Different Approaches to Achieving Device Independent Services – an Overview

Stina Nylander, 2003
Swedish Institute of Computer Science
16 october 2003
E-mail: stina.nylander@sics.se
SICS Technical Report T2003:16
ISSN 1100-3154
ISRN:SICS-T--2003/16-SE

Keywords: device independence, user interface adaptation, mobile services

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

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

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

1.1 Problem description

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

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

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

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

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

To achieve device independence through the use of abstractions, a suitable level of abstraction, and useful units of description need to be identified. The level of abstraction can be, for example, the user interface level with user interface components as units of description, or the user interaction level with user actions as units of description. It is also possible to use models as descriptions of applications, for example task models or device models. The targeted range of modalities, devices and applications highly affects the choice of abstraction level.
Abstractions must be made concrete to create user interfaces for applications. Units of description need to be mapped to user interface components that fit the target device. This mapping can be made explicitly by developers, or defaults can be specified for the system to use. The Ubiquitous Interactor uses a combination of these two. It is also possible to leave the choice to the system by encapsulating knowledge on user interface creation in the system. This is usually made through a set of rules that the system uses to interpret the description of the application. In some systems developers are offered mapping suggestions from the system and have the power of choosing and overriding the system (Eisenstein and Puerta, 2000a). In the choice of leaving mapping to developers, system or both parties, control is often traded for efficiency. When designers make the mappings they have a good control over what user interfaces will be produced. When the system handles the mapping, the user interface creation is much faster, but usually less predictable. A problem for model-based user interface generation has been that the generated user interfaces were unpredictable (Myers et al., 2000).

<table>
<thead>
<tr>
<th>Description units</th>
<th>Interaction tasks (1984)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Meta-Widgets (1996)</td>
</tr>
<tr>
<td></td>
<td>UIML (1999)</td>
</tr>
<tr>
<td></td>
<td>Plastic user</td>
</tr>
<tr>
<td></td>
<td>interfaces (2001)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>Mike (1987)</td>
</tr>
<tr>
<td></td>
<td>ITS (1990)</td>
</tr>
<tr>
<td></td>
<td>XIML (2001)</td>
</tr>
</tbody>
</table>

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

The systems that inspired the work with UBI the most are those that provide designers and developers with a set of abstract description units to describe the applications, or allow designers to create the set. Even if
all the projects did not use their description units to generate user interfaces they provided inspiration in the establishment of interaction acts for UBI. How the systems that do generate user interfaces map the description units to concrete user interface representations is not important here. Some of them use knowledge-based methods, while some let the designers do the mapping. A more detailed description of the different systems will be given in section 3.1. The model-based approaches have not been an important inspiration but they will still be treated below in section 3.4. Figure 1 shows the systems in section 3 and 4 grouped according to their primary target group and their way of describing applications.

2.1 Fixed or open set of description units

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

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

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

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

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

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

3 Sets of Description Units

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

3.1 Fixed set of abstractions

3.1.1 Interaction tasks

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

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

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

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

3.1.3 Jade

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

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

The classification of the interaction techniques is based on graphical user interfaces, and it is not modality independent. For example, the distinction between single-choice and single-choice-with text would not be meaningful in a text or speech based user interface. It is possible that this distinction reflects the possibility to make commands both with text, using the keyboard, and with the mouse in graphic user interfaces. Number-in-a-range might also only be useful for a small range of applications.

3.1.4 Selectors

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

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

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

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

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

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

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

The Personal Universal Controller (PUC) (Nichols et al., 2002) is an attempt to provide simpler user interfaces to home appliances. Users have their personal computing device that they carry with them, for example a PDA, which can present user interfaces to appliances surrounding them, VCRs, stereos, microwave ovens etc. With this approach, users can interact with all appliances surrounding them using the same device. The PUC downloads a description of the appliance’s functions, creates a PUC specification and generates a GUI or a speech user interface without intervention from the user.

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

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

3.1.7 Alternate Interface Access Protocol

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

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

### 3.2 Open set of abstractions

#### 3.2.1 Meta widgets

The concept of metawidgets (Blattner and Glinert, 1996) was created in 1996 to simplify creation of multi-modal user interfaces, specifically those cases where information is encoded for a certain output modality. Metawidgets are multimodal widgets, used to abstract information from the application for the user. To be able to present themselves in different ways for different modalities, metawidgets contain representations for different modalities, or combinations thereof, and methods for selecting among them.

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

#### 3.2.2 User Interface Markup Language

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

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

Unified User Interfaces (UUI) is a design and engineering framework composed by three parts: a method for design, a software architecture, and tools (Stephanidis, 2001a, Savidis et al., 2001, Stephanidis, 2001b). The overall goal of UUI is to create accessible user interfaces with high quality of interaction by providing user interfaces tailored to different user groups and situations of use. Situations of use include external factors as noise, as well as the particular device used for access. A UUI should be possible to use for anyone independently of physical abilities or computer experience.

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

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

3.3 The Web approach

The Web was created as a way to share documents between researchers around the world (Berners-Lee et al., 1992). The abstractions used to describe the documents allowed people to access them from different computers and different operating systems using Web browsers. As long as there existed a browser that users could run on their machine, they could access the Web of documents. Today, the Web offers a lot more
than simple documents, and the devices used for access shows a huge diversity in capabilities of interaction and presentation. A simple approach to achieve device independence for the Web would be to create Web browsers for every device. However, the differences of the devices are too great for them to show the same Web pages. For example, some devices cannot handle pictures or sound, while others have too small screens to display a regular Web page. To address this problem, standards for describing device capabilities have been created, for example the Composite Capability/Preference Profile (Reynolds et al., 1999). Several attempts have been made to automatically adapt Web pages to devices with small screens (Bickmore and Schilit, 1997, Wobbrock et al., 2002, Trevor et al., 2001). There are also attempts to create an interlingua for Web markup languages, a general markup that can be converted to the others, to simplify Web design for different devices (Menkhaus, 2002). But even if the problems of device differences could be solved, the suitability of the Web as a foundation for device independent services can be questioned.

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

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

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

3.3.1 HTML, XHTML and Cascaded Style Sheets

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

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

3.3.2 XML

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

3.3.3 W3C Device Independence Working Group

The World Wide Web Consortium has created a working group addressing the problems of device independence for the Web. The purpose of this group, created in February 2001, is to review the
challenges, possibilities and problems that arise in Web authoring due to the emergence of new devices such as palm computers and Web TV. The focus is entirely on the developer side, particularly on how to adapt and present Web applications, and how Web applications can be accessed from different devices. The long-term goals of the working group are that different devices can get Web content adapted to their presentation capabilities and that authors can provide this in an efficient way (W3C, 2001). The group has not yet issued any W3C Recommendations, but there is a working draft identifying various challenges concerning both application and device side of device independent authoring (Lewis, 2002).

4 Examples of Model-Based Systems

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

4.1 ITS

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

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

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

specific and create many different styles for a single application, thus creating different user interfaces.

4.2 Plastic User Interfaces

Plastic user interfaces are an attempt from 2001 to create applications with user interfaces adapted to different devices (Calvary et al., 2001). A plastic user interface is defined as a user interface that has the capacity to "withstand variations of context of use while preserving usability" (Calvary et al., 2001). This might be accomplished automatically or with human intervention. Context of use in this setting includes platform related issues like device features and bandwidth, and also the users’ environment, including peripheral people and objects.

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

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

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

4.3 XIML

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

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

5 Summary

Device independent services have been a research issue during different periods of the computer science history, and each period has had its specific reason. During the seventies and early eighties, developers were up against large variations in hardware and needed methods for developing device independent applications to reduce workload and increase the number of available applications. At the beginning of the 21st century, we are trying to put ubiquitous and mobile computing in place. Mobile users that use different devices in different situations want to access services from wherever they are call for services that can be accessed from a wide range of devices.
The older approaches may have had different purposes, but can still give inspiration to device independence in the ubiquitous computing era. The Ubiquitous Interactor is a new approach that draws inspiration from earlier projects to face the challenges of the new computing.

6 References


Different Approaches to Achieving Device Independence


