



UPPSALA
UNIVERSITET

The Ubiquitous Interactor –
Mobile Services with Multiple User Interfaces

BY
STINA NYLANDER

November 2003

DIVISION OF HUMAN-COMPUTER INTERACTION
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in
Human-Computer Interaction
at Uppsala University 2003

The Ubiquitous Interactor –
Mobile Services with Multiple User Interfaces

Stina Nylander

stina.nylander@sics.se

*Division of Human-Computer Interaction
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden*

<http://www.it.uu.se/>

© Stina Nylander 2003
ISSN 0346-8887

Printed by the Department of Information Technology, Uppsala University,
Sweden

Abstract

This licentiate thesis addresses design and development problems that arise when service providers, and service end-users face the variety of computing devices available on the market. The devices are designed for many types of use in various situations and settings, which means that they have different capabilities in terms of presentation, interaction, memory, etc. Service providers often handle these differences by creating a new version for each device. This creates a lot of development and maintenance work, and often leads to restrictions on the set of devices that services are developed for. For service end-users, this means that it can be difficult to combine devices that fit the intended usage context and services that provide the needed content. New development methods that target multiple devices from the start are needed. The differences between devices call for services that can adapt to various devices, and present themselves with device specific user interfaces.

We propose a way of developing device independent services by using *interaction acts* to describe user-service interaction. Devices would interpret the interaction acts and generate user interfaces according to their own specific capabilities. Additional presentation information can be encoded in *customization forms*, to further control how the user interface would be generated. Different devices would generate different user interfaces from the same interaction acts, and a device could generate different user interfaces from the same interaction acts combined with different customization forms.

In this thesis, the interaction act and customization form concepts are described in detail. A system prototype for handling them and two sample services have been implemented. Preliminary evaluations indicate that interaction acts and customization forms constitute a feasible approach for developing services with multiple user interfaces. The thesis concludes with a discussion of the problems arising when evaluating this kind of systems, and some conclusions on how to continue the evaluation process.

Acknowledgements

I would like to thank my supervisors Bengt Sandblad and Annika Waern that have guided me through this work, each one with their own strategy to keep me on track and make me finish. Thanks also to my co-authors Magnus Boman, Markus Bylund and Annika Waern for fruitful cooperation and valuable writing help.

I would also like to thank the Swedish Agency of Innovation Systems (VINNOVA) for funding most of my work.

Special thanks to Markus Bylund who has followed every step of my work, and been my project leader in the ADAPT project. Without his patience, encouragement, and ability to clarify things on a whiteboard this work would not be finished today.

Many people have given me valuable feedback during the writing process: Magnus Boman, Markus Bylund, Fredrik Espinoza, Björn Gambäck, Fredrik Olsson, and Åsa Rudström in the HUMLE laboratory at SICS, and Brad Myers and Jeffrey Nichols at Carnegie Mellon University. Evy-Ann Forssner and Nils af Geijerstam have given me additional linguistic advice.

Anna Sandin and Ola Hamfors have given me important practical help: Anna with the implementation of the HTML interaction engine, and Ola with crucial advice concerning the sView system.

Finally, I would like to thank my friends that have supported me through ups and downs during these three years: Ingrid, Madeleine, and Eva for listening to endless stories from my work; my wonderful aunt Evy-Ann for always being there for me; Lisa for letting me ride her horse; Qrut for making me leave work; Åsa for being my very best and most patient friend, and Lukas for his fabulous talent of saving even the worst day of thesis writing.

Preface

This licentiate thesis has two sections. The first section contains a background and a summary of the work. The second section contains four papers giving more details on related work (paper 1), the implementation (papers 2 and 3), and the evaluation (paper 4). The papers are:

1. **Different Approaches to Achieving Device Independence.** Stina Nylander. Published as a SICS Technical Report, no. T2003-16.
2. **The Ubiquitous Interactor – Mobile Services with Multiple User Interfaces.** Stina Nylander, Markus Bylund, Annika Waern. Published as a SICS Technical Report, no. T2003-17. Shorter version submitted to the conference Computer-Aided Design of User Interfaces.
3. **Mobile Access to Real-Time Information.** Stina Nylander, Markus Bylund, Magnus Boman. Accepted for publication in the journal Personal and Ubiquitous Computing, Springer Verlag.
4. **Evaluating the Ubiquitous Interactor.** Stina Nylander. Published as a SICS Technical Report, no. T2003-19.

Parts of this work have previously been published in:

Nylander, S. and Bylund, M. (2002) Device Independent Services, SICS Technical Report T2002-02, Swedish Institute of Computer Science.

Nylander, S. and Waern, A. (2002) Interaction Acts for Device Independent Gaming, SICS Technical Report T2002-04, Swedish Institute of Computer Science.

Nylander, S. and Bylund, M. (2002) Mobile Services for Many Different Devices, in Proceedings of Human Computer Interaction 2002, London (poster).

Nylander, S. and Bylund, M. (2002) Providing Device Independence to Mobile Services, in Proceedings of 7th ERCIM Workshop User Interfaces for All.

Nylander, S. and Bylund, M. (2003) The Ubiquitous Interactor: Universal Access to Mobile Services, in Proceedings of HCI International, Crete.

Outline

The purpose of this thesis is to discuss the problems that arise when designing and developing services for mobile devices, and to suggest a solution. The first three sections of the summary give an introduction to the problems with some background information and related work. In section 4, our design solution is presented, and in section 5 the prototype implementation is described. Section 6 presents the results and an evaluation of the work, and section 7 gives some directions for the future.

The first paper is a thorough overview of related work and systems that provided inspiration to the Ubiquitous Interactor. The second paper is a technical paper that describes the whole development of the Ubiquitous Interactor, the background, the system implementation, and the services. The third paper elaborates on the push feature of the Ubiquitous Interactor. It also describes the implementation of a stockbroker service that depends on information push in detail. The last paper contains a preliminary evaluation of the system.

Table of Contents

1	Introduction	1
1.1	<i>Devices and Services</i>	<i>1</i>
1.2	<i>Research Goal.....</i>	<i>2</i>
1.3	<i>Scope and Limitations.....</i>	<i>3</i>
2	Background	4
3	Related Work	7
4	System Design	9
4.1	<i>Interaction Acts.....</i>	<i>10</i>
4.2	<i>Presentation Control.....</i>	<i>12</i>
5	System Implementation.....	14
5.1	<i>Encoding Interaction Acts.....</i>	<i>14</i>
5.2	<i>Customization Forms.....</i>	<i>15</i>
5.3	<i>Interaction Engines.....</i>	<i>17</i>
5.4	<i>Services</i>	<i>18</i>
6	Results and Evaluation.....	21
7	Future work	24
8	Conclusion.....	25
9	Overview of the Thesis	26
10	References	28

Paper 1

Paper 2

Paper 3

Paper 4

Appendix A

Appendix B

Appendix C

Appendix D

1 Introduction

This thesis is addressing the problems with design and development that arise when service providers, and service end-users, face the vast range of computing devices available on the consumer market. The emergence of mobile and ubiquitous computing has brought new devices to the market and made way for new types of services. For service developers, this means more work with different versions of services for different devices with a large variety in presentation and interaction capabilities. For end-users, this means trouble combining services and devices since many services are available only for a small set of devices. To give users a true choice in combining services and devices, without multiplying development and maintenance work, services must be able to present themselves differently on various devices.

The Ubiquitous Interactor (Paper 2) offers a development method and user interface handling for services with multiple user interfaces. By separating interaction from presentation, multiple user interfaces can be created for a service without any changes in the service logic. We use *interaction acts* (Paper 2) to describe the user-service interaction in a device and modality independent way. The interaction acts are combined with *customization forms* (Paper 2) that contain presentation information for given services on given devices. Different user interfaces for different services are obtained by combining the interaction acts with different customization forms. In this way, services can provide tailored user interfaces to many devices.

1.1 Devices and Services

We have seen a tremendous change in computing, both in hardware and in usage, since the emergence of the personal computer in the eighties. Today, computers are much more than a box on the office desk. We see laptop and palmtop computers fully capable of connecting to the Internet both through cable and different wireless connections, and Java enabled cellular phones that can connect to the Internet through 2.5G or 3G technology. We have computers built into our cars, washing machines, and VCRs. Computers have been the main tool for office work for a long time, but they are also used for instant messaging, playing games, watching movies, and editing personal photographs. In their new forms and sizes they have left the office and the work setting and spread out into our environment and our leisure time. Devices like mobile phones

and handheld computers are personal and follow us everywhere. Gaming devices and robotic pets are entering our homes, all of them fundamentally changing the traditional view of computing. Users can choose their device not only based on which services they want to access and what tasks they want to perform, but also based on their context. This multitude of devices creates both new opportunities and new challenges. Users can access services in many different situations, if adequate user interfaces can be provided. To provide adequate user interfaces, services need to be able to adapt their presentations, since differences between devices are too great. No single user interface will be able to provide good user interaction for all devices, and no device will be appropriate in all contexts.

Along with the development on the device side, applications and services for computers have developed tremendously during the last fifteen years. Word processors and spreadsheets were among the earliest applications, and we can now add multimedia players and editors, advanced games, e-mail, and other messaging services, shopping services, and bank services to mention a few examples.

The wide range of devices gives end-users the opportunity to choose a device that suits their preferences and is designed for the intended use in terms of size, interaction capabilities, memory, and other features. In return, a large number of devices create problems for service providers. They are forced to deal with differences in processing power, memory capacity, and interaction capabilities along with different operating systems. To deliver content under such varying conditions, we need to find new ways of developing services for different devices that do not multiply development and maintenance work (Myers et al., 2000). It is not reasonable to force users to use different services for different devices to get the same content (Schneiderman, 2002).

1.2 Research Goal

The goal of this research has been to suggest a development method adapted to the current situation with many types of services accessed from a wide range of devices. Such a method needs to target multiple user interfaces from the start, and also allow service development for an open set of devices.

This work has had several subgoals. We believe that multiple user interfaces are best achieved by describing services in an abstract and

device independent way, and then create tailored user interfaces based on the abstract description in combination with presentation information. Thus, three subgoals were identified from the start:

- finding a suitable level of abstraction,
- identifying adequate units of description for services, and
- identifying adequate units of presentation information.

Finally, our intention was to validate the approach with a working prototype and sample services.

1.3 Scope and Limitations

The focus of this work has been to establish principles of device independent development. This has been done through identification of the concepts of interaction acts and customization forms, and the implementation of a prototype system with a few sample services as a proof of concept. This means that the design of the concepts and the prototype is made with a wide range of user interfaces, modalities and devices in mind. However, the prototype implementation is restricted to graphical user interfaces and Web user interfaces for desktop computers, handheld computers, and cellular phones.

The customization forms of the prototype system only cover mappings for service specific matters of the user interface types such as widget preferences. No mappings involving hardware features such as scroll wheels and hard buttons are handled yet.

The adaptations supported in the prototype only concern the relationship between service and device. No information about users' external environment, or user preferences is handled. However, this can be added to the prototype implementation without changes in the architecture.

2 Background

A large part of the research in human-computer interaction is (and should be) focused on users. Issues addressed are, among others, user needs and preferences, user tasks, cognitive aspects of computer use, and physical aspects or health issues. A smaller, but still important, part of human-computer interaction research is concerned with user interface development tools. Tools have a large impact on how user interfaces to commercial applications look and function, since virtually all commercial user interfaces are developed using some kind of tool (Myers et al., 2000). The use of well functioning tools normally reduces the amount of code developers need to write, and saves time in the development process. This time could instead be used for more iterations in the design process.

Today, computing is becoming truly mobile, with devices smaller than pocket size following their owners and providing access to e-mail, news and shopping. We are also getting closer to the vision of ubiquitous computing (Weiser, 1991), where objects in our environment have built-in computers, and connect to the Internet and to each other to serve their users better. Many objects around us, like cars, washing machines, telephones, and vending machines, have computers in them, even if they cannot communicate with each other yet. However, mobile and ubiquitous computing create new challenges, both for human-computer research and development tools. Human-computer interaction has evolved during its existence to a cross-disciplinary research field with a view of people using computers that includes, not only work tasks, hardware and user preferences, but also context, environment, and user feelings. A solid base of expertise has developed, and even if we have not reached perfection, much progress has been made. Mobile and ubiquitous computing brings new aspects into human-computer interaction. Small devices call for new types of services (Landay and Kaufmann, 1993), and new methods for presenting information (Ericsson et al., 2001), and new contexts for computing call for new ways of interaction (Guimbretière et al., 2001). New devices and combinations of devices, new applications and new usage patterns call for new types of interaction and new types of user interfaces. All this together call for new methods of development and evaluation, and new development tools. The Ubiquitous Interactor (Paper 2) is an attempt to provide new methods and new tools that promote good design for mobile and ubiquitous computing.

Our interest and need for device independent services originates from work with the next generation electronic services in the sView project, but the need for device independent applications is old. During the seventies and early eighties, developers faced large variations in hardware. That time the problem disappeared when the personal computer hardware got standardized to mouse, keyboard and desktop screen, and the development of direct manipulation user interfaces worked similarly in different operating systems (Myers et al., 2000).

Today, the situation is different. We are currently experiencing a paradigm shift from application-based personal computing to service-based ubiquitous computing. In a sense, both applications and services can be seen as sets of functions and abilities that are packaged as separate units (Espinoza, 2003). However, while applications are closely tied to individual devices, typically by a local installation procedure, services are only manifested locally on devices and made available when needed. The advance of Web-based services during the last decade can be seen as the first step in this development. Users were able to access functionality remotely from any Internet connected PC instead of accessing functionality locally on single personal computers. This will change though. With the multitude of different devices that we see today (e.g. smart phones, Personal Digital Assistants, and wearable computers) combined with growing requirements on mobility and ubiquity, the Web-based approach is no longer enough.

The sView system (Bylund and Espinoza, 2000, Bylund, 2001) provides an example of what the infrastructure for the next generation service-based computing could be like. In sView, each user has a personal service briefcase in which electronic services from different vendors can be stored. When accessing these services, users not only get a completely personalized usage experience, they can also benefit from the use of a wide variety of different devices, continuous usage of services while switching between different devices, and network independence (completely off-line use is possible).

The original sView system required that service developers create separate user interfaces for all devices or interface paradigms that a service was intended to be accessed from. A typical end-user service implemented four user interfaces: a traditional graphical user interface specified in Java Swing, an HTML and a Wireless Markup Language (WML) interface for remote access over HTTP, and an SMS interface for remote access from cellular phones. While the sView system provided

support for handling transport of user interface components, presentation, events etc, service providers still had to implement the actual user interfaces (Swing widgets, HTML/WML documents, and text messages) and interpret user actions (Java events, HTTP posts from HTML and WML forms, and text input).

In summary, this approach required huge implementation and maintenance efforts from the service providers. The standard solution to the problem was no longer viable however, and alternative solutions needed to be explored. The multitude of device types we see today is not only due to competition between vendors as before, but rather motivated by requirements of specialization. Devices look different because they are designed for different purposes. As a result, the solution this time needs to support simple implementation and maintenance of services without losing the uniqueness of each type of device. This is what we set out to solve with the Ubiquitous Interactor.

3 Related Work

Much of the inspiration for the Ubiquitous Interactor (UBI) comes from early attempts to achieve device independence or in other ways simplify development work. For a more comprehensive overview, see Paper 1.

MIKE (Olsen, 1987) and ITS (Wiecha et al., 1990) were among the first systems that made it possible to specify presentation information separately from the application, and thus change the presentation without changes in the application. Both are limited to graphical user interfaces, and introduce important restrictions on the interfaces they can generate. MIKE, for example, could not handle application specific data. In ITS, presentation information was considered to be application independent and stored in style files that could be moved between applications. As pointed out by Wiecha et al. this is not an adequate approach. In UBI, we instead consider presentation as both application specific and tailored to different devices (Paper 2).

During the eighties, the hardware for the personal computer was standardized, and the need for device independent applications and methods to develop them diminished. The problem has returned with mobile and ubiquitous computing, and the multitude of new computing devices. Again, service logic is separated from presentation to create device independent services. Standardization in hardware will appear in ubiquitous computing too, but that will not erase differences between devices. Devices are designed for different uses and those differences will persist.

XWeb (Olsen et al., 2000) and PUC (Nichols et al., 2002) encode the data sent between application and client in a device independent format using a small set of predefined data types, and leave the generation of user interfaces to the client. Unlike UBI, they do not provide any means for service providers to control the presentation of the user interfaces. It is completely up to the client how a service will be presented to end-users. In other words, these approaches enable device specific but not service specific presentations

The Web has often been presented as a way of achieving device independent applications. Most devices can run a Web browser and thus access any service with a Web user interface. The Web has some drawbacks though. It can only provide page-based, user-driven

interaction, which makes it less suitable for real-time applications (for example games). The device independence of Web pages can also be questioned. In many cases transformations or adaptations of pages are needed, for example to display a regular Web page on a handheld device with a smaller screen (Bickmore and Schilit, 1997, Trevor et al., 2001, Wobbrock et al., 2002, Menkhaus, 2002). To face these problems the World Wide Web Consortium created a working group addressing device independence for the Web. The group has not issued any recommendations yet, but two working group notes have been published, one outlining the vision of a device independent Web (Gimson, 2003), and one presenting the challenges of device independent authoring (Lewis, 2003). These documents are very general and present no solutions, which makes it too early to assess their impact on Web service development.

However, allowing separation of service logic and presentation is not enough for service providers. They also want to be able to control how their services are presented to the end-users (Myers et al., 2000, Esler et al., 1999). The user interface is the promoting channel for the provider, and it is important to be able to control that. Control of the presentation of user interfaces is provided in UBI, and also in the Unified User Interface system.

Unified User Interfaces (UII) (Stephanidis, 2001) is a design and engineering framework for adaptive user interfaces. In UII, user interfaces are described in a device independent way using categories defined by designers. Designers then map the description categories to different user interface elements. This means that designers have control of how the user interface will be presented to the end-user, but since different designers can use their own set of description categories the system cannot provide any default mappings. In UBI, we have chosen to work with a pre-defined set of description categories, along with the possibility for designers to create mappings. This makes it possible for the system to provide default mappings at the same time as designers can control the presentation of the user interface.

4 System Design

There are two main approaches to creating multiple user interfaces for services: creating a new version for each device, or using the same for all devices. Both have their drawbacks. Creating a new version of the user interface, and often of the service itself, generates a lot of development and maintenance work. The number of devices available on the market is already too large for service providers to keep up with, and will increase further. It would also be very cumbersome to keep consistency between the different versions. Using the same user interface for all devices means that the thinnest device will set the limits for the user interface, and unless the user interface is extremely simple, some device categories necessarily will be excluded. It will also be impossible to take advantage of device specific features such as shape, or interaction tools. We believe that neither of these methods offers service providers what they need to develop services for the new ubiquitous computing. Instead, we need to find ways to develop device independent services that can adapt to various devices. That way, a single implementation of a service can be tailored to present itself differently on different devices.

In the Ubiquitous Interactor (UBI), services are developed separately from the user interface. Services express their interaction in an abstract description that is interpreted by the device used to access the service. Service providers can also provide device specific presentation information for each service if they want to. At run-time, the device generates a suitable user interface based on the interpretation of the abstract description, and the optional presentation information. Different devices will generate different user interfaces from the same abstract description, and the same abstract description will generate different user interfaces when combined with different presentation information. Since the device itself generates the user interface, it is easy to use device specific features in the user interface.

For the abstract description of services, we have chosen the interaction between users and services as our level of abstraction in order to obtain units of description that are independent of device type, service type, and user interface type. Interaction is defined as *actions that services present to users, as well as performed user actions, described in a modality independent way*. This way we avoid focusing on how the interaction is done in particular user interfaces. Some examples of interaction

according to this definition would be: making a choice from a set of alternatives, presenting information to the user, or modify existing information. Pressing a button, or speaking a command would not be examples of interaction, since they are modality specific actions. By describing the user-service interaction this way, the interaction can be kept the same regardless of device used to access a service. It is also possible to create services for an open set of devices.

The interaction is expressed in *interaction acts* that are exchanged between services and devices. Throughout this work, we assume that most kinds of interaction can be expressed using a fairly limited set of interaction acts in different combinations. Based on this assumption, we have chosen to work with a small, fixed set of eight interaction acts. Interaction acts are interpreted on the device side and user interfaces are generated based on interaction acts and additional presentation information, see figure 1.

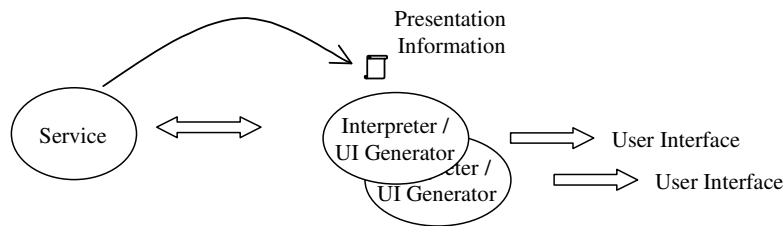


Figure 1: Services offer their interaction expressed in interaction acts, and an interpreter generates a user interface based on the interpretation. Different interpreters generate different user interfaces.

4.1 Interaction Acts

Interaction acts are abstract units of user-service interaction that contain no information about modality or presentation. This means that they are independent of devices, services, and interaction modality. User-service interaction for a wide range of services can be described by combining single interaction acts and groups of interaction acts.

The initial set of interaction acts had four members: `input`, `output`, `select`, and `confirm`. It was defined based on several sources of information. In basic human-computer interaction textbooks, making selections from a set of alternatives and presenting an object to the user

are often used as examples of user-service interaction (Dix, 1998, Preece et al., 1994). They were also included in the first set of interaction acts as `select` and `output`. Earlier research categorizations of interaction provided additional information, even though much of that work was limited to graphical user interfaces (Myers, 1990, Foley et al., 1984, Olsen, 1987, Mine, 1995), or virtual reality (Mine, 1995). We also looked at the user-service interaction in existing applications (Paper 4) before adding `input` and `confirm` as the last two members of the initial set.

To validate the set of interaction acts we created a sample service. We chose a calendar service, since that is a good example of a service that benefits from multiple user interfaces and access from different devices (see section 5.4.1). The initial set proved its functionality in the implementation of the service. All calendar operations were expressed using the four interaction acts. However, analysis of a new area of applications, computer games (Nylander and Waern, 2002), together with the character of the calendar suggested changes and additions to the set of interaction acts. The `input` category was extended to several interaction acts: `input`, `create`, `destroy`, and `modify`. `confirm` was included in `modify`. The analysis of games also suggested an interaction act handling position and movement, since that is fundamental in many high-end games. However, such an interaction act was never implemented due to difficulties finding a generic representation of positions.

The final set of interaction acts (see Paper 2 for details) that are supported in UBI has eight members: `input`, `output`, `select`, `modify`, `create`, `destroy`, `start`, and `stop`. `input` and `output` are defined from the system's point of view. `select` operates on a predefined set of alternatives. `create`, `destroy`, and `modify` handle the life cycle of service specific data, while `start` and `stop` handle the interaction session. All interaction acts except `output` return user actions to services. `output` only presents information that users cannot act upon. A new service, the TapBroker service (Paper 3), a feedback service for stock broking agents, was implemented without revealing new requirements on the set.

During user-service interaction, the system needs more information about the interaction acts than its type. Interaction acts are uniquely identifiable, so that user actions can be associated with them and interpreted by services. It is also possible to define for how long a user interface component based on an interaction act should be present in the user interface before being removed. Otherwise only static user interfaces can

be created. It is possible to create modal user interface components based on interaction acts, e.g. components that lock the user-service interaction until certain user actions are performed. This way, user actions can be sequenced when needed. All interaction acts also have a way to hold information, as a default base for the generation of user interface components. Finally, metadata can be attached to interaction acts. Metadata can for example contain domain information, or restrictions on user input that are important to the service.

To allow for association of presentation information with interaction acts and groups of interaction acts, both groups and individual interaction acts have symbolic names that are used in customization form mappings. Groups and individual acts can also be arranged in named presentation sets to allow for association of media resources to many interaction acts at a time.

In more complex user-service interaction, there might be a need to group several interaction acts together, because of their related function, or the fact that they need to be presented together. An example could be the create mail, reply, reply to all, and forward mail functions of a e-mail application. The structure obtained by the grouping can be used as input when generating the user interfaces. These groups allow nesting.

4.2 Presentation Control

It is not enough to use the same version of services for all devices and let the devices generate their user interfaces. Control of presentation has proven to be an important feature of methods for developing services (Esler et al., 1999, Myers et al., 2000), since it is used for e.g. branding. Service providers need means to control how user interfaces will be presented to end-users; otherwise they cannot promote their brands. If it was entirely up to the device, it would be cumbersome to differentiate similar services from different providers, for example two e-mail applications. To give service providers this possibility, services must be able to provide devices with detailed presentation information.

In UBI, presentation information is specified separately from user-service interaction. This allows for changes and updates to the presentation information without changing the service. The main forms of presentation information are *directives* and *resources*. Directives link interaction acts to for example widgets or templates of user interface components. Resources are used to provide pictures, sounds, or other media that are

used to present an interaction act in the user interface. Both directives and resources can be specified on three different levels: group level, type level or name level. Information on group level affects all interaction acts of a group, information at type level provides information for all interaction acts of the given type; and information on name level provides information about all interaction acts with the given symbolic name. The levels can also be combined, for example creating specifications for interaction acts in a given group of a given type, or in a given group with a given name.

It is optional to provide presentation information in UBI. If no presentation information, or only partial information is provided, user interfaces are generated with default settings. However, by providing detailed information service providers can fully control how their services will be presented.

5 System Implementation

The Ubiquitous Interactor (UBI) is composed of three parts: the Interaction Specification Language (ISL), customization forms, and interaction engines. The ISL is used to encode user-service interaction, and customization forms contain device and service specific presentation information. Interaction engines generate user interfaces based on interaction acts and customization forms. Different user interfaces will be generated from the same interaction acts, using different interaction engines or different customization forms. The three parts of the system are defined at different levels of specificity, where interaction acts are device and service independent, interaction engines are device dependent, and customization forms are service and device dependent, see figure 2.

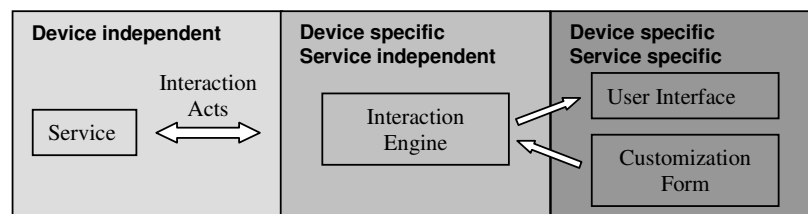


Figure 2: The three layers of specification in the Ubiquitous Interactor. Services and interaction acts are device independent, interaction engines are service independent and device or user interface specific, and customization forms and generated user interfaces are device and service specific.

The prototype has been iteratively designed to keep up with changes in the set of interaction acts, and new interaction engines and services have been developed during the work. The current version is number three. More details on the implementation can be found in Paper 2 and 3.

5.1 Encoding Interaction Acts

Interaction acts are encoded using the ISL, which is XML compliant. Each interaction act has a unique id that is used to map performed user interactions to it. It also has a life cycle value that specifies when components based on it are available in the user interface. The life cycle can be *temporary*, *confirmed*, or *persistent*. The default value is

persistent. Interaction acts also have a modality value that specifies if components based on them will lock other components in the user interface. The value of the modality can be true or false, where the default value is false. All interaction acts can be given a symbolic name, and belong to a named presentation group in a customization form. They also contain a holder for default information, and can have metadata attached to them. Listing 1 shows the ISL of an `output` interaction act.

```
<output>
  <id>235690</id>
  <name>sicsLogo</name>
  <group>calendar</group>
  <life>persistent</life>
  <modal>>false</modal>
  <string>SICS AB</string>
</output>
```

Listing 1: ISL encoding of an `output` interaction act with id, life cycle, modality, and default content information. It also has a symbolic name and belongs to a customization form group called *calendar*.

Interaction acts can be grouped using a designated tag `isl`, and groups can be nested to provide more complex user interfaces. These groups contain the same type of information assigned to single interaction acts.

The ISL code sent from services to interaction engines contains all information about the interaction acts: id, name, group, life cycle, modality, default information and metadata. A large part of this information is only useful for the interaction engine during generation of user interfaces. Thus, when users perform actions, only the relevant parts of interaction acts are sent back to the service. Two different grammars, or DTDs (Document Type Definitions) (Bray et al., 2000), have been created for the ISL, one for encoding interaction acts sent from services to interaction engines, and one for encoding interaction acts sent from interaction engines to services, see appendix A and B.

5.2 Customization Forms

Customization forms contain device and service specific information about how the user interface of a given service should be presented. They can be created in parallel with the service, or at a later point, for example when new devices are released on the market. Service providers can create customization forms to control the presentation of the user interface, but third party providers can also provide forms to support the needs and preferences of certain user groups.

Customization forms are specified in XML files according to a DTD created for this purpose, see appendix C. They are structured, and can be arranged in hierarchies. This allows for inheriting and overriding information between customization forms. A basic form can be used to provide a look and feel for a family of services, with different service specific forms adding or overriding parts of the basic specifications to create service specific user interfaces. A customization form does not need to contain information for all interaction acts of a service. Specified information takes precedence, but interaction acts with no presentation information specified in the form are presented with defaults.

ISL contains attributes for creating directives and resource mappings. Each interaction act or group of interaction acts has a symbolic name that is used in mappings where the name level is involved. This means that each interaction act with a certain name is presented using the information mapped to the name. Interaction acts or groups of interaction acts can also belong to a named group in a customization form. All interaction acts that belong to a group are presented using the information associated with the group (and possibly with additional information associated with their name or type).

Listing 2 shows an example of a directive mapping based on the type of the interaction act, in this case `output`, and listing 3 shows an example of a resource mapping for the interaction act in listing 1, based on its symbolic name.

```
<element name="output">
  <directive>
    <data>
      se.sics.ubi.swing.OutputLabel
    </data>
  </directive>
</element>
```

Listing 2: A mapping on type level for an `output` interaction act.

```
<id name="sicsLogo">
  <resource name="logotype" type="url">
    <data>
      <url>http://www.sics.se/logos/smallLogo.jpg</url>
    </data>
  </resource>
</id>
```

Listing 3: A resource mapping on name level, in the form of an URL to a picture.

5.3 Interaction Engines

Interaction engines interpret interaction acts and customization forms and generate user interfaces. At run-time, user interfaces are updated both on user actions and on system initiatives. Each interaction engine only generates one type of user interface for a given device or family of devices. Devices that can handle several types of user interfaces can have several interaction engines installed. Interaction engines also encode performed user actions as interaction acts and send them back to services. Examples of interaction engines are an engine for graphical user interfaces on desktop computers and an engine for Web user interfaces on handheld computers.

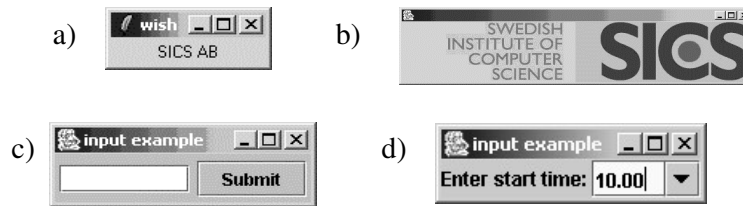


Figure 3: Rendering examples of an `output` and an `input` interaction act.

Interaction engines parse interaction acts sent by services, and generate user interfaces by creating presentations of each interaction act. Otherwise defaults are used. For example, an `output` could be rendered as a label, or speech generated from its default information, while an `input` could be rendered as a text field or a standard speech prompt. Figure 3 shows different presentations of an `input` and an `output` interaction act. Figure 3a shows the `output` interaction act in listing 1 rendered as a Tcl/Tk label displaying the default information of the interaction act. Figure 3b shows the same interaction act rendered as a picture specified as the resource mapping in listing 3. Figure 3c shows an `input` interaction act rendered as a Java Swing text field with a button to submit entered text, and figure 3d shows the same interaction act rendered as a Java Swing label and an editable combobox for choosing or entering time expressions.

We have implemented interaction engines for Java Swing, Java Awt, HTML, and Tcl/Tk user interfaces. All four interaction engines can generate user interfaces for desktop computers, but the default presentations of the Tcl/Tk and the Java Awt engine are designed to generate user interfaces for handheld computers and cellular phones

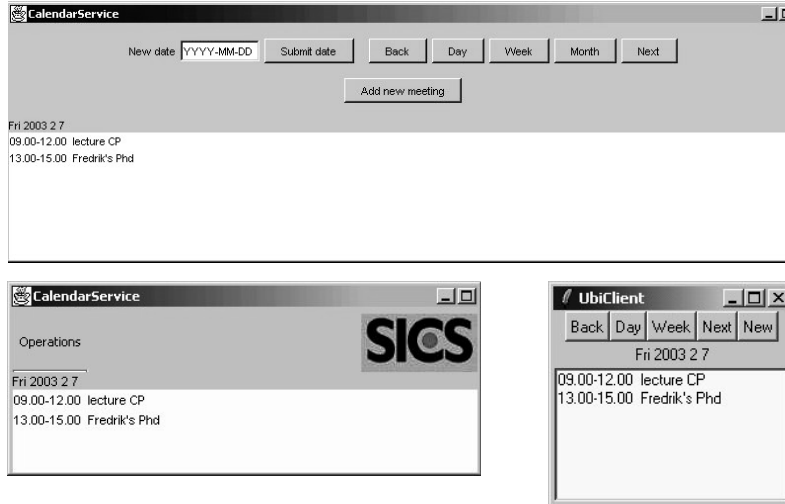


Figure 4: Three different user interfaces to the calendar service generated from the same interaction acts. The two to the left are generated by the Java Swing interaction engine using two different customization forms. The one to the right is generated by the Tcl/Tk interaction engine.

respectively. Each interaction engine is implemented as a service in the sView system (Bylund, 2001), to take advantage of the user interface handling capabilities of sView.

5.4 Services

We have developed two different services to serve as proof of concept in the Ubiquitous Interactor, a calendar service (Paper 2) and a stockbroker service (Paper 3). Both services are implemented as sView services (Bylund, 2001).

5.4.1 Calendar Service

The calendar provides an example of a service that it is useful to access from different devices. Calendar data may often be entered from a desktop computer at work or at home, but mobile access is needed to consult the information on the way to a meeting or in the car on the way home. Sometimes appointments are set up out of office (in meeting rooms or restaurants) and in those cases it is practical to use mobile devices to enter information into the calendar.

The calendar service supports basic operations as enter, edit and delete information, navigate the information, and display different views of it. The service is accessible from three types of user interfaces: Java Swing and HTML user interfaces for desktop computers, and Tcl/Tk user interfaces for handheld computers. Two different customization forms have been created for Java Swing, and one each for Tcl/Tk and HTML. These four forms generate different user interfaces from the same interaction acts, see figure 4 for examples.

5.4.2 Stockbroker Service

TapBroker (Paper 3) is a notification service for agents trading stocks on an Agent Trade Server (Boman and Sandin, 2003) on the behalf of users. Each user can have one or more agents, and TapBroker provides feedback on how agents perform so that users know when to change agent or shut them down. TapBroker also gives some context information to help users understand and assess the behavior of agents. Agents work autonomously on the server, and thus cannot be controlled through TapBroker due to security reasons (Lybäck and Boman, 2003).

The TapBroker service gives feedback on the agent's actions: order handling and performed transactions. It also provides information on the account state (the amount of money it can invest), status (running or paused), activity level (number of transactions per hour), portfolio content, and the current value of the portfolio.

We have implemented customization forms for Java Swing, HTML and Java Awt, see figure 5. Figure 5a and 5c is generated by the same interaction engine from the same interaction acts but with different customization forms. Figure 5b is generated by another engine with a third customization form.

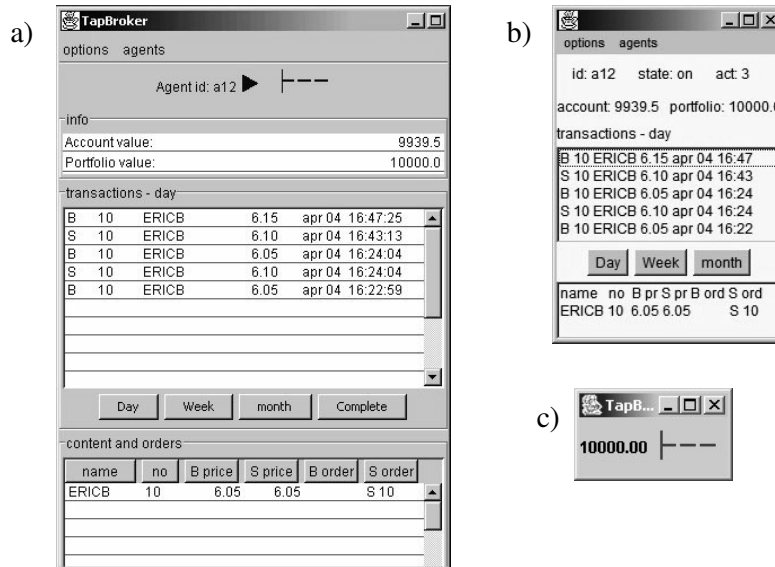


Figure 5: User interfaces for the TapBroker: Picture a shows a Java Swing user interface for desktop computers and picture b shows a Java Awt user interface for mobile phones. Picture c shows a Java Swing user interface for very small devices.

6 Results and Evaluation

The results from the Ubiquitous Interactor (UBI) project comprise both a development method for services with multiple users, and several versions of a working prototype serving as a proof-of-concept. Below the main findings from the evaluation process is presented, along with some ideas on how to further evaluate the system. More details can be found in Paper 4.

The development part of this work together with the resulting prototype is a proof-of-concept. We have identified a problem, the difficulty for users and service providers to handle and combine the multitude of devices and services, and proposed a way to solve it by separating function from presentation and providing developers with new units of description. However, this kind of solution needs to fulfill two kinds of requirements: “hard” requirements concerning the implementation of the solution, and “soft” requirements concerning the support the implemented system actually gives the developers. The work needed to fulfill the two kinds of requirements is quite different. The “hard” requirements need engineering work to show the feasibility of the solution, and in some cases also the efficiency. The “soft” requirements need study of the use of the system, how useful it is, and how it is gaining acceptance outside its development settings.

We believe that the prototype of UBI together with the two sample services show that the “hard” requirements of our proposed solution are fulfilled. It has been possible to create two different services using the set of interaction acts, and to create different user interfaces to them using customization forms with mappings and resources. This shows that our approach of separating function from presentation is feasible. The prototype handles all interaction acts, the information in customization forms, and different types of user interfaces. It also handles the different characteristics of the sample services. Even though the calendar and the TapBroker are both information-based services, they differ on some points. The calendar is user-driven; all functionality and updating of the user interface is driven by user actions. The TapBroker is mainly service driven: TapBroker subscribes to information from the Agent Trade Server and updates the user interface each time an agent makes a transaction. The two services have different information sources. The calendar relies on users entering information, while TapBroker collects

data from external sources. The calendar is thus the more interactive of the two services. Calendar users can enter, edit, delete and browse information. TapBroker users can add and remove agents but not perform any operations on the information content.

It is much more difficult to evaluate if the “soft” requirements are fulfilled, mostly because they involve human users and depend on a wide use of the system. In this aspect we can compare UBI with the sView project (Bylund, 2001, Bylund and Espinoza, 2000) at SICS, which aimed at a new, user-centered usage of electronic services. We can also compare it to larger projects as the development of a programming language, or even the emergence of the Web and HTML. All of these projects depend on acceptance from a large community of users outside their original inventors to gain their full power and show their utility. Before that has happened, it is very difficult to evaluate their contribution to the development process.

The assessment of how UBI fulfills the “soft” requirements would demand that we let many developers and service providers create a large number of services with multiple user interfaces and observed and interviewed them. This has not been feasible within the scope of this work. However, we plan to make a study of developers and have conducted a pilot study to inform the study design (Paper 4). Moreover, it is not possible to prove that this kind of requirements are fulfilled, it is only possible to successively gather more evidence that supports that they are fulfilled.

Some small steps to evaluate the “soft” requirements have been taken. We have conducted a pilot study on customization form development for existing services in UBI, as a preparation for a larger study on developers experience working with interaction acts and customization forms. The pilot study comprised four students that worked in pairs on creating a customization form for the TapBroker service. The study showed that the participants had no problem understanding the concepts and over all function of UBI, and gave some valuable information on how a larger study should be conducted, for more details see Paper 4. The interaction acts and the Interaction Specification Language are also used in the Clarity research project (<http://clarity.shef.ac.uk>) to generate user interfaces for information retrieval systems. To get more reliable indications on how developers feel about working with UBI, we need to conduct larger studies.

An area that has not been evaluated yet is the quality of the user interfaces produced with UBI. The design of the TapBroker user interfaces were based on interviews with a small number of users and discussions with human-computer interaction researchers, but no user study has been performed.

7 Future work

In the current version of the Ubiquitous Interactor (UBI) we have shown that it is possible to create multiple user interfaces to a service using interaction acts and customization forms. However, we will extend the system to demonstrate that the UBI approach can handle mappings to device hardware features like buttons or scroll wheels. It is also important to show that UBI can handle other user interface types than graphical user interfaces and Web user interfaces. We plan to investigate this through realizing an interaction engine for speech user interfaces.

The use of default presentations needs to be investigated. The ultimate goal would be to compose a toolkit of presentations for each interaction act that designers can choose from, and not only a single default presentation. The toolkit could also provide possibilities to configure the presentations in the toolkit using the customization forms.

The current model for resource management in UBI does not support dynamic resources. Services that use lots of dynamic media resources, e.g. a service for browsing a video database, might need an extension of our customization form approach to work efficiently for different modalities. An alternative solution would be to place the choice of media type in the interaction act and not in the customization form.

Developers' experiences in working with UBI need to be further studied. The pilot study indicated that it is quite easy for developers that have not previously worked with UBI to develop customization forms for existing services, but the use of UBI in original service development remain to be studied. The students that participated in the pilot study had no problem understanding the concepts of the system, but experienced trouble with the user interface programming and understanding the functions of the service. Further studies are needed on how to present existing UBI services to developers creating customization forms.

It is important to evaluate the quality of the generated user interfaces, and investigate if possible problems originate from the nature of interaction acts or customization forms, or other aspects of UBI.

8 Conclusion

The purpose of this work was to suggest a method for creating services with multiple user interfaces for different devices. To achieve this, we have established the concepts of interaction acts and customization forms. Interaction acts describe the user-service interaction in a device and modality independent way, and customization forms contain presentation information for a given service on a given device. Combining the same set of interaction acts with different customization forms generates different user interfaces for a service.

We have built a prototype, the Ubiquitous Interactor, which generates user interfaces based on interaction acts and customization forms, and two sample services with three customization forms each. We believe that the different user interfaces of the sample services serve as proof-of-concept. It is possible to develop services with multiple user interfaces using interaction acts and customization forms.

However, to confirm the utility of the concepts and the system it is crucial to continue the evaluation of how developers make use of them. We need to continue the work on service development to show that UBI can handle other services than those already developed. It is also important to add support for non-graphical user interfaces such as speech user interfaces. Before this is done, it will be difficult to make a correct assessment of UBI's capabilities.

We need to study developers working with UBI and get their feedback to improve the system. The pilot study we conducted gave some indications on how we should proceed, but it only concerned customization form development. Service development need to be studied to evaluate the concepts of interaction acts and customization forms correctly. The quality of the generated user interfaces also need to be evaluated.

9 Overview of the Thesis

Paper 1: Different Approaches to Achieving Device Independent Services – an Overview.

Author: Stina Nylander.

Description: This paper gives a thorough overview of previous work in the area of developing device independent applications and services and their relation to the Ubiquitous Interactor. Early work in user interface development is described, as well as recent approaches in a ubiquitous computing setting.

My contribution: This work was produced by the author alone.

Publication status: Published as a technical report, Swedish Institute of Computer Science, report number T2003:16

Paper 2: The Ubiquitous Interactor – Mobile Services with Multiple User Interfaces.

Authors: Stina Nylander, Markus Bylund and Annika Waern.

Description: This paper describes the design and the implementation of the Ubiquitous Interactor. The main concepts of the system, interaction acts, customization forms and interaction engines are presented along with a detailed description of how they are implemented. Two sample services for UBI are also described, a calendar service and a stockbroker service.

My contribution: The design work presented in this paper is a joint work between the co-authors. Stina Nylander has made most of the implementation work and the writing of the paper.

Publication status: Published as a technical report, Swedish Institute of Computer Science, report number T2003:17. A shorter version has been submitted to the conference Computer-Aided Design of User Interfaces.

Paper 3: Mobile Access to Real-Time Information – The Case of Autonomous Stock Brokering.

Authors: Stina Nylander, Markus Bylund and Magnus Boman.

Description: This paper focuses on one aspect of the Ubiquitous Interactor: the ability to provide information push. The system features that make this possible are described, and the stockbroker service is presented in detail as an example of a service that depend on push of real-time information.

My contribution: Stina Nylander has made the design and implementation work with the TapBroker service, and most of the writing of this paper.

Publication status: Accepted for publication in the journal Personal and Ubiquitous Computing, Springer Verlag.

Paper 4: Evaluating the Ubiquitous Interactor

Author: Stina Nylander.

Description: In this paper the work with the Ubiquitous Interactor is evaluated, both in terms of the quality of the results and in terms of how well the research goals are achieved. A pilot study is presented and some pointers on how the evaluation will proceed are also given.

My contribution: This work was produced by the author alone.

Publication status: Published as a technical report, Swedish Institute of Computer Science, report number T2003:19

10 References

Bickmore, T. W. and Schilit, B. N. (1997) *Digestor: Device-independent Access to the World Wide Web*, in Proceedings of 6th International World Wide Web Conference.

Boman, M. and Sandin, A. (2003) *Implementing an Agent Trade Server*, Available at arxiv.org/abs/cs.CE/0307064.

Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. (2000) *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, World Wide Web Consortium, <http://www.w3.org/TR/REC-xml>.

Bylund, M. (2001) *Personal Service Environments - Openness and User Control in User-Service Interaction*, Licentiate thesis, Department of Information Technology, Uppsala University.

Bylund, M. and Espinoza, F. (2000) *sView - Personal Service Interaction*, in Proceedings of 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology.

Dix, A. (1998) *Human-Computer Interaction*, Prentice Hall.

Ericsson, T., Chincholle, D. and Goldstein, M. (2001) *Both the Cellular Phone and the Service Impact WAP Usability*, in Proceedings of IHM-HCI 2001, Lille, France.

Esler, M., Hightower, J., Anderson, T. and Borriello, G. (1999) *Next Century Challenges: Data-Centric Networking for Invisible Computing*. The Portolano Project at the University of Washington, in Proceedings of The Fifth ACM International Conference on Mobile Computing and Networking, MobiCom 1999.

Espinoza, F. (2003) *Individual Service Provisioning*, PhD, Department of Computer and Systems Science, Stockholm University/Royal Institute of Technology.

Foley, J. D., Wallace, V. L. and Chan, P. (1984) The Human Factors of Computer Graphics Interaction Techniques, *IEEE Computer Graphics and Applications*, **4**, (6), 13-48.

Gimson, R. (2003) *Device Independence Principles*, W3C Working Group Note, World Wide Web Consortium, <http://www.w3.org/TR/2001/WD-di-princ-20010918/>.

Guimbretière, F., Stone, M. and Winograd, T. (2001) Fluid Interaction with High-resolution Wall-size Displays, in Proceedings of 14th Annual Symposium on User Interface Software and Technology, Florida, Orlando, 21-30.

Landay, J. A. and Kaufmann, T. R. (1993) User Interface Issues in Mobile Computing, in Proceedings of 4th Workshop on Workstation Operating Systems, Napa, CA.

Lewis, R. (2003) *Authoring Challenges for Device Independence*, W3C Working Group Note, World Wide Web Consortium.

Lybäck, D. and Boman, M. (2003) Agent trade servers in financial exchange systems, *ACM Transactions on Internet Technology*, (In press.).

Menkhaus, G. (2002) *Adaptive User Interface Generation in a Mobile Computing Environment*, PhD thesis, University of Salzburg.

Mine, M. R. (1995) *Virtual Environment Interaction Techniques*, Computer Science Technical Report, TR95-018, UNC Chapel Hill, http://www.cs.unc.edu/~mine/mine_publications.html.

Myers, B. A. (1990) A New Model for Handling Input, *ACM Transactions on Information Systems*, **8**, (3), 289-320.

Myers, B. A., Hudson, S. E. and Pausch, R. (2000) Past, Present and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **7**, (1), 3-28.

Nichols, J., Myers, B. A., Higgings, M., Hughes, J., Harris, T. K., Rosenfeld, R. and Pignol, M. (2002) Generating Remote Control Interfaces for Complex Appliances, in Proceedings of 15th Annual ACM Symposium on User Interface Software and Technology, Paris, France, 161-170.

- Nylander, S. and Waern, A. (2002) *Interaction Acts for Device Independent Gaming*, Technical report, T2002-04, Swedish Institute of Computer Science.
- Olsen, D. J. (1987) MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics*, **5**, (4), 318-344.
- Olsen, D. J., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P. (2000) Cross-modal Interaction using XWeb, in Proceedings of Symposium on User Interface Software and Technology, UIST 2000, 191-200.
- Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. and Carey, T. (1994) *Human-Computer Interaction*, Addison-Wesley.
- Schneiderman, B. (2002) *Leonardo's Laptop*, MIT Press.
- Stephanidis, C. (2001) The Concept of Unified User Interfaces, In *User Interfaces for All - Concepts, Methods, and Tools* (Ed, Stephanidis, C.) Lawrence Erlbaum Associates, pp. 371-388.
- Trevor, J., Hilbert, D. M., Schilit, B. N. and Khiau Koh, T. (2001) From Desktop to Phonetop: A UI for Web Interaction on Very Small Devices, in Proceedings of 14th Annual ACM Symposium on User Interface Software and Technology, Orlando, FL, 121-130.
- Weiser, M. (1991) The Computer for the 21st Century, *Scientific American*, (September 1991).
- Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. (1990) ITS: a Tool for Rapidly Developing Interactive Applications, *ACM Transactions on Information Systems*, **8**, (3), 204-236.
- Wobbrock, J., Forlizzi, J., Hudson, S. and Myers, B. A. (2002) WebThumb: Interaction Techniques for Small-Screen Browsers, in Proceedings of 15th Annual Symposium on User Interface Software and Technology, Paris, France, 205-208.

Paper 1

Different Approaches to Achieving Device Independent Services – an Overview

Stina Nylander

Different Approaches to Achieving Device Independent Services – an Overview

Stina Nylander
Swedish Institute of Computer Science
SICS Technical Report T2003:16
E-mail: stina.nylander@sics.se

Abstract

This report provides an overview of different approaches to device independent development of applications and a background to why it is important for mobile computing. It also describes the different sources of inspiration for the work with the Ubiquitous Interactor.

1 Introduction

The purpose of this paper is to provide a thorough overview of the past research in how to develop device independent applications. Many different approaches have been proposed through the years, with different purposes and motivations, but no solution has been widely accepted yet. The most successful and widely spread approach so far is the Web, even if it has problems adapting to small devices and suffers from a limited interaction model (see below).

The systems described below have also served as foundation and background for the research on device independent mobile services that resulted in interaction acts, and the Ubiquitous Interactor system (UBI) (Paper 2). Designing UBI, we have strived to create a straightforward, predictable and controllable system that allows the creation of services with device specific user interfaces. In this work, the systems that provide designers with abstract description units for describing the application have inspired us, in particular those who use user-service interaction as level of abstraction.

1.1 Problem description

The need to make applications and services available from many different devices is not new. Before the emergence of the personal computer, software engineers faced a wide range of hardware standards. Computers were more or less custom made; used different input and output devices, and applications needed to be written in different programming languages to run on different systems. To reduce development work, and increase the amount of available software, efforts to obtain device independent applications were made. That time, the problems were practically solved with the introduction of the personal computer, where the hardware got standardized and the development of desktop user interfaces worked similarly for many operating systems (Myers et al., 2000).

Today, we face the same problem of variations in hardware, but for fundamentally different reasons. While the differences in hardware during the seventies mostly were due to the fact that new technology was developed and not yet standardized, many of the differences we now see in screen-size, keyboard-size or other interaction capabilities are due to design decisions. The design of today's devices may not be flawless, but the differences in size and presentation or interaction capabilities reflect the intended use of the devices: mobile use, use in public spaces, office use, leisure use. Even if design progress is made, those differences will persist, and no single device will be able to cover all different uses. Thus it is unlikely that standardization of hardware will eliminate the need for device independent services.

There is already a wide range of devices available on the market. To face this development on the device side, service providers need to tailor the user interfaces of services. Desktop computers, PDAs and cellular phones cannot use the same user interface to a service, they are simply too different. To provide good user interaction on different devices, services need to be able to provide user interfaces that are adapted to the device that is used for access. This is often done by creating different versions of services, which is costly in terms of development and maintenance work. To be able to provide the wide range of devices with suitable applications, we need to find robust methods for development and maintenance that allow applications to be easily tailored for different devices (Myers et al., 2000).

1.2 Outline

The rest of this report is organized as follows: In section 2, the use of abstraction in development is described, with sets of description units discussed in section 2.1, and models discussed in section 2.2. Section 3 describes systems using different sets of description units, and section 4 describes model-based systems. Finally, some closing comments are given in section 5.

2 The Use of Abstraction

The computer science community has strived for a higher level of abstraction for a long time. Operating systems, toolkits and developer tools handle low-level details of applications and hardware, and make it simpler for developers to create both back-end applications and user interfaces. This way of thinking can also help us to achieve device independent applications. Abstractions hide differences between devices to systems, and to developers. Developers do not need to keep device specific details in mind, or create different versions for each device, and systems do not need to handle different versions of applications. This way, less time and effort is needed for development and coding, and more time can be used for design and user interface improvement. All functionality that is common for the targeted range of devices can be handled once, and only device specifics need to be handled for each device. The ultimate goal is of course to create better services and better user interfaces for end-users.

To achieve device independence through the use of abstractions, a suitable level of abstraction, and useful units of description need to be identified. The level of abstraction can be, for example, the user interface level with user interface components as units of description, or the user interaction level with user actions as units of description. It is also possible to use models as descriptions of applications, for example task models or device models. The targeted range of modalities, devices and applications highly affects the choice of abstraction level.

Abstractions must be made concrete to create user interfaces for applications. Units of description need to be mapped to user interface components that fit the target device. This mapping can be made explicitly by developers, or defaults can be specified for the system to use. The Ubiquitous Interactor uses a combination of these two. It is also possible to leave the choice to the system by encapsulating knowledge on

user interface creation in the system. This is usually made through a set of rules that the system uses to interpret the description of the application. In some systems developers are offered mapping suggestions from the system and have the power of choosing and overriding the system (Eisenstein and Puerta, 2000a). In the choice of leaving mapping to developers, system or both parties, control is often traded for efficiency. When designers make the mappings they have a good control over what user interfaces will be produced. When the system handles the mapping, the user interface creation is much faster, but usually less predictable. A problem for model-based user interface generation has been that the generated user interfaces were unpredictable (Myers et al., 2000).

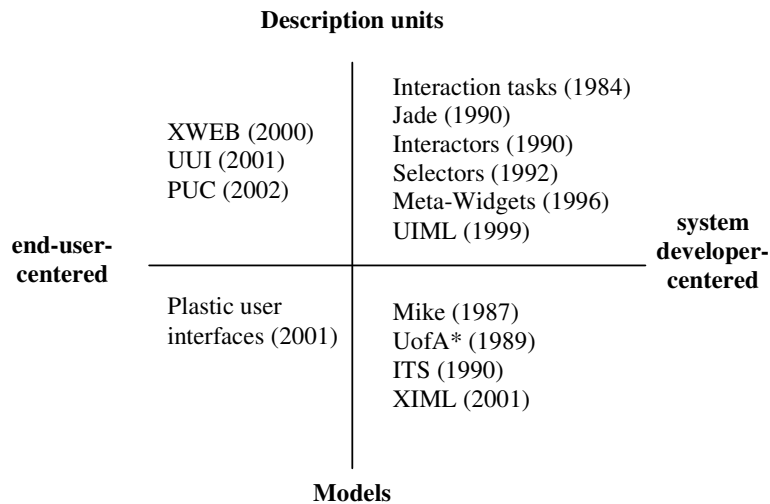


Figure 1: The systems discussed in sections 3 and 4 displayed based on them providing sets of abstract description units or models, and focusing on end-users or system developers.

The systems that inspired the work with UBI the most are those that provide designers and developers with a set of abstract description units to describe the applications, or allow designers to create the set. Even if all the projects did not use their description units to generate user interfaces they provided inspiration in the establishment of interaction acts for UBI. How the systems that do generate user interfaces map the description units to concrete user interface representations is not important here. Some of them use knowledge-based methods, while some let the designers do the mapping. A more detailed description of the

different systems will be given in section 3.1. The model-based approaches have not been an important inspiration but they will still be treated below in section 3.4. Figure 1 shows the systems in section 3 and 4 grouped according to their primary target group and their way of describing applications.

2.1 Fixed or open set of description units

The underlying assumption in systems based on a fixed set of description units is that there are similarities between all applications that can be used to facilitate development. If these similarities can be correctly identified, they provide the basis for a set of description units that characterizes applications in a meaningful way, and thus provides support for developers in designing user interfaces. A predefined fixed set helps keeping the design independent of devices and applications, since developers are not tempted to construct new categories that are influenced by a certain type of user interface or application. Instead, they will work with the same set for all applications and accumulate skills in how to use the abstractions. It also makes it possible to reuse templates between applications and over families of devices, and to provide default mappings that can be used when nothing in particular is specified about the presentation of a unit. However, it might be very difficult to identify a set of abstractions that is truly independent of devices, interaction styles, and application types, and still specific enough to support the development process.

Using an open set of abstractions builds on the assumption that the functions of the application should determine what description units to use. Applications with similar functions can share abstractions. This allows more room for adapting to applications since the set of abstractions can be partially or completely redefined for each application. The abstractions might also be easier to identify since functions of the application give guidance. In return, an open set of abstractions gives less support for application and interaction style independence in the design process since it allows application specific abstractions. The more application specific abstractions get, the more difficult it will be to reuse them in other applications. An open set of abstractions might allow for easier design of single device independent application, as long as the identification of the abstractions does not take too much time. However, it might be difficult to design a large group of applications. If the applications differ significantly, designers will not accumulate skills and lots of extra work is needed since new abstractions must be defined for

each application. It will also be difficult to reuse created mappings and realisations of abstractions.

2.2 Models

Many of the systems that do not use a set of abstract description units use one or several models instead. The system then uses rules or constraints to interpret the model and generate a user interface. If the models are general enough, and for example device information is stored separately, different user interfaces for the same application can be generated by changing the device information.

In model-based user interface development, developers would write user interface specifications in a high-level specification language. The description is divided into two parts, an *abstract UI specification* that contains abstract interaction objects (AIOs) and information that is not specific for the presentation, and a *concrete UI specification* that contains *concrete interaction objects* (CIOs) and information about the rendering style of the abstract objects (Szekely, 1996). The AIOs are grouped into *presentation units*, i.e. windows. The specification would be translated into an executable program, or interpreted at run-time. The models used are generally a task model and a domain model, even though some systems only use one of the two.

Early knowledge-based systems did not separate models and device information sufficiently well to be able to create different user interfaces for different devices. They also had problems with too specific systems that could not be used for different application domains (Eisenstein and Puerta, 2000a).

Model-based development of user interfaces has had problems catching on. This is due to the fact that models are complicated to work with and tend to produce unpredictable user interfaces. In more recent model-based approaches the developer has been brought into the loop to prevent surprises from the system. Adaptive tools have been created, that instead of generating the user interface presents alternatives for the developer to choose from (Eisenstein and Puerta, 2000b). The tools also monitor the choices and adapt its future suggestions. With these modifications, some researchers believe that model-based techniques offer promising means to create services for many different mobile devices (Eisenstein et al., 2001, Szekely, 1996).

3 Sets of Description Units

In this section, examples of systems using both fixed and open sets of description units are presented. Even if many of them did not have the creation of device independent applications as a goal, their ways of describing applications have provided inspiring examples in the work with the Ubiquitous Interactor.

3.1 Fixed set of abstractions

3.1.1 Interaction tasks

In 1984, Foley et al. (Foley et al., 1984) defined a set of *interaction tasks* and a set of *control tasks*. The two sets of tasks were intended to guide designers in assigning appropriate interaction devices to graphical user interfaces, e.g. mouse, light pen, keyboard. An interaction task is a primitive action unit performed by the user, and is associated with a set of example interaction techniques. The six interaction tasks are *select*, *position*, *orient*, *path*, *quantify*, and *text*. An interaction task does not modify the object displayed on the screen, that is done using control tasks. The control tasks are *stretch*, *sketch*, *manipulate*, and *shape*.

Both the categories of interaction tasks and control tasks clearly reflect the visual perspective. The orient and path tasks are not very relevant in non-visual user interfaces, and of the control tasks stretch, sketch and shape are exclusively visual. What is important though, is the foundation of the categories. They are based on what the user is doing at a higher level than the application (even if the quantify task could be classified as a task that is only relevant in a small set of applications), and partly at a higher level than the interaction mode, much like the interaction acts in the Ubiquitous Interactor (Paper 2). It must be taken into account that Foley et al. were not targeting different modalities; they wanted to create a support that would help designers of graphical user interfaces to choose interaction techniques.

3.1.2 Interactors

Myers' *interactors* (Myers, 1990) from 1990 were an effort to standardize user input to applications on a higher level. This way, developers would get device independent user input, and would not need to treat user input from various input devices. Interactors did not handle output and was restricted to mouse and keyboard input in graphic user interfaces.

The interactor categories are based on the interaction tasks of Foley et al. described above, and are *Menu-Interactor* (select), *Move-Grow-Interactor* (position), *New-Point-Interactor* (position), *Angle-Interactor* (orient), *Text-Interactor* (text), and *Trace-Interactor* (path). The graphical perspective is clearly reflected in the categories; the angle and trace interactor as they are defined by Myers would not be useful in a non-graphical user interface. The text interactor would also need to be generalized to fit in for example speech user interfaces; for Myers it takes care of all input that is not made with the mouse. If the scope of the system would be widened to other modalities than graphic direct manipulation user interfaces, that interactor would need to handle not only text, but also for example speech input and input from numeric keypads.

3.1.3 Jade

Jade (Vander Zanden and Myers, 1990) is a development tool from 1990 for graphical user interfaces where the look and feel could be changed easily. Dialogs are generated from two information sources: a textual specification of the dialog contents and separate style information. Content specifications are intended to be written by application developers, and style information is intended to be created by graphic artists. Since style information is separate from content definition, the look and feel of the user interface can be changed without any change in the application. However, the styles in Jade are defined as a single entity. You can exchange one look and feel for another, but not take some parts from one and some parts from another.

The application programmer can use a set of seven different *interaction techniques* to specify the behavior of dialogs or parts of dialogs: *single-choice*, *multiple-choice*, *single-choice-with-text*, *multiple-choice-with-text*, *command* (menu choice) and *number-in-a-range*. These interaction techniques are then used in combination with style information to generate dialogs. If no style information is associated with an interaction technique, defaults are used. Moreover, you cannot specify style rules for individual instances of an interaction technique. All instances of an interaction technique category within a look and feel will be presented in the same way.

The classification of the interaction techniques is based on graphical user interfaces, and it is not modality independent. For example, the distinction between single-choice and single-choice-with text would not be meaningful in a text or speech based user interface. It is possible that

this distinction reflects the possibility to make commands both with text, using the keyboard, and with the mouse in graphic user interfaces. Number-in-a-range might also only be useful for a small range of applications.

3.1.4 Selectors

Johnson (Johnson, 1992) established a classification of interactive controls in the ACEKit based on application semantics rather than on appearance. The purpose was to go one step further than Jade, and not only provide possibilities to change the look and feel of an application but also possibilities to specify the presentation for individual elements of the user interface. This corresponds to the way individual interaction acts can be mapped to different presentations in the Ubiquitous Interactor.

The semantic base of the controls results in units of the user interface that “knows” what kind of data they will receive and display, and what operations the user can perform on it. Thus, designers only need to define and label the data for the application, and not again for the user interface. The classification has two sets of *selectors*, data selectors and command selectors, where data selectors allow users to set application variables, and command selectors to invoke actions. Data selectors are divided into two groups: basic semantic objects and choice semantic objects. Basic objects represent data values, numbers, colors, dollar amounts, etc., and choice objects represent different kinds of choices from a set, single-choice, multiple-choice, etc. Collections of presentations to basic and choice objects are then provided. Command selectors are also divided into two groups, operations and commands. Operations take arguments and affect the applications data-state, while commands gather arguments for operations and invoke them. The ACEKit enforces the distinction between data selectors and command selectors by providing different presentations for them.

Having only two categories of data selectors, data types and choices, might be too restraint for many types of applications. For configuration applications, or other types of applications where the interaction mostly is composed of users entering values for different entities, it might be suitable. However, when it comes to informative applications that mostly display information to the user, e.g. document or photo browsers, word processing, or communication applications, a more elaborate set is needed. Selectors also only deal with graphical user interfaces.

3.1.5 XWeb

The XWeb project (Olsen et al., 2000) from 2000 has been inspired by the Web and Web browsers. Its purpose is to provide different user interfaces to applications so that users can choose the modality or the type of user interface they prefer.

Data that is sent between services and user interfaces are encoded in a general way, and client side software generates a user interface. Measures have been taken in the XWeb architecture to provide more interactivity than the traditional, user-driven, page-based Web user interfaces. Users can choose a client that interprets data and presents a user interface in a modality that they prefer. This makes it possible for the user to have the same type of user interface to many services, thus reducing learning and memory load. XWeb clients for desktop computer, pen-based wall display, and speech have been developed.

XWeb uses a set of eight *interactors*, divided into two groups, atomic and aggregated, based on the type of information that they contain. The atomic interactors are *numbers*, *dates*, *times*, *enumeration of finite choices*, *text*, and *links*. The aggregate interactors are *groups* and *lists*. Interactors are units of information that do not dictate how it should be presented, or what kind of interaction technique should be used. Different resources for presentation can be associated with the interactor, and the client can then choose the appropriate resources for presentation.

A drawback of XWeb is that it leaves the whole process of generating user interfaces to the clients and provides no means for service providers to control how user interfaces are presented to end-users. This also means that they cannot take advantage of device specific features in the generated user interfaces. Associating resources for presentation with the individual interactor also mean that if changes for example in the look and feel need to be done, every interactor needs to be modified.

3.1.6 Personal Universal Controller

The Personal Universal Controller (PUC) (Nichols et al., 2002) is an attempt to provide simpler user interfaces to home appliances. Users have their personal computing device that they carry with them, for example a PDA, which can present user interfaces to appliances surrounding them, VCRs, stereos, microwave ovens etc. With this approach, users can interact with all appliances surrounding them using the same device. The PUC downloads a description of the appliance's functions, creates a PUC

specification and generates a GUI or a speech user interface without intervention from the user.

The PUC specification consists of state variables that are grouped together based on similarity. The state variables are then complemented with labels that are used in rendering, and dependency information that are used to control which parts of the description that will be rendered together and which will not. There are seven different state variables: *boolean*, *integer*, *fixed point*, *floating point*, *enumerated*, and *custom*. There is also an additional type, *command*, that is used for interface elements that cannot be represented as state variables.

PUC is designed to create user interfaces automatically for a family of applications, home appliances. This means that no means for tailoring the user interface are provided. All appliances with the same functions will get the same user interface, and it is not possible to create differences between brands or different user groups. It might even be difficult to distinguish for example several different radios in the same house.

3.1.7 Alternate Interface Access Protocol

The Alternate Interface Access Protocol is a set of standards for allowing users to control public services, consumer devices and home services with any kind of personal computing device. All services that adhere to the standard would be accessible from any personal device that can interpret the standard. Services would provide a Presentation Independent Template that gives access to all the functions of a service. Information about how a user interface for a service should look on different devices is provided by a User Interface Implementation Description. The implementation description can be provided by service providers, third parties or users (Vanderheiden et al., 2003).

The work with the Alternate Interface Access Protocol is ongoing, and no final documents have been produced yet. Several prototypes have been implemented (Zimmerman et al., 2002), but we still need to see how this standard will evolve.

3.2 Open set of abstractions

3.2.1 Meta widgets

The concept of metawidgets (Blattner and Glinert, 1996) was created in 1996 to simplify creation of multi-modal user interfaces, specifically

those cases where information is encoded for a certain output modality. Metawidgets are multimodal widgets, used to abstract information from the application for the user. To be able to present themselves in different ways for different modalities, metawidgets contain representations for different modalities, or combinations thereof, and methods for selecting among them.

Storing the different presentations of widgets in the widgets themselves poses problems whenever changes need to be done in the look and feel of a modality, or when a new modality is added to an application. In both cases, changes need to be done in each widget of the application, which is both cumbersome and time consuming. It is also a source of consistency errors.

3.2.2 User Interface Markup Language

User Interface Markup Language (UIML) (Abrams et al., 1999) is an XML compliant markup language created in 1999 to describe user interfaces in a device independent way. Style sheets are used to adapt the user interface to different devices. The description of a user interface is made in five sections: `description` lists the individual elements of the user interface, `structure` specifies the organization of elements and which elements in the description are present for a given device, `data` contains data that are device independent but service specific, `style` contains style sheets and device dependent data, and `events` describe the run-time events that can be sent between the user interface and the service. This gives a separation between the application code, the application data, and the user interface code, but it also results in that the UIML description of the user interface quickly gets lengthy and detailed.

User interfaces built with UIML cannot take advantage of device specific features, and they only support user-driven interaction. At the writing moment (oct 2003), renderers for Java, WML, HTML, and Voice XML are available.

3.2.3 UUI

Unified User Interfaces (UUI) is a design and engineering framework composed by three parts: a method for design, a software architecture, and tools (Stephanidis, 2001a, Savidis et al., 2001, Stephanidis, 2001b). The overall goal of UUI is to create accessible user interfaces with high quality of interaction by providing user interfaces tailored to different user groups and situations of use. Situations of use include external

factors as noise, as well as the particular device used for access. A UUI should be possible to use for anyone independently of physical abilities or computer experience.

A UUI is defined as “an interactive system that comprises a single (i.e., unified) interface specification, targeted to potentially all user categories and contexts of use” (Stephanidis, 2001a) pp 376-377. At design time useful abstractions are identified for the application, and alternative dialogue patterns for their realization are implemented. No fixed set of abstractions is provided; the designer is allowed to create application specific abstractions. At run-time, users’ interaction with the system, as well as the users’ context is monitored to detect usage patterns or contextual changes. Based on this information, a decision component is choosing between the alternative dialog patterns in such a way that the speech patterns might be used in a noisy environment or extra help functions might be provided if the user fails to achieve a task.

UUIs address several difficult problems: monitoring user behavior to adapt the user interface, monitoring the user context to adapt the user interface, identifying possible user interface alternatives, choosing the right dialogue pattern. Making conclusions based on user behavior is always difficult, since user behavior often is ambiguous, and making conclusions based on usage context is even more difficult. Having many ambiguous information sources for the decision component can make the system very unpredictable and difficult to use. It is also difficult to identify every alternative dialogue already at design time.

3.3 The Web approach

The Web was created as a way to share documents between researchers around the world (Berners-Lee et al., 1992). The abstractions used to describe the documents allowed people to access them from different computers and different operating systems using Web browsers. As long as there existed a browser that users could run on their machine, they could access the Web of documents. Today, the Web offers a lot more than simple documents, and the devices used for access shows a huge diversity in capabilities of interaction and presentation. A simple approach to achieve device independence for the Web would be to create Web browsers for every device. However, the differences of the devices are too great for them to show the same Web pages. For example, some devices cannot handle pictures or sound, while others have too small screens to display a regular Web page. To address this problem, standards for describing device capabilities have been created, for example the

Composite Capability/Preference Profile (Reynolds et al., 1999). Several attempts have been made to automatically adapt Web pages to devices with small screens (Bickmore and Schilit, 1997, Wobbrock et al., 2002, Trevor et al., 2001). There are also attempts to create an interlingua for Web markup languages, a general markup that can be converted to the others, to simplify Web design for different devices (Menkhaus, 2002). But even if the problems of device differences could be solved, the suitability of the Web as a foundation for device independent services can be questioned.

One problem is the interaction model of the Web. Web browsers of today can only provide page based, user-driven interaction, which makes them less suitable for the range of applications that depend on pushing data to the client (for example games).

Another problem is control of the presentation. At the beginning, the point of using the Web and HTML was that the HTML code was free from information about the presentation, and the browser would take care of the rendering of the Web page. Today, Web service providers make great efforts to control how their services are rendered to end-users: tables are specified on pixel level to control layout, and plug-ins are created for the same purpose (Esler et al., 1999). Many pages are so specialized for a given browser that they display a logo “best viewed in browser x”. This is a problem that might be partially solved with XHTML 1.0 (W3C, 2002) and Cascading Style Sheets (Bos et al., 1998).

One could also argue that the emergence of the Web itself has restricted the access to the internet for people with certain disabilities (Vanderheiden, 1998). In the beginning, before HTML and Mosaic, the Internet was entirely text-based and the information was essentially modality-neutral. The information could easily be transformed to for example Braille or audio output. This meant that available services, as e-mail, news groups, and chat rooms, were accessible to blind people, deaf people, or people with other disabilities, as long as their assistive technology could handle text. The emergence of HTML and Web browsers changed all this. The benefits of graphical presentation, animation, sound effects, and other features of the Web as we know it today comes with the drawback of limited access for users with disabilities.

3.3.1 HTML, XHTML and Cascaded Style Sheets

One way to create Web pages that are not specialized for a given Web browser but still provide means to control the physical presentation is to use style sheets. The Web page is written in HTML (Raggett et al., 1999) or XHTML (W3C, 2002) without presentation information, and then different style sheets are used to describe the presentation of the page for different browsers or devices.

XHTML 1.0 is the successor of HTML 4, and is created to be more flexible due to its XML conformance. In XHTML 1.0 style sheets are the default way to control presentation. The content is described in XHTML and Cascading Style Sheets (Bos et al., 1998) are used to describe the presentation. This decision is an effort from the World Wide Web Consortium to solve the problem with Web pages targeted for a specific Web browser, and links presented as pictures to control their appearance. With the strict partition in content and presentation it is easier to achieve full control of the presentation of Web pages. However, XHTML 1.0 has proven to be too complicated for some devices, so the specification has been split up in several modules (Butler, 2001).

3.3.2 XML

The eXtensible Markup Language (XML) (Bray et al., 2000) is a descendant from SGML and HTML, and designed to structure data and describe information. The set of tags is not defined as in HTML, but the author of an XML document can define their own tags for specific purposes. A Document Type Definition is then used to specify the vocabulary and syntax of the defined tags. Using XML, Web pages and Web content can be encoded in a device independent way, and then transformed to different output formats. One way to do this is using the EXtensible Stylesheet Language Transformations (XSLT) (Kay, 2002).

3.3.3 W3C Device Independence Working Group

The World Wide Web Consortium has created a working group addressing the problems of device independence for the Web. The purpose of this group, created in February 2001, is to review the challenges, possibilities and problems that arise in Web authoring due to the emergence of new devices such as palm computers and Web TV. The focus is entirely on the developer side, particularly on how to adapt and present Web applications, and how Web applications can be accessed from different devices. The long-term goals of the working group are that different devices can get Web content adapted to their presentation

capabilities and that authors can provide this in an efficient way (W3C, 2001). The group has not yet issued any W3C Recommendations, but there is a working draft identifying various challenges concerning both application and device side of device independent authoring (Lewis, 2002).

4 Examples of Model-Based Systems

In this section, systems with model-based approaches are presented.

4.1 ITS

ITS was a model-based development tool created in 1990 at IBM T.J. Watson research center (Wiecha et al., 1990). The purpose of ITS was to provide a tool for developing applications that could adapt to differences in for example screen size, color capability, input devices and user language. However, ITS never targeted other user interfaces than GUIs, nor did it target mobile devices.

The ITS architecture separates applications into four layers: actions, dialog, style rules, and style programs. The actions layer handles reading and writing of data, dialog handles the content and the sequencing of the interaction, style rules are specifications of how parts of the user interface should be rendered, and style programs execute the style rules at run-time. This allows non-programmers to design user interfaces by writing style rules instead of programming codes.

A major drawback of ITS was the way the style rules were considered. A set of style rules, i.e. a style, was considered as something general that could be moved between different applications. Moving styles between applications could of course never produce optimal user interfaces for the different applications, especially if the styles used application specific data types in the rules. This lead the developers of ITS to the conclusion that in was preferable not to use applications with different styles. In the Ubiquitous Interactor we have chosen to consider the style as application specific and create many different styles for a single application, thus creating different user interfaces.

4.2 Plastic User Interfaces

Plastic user interfaces are an attempt from 2001 to create applications with user interfaces adapted to different devices (Calvary et al., 2001). A

plastic user interface is defined as a user interface that has the capacity to “withstand variations of context of use while preserving usability” (Calvary et al., 2001). This might be accomplished automatically or with human intervention. Context of use in this setting includes platform related issues like device features and bandwidth, and also the users’ environment, including peripheral people and objects.

Plastic user interfaces are using six different models: the concepts model, the task model, the platform model, the environment model, the interactors model and the evolution model (Calvary et al., 2001). The *concepts model* covers the concepts that users can manipulate in different contexts, the *task model* describes the tasks users can accomplish, the *platform model* and the *environment model* describes the context of use, the *evolution model* specifies allowed changes of state within a context as well as conditions for changing context, and the *interactors model* describes “resource sensitive multimodal widgets” that are available for generation of the concrete user interface. To generate a user interface, first a task specification is created with information from the concepts, task, and platform model, then an abstract UI description is created with information from the platform model. A concrete UI specification is created with information from the environment and interactors model. All the models described above are *initial* models, i.e. they are created by the developer.

When a user interface for a new context is created, a translation from the first UI takes place in each step (task specification, abstract UI specification, concrete UI specification), where information from new concepts, platform, etc. models are incorporated. An application can also have different predefined user interfaces to choose from.

Calvary et al. provides a good foundation for creating device independent applications by having different models for different information sources, e.g. platform, environment and tasks. However, by using information from the platform model in the task specification and the abstract UI specification all steps of the UI generation must be translated for a new user interface. If the task specification and the abstract UI specification were kept device independent, only the generation of the concrete UI specification would need to be done for a new user interface. The filtering of tasks, i.e. not all tasks are available for all devices, could be made at that stage instead of in the task specification.

4.3 XIML

The eXtensible Interface Markup Language (XIML) (Puerta and Eisenstein, 2001) is an XML-based attempt from 2001 to capture *interaction data* – the data defining and relating the elements of the user interface – and thus creating device independent user interfaces. In XIML designers can define *intermediate presentation elements* that are device independent. By creating relations between intermediate presentation elements and different user interface widgets, multiple user interfaces can be derived from a single specification. The relations are dynamic, and can be redefined during run-time if the context changes (for example the amount of available screen space). At the moment, XIML is only aimed at Web-based applications but according to the authors the approach could be extended to other types of applications as well.

XIML has a wider scope than the Ubiquitous Interactor. In XIML, the application domain, users, tasks and user-application dialog is modeled. The adaptations are based on user preferences, devices, and context of use, whereas UBI only takes device features in consideration. This makes XIML a more complex approach than UBI. UBI can in return provide more predictable, but still device tailored user interfaces, which could benefit both service providers and users.

5 Summary

Device independent services have been a research issue during different periods of the computer science history, and each period has had its specific reason. During the seventies and early eighties, developers were up against large variations in hardware and needed methods for developing device independent applications to reduce workload and increase the number of available applications. At the beginning of the 21st century, we are trying to put ubiquitous and mobile computing in place. Mobile users that use different devices in different situations want to access services from wherever they are call for services that can be accessed from a wide range of devices.

The older approaches may have had different purposes, but can still give inspiration to device independence in the ubiquitous computing era. The Ubiquitous Interactor is a new approach that draws inspiration from earlier projects to face the challenges of the new computing.

6 References

- Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M. and Shuster, J. E. (1999) UIML - an appliance-independent XML user interface language, *Computer Networks*, **31**, 1695-1708.
- Berners-Lee, T., Caillau, R., Groff, J.-F. and Pollerman, B. (1992) World-Wide Web: The Information Universe, *Electronic Networking: Research, Applications and Policy*, **2**, (52-58).
- Bickmore, T. W. and Schilit, B. N. (1997) Digestor: Device-independent Access to the World Wide Web, in Proceedings of 6th International World Wide Web Conference.
- Blattner, M. M. and Glinert, E. P. (1996) Multimodal Integration, *IEEE Multimedia*, **3**, (4), 14-24.
- Bos, B., Wium Lie, H., Lilley, C. and Jacobs, I. (1998) *Cascading Style Sheets, level 2. CSS2 Specification*, W3C Recommendations, World Wide Web Consortium, <http://www.w3.org/TR/REC-CSS2/>.
- Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. (2000) *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation 6 October 2000, W3C Recommendation, World Wide Web Consortium, <http://www.w3.org/TR/REC-xml>.
- Butler, M. H. (2001) *Current Technologies for Device Independence*, Technical Report, HP Laboratories Bristol, www.hpl.hp.com/techreports/2001/HPL-2001-83.pdf.
- Calvary, G., Coutaz, J. and Thevenin, D. (2001) A Unifying Reference Framework for the Development of Plastic User Interfaces, in Proceedings of Engineering HCI.
- Eisenstein, J. and Puerta, A. (2000a) Adaptation in Automated User-Interface Design, in Proceedings of International Conference on Intelligent User Interfaces.
- Eisenstein, J. and Puerta, A. (2000b) Adaption in Automated User-Interface Design, in Proceedings of International Conference on Intelligent User Interfaces.

Eisenstein, J., Vanderdonckt, J. and Puerta, A. (2001) Applying Model-Based Techniques to the Development of UIs for Mobile Computers, in Proceedings of International Conference on Intelligent User Interfaces.

Esler, M., Hightower, J., Anderson, T. and Borriello, G. (1999) Next Century Challenges: Data-Centric Networking for Invisible Computing. The Portolano Project at the University of Washington., in Proceedings of The Fifth ACM International Conference on Mobile Computing and Networking, MobiCom 1999.

Foley, J. D., Wallace, V. L. and Chan, P. (1984) The Human Factors of Computer Graphics Interaction Techniques, *IEEE Computer Graphics and Applications*, **4**, (6), 13-48.

Johnson, J. (1992) Selectors: Going Beyond User-Interface Widgets, in Proceedings of Human Factors in Computing Systems.

Kay, M. (2002) *XSL Transformations (XSLT) Version 2.0, W3C Working Draft 16 August 2002*, Working Draft, World Wide Web Consortium, <http://www.w3.org/TR/xslt20/>.

Lewis, R. (2002) *Authoring Challenges for Device Independence*, W3C Working Draft, World Wide Web Consortium.

Menkhaus, G. (2002) *Adaptive User Interface Generation in a Mobile Computing Environment*, PhD thesis, University of Salzburg.

Myers, B. A. (1990) A New Model for Handling Input, *ACM Transactions on Information Systems*, **8**, (3), 289-320.

Myers, B. A., Hudson, S. E. and Pausch, R. (2000) Past, Present and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **7**, (1), 3-28.

Nichols, J., Myers, B. A., Higgings, M., Hughes, J., Harris, T. K., Rosenfeld, R. and Pignol, M. (2002) Generating Remote Control Interfaces for Complex Appliances, in Proceedings of 15th Annual ACM Symposium on User Interface Software and Technology, Paris, France, 161-170.

Olsen, D. J., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P. (2000) Cross-modal Interaction using XWeb, in Proceedings of

Symposium on User Interface Software and Technology, UIST 2000, 191-200.

Puerta, A. and Eisenstein, J. (2001) *XIML: A Universal Language for User Interfaces*, White Paper, RedWhale Software, www.redwhale.com.

Raggett, D., Le Hors, A. and Jacobs, I. (1999) *HTML 4.01 Specification*, W3C Recommendation, World Wide Web Consortium, <http://www.w3.org/TR/html4/>.

Reynolds, R., Hjelm, J., Dawkins, S. and Singhal, S. (1999) *Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation*, W3C Note, World Wide Web Consortium W3C.

Savidis, A., Akoumianakis, D. and Stephanidis, C. (2001) The Unified User Interface design method, In *User interfaces for all - concepts, methods and tools*, pp. 417-440.

Stephanidis, C. (2001a) The Concept of Unified User Interfaces, In *User Interfaces for All - Concepts, Methods, and Tools* (Ed, Stephanidis, C.) Lawrence Erlbaum Associates, pp. 371-388.

Stephanidis, C. (2001b) Unified User Interface Software Architecture, In *User Interfaces for All - Concepts, Methods, and Tools* (Ed, Stephanidis, C.) Lawrence Erlbaum Associates, pp. 389-415.

Szekely, P. (1996) Retrospective and Challenges for Model-Based Interface Development, in Proceedings of International Workshop of Computer-Aided Design of User Interfaces, xxi-xliv.

Trevor, J., Hilbert, D. M., Schilit, B. N. and Khiau Koh, T. (2001) From Desktop to Phonetop: A UI for Web Interaction on Very Small Devices, in Proceedings of 14th Annual ACM Symposium on User Interface Software and Technology, Orlando, FL, 121-130.

W3C (2001) *Device Independence Working Group Charter*, Working Group Charter, World Wide Web Consortium.

W3C (2002) *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, W3C Recommendation, World Wide Web Consortium, <http://www.w3.org/TR/xhtml1/#acks>.

Vander Zanden, B. and Myers, B. A. (1990) Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces, in Proceedings of Human Factors in Computing Systems, CHI.

Vanderheiden, G. C. (1998) Cross-modal access to current and next-generation Internet - fundamental and advanced topics in Internet accessibility, *Technology and Disability*, **8**, (3), 115-126.

Vanderheiden, G. C., Zimmerman, G. and Trewin, S. (2003) A Standard for Controlling Ubiquitous Computing and Environmental Resources from Any Personal Device, in Proceedings of Human-Computer Interaction International, 499-503.

Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. (1990) ITS: a Tool for Rapidly Developing Interactive Applications, *ACM Transactions on Information Systems*, **8**, (3), 204-236.

Wobbrock, J., Forlizzi, J., Hudson, S. and Myers, B. A. (2002) WebThumb: Interaction Techniques for Small-Screen Browsers, in Proceedings of 15th Annual Symposium on User Interface Software and Technology, Paris, France, 205-208.

Zimmerman, G., Vanderheiden, G. C. and Gilman, A. (2002) Universal Remote Console - Prototyping for the Alternate Interface Access Standard, in Proceedings of 7th ERCIM International Workshop on User Interfaces for All, 524-531.

Paper 2

The Ubiquitous Interactor – Mobile Services with Multiple User Interfaces

Stina Nylander, Markus Bylund, and Annika Waern

The Ubiquitous Interactor – Mobile Services with Multiple User Interfaces

Stina Nylander, Markus Bylund, Annika Waern
Swedish Institute of Computer Science
SICS Technical Report T2003:17

E-mail: {stina.nylander, markus.bylund, annika.waern}@sics.se

Abstract

The Ubiquitous Interactor (UBI) addresses the problems of design and development that arise around services that need to be accessed from many different devices. In UBI, services present themselves with different user interfaces on different devices. This is done by separation of user-service interaction and presentation. The interaction is kept the same for all devices, and different presentation information is provided for different devices. This way, tailored user interfaces for many different devices can be created without multiplying development and maintenance work. In this paper we describe the design of UBI, the system implementation, and two services implemented for the system: a calendar service and a stockbroker service.

1 Introduction

The Ubiquitous Interactor (UBI) is a system addressing the problems with design and development that arise when service providers face the vast range of computing devices available on the consumer market.

Users have a wide range of devices at their disposal for accomplishing different tasks: desktop computers and laptop computers for office work, wall-sized screens for presentations in large groups, PDAs and cellular phones for mobile tasks. The range of services is equally wide: information services, shopping and entertainment. This opens for using services from different devices in different situations. Users could access for example their shopping services from a desktop computer at home and from a cellular phone on the bus. Unfortunately, this is often not possible since devices and services cannot be freely combined. Devices

have different capabilities of user interaction and presentation, and most services cannot adapt their user interfaces to these differences. This means that users often have to use different versions of a service from different providers to access the same functionality. This causes problems of synchronization and compatibility.

The main approach to making services accessible from multiple devices today is versioning. However, with many different versions of services, development and maintenance work get very cumbersome, and it is difficult to keep consistency between different versions. Another popular method is to use Web user interfaces since most devices run a Web browser. However, adaptations are still needed, for example translation between markup languages and layout changes for small screens. It is also difficult to take advantage of device specific features and to control how user interfaces will be presented to end-users. Thus, we need new and robust methods for developing services that can adapt to different devices.

UBI offers a possibility to develop a single device independent version of a service, and then create device specific user interfaces for it. To accomplish this, UBI uses interaction acts (Nylander and Bylund, 2002) (see section 4.1) to describe the user-service interaction in a device independent way. This description is used by all devices to generate an appropriate user interface. The presentation of user interfaces can be controlled through customization forms (Nylander and Bylund, 2002) (see section 4.2), which contain service and device specific information of how user interfaces should be presented. This makes it possible to develop services once and for all, and tailor their user interfaces to different devices.

The rest of the paper is outlined as follows: First the background to the UBI system and some related work is discussed. Then the design decisions are described and motivated, followed by a description of the implementation of the system and services for it. Finally some conclusions are presented.

2 Background

Our interest and need for device independent services are results from our previous work with the next generation electronic services in the sView project (see below). However, the need for device independent applications is not new. During the seventies and early eighties,

developers faced large differences in hardware. That time the problem disappeared when the personal computer emerged. The hardware got standardized to mouse, keyboard and desktop screen, and development of direct manipulation user interfaces worked similarly in different operative systems (Myers et al., 2000).

The situation that we face today is different. We experience a paradigm shift from application-based personal computing to service-based ubiquitous computing. In a sense, both applications and services can be seen as sets of functions and abilities that are packaged as separate units (Espinoza, 2003). However, while applications are closely tied to individual devices, typically by a local installation procedure, services are only manifested locally on devices and made available when needed. The advance of Web-based services during the nineties can be seen as the first step in this development. Instead of accessing functionality locally on single personal computers, users could access functionality remotely from any Internet connected PC. This will change though. With the development of the multitude of different devices that we see today (e.g. smart phones, PDAs, and wearable computers) combined with growing requirements on mobility and ubiquity, the Web-based approach is no longer enough.

For this reason, we have developed the sView system (Bylund, 2001, Bylund and Espinoza, 2000) that provides an example of what the infrastructure for the next generation service-based computing could be like. With sView, each user is provided with a personal service briefcase in which electronic services from different vendors can be stored. When accessing these services, users not only get a completely personalized usage experience, they can also benefit from the use of a wide variety of different devices, continuous usage of services while switching between different devices, and network independence (completely off-line use is possible).

For a long period, our only way of supporting the versatility of the range of device types in sView was to require service providers to implement many alternative user interfaces for their services. A typical end-user service for example implemented a traditional GUI specified in Java Swing, an HTML and a WML interface for remote access over HTTP, and an SMS interface for remote access from cellular phones. While the sView system provides support for handling transport of UI components, presentation, events etc, service providers still had to implement the actual user interfaces (Swing widgets, HTML/WML documents, and text

messages) and interpret user actions (Java events, HTTP posts from HTML and WML forms, and text input).

This approach required great implementation and maintenance efforts of the service providers. The standard solution to the problem was no longer viable however, and alternative solutions needed to be explored. The multitude of device types we see today is not due to competition between vendors as before, but rather motivated by requirements of specialization. Different devices are designed for different purposes and thus their diverse appearance. As a result, the solution this time needs to support simple implementation and maintenance of services without losing the uniqueness of each type of device. This is what we set out to solve with UBI.

3 Related Work

Much of the inspiration for the Ubiquitous Interactor (UBI) comes from early attempts to achieve device independence, or in other ways simplify development work by working on a higher level than device details.

We have already mentioned that lack of hardware standards created a need of device independent applications during the seventies and the eighties. User Interface Management Systems like Mike (Olsen, 1987) and UofA* (Singh and Greene, 1989) addressed this problem, together with model-based approaches like Humanoid (Szekely et al., 1993). Others proposed more partial solutions to shield developers from differences in input devices (Myers, 1990), or guide them in the selection of input devices and interaction techniques (Foley et al., 1984).

In current research, device independence is addressed in two different research fields, that of ubiquitous and mobile computing and that of universal access. UBI has its origin in the ubiquitous and mobile research, but provides solutions that can be of use in universal access too.

XWeb (Olsen et al., 2000) and PUC (Nichols et al., 2002) encodes the data sent between application and client in a device independent format using a small set of predefined data types, and leaves the generation of user interfaces to the client. Unlike UBI, they do not provide any means for service providers to control the presentation of the user interfaces. It is completely up to the client how a service will be presented to end-users.

User Interface Markup Language (UIML), is an XML compliant markup language for specification of user interfaces (Abrams et al., 1999). This description is converted to another language, for example Java or HTML. UIML differs from UBI in that its descriptions cannot take advantage of device specific features, and it only supports user-driven interaction.

Unified User Interfaces (UUI) (Stephanidis, 2001) is a design and engineering framework for adaptive user interfaces. In UUI, user interfaces are described in a device independent way using categories defined by designers. Designers then map the description categories to different user interface elements. This means that designers have control of how the user interface will be presented to the end-user, but since different designers can use their own set of description categories the system cannot provide any default mappings. In UBI, we have chosen to work with a pre-defined set of description categories, along with the possibility for designers to create mappings. This makes it possible for the system to provide default mappings at the same time as designers can control the presentation of the user interface.

4 Design

In the Ubiquitous Interactor (UBI), we have chosen the interaction between users and services as our level of abstraction in order to obtain units of description that are independent of device type, service type, and user interface type. Interaction is defined as *actions that services present to users, as well as performed user actions, described in a modality independent way*. Some examples of interaction according to this definition would be: making a choice from a set of alternatives, presenting information to the user, or modify existing information. Pressing a button, or speaking a command would not be examples of interaction, since they are modality specific actions. By describing the user-service interaction this way, the interaction can be kept the same regardless of device used to access a service. It is also possible to create services for an open set of devices.

The interaction is expressed in interaction acts that are exchanged between services and devices. In some cases the service in question will actually be running on the device, in other cases it might be on a server. Interaction acts are interpreted by the device and user interfaces are generated based on interaction acts and additional presentation information, see figure 1. Whether services are running locally or on a

server does not affect the way services express themselves, or the way interaction acts are interpreted.

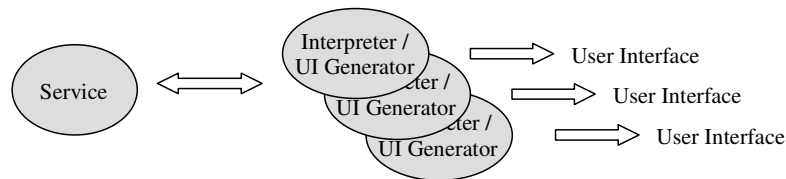


Figure 1: Services offer their interaction expressed in interaction acts, and an interpreter generates a user interface based on the interpretation. Different interpreters generate different user interfaces.

4.1 Interaction Acts

Interaction acts are abstract units of user-service interaction that contain no information about modality or presentation. This means that they are independent of devices, services and interaction modality. Throughout this work, we assume that most kinds of interaction can be expressed using a fairly limited set of interaction acts. User-service interaction for a wide range of services can be described by combining single interaction acts and groups.

Through analysis of existing services and applications, we have defined a set of eight interaction acts that are supported in UBI: `input`, `output`, `select`, `modify`, `create`, `destroy`, `start` and `stop`. In this definition `input` is input to the system, `output` is output to the user, `select` is selection from a set of alternatives, and `modify` is modification of information stored in the system. `create` is creation of new objects, `destroy` is deletion of existing objects, and `start` and `stop` handle the interaction session with the service. All interaction acts except `output` returns user actions to services. `output` only presents information that users cannot act upon.

During the user-service interaction, the system needs more information about the interaction acts than its type. Interaction acts need to be uniquely identifiable, so that user actions can be associated with them. Users perform actions on user interface components, and those actions need to be linked to the original interaction acts so that services can interpret them correctly. Most services will offer several interaction acts

of the same type, and need a way to identify which one users acted upon. It must also be possible to define for how long a user interface component based on an interaction act should be present in the user interface and when it should be removed. Otherwise only static user interfaces can be created. It must be possible to create modal user interface components based on interaction acts, e.g. components that lock the user-service interaction until certain actions are performed by users. This way, user actions can be sequenced when needed. All interaction acts also need a way to hold default information, so that there always is something on which to base the rendering of interaction acts. Finally, it is important to be able to attach metadata to interaction acts. Metadata can for example contain domain information, or restrictions on user input that are important to the service.

In more complex user-service interaction, there is a need to group several interaction acts together, because of their related function, or the fact that they need to be presented together. An example could be the play, rewind, forward and stop functions of a CD player. The structure obtained by the grouping can be used as input when generating the user interfaces. In order to be useful, these groups should allow nesting.

4.2 Controlling the Presentation

To give service providers a possibility to specify how user interfaces of their services will be presented to end-users, services must be able to provide detailed presentation information. Control of presentation has proven to be an important feature of methods for developing services (Esler et al., 1999, Myers et al., 2000), since it is used for example for branding.

In UBI, presentation information is specified separately from user-service interaction. This allows for changes and updates in the presentation information without changing the service. The main forms of presentation information are *directives* and *resources*. Directives can link interaction acts to for example widgets or templates of user interface components. Resources could be pictures, sounds or other media that are used in the rendering of an interaction act.

It is optional to provide presentation information in UBI. If no presentation information is specified, or only partial information is provided, user interfaces are generated with default settings. However, by providing detailed information service providers can fully control how their services will be presented to end-users.

5 Implementation

The Ubiquitous Interactor (UBI) has three main parts: the Interaction Specification Language, customization forms, and interaction engines. The Interaction Specification Language is used to encode the interaction acts sent between services and user interfaces, customization forms are used to control the presentation of user interfaces, and interaction engines generate user interfaces based on interaction acts and presentation information in customization forms. The different parts are defined at different levels of specificity, where interaction acts are device and service independent, interaction engines are device dependent, and customization forms are service and device dependent, see figure 2.

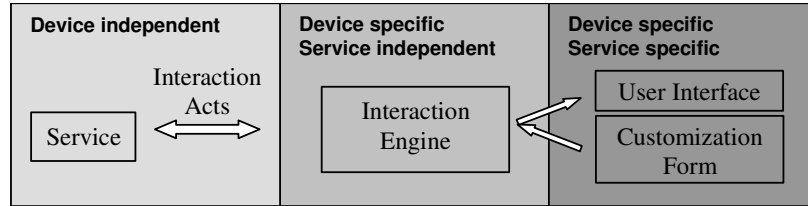


Figure 2: The three layers of specification in the Ubiquitous Interactor. Services and interaction acts are device independent, interaction engines are service independent and device or user interface specific. Customization forms and generated user interfaces are device and service specific.

5.1 Interaction Specification Language

Interaction acts are encoded using the Interaction Specification Language (ISL), which is XML compliant.

Each interaction act has a unique id that is used to map performed user interactions to it. It also has a life cycle value that specifies when components based on it are available in the user interface. The life cycle can be *temporary*, *confirmed*, or *persistent*. Interface components based on temporary interaction acts are presented in the user interface for a specified time and then removed by UBI, for example a logotype shown for a few seconds when a service is starting. Interface components based on confirmed interaction acts are presented in the user interface until the user has performed a given action, for example entered required login information. Interface components based on persistent interaction acts are available in the user interface during the whole user-service interaction, or until UBI removes them. The default life cycle value is persistent. All

interaction acts can be given a symbolic name, and belong to a named presentation group in a customization form. This will be discussed further in the next section.

Interaction acts also have a modality value that specifies if components based on them will lock other components in the user interface. The value of the modality can be true or false. If the modality value is true, the component is locking other components in the user interface until the user performs a given action, for example confirming an earlier action. The default modality value is false. All interaction acts contain a string that is used to hold default information. It is also possible to attach metadata to all interaction acts. Listing 1 shows the ISL encoding of a `select` interaction act.

```
<select>
  <id>235690</id>
  <life>persistent</life>
  <modal>>false</modal>
  <response-number>1</response-number>
  <string>Browse</string>
  <alternative>
    <id>98770</id>
    <string>Previous</string>
    <return-value>prev</return-value>
  </alternative>
  <alternative>
    <id>66432</id>
    <string>Next</string>
    <return-value>next</return-value>
  </alternative>
</select>
```

Listing 1: ISL encoding of a `select` interaction act with id, name, life cycle, modality, and default content information. `select` interaction acts also contain a value for the number of alternatives that can be selected. Alternatives inherit life cycle and modality from the selection interaction act.

Interaction acts can be grouped using a designated tag `isl`, and groups can be nested to provide more complex user interfaces. These groups of interaction acts contain the same type of information assigned to single interaction acts: life cycle, modality, default information and metadata. Listing 2 shows the ISL encoding of a simplified example of two interaction acts grouped using the `isl` tag.

```

<isl>
  <id>980796</id>
  <life>persistent</life>
  <modal>>false</modal>
  <string>SICS info</string>
  <output>
    <id>235690</id>
    <life>persistent</life>
    <modal>>false</modal>
    <string>SICS AB</string>
  </output>
  <output>
    <id>342564</id>
    <life>persistent</life>
    <modal>>false</modal>
    <string>http://www.sics.se</string>
  </output>
</isl>

```

Listing 2: ISL encoding of two `output` interaction acts grouped using the `isl` tag.

The ISL code sent from services to interaction engines contains all information about the interaction acts: id, name, group, life cycle, modality, and metadata. A large part of this information is only useful for the interaction engine during generation of user interfaces. There is no point in sending information concerning user-service interaction handling back to the service. Thus, when users perform actions, only the relevant parts of interaction acts are sent back to the service. This includes the id for all interaction acts and for those interaction acts that imply user data input it also includes the data, for example the value of the selected alternative in selection interaction acts, the parameters of create interaction acts, or other input data. Two different DTDs have been created for this purpose, one for encoding interaction acts sent from services to interaction engines, and one for encoding interaction acts sent from interaction engines to services, see appendix A and B.

5.2 Customization Forms Implementation

Customization forms contain device and service specific information about how the user interface of a given service should be presented. Information can be specified on three different levels: group level, type level or name level. Information on group level affects all interaction acts of a group, and can be used to provide a look and feel for whole services or parts of services. Information at interaction act type level concerns all interaction acts of the given type; and information on name level concerns all interaction acts with the given symbolic name. The levels can also be combined, for example creating specifications for interaction

acts in a given group of a given type, or in a given group with a given name.

The Interaction Specification Language contains attributes for creating the different mappings. Each interaction act or group of interaction acts can be given an optional symbolic name that is used in mappings where the name level is involved. This means that each interaction act with a certain name is presented using the information mapped to the name. Interaction acts or groups of interaction acts can also belong to a named group in a customization form. All interaction acts that belong to a group are presented using the information associated with the group (and possibly with additional information associated with their name or type).

```
<select>
  <id>235690</id>
  <name>browseSelect</name>
  <group>calendar</group>
  <life>persistent</life>
  <modal>>false</modal>
  <response-number>1</response-number>
  <string>Browse</string>
  <alternative>
    ...
  </alternative>
  <alternative>
    ...
  </alternative>
</select>
```

Listing 3: Shortened ISL encoding of the `select` interaction act from listing 1, with an additional symbolic name `nextSelect`, that belongs to the customization form group `calendar`.

Listing 3 shows a shortened encoding of the `select` interaction act from listing 1 with a symbolic name, and as a member of the customization form group `calendar`. Customization forms are structured, and can be arranged in hierarchies. This allows for inheriting and overriding information between customization forms. A basic form can be used to provide a look and feel for a family of services, with different service specific forms adding or overriding parts of the basic specifications to create service specific user interfaces. Customization forms are encoded in XML and a DTD can be found in appendix C. An entry in a customization form can be either a directive or a resource. Directives are used for mappings to widgets or other user interface components and resources are used to associate media resources to interface components. Both directive mapping and resource association can be made on all three

levels, group, type and name. Listing 4 shows an example of a directive mapping based on the type of the interaction act, in this case `output`.

```
<element name="output">
  <directive>
    <data>
      se.sics.ubi.swing.OutputLabel
    </data>
  </directive>
</element>
```

Listing 4: A mapping on type level for an `output` interaction act.

A customization form does not need to be complete. Interaction acts that have no presentation information specified in the form are presented with defaults.

5.3 An Example

To illustrate the user-service interaction in more detail we will examine an example. The `select` interaction act in listing 3 has a name that can be used in mappings in customization forms. Listing 5 shows a sample mapping on name level from a customization form.

```
<id name="browseSelect">
  <directive>
    <data>
      se.sics.ubi.swing.SelectButton
    </data>
  </directive>
</id>
```

Listing 5: A mapping on name level in a customization form.

This mapping instructs the interaction engine to use a certain widget when presenting the interaction act. The generated presentation could look like figure 3.



Figure 3: An example rendering of the `select` interaction act in listing 3.

We can imagine that this interaction act is used to browse a list of items using two different operations: `new` and `next`. When a button is pressed, the `select` interaction act in listing 6 is returned to the service.

```
<select>
  <id>98770</id>
  <alternative>
    <id>33465</id>
    <return-value>next</return-value>
  </alternative>
</select >
```

Listing 6: A `select` interaction act returning one selected alternative to a service.

The service would interpret the interaction act and update the user interface if necessary.

5.4 Interaction Engines Implementation

Interaction engines interpret interaction acts and generate suitable user interfaces of a given type for services on a given device or family of devices. Interaction engines also encode performed user actions as interaction acts and send them back to services. Examples of interaction engines are an engine for Web user interfaces on desktop and laptop computers, and an engine for Java Swing GUIs on handheld computers. Devices that can handle several types of user interfaces can have many interaction engines installed.

During user-service interaction, interaction engines parse interaction acts sent by services, and generate user interfaces by creating presentations of each interaction act. If specific presentations, or media resources, are specified for an interaction act in the customization form of a service, that presentation is used. Otherwise, interaction engines have defaults for each type of interaction act. For example, an `output` could be rendered as a label, or speech generated from its default information, while an `input` could be rendered as a text field or a standard speech prompt. Figure 4 shows presentations of an `output` and an `input` interaction act. The `output` interaction act is presented as a Tcl/Tk label showing the default information of the interaction act, and as a Java Swing label displaying an image specified in the customization form (picture 4a and 4b). An alternative presentation could be generated speech saying "SICS AB". The `input` interaction act is presented using Java Swing as a text field with a submit button, and an editable combo box with a text label (picture 4c and 4d).

We have implemented interaction engines for Java Swing, HTML, and Tcl/Tk user interfaces. All three interaction engines can generate user

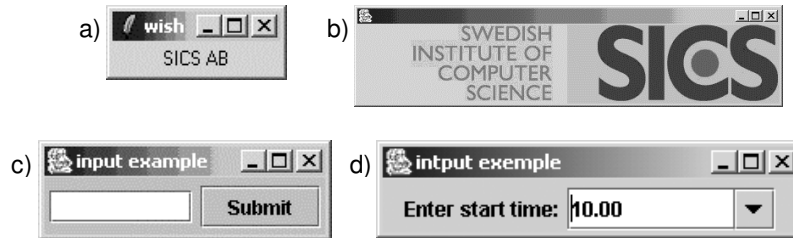


Figure 4: Rendering examples of an output and an input interaction act. Picture a and b are renderings of an output interaction act, and picture c and d are renderings of an input interaction act. Picture a is a Tcl/Tk label using the default information of the interaction act, while picture b is a Java Swing label displaying an image specified in the customization form. Picture c is a Java Swing text field with a button to submit entered text, while picture d is a Java Swing label and an editable combobox for choosing or entering time expressions.

interfaces for desktop computers. The default renderings of the Tcl/Tk interaction engine are designed to create user interfaces suitable for PDAs.

5.4.1 Java Swing Interaction Engine

The Java Swing interaction engine creates Java Swing widgets based on interaction acts and customization forms. Mappings are made between single interaction acts and widgets, as well as between groups of interaction acts and widgets. Mappings can be made to single widgets (e.g. a button) or to complex ones (e.g. panels with many widgets in). The Swing interaction engine can make use of both the specified lifecycle and modality of interaction acts. Interaction acts with confirmed life cycle can be rendered in a dialog window, and if the interaction acts are modal that dialog window can be made modal.

5.4.2 HTML Interaction Engine

The HTML interaction engine translates between interaction acts and HTML code and user feedback is handled with HTML Forms. The nature of HTML user interfaces does not support all features of interaction acts. Since HTML user interfaces are user-driven and non-modal, the different life cycle and modality values of interaction acts are not supported.

5.4.3 Tcl/Tk Interaction Engine

The Tcl/Tk interaction engine generates Tcl/Tk code based on interaction acts and customization forms to produce graphical user interfaces for PDAs. The code is executed by a small tcl client running on the device.

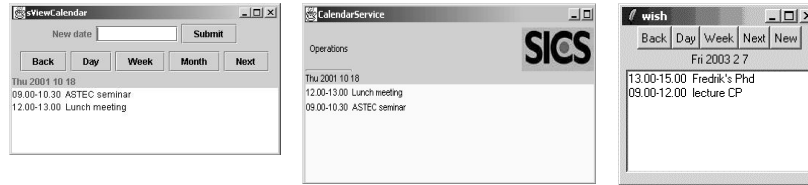


Figure 5: Three user interfaces to the calendar service generated from the same interaction acts. The two to the left are generated by the Java Swing interaction engine using two different customization forms. The one to the right is generated by the Tcl/Tk interaction engine.

User actions are encoded in an internal format that is converted to interaction acts by the interaction engine and sent back to services. Mappings in customization forms are made between interaction acts, and chunks of Tcl/Tk code. The Tcl/Tk interaction engine is currently not using the life cycle or modality information of the interaction acts. The Tcl/Tk interaction engine is not running on the PDA. Instead, it is running on the same machine as the service, and the generated Tcl/Tk code is sent to the device over a socket connection. Our test machine has been a Compaq Ipaq 3850 with a Tcl/Tk version for Windows CE available from <http://www.rainer-keuchel.de/wince/tcltk-ce.html>.

6 Services

We will present two different services to illustrate how the Ubiquitous Interactor (UBI) works, a calendar service and a stockbroker service.

6.1 Calendar Service

The calendar service was the first service created for UBI and provides a good example of a service that it is useful to access from different devices. Calendar information may often be entered from a desktop computer at work or at home, but mobile access is needed to consult the information on the way to a meeting or in the car on the way home. Sometimes appointments are set up out of office (in meeting rooms or restaurants) and it is practical to be able to enter that information immediately and not wait to get back to the office.

The calendar service supports basic calendar operations as entering, edit and delete information, navigate the information, and display different views of the information. The service is accessible from three types of user interfaces: Java Swing and HTML user interfaces for desktop computers, and a Tcl/Tk user interface for handheld computer.

Two different customization forms have been created for Java Swing, and one each for Tcl/Tk and HTML. An example of different presentations could be a `select` interaction act presented as a panel with five buttons (back, day view, week view, month view, next) in one of the Swing UIs, as a pull-down menu in the other, and with only four buttons in the PDA UI (a decision on customization form level not to present a month view on the PDA) (see figure 5). These different presentations are created from the same interaction act, combined with different presentation information.

6.2 Stockbroker Service

The stockbroker service TAP Broker has been developed as a part of a project at SICS that works with autonomous agents that trade stocks on the behalf of users (Lybäck and Boman, 2003). Autonomous agents trade stocks on the behalf of users. Each agent is trading according to a built in strategy (for example buy low, sell high, or buy and hold (Boman et al., 2001)), and users can have one or more agents trading for them. Our service provides users with feedback on how their agents are performing so that they know when to change agent, or shut them down.

The TAP Broker service provides agent owners with feedback on the agent's actions: order handling of the agent (placing and canceling orders), and transactions performed by the agent (buying or selling stocks). It also provides information about the agent's state: the account state (the amount of money it can invest), status (running or paused), activity level (number of transactions per hour), portfolio content, and the current value of the portfolio. However, it does not provide any means to configure or control the agent. The agents are created to work autonomously and cannot be manipulated from outside for security reasons.

We have implemented customization forms for Java Swing, Tcl/Tk and HTML (see figure 6 for example pictures). For Java Swing, two quite different customization forms have been developed: one that generates a user interface appropriate for desktop screens, and one that generates a user interface for very small devices like java enabled cellular phones. Since the screen size and presentation capabilities of desktop computers, PDAs and cellular phones are very different, user interfaces for the smaller devices only present parts of the available information.

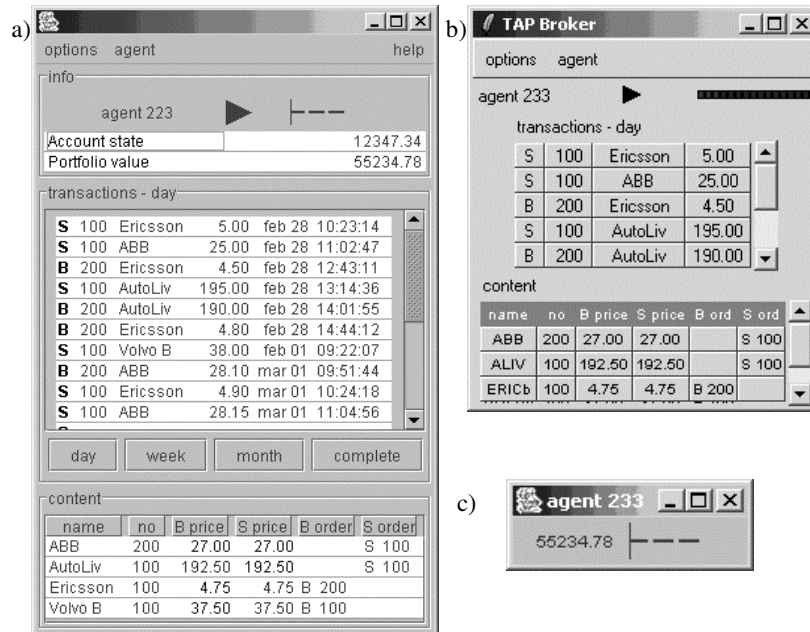


Figure 6: Three different examples of user interfaces to the TAP Broker service. Picture a shows a Java Swing user interface for desktop or laptop computers, picture b a Tcl/Tk user interface for PDA, and picture c a Java Swing user interface for very small devices (for example Java enabled cellular phones). All three user interfaces are based on the same interaction acts.

6.2.1 The Java Swing Desktop User Interface

The user interface generated from the desktop customization form provides updated information about all the actions of the trading agent, and about the account and the portfolio. The state of the agent, and its level of activity are also shown, see figure 6a. It can provide a history of transactions in different views (current day, latest week, latest month, and complete history) in a new window. Users can also switch between agents if they own more than one. This user interface is not intended to cover the whole screen, but to be present on the screen while users attend to other tasks.

6.2.2 The Java Swing Small Device User Interface

The user interface generated from the small device customization form shows considerably less information than the desktop user interface. To minimize the window, only the value of the portfolio and the activity

level is shown. The value of the portfolio is color coded, red for downward trend and blue for upward trend, see figure 6c. As for the desktop user interface, the purpose of this user interface is not to use small devices maximal screen resources but to be present and still leave room for other interaction.

6.2.3 The HTML User Interface

The HTML user interface displays all available information about the current agent: transactions, orders, account state, and portfolio content and value. It also provides information about the state and the activity level of the agent. As in the Java Swing desktop user interface, transaction history can be presented in different views (latest day, latest week, latest month, and complete history). Since the list of transactions quickly gets long, the content of the portfolio is presented before the transactions to avoid excessive scrolling. Due to the nature of HTML user interfaces, the information cannot be updated through system push. Updates will be made upon user actions. This means that temporary life cycle of interaction acts is not supported.

6.2.4 The Tcl/Tk User Interface

The Tcl/Tk user interface is designed for PDA use, and thus a smaller screen. To adapt to this, the Tcl/Tk user interface does not show the account state and the portfolio value. A smaller number of transactions are shown, and the buttons for choosing different transaction history views are rendered as menu alternatives in the option menu, see figure 6b.

7 Future Work

In the TAP Broker service, there is a great difference in the amount of information presented in different user interfaces. However, all interaction engines get the same interaction acts, thus the same amount of information, to base their user interfaces on. Thus, in those cases when the interaction engine is running on the device, and the service is running remotely, lots of superfluous interactions are sent to an interaction engine. This could be a problem when network capacity is limited. We will look at ways of server side filtering for those cases to avoid sending interaction acts that will not be used in the generation process.

Adaptation of user interfaces to device features and capabilities need to be combined with service personalization. User preferences must affect

the way services present themselves. Preferences can be collected by letting users set up profiles, or by monitoring user interaction. We believe that customization forms can be used for personalization in UBI. User preferences could be stored in separate customization forms that interaction engines combined with other presentation information when generating user interfaces. Customization forms for personalization would be device and service specific just as the forms created by service providers.

We will also investigate how to handle dynamic resources in UBI. Services that use lots of dynamic media resources, e.g. a service for browsing a video database, might need an extension of our customization form approach to work efficiently for different modalities. One solution could be to handle the choice of media type outside the customization form.

8 Conclusion

We have presented the Ubiquitous Interactor (UBI), a system for development of device independent mobile services. In UBI, user-service interaction is described in a modality and device independent way using interaction acts. The description is combined with device and service specific presentation information in customization forms to generate tailored user interfaces. This allows service providers to develop services once and for all, and still provide tailored user interfaces to different services by creating different customization forms. Development and maintenance work is simplified since only one version of each service need to be developed. New customization forms can be created at any point, thus services can be developed for an open set of devices.

9 Acknowledgements

This work has been funded by the Swedish Agency for Innovation Systems (VINNOVA). Thanks to the members of the HUMLE laboratory, in particular Anna Sandin for help with the implementation of the HTML interaction engine.

10 References

- Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M. and Shuster, J. E. (1999) UIML - an appliance-independent XML user interface language, *Computer Networks*, **31**, 1695-1708.
- Boman, M., Johansson, S. and Lybäck, D. (2001) Parrondo Strategies for Artificial Traders, In *Intelligent Agent Technology* (Eds, Zhong, Liu, Ohsuga and Bradshaw), pp. 150-159.
- Bylund, M. (2001) *Personal Service Environments - Openness and User Control in User-Service Interaction*, Licentiate thesis, Department of Information Technology, Uppsala University.
- Bylund, M. and Espinoza, F. (2000) sView - Personal Service Interaction, in Proceedings of 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology.
- Esler, M., Hightower, J., Anderson, T. and Borriello, G. (1999) Next Century Challenges: Data-Centric Networking for Invisible Computing. The Portolano Project at the University of Washington, in Proceedings of The Fifth ACM International Conference on Mobile Computing and Networking, MobiCom 1999.
- Espinoza, F. (2003) *Individual Service Provisioning*, PhD, Department of Computer and Systems Science, Stockholm University/Royal Institute of Technology.
- Foley, J. D., Wallace, V. L. and Chan, P. (1984) The Human Factors of Computer Graphics Interaction Techniques, *IEEE Computer Graphics and Applications*, **4**, (6), 13-48.
- Lybäck, D. and Boman, M. (2003) Agent trade servers in financial exchange systems, *ACM Transactions on Internet Technology*, (In press.).
- Myers, B. A. (1990) A New Model for Handling Input, *ACM Transactions on Information Systems*, **8**, (3), 289-320.
- Myers, B. A., Hudson, S. E. and Pausch, R. (2000) Past, Present and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **7**, (1), 3-28.

Nichols, J., Myers, B. A., Higgings, M., Hughes, J., Harris, T. K., Rosenfeld, R. and Pignol, M. (2002) Generating Remote Control Interfaces for Complex Appliances, in Proceedings of 15th Annual ACM Symposium on User Interface Software and Technology, Paris, France, 161-170.

Nylander, S. and Bylund, M. (2002) Providing Device Independence to Mobile Services, in Proceedings of 7th ERCIM Workshop User Interfaces for All.

Olsen, D. J. (1987) MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics*, **5**, (4), 318-344.

Olsen, D. J., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P. (2000) Cross-modal Interaction using XWeb, in Proceedings of Symposium on User Interface Software and Technology, UIST 2000, 191-200.

Singh, G. and Greene, M. (1989) A high-level user interface management system, in Proceedings of Conference on Human Factors and Computing Systems, CHI 89, 133-138.

Stephanidis, C. (2001) The Concept of Unified User Interfaces, In *User Interfaces for All - Concepts, Methods, and Tools* (Ed, Stephanidis, C.) Lawrence Erlbaum Associates, pp. 371-388.

Szekely, P., Luo, P. and Neches, R. (1993) Beyond Interface Builders: Model-Based Interface Tools, in Proceedings of INTERCHI'93, 383-390.

Paper 3

Mobile Access to Real-Time Information – The Case of Autonomous Stock Brokering

Stina Nylander, Markus Bylund, and Magnus Boman

Mobile Access to Real-Time Information – The Case of Autonomous Stock Brokering

Stina Nylander, Markus Bylund, Magnus Boman
Swedish Institute of Computer Science

E-mail: {stina.nylander, markus.bylund, magnus.boman}@sics.se

Abstract

If services providing real-time information are accessible from mobile devices, functionality is often restricted and no adaptation of the user interface to the mobile device is attempted. Mobile access to real-time information requires designs for multi-device access and automated facilities for adaptation of user interfaces. We present TapBroker, a push update service that provides mobile and stationary access to information on autonomous agents trading stocks. TapBroker is developed for the Ubiquitous Interactor system and is accessible from Java Swing user interfaces and Web user interfaces on desktop computers, and from a Java Awt user interface on mobile phones. New user interfaces can easily be added without changes in the service logic.

1 Introduction

Users of real-time services often encounter problems with persistent control when they leave their desktop computer. For instance, professional stock traders would like to bring their desktop environment on a laptop, PDA, or cellular phone to meetings as well as to the coffee room (Blomberg, 2001). Non-professional investors, i.e. people that actively manage their savings, would also welcome services for being kept up to date outside the home. Unfortunately, this is seldom possible. In many cases, mobile users lose the service altogether, and in the few cases when it is possible to access the service from a mobile device, the functionality is restricted, or the adaptation to the device is insufficient. The prevailing method for making services available from various devices is by making different versions, as in the case of Web services for WAP. This quickly becomes infeasible as the range of devices grows wider, or when extensive personal customization is necessary.

Development and maintenance work becomes difficult, and version differences increase the risk of error and inconsistencies.

To accommodate real-time mobile access, services have to be designed for multi-device access, and automated facilities for adaptation of user interfaces to different devices should be provided. Our case in point is *TapBroker*, a mobile and stationary push update service, aimed at providing users with notifications on autonomous agents (Maes, 1994), trading on a financial exchange. Trading agents code the preferences of their owners, in our case limited to stock portfolios. While agent trading in theory relaxes the agent owner from monitoring market data, in practice there will be long periods of intense trading, radical market changes, preference changes, and other factors that will contribute to the need for full control. Agent owners will carry with them persistent trading services, and will expect at all times at least one channel of swift and consistent interaction means. In cooperation with OM, the world's largest provider of software for financial exchanges (Sales, 2001), we have implemented a so-called Agent Trade Server (ATS) (Lybäck and Boman, 2003). In addition to our proof-of-concept implementation, we have implemented agents of varying levels of sophistication, and also various services for agent interaction with their owners.

It might be argued that the best and easiest way to create services with multiple user interfaces is to use Web user interfaces. Most devices are capable of running a WWW browser and could thus support WWW interaction. However, Web interaction has several drawbacks. It is user-driven, and even if there are ways to get around that, it is unsuitable for services relying on push data, like *TapBroker*. It is also difficult to control how Web user interfaces are presented to the end-user, cf. (Esler et al., 1999).

We will use the Ubiquitous Interactor (Nylander and Bylund, 2002), which supports information push and the development of services that present themselves differently on different devices. This is done by separating user-service interaction from presentation. User-service interaction is kept the same, and information about how a user interface should be generated on different devices is provided separately. Services can be created once and presentation information for new devices can be specified at any time without causing any changes in the service logic. The possibility to push information from the service to the user interface allows for service-driven interaction, which is useful in other mobile and context-aware applications (Cheverst et al., 2002).

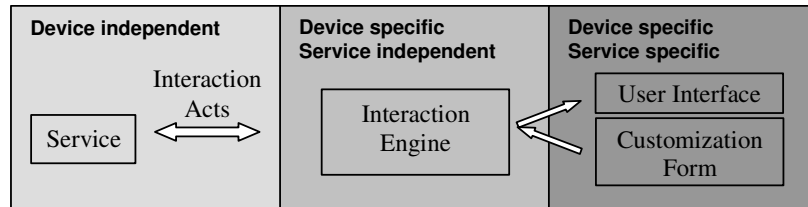


Figure 1: The three layers of specification in the Ubiquitous Interactor. Services and interaction acts are device independent, interaction engines are service independent and device or user interface specific. Customization forms and generated user interfaces are device and service specific.

The following section briefly describes the Ubiquitous Interactor. Section 3 details the TapBroker service, the user interface aspects of which are discussed in Section 4.

2 The Ubiquitous Interactor

The Ubiquitous Interactor (UBI) (Nylander and Bylund, 2002) is a system for developing device independent services. It has three main parts (see Figure 1): interaction acts, customization forms, and interaction engines. In brief, services offer functionality to users encoded in interaction acts, customization forms contain information about how a given service should be presented on a given device, and finally interaction engines generate user interfaces based on the interaction acts and customization forms. This architecture makes it possible to develop a service for an open set of devices, and to add customization forms for new devices whenever desired.

The issue of device independence has been addressed during different periods of computer history. During the early eighties, attempts were made to overcome differences in hardware standard (Olsen, 1987, Wiecha et al., 1990). More recently, the issue has been approached in both the mobile research community (Olsen et al., 2000, Abrams et al., 1999) and in that of universal access (Stephanidis, 2001). However, none of the mentioned systems has been able to address the three main problems device independence, control of presentation, and information push in a satisfying way.

2.1 Interaction Acts

An interaction act is an abstract unit of user-service interaction that contains no information about presentation at all. The UBI concept builds on the assumption that user-service interaction for a wide range of services and devices can be captured with a small set of interaction acts in different combinations. The UBI supports eight different interaction acts: `start`, `stop`, `create`, `destroy`, `input`, `output`, `select`, and `modify`. `start` and `stop` refer to the starting and stopping of services. `create` and `destroy` refer to creation and deletion of service specific objects, `input` is input to the service, `output` is output to the user, `select` is selection from a set of alternatives, and `modify` is modification of service specific objects. `input` is mainly used for data not stored in the service, such as data for navigation operations. `select` is used for similar cases where the range of input is limited to a few alternatives. `create`, `destroy`, and `modify` are mainly used for application-specific data that users can manipulate, for example meetings in a calendar, or avatars in a game.

2.2 Customization Forms

Customization forms provide means for service providers to specify how a service should be presented to end-users. Control of presentation is an important issue in commercial development (Myers et al., 2000) since it is used, for example, for branding. By providing a detailed customization form, service providers have full control over how the user interface will be generated. Customization forms are optional. If there is no customization form, or if the form is not exhaustive, defaults are used to generate the user interface. The main forms of presentation information in a customization form are mappings and media resources. Mappings are links between interaction acts and widgets or other user interface components. Media resources are links to pictures, sounds, or other resources that a particular user interface might need to present an interaction act.

2.3 Interaction Engines

Interaction engines are specific to a device, to a family of devices, and to a type of user interface, but they are service independent. For example, an interaction engine for HTML user interfaces could be used on both desktop and laptop computers, while handheld computers would need a special engine. Each device used for accessing a UBI service needs an

interaction engine installed. In the ideal case, devices would be delivered with interaction engines pre-installed. Devices that can handle several types of user interfaces can have several interaction engines installed. For example, a desktop computer can have interaction engines for both Java Swing user interfaces and Web user interfaces. During user-service interaction, interaction engines interpret interaction acts and customization forms (when they are available) and generate user interfaces for services. Interaction engines are also responsible for interpreting user actions and for sending them back to services and update user interfaces. User interfaces can be updated both on initiative from services and as a result of user action.

3 Implementation

The Ubiquitous Interactor is a working prototype with several interaction engines that handles the full set of interaction acts. Interaction acts are in turn encoded using the Interaction Specification Language (ISL) (Nylander and Bylund, 2002), which is XML compliant. Each interaction act has a unique id, a symbolic name, a life cycle value, a modality, an information holder, and a possibility to carry metadata. Customization forms are also encoded in XML. DTDs for ISL and customization forms can be found in appendix A-C.

Each interaction engine contains modules for parsing ISL and customization forms, as well as generating responses to services from user actions. In an earlier project (Nylander and Bylund, 2002), we have implemented interaction engines for Java Swing, HTML, and Tcl/Tk. All three engines can create user interfaces for desktop or laptop computers. However, the default renderings for the Tcl/Tk interaction engine are most suitable for PDA user interfaces. We have also implemented a calendar service as a sample service (Nylander and Bylund, 2002). It provides basic calendar functions as entering, editing and deleting information, navigating the information, and displaying different views of the information. The calendar service has customization forms for all three interaction engines. Both interaction engines as well as the calendar service are implemented as services in the sView system (Bylund, 2001, Bylund and Espinoza, 2000), to take advantage of the user interface handling features in sView. An sView service is a collection of java class definitions and resources packaged in a standard jar-file that can be loaded and executed in sView (cf. `sview.sics.se`).

4 The Tapbroker Service

The TapBroker notification service gives users feedback on the actions of their autonomous trading agents running on the Agent Trade Server (ATS). Due to security constraints, agents cannot be accessed from outside the ATS during run-time. The TapBroker has access to the XML-format agent logs and can thus provide information about the actions of agents. Users register their agents with TapBroker, which connects to the ATS and subscribes to the relevant log data. The primary source of input data to the TapBroker is thus the agent log. The ATS is pushing log data to the TapBroker, which in turn updates the user interface. This means that most of the changes in the user interface will not be user-driven but service-driven. This is important for a service like TapBroker. Agents buy and sell stocks with users' money, and it is very important that users get updated information every time they access the service. This cannot depend on users refreshing the user interface.

TapBroker gives feedback on the actions of the agents and also some information about their state. The service shows the transactions performed by agents, the content of agents' stock portfolios, the amount of money that agents have available, and the value of their portfolios. The current intentions of agents are shown through the active buy or sell orders. It also shows the agent state (active or shut down) and computes an activity level based on the number of transactions performed during the last 30 minutes. Users can switch between agents, and delete and register new agents to the service. They can also choose the amount of transactions to be shown: those made in the last day, the last week, the last month, or a complete transaction list. Since agents currently do not log the reasons or motivations for their actions, TapBroker cannot give any explanations to agent transactions.

The service can handle multiple agents and can be accessed from three different types of user interfaces: an HTML user interface via a Web browser, a Java Swing user interface from desktop or laptop computer, and a Java Awt user interface from an Ericsson P800 mobile phone. Besides server access, all one needs to run TapBroker is sView and the Ubiquitous Interactor prototype, both publicly available (see sview.sics.se).

5 User Interface Design

The different user interfaces to the TapBroker service have been designed in several cycles, in which both stock traders and researchers were consulted. First, unstructured interviews were conducted with a small number of non-professional traders. They were asked to think about what feedback they would be interested in if they had agents trading on their behalf, in what way they wanted it presented, what devices they would be interested in using to access the information, and whether or not they were interested in mobile access to the information. They were also asked which kind of information that was necessary and which was optional. All traders agreed that the transactions were the most important information, whereas active orders, portfolio content, available amount of money, and portfolio value can be sacrificed in a user interface where screen space is scarce. We also conducted an interview with a professional trader, in order to informally investigate if there were any obvious differences in requirements, which indeed there were. In a professional setting, a motivation for each transaction of an agent, and an indicator of the level of performance for the agent were ranked as important. All traders agreed that mobile access was necessary. A first set of user interface sketches for desktop computer and PDA was created based on these interviews.

Second, the traders were asked to comment the first sketch of the user interfaces and suggest improvements. Preferences again differed between the professional trader and the non-professional traders. The professional traders preferred the sketches that showed the most information, and did not see window size as a problem. The non-professional traders preferred a small window that can be visible on the screen all the time, even if less information was presented. The most probable reason for this difference in preferences is that to professional traders, trading is a main activity and they perform many transactions every day. To non-professional traders, trading is a side activity and the number of performed transactions is low. In a sense, this difference will not persist in the case of trading agents. An agent will trade according to its strategy and the market conditions; regardless of how much attention it gets from its owner.

Third, two researchers in HCI were asked to comment the improved sketches of the user interface. They suggested a state indicator (showing if the agent is active or shut down), and an indicator of activity level (showing the number of transactions per 30 minutes). Three of the resulting user interfaces can be viewed in figures 2 and 3.

The TapBroker service has customization forms for three different user interfaces: Java Swing, HTML, and Java Awt.

5.1 The Java Swing Customization Form

The Java Swing user interface provides all the available information from the agent. The main part of the user interface is devoted to performed transactions and portfolio content, including additional information about the stocks in the portfolio such as the current buy and sell price. The user interface also shows the agent's id, state and activity level, account information, and information about submitted orders. The state of the agent is showed as a play/paused icon, and the activity level as a progress bar. Users can switch between agents in the pull down menu "agent", and add or remove agents in the pull down menu "options", see Figure 2.

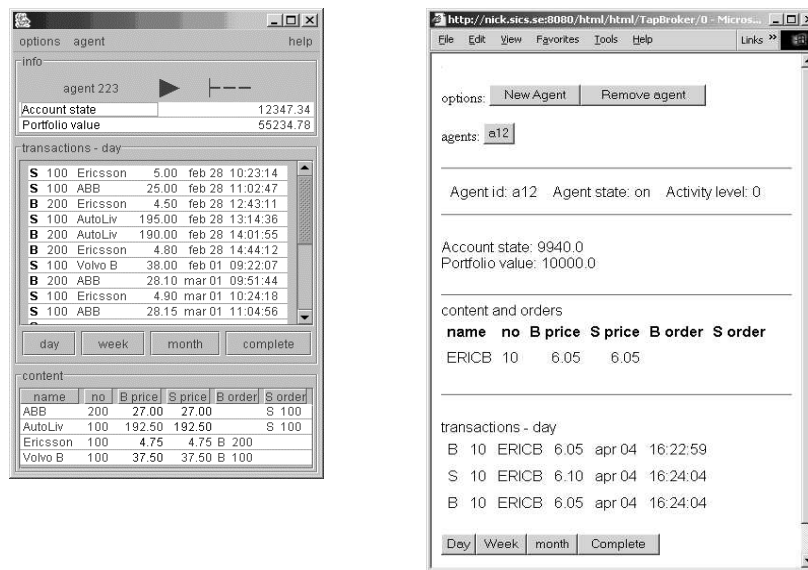


Figure 2: The Java Swing user interface (left) and the HTML user interface of the TapBroker service.

5.2 The HTML Customization Form

The HTML user interface provides the same information as the Java Swing user interface but presented in a slightly different way. The pull

down menus for options and switching agents are presented as sets of buttons. Each operation is presented as a button instead of a menu option. The agent state and the activity level are displayed as text strings. The portfolio content is presented before the transactions to avoid excessive scrolling since the number of performed transactions is likely to be much larger than the number of different stocks in the portfolio, see Figure 2.

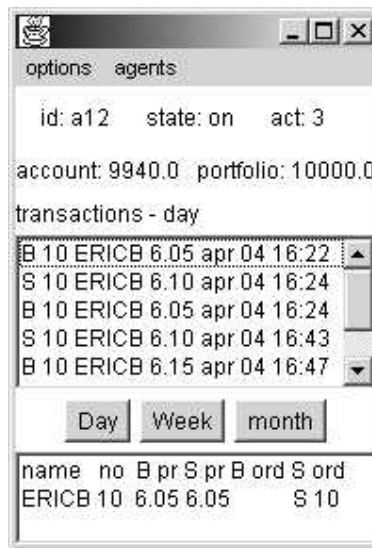


Figure 3: The Java Awt user interface of the TapBroker service.

5.3 The Java Awt Customization Form

The Java Awt user interface is designed to fit a small screen and is thus using shorter names for the different values presented, and sometimes also a shorter format on the value itself (for example no seconds shown in the time stamp of the transactions). The layout of the user interface is also more compact than in the other two, see Figure 3.

6 Conclusions

We have described TapBroker, an implemented service accommodating real-time mobile access, and facilitating automated adaptation of user interfaces to different devices. It provides continuously updated trading

agent information by pushing it to the user. The service is available from different devices and presents itself with device specific user interfaces, using the push and the device independence features of the Ubiquitous Interactor system. TapBroker shows that it is possible to create device independent services that rely on information push, which is an important fact for example for the development of mobile context-aware applications. The TapBroker service will next be made available to the group that are currently developing agents for Agent Trade Server execution, see (Lybäck and Boman, 2003), to allow for further usability testing.

7 Acknowledgement

This work has been funded by the Swedish Agency of Innovation Systems (VINNOVA) through the projects TAP and ADAPT.

8 References

- Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M. and Shuster, J. E. (1999) UIML - an appliance-independent XML user interface language, *Computer Networks*, **31**, 1695-1708.
- Blomberg, J. (2001) *Narratives and Performance - the Case of Stock-brokering*, SSE/EFI Working Papers series in Business Administration, 2001:2, Stockholm School of Economics.
- Bylund, M. (2001) *Personal Service Environments - Openness and User Control in User-Service Interaction*, Licentiate thesis, Department of Information Technology, Uppsala University.
- Bylund, M. and Espinoza, F. (2000) sView - Personal Service Interaction, in Proceedings of 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology.
- Cheverst, K., Mitchell, K. and Davies, N. (2002) Exploring Context-aware Information Push, *Personal and Ubiquitous Computing*, **6**, (4), 276-281.
- Esler, M., Hightower, J., Anderson, T. and Borriello, G. (1999) Next Century Challenges: Data-Centric Networking for Invisible Computing. The Portolano Project at the University of Washington, in Proceedings of

The Fifth ACM International Conference on Mobile Computing and Networking, MobiCom 1999.

Lybäck, D. and Boman, M. (2003) Agent trade servers in financial exchange systems, *ACM Transactions on Internet Technology*, (In press.).

Maes, P. (1994) Agents that Reduce Work and Information Overload, *Communications of the ACM*, **37**, (7), 31-40.

Myers, B. A., Hudson, S. E. and Pausch, R. (2000) Past, Present and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction*, **7**, (1), 3-28.

Nylander, S. and Bylund, M. (2002) Providing Device Independence to Mobile Services, in Proceedings of 7th ERCIM Workshop User Interfaces for All.

Olsen, D. J. (1987) MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics*, **5**, (4), 318-344.

Olsen, D. J., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P. (2000) Cross-modal Interaction using XWeb, in Proceedings of Symposium on User Interface Software and Technology, UIST 2000, 191-200.

Sales, R. (2001) First Europe, now the world, *Wall Street and Technology online*.

Stephanidis, C. (2001) The Concept of Unified User Interfaces, In *User Interfaces for All - Concepts, Methods, and Tools* (Ed, Stephanidis, C.) Lawrence Erlbaum Associates, pp. 371-388.

Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. (1990) ITS: a Tool for Rapidly Developing Interactive Applications, *ACM Transactions on Information Systems*, **8**, (3), 204-236.

Paper 4

Evaluating the Ubiquitous Interactor

Stina Nylander

Evaluating the Ubiquitous Interactor

Stina Nylander
Swedish Institute of Computer Science
SICS Technical Report T2003:19
E-mail: stina.nylander@sics.se

Abstract

This paper is composed of two parts. The first one gives a review of the design iterations for the Ubiquitous Interactor prototype. The second part makes a primary evaluation the concepts and the implementation, describes a small pilot study and gives some pointers on how the future evaluation will be conducted.

1 Introduction

Other parts of this thesis have presented the design and implementation of the Ubiquitous Interactor (UBI). The purpose of this paper is to evaluate the current status and determine to what degree UBI is fulfilling its goals.

Two main evaluation methods have been used in the design and development of UBI. Firstly, the original design has been refined through iterative experiments with different applications and device technology. Secondly, the utility of UBI as a development tool has been evaluated through a minor user study. The user study has only been performed on the third version of UBI, since previous versions were judged too immature to support third-party development. It also only covered development of a new user interface for an existing service, not service development. In subsequent versions of the tool, user studies should form an integral part of the development cycle and include service development. In the last section of this paper, we discuss some methodological issues for user involvement in the iterative design of a development tool.

2 Iterative Design

The purpose of iterative design is to find problems, errors and potential improvements in a systems design at a stage in the development process where it still is possible to correct the errors and make the necessary improvements. By evaluating the system in each iteration cycle, valuable feedback for the design process is gained not only on the implementation of design ideas, but also on areas that cannot be tested before the system is partly implemented, such as acceptance and user performance.

The Ubiquitous Interactor (UBI) has been designed in three iterations so far, with a new version of the system as a result of each cycle. The input to the design process has been of three main types: service analysis, analysis of device and interface technology, and informal user studies. We have foremost used services for UBI as evaluation tools. In the service analysis we have used both scenarios, commercial applications and services implemented for the system, and analyzed how well the interaction acts reflect the user-service interaction and the character of the services. We have also compared services in terms of basic features and interaction. Modules for handling different types of user interfaces on different devices have been developed and added to the system, and user interfaces for services have been created to evaluate how well the approach can be extended to new technology. The third version of the system has been the subject of a pilot user study where students developed user interfaces for an existing service.

3 History of the Ubiquitous Interactor

Our interest and need for device independent services are results of our previous work with the next generation electronic services in the sView project (Bylund, 2001, Bylund and Espinoza, 2000). sView was developed to face the new service-based computing, where services are not considered as applications installed on individual devices but as sets of functions manifested when needed (Espinoza, 2003). sView provided service access from different devices, but service providers had to implement each user interface, which required great implementation and maintenance efforts. This is not a working approach in a realistic setting. To keep up with the large and changing set of available devices, we need to find methods that support simple implementation and maintenance of services without losing the uniqueness of each type of device. This is what we set out to solve with the Ubiquitous Interactor.

3.1 Ubiquitous Interactor version 1.0

The basic assumption behind UBI is that it is possible to describe user-service interaction for a wide range of services with a fairly small set of description units. Such a description could be combined with device specific presentation information, and used to generate different user interfaces for different devices. Since we wanted a device and modality independent description, we chose user-service interaction as level of description to keep the focus on what the user is doing instead of how it is done.

The concept of *interaction acts* is used to describe the user-service interaction. Interaction acts are abstract units of description that contain no information about presentation. The abstract description of the user-service interaction is device and modality independent, and can be combined with different presentation information to create device specific user interfaces. The presentation information is specified in *customization forms* that are device and service specific.

3.1.1 Interaction acts

The first set of interaction acts was identified through analysis of existing services and applications. We looked at functionality and user-service interaction in services on the Web, such as ticket reservation services for trains and movie theatres, telephone services such as bank services and train time tables, and a desktop home care planning tool. Live face-to-face instructions were also studied informally. The first design was deliberately minimalistic to investigate to what extent a very small definition would still provide sufficient support for efficient device-independent application development. The initial set had four members: `input`, `output`, `select` and `confirm`. `input` was defined as input to the system made by the user, `output` as output from the system to the user, `select` as selection from a finite set of alternatives, and `confirm` as confirmation of entered user data. Each interaction act had a string of basic information that could be used in the presentation. It was also possible to assign a symbolic name to all interaction acts, and to attach metadata to them. The `select` and `confirm` interaction acts also contained a set of alternatives. Interaction acts could be grouped using the `isl` tag, and groups could be nested to provide more complex user interfaces. Groups of interaction acts have the same information attached to them as individual interaction acts. Interaction acts are encoded in an XML compliant language, the Interaction Specification Language. A first

grammar for ISL, also called a Document Type Definition (DTD) (Bray et al., 2000), for the Interaction Specification Language was created, see appendix D. Listing 1 shows an example of a group with an `output` and a `select` interaction act.

```
<isl>
  <name>example</name>
  <string>Test application</string>
  <output>
    <name>text</name>
    <string>Item 2</string>
  </output>
  <select>
    <name>listBrowse</name>
    <string>List of items</string>
    <alt>
      <name>alt</name>
      <string>Previous</string>
      <retVal>0</retVal>
    </alt>
    <alt>
      <name>alt</name>
      <string>Next</string>
      <retVal>1</retVal>
    </alt>
  </select>
</isl>
```

Listing 1: A group of interaction acts, containing an `output` and a `select` interaction act.

3.1.2 Customization Forms

The first version of UBI allowed mapping of interaction acts to user interface components, but only in a very simple way. Customization forms were implemented as hashtables with the symbolic name of an interaction act as key, and the class name of the user interface component as value. This made it possible to map interaction acts to any kind of user interface component, but also meant that the customization forms actually were a part of the device specific interaction act interpreter. Since customization forms were implemented in java, they were still downloadable and could be specific for each service, but any change in the form required recompilation. The solution also meant that the interaction act interpreter had to be a Java program, and that it was difficult to update the customization forms at run-time.

3.1.3 Interaction Engines

Interaction engines handle the parsing of the interaction acts, the lookup in the customization forms, and the generation of user interfaces based on interaction acts and presentation information from customization forms. Each interaction engine handles one type of user interface for a given device or family of devices. Three interaction engines were developed for the first version of UBI: one for Java Swing user interfaces, one for std/IO user interfaces, and one for HTML user interfaces. All interaction engines were implemented in Java.

3.1.4 Services

To evaluate the usefulness of the set of interaction acts in designing applications, we developed a calendar service. The service supported basic calendar operations as entering, editing and deleting information, navigating the information, and displaying different views of the information. We developed customization forms for Java Swing, std/IO, and for HTML user interfaces. Due to time restrictions, the HTML user interface only supported a subset of the calendar functions: navigating the calendar information and displaying different views of the information.

3.1.5 Evaluation

For the evaluation of the first version of UBI we used the calendar service.

The functionality of the calendar suggested some modifications to the interaction acts. A calendar is based on input, editing and deletion of information, and that did not seem unique for calendars but could extend to other types of services. The initial set of interaction acts only provided the `input` act for this. The `input` category was diversified into four interaction acts, `input`, `create`, `destroy`, and `modify`. The new interaction acts were used to handle application specific data (in the case of the calendar, meeting information), and the `input` interaction act was kept and used for input of transient data (for example navigation data).

We concluded that the customization forms needed to be more advanced. In the first version we could only create mappings based on the name of interaction acts. There was no possibility to create mappings based on the type of interaction act, or to create a mapping for a set of interaction acts.

The calendar and the implemented interaction engines showed that it is possible to generate three different user interfaces from the same

interaction acts. Even if that was a small-scaled evaluation, we believed that it supported further work on device independent development using interaction acts and customization forms.

3.2 Ubiquitous Interaction version 2.0

In the second version of UBI, new interaction acts were introduced, new information was associated with interaction acts, and customization forms were improved as a result from the evaluation of the first version. An analysis of computer games was made to further inform the design (Nylander and Waern, 2002). The calendar service and the interaction engines were updated to conform with the changes (Nylander and Bylund, 2002a, Nylander and Bylund, 2002b).

3.2.1 Analysis of Computer Games

The analysis of computer games revealed some characteristic game features relevant to this work. Computer games are closely tied to a platform, to be able to provide an as rich interaction experience as possible. When they are available on several platforms, considerable changes in the application often have been made. Games are proactive, and rely on pushing data and information to the user. They are often described as object spaces containing many individual objects, for example avatars and artefacts residing in a ‘game world’.

The fact that games and user-game interaction are closely tied to a platform and that they rely on information push make them an excellent domain for UBI. The precise goal of UBI is to provide device specific user interaction by tailoring user interfaces to devices, and information push is a fundamental feature. Modeling an object space using interaction acts, however, needs a more elaborate set of interaction acts than the initial one. We found it useful to introduce an additional level of abstraction that apply to an object space and that offers applications and users acts that manipulate the objects and control their life cycle: `create`, `destroy`, `modify`, and `move` (together with the original `select` interaction act). `create` and `destroy` handles creation and deletion of objects in the space, while `modify` offers means to modify objects. This mapped nicely on the analysis of the calendar service that was made to analyze the initial set of interaction acts. `move` handles relocation of objects or sets of objects to new positions in the space. We also introduced `start` and `stop` as interaction acts on object space level to handle the interaction session.

Game user interfaces use a lot of media resources. Large parts of the user interfaces are generated from resources that are distributed with the application. This means that customization forms must offer possibilities to link media files to the presentation of interaction acts.

To support pro-active computer games with a dynamic object space, we found that interaction acts need to have a pre-defined life cycle. A life cycle determines the availability of the interaction act in a user interface. We identified three different life cycles: *temporary*, *confirmed* and *persistent*. Temporary interaction acts are presented once during a predefined short time in the user interface, for example a welcome screen. Confirmed interaction acts are presented until users have performed a given action, for example entered a login name. Persistent interaction acts are available during the whole interaction session, or until the application removes it, for example an avatar in a game or a help function in a telephone bank service.

To handle dynamic user interfaces like games that handle many objects and use a lot of media resources, it is important with a cache function. This would alleviate the user interface engine from generating the same part of a user interface more than once. Only updated parts are re-generated, and the rest is retrieved from the cache.

3.2.2 Interaction acts

The second set of interaction acts had nine members: `input`, `output`, `select`, `modify`, `create`, `destroy`, `move`, `start` and `stop`. `input`, `output` and `select` are defined as in the first version. `modify` handles modification of entered application specific data and includes the function of the `confirm` interaction act in the first version. `create` and `destroy` inserts and removes application specific objects (for example meetings in the calendar service), while `move` changes the position of such an object. `start` and `stop` handles the interaction session with the service. The `move` interaction act was never implemented since it turned out to be very difficult to represent positions without taking dimensions into account. Should a two dimensional representation be used as a generic position and be mapped to three dimensions in a 3D user interface? Or the other way around? Position can also be relative or absolute. We preferred to leave the position problem outside of UBI for a start. As in the first version, interaction acts have symbolic names, default information holders, and metadata can be attached to them. In this version, interaction acts also have a life cycle value and a modality value.

The life cycle value can be *temporary*, *confirmed* or *persistent*, and specifies how long a user interface component based on the interaction act should be available in the user interface. The default value is persistent. The modality value can be true or false and specifies if a user interface component based on the interaction act should lock other interaction acts while available. The default value is false. Each interaction act can also belong to a customization form group, which allows it to inherit presentation resources associated with that group. A new DTD for encoding interaction acts was created, see appendix B.

3.2.3 Customization Forms

The second version of customization forms provides two kinds of presentation information: *directives* and *resources*. Directives map interaction acts to user interface specific components. Resources are media files (e.g. pictures or sounds) that are associated with the interaction act and can be used in generating interface components. It was obvious both from the calendar service and the computer game analysis that it is important to provide means for using external media resources. Directives and resource links can be associated with a group of interaction acts or a type of interaction acts, as well as to all interaction acts with the same name. Links to groups affect all interaction acts belonging to the group; links to types affect all interaction acts of the given type.

In this version of UBI, customization forms are specified as XML files that are parsed by the interaction engines with a special customization form parser. The customization form can be reset during run-time and thus change the appearance of a service completely without restarting it. A DTD for customization forms has been created, see appendix C. Furthermore, since customization forms are XML documents the device specific interaction engine can be implemented in any technology, whereas the interaction engines in the first version was Java only. Listing 2 shows an example of a directive mapping in a customization form.

```
<element name="output">
  <directive>
    <data>
      se.sics.ubi.swing.OutputLabel
    </data>
  </directive>
</element>
```

Listing 2: A directive mapping in a customization form.

3.2.4 Interaction Engines

In this version of UBI, the interaction engine has been split up into two parts: interaction engine and user interface manager. This means that interaction engines handles the parsing of interaction acts and customization forms, and the generation of user interface components, while the user interface manager handles the details of presenting the user interface. For a Java Swing user interface the interaction engine would generate widgets in a panel that would be sent to the user interface manager. The manager would then create a window, add the panel and present the window on the screen. This partition was made to comply with the sView system (Bylund and Espinoza, 2000, Bylund, 2001), where services register their user interfaces with user interface managers, to facilitate a future integration of the Ubiquitous Interaction in sView.

The cache function was not implemented in the engines. This means that in the calendar the whole user interface was regenerated upon each user action. Only two life cycles values were supported in the interaction engines in this version: confirmed and persistent.

3.2.5 Services

The calendar service was updated to the new version of interaction acts and interaction engines. Customization forms for Java Swing and HTML were written for the new version of customization forms. This version of the HTML user interface supported all calendar functions. We also created a second customization form for Java Swing that used media resources.

No game services were implemented due to lack of resources. HTML user interfaces are not very suitable for interactive games, so we would have needed to implement an interaction engine for another user interface type to be able to present the game service with multiple user interfaces (Swing and the new type). This was not possible at the time.

3.2.6 Evaluation

To evaluate the second version of UBI we used the calendar service with three customization forms that generated fully functional user interfaces. This showed that the new set of interaction acts was functional as well as the new customization forms.

3.3 Ubiquitous Interaction version 3.0

In version 3 of UBI services and interaction engines are implemented as sView services. Two new interaction engines were developed, one for Tcl/Tk (Paper 2) and one for Java Awt (Paper 3). A stockbroker service was also developed (Paper 3).

However, further evaluation of the system was needed and thus more services. The main reason for making a third version of UBI was to integrate the system with the sView (Bylund, 2001, Bylund and Espinoza, 2000) system developed in an earlier project at SICS. In sView, UBI would be placed in a better context for service development. For example we could take advantage of sView features such as inter-service communication and user interface handling.

3.3.1 Interaction acts

The set of interaction acts stayed the same from version 2 to version 3. The only addition was that a unique id was assigned to each interaction act to make it easier for services to associate user actions to a particular interaction act.

3.3.2 Interaction Engines

In version 3 of UBI, all interaction engines were implemented as services in the sView system. Two new interaction engines were created, one for Tcl/Tk and one for Java Awt. The Java Awt interaction engine is implemented in Personal Java to be able to execute on cellular phones (in our case an Ericsson P800). The Tcl/Tk engine has default mappings designed to generate user interfaces suitable for PDAs, and the Java Awt engine has defaults designed for mobile phones.

The interaction engines were also updated to support partial updating of user interfaces. In version 3, services only send changed interaction acts to interaction engines, and the components based on those interaction acts are the only ones affected by the change. This alleviates the interaction engines from parsing unchanged interaction acts, and updating unchanged parts of user interfaces. It also supports the whole life cycle of interaction acts, which was not fully supported in previous versions.

sView provides support for inter-service communication that allows interaction engines to handle several services simultaneously, without any extra implementation work.

3.3.3 Services

The calendar service was updated for version 3 of the Ubiquitous Interactor, and integrated as a service in sView. A customization form for the Tcl/Tk interaction engine was added to the forms for Java Swing and HTML.

A new service was implemented for version 3, the TapBroker stockbroker service (Paper 3). Agents residing on an Agent Trade Server (Boman and Sandin, 2003) trade stocks on the behalf of the user and TapBroker provides feedback on the actions of agents. Customization forms were implemented for Java Swing, HTML and Java Awt.

3.3.4 Evaluation

When the TapBroker service was implemented, the set of interaction acts was stable enough to handle a new type of service without changes. We had no problems describing the TapBroker with the interaction acts at hand.

At this point, we had two different services with three user interfaces each. We believe that this supports the fact that it is possible to create services with multiple user interfaces using interaction acts and customization forms.

The integration of UBI into sView also shows that the approach with interaction acts and customization forms is not dependent of a certain implementation to work.

4 Design Evaluation

Services and interaction technology has affected different parts of the Ubiquitous Interactor (UBI). The identification of interaction acts has been based entirely on analysis of service interaction, while the implementation of the prototype has been influenced both by different interaction technologies and technical features of the services.

4.1 Comparison of sample services

The TapBroker service and the calendar service are similar in the way that they are both information services. However, they differ in some important aspects:

- information sources,
- user interaction,
- user driven vs. service driven, and
- dynamic vs. stable number of interaction acts.

The only information source for the calendar is user input, as long as a user has not entered any information the calendar will be empty. In the case of the TapBroker, the only information input from the user is the registration or deletion of an agent. All other information comes from the Agent Trade Server (Boman and Sandin, 2003).

In the calendar service, users are not only responsible for input of all the information content, they can also change the way the whole information content is displayed, and navigate the information. The TapBroker is designed to always display as much information as possible, and only provide different views on the transactions. The other parts of the user interface only update their values and cannot be affected by user actions. The only actions for users to perform except scrolling the transactions, are adding or removing an agent, and switching between agents.

The calendar service is purely user-driven. Since the only reason for updating the user interface is that a user has added, edited or deleted information, the user interface is only updated on user actions. The TapBroker is both service-driven and user-driven. TapBroker updates the user interface each time information about the current agent has changed, which is based on log information from the Agent Trade Server and thus service-driven changes. It also updates each time the user adds or deletes an agent, or switches agents, thus user-driven changes.

The calendar service presents itself with different numbers of interaction acts depending on the current view of the information. The service sends out one `output` interaction act for each day to be displayed to the user, which means one for a day, seven for a week view and 28-31 for a month view. This means that the update procedure must be able to handle a changing number of interaction acts and user interface components. The TapBroker service presents itself with an unchanging number of interaction acts. This means that updates only concern the information content of the interaction acts, not the layout of the user interface.

These different service characteristics have highlighted desirable features and influenced the implementation of UBI. They have not influenced the identification of interaction acts.

4.2 Comparison of Interface Technology

The interaction technology used in UBI so far only concerns graphical user interfaces. We have worked with several widget toolkits for user interface creation (Java Swing, Tcl/Tk, and Java Awt), and Web-based user interfaces (HTML). They differ on some important points:

- update of the user interface,
- user interface cache, and
- connection between widget and user action.

The widget-based technologies that we have used are event-based and can update their user interfaces directly on changes in the code. Web-based user interfaces are only updated when the browser sends a request to the Web server. This can be automated, but is often left to the user who has to push the reload button.

In Java, widgets and user actions associated with widgets are handled with events, and the widget and the code handling user actions reside in the same object. Listeners direct user actions to the correct object and appropriate responses are generated. Tcl/Tk is also event-based, and each widget has a command option that is evaluated upon user actions. No external means like widget or object id are needed for the connection in these cases. In Web-based user interfaces, components are generated by the browser from the markup code, and user actions are returned to the server as POST or GET requests. This requires an explicit mapping between actions performed on interface components and service responses using for example a component id.

The different Java and Tcl/Tk user interfaces can be described as implementing their own cache function since window handlers work as a built-in cache. Single widgets can be updated without regeneration of others, and an application can be split over separate windows that do not affect each other. HTML user interfaces are page-based and in that sense transient. The whole page is regenerated each time the user interface is updated from the server side. This can sometimes be handled by dividing

the user interface into several frames that can be updated individually. However, too many frames complicate both user interface handling and user interaction.

The differences between user interface technologies revealed in the analysis above have affected the design and the implementation of the UBI.

The fundamental principle of information handling and user interface update in UBI is push. Services can initiate changes in the user interface by pushing information to the interaction engine. Not all user interface technologies support this, and will either provide user interfaces based on information pull, or need additional features to “fake” information push. For the HTML user interfaces in UBI, we have solved this by forcing the Web browser to make a new GET request on each update of the user interface.

The interaction engines updates user interface components based on changed interaction acts. In the Java user interfaces this is handled through simple updating of individual widgets. In HTML the interaction engine caches the code and makes partial updates on the cache. When a GET request is made from the browser, a copy of the cached code is sent. The Tcl/Tk interaction engine uses the same approach as HTML due to the organization of the interaction engine. The Tcl/Tk interaction engine generates user interfaces tailored for small devices, and to minimize the resources needed on the target device the interaction engine is not executing on that device. The engine executes remotely and sends the generated Tcl/Tk code to a small client on the target device that generates the user interface. Updates are made on a cached copy of the code that is resent to the target device client for each update. This is an implementation decision made at the expense of bandwidth. Minimizing the computing on the target device means sending more code through the network.

In the Java and Tcl/Tk user interfaces generated in UBI, the connection between widget and user action is handled solely using events. No explicit means as widget id is needed. The HTML interaction engine depends on an id for each user interface component to be able to map browser requests to individual interaction acts. This caused the addition of a unique id for each interaction act.

Interaction technology has mostly affected the implementation of UBI. Differences between technologies require different implementations to provide the same functions, e.g. partial update sometimes requires an extra cache function to work. The only issue where interaction technology has affected interaction acts and the parameters associated with them concerns the individual id. That was added when we implemented the interaction engines for HTML. The set of interaction acts has not been affected by interaction technology so far.

4.3 Evaluation of Interaction Acts

The interaction acts have been identified through analysis of user-service interaction, and not through analysis of technical features of services (cf. section 4.1).

The initial set of interaction acts had four members but was subsequently modified, mainly as a result of analyzing computer games and implementing the calendar and the TapBroker service. The implementation of several different interaction engines had no effect on the set of interaction acts.

Analysis of the calendar service and the computer games domain suggested some modifications of the set. The `create`, `destroy`, `modify` and `move` interaction acts were added to the set. The `move` interaction act has not been implemented in any of the versions of the Ubiquitous Interactor, due to problems with representing position in different spaces (e.g. translating from 2D to 3D). The `start` and `stop` interaction acts were introduced to manage the life cycle of a service. In the current implementation, where services and interaction engines are implemented as sView services, the service life cycle is handled by sView.

The calendar service and the TapBroker service show that it is possible to describe user-service interaction in a device independent way using our set of interaction acts.

4.4 Evaluation of Customization Forms

The evaluation of customization forms was mainly done from the software engineering perspective. The format of the customization forms was determined entirely by this. The need for media resource mappings was identified through service and user interface analysis. In

implementing several different interaction engines we could conclude that our customization forms were rich enough to support a wide range of presentations.

In the first version of UBI, customization forms were specified as simple Java hash tables, and they only provided one form of mapping: the linking of the name of the interaction act to a user interface component. This proved to be too restraint, and it was also cumbersome to recompile the forms after each change.

In the current version, customization forms are specified as XML files and parsed by the interaction engines customization form parser. Customization forms can be reset during run-time and thus change the user interface of a service without restarting it. There are two kinds of information in the customization form, directives and resources, and the information can be specified on three different levels: group, type and name. This allows for mappings between names or types of interaction acts, and user interface components. Mappings can be based on both name and type in combination, and also be created for a specified group of interaction acts. The three levels of specification make it possible to specify how groups or sets of interaction acts should be presented without creating a mapping for each interaction act.

The different customization forms that have been developed for the calendar service and the TapBroker show that it is possible to create different user interfaces using interaction acts and customization forms. Resources can handle different media types and could for example provide recorded sound for speech user interfaces.

5 Potential Approaches to User Evaluation

The Ubiquitous Interactor (UBI) has a twofold goal: facilitate development of services with multiple user interfaces, and provide end-users with more device-adapted user interfaces and better possibilities to combine devices and services. Evaluating these two goals means evaluating two different types of requirements: system requirements that concern implementation and performance issues, and utility requirements that concern for example acceptance and large-scale effects. Facilitating development includes both types of requirements, while providing better

user interfaces and more choices to end-users mostly concerns utility requirements.

We believe that UBI fulfils the system requirement for facilitating development. It provides concepts that separate functionality from presentation, and the prototype proves that it is possible to implement. Service designers and developers use interaction acts to describe services, and user interface designers and developers use customization forms to create presentations for services. Due to this separation, service developers do not need to know what user interfaces will be available for the service, and user interface developers do not need to know any implementation details about service, such as programming language or communication protocol, or what other user interfaces there are for the service. However, user interface developers still need to be skilled in the language the user interface is created in. Developers that do not know Java Swing cannot create a Swing user interface for a service in UBI, unless the user interface can be created only with defaults provided by the interaction engine.

The utility requirements of the first goal concern how useful the concepts of UBI are to developers compared to other tools, how easy it is to develop services using interaction acts and customization forms and how much time it saves, compared to using other methods. The utility requirements of the second goal concerns to what degree services developed with UBI provide end-users with better user interfaces, or offer more flexibility in combining devices and services.

One of the challenges of creating a system like UBI, which partly is a conceptual tool, is to find ways to explain and present the concepts of the system. Service developers need good explanations of interaction acts to be able to use them, and user interface developers need to understand customization forms. We also need to provide a connection between the two teams. User interface developers need to understand the functions and the structure of a service to be able to create user interfaces for it. This can be made by written presentations of service functions, maps of the interaction act structure, or other documentation. It can also be complemented by functions in the UBI system that display the interaction act structure during development.

An important measure of the utility of a system is how easy it is for new users to work with it, in this case develop services and customization forms. We have not conducted any large scale testing of this, but we have

had two experiences that provide some insight. The first was when an external person came in to help with the HTML interaction engine and the HTML customization form for the calendar service. In that case, a person who had not been involved at all before in the development of the Ubiquitous Interactor system with a short introduction and some code from the Java Swing interaction engine and customization form had no problem understanding the system and producing working modules. The second was a pilot study we conducted where students developed customization forms for the TapBroker (see below).

The second goal of UBI, to facilitate the lives of end-users by providing them better possibilities to combine devices and services, and also more device adapted user interfaces, is entirely about utility requirements. Evaluating to what degree UBI fulfils this goal is very difficult on a research stadium. At this point we can only establish that UBI provides means to fulfil the goal. To evaluate if UBI really could change the end-users situation, it would need to be adapted as a commercial development philosophy and method. We can also compare it to larger projects as the development of a programming language, or even the emergence of the Web and HTML. They depend on acceptance from a large community of users outside their original inventors to gain their full power and show their utility. Before that has happened, it is very difficult to evaluate their full contribution to the development process.

As we said in the introduction, further design iterations for UBI should include user studies. This means that studies need to be conducted involving both service development in different domains, and customization form development for different services and different user interfaces. Studies of this kind are cumbersome to conduct since they involve many participants, and the participants need a lot of time to get familiar with the system and then work with it before they can provide any feedback. Still, these studies can never prove that the soft requirements are fulfilled. They can however collect more and more supporting evidence. The final evaluation of the soft requirements will take place when the system spreads out of the initial setting, and is getting used by a large community outside its original creators.

5.1 Pilot Study

We have conducted a small pilot study where we let students try to develop customization forms. The goal of the study was to find out if people that had not been involved in the development of UBI had problems understanding the concepts, and to collect information about how a larger study should be conducted. We used the TapBroker service and decided to let the participants create a customization form for Java Swing, since we knew that they were familiar with Swing.

Four students participated in the study. They were all doing their Master Thesis at SICS. During the study we made them work in pairs to force them to articulate their thoughts and problems so that we could capture them. They were informed that the focus of the study was on the system and not on their own performance.

The students were introduced to UBI, the purpose of the system and the concepts used (interaction acts, interaction engines and customization forms). They also got a description of the service they were going to develop a customization form for (the TapBroker) which included a map of the interaction acts of the service. We provided them with templates for mappings in customization forms and templates for user interface widgets. The TapBroker Swing user interface generated only with default mappings was demonstrated. No user interface generated with an appropriate customization form was shown, to avoid giving them a picture of how it “should” look. They were also told to ask the study leader as soon as they ran into trouble. The introduction took approximately 30 minutes.

After the introduction, the students worked for maximum two hours. The study leader stayed in the room to answer questions, take notes about the problems the students discussed, and to solve problems with the software. After the session, a short semi-structured interview was made where they for example were asked how well they felt that they understood the system, which problems they had run into and if some additional information could have helped them to perform better. The participants were also encouraged to suggest improvement to the system. The introduction, the surveillance of the work, and the final interview were all made by the same person.

The participants had no problems understanding the concepts of the system and the procedure of creating a customization form. However,

neither of the two pairs got even close to create a sufficiently complete customization form during the two hours. This was mostly due to the participants' problems with Swing programming but also to the specialized output of the TapBroker. A minor problem was that they were used to other development tools than those used in the study setup. Given that most of the participants' problems were Swing problems, it is highly possible that some problems with UBI did not show in this pilot study.

In the post-interview, the participants said that they got a good understanding of the basic principles of UBI, separation between function and presentation and add presentation separately, and the construction of customization forms, but a more vague understanding of the TapBroker service. They had no problems working with a service they had not developed themselves. However, they stated that it is even more important that the documentation of the service is good if the service is developed by someone else.

5.1.1 Pilot Study Evaluation

For a larger study on customization form development we believe that it is important to give more guidance to the participants at the beginning of the study. Based on the experiences from this pilot study, it would be better to start with a very simple sample service and give the participants a target user interface that their customization form should produce. That way they would get started quickly, and with a few simple mappings create a working user interface and get a good feeling of how the system works. It is also important to show them a working user interface to the service they are working with, so that they fully understand the functionality of the service.

It was also clear from the study that the participants needed more information about the service to fully understand it. A useful feature would be a basic general customization form that displays the structure of the interaction acts during development of service specific forms.

To rule out problems with the user interface programming language and the development environment, it is important in a larger study to use subjects that are experienced in the user interface language, and to let them work with a familiar development environment. User studies are very time consuming both for those planning and conducting it, and those participating in it. It is therefore important to reduce all problems that are not the focus of the study to a minimum.

6 Conclusions

We believe that interaction acts and customization forms are proven to be a feasible way of developing services with multiple user interfaces. The Ubiquitous Interactor prototype shows that it is possible to generate different user interfaces from a set of interaction acts combined with different customization forms. However, the approach needs further evaluation with more services from different domains, and new types of user interfaces, in particular speech user interfaces.

For the utility requirements, we can only present initial and tentative results, along with some pointers on how to proceed further. Since we have had external help with the HTML interaction engine and the HTML customization form for the calendar service, and conducted a pilot study where the participants developed customization forms, it seems like people not involved in the development of UBI can make use of the concepts. It was, however, obvious in the pilot study that it is very important to provide good explanations of the interaction acts concept, and the interaction act structure of a service, both in the design discussions preceding development and during the development process. To improve this, we need to conduct a larger study where both services and customization forms are developed. A study like that could also strengthen the evidence on the usability of UBI outside its original settings.

We cannot say anything at this point on how much development time that would be saved using interaction acts and customization forms. To evaluate that, a comparative study would need to be conducted where one group of subjects developed services with multiple user interfaces using UBI concepts, and other groups used other tools and techniques. In such a study, it would be very important to let the UBI users work with the system and the concepts for quite some time to rule out differences in experiences with development methods. It is, however, unlikely that a comparative study would give any clear results since too many factors affect the development time, e.g. experience with the development tool and maturity of the tool. UBI as a research prototype would have difficulties to match commercial tools. However, if a comparative study would be made, it would be important that results from a study described in the previous paragraph is available to rule out problems due to insufficient understanding of concepts or services.

Iterative design of development support systems like UBI can be quite cumbersome when the system is mature enough to be submitted to potential users. To evaluate UBI as a development tool, users need to work with the system for a long time to build up their working skill. Otherwise, it is the learnability of the system will be measured instead of its utility or efficiency. To evaluate what kind of information support developers need, many subjects are needed to give reliable results. Studies like this are very time-consuming both in the preparation phase and the actual study phase, both for researchers and subjects. An obvious problem in all user study settings is to find enough subjects that can spare the time to participate. This makes it even more important to eliminate all problems that are not the focus of the study, as concluded from the pilot study.

The aspects that need to be studied to evaluate the fulfilment of the goals of UBI are not trivial to measure. If the system saves time but produce low quality user interfaces, the system does not contribute to the development process. If no time is gained, but significant higher quality user interfaces are produced, the system still made a valuable contribution. Needless to say, neither time gain nor user interface quality in these cases is easy to measure. However, informal measures and personal opinions of subjects can give both good indications of performance and valuable design input. The full potential of the system both in facilitating development and providing more choices for end-users of services will not show until it is used in a wide development community, and a large number of services is created.

7 References

Boman, M. and Sandin, A. (2003) Implementing an Agent Trade Server, Available at arxiv.org/abs/cs.CE/0307064.

Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. (2000) *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, World Wide Web Consortium, <http://www.w3.org/TR/REC-xml>.

Bylund, M. (2001) *Personal Service Environments - Openness and User Control in User-Service Interaction*, Licentiate thesis, Department of Information Technology, Uppsala University.

Bylund, M. and Espinoza, F. (2000) *sView - Personal Service Interaction*, in Proceedings of 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology.

Espinoza, F. (2003) *Individual Service Provisioning*, PhD, Department of Computer and Systems Science, Stockholm University/Royal Institute of Technology.

Nylander, S. and Bylund, M. (2002a) *Mobile Services for Many Different Devices*, in Proceedings of Human Computer Interaction 2002, volume 2, London, 183-185.

Nylander, S. and Bylund, M. (2002b) *Providing Device Independence to Mobile Services*, in Proceedings of 7th ERCIM Workshop User Interfaces for All.

Nylander, S. and Waern, A. (2002) *Interaction Acts for Device Independent Gaming*, Technical report, T2002-04, Swedish Institute of Computer Science.

Appendix A

DTD for ISL sent from services to interaction engines.

```
<?xml version="1.0"?>

  <!ELEMENT start (id, group?, name?)>
  <!ELEMENT stop (id, group?, name?)>

  <!ELEMENT create (id, group?, name?, life, modal,
    string, parameter*)>
  <!ELEMENT destroy (id, group?, name?, life, modal,
    string)>

  <!ELEMENT select (id, group?, name?, life, modal,
    response-number, string, alternative+, meta?)>
  <!ELEMENT modify (id, group?, name?, life, modal,
    string, meta?)>
  <!ELEMENT in (id, group?, name?, life, modal,
    string, meta?)>
  <!ELEMENT out (id, group?, name?, life, modal,
    string, meta?)>

  <!ELEMENT alternative (id, group, name, string,
    retval, meta?)>
  <!ELEMENT parameter (id, value)>

    <!ELEMENT id (#PCDATA)>
    <!ELEMENT group (#PCDATA)>
    <!ELEMENT life (#PCDATA)>
    <!ELEMENT modal (#PCDATA)>
    <!ELEMENT string (#PCDATA)>

    <!ELEMENT retval (#PCDATA)>
    <!ELEMENT response-number (#PCDATA)>
    <!ELEMENT meta (#PCDATA)>
    <!ELEMENT value (#PCDATA)>

]>
```

Appendix B

DTD for ISL sent from user interfaces to interaction engines.

```
<?xml version="1.0"?>

  <!ELEMENT start (id)>
  <!ELEMENT stop (id)

  <!ELEMENT create (id, parameter*)>
  <!ELEMENT destroy (id)>

  <!ELEMENT select (id, alternative+)>
  <!ELEMENT modify (id)>
  <!ELEMENT in (id)>

  <!ELEMENT alternative (id,retVal)>
  <!ELEMENT parameter (id, value)>

    <!ELEMENT id (#PCDATA)>
    <!ELEMENT retval (#PCDATA)>
    <!ELEMENT value (#PCDATA)>

]>
```

Appendix C

DTD for specification of customization forms.

```
<?xml version="1.0"?>

<!DOCTYPE cf [
  <!ELEMENT cf (directive?, resource*, element*, id*,
    group*)>

  <!ELEMENT group (directive?, resource*, element*,
    id*, group* )>
  <!ELEMENT element (directive?, resource*, id*)>
  <!ELEMENT id (directive?, resource*)>

  <!ELEMENT resource (data|file|url)>
  <!ELEMENT directive (data)>

  <!ELEMENT data (#PCDATA)>
  <!ELEMENT file (#PCDATA)>
  <!ELEMENT url (#PCDATA)>

  <!ATTLIST resource
    name CDATA #REQUIRED
    type CDATA #REQUIRED>
  <!ATTLIST group
    name CDATA #REQUIRED>
  <!ATTLIST element
    name CDATA #REQUIRED>
  <!ATTLIST id
    name CDATA #REQUIRED>
]>
```

Appendix D

DTD for the initial set of four interaction acts.

```
<?xml version="1.0"?>

<!DOCTYPE isl [
<!ELEMENT isl (name, string,
(in|out|select|confirm|isl)+, meta?)>

    <!-- interaction acts on object level -->
    <!ELEMENT select (name, string, alt+, meta?)>
    <!ELEMENT confirm (name, string, alt+, meta?)>
    <!ELEMENT in (name, string, meta?)>
    <!ELEMENT out (name, string, meta?)>

    <!ELEMENT alt (name, string, retval, meta?)>

        <!ELEMENT name (#PCDATA)>
        <!ELEMENT string (#PCDATA)>
        <!ELEMENT retval (#PCDATA)>
        <!ELEMENT meta (#PCDATA)>
]>
```