

The Ubiquitous Interactor – Mobile Services with Multiple User Interfaces

Stina Nylander, Markus Bylund, Annika Waern
Swedish Institute of Computer Science
16 October 2003

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

SICS Technical Report T2003:17

ISSN 1100-3154

ISRN:SICS-T--2003/17-SE

Keywords: multiple user interfaces, device independence, interaction acts, mobile services.

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.

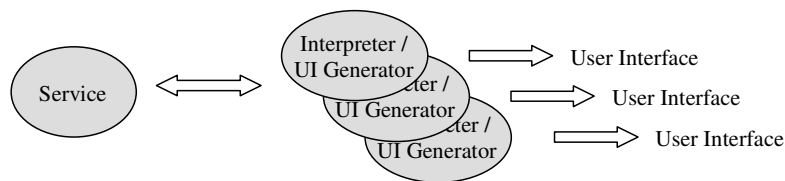


Figure 1: Services offer their interaction expressed in interaction acts, and an interpreter generates a user interface based on the interpretation. Different interpreters generate different user interfaces.

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.

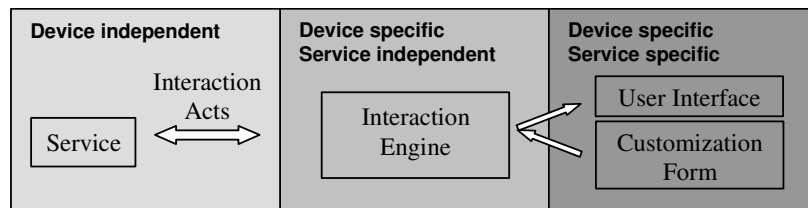


Figure 2: The three layers of specification in the Ubiquitous Interactor. Services and interaction acts are device independent, interaction engines are service independent and device or user interface specific. Customization forms and generated user interfaces are device and service specific.

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.

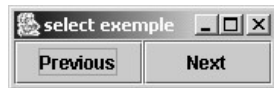


Figure 3: An example rendering of the `select` interaction act in listing 3.

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

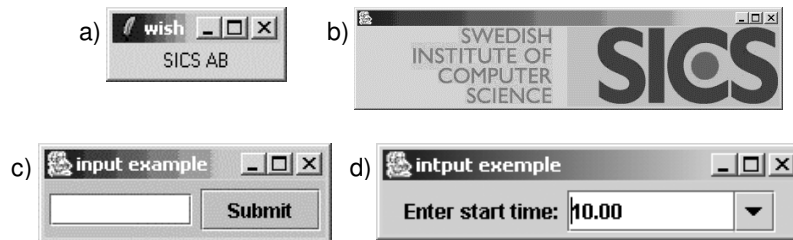


Figure 4: Rendering examples of an output and an input interaction act. Picture a and b are renderings of an output interaction act, and picture c and d are renderings of an input interaction act. Picture a is a Tcl/Tk label using the default information of the interaction act, while picture b is a Java 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

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,

PDA's and cellular phones are very different, user interfaces for the smaller devices only present parts of the available information.

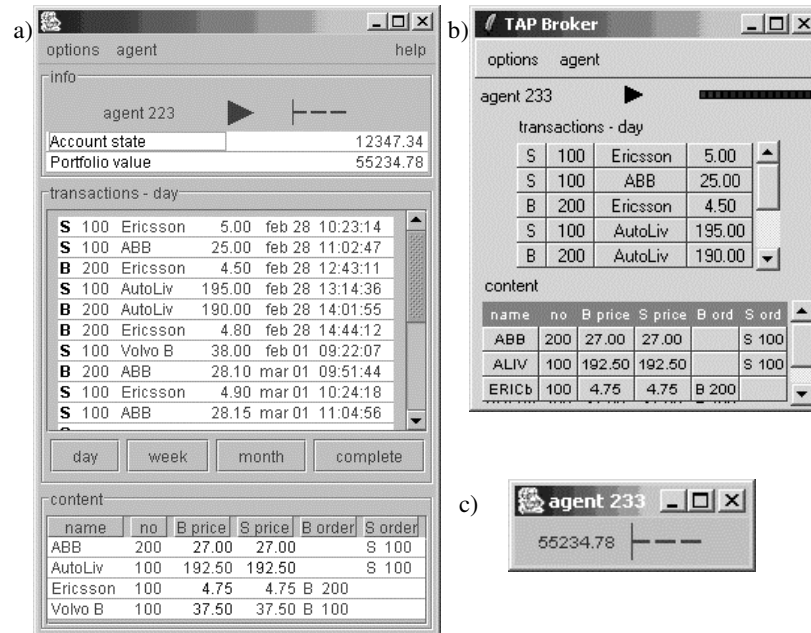


Figure 6: Three different examples of user interfaces to the TAP Broker service. Picture a shows a Java Swing user interface for desktop or laptop computers, picture b a Tcl/Tk user interface for PDA, and picture c a Java Swing user interface for very small devices (for example Java enabled cellular 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 (www.vinnova.se). 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.