Chapter 22

THE UBIQUITOUS INTERACTOR – DEVICE INDEPENDENT ACCESS TO MOBILE SERVICES

Stina Nylander, Markus Bylund, and Annika Waern
Swedish Institute of Computer Science
Box 1263, 16429 Kista (Sweden)
Tel.: +46 70 {3530369, 6615460, 3363916} – Fax: +46 8 751 7230
E-mail: {stina.nylander, markus.bylund, annika.waern}@sics.se – URL: www.sics.se

Abstract The Ubiquitous Interactor (UBI) addresses the problems of design and development arising around services that need to be accessed from many different devices. In UBI, the same service can present different user interfaces on different devices by separating user-service interaction from 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 system design of UBI, the system implementation, and two services implemented for the system: a calendar service and a stockbroker service.

Keywords: Device independence, Interaction acts, Mobile services, Multiple user interfaces.

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. Today, users have a wide range of devices at their disposal for accomplishing different tasks: desktop and laptop computers, wall-sized screens, PDAs and cellular phones. The range of services is equally wide: information services, shopping and entertainment. This creates a need for service use from different devices in different situations. Users could for example access 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 for 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, which causes problems of synchronisation 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 mark-up 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 [6]. It is not reasonable to force users to use different services for different devices to get the same content [11]. 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* [8] (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 *customisation forms* [8] (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.

2. BACKGROUND

Our interest and need for device independent services are results from previous work with the next generation electronic services [1,2]. However, the need for device independent applications is not new. During the seventies and eighties, developers faced large differences in hardware. That time the problem disappeared when personal computers emerged. The hardware got standardized to mouse, keyboard and desktop screen, and direct manipulation user interfaces worked similarly in different operating systems [6].

The situation that we face today is different. We are currently experiencing a paradigm shift from application-based personal computing to service-based ubiquitous computing. In a sense, both applications and services can be seen as sets of functions and abilities that are packaged as separate units

[4]. 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 got used to 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., cellular phones, PDAs, and wearable computers) combined with growing requirements on mobility and ubiquity, the Web based approach is no longer enough.

The multitude of device types we see today is not due to competition between vendors as before, but rather motivated by requirements of specialisation. 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.

Mike [10] and ITS [13] were among the first systems that made it possible to specify presentation information separately from the application, and thus change the presentation without changes in the application. However, they only handled graphical user interfaces, and they had other important limitations. Mike could not handle application specific data. In ITS, presentation information was considered as application independent and stored in style files that could be moved between applications, something that was not very useful [13]. In UBI, we instead consider presentation as application specific and tailor it to different devices.

Personal Universal Controllers (PUC) [7] encode the data sent between application and client in a device independent format using a set of state variables combined with dependency information, and leaves the generation of user interfaces to the client. Unlike UBI, PUC does 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.

Unified User Interfaces (UUI) [12] is a design and engineering framework that aims to provide user interfaces tailored to different user groups and situations of use in terms of users' physical capabilities, preferences and usage context. Since UUI is a project with very large scope, making all user interfaces accessible to all users, they take into account a large number of factors (e.g., contextual and environmental) that make the system more complex than we believe is necessary to solve the problems UBI is addressing.

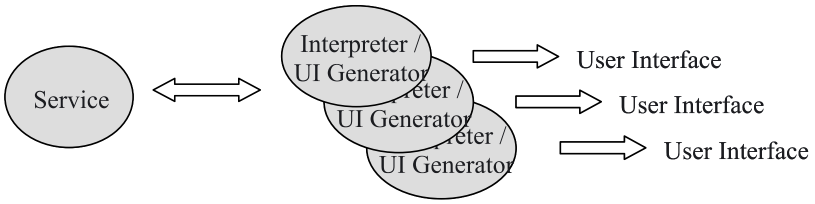

Service
Interpreter / UI Generator
Interpreter / UI Generator
Interpreter / UI Generator
User Interface
User Interface
User Interface

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

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. User interfaces are generated based on interaction acts and additional presentation information (Fig. 1). In the standard case, interaction acts are interpreted and user interfaces generated on the device side, but for thin clients the interpretation and generation can be made on a server. Although we are aiming for general solutions, which cover interaction with many sorts of applications via a large range of interface types, we realise that it might be difficult and in some cases not even desirable to develop services using interaction acts. Some services might be too complex, while others might be too device dependent (like a high-end multi-player game) to benefit from this approach. For the time being, we are therefore limiting our vision to a few interface types, (mainly windows-based GUIs, command-line interfaces, and speech interfaces), and more simple services (e.g., information services). However, these types of services and UIs cover most of what is available today and will be available in the near future.

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 of interaction acts.

The set of interaction acts have been established through analysis of existing services and applications [8]. We examined 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, a desktop home care planning tool, and computer games. Live face-to-face instructions were also studied informally. The current set have eight members supported in UBI: `input`, `output`, `select`, `modify`, `create`, `destroy`, `start` and `stop`. `Input` and `output` are defined from the systems point of view. `Select` operates on a predefined set of alternatives. `Create`, `destroy` and `modify` handles the life cycle of service specific data, while `start` and `stop` handles the interaction session. All interaction acts except `output` returns user actions to services. `Output` only presents information that users cannot act upon.

During user-service interaction, the system needs more information about the interaction acts than its type. Interaction acts are uniquely identifiable, so that user actions can be associated with them and interpreted by services. It is also possible to define for how long a user interface component based on an interaction act should be present in the user interface before being removed. Otherwise only static user interfaces can be created. It is possible to create modal user interface components based on interaction acts, e.g. components that lock the user-service interaction until certain user actions are performed. This way, user actions can be sequenced when needed. All interaction acts also have a way to hold information, as a default base for the rendering of interaction acts. Finally, meta-data can be attached 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. These groups allow nesting.

4.2 Controlling the Presentation

To give service providers a way to specify how 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 [3,6], since it is used for e.g., 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 link interaction acts to for example widgets or templates of user interface components. Resources could be pictures or sounds that are used in the rendering of an interaction act.

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

5. IMPLEMENTATION

The Ubiquitous Interactor (UBI) has three main parts: the Interaction Specification Language (ISL), customisation forms, and interaction engines. ISL is used to encode the interaction acts sent between services and user interfaces, customisation forms contain presentation information, and interaction engines generate user interfaces based on interaction acts and information from customisation 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 customisation forms are service and device dependent.

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 available in the user interface for a specified time and then removed by UBI, confirmed components are available until the user has performed a given action, and persistent components are available in the user interface during the whole user-service interaction. The default value is persistent. All interaction acts can be given a symbolic name, and belong to a named presentation group in a customisation form. This will be discussed further in Section 5.2.

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. The default value is false. All interaction acts contain a string value used to hold default information. It is also possible to attach meta data to all interaction acts. Listing 1 shows the ISL of a select interaction act.

```
<select>
  <id>235690</id>
  <life>persistent</life>
  <modal>false</modal>
  <string>browseList</string>
  <alternative>
    <id>5463</id>
    <name>alt</name>
    <string>Previous</string>
    <retVal>0</retVal>
  </alternative>
  <alternative>
    <id>5893</id>
    <name>alt</name>
    <string>Next</string>
    <retVal>1</retVal>
  </alternative>
</select>
```

Listing 1: ISL encoding of a select interaction act with id, life cycle, modality, and default content information.

Interaction acts can be grouped using a designated tag `isl`, and groups can be nested to provide more complex expressions of interaction. These groups contain the same type of information assigned to single interaction acts. The ISL code sent from services to interaction engines contains all information about the interaction acts: id, name, group, life cycle, modality, default information and metadata. A large part of this information is only useful for the interaction engine during generation of user interfaces. Thus, when users perform actions, only the relevant parts of interaction acts are sent back to the service. Two different DTDs have been created for this, one for encoding interaction acts sent from services to interaction engines, and one for encoding interaction acts sent from interaction engines to services. The DTDs are available at http://www.sics.se/~stny/UIB/DTDs/dtd.html.

5.2 Customisation Forms

Customisation 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, information at type level provides information for all interaction acts of the given type; and information on name level provides information about all interaction acts with the given symbolic name. The levels can also be combined, for example creating specifications for interaction acts in a given group of a given type, or in a given group with a given name.

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 customisation 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).

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

Listing 2: ISL encoding of an output interaction act with a symbolic name, and that belongs to a customisation form group called calendar.

Listing 2 shows an encoding of the output interaction act from listing 1 with a symbolic name, and as a member of a customisation form group.

Customisation forms are structured and can be arranged in hierarchies which allows for inheriting and overriding information between forms. A basic form can be used to provide a look and feel for a family of services, with different service specific forms adding or overriding parts of the basic specifications to create service specific user interfaces. A customisation form does not need to be complete. Interaction acts that do not map to any presentation information specified in the form are rendered with defaults.

Customisation forms are encoded in XML and a DTD can be found at http://www.sics.se/~stny/UBI/DTDs/dtd.html. An entry in a customisation 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 3 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 3: A mapping on type level for an `output` interaction act that maps a named interaction act to a Java class that is used to render it.

5.3 Interaction Engines

Interaction engines interpret interaction acts and generate suitable user interfaces of a given type for services on a given device or family of devices. They also encode performed user actions as interaction acts and send them back to services. 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 customisation form of a service, that presentation takes precedence. Otherwise, defaults are used. For example, an output could be rendered as a label, or speech generated from its default information, while an input could be rendered as a text field or a standard speech prompt. We have implemented interaction engines for Java Swing, Java Awt, HTML, and Tcl/Tk user interfaces. All four interaction engines can generate user interfaces for desktop computers, but the Tcl/Tk and the Java Awt engine are designed for PDA and cellular phone respectively. The HTML interaction engine generates HTML code and sends it to a browser via HTTP. The Tcl/Tk interaction engine is designed to generate Tcl/Tk code and send it to a PDA that will interpret the code and render the user interface. In these cases, the interpretation and generation is not executed on the PDA to save computational resources. Both the Java Swing and the Java Awt interaction engines interpret interaction acts and generate user interfaces on the target device.

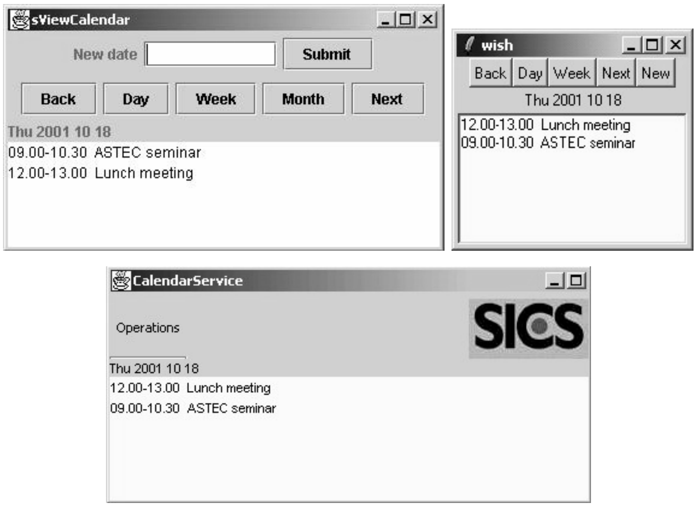
6. SERVICES

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

The calendar service supports basic calendar operations as entering, edit and delete information, navigate the information, and display different views of it. The service is accessible from three types of user interfaces: Java Swing and HTML user interfaces for desktop computers, and Tcl/Tk user interfaces for handheld computer. Two different customisation forms have been created for Java Swing, and one each for Tcl/Tk and HTML. These four forms generate different user interfaces from the same interaction acts. See Fig. 2 for pictures of three of the generated user interfaces.



Thu 2001 10 18
09.00-10.30 ASTEC seminar
12.00-13.00 Lunch meeting

Thu 2001 10 18
12.00-13.00 Lunch meeting
09.00-10.30 ASTEC seminar

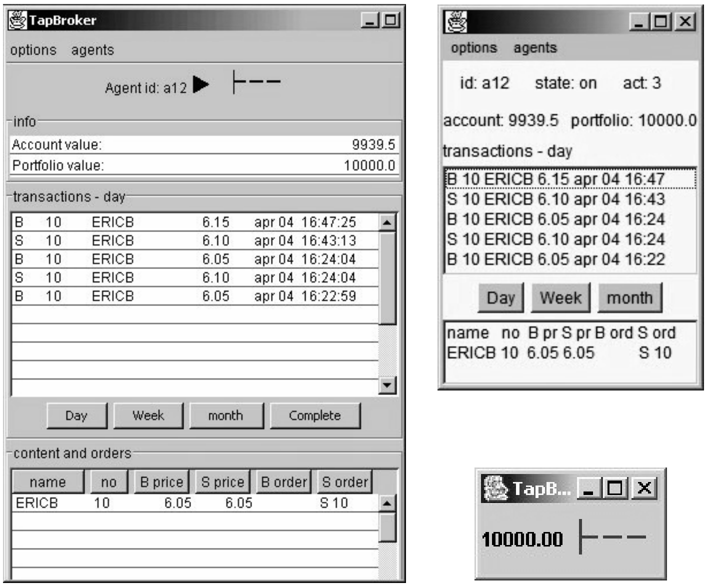
Thu 2001 10 18
12.00-13.00 Lunch meeting
09.00-10.30 ASTEC seminar

Figure 2. User interfaces for the calendar service. The two to the left are generated by the Java Swing interaction engine using two different customisation forms. The one to the right is generated by the Tcl/Tk interaction engine.

6.2 Stockbroker Service

The stockbroker service TapBroker [9] has been developed as a part of a project at SICS that works with autonomous agents that trade stocks on the behalf of users [5]. Each agent is trading according to a built in strategy, and users can have one or more agents trading for them. TapBroker provides feedback on how their agents are performing so that users know when to change agent, or shut them down.

The TapBroker service provides agent owners with feedback on the agent's actions: order handling, and performed transactions. It also provides information on the account state (the amount of money it can invest), status (running or paused), activity level (number of transactions per hour), portfolio content, and the current value of the portfolio. However, it does not provide any means to configure or control the agent. Agents work autonomously and cannot be manipulated from outside for security reasons. We have implemented customisation forms for Java Swing, HTML and Java Awt (Fig. 3).



Agent id: a12

info

Account value:	9939.5
Portfolio value:	10000.0

transactions - day

B	10	ERICB	6.15	apr 04 16:47:25
S	10	ERICB	6.10	apr 04 16:43:13
B	10	ERICB	6.05	apr 04 16:24:04
S	10	ERICB	6.10	apr 04 16:24:04
B	10	ERICB	6.05	apr 04 16:22:59

content and orders

name	no	B price	S price	B order	S order
ERICB	10	6.05	6.05		S 10

id: a12 state: on act: 3

account: 9939.5 portfolio: 10000.0

transactions - day

```
B 10 ERICB 6.15 apr 04 16:47
S 10 ERICB 6.10 apr 04 16:43
B 10 ERICB 6.05 apr 04 16:24
S 10 ERICB 6.10 apr 04 16:24
B 10 ERICB 6.05 apr 04 16:22
```

```
name  no  B pr S pr B ord S ord
ERICB 10  6.05 6.05       S 10
```

10000.00

Figure 3. Three user interfaces to the TapBroker: a Java Swing UI for desktop computers (a), a Java AWT UI for mobile phone (b), and a UI for a very small device (c).

7. FUTURE WORK

We will investigate how to handle dynamic resources in UBI. Services that use dynamic media resources extensively, e.g., a service for browsing a video database, might need an extension of our customisation form approach to work efficiently for different modalities. One solution could be to handle the choice of media type outside the customisation 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 customisation 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 customisation forms. Development and maintenance work is simplified since only one version of each service need to be developed. New customisation forms can be created at any point, thus services can be developed for an open set of devices.

ACKNOWLEDGEMENTS

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

REFERENCES

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

[2] Bylund, M. and Espinoza, F., *sView - Personal Service Interaction*, in Proceedings of 5th International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology PA EXPO'2000 (Manchester, 10-14 April 2000), 2000.

[3] Esler, M., Hightower, J., Anderson, T., and Borriello, G., *Next Century Challenges: Data-Centric Networking for Invisible Computing. The Portolano Project at the University of Washington*, in Proceedings of 5th ACM International Conference on Mobile Computing and Networking MobiCom'1999 (Seattle, 15-20 August 1999), ACM Press, New York, 1999.

[4] Espinoza, F., *Individual Service Provisioning*, Ph.D. thesis, Department of Computer and Systems Science, Stockholm University/Royal Institute of Technology, 2003.

[5] Lybäck, D. and Boman, M., *Agent Trade Servers in Financial Exchange Systems*, accessible at http://arxiv.org/abs/cs.CE/0203023.

[6] Myers, B.A., Hudson, S.E., and Pausch, R., *Past, Present and Future of User Interface Software Tools*, ACM Transactions on Computer-Human Interaction, Vol. 7, No. 1, 2000, pp. 3-28.

[7] Nichols, J., Myers, B.A., Higgings, M., Hughes, J., Harris, T.K., Rosenfeld, R., and Pignol, M., *Generating Remote Control Interfaces for Complex Appliances*, in Proceedings of 15th Annual ACM Symposium on User Interface Software and Technology UIST'2002 (Paris, 27-30 October 2002), ACM Press, New York, 2002, pp. 161-170.

[8] Nylander, S., *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*, Licentiate Thesis, Department of Information Technology, Uppsala University, 2003.

[9] Nylander, S., Bylund, M. and Boman, M., *Mobile Access to Real-Time Information - The case of Autonomous Stock Brokering*, Journal of Personal and Ubiquitous Computing, Vol. 8, No. 1, 2003, pp. 42-46.

[10] Olsen, D.J., *MIKE: The Menu Interaction Kontrol Environment*, ACM Transactions on Graphics, Vol. 5, No. 4, 1987, pp. 318-344.

[11] Shneiderman, B., *Leonardo's Laptop*, The MIT Press, Cambridge, 2002.

[12] Stephanidis, C., *The Concept of Unified User Interfaces*, in C. Stephanidis (ed.), "User Interfaces for All - Concepts, Methods, and Tools" Lawrence Erlbaum Associates, Mahwah, 2001, pp. 371-388.

[13] Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., *ITS: a Tool for Rapidly Developing Interactive Applications*, ACM Transactions on Information Systems, Vol. 8, No. 3, 1990, pp. 204-236.