Evaluating the Ubiquitous Interactor

Stina Nylander
Swedish Institute of Computer Science
16 October 2003
E-mail: stina.nylander@sics.se
ISSN 1100-3154
ISRN:SICS-T--2003/19-SE

Keywords: Ubiquitous Interactor, evaluation, development tools, device independence.

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.

```xml
<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.

```xml
<element name="output">
  <directive>
    <data>
```
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


