Geography, University of Zürich, 2006.

Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A., and Estrin, D. "Habitat Monitoring with Sensor Networks." *Communications of the ACM* 47, no. 6. 2004. 34-40.

Teo, L. and John, B. E. "Comparisons of Keystroke-Level Model Predictions to Observed Data." *CHI'06 Extended Abstracts.* 2006. 1421-1426.

Teo, L., John, B. E., and Pirolli, P. "Towards a Tool for Predicting User Exploration." *CHI'07 Extended Abstracts.* 2007. 2687-2692.

Terrenghi, L., Kranz, M., Holleis, P., and Schmidt, A. "A Cube to Learn: a Tangible User Interface for the Design of a Learning Appliance." *Personal and Ubiquitous Computing* 10, no. 2-3. 2005. 153-158.

Thimbleby, H. and Gow, J. "Applying Graph Theory to Interaction Design." *DSVIS'07.* 2007.

Thimbleby, H., Cairns, P., and Jones, M. "Usability Analysis with Markov Models." *ACM Transactions on Computer-Human Interaction* 8, no. 2. 2001. 99-132.

Thomas, B. H. "Minimal Social Weight User Interactions for Wearable Computers in Business Suits." *ISWC'07.* 2002. 57.

Thomas, B., Grimmer, K., Makovec, D., Zucco, J., and Gunther, B. "Determination of Placement of a Body-Attached Mouse as a Pointing Input Device for Wearable Computers." *ISWC'99.* 1999. 193.

Thompson, D. *Embedded Programming with the Microsoft .NET Micro Framework (Pro - Developer).* Microsoft Press, 2007.

Toney, A., Thomas, B. H., and Marais, W. "Managing Smart Garments." *ISWC'06.* 2006. 91-94.

Tse, E. and Greenberg, S. "Rapidly Prototyping Single Display Groupware through the SDGToolkit." *AUIC'04.* 2004. 101-110.

Tuulos, V., H., Scheible, J., and Nyholm, H. "Combining Web, Mobile Phones and Public Displays in Large-Scale: Manhattan Story Mashup." *Pervasive'07.* 2007. 37-54.

Välkkynen, P. and Tuomisto, T. "Physical Browsing Research." *PERMID'05.* 2005. 35-38.

Välkkynen, P., Korhonen, I., Plomp, J., Tuomisto, T., Cluitmans, L., Ailisto, H., et al. "A User Interaction Paradigm for Physical Browsing and Near-object Control Based on Tags." *Workshop PI'03, MobileHCI'03.* 2003. 31-34.

Van Kleek, M., Kunze, K., Partridge, K., and Begole, J. "OPF: A Distributed Context-Sensing Framework for Ubiquitous Computing Environments." *Ubicomp'06.* 2006. 82-97.

Van Laerhoven, K., Schmidt, A., and Gellersen, H.-W. "Pin&Play: Networking Objects through Pins." *Ubicomp'02.* 2002. 1-8.

Villar, N. and Gellersen, H.-W. "A Malleable Control Structure for Softwired User Interfaces." *TEI'07.* 2007. 49-56.

Wade, E. and Asada, H. "Conductive Fabric Garment for a Cable-Free Body Area Network." *IEEE Pervasive Computing* 6, no. 1. 2007. 52-58.

Wade, E. and Asada, H. "Wearable DC Powerline Communication Network Using Conductive Fabrics." *ICRA'04.* 2004. 4085-4090.

Wagner, D., Pintaric, T., Ledermann, F., and Schmalstieg, D. "The Invisible Train." *Pervasive'05.* 2005.

Wahlster, W. and Kobsa, A. "User Models in Dialog Systems." In *User Models in Dialog Systems*, by A. Kobsa and W. Wahlster, 4-34. 1989.

Want, R. and Pering, T. "System Challenges for Ubiquitous & Pervasive Computing." *ICSE'05.* 2005. 9-14.

Want, R., Fishkin, K. P., Gujar, A., and Harrison, B. L. "Bridging Physical and Virtual Worlds with Electronic Tags." *CHI'99.* 1999. 370-377.

Weis, T., Handte, M., Knoll, M., and Becker, C. "Customizable Pervasive Applications." *PerCom'06.* 2006. 239-244.

Weis, T., Knoll, M., Ulbrich, A., Mühl, G., and Brändle, A. "Rapid Prototyping for Pervasive Applications." *IEEE Pervasive Computing* 6, no. 2. 2007. 76-84.

Weiser, M. and Brown, J. S. "Designing Calm Technology." *PowerGrid Journal* 1, no. 1. 1996.

Weiser, M. "The Computer for the Twenty-First Century." *Scientific American* 265, no. 3. 1991. 94-104.

Werner, J., Wettach, R., and Hornecker, E. "United-pulse: Feeling your Partner's Pulse." *MobileHCI'08.* 2008. 535-538.

Wigdor, D. and Balakrishnan, R. "A Comparison of Consecutive and Concurrent Input Text Entry Techniques for Mobile Phones." *CHI'04.* 2004. 81-88.

Witt, H., Nicolai, T., and Kenn, H. "The WUI-Toolkit: A Model-Driven UI Development Framework for Wearable User Interfaces." *ICDCSW'07.* 2007. 43-48.

Yau, S. S., Karim, F., Wang, Y., Wang, B., and Gupta, S. K. S. "Reconfigurable Context-sensitive Middleware for Pervasive Computing." *IEEE Pervasive Computing* 1, no. 3. 2002. 33-40.

Zeleznik, R., Miller, T., and Forsberg, A. "Pop Through Mouse Button Interactions." *UIST'01.* 2001. 195-196.

Zhang, J., Chen, X., Yang, J., and Waibel, A. "A PDA-based Sign Translator." *ICMI'02.* 2002. 217-222.

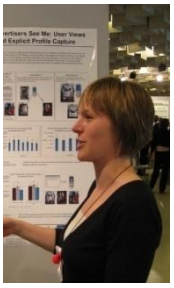
Acknowledgements

The following people supported me, this thesis or the projects described in this thesis. I want to thank all of them for their contributions. My general thanks go to the members of the thesis committee, especially to Prof. Dr. Antonio Krüger (from the University of Münster) for acting as external supervisor and offering extremely valuable input and motivation. In addition, I want to specifically thank Prof. Dr. Heinrich Hußmann (from the Ludwig-Maximilians-University of Munich) for providing numerous valuable comments on my thesis and work in general as well as for his strong support throughout my time in Munich and beyond.

The list is surely not exhaustive and I apologise for everyone and every effort I might have missed to note here. Be assured that it happened by pure obliviousness. With the notable exception of the first and last entry, the list is meant to be sorted alphabetically.

Albrecht Schmidt who initially brought me to those fascinating topics, taught me so much over the years, and made my work so enjoyable. He brilliantly manages to personify the best out of an idea generator, a library on HCI and pervasive computing, an advising professor, and just a most amiable and cool person. Also thanks a lot for being the living proof that a completely full timetable still allows for a lot of guidance, help, and challenging discussions!

Dagmar Kern with whom I spent a great time as colleague in Bonn and in Essen. Thanks for using and further developing the EIToolkit as well as KLM for specific settings such as car interfaces and, considerably more important, in general for always caring for so much more than necessary!



Dave Molyneaux who should not only be known because he was in the Antarctic for several years but also for his work combining smart artefacts and projector-camera systems, and because he is so much fun to work with. Thanks for using and working with the EITookit!

Dominik Schmidt who created the 3D design of the wireless display housing which I liked and used a lot.

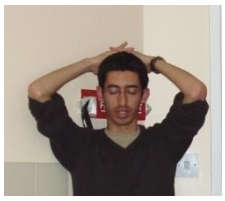
Friederike Otto who put much effort and wit into the studies for the advanced mobile phone KLM.

Enrico Rukzio who is always a great inspiration and someone who one can absolutely rely upon. Thanks for the many common projects (hopefully also in the future) and also for organising equally many entertaining events.

Florian Alt with whom I worked together at the Universities of Bonn and Duisburg-Essen, who made me enjoy playing tennis again, and who offered many valuable comments to this thesis.

Jonna Häkkilä who managed my stay in the Nokia research centre in Helsinki and who – together with her great team – turned that summer into a fantastic and really exceptional experience for me!

Matthias Kranz who was my colleague in Munich for three years and whose dedication towards his projects and his many ideas are impressive. He had much influence on pushing many projects forward but we still never forgot to have a good laugh together!



Nic Villar who always comes up with something very cool and unexpected and who is bringing VoodooIO to perfection. Thanks for the good fun in working together on, among many things, the EIToolkit!

Raphael Wimmer who is great to work with, never stops having funny ideas and often shared his valuable knowledge about capacitive sensing with us (see his ongoing work on the CapToolKit)!

Corinna Seitz who not only was with me and supported me for the last seven years (and will hopefully continue to do so as my wife) but who also endured long periods of being separated because I was at conferences, four weeks at the University of Lancaster, four months at Nokia Research in Helsinki, or finally over one year finishing my PhD and working with Albrecht Schmidt's group in Bonn and the University of Duisburg-Essen. It is also so great to have someone who understands me not only as a person but also what I am doing at work (and added +1 to the number of people who actually read this thesis)!
Thank you for all your love!

This work has been conducted in the context of the Research Group Embedded Interaction ('Eingebettete Interaktion') and was funded by the DFG (Deutsche Forschungsgemeinschaft).

Curriculum Vitae / Lebenslauf

Paul Holleis

Personal information	Nationality	German
	Date of birth	09/12/1978
	Place of birth	Bad Reichenhall, Germany
Work	since Oct. 08	Researcher, NTT DOCOMO Euro Labs, Munich
	Oct. 07 – Sep. 08	Research Assistant, University of Duisburg-Essen
	May 07 – Aug. 07	Research Intern, Nokia Research, Finland
	Apr. 07 – Oct. 07	Research Assistant, University of Bonn
	May 04 – Mar. 07	Research Assistant, Ludwig-Maximilians-University Munich
Education	Nov. 98 – Jan. 04	Diploma, Computer Science, University of Passau
	Oct. 00 – Aug. 01	Computer Science, University of Edinburgh, Scotland
	Sep. 89 – Jun. 98	Karlsgymnasium Bad Reichenhall

Persönliche Informationen	Nationalität	deutsch
	Geburtsdatum	09.12.1978
	Geburtsort	Bad Reichenhall, Deutschland
Anstellungen	seit Okt. 08	Researcher, NTT DOCOMO Euro Labs, München
	Okt. 07 – Sep. 08	Wissenschaftl. Mitarbeiter, Universität Duisburg-Essen
	Mai 07 – Aug. 07	Research Intern, Nokia Research, Finnland
	Apr. 07 – Okt. 07	Wissenschaftl. Mitarbeiter, Universität Bonn
	Mai 04 – Mär. 07	Wissenschaftl. Mitarbeiter, LMU München
Ausbildung	Nov. 98 – Jan. 04	Diplom, Informatik, Universität Passau, Deutschland
	Okt. 00 – Aug. 01	Auslandsstudium, University of Edinburgh, Schottland
	Sep. 89 – Jun. 98	Karlsgymnasium Bad Reichenhall