A User Interface Toolkit for a Small Screen Device

Aleksi Uotila

University of Tampere
Department of Computer and
Information Sciences
Master’s thesis
February 2000

University of Tampere
Department of Computer and Information Sciences
Aleksi Uotila: A User Interface Toolkit for a Small Screen Device
Master's thesis, 69 pages

February 2000

The appearance of different kinds of networked mobile devices and network appliances creates special requirements for user interfaces that are not met by existing widget based user interface creation toolkits. This thesis studies the problem domain of user interface creation toolkits for portable network connected devices. The portable nature of these devices places great restrictions on the user interface capabilities. One main characteristic of the devices is that they have small screens compared to the existing desktop environment displays.

Mobile users want to be able to use their applications whenever and wherever possible. The same applications need to be accessible in different situations and environments and from different platforms and devices. Code mobility and wide area wireless data networks will enable us to transmit application code virtually everywhere, although existing user interface toolkits form a bottleneck for universal portability and access. The conclusion is that for globally accessible and portable applications explicit toolkit level support for application semantics is needed.

Keywords: user interface toolkit, widgets, application semantics, semantic controls, software portability, graphical user interfaces, Java, WAP, hand-held devices

Acknowledgements

I wish to thank Nokia Mobile Phones for the opportunity to carry out my master's thesis in conjunction with my regular duties. I have had a chance to enjoy a pleasant working atmosphere, and I wish to express my gratitude for everyone in my group. Thanks to my bosses Jyrki Yli-Nokari, Kari Systä and Erkki Rysä for helping me to find this interesting thesis subject.

My special thanks go to my parents who have always supported me in my studies and my fiancée, Kaisa, for her encouragement and support. Thanks also to Mikko Kauppinen for detailed language corrections.

Finally, I would like to thank Professor Roope Raisamo at the University of Tampere who acted as my supervisor and gave me valuable advice and comments throughout the project.

Tampere, February 16th, 2000

Aleksi Uotila

Contents

1. Introduction.......1
2. Concepts.......3
2.1 Platform independent graphical user interfaces.......3
2.1.1 User interface toolkits.......3
2.1.2 Code mobility.......5
2.1.3 Application context.......6
2.2 Devices.......6
2.3 Differences with desktop user interfaces.......9
2.3.1 Output.......9
2.3.2 Input.......11
3. Technologies.......15
3.1 Current Java user interface concepts.......15
3.2 Layout management.......19
3.3 WAP technology and its user interface concepts.......24
3.4 General WAP and Java interoperability.......27
4. Different user interface toolkit concepts.......30
4.1 A subset of AWT or Swing.......31
4.1.1 A subset of AWT for WAP user interface elements.......32
4.2 A toolkit based on WAP user interface.......33
4.2.1 WAP concept based.......34
4.2.2 Wapplets embedded in Microbrowser.......34
4.3 Configurations and device profiles.......35
4.4 Lower level user interface API.......36
4.5 Existing implementations.......38
4.5.1 Existing Java toolkits.......38
4.5.2 User interface markup languages.......39
4.5.3 Using markup languages for specifying user interface.......40
4.5.4 The Spotless system.......41
5. User interface toolkit designed for small screens.......44
5.1 Requirements.......44
5.1.1 Interaction devices.......45
5.1.2 Accessibility.......46
5.1.3 The toolkit.......48
5.2 Design issues.......49
5.2.1 The screen size problem.......49
5.2.2 The widget problem.......50

5.2.3 The abstract factory problem 54

5.3 Details 56

5.3.1 Associations between semantic controls 56

5.3.2 Input abstractions 59

5.3.3 Capability mechanisms 61

5.3.4 Compound interactive controls 62

6. Conclusions 64

References

1. Introduction

There is an ever-increasing need for technology that enables software to be executed on different systems and platforms without compilation or other changes in the executable binaries of the software. Existence of different communication and computer networks and the idea of code mobility in addition to information mobility enhance this need. The main obstacles to portable software have been the user interface issues. Portability has been achieved only among systems that do not have any great differences. In the past decade, several user interface portability technologies have been designed and implemented for desktop systems [McKay, 1997]. However, the portable user interface technologies still form the bottleneck for software portability at large. There is even more demand for portable applications in the hand-held device category, because these devices and the networks they are connected to will be used for new kinds of dynamic services. Since there are no de facto device capabilities or platform, a standard mechanism for specifying user interfaces for these devices is needed. Without a standard these services would not be realised or would only be realised in niche markets.

Appearance of different kinds of mobile devices, such as personal digital assistants and mobile phones, has created completely new requirements for portability and accessibility. The applications need not only be accessible from different platforms but also from different kinds of environments and situations the mobile user happens to be in. This places great emphasis on user interface design and implementation. However, existing user interface toolkits do not have any support for problems related to the implementation of portable and accessible user interfaces. Often the toolkits even force applications to use specific screen configurations and platforms. This means that new kinds of user interface toolkits are needed and that the problems need to be explicitly addressed at the toolkit level, thus relieving some pressure from developers to the system designers. This thesis posits that application semantics [Johnson, 1992] should be used explicitly by application developers when designing user interfaces.

This thesis is divided into six chapters. Chapters 2 and 3 introduce the terminology, theoretical concepts and technologies that the rest of the thesis bases the discussion on. Chapter 2 presents the theoretical concepts called code mobility and application context. Next, a general classification of hand-held devices and more detailed discussion about input and output methods as compared to desktop systems are presented to illustrate the diversity of hand-held device capabilities. Chapter 3 gives a general overview of Java and WAP

technologies. The user interface practicalities of both technologies are discussed in more detail. The chapter also discusses the layout management issues of user interfaces. In Chapter 4, the user interface toolkit concepts are presented. Chapter 5 gives the requirements and design issues for a small screen device user interface toolkit. Accessibility is recognised as one special requirement that will have large impact on the toolkit and the programming model for user interface creation. Some details on how to implement and address the problems that appear as a result of this model are given at the end of the chapter. The conclusions of the thesis are presented in Chapter 6.

2. Concepts

In this chapter the theoretical concepts and background for the user interface problem domain are presented. This discussion attempts to keep out specific technologies; for example, a short history of platform independent graphical user interfaces is presented and specific technologies are used only as examples. This thesis concentrates mainly on the procedural user interface specification mechanisms, such as user interface toolkits. After the platform independence issues, the theoretical concepts called code mobility and application context are presented. These concepts shape the requirements for user interface toolkits for small screens.

2.1 Platform independent graphical user interfaces

Platform independence is important thing in heterogeneous computing systems. The Internet is an ultimate example of a heterogeneous system. Global standards are needed to enable worldwide access to different information systems. Current mobile devices connect to standard networks that are isolated from the Internet. In the future, more and more hand-held devices use wireless connections to corporate intranets and public Internet services. Software downloading to hand-held devices has been developed to leverage code mobility among mobile users. These developments introduce a strong need for platform independent graphical user interfaces for small hand-held devices.

2.1.1 User interface toolkits

A *toolkit* is used to create a graphical user interface (GUI) for an application. A toolkit is a library of “widgets” that can be called by application programs. A widget is a display-oriented user interface element that binds both input and output to a single software component. Widgets use visual output as widgets are drawn on screen to present some application value. Widgets are also a way of using a physical input device to input a certain type of value [Myers, 1995]. Typically, widgets in toolkits include menus, buttons, scroll bars, text type-in fields and so forth. Toolkits provide an application programming interface (API) that is a procedural or object-oriented interface for creating the GUI and managing interactions between the user and the GUI. This means that a GUI can be created and managed only by a programmer when a toolkit is used.

Virtual toolkits are toolkits that try to hide the differences between the various toolkits. Widgets of a virtual toolkit are mapped into the widgets of each platform’s *native* toolkit that the virtual toolkit is implemented on. Virtual toolkits are a method for cross-platform development. They try to make

application porting to different systems easier since the user interface should run without changes on different platforms.

There are two groups of virtual toolkits. Toolkits that use *heavyweight* widgets link to the different native toolkits' widgets on the host machine. This provides the application the native look and feel that the user expects and is accustomed to. There are several problems with this kind of approach; for example, the application must be carefully tested on every platform since the sizes of widgets vary between platforms. This could also break the user interface design on some platforms. Also, the virtual toolkit could only provide services that are implemented on all platforms the virtual toolkit should run on. This is called the *common denominator programming model* and it restricts toolkits' power of expression. An application also often needs access to the graphics system so the virtual toolkit needs some kind of a lower level graphics interface. An example of a heavyweight toolkit is Abstract Window Toolkit (AWT) of Java. AWT is discussed in more detail in section 3.1.

The second group of virtual toolkits uses *lightweight* widgets that use the abstract graphics interface of the virtual toolkit to draw themselves on screen. The widgets are thus re-implemented on the virtual toolkit and there is no coupling to actual widgets on the underlying system. This adds overhead because of the need of a large runtime system for the virtual toolkit. Problems with lightweight widgets result from the fact that widgets look the same on all platforms, so the applications using lightweight widgets often provide a different look and feel than the one the user is accustomed to. The porting of a virtual toolkit that uses lightweight widgets is often faster because only the abstract graphics interface must be implemented on a host platform in contrast to the implementing of all the virtual toolkit/host platform widget mappings that need to be done when porting a heavyweight widget based toolkit. The Swing toolkit of Java platform is a lightweight toolkit. Swing builds on top of AWT graphics primitives and is fully written in Java, so it works in every environment that implements the AWT graphics primitives.

Current virtual toolkits are targeted for desktop-computing environments with an established set of input and output devices. There are many questions related to platform independent user interface toolkits for hand-held device market [Keronen, 1999]. It is impossible to know yet what will be the best abstractions for these interfaces, because the user interface hardware in these devices is constantly evolving and product categories are shaping themselves. Further, there will be a need for multimodal interfaces for hands-free, ears-free or eyes-free operation. This means that the same application software could be

used in different situations that require different output/input channels. How this adaptive user interface technology should be done is not known.

The move from presentation to semantics based markup languages for static content on the web will be paralleled by a move from syntax based laid out widget hierarchies to semantics based user interface abstractions for procedural content. As in static content, style sheets can be used as rules for producing the visual presentation of the user interface, or the presentation can be totally implementation dependent. This kind of technology is needed to realise code mobility and software downloading in open networked systems.

2.1.2 Code mobility

Code mobility is the primary driver for platform independent graphical user interfaces. Code mobility is needed to distribute user interfaces between terminals in networks. Code is usually pointless without a user interface, so downloaded code most probably introduces a new user interface to the device. In this section, code mobility is defined more formally. In addition, some examples are given about where the code mobility is or will be in use and how it will affect the user interface issues.

First, a broader concept called computational mobility as described by Dix *et al.* [1999] is defined. Computational mobility means that computation may start at one network site, then move on and continue to execute at another network site. Network sites can be any nodes connected to network including, but not limited to, terminals, gateways and servers. Dix *et al.* present that computational mobility may involve code mobility, control mobility, data mobility or link mobility. Code mobility is the term for exchanging behavioural information, like procedural programming code, between different parties in a network. In control mobility, the thread of control changes between parties, as in Remote Procedure Call (RPC), for example. In data mobility the non-behavioural information is exchanged, in the form of parameters, for example. In link mobility, the endpoint of one network connection is sent to another network connection to allow the receiving party to connect to it. For example, in Jini [Arnold *et al.*, 1999] introduction of a new service to network uses link mobility to provide other networked sites access to this new service.

Data mobility is in broad use and currently most data delivered in computer networks is static data. At least code or control mobility is needed in addition to data mobility to provide computational mobility. On the web, a Java applet has a notion of code mobility in the form of software downloading. To be more useful, the software downloading needs system support for dynamic installation and removal of new software components to and from the

execution environment. In applets this support is implemented in the form of dynamic binding and class loaders. Additionally, a web applet can leverage control mobility, for example, by using RMI to make remote calls. Java is discussed in more detail in section 3.1. In wireless networks the Wireless Application Protocol (WAP) [WAP Architecture, 1998] introduces an application environment that enables simple code mobility in the form of a simple scripting language. WAP and its application environment are discussed in more detail in section 3.3. Code mobility is the most important aspect of computational mobility since a user interface is defined by code. The visual part of a user interface can be defined by other means, like some markup language (see section 4.5.2), but user interface semantics is always defined in code. Even if the behaviour of a user interface is distributed from a terminal to some other network nodes by using control mobility, the user interface has some notion of code that carries the RPC call.

2.1.3 Application context

Applications are executed in some environment. The term application context is used to describe the environment of the hand-held device the application is executed and the user interface displayed on. For example, in some devices applications can be executed from the permanent memory of a device and the application user interfaces would take the whole screen or be displayed as a window, like firmware applications in some PDA devices. In some other context applications can be downloaded from the network and displayed as embedded objects in some other application, like applets in a web page.

The discussion is not tied in any specific application context; in fact a good user interface toolkit would be such that it has few dependencies on any specific application environment, since no type of application environment is well established among hand-held systems. This also means that application vendors most probably do not want to tie their applications to any specific context, so portability between different application environments is a requirement for the toolkit.

2.2 Devices

In this section a classification of target devices for the user interface API is presented. While the discussion and examples concentrate mainly on the hand-held device market, there is also a need for platform independent user interface toolkits in other types of small screen devices, for example, in point of sale terminals. The chosen view does not mean that the discussion is not valid for user interface toolkits for these other types of devices. The hand-held systems market is relatively new, so the different device categories presented here are

not well established. The given classification tries not to be exhaustive or formal and is given here only for informational purposes so that the reader could understand the diversity of the target devices.

The main focus is on all hand-held devices that contain a physically small display screen. Although, when portability and accessibility issues are discussed, we find that the toolkit needs to be even more general. To certain point it cannot even be expected at the toolkit level that a display screen of any kind exist in a device.

The classification presented in Table 1 has three parameters: supported input devices, capabilities of screen configuration and type of network access. More generally, these could be seen as input and output channels, but for output the dominant device is the screen and other output channels like sound and tactile feedback are in rather limited use in the user interfaces for these devices. When multimodal interfaces and additional accessibility features will appear, the use of other output channels will be more frequent. The demand of multimodal user interface features will grow as usage of mobile computing grows and support for these interfaces will become more and more important in future devices. Additionally, other output channels are more significant in hand-held devices than in desktop systems since the screen size is much smaller and the usage context of these devices is different. For example, the auditory output like speech and sound is more important in hand-held systems than in desktop systems due to the fact that the screen is smaller and that the hand-held devices are not constantly in active use.

It is envisioned that in the future all hand-held devices will have some kind of a wireless network connection. Terms like "mobile" or "wireless information devices" have been used for this kind of devices. The trend for wireless network connection is currently visible as mobile phones are used to access network resources with laptop computers and PDA devices. More integrated systems have also been developed, such as the Nokia Communicator devices or different GSM card phones for laptop computers.

Category	***Input***	***Screen configuration***	***Network access***
Mobile communication devices - Standard phones - PIM phones - Two way pagers	Keypad	< 100*100 pixels < 5 cm^2 Single colour	Low-end network access
Mobile computer devices - PDA & PIM devices - Pads - Sub notebooks	Touchscreen (Keyboard)	> 100*100 (<= 640*480) pixels > 100 cm^2 >= 16 colours	No direct access
Mobile information devices			High-end network access
- Communicators	Keyboard	As in mobile computer Landscape display Half VGA resolution	
- Smart phones	Touchscreen	As in mobile computer Portrait display	

Table 1. Classification of target devices.

Table 1 presents the three device categories: mobile communication devices, mobile computer devices and mobile information devices. Mobile communication devices include devices that are physically small and thus have limited input and output capabilities. They are capable of peer to peer text or voice communication and limited information retrieval. Mobile computer devices include different PDA and PIM devices and hand-held computers. These devices may allow installation of third party applications and are thus more general purpose and personalisable for specific users' tasks. They provide better user interface capabilities for creating new information than the communication devices. The mobile information devices include devices that are hybrids of the other two categories. They provide better access to networked information systems than mobile communication devices by providing larger display screens and better interaction devices. This category has two sub-categories: communicators that provide better means for creating new content, and smart phones that are designed only for information retrieval and browsing. Since it is envisioned that in the future the mobile computer devices will have some kind of wireless network access, the whole category will blend with the communicator category.

As the classification shows the mobile computer and mobile information devices have physically larger screens than mobile communication devices. The screens in these two categories are in the upper region of the target range for the user interface toolkit. The basis for the rest of the discussion is a device that has following characteristics [JSR-0000037, 1999]:

- 512 K total memory (RAM+ROM) available for Java runtime and libraries,
- limited power, typically battery operated and
- user interfaces with varying degrees of sophistication.

The user interface toolkit should allow upward compatibility from this base target. This means that applications written for the toolkit should work without modifications or re-compilation on systems that implement a much richer set of functionalities. However, larger screen devices will most probably have features that cannot be used by the API since suitable up and down scaleable abstractions cannot be found. This high-level problem is addressed later on in this thesis.

2.3 Differences with desktop user interfaces

In this section the differences related to output and input capabilities between desktop computer and hand-held device user interfaces are discussed.

2.3.1 Output

The most significant difference between desktop computers and hand-held devices is the size of the screen. Today's devices have enough computing power to present complex GUIs but the screen size limits the way GUIs should be presented. It is often impractical to simply apply the user interface for desktop computers to personal digital assistants (PDAs), cellular phones and other small screen devices. Hardware developments in this area will not solve the problem entirely since this and many of the aspects of hand-held devices are dependent on the capabilities of the users of these devices. Technological advances cannot solve all problems since human factors remain constant. For screen configurations this affects the physical screen size since the resolution of the screens keeps growing constantly. There is also research about solving the physical size issue completely; for example, there have been concepts about using head mounted displays with mobile devices.

The multiple window metaphor has become a common way to present the GUI on desktop machines. In this metaphor user tasks and applications are presented as windows. Users of multiple-window GUI systems often leave multiple unfinished tasks on screen as inactive or minimised windows. This

consumes much screen real estate and desktop becomes cluttered with number of windows in use. This can hinder the user during task performance also when using desktop sized screens [Miah and Alty, 1998]. In small screens there is no space for inactive windows; even minimised windows are a problem because the minimised windows are normally iconised and even icons consume too much real estate. Active windows hide icons and window management is cumbersome since there are often no mouse-like pointing devices in hand-held devices with very small screens. The stylus or pen is slightly different from a mouse, as it can draw very easily and it can select and point just as easily as a mouse, but it cannot hover or right-click, and double-click and dragging are error-prone [Tasker, 1999]. Leaving out window movement and resize capabilities normally solves these difficulties. The feature omission leads to a non-window GUI where one screen shows only one application "window" at a time and application can be changed using application keys (softkeys or hardkeys) or lists. The non-window GUI has been found efficient and is used for example in many PDA devices like Psion Series 5, Nokia Communicators and 3Com Palm Organizers. All these devices are implemented on different computing platforms and operating systems.

Because the multiple window metaphor or the desktop metaphor is not very effective for the small screen, device vendors have sought different metaphors. The deck of cards metaphor is one designed for small screens. This metaphor is used, for example, in the Wireless Mark-up Language (WML) [WML 1.1, 1999] of Wireless Application Protocol (WAP) [WAP Architecture, 1998]. User interface objects in desktop computing environments consume much screen real estate and thus cannot be represented efficiently as such on a PDA screen. For example, icons, multiple windows, pull-down and pop-up menus are GUI objects that consume too much space [Kamba *et al.*, 1996].

Jacob [2000] has classified user interfaces more generally. He separates user interfaces currently in use to command languages, menus, forms, natural language, direct manipulation, virtual reality, and combinations of these. Current desktop systems provide means for creating user interfaces that combine many of these styles. Mobile phone devices use often only one or couple of these styles, mainly menus with user input carried through simple forms. PDA devices have richer set of user interface styles; some devices even have software that uses direct manipulation. In practice, direct manipulation needs a pointing device and a fairly large screen. Small screen devices, like mobile phones and pagers, often have such small screens that direct manipulation is unsuitable. To date, natural language and virtual reality has not been used very effectively even in desktop environments. Virtual reality

requires a rich set of output and input devices so it is not a very interesting style of interaction for compact small screen devices. Portable devices that use virtual reality user interfaces will also need interaction and sensory equipment that are separated from the rest of the device; for example, head mounted displays and other wearable devices. This kind of additional gadgetry can be cumbersome to handle. An important aspect for the success of this kind of user interfaces will be that wearable devices will be sufficiently ubiquitous and hidden.

Natural language could be used in small mobile devices since it does not require large physical interaction devices, as text or speech could be used for both output and input. There are many unsolved problems related to natural language user interfaces, for example, in fields of speech recognition and semantic analysis of language. Even if technical problems will be solved it has been criticised that conversation may not always be the most effective way of commanding a machine [Jacob, 2000] [Dix *et al.*, 1993].

The main focus here is on the more traditional interface styles: menus, forms and direct manipulation. Jacob's classification provides us with an interface classification that is based on how the end user experiences the interface, and it is thus very relevant when designing application interfaces. On the other hand, the application programming toolkit provides interface building blocks for interface programmers. A good toolkit is such that it provides sufficient means for building an application interface that can use as many of these suitable interaction styles for the target device.

2.3.2 Input

The input method is another difference between PCs and small-screen devices. Small screen devices often do not have similar QWERTY keyboards or mouse-like devices than desktop computers. Kamba [1996] states that in small devices the keyboard is perhaps the most cumbersome of input methods. Cellular phones often just have standardised ITU-T [ITU-T E.161, 1995] and GSM [GSM MMI, 1998] keys. Styluses or pens and touch sensitive screens are often used for personal digital assistant type of devices. This input mechanism comes with several disadvantages. For example, it requires two hands to operate the device. One hand holds the device and other hand operates the device with a stylus. The pens are small and they can be easily dropped and lost. The pen also hides the screen area. It is not feasible to implement devices with very small screens that are pen operated. Also, other innovative input mechanisms can be created, like devices that handle the tilting of the device as an input mechanism [Rekimoto, 1996] and so on. Where a pen is used, handwriting recognition

technologies are often used as well, for example, Palm Computing's organizers and Apple Newton. In spite of the disadvantages of styluses, touchscreens have advantages. For example, Shneiderman [1991] has presented a list of advantages over other pointing devices:

1. Touching a visual display of choices requires little thinking and is a form of direct manipulation that is easy to learn.
2. Touchscreens have easier hand-eye co-ordination than mice or keyboards.
3. No extra workspace is required as with other pointing devices.
4. Touchscreens are durable in public access and in high-volume usage.

The third advantage is important for small screen devices since with a touchscreen there is no need for other keypads in the device; the screen can take the space of an entire side of a device since user interface components can be used instead of normal hardware keys. The second advantage is also important since the usage of the device is easier in lightless spaces because the user does not need to search for the often unilluminated keys. There is no need to design keys to be back illuminated, which reduces the size of the device, power consumption and therefore overall cost. Only the screens have to be back illuminated. Of course software user interface elements used as keys also have many limitations, the biggest probably being lack of tactile feedback. This can be somewhat alleviated with usage of sound as the user presses the screen keys.

A small screen also restricts the usage of a touchscreen: it cannot be used without a stylus in these devices because the user interface elements are often smaller than the size of a human fingertip. For example, selection of a text block is too difficult without a stylus. Shneiderman [1988] presents several methods for adding precision to selection without using a stylus. One method is a lift-off strategy that allows higher precision by showing users a cursor on the screen slightly above their fingers. Shneiderman states that with this method users could easily select targets the size of a pair of letters.

The differences in input mechanisms between the desktop computers and the portable devices and between the small screen device category itself lead to the fact that a portable user interface toolkit for small screen devices should be independent of any particular input mechanism. Because application portability is a major driver for applications the user interface toolkit should not even have any expressive power upon the input mechanism. The application programmer should not be able to build an application that states something about the input mechanism, for example, about mouse pointer

movement. This restricts the expressive power of the API since the user interface concepts must be highly abstracted. The fact that any particular input mechanism has not become a standard leads to this conclusion. If programmers have to deal with several different input mechanisms, application programming becomes too difficult and the application code size grows because applications need to have code for all input device cases. Often application programmers just make applications to work on specific devices and do not support a wide variety of devices. Since application portability *is* the key element and target of Java programming, this could have an impact on the success of Java in the hand-held market.

On the other hand, if the application programmer cannot access the whole functionality the device supports, the user interface toolkit limits the set of applications which in turn will restrict the growth of third party software. A device can provide access to the functionality in two different ways: either directly with manufacturer proprietary lower level API or indirectly with standard higher-level API. Often the application programmer can choose to use the lower level APIs of the device to program those parts that higher-level APIs do not allow. This is often a choice between application portability and all other aspects of the application, such as usability, speed and memory footprint. If the toolkit limits application programming, then the application code will easily become slow and bloated and the user interface will become hard to use. Some of the issues against application portability are resolved in time, for example, memory footprints and speed issues but some are not, like limitations in API abstraction.

Of course this choice is not relevant just in the context of small screen devices but also in the programming of desktop applications, although some of these problems either do not exist or have been solved in desktop computing. As in user interface programming, there are established user interface elements, and therefore de facto standard Java user interface APIs and an extensive installed base of proprietary Windows operating system APIs (WinAPI) implementations. This is not the case in the small screen device category since currently there are no de facto standard user interface toolkits in any programming language. The problem in the small screen device category is that the user interface elements of these devices are not established because there are many different manufacturers and device concepts. This leads to a conclusion that there is a need of a general, very highly abstracted user interface toolkit for applications that do not need any platform specific user interface elements. There is also a need for a separate lower level toolkit that has more expressive power. The higher-level toolkit is targeted for application

programmers that use the huge set of different devices as the market driver for their applications. On the other hand, the lower level API is used by developers that make specialised applications targeted for specific device categories and niche markets. Sun has already announced the concepts of API profiles on top of different Java platform editions. Profiles can also specify user interface toolkits since the Java 2 Platform, Micro Edition itself does not define any user interface toolkit. These and other Java related concepts are presented in the next chapter.

3. Technologies

In this chapter, two important application technologies, Java and Wireless Application Protocol (WAP) [WAP Architecture, 1998], are presented. The user interfaces for hand-held devices use and are built on these two technologies. The discussion herein contains sufficient background information about the technologies for understanding the rest of the thesis. However, the discussion mainly concentrates on the issues related to user interface aspects. First, the Java user interface concepts are presented with a discussion that is mainly concentrated on the AWT and Swing toolkits and their problems where application portability is concerned. After the Java user interface concepts, a section about dynamic layout management is given. Layout management problems specific to small screen systems are also approached in this section. After that, the chapter continues with WAP related discussion. This includes a brief presentation of WAP architecture so that the reader can understand the context of WAP applications better. The chapter ends in a discussion how these two separate technologies could be integrated and used together to leverage a more interactive and dynamic wireless application environment.

3.1 Current Java user interface concepts

Java is a programming language [Gosling *et al.*, 1996], a virtual machine (JVM) [Lindholm and Yellin, 1996] and common application programming interface (API) set. The Java Platform consists of the programming language, JVM, API set, class libraries that implement the API functionality and tools used in developing, compiling and error-checking the applications [Java Glossary]. The Java programming language is a simple, object-oriented, multithreaded and highly dynamic language. The origin of the Java technology is based on Oak, a technology created for television set-top boxes. Later the focus of the technology changed and the name was also changed. Java was designed for the heterogeneous network environment, the Internet. Because of this, the designers of the language had to make it secure and portable. The binary portability is guaranteed by the fact that the Java is interpreted. Every supporting platform must implement the JVM specification that defines the bytecodes that a Java compiler generates into the class files. This means that the language itself is not interpreted, but the compiled bytecode class files are. The heterogeneous environment forces Java to be independent of the underlying architecture also in other means. The platform independence is not only acquired by the usage of JVM but also with a way the Java APIs are designed.

The user interface toolkit is the part of the API that is used to create application user interfaces. Currently there are two publicly available and generally accepted user interface toolkits for Java, the AWT and Swing GUI programming APIs. Both are developed by Sun, although Internet Foundation Classes (IFC) from Netscape [IFC Developer Central] were used as a starting point for the Swing project.

AWT stands for Abstract Window Toolkit. It is a simple application programming toolkit that has been in the Java platform since the beginning. Swing is a newer and more powerful toolkit that is based on AWT. The basic difference in how these two toolkits are implemented is that AWT builds on top of the GUI widgets the underlying system implements, and Swing is implemented completely in Java classes on top of the Java graphics system. Swing is dependent on some primitives of AWT, as the graphics system. Figure 1 shows the stack of layers of an AWT implementation. Calls from AWT objects propagate through the stack to the native widgets. Every AWT widget has accompanying peer objects, one in the Java side and one in the native side. The native peer widget is created using the native user interface toolkit. Since application programmers write code that is dependent only on AWT objects, the applications are portable between all systems that have compliant Java AWT implementation.

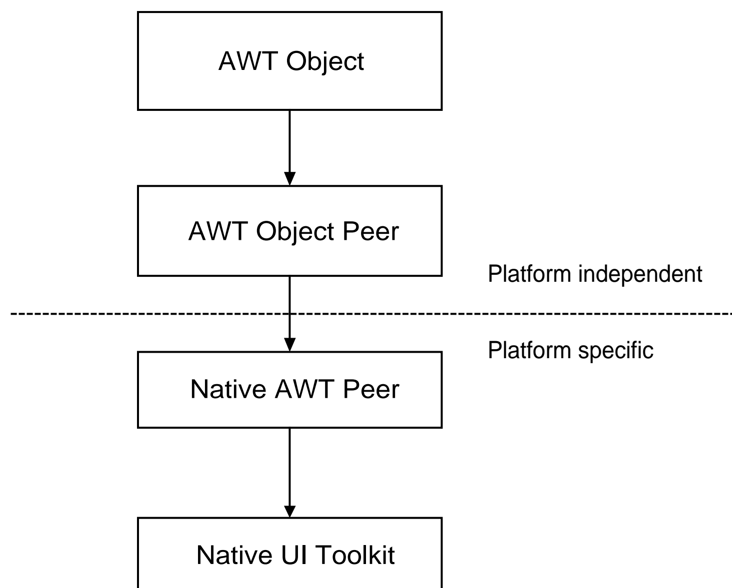

Figure 1. The layers of the Abstract Window Toolkit

AWT is based on the least common denominator design principle. The GUI concepts that AWT implements are available in all the common desktop environments, such as Microsoft Windows, Sun Solaris, various X Window System widget toolkits and Apple Macintosh. This limits the expressive power

of AWT. The widget set of AWT is listed in Table 2. Java uses the term component for widgets, since every Java widget is inherited from a single base class called Component.

Widget usage	***Class name (in java.awt package)***
Top level window	Frame
Drawing canvas	Canvas
Menus (with checkbox menu items)	*Descendants of* MenuComponent
Choice box (single item presented a time)	Choice
List of items (multiple rows presented)	List
Check box, radio button	Checkbox
Button	Button
Editable text field (single row)	TextField
Editable text area (multiple rows)	TextArea
Non editable text label	Label
Scrollbar	Scrollbar
Common file dialogue	FileDialog

Table 2. Widgets available in AWT.

Since AWT was designed by following the least common denominator design principle, there are problems in implementing AWT on top of systems that do not follow the desktop metaphor. Even if AWT implementation is compliant with the specifications and test suites, problems can still show up either as application portability problems or as usability problems when running the Java applications on these systems. Portability problems show up if applications rely on specific details of the system, such as low level events that are accessible from AWT. Usability problems arise from the fact that, to pass the compatibility tests, the systems must implement full AWT functionality even if the system does not support it. To support the functionality, for example, software emulation will be used. An implementation could also have bad mapping of AWT functionality to some system functionality that will give semantically different results. Even small variations in semantics could hinder the usability or result in unwanted behaviour on applications that rely on these functions.

PersonalJava (pJava) is a Java technology attempting to scale down the Java platform requirements so that it could be implemented on top of different kinds of consumer devices that still have display screens. Another target was that most applications written for pJava would be upward compatible with the full desktop Java platform.

PersonalJava defines a set of Java APIs derived from a standard desktop environment targeted Java platform. In addition pJava declares a few new APIs. PersonalJava specification specifies the ways the technology differs from desktop Java APIs. Current pJava is based on the Java Development Kit (JDK) version 1.1 API releases. The pJava specification relaxes some user interface requirement issues. For example, it is optional for pJava platform implementation to support multiple top-level windows. The following list defines the main differences between pJava technology and standard JDK 1.1 releases relating to the user interface issues:

- optional multiple top level window and dialogue support,
- optional support for resizeable dialogues,
- optional menu support (must be supported if windows and dialogues are supported),
- optional printing support,
- optional support for different mouse pointer images (mouse cursor in Java terminology), and
- optional support for ScrollBar and FileDialog classes.

Because the pJava user interface toolkit is derived from AWT, which was designed for the common desktop computing environment, it is not feasible for small screen devices. In practice, both PersonalJava AWT and full AWT require a pointing device of some kind. At the lowest level, this can be implemented using a virtual mouse pointer that is a keypad navigated mouse cursor. This is a very cumbersome input mechanism since it does not support the same feel than normal mouse devices do. The speed of moving the cursor is the main limitation of virtual cursors.

The Java 2 Platform, Micro Edition (J2ME) is a new Java platform edition for consumer electronics space by Sun Microsystems. It is supposed to be a feasible platform for different kinds of consumer devices from mobile phones and pagers to set-top boxes. J2ME does not specify a user interface toolkit as it was seen that consumer electronics are not capable of supporting AWT or Swing because of the differences with standard desktop computing environments noted under section 2.3. J2ME introduces the concepts of *configuration* and *profile*. There will be several different configurations and many profiles. A configuration specifies a minimum set of core functionality for J2ME platform. This functionality defines the APIs and the minimum level of VM support if a new scaleable virtual machine called KVM is used. Every J2ME platform implements one of these configurations, which can be extended by several profiles. A profile is a set of APIs that are somehow related. For example,

JavaPhone [1999] specification is a profile for smart phones and communicators that are capable of presenting pJava AWT. A profile does not necessarily specify user interface APIs; it can also specify different system APIs that can be used in vertical markets. For example, JavaPhone is targeted for phone manufacturers and therefore contains many telephony related APIs like Java Telephony (JTAPI), address book, calendaring and power management APIs.

J2ME allows the user interface toolkit to be customised for different sets of devices and *shared* among the devices in the same category. This was not possible earlier due to constraints set by the Sun Microsystems licensing model. Application portability is preserved within a profile but not necessarily within the whole J2ME platform. It is suggested that J2ME would be as much as possible upward compatible with Java 2, Standard Edition (J2SE), the successor of Java 1.1.

3.2 Layout management

Dynamic layout management has become a normal support mechanism for creating platform independent GUI in widget based toolkits. Dynamic layout management means that an application can change in runtime its layout of widgets and other presentation components. When an application uses dynamic layout, the application presentation will be able to adapt to different visualisation configurations. The use of absolute co-ordinates can lead to troubles even in single machine architecture. For example, if you create a dialogue on a Windows machine using small fonts, and then run it on a machine using large fonts, the layout can become completely disordered. Between machine architectures or between different hardware and screen size configurations, not only the size of the fonts can vary but also the size of the widgets themselves.

Toolkits implement dynamic layout management in different levels of support. In the lowest level, dynamic layout management is left totally for software developers. The normal level of support includes toolkit integrated layout management. Additional support includes, for example, configurable layout management or user defined layout managers. Java standard user interface toolkits AWT and Swing as well as other commercial user interface toolkits have integrated support for the layout management inside the toolkit.

In AWT and Swing, dynamic layout management is handled by layout managers. Every widget container object has an associated layout manager object that handles the way the widgets should be laid out in the container. In other words, *Container* is a Java widget that contains other widgets. Container includes a reference to a layout manager object that tells how contained widgets should be laid out in the screen space that is reserved for the container.

Every layout manager object implements the LayoutManager interface. Container uses the methods through the LayoutManager interface to query for the specific co-ordinates for every widget. If layout manager is set to null, then the widgets are laid out by the absolute co-ordinates set in them.

Table 3 shows the layout manager implementations in Java 2. Two layout managers, ScrollPaneLayout and ViewportLayout are not shown since they are tightly coupled to their respective JScrollPane and JViewport components and they are virtually useless in other contexts. In the first column is the class name of the layout manager implementation, in the second the user interface toolkit the class resides in, and in the third a description of how the layout manager lays components in a container.

Layout manager	***Toolkit***	***Description***
BorderLayout	AWT	Arranges and resizes components into five regions: north, east, south, west and centre
CardLayout	AWT	Acts like a deck of cards, only one component visible at a time
FlowLayout	AWT	Arranges components in a left-right flow, like lines of text in a paragraph
GridLayout	AWT	Lays out components in a rectangular grid, all the components have same size
GridBagLayout	AWT	Lays out components based on a comprehensive constraint object that defines relative positional information
BoxLayout	Swing	Allows multiple components to be laid out either vertically or horizontally
OverlayLayout	Swing	Arrange components on top of each other, overlapping where their dimension require it

Table 3. Java standard layout managers.

Although layout managers are only standard way to do portable GUI between desktop systems the standard layout manager mechanisms of Java often do not scale down to small screen systems as such. For example, the FlowLayout drops out a component completely if the component can not be fitted to the screen space and GridLayout requires a great deal of screen real estate. More generally speaking, the layout managers are a method to do portable GUI but they do not guarantee it. The problem is that the layout managers are concrete implementations, which means that they are implemented in the same way in all platforms. Standard LayoutManager implementations are implemented in very generic code and do not have

information about specialities of a platform the application is executed on. Application code is dependent on selected layout manager implementations. Furthermore, layout managers often assume some characteristics about height, width or ratio of the dimensions of a container, for example, border layout needs a fairly square space and will not work if the space is limited either in horizontal or vertical dimension.

The JDK 1.1 based AWT introduced a new LayoutManager2 interface. In the JDK 1.0 API LayoutManager interface did not contain methods that accepted constraint objects. These objects are needed to create more flexible layout managers. Since LayoutManager was an interface and not an abstract class, a new subinterface had to be created and Java platform code now has to make checks whether supplied LayoutManager implement LayoutManager2 methods [Davis, 1999]. Abstract class is safer to change later on. Constraint objects are descriptions to the manager about how it should treat the associated component. For example, simple constraint objects are Strings like "NORTH" and "SOUTH" used in BorderLayout. Constraint objects are used to configure the layout manager for a specific purpose.

Since a container has dependencies only to the LayoutManager interface, custom layout managers can be created by implementing the LayoutManager preferable by implementing LayoutManager2. The portability of an application that uses custom layout managers depends on the way the layout manager implementation is done. If a layout manager uses absolute co-ordinates, it is no better a solution than setting a layout manager of a component to null and laying out the components by hard coded pixel values.

The layout manager framework assumes that the components can be laid out freely in every platform. This is not the case, for example, in the embedded device market. Layout is an important part of the look and feel of the user interface and terminal manufacturers might want to preserve their native layouting policies also for third party software. If toolkits allow third party software to use similar layout managers that reside, for example, in AWT and Swing, it means that third party applications define the layout completely and layout principles of a device are not mandated. Moreover, if a layout is optimised for the layout style of one specific device, then the styles of other devices are most probably broken. Terminal manufacturers have designed their native user interface layout to be used with the input devices the terminal uses. For example, a keypad as a sole input device restricts the use of different layouting schemes, since it is much harder to navigate between user interface widgets without a pointing device.

As already discussed, all standard and custom Java layout managers are directly connected to a specific implementation class. When using any layout manager the application code references the concrete implementation object and that object is set to container object. There is no framework for creating abstract layouts in the way, that for example, instances of *java.util.Calendar* class can be created with a factory method. Java API allows Calendar objects to be created depending on the locale the application is executed in. The layout managers could be abstracted in similar way. Application programmers should just state the abstract type of layout they want to use and create the concrete object using static factory methods that return the concrete layout manager created specifically for the device or platform the application is executed on. This object has the implementation code to lay the components according to the screen space, resolution and so forth on this specific device or platform. This kind of solution is more scaleable but it is also more restrictive, since the layout of widgets is completely known only in execution time. This shifts many of the traditional GUI design problems from application vendors to system or device manufacturers. In the mobile phone business many terminal manufacturers even state this kind of solution as a requirement. On the other hand, terminal manufacturers want to control the user interface of the devices and do not want several different kinds of user interface look and feels on their terminals, and they also want to leverage third party applications.

The primary problem with different screen sizes is that the user interface designer's design will be violated when the user interface is run on a screen size it was not designed for. Existing dynamic layout management technologies can only retain the design with rather small changes, like that of the widget size being slightly different. Jones and Marsden [1997] have studied different ways to retain the screen design when larger changes happen in the screen size. Although the context of Jones and Marsden is slightly different than in this thesis since they discuss about media rich textual pages, such as web pages, the problems are very similar in toolkit based user interfaces. Jones and Marsden present four potential solutions for adapting interfaces to the small screen that have been suggested in the previous and on-going research. Jones and Marsden also present a *new* approach that is an attempt to overcome the limitations in these earlier solutions. The discussion below will adopt these *five* solutions to the problem domain of this thesis.

1. *Explicitly customise the user interface for the smaller screen.* This is in fact what is done today so it provides no solutions for the problems. Explicit customisation will also be needed in the future, at least to some degree,

since even if the screen size problems are solved other capabilities of devices will remain different.

2. *Using style sheets.* Style sheets or similar techniques could also be used in toolkit based user interface construction. The problem with style sheets is that they just isolate the explicit customisation from the content, in this case from the user interface description. Separate style sheets still need to be created for every target device or category. These kinds of style sheets are not the solution for user interfaces. The problem should be investigated in a more general view. A more general solution would be to separate the user interface visualisation from the user interface semantics. The style sheets are one implementation for this general solution, but not the best. A more general solution is discussed later on in this thesis when concrete solutions for the problems are described in Chapter 5.
3. *Data visualisation.* Very large data sets are problematic to display even on desktop screens. Different interactive visualisation techniques, like panning and zooming, have been developed to deal with this problem. The different visualisation techniques do not scale very effectively to small screen. In pan and zoom visualisation the data cannot be zoomed out in as many levels as in desktop screens since the viewport is much smaller and because of the small resolution the data becomes easily too obscure. The techniques also work on static data and cannot be used in normal graphical user interfaces. Nevertheless, if the toolkit is sufficiently layered then the implementation could use different kinds of visualisation techniques to present the GUI on screen.
4. *Model-based user interface generation.* An interface can be generated automatically or semi-automatically from an interface model defined by application developers. This model does not specify the visual look of an interface. The model could specify, for example, the domain specific user interface objects, user tasks and design relations between the objects [Puerta, 1997]. If the visualisation is generated automatically, then the model-based user interface generation needs visualisation rules that realise the visual part of the interface. These rules are implemented on the user interface toolkit of the specific system in which the user interface is executed on. The rules are thus independent of any specific application. The complexity of the visualisation rules are dependent on the complexity and power of expression of the system used to describe the models themselves. If the visualisation is generated semi-automatically, with a developer assisting the process in design time,

then the user interface becomes highly targeted to a specific system. This kind of environment was presented in Puerta [1997].

5. *Preserve the original layout characteristics.* Jones and Marsden introduce this as a new approach. They do not give any concrete answers as to how the solution for this approach is reached. For application usability this approach is, of course, the best alternative because the user interaction remains similar. However, it is almost impossible to implement if the presented content does not already scale down well. Content that is scaleable is well structured and its visualisation is separated from interaction semantics. This would apply to similar solutions, such as in items 2 or 4.

For the problem domain of mobile devices some kind of a model based interface generation is most suitable, because of the constraints set by code mobility and different application contexts. This introduces rather dramatic changes to the model of user interface creation. Application designer no longer designs the visual look of an application, but the semantic interaction model. This is discussed further later on in this thesis.

3.3 WAP technology and its user interface concepts

The Wireless Application Protocol (WAP) was designed for mobile devices with limited display and user input capabilities, small memory, low processing power and low bandwidth network connection. Currently WAP is becoming an important element of application execution in mobile terminals since there is no other recognised and open competing technology. WAP can be used also from normal PDA type devices that are not directly connected to network but to whom the network connection is provided via an external mobile phone or radio unit or standard landline network. WAP is developed by WAP Forum [WAP Forum Website], an industry association, currently comprising over 200 members.

WAP is a set of protocols and conventions that can be used to create manufacturer independent applications for mobile phones. WAP defines two essential elements of wireless communication: an end-to-end application protocol and an application environment based on a browser. The protocol is a layered communication protocol that is implemented in each WAP-enabled terminal. Another part of the current WAP version 1.1 specifications defines an application environment. The key parts of the environment are a markup language that can be used to define the content and a scripting language for defining simple behaviour for the application. The markup language is called Wireless Mark-up Language (WML) and it is defined as an application to the

Extensible Mark-up Language (XML). XML was defined by the World Wide Web Consortium [W3C]. XML can be seen both as an extremely simple dialect and as a subset of Standard Generalized Mark-up Language (SGML) [ISO 8879, 1986] for use on the World Wide Web. XML is a meta-language, which means that it is a language that defines rules for defining different kinds of structured languages like HTML and WML. WML is the language that is used to create user interfaces for WAP applications. The behaviour of the applications can be extended with WMLScript scripting language. WMLScript is a subset of the standard scripting language ECMAScript [ECMA-262, 1998].

The model of how WAP works is similar than in normal World Wide Web technology. The technological similarity is presented in Table 4. The biggest difference is that the client or terminal does not have to be capable of parsing WML. WAP defines a binary version of XML to be used in low bandwidth channels (WBXML) such as wireless networks. The WAP gateway is a server that can encode the WML content to WBXML before it is sent to a low-bandwidth network. Figure 2 shows a general model of a WAP request. A micro-browser on a terminal is used to request a specific URL. The URL is supposed to reference content that the user agent (UA), in this case the micro-browser, is capable to present. Normally the content is a WML page stored in or generated by a web server. The UA uses the WAP protocol stack to contact the web server. The request is carried through a WAP gateway that resides between the wireless network and a landline network. The gateway generates a normal HTTP request to the web server, which resides in landline network, normally in the Internet. The server responds with a WML content and mediates the packets to the WAP gateway. The gateway encodes the WML packets to WBXML and sends the encoded content to the terminal and finally to the UA that requested the URL. The WAP gateway thus builds a bridge between a telecom and a computer network.

Because of this architecture, the content the URL references can also be something else than WML, Java classes, for example. Terminals of course need UAs that can recognise and handle the content. For Java content this means that there is an implementation of Java VM and class libraries in a device. The WAP application environment also needs to specify the application context issues for Java. Nevertheless, this could be one method for distributing user interfaces to the WAP enabled devices. Java and WAP interoperability is discussed more in section 3.4.

Concept of WAP	*Concept of WWW*
WML	HTML
WMLScript	JavaScript, (ECMAScript)
Hypertext Transport Protocol (HTTP)	Wireless Transaction Protocol (WTP)

Table 4. WAP technologies with corresponding WWW technologies.

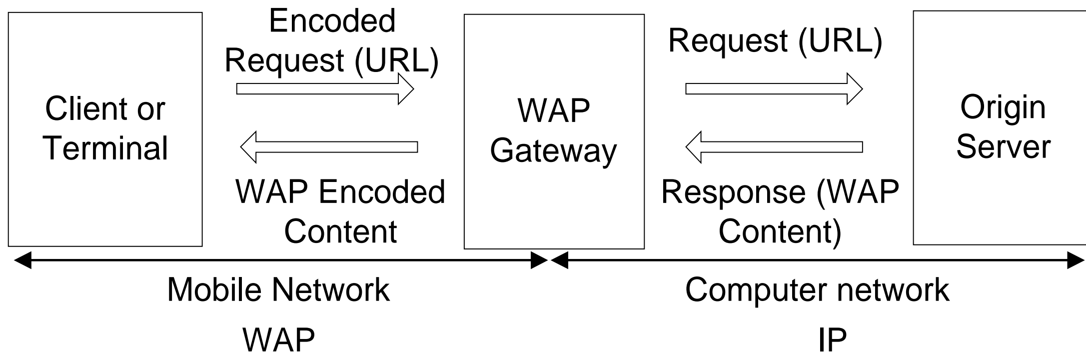

Figure 2. Transmission of WAP request to a server and content back to a terminal. Adapted from [WAP Architecture, 1998].

Since the WAP protocol stack is sufficiently layered the user interface discussion can be narrowed to WML and to some parts of WMLScript. WML is used to define the user interface for WAP applications. It is very similar to HTML, but there are also some differences. WML is designed especially for small screen devices. It uses the deck and card metaphor illustrated in Figure 3. Every WML file contains one WML deck. The deck can contain one or more cards. One card is displayed at a time to the user of UA. A card can contain text and other WML user interface elements. WML and WMLScript contain methods to change the active card. This enables navigation from one card to another. Specifically, WML contains event bindings, timers and similar anchors than in HTML ('A' tag) that can be used to support navigation.

The user interface concepts of WML are highly abstracted from any particular implementation. Basically, to be capable of supporting WAP, the user interface of a device needs some input mechanism for the user to input textual content and some output mechanism to notify the user of the results. The input mechanism can be any normal computing technology oriented mechanism like keypad, stylus or mouse, or some more exceptional mechanism like speech input.

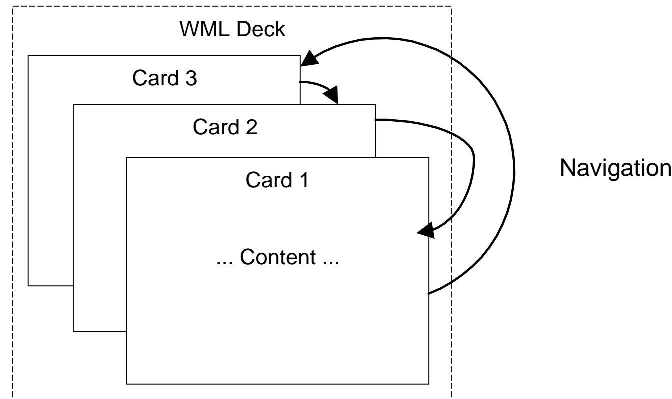

Figure 3. The deck of cards user interface metaphor used in WML.

The content that can be presented with WML is very limited. The main content is text with simple formatting elements, but in addition small images and tables can be specified, although terminals may not display them at all. Input elements of WML are also very restricted. Input elements the WML contains are

- input text field with variable formatting constraints and
- single and multiple selection lists.

Input elements can be used to query information and send the results to server. WMLScript can be used to pre-process the information (see WML Reference [1999] for complete description of elements in WML).

WAP is an attempt to achieve platform independence through the usage of rather abstract user interface elements, although the elements are still quite heavily tied to visual presentation. This has already caused problems when implementing WAP to different platforms. One other problem is that WAP actions cannot be classified in any way. At least two classifications for actions would have been needed: (1) actions that set the application state and (2) actions that invoke some application operation. These problems need to be addressed when designing procedural user interface specification mechanisms, like when designing the Java user interface toolkit for hand-held devices. The problems are related to separation of visualisation and semantics and are discussed in more detail in section 5.2.2.

3.4 General WAP and Java interoperability

This section discusses the application environment of WAP that will most probably use some kind of a Java user interface toolkit in the future. This means that a hand-held user interface toolkit must be usable in the context of WAP application environment. This discussion illustrates one special dependency for the hand-held user interface toolkit. It is also interesting to note that WAP itself

is an abstract application environment that was designed to be implementable on top of many small hand-held devices. Because of this, some will see Java as a technology competing with WAP. The discussion here does not take part in this rather political discussion and it is seen that these two technologies can co-exist and enhance each other.

WAP was designed for devices with limited resources. In time, even the low-end devices will have enough resources and current WAP limitations can be relaxed. As mobile terminals become more prominent WAP will need more expressive application tools for the terminal side. The current scripting language, WMLScript, is very limited. For example, dynamic allocation of new objects was removed from it because of limitations in the amount of heap memory available in most target devices. Java is one obvious option for bringing advanced functionality to the WAP application environment. There have been ideas that the Java user interface targeted for small screen devices could use the same user interface functionality or user interface widgets already used for displaying WML elements in WML browsers. Therefore, if the standard user interface elements of corresponding vertical Java profile are similar as in WAP, the integration of WAP and Java could be much easier.

There are several high-level possibilities to make WAP and Java interoperable. Five alternatives are introduced below:

1. Introduce Java as the WAP programming language, replacing WMLScript,
2. Introduce a concept similar to applets in Internet browsers to WAP,
3. Introduce WMLScript and a Java interoperability API such as Netscape's LiveConnect [Hoque, 1997] for JavaScript and Java interoperability,
4. Introduce Java class files or JAR files as a defined mime content type in the WAP application environment, and
5. Introduce a Java API that can access the WAP protocol stack and can produce WML content.

Note that these are not competing approaches; in theory the most powerful application environment would have implemented all the above elements. However, because of the resource limitations of terminals in the near future all the above functionalities cannot be implemented. It is also not sufficient to implement just one of the presented ideas. For example, the first element does not specify any user interface interoperability. If something similar to element five is not specified, then Java functionality cannot be used fully because the full power of WML is not available to applications. If just element 2 is specified without any additional APIs, then the Java could be used only as in the

sandbox model that is used in web applets, thus the wireless terminal functionality cannot be used through a Java code. If fine grained security is not implemented in devices, the system could require human interaction to restrict some application operation. For example, when an application wants to make some network operation, like a call, then the system will present a dialogue to the user that will initiate the operation. This will provide a simple way to provide security.

Current JavaPhone API is not implementable in low-end devices since it has a too big memory footprint. Element 3 would allow Java to have access to WMLScript libraries and variables in WML deck. This can be used for example with element 2 to allow more expression power for wapplets. Element 4 is similar to element 2, but there the Java content is not embedded in a WML card, being presented in UA some other way.

The choice of how Java and WAP should interoperate affects the small screen user interface concept of network connected devices. Since network connection is becoming a more and more important feature for hand-held devices, this decision affects the majority of small screen devices on the market in the future. There are several companies, industry coalitions and forums, and at least few standardisation bodies that are affected in this kind of work.

4. Different user interface toolkit concepts

In this chapter, different concepts in implementing a user interface toolkit specification are presented. Different concepts are discussed from two point of views: portability and power of expression. These two things are often competing. If a toolkit and applications written in it are highly portable, then the expressive power of the toolkit is more limited and the applications written with it are more general and cannot use any special features of any particular device or platform.

There are several opinions on what is a good toolkit and these two previously presented views are often stated. Traditionally, three methods have been used to achieve programming portability: *intersection*, *emulation* and *abstraction*. When implementing the first generally accepted virtual toolkit, the XVT, Rochkind [1992] reasoned that abstraction is the most powerful of all these approaches and intersection, also known as the least common denominator, is the simplest. When using intersection, programmers just restrict the use of functions that are not available in all systems the application should execute on. This method works well when the systems are nearly similar. For example, intersection has been widely used when programming to systems that implement different nuances of the C language, like ANSI C and K&R, or when building a user interface that should run on different Microsoft Windows operating systems like Windows 3.X, NT and 95. Intersection has also been used as a design principle for virtual toolkits. In virtual toolkit space the term least common denominator is more well-known. Rochkind noted that intersection does not work well in platform independent GUI programming, because different platform APIs did not evolve from a common base and have hardly anything in common.

The second traditional method, *emulation*, involves implementing one API in terms of another. Emulation has been used to achieve GUI portability. There have been products that implement one platform, for example, MS-Windows API on OS/2 Presentation manager. The problem with emulation is that the native APIs are normally designed for one platform so it is hard to make the emulation library efficient. In addition, normally the native APIs are not very convenient programming interfaces anyway.

Rochkind reasoned that the abstraction is the only viable method to build a GUI portability toolkit. With abstraction, the native APIs that cause the portability problems can be totally abandoned and a new higher-level API can be introduced. This is actually what is done in Java. After XVT, a lot of research has been done for platform independence. Rochkind did not realise that the

three methods for portability are not exclusionary. For example, Java AWT is a totally new abstraction, at least in Rochkind's terminology, and the least common denominator is used as a design method for the *features* of the API and not as a programming method.

Figure 4 shows two methods to add portability to the API, abstraction and the least common denominator programming model. Adding portability means two things here. The first is to add portability to the API so that the API is easier or indeed possible to implement on different computing platforms. The second issue is adding portability of applications so that the programs written in API can be executed on different platforms and give anticipated results. Normally, when you add portability, the expressive power of API decreases. The method chosen to add portability or expressive power has a relevance, but it depends on the case of which method gives the best results. Often, when *de facto* or *de jure* standard user interface toolkits, or more generally any standard APIs, are designed, it is not known what platforms the API should work on. This is of course a common problem in software engineering as a whole: what is the right level of abstraction? This design problem is even more difficult in small screen device category because of the limited memory, processing resources and diversity of the device features.

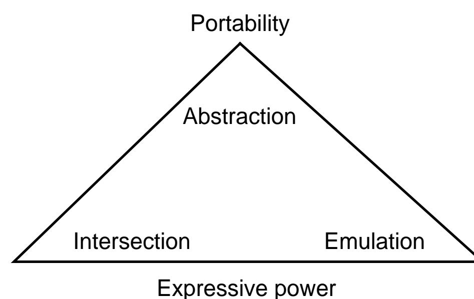

Figure 4. The three methods to achieve portability.

4.1 A subset of AWT or Swing

One possible approach to specifying the small screen user interface is to subset the Abstract Window Toolkit (AWT). Currently the PersonalJava user interface is a subset of AWT, and because JavaPhone specifies PersonalJava as a base API set, it is seen that AWT can be implemented at least on a Communicator type of device. Symbian device families [Symbian] specify that the Communicator types of devices have displays with sizes of approximately 320 x 240 and up for portrait displays and 640 x 200 and up for landscape displays.

The PersonalJava subset retains upward compatibility. This means that an application written for pJava should work without modifications or re-

compilation on systems that run a standard edition of Java. If this kind of upward compatibility is needed, then a subset can only subset the high-level user interface components and leave the underlying set of classes unchanged. This is because all the higher-level classes depend on the lower level classes. For example, underlying concepts that are still in public API of AWT like Toolkit and Graphics need to be implemented as such because all the widget classes depend on them.

To port the Java AWT on a new platform, platform-dependent implementations of the following public Java API classes are needed

1. java.awt.Toolkit,
2. java.awt.Graphics,
3. java.awt.Image,
4. java.awt.FontMetrics,

and implementation for each widget peer in package java.awt.peer. This is not a straightforward process because native event system notifications must be translated to AWTEvents and the dispatching of these events for AWT processing must also be handled. The image system also needs a lot of background classes; for example, implementations for ImageProducer and ImageConsumer are needed.

A subset of AWT could be built that eases the portability of existing applications to small screen devices but binary compatibility would not be possible. The main problem is that it would be hard to build AWT applications that are portable within different devices that would implement even the *same* subset. Because of this, an AWT subset is not seen as the right choice for a more general user interface toolkit.

4.1.1 A subset of AWT for WAP user interface elements

The Nokia WAP Toolkit [1999] is a WAP development environment written completely in Java. It contains a fully functional WAP stack and WAP browser. The Nokia WAP Toolkit demonstrates that it is possible to implement a WAP style user interface with AWT; therefore it is theoretically possible to specify an appropriate AWT subset for WAP-class devices. A WAP user interface is defined in elements of WML markup language. Because these elements can be implemented in many possible ways, they are implementation independent as noted earlier in section 3.3.

There are many problems with this approach. One problem was already presented in section 2.3.1: AWT was originally designed for platforms with windowing capabilities. PersonalJava subset of AWT specifies that class

java.awt.Window and all associated windowing classes are optional in implementation. This and many other optional classes and methods result in a platform that is more difficult to build applications upon. The programmer has to make tests and alternative solutions for all the different PersonalJava implementations. Normal WAP style devices do not have windowing capabilities and therefore it would be awkward to have windowing as optional feature in the API if the majority of applications would be developed for devices without the feature. The input mechanisms have the same problems: WAP devices have limited input capabilities and AWT is built heavily on the notion that a mouse or other pointing device can be used as an input method. In practise, AWT cannot be heavily subsetted since there are many concepts such as the event mechanism and layout system that cannot be removed without removing upward compatibility.

An AWT subset for WAP-class devices would really just be a set of AWT style wrappers around a WML element style user interface API. It would be much more limited than pJava subset of AWT and therefore applications should have been written directly to that API and quite possibly would not work on desktop AWT or in other small screen devices that have for example pJava.

4.2 A toolkit based on WAP user interface

Section 3.3 described the user interface concept of WAP. WML is used to create user interface in WAP applications. The WML specification only specifies semantics for WML elements; for example, the WML element INPUT is used to provide a user means to input text to an application. WAP does not specify how the user interface should be realised for this element or how the input should be physically done. There is a similar need for this kind of a user interface toolkit for Java applications. One solution to allow many small screen devices that are capable of running Java to have unified user interface API would be to implement an API that represents a similar set of user interaction concepts that WAP does. This API would also be very efficient and straightforward to implement on devices that have WAP browsers, since the Java user interface could use the same native user interface components than the WAP browser does. On the other hand, this way the WAP browser could be implemented completely in Java and this would provide terminal manufacturers a way to update the browser software more easily after the product has been shipped. Figure 5 presents an example class diagram of this kind of API. Associations of primitive classes are not shown in the diagram.

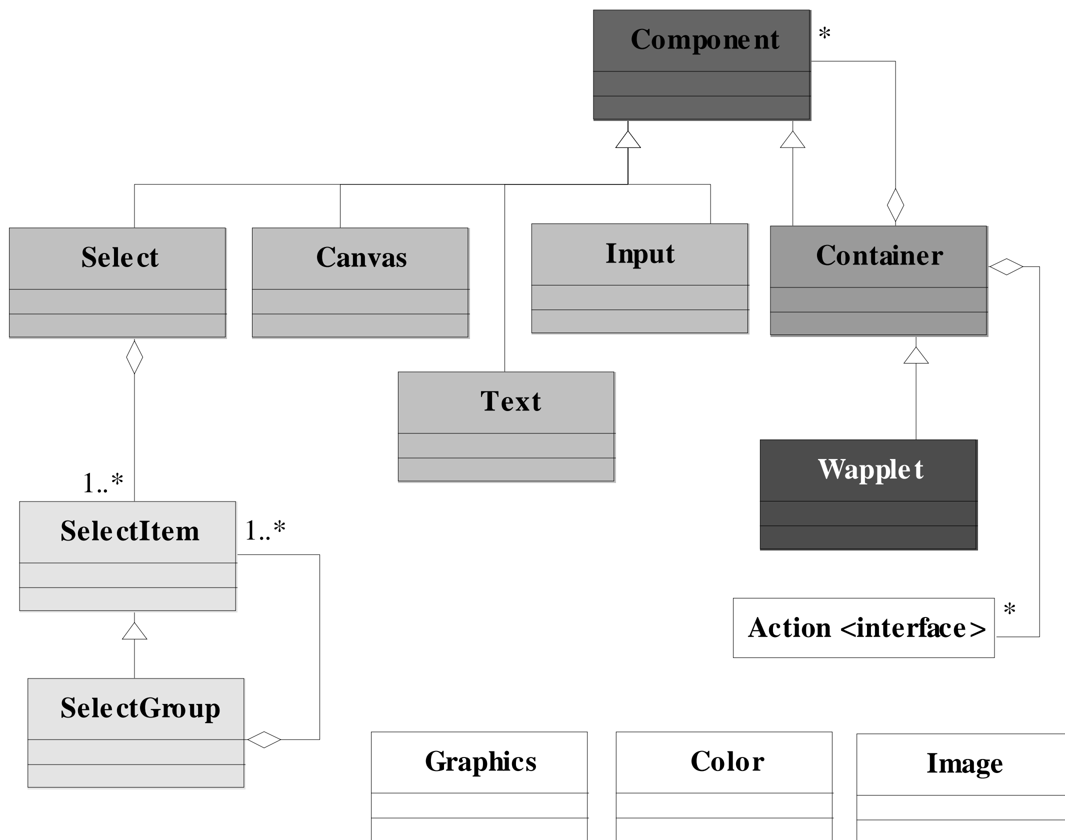

Figure 5. Example of API highly emphasized towards WAP.

4.2.1 WAP concept based

One solution for a GUI would be to adopt the concept of user interface implementation independence from WAP. This would mean that the API classes would not be straight representations of WML elements or user interface widgets realised by WML, but lower level constructs that could be used to create, for example, a WAP browser. These lower level classes would then allow more expressive power than the higher-level classes that are straight mappings of WML elements.

4.2.2 Wapplets embedded in Microbrowser

Wapplet is an idea to provide applet like Java applications in WAP microbrowser context. The term has been used in WAP Forum discussions [WAP Forum Website]. This section gives a more formal discussion. A wapplet would work in a way similar to a normal applet embedded on a web page, although, since screen real estate is very limited, it would be best that the wapplet would take the whole browser area. Since WML does not specify generic embedded objects, some mechanism is needed to reference wapplets

from a WML document. Hypertext markup language (HTML) 4 specification [HTML 4.01, 1999] uses OBJECT tag for generic embedded object inclusion. WML could adopt this tag from HTML. This would allow embedding a wapplet in a WML document, in the following way, for example,

```
<object classid="java:MyJavaCard.class"
        codetype="application/java">
</object>
```

Alternatively, the content could be recognised as a Java application from the content-type header. The difference between these two is that in the former solution the WML Card contains the Java wapplet and in the latter the browser does not receive any WML content, the wapplet being the whole object that is delivered to a terminal. The latter is preferred since it does not require any modifications to existing WML specifications: browsers would only need to recognise a new content-type and perhaps only propagate execution to the Java VM. Note, however, that this new content-type should be specified in WAP application environment specifications. The former solution would be a better alternative if there is a need for WMLScript and Java interaction, because the wapplet WML element could contain variables and parameters such as the name of the component embedded in WML Card and the classpath of dependent class files. The content-type recognition is not tied to any single markup language; this mechanism can be used also by services that make use of a different markup language than WML. Content-type recognition can also be added to Web browsers in desktop environments. The problems with parameters can be resolved using some packaging mechanism, like JAR files. JAR specification defines a manifest file [Manifest Format, 1996] that can store parameters and dependency information. Some new manifest attributes could be added.

4.3 Configurations and device profiles

The Java 2 Micro Edition (J2ME) is deployed in several configurations. Each configuration addresses a particular “class” of devices and specifies a set of APIs and Java virtual machine features that users and content providers can assume are present on those devices. Application developers and content providers must design their code to stay within the bounds of the Java virtual machine features and APIs specified by that configuration. In this way, interoperability can be guaranteed among the various applications and devices in a particular class.

Configurations include components of the *java.io*, *java.net*, *java.util*, and the *java.lang* packages; thus no GUI toolkit is mandated in J2ME. Profiles are sets of APIs and a specific configuration that provide domain-specific capabilities for devices in a specific vertical market.

J2ME Kauai Virtual Machine [KVM, 1999] is an optimized VM for resource limited consumer electronics space. KVM has the same set of functionalities than the standard JVM but many of the features are optional. Each J2ME configuration specification states what KVM features must be in the implementations of that specification.

Now it is possible to have a more standard Java GUI toolkit in a device, although the toolkit must be accepted through the JCP process defined by Sun. In Chapter 5, features for a possible GUI toolkit solution are presented.

4.4 Lower level user interface API

In AWT and Swing the class *java.awt.Graphics* provides an API that has services to draw graphics primitives such as single pixels or lines on screen. This kind of an API provides an access to the graphics system of the native platform. Similar rather direct API to graphics system functionality is provided in nearly all general application user interface toolkits.

Normally graphics drawing is targeted on a single user interface component. User interface API often provides a clean canvas component for drawing graphics primitives on it. The level of graphics drawing support varies between toolkits; for example, the graphics system can be used only internally or its services can be provided for application programmers. A toolkit could also introduce higher-level features like support for text, images, animations and graphical transformations.

Using the graphics system, programmers can, for example, extend the built-in widget set. Graphics services are a required feature in an adaptive application platform if the standard widgets do not provide sufficient needs for application programmers to represent information they need in their applications. More specialised systems can restrict direct calls to graphics system; for example, mobile phone manufacturers often want to have full control over the look and feel and they want to mandate their own user interface style even for the third party applications.

A Java user interface API that provides graphics services can be used to render the widgets using pure Java code. This kind of solution is used in the Swing system and Palm Pilot implementation of KJava API Specification [KJava, 1999] presented in JavaOne 99 conference. Since many device manufacturers want to keep their own look and feel, the standard Java user interface API cannot have mandated implementation on look and feel. The

manufacturers still can provide the look and feel as pure Java implementation or optionally through the native peer widgets.

Hand-held devices cannot support the huge set of graphics features built into the desktop user interface toolkits. The minimum service for graphics class would be an operation that sets a pixel value on screen. Additional behaviour should be implemented to reduce the code that has to be written in applications. There is also a need for some method to query the information about screen configuration, for example, about the height and width of the display in pixels.

The JavaOne99 KJava specification [KJava, 1999] *com.sun.kjava.Graphics* class contains 16 public methods. Compared to the *java.awt.Graphics* class of Java 2 and to its 50 public methods, there is a significant change. Note also that in Java 2 there is the extended Graphics2D class that has an additional 40 methods. Because there is no upper level for functionalities, very fine judgement is needed when deciding upon the feasible feature set for the graphics system. The existing functionality of the kJava Graphics contains

- drawing a line and a rectangle,
- drawing a border,
- drawing a bitmap,
- drawing a string,
- copying a region on/off screen with AND, AND_NOT, XOR, OR, INVERT, OVERWRITE copy modes,
- clearing the screen,
- querying the height and width of drawing area,
- setting/resetting a drawing region, and
- playing predefined system sounds.

Colour support is missing from the list and it should be added since there are already some hand-held systems that have colour screens. This will also introduce a new software portability problem between devices with different colour support. This problem is similar than that experienced when desktop systems gradually moved to colour screens. However, it can be predicted that if a standard user interface toolkit will not support colours then the manufacturers that choose to have costly colour screens on their devices will implement non-standard solutions. This will create even more software portability problems. If colour is introduced, problems with portability can be decreased in API design by providing a set of colours that are mandated to map to specific colours or to specific levels of grey. The problem is of course that

some devices do not even have support for different levels of grey. Additionally, emulator tools could be provided, with which the developers can test their applications' behaviour with different levels of colour support. Because of the problems, the level of colour support the implementation provides is one of the few capabilities which the applications should be able to query from the toolkit. With the colour support in the feature set of the graphics system it should be well enough for hand-held systems also in the future since different higher-level support can be introduced later on top of this API.

4.5 Existing implementations

In this section, some existing user interface creation technologies are presented. First, some Java user interface toolkits targeted to the problem domain are introduced. Then, in the next two sections, markup languages are discussed as an alternative method to specify user interfaces. The section ends in a discussion about Spotlet system that the J2ME and KVM were based on.

4.5.1 Existing Java toolkits

There are several non-standard Java user interface toolkits targeted to different platforms created by different companies. In this section toolkits, that are targeted especially to hand-held systems are examined.

Waba is a VM technology and a class library set [Wild, 1999] designed for hand-held devices. It has been implemented on Palm and Windows CE devices. Waba uses a subtly different VM technology than standard Java and it does not support threads, exceptions, long integer and double floating point primitive types. The Waba VM therefore does not support full bytecode of the Java VM specification. Note that the KVM uses the same technique, limiting the features of the VM to provide momentum for Java in resource limited devices, although the bytecodes in KVM class files are a subset of standard Java.

The user interface toolkit of Waba was designed especially for hand-held devices. It has a minimum set of components designed mainly for devices that have touchscreens and pen input devices. The Waba user interface toolkit is very traditional in every aspect; it contains widgets that are direct representations of widgets presented on screen. Component layouting in the toolkit is based on pixel values specified by the application programmer. There are no dynamic layout management methods; for example, no layout managers to do the layout. This traditional approach implies that every application has to be separately ported to other devices even if the devices have only small differences in screen configurations.

Bongo is another commercial user interface toolkit. It was designed by Marimba for its Castanet product. Bongo is also available as open source version FreeBongo. Bongo was created for desktop systems and for their resources. The Widget class, for example, which is the base class of all the user interface components, contains about 170 non-deprecated methods. Despite the fact, the party involved in development of FreeBongo has suggested that Bongo would be the user interface toolkit for J2ME. Bongo is also highly tied to the AWT primitives, like Graphics, Color, Font and so forth. Since Bongo is heavily widget based it provides little that is new for the hand-held device API.

Kalos Espresso by Espial Group is a user interface toolkit designed especially for consumer devices running on the PersonalJava platform. It is a collection of lightweight user interface components that are implemented on top of the AWT graphics system in a very similar way than in Swing. In addition, Kalos provides a special image resource management system, which ensures that similar images are never loaded more than once. The system achieves this by using a shared image table. Kalos does not provide any solutions for the problems of standard hand-held device user interface toolkits. Since it is heavily tied to AWT, it contains the AWT layout manager system and has a similar presentation oriented component model as AWT. Image resource management can be handled in platform level rather than in the user interface toolkit.

The conclusion is that all these toolkits are mainly slimmed down versions of traditional desktop environment toolkits. The widget set they provide is designed for consumer devices so they are good tools for creating better user interfaces for consumer devices. However, these toolkits fail to support portability and accessibility which are the key features for standard user interface toolkit for hand-held devices.

4.5.2 User interface markup languages

There are several subsets of HTML specifications designed for hand-held device market, for example, Compact HTML [C-HTML, 1998], Hand-held Device Mark-up Language [HDML, 1997], Tagged Text Mark-up Language (TTML) and *i Mode* HTML. All of these were or are developed in a single company or by a small group of companies; thus no standardisation body was present in the development of these specifications. C-HTML was developed by Access Co. Ltd., HDML by Unwired Planet (later known as Phone.com), TTML by Nokia and *i Mode* HTML by NTT DoCoMo. Both Access and Unwired Planet tried to build up a more standard specification by submitting their specifications (C-HTML and HDML) to W3C, although this resulted in no W3C

recommendation. Later WAP Forum took HDML as a base for its WML specification. The two other specifications, TTML and *i Mode* HTML, are more closed: TTML is defined in Nokia Smart Messaging specification [1999] and *i Mode* HTML is part of the *i Mode service*, which is a similar kind of service concept for wireless devices than WAP. The *i Mode* HTML is very similar to C-HTML, but the *i Mode system* differs from all these other technologies in a way that it is currently implemented and has many services and customers but only in Japan. In addition, NTT DoCoMo is going to implement a richer application environment that is based on Java in *i Mode* mobile terminals in the future.

All these markup specifications are somehow based on or similar to the Web document model and in some level to HTML. Their implementations are incompatible but they provide a rather similar set of capabilities for application developers, TTML being the smallest in features and all the others having a virtually similar feature set. This could mean that the "WAP like" user interface mentioned in section 3.4 would be suitable for standard small screen user interface API. Since the features are nearly the same, this API should be implementable on all devices that these markup language implementations are designed for, as long as those devices are capable of running Java environment.

Since *i Mode* terminals *will* have a Java environment in the future, and if Java is also added to the WAP application environment, then Java could be used as a means for harmonising these currently competing technologies. A standard small screen Java user interface toolkit has an important role in this harmonisation. If both of these technologies will have the same user interface toolkit, then also the semantics of application downloading and execution will have to be the same to support the harmonisation. This kind of network oriented application environment clearly needs standard APIs.

4.5.3 Using markup languages for specifying user interface

User interface presentations are being created more extensively with markup languages. The introduction of XML has created even more interest. Markup languages are being used in several levels in user interface creation, for example, design phase specific markup languages for user interface specification have been created [Abrams *et al.*, 1999]. In some projects, like the open source Mozilla browser, whole user interfaces are not only specified but also implemented using some markup language. Mozilla uses HTML 4.0 and XML based user interface language (XUL) [Hyatt, 2000] for specification. This provides several advantages over previous user interface implementation methods

- user interfaces can be created directly by graphic designers,

- forces the separation of presentation and user interface logic,
- user interfaces can be more easily changed,
- provides better personalisation, and
- unifies the web user interface creation model with the native user interface creation model.

In general there are advantages that easify the user interface creation, and thus the whole product creation process, and provide means for supporting the needs of end users better. In the network appliances space the unification of two existing methods to specify user interfaces is also a huge advantage. The personalisation is even more important in hand-held devices since those devices often contain one's personal and trusted information like contacts and calendar information. If markup languages are used to specify user interfaces, then users can easily download a new look and feel to their devices. High level tools could also be created to support the end users in customising the user interfaces.

The markup language based applications are often generated and transient so that the clients often do not store the markup since it is outdated soon after it was initially created. The transient content could be, for example, business data stored in corporate databases. The problem with procedural content based user interfaces, like user interfaces specified in class files, is that the transient content cannot be passed easily to clients in the initial connection. The clients need to connect back to the server to transmit the transient data. Often clients also need to formulate queries in a form of middle-tier procedures since simple stream connection is often not possible. Like frames in HTML, this creates a round trip to the server since a whole application cannot be transferred to the client in one connection. This has even more significance in more unreliable wireless networks than in fixed networks. Several solutions could be created to solve the problem. Transient content could be generated and passed in a package. This would need a method in the client to reference the data. The problem with this solution is that different behaviour is needed in the case when the application was first downloaded to the client, and in the rest of the cases where classes already exist in the device.

4.5.4 The Spotless system

The Spotless system by Taivalsaari *et al.* [1999] is a test implementation of Java VM and a small set class libraries targeted for resource constrained devices. The KVM and J2ME discussed in earlier chapters were further developed from this research project of Sun Laboratories. When creating the Spotless system

Taivalsaari *et al.* studied the reason why Java systems are so large. They state that modern JVM implementations have lots of complexity caused by just-in time compiler technology, native threading and pre-emptive scheduling. However, by far the most important reason for the large memory footprint and consumption of Java systems is the size of the standard class libraries. They give a few simple techniques to shrink the class libraries. Below is a list of those that are directly related to toolkit design:

1. Avoid classes that create many short term objects, such as Point.
2. Reduce the number of distinct exceptions to the bare minimum, because each exception is its own class with all associated overhead.
3. Take advantage of native platform support for graphics.
4. Eliminate classes that are seldom used or that can be added by the user if necessary.
5. Remove alternate forms of functions.

Items one and two are good rules when designing and implementing toolkits and applications. Both of these items relate to the garbage collection feature of the Java VM. Garbage collection is a very demanding operation especially in environments constrained by memory and processing power. Therefore, use of garbage collection should be avoided as much as possible in these environments. From item three it can be concluded that a standard user interface API should not mandate either lightweight or heavyweight components. The implementation should be able to select the way the components are implemented.

All these items also reduce the class size by limiting constant data in the internal class structures. Taivalsaari *et al.* also noted that removing dependencies between classes reduces the overall size of the API since constant data in different classes is reduced. Item two tackles this problem in exceptions since they are classes that normally have little or no code at all but introduce a lot of constant data. By limiting the number of exception classes the size of the API can be minimised. Items four and five also directly reduce the constant data in class files.

Taivalsaari *et al.* state that small API design should be started from scratch, adding only things that are absolutely necessary. It is impractical to start from some earlier design and remove features from that since the design was done completely from another perspective.

The Spotless system contains a minimum set of classes from *java.lang* package. All of these classes do not contain the same set of methods that Java Standard Edition has and, for example, *java.lang.Runtime* also contains services

of SE class *java.lang.System* that is not in the Spotless system. In addition to the *java.lang* package, there are three completely new user interface classes designed for PDA devices, although it seems that these were designed mainly for the Palm Connected Organizer: *spotless.Spotlet*, *spotless.Graphics* and *spotless.Bitmap*.

The Spotlet class implements pen-based event handling. It contains handler methods that are invoked when associated events occur. The handler methods are penDown, penUp, penMove and keyDown. Taivalsaari *et al.* introduce a new name *the spotlet* for applications that use this class.

The Graphics class provides straightforward access to the underlying functionality of the Palm platform. Graphics has methods to draw shapes (lines, rectangles and border), display strings, and get display height and width, set clipping rectangle, move a region and draw bitmaps.

The Bitmap class is used to present simple black and white bitmaps. Taivalsaari *et al.* state that having a bitmap object simplifies the use of bitmaps, since the bitmap data need only be specified once as opposed to every time a bitmap is drawn.

5. User interface toolkit designed for small screens

Requirements and design choices for a standard user interface toolkit for hand-held devices were covered in the previous parts of the thesis. The first section of this chapter states and discusses the requirements in more detail. The next section concentrates on design problems that are raised, directly or indirectly, by these requirements. The last section expands on some important aspects of the toolkit. The discussion on these sections is not related to one single design, attempting to be more general. The discussion addresses only the key problems in designing a platform independent user interface toolkit for small screens. This means that this chapter does not present a specification of the toolkit. Nor is it an exhaustive listing of issues related to toolkit design.

5.1 Requirements

In traditional software engineering methodologies, requirement analysis is the first phase of a design process. In the analysis phase, user requirements are gathered and processed for design phase, where concrete design of a product is created based on the results of the analysis. The problem with this kind of a takeoff is that high-technology products often provide services to users they themselves do not even know to demand. The products are often totally new concepts the designers only think could interest the users. This means that often the only way to get reliable information about user needs is to build and demonstrate the technology to users in practice.

The users of a graphical toolkit are programmers. These programmers are either application developers or higher-level tool developers. The users of a graphical toolkit *specification* are also programmers. These programmers implement the toolkit on a specific platform. This means that basically the requirements are drawn both from the needs of application programmers and from the platform capabilities.

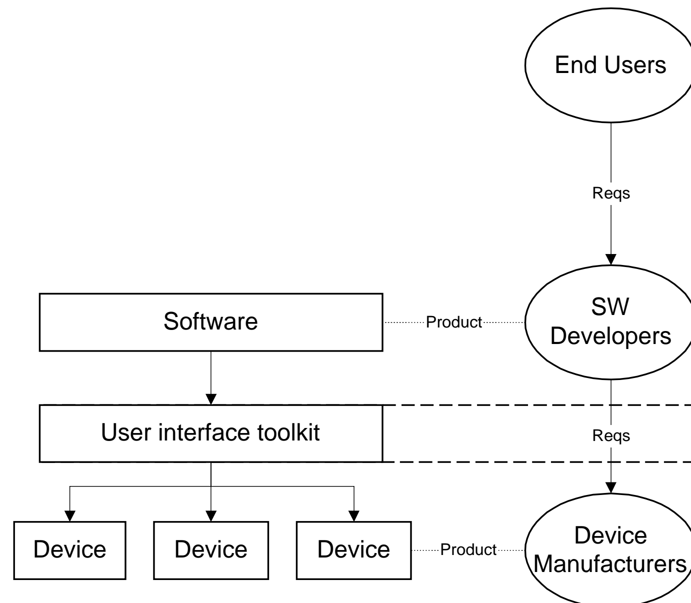

Figure 6. Interest groups of user interface toolkit.

Figure 6 presents an abstract model of the requirements flow from end users to device manufacturers. A standard user interface toolkit enables software to be run on different devices of different device manufacturers. Software developers use the capabilities of the devices through abstract interfaces. The user interface is one of these capabilities which provide an abstract interface for developers. The requirements from software developers shape the capabilities of the devices and that reflects in the abstract interface.

Generally, user interface design work will face a difficult challenge to fulfil two compatibility requirements that are potentially conflicting:

- The UI of the downloaded applications should be compatible with the resident system. This is necessary for a unified user experience. It is well known how important this is for usability.
- The applications should be portable across the devices. The application developers should be able to develop applications for devices manufactured by several manufacturers and at the same time ensure usability of their applications.

5.1.1 Interaction devices

In desktop systems, mouse and keyboard are the two established input devices that are present in almost every modern computer. The input devices in hand-held systems are not so well established. There is not a single input device that has become so common that it can be found in every hand-held system. On the other hand, among the output devices, there is one device that is present in

almost every interactive desktop and hand-held system: the display. The diversity of hand-held devices manifest in a number of display configurations. As there are de facto standard screen sizes and resolutions in desktop systems, in hand-held arena screens come in a broad range of different pixel resolution, colour resolution and physical size configurations. This creates one of the most problematic user interface design issues for us. This problem is discussed more in section 5.2.1. In addition to display output, there is sound and tactile output (mainly vibration alerts) present in hand-held systems. Both sound and tactile output are used mainly in different kinds of alert signals and often are the only output channels that can be used since the device is out of sight. Generally hand-held systems are most of the time in passive use. For example, an electronic calendar is used as a notifier, a pager waiting for incoming message, a phone for incoming call, and so forth. For a toolkit it is as important to support these other output channels as it is to support the visual channel.

The different input devices that have been used in hand-held systems come in forms of keypads and keyboards (qwerty, and other layouts), arrow keys, sliders, knobs, control wheels, touchscreens (with or without styluses) and isometric joysticks. All these input devices could be categorised to be continuous and discrete input devices. The discrete input devices can be used for character input and other command signals and the continuous input devices for motion and location input. Additionally, several higher-level input methods have been created, such as handwriting recognition and voice activation, that make use of one or more input devices in conjunction with other computing resources of a device. These resources could be both software and hardware; for example, a DSP can be used for voice recognition that can in part be used, for example, in text input.

5.1.2 Accessibility

The accessibility features of computer systems are becoming more important because of hand-held systems and code mobility. Mobile devices will need end-user applications that adapt to different environments the mobile user faces. Code mobility will need systems that will allow different applications to adapt to the system the applications are executed on.

Figure 7 presents the source of requirements for accessibility support in computer systems. The interaction between different parties is presented with two-headed arrows. Different aspects affect the interaction between the user and the terminal. Environmental aspects could restrict the use of some input or output channels in certain situations. In a noisy environment, a user might not be able to hear sounds well enough, and when driving a car normal interaction

is not suitable and access to the application could be provided through sound interfaces. Capabilities of the user are currently the major reason for developing accessible systems. A user may have permanent or temporal disabilities due to physical impairment or the influence of environment. Capabilities of the other parties the terminal is in interaction with will also have an impact on the user interaction. For example, network bandwidth is this kind of a capability. In mobile devices, network capabilities may change dramatically when the user moves from one point to another. This will raise a need for adaptive services [Davies *et al.*, 1995]. The figure presents things that can be controlled in quadrangles and things that cannot be in circles. Terminal capabilities can be built so that they will provide a good match for the capabilities of a user and environment but not vice versa. Therefore accessibility features must be built inside the systems as in user interface toolkits.

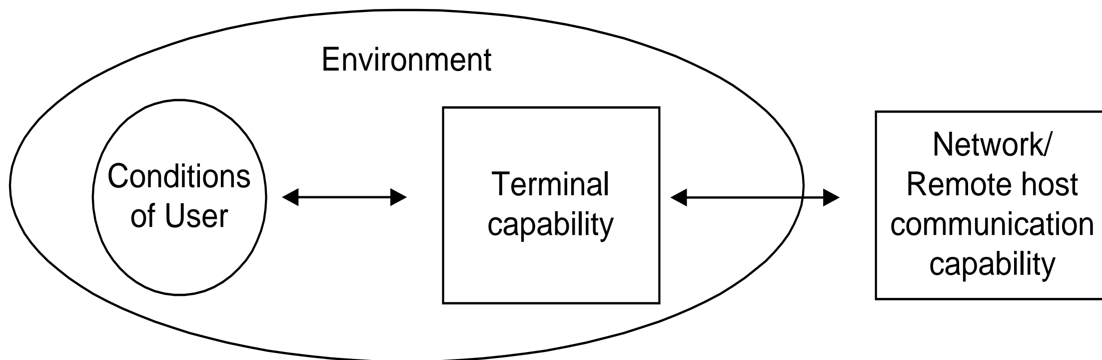

Figure 7. User interaction in the context of accessibility.
Adapted from [Kawai *et al.*, 1996].

In some countries, accessibility of computer systems is even forced by legislation. Accessibility is even more important in hand-held systems than in desktop systems because of the environmental changes the users face. In many countries it is illegal to drive a car and use a mobile phone at the same time. Some countries allow this if a special hands-free kit is used. Currently this specialised access technology can be used only for calls and even the call management cannot be handled hands-free. Better accessibility is needed in these kinds of situations that require access to the user interfaces. This will happen gradually: first specialised access technology will be built on top of existing systems, and later the adaptive technologies will be integrated in the system.

For user interface toolkits this has great impact. Currently applications are built for specific input and output channel configurations. Existing toolkits have no support for multimodal user interface development. The separation of

user interface semantics from presentation will also help us provide more accessible applications. In fact, this is a major requirement for more accessible systems.

5.1.3 The toolkit

It is proposed that two separate APIs are needed in a standard user interface toolkit for hand-held device market [Keronen, 1999]:

1. A robust low level API that gives access to all hardware features. This is needed because it is likely that there will be a range of user interface abstractions, and the abstractions should be implemented using this low-level API.
2. Device independent user interface abstractions implemented purely using the low-level API. This should initially be the best effort in this area, but it should be prepared to let these abstractions diversify and evolve. When applications are built on this level, they can be more portable and more accessible for different users, systems and environments.

Where strict binary portability is needed, access to the low level API can be restricted to internal system classes only; thus downloaded code would not be allowed to access lower level system primitives.

Figure 8 shows an abstract model for a user interface toolkit. The two implementation layers described previously map directly to this model. The lower level API contains both the primitives layer and the presentation layer. These two layers provide access to graphics primitives but also to the widgets that use these primitives. This functionality can be found from all modern user interface toolkits. The device independent API is modeled by the abstraction layer. This layer is introduced on top of the presentation layer. It provides the device independent user interface abstractions in a form of semantic controls. In the figure, any resource can be accessed from end-user applications. Restricting the access to the underlying resources will provide better portability in heterogeneous environments. The semantic control layer is not available in existing user interface toolkits. This layer is discussed in detail in section 5.2.2 where the problems of existing widget based toolkits are described.

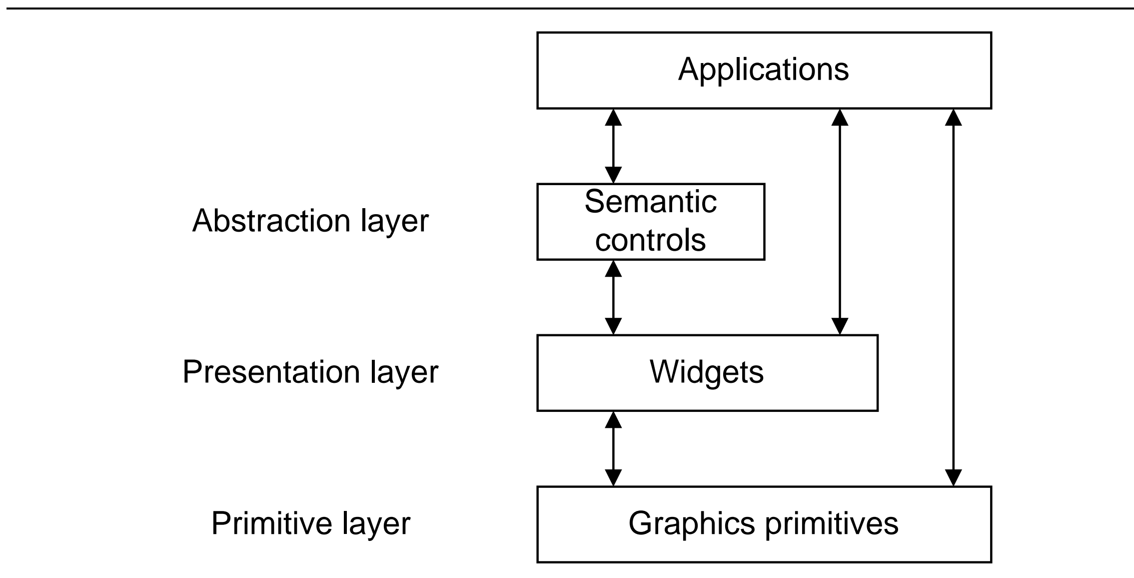

Figure 8. Layers of user interface toolkit.

5.2 Design issues

The requirements presented in section 5.1 shape the design process. One main issue when designing for hand-held systems is the total footprint of the API implementation. This impacts the whole design process and every design choice needs to be examined in this light. Since implementations and platforms vary, the footprint cannot be estimated very well at the design time. Still, quite accurate estimates are needed for a reference in design work.

There are known limitations concerning the size of Java class files. Pugh [1999] has presented a method to compress a set of class files more efficiently than the JAR packaging method of the Java platform allows. Pugh makes the important observation that the constant pool entries in class file often make up most of the size of class files. Constant pool entries are things that can be referenced within a class file, examples include classes, methods, integers, doubles and utf8 encodings of Unicode strings. The utf8 encoded strings alone often make up the most out of the size. Thus limiting the number of methods, fields and dependencies to other classes within a class makes the class file dramatically smaller in size. Moreover, dropping out complete classes from the API reduces the size of other class files. This characteristic of the class files backs the design principles that Taivalsaari *et al.* have presented and that was discussed earlier in section 4.5.4.

5.2.1 The screen size problem

The problems in designing user interface APIs for small screen devices are not just caused by a small screen but the fact that screen size ratio changes more dramatically between different kinds of hand-held devices. Moreover, in some

devices the screens are physically smaller but the actual screen resolution can be greater than in other devices that have physically larger displays. The Table 5 shows that the screen size can change by a factor of 10 or even more between devices. For example, in the Nokia 9000 Communicator screen resolution is 640x200 pixels measuring 115 by 36 millimetres (usable screen dimensions of an application will be less than or equal to 560x200 pixels). The resolution of the high end WAP phone Nokia 7110 is only 96x65 pixels and for low end phones screen resolution is even smaller. So screen resolution in the 9000 is about 20 times larger than in the 7110. For normal desktop systems the display can be only about factor of 7 times larger as low-end 640x480 is compared to very high-end 1600x1200 displays. Size ratios of hand-held systems will become even bigger as more and more high-resolution display technologies emerge, since some hand-held devices continue to have physically very small screens. For example, in watches you just can not have physically large display. And, on the other hand, desktop systems have even now sufficiently large screens that can support larger resolutions and thus the size ratio will remain rather small.

System	***Resolution ratio***
Desktop high-end normal / smallest	(1280*1024) / (640*480) ≈ 4
Desktop largest / smallest	(1600*1200) / (640*480) ≈ 6
9000 Communicator (appl. area) / 7110	(560*200) / (96*65) ≈ 18
9000 Communicator (total) / 7110	(640*200) / (96*65) ≈ 21

Table 5. Display resolution comparison of existing desktop screens and hand-held device screens.

Java AWT and Swing make use of layout managers (as discussed previously in section 3.2) to tackle the changing screen resolution problem. Traditional layout managers do not scale in small screens very well. Some abstract layout managers could be a solution to the problem. For example, DialogLayout with constraints like 'command_object' and 'content_object' could be added to a toolkit. On every platform DialogLayout is mapped to some platform dependent layout style that can be implemented on pure Java code or on native development language.

5.2.2 The widget problem

Traditional toolkits use the notion of widgets. Widgets are the basic building blocks of application user interfaces. Most widgets in the user interfaces are used to display values of application variables and allow a user to set new values on them. This kind of usage of widgets set the *application state*. Traditional widgets do not set the application state directly, but provide hooks

for a programmer to be notified when the internal state of a widget has been changed. Some widgets are not connected to any application internal data value but trigger some *application operation*. These two usage cases of widgets are fundamentally different in semantics. There are some user interface design policies and guidelines for different platforms that instruct one to use specific widgets for specific purposes; for example, a push button widget in dialogue screens should be used only for application operations. Application programmers can choose to either follow these guidelines or not. Often the guidelines are so exhaustive [Smith and Mosier, 1986] that there are too many things for average people to recollect.

In Figure 9 a conceptual model of application layers is presented. It depends on the application, the operating environment and the user interface toolkit which parts of the model are defined by the application and which are defined by the system. The two topmost layers, "look & feel" and "visualisation" form the visual part of the application user interface. Look and feel contains presentation constructs, such as what kind of colour, animations, fonts and graphical styles are used. In modular systems like Swing this part can be changed dynamically. The visualisation layer contains parts that tell what are the visualisation rules for the logical user interface constructs. These rules will either completely or only in part realise the user interface look and feel. The user interface logic layer contains constructs that define applications interaction semantics. In the logical layer the semantics are not tied to any specific input or output channels. These three layers form the user interface part of an application. The two undermost layers "application semantics" and "application data model" form the internal structure of an application. Application semantics contain constructs that describe the core and internal behaviour of the application. In good architectures the core is also separated from the data model. In the next paragraphs the discussion gets into the problems that arise from the usage of widgets. In short, the problem is that the widgets do not allow the building of a user interface logic without binding it tightly to the visualisation and look and feel layers.

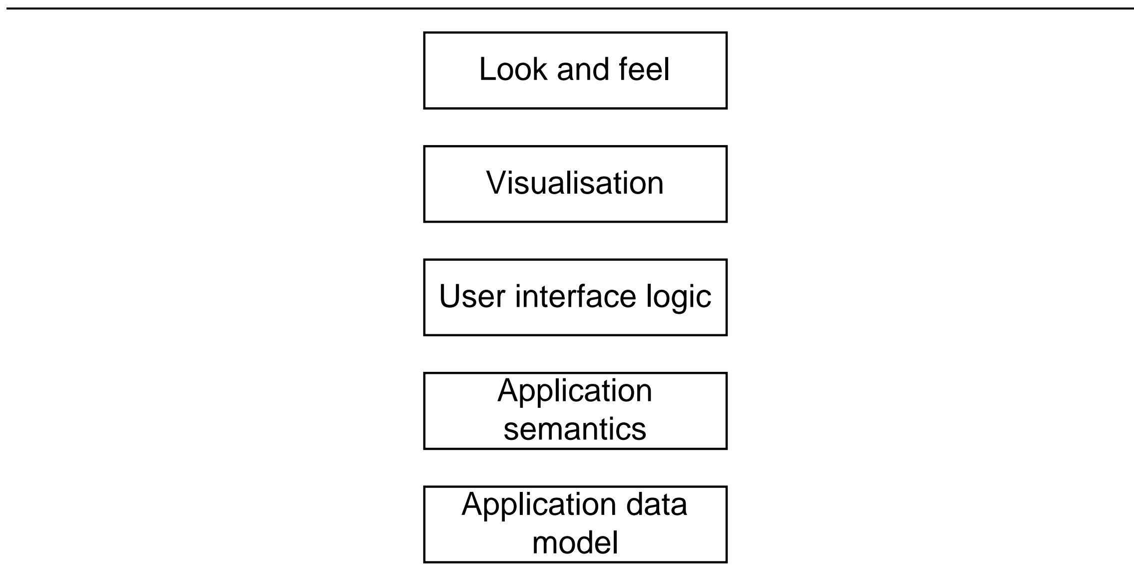

Figure 9. Conceptual model of an application architecture. The model is separated into layers that each handle a distinct transformation of input and output channels in the use of a complete system.

There are several problems in using widgets as basic building blocks for user interface programming. One problem is that the widgets require a programmer to fiddle with appearance and layout issues. A great amount of time is used, for example, in aligning components like labels and input widgets properly. Since there are often many widgets that provide similar semantics with different visualisation, a programmer has to select a suitable presentation for a specific task. This selection is often done without good insight and thus results in bad user interfaces. The selection of a right widget depends on the semantics of both the user interaction and the application.

Because widgets are presentation oriented, they are poorly connected to application semantics. This is the main cause for many of the problems related to widgets. In traditional widget based toolkits, the widgets know nothing about the application data types. A radio button or a check box has its own internal state that is presented on screen and programmer has to make mapping code to the application model. For example, a selection of a colour from an enumerated set {red, green, blue} is mapped in a user interface as three radio buttons that have text labels “Red”, “Green”, “Blue”. When using traditional widget toolkits the radio-buttons are poorly connected to the application model {red, green, blue}. This means that programmers have to convert values returned from the user interface widgets to the values of an application model. In the given radio button example an index that represents the radio button that was selected needs to be converted to a value of an

enumerated set. These kinds of type conversions often create type errors that are in the worst case detected only in runtime.

The mapping code between widgets and an application model normally makes the user interface and the rest of the application tightly coupled. This means that changing the user interface is not possible without a lot of additional coding work. To ease this problem, there are well known design patterns that help to separate the user interface and the internal application model; for example, the Model-View-Controller (MVC) architecture that was originally used to build interfaces in SmallTalk-80 [Krasner and Pope, 1988]. In the MVC architecture, a model and a view are separated by a controller that communicates with the model and the view through well-known interfaces. With this architecture, the view can be changed without any changes done to the model. There can also be multiple views presenting the same application model. MVC architecture is built into the Swing toolkit, so an application programmer can more easily build applications that separate user interface and application logic.

Research has been done about going beyond the traditional widgets. Blattner *et al.* [1992] have tried to develop *metawidgets* that are capable of manifesting themselves to users through different modalities, as a sound or as an image or combinations of these two media. This multimodal use of metawidgets is also incorporated in the semantic control layer of the user interface toolkit that was presented in the Figure 8. In the domain of hand-held devices, some kind of metawidgets could also solve the problems with accessibility and portability. When applications are built with different traditional widget based toolkits, separate user interfaces need to be created for every environment and use situation. If applications could use semantic controls that abstract from the actual presentation and modality, and if the system would handle the specific implementation of the widgets that present these controls, the mobile users would be able to use same applications in different devices and also in different use situations and environments.

Johnson [1992] introduced the concept of selectors. Selectors, unlike widgets, are bound to semantics, not to any single visual appearance. Johnson classified the use of different interactive controls in applications according to their semantics. He found the two classifications of widgets: widgets that *set application state* and widgets that *trigger some application operation*. From this classification it follows that two different kinds of semantic controls are needed, namely *Data selectors* and *Command selectors*. Both present users their options and allow them to select the ones they want.

The primary idea of Johnson is to 1) pay more attention to application semantics and less to the appearance and layout of applications and 2) provide the system with much more information about the application and interface semantics than they traditionally have. Johnson's idea was that the application developer specifies only semantics and then the toolkit in a *development system* can present available presentations suitable for these semantics. However, this is not possible in the problem domain of this thesis, since the mapping of semantics is not done in design time but in runtime. This appears, for example, when an end user downloads some application package from a network to a terminal. The application has to adapt in runtime to the user interface of the terminal. This creates additional constraints to the user interface toolkit, and even more dynamic solution than the one Johnson has presented is needed.

The approach of concentrating on user interface semantics in development time will also be ideal for us. This is the key point why the user interface constructs of the abstraction layer are called semantic controls. Widget based UI toolkits would bind the applications to platforms with similar capabilities even if application programmers would use a standard UI toolkit. Semantic controls will help in application portability and the support of system level accessibility features.

5.2.3 The abstract factory problem

To support multiple platforms, the API has to be modular enough to allow replaceable implementation. One thing that maintains the compatibility of Java is compatibility tests. These tests are conducted with software called Java Compatibility Kit (JCK). Different versions of JCK are provided for different Java platform configurations and versions. Also, JCP states that JCK shall be created for outputs of the standardisation process. This means that *implementation* of a public API must be same for all the platforms that comply with the API specifications. The implementation specific code is separated from a public API with some well known method, such as Abstract Factory and/or Factory Method design patterns [Gamma *et al.*, 1995]. The problem with all these design patterns is that they add a new layer of abstraction to the system. Normally this means a lot of new classes and thus additional consumption of storage space and a need of bigger runtime memory.

Abstract Factory also has creation methods for all the widgets that are accessible from this API. The number of methods is doubled by the concrete factory implementation. Adding a new widget to the platform also requires the changing of the abstract factory interface and all the classes that depend on it. These issues can be relaxed through the use of a single generic *make* or *create*

method for all the widgets. This method would get some parameter that specifies which widget should be created. Possible parameters can be integer numbers, strings or class objects. For statically typed languages, like Java, the use of generic create method limits the return values to subclasses of specified return type. In Java the method can easily return *java.lang.Object* instances, but this easily results in runtime errors since static type checking cannot be used. In this case all the widgets will most probably have a single parent class that can be used as a return type for the method. Another inconvenience is that in an application code there is a need to cast the returned widget object to the dynamic object that was requested. This results in too many redundant cast operations. So it would seem that this kind of an abstract factory solution would be feasible only if the language was dynamically typed.

Every commercial Java platform or API implementation must pass JCK conducted testing before it can be shipped as a product if it uses the Sun codebase. The JCP paper states that similar software will be built by specification lead for outputs of all JCP projects. It is not specifically stated *what* the technology compatibility kit (TCK) should test. It is at least necessary for profile APIs of J2ME that platform specific implementation code can also reside in public interface classes. This means that the compatibility check should only test the public interface and its external functionality (outputs with specific inputs), not how the interface is implemented.

In J2SE, there are at least two layers in user interface widget classes: shared public Java classes and implementation specific peer classes. This layering is done with abstract factory design patterns. The Toolkit class represents an abstract factory. A default concrete factory is determined by the Java system properties.

Design patterns are used to create modular frameworks that will meet future needs for change. A downside of design patterns is that this extra layer easily doubles the number of classes and thus adds more references to other classes. This creates too much overhead to the API because the domain contains embedded systems.

It seems that currently it is feasible to have an API that could be implemented with as small number of classes as possible. The public API should not state anything about the internal software architecture; for example, it should not state which design patterns should be used. One implementation that has very constrained memory resources could choose to use no Abstract Factory classes and their concrete counterparts. Implementations that do not have such constraints can use more scaleable software architectures. For public API, this would mean that the widget classes could be modified for every

implementation. Evolution to a much finer class framework that can handle future extensions can be implemented by not using constructors in the public API of widget classes but factory methods that in constrained implementation would call the normal constructors that are declared protected. This design pattern can be seen in Figure 10.

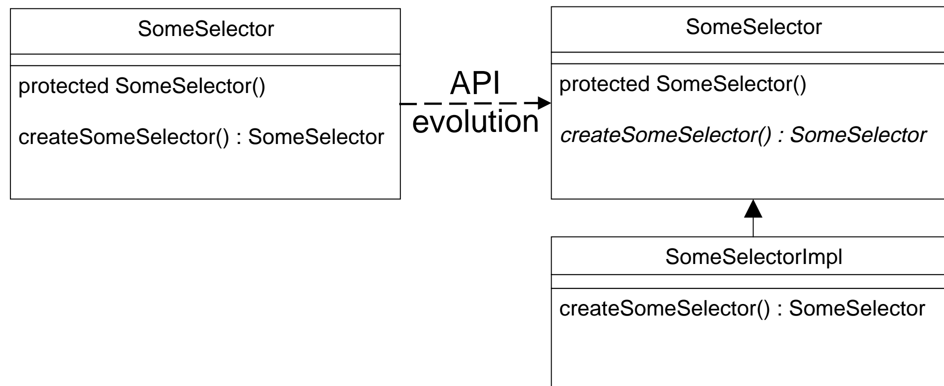

Figure 10. Usage of Factory method design pattern to leverage API implementations on more constrained environments.

Identical binary codebases for all the implementations can be achieved in the future. As the API evolves and some memory constraints are relaxed, subclasses can be added to the base classes. These subclasses will serve as implementation classes and make the original classes abstract. This would allow freezing the abstract classes as identical in every platform.

The Java language specification does not recommend changing a non-abstract class to abstract since normally it would result to instantiation errors in linktime or runtime. In this case these errors would not occur, since the constructors are declared private and the instantiation is done with a factory method. As stated previously, this design choice would work if compatibility tests would only test the functionality of the public interfaces.

5.3 Details

In the following section, several important details about the implementation of the semantic controls concept will be discussed.

5.3.1 Associations between semantic controls

Application developers define the application semantics. This means that semantic information is *within* the applications. Traditionally the underlying system has not been aware of this semantic information. Because the semantic controls concept draws away control over of the application look and feel and the layouting of components from the application developers, some means are

needed for communicating the needs and semantics of applications to the systems.

Traditionally application developers have defined associations between semantic controls implicitly. These implicit definitions show up in the user interface design choices that developers make; for example, in the layout of components, a password text box could have a label near it titled "password". Thus the string "password" and the text box are two totally separate user interface constructs from the system point of view. Mainly, all widespread user interface toolkits have no programmed knowledge about the binding between them. These kind of semantics are totally hidden from the system but are clearly expressed by developers and experienced by users. A semantic control approach limits the ability of application developers to implicitly express application semantics so an alternative and explicit way to express them is needed. Associations between different user interface constructs could be one convention to leverage semantics in a way the semantics are also communicated to the system level.

The implementation of associations could be as simple as an association object that is attached to two user interface constructs and some associations could be handled distinctly by the API in a form of methods and classes tailored for the specific case. If the direction of association is needed, then association object could also have a reference to some user interface construct. The implementation details could be handled in the way associations are mapped from design diagrams of different modelling languages like those of OMT or UML. An association object contains all the information about a specific association. The vocabulary and semantics of feasible associations are of course dependent on the usage of the user interface toolkit. A generic set of associations could contain information like that associated constructs *relate* to each other in *some way*. The problem with very generic associations, similar to ones in constraint systems, is that the association solving engine will become more complex and the rules for mapping different associations to presentation could easily conflict each other. In addition, portability could be at stake since different association solving engine implementations could interpret them differently. Because these problems are rather pragmatic, information should not be left behind in toolkit designs.

Pragmatic information about application semantics that could, for example, be used in associations specifies information about

- navigation and application flow,
- traversing of components inside some context (screen or component),
- context of on-line help,

- context of application defined commands, and
- relation of limited set of controls that should be handled as a group.

These could be handled in a generic way with association objects or by some mechanism built to the toolkit for each case. For example, if some application dialogue has specific order of use (traversing of components) the order could be specified by association objects linked between user interface constructs or the toolkit could be tailored for this semantic information so that construct objects contain some defined field or method that stores and reports the order to the system. If some form of more generic associations that could be parametrized for a specific case are used, the toolkit should allow many-to-many relations in associations between the user interface constructs.

The navigation and application flow semantics are most probably specified by some other means than as association objects. Context of on-line help could mitigate to a single help string associated with a user interface construct. The internationalisation requirements could demand some more elegant solution; for example, a help command associated with construct with hooks to the help system of implementation. Platform independence could still require a method that requires a help to be specified as some structured text format supported by all target platforms, like WML in wireless device space. The help semantic information in commands is needed so that systems could handle the help in a context sensitive and system specific way; for example, a command with help semantics information attached to a text entry control will give system a hint that the command will be used to display on-line help about the text entry. If the system has more detailed knowledge about the user context within the user interface, then the system could present the command only at an appropriate place or time, thus removing clutter from user interface. Similar context information could also be attached to other commands.

Context information could contain information on whether a command is global, specific to a single control or specific to a set of controls. The set of commands could be any grouping provided by the toolkit. Context can also contain information whether a command is system or application defined. This information separates the actions invoked by commands to the system domain and to the application domain. System defined commands are either implicitly or explicitly defined in the toolkit specification. Explicitly defined commands mean that the specification states directly which commands need to be supported in specific user interface controls. For example, a text handling control could allow a reset or a clear command or both. The toolkit could also allow application defined configuration of system commands for user interface controls. For example, an application could define that a specific system

command is needed in user interface control. This kind of configurability is important for more accessible systems, for example, when a user needs to use the application through a speech user interface system that needs detailed information on which commands are absolutely necessary and which are redundant. If it is defined explicitly then the implementations will be more tied. Implicit definition allows more flexible implementations but application user interfaces will behave differently between implementations. This would suggest that complete list of actions supported by a user interface control should be exposed.

For example, WML does not provide this kind of detailed semantic information on the system. This results in that WAP browser implementations can not make use of the system efficiently or provide more intelligent user interfaces that adapt to the user's needs. For example, a WAP application can not have context sensitive command menus; a WAP user agent always needs to present to the user all the commands that exist within a card even if most of them are redundant in a specific situation.

This semantic information will allow the system more easily provide a better user interface for the application than existing toolkits provide. The application or system should not depend in any way on human presentable command names or descriptions because it will affect the localisation of the applications written for the toolkit. Neither should commands be bound or have any information on hardware level input events like keyboard scan codes that caused the command since these would make applications dependent on different keyboard layouts or key sets.

5.3.2 Input abstractions

In semantic control layer direct input events from input devices should not be accessible to applications. However, some application types, for example games and other highly interactive graphical applications, need this lower level information from interaction devices. This section discusses one possible solution to providing events from interaction devices to applications and still maintaining application portability.

The main problem here is the diversity of input devices. To support application portability between devices that have different input capabilities input abstractions are proposed. This abstraction allows better emulation of input actions not supported natively in a specific platform. For example, better support for applications that need positional input could be provided in platforms that do not have touchscreen hardware.

Input devices have been tried to abstract in many ways in existing studies, for example, Buxton [1983] presents a taxonomy of devices to help create a model for input device abstractions. Buxton points that effective input device abstractions cannot be done without taking pragmatic issues into account. These issues contain, for example, things like input gestures, available space and attributes of input devices. The point here is that the effectiveness of a particular user interface is often due to the use of a particular device, and that effectiveness is lost if some other device with slightly different attributes is used, even if the device belongs to the same logical class of the abstraction model in question. Buxton argues that pragmatic issues play major role in interaction but that does not solve the problem in question. Most of the mobile devices either have keypads (discrete input) or touchscreens (continuous input). Buxton's taxonomy did not even take discrete devices into consideration. The presented solution is not very effective in this sense. The point is that it will allow application developers to provide more accessible systems and portable applications at the cost of complete control and perhaps usability.

Figure 11 presents the idea of providing input abstractions. Only the abstract input events will be transmitted to applications. In the figure the topmost layer is the only layer accessible from applications. However, there are some tradeoffs here, since applications that use detailed low-level input cannot use this abstraction. These tradeoffs are discussed in detail later on in this section. Since applications make use of abstract input events, the system can easily emulate these events in software. For example, a portable device that does not have discrete input devices such as keypads that create directional input can emulate abstract events with positional input. This could be implemented, for example, with on-screen keyboards.

Key events have been abstracted in some level already in desktop systems. A problem with AWT keyboard events is that they do not try to abstract the set of different key presses. KeyEvent class has virtual key flags for every key that is resident in standard QWERTY keyboards. More abstract solution would be to provide only a set of high-level directional events like UP, DOWN, LEFT, RIGHT. Mapping of these events to specific hardware is implementation dependent. Some platforms could even provide more dynamic mapping by providing the user a customisation tool for these mappings. A timestamp when event was created could be provided, and additional information like whether the event key was a press or a release event. This additional information would make more constraints on toolkit portability.

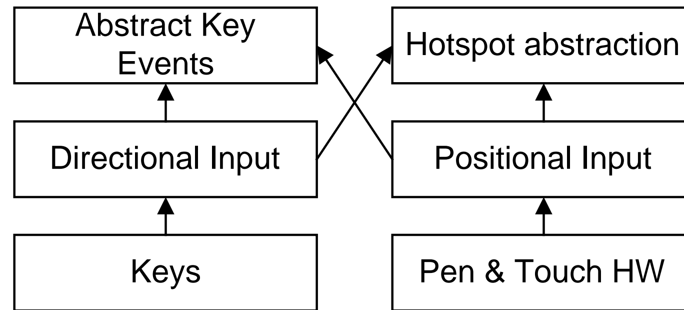

Figure 11. Input abstraction for allowing better input emulation.

The term hotspot abstraction is used in a technique for providing easier emulation of positional input information on systems that do not provide direct method for two-dimensional continuous input like touchscreens or mouse-like input devices. Application programmers define hotspots on drawing canvas that they are interested. Hotspots are geometric areas, in simple implementations they can be just rectangles but more complex areas can also be used if the graphics system of the toolkit provides support. An abstract event is created if positional input is generated in this area. Systems that do not have capability to provide positional input can use the information supplied by the application to create, for example, a list of areas or focus traversal between areas with keypad that can be used to create positional input inside the area.

Because a hotspot event does not give information about the actual co-ordinates the input was generated in, totally random positional events cannot be handled with this method. This includes, for example, applications that allow free drawing and map applications that need very detailed information about the co-ordinates. This limits the use of a toolkit that does not provide lower level access.

5.3.3 Capability mechanisms

If suitable abstractions cannot be found for specific features, then API designers often use mechanisms that allow applications to query capabilities of concrete implementations. Often when designing API standards and adequate consensus is not achieved some parts are just left as optional or totally out of scope of the standard. In these standards word "optional" often means that if support for this feature is implemented then the implementation must comply with the standard. When standard specifies optional features there is also a need for capability mechanisms.

Capability mechanisms are often needed at least in some level in user interface toolkits. For example, methods that are used to query the screen size or colour depth of specific device are part of the capability API of the toolkit.

Without this information it is hard to use a graphics system of the device to draw anything. On the other hand, some applications could use the capability mechanisms to check whether some feature is implemented and completely rely on the implementation on the feature. This means that slightly different device capabilities are not supported at all. In general, the definition of capability features in open API reduces portability among all applications. This means that applications that make use of an API that states a lot of optional features tend to be non-portable. When designing capability mechanisms, it should be remembered that applications hardly need specific low-level implementation information.

APIs should not be bound to any specific software or hardware technology but they should reflect things in conceptual level. For example, in the user interface toolkit context, most applications do not need to know details about the specific user interaction device the portable device holds. The input devices should be abstracted and the toolkit should handle these interaction devices in a more abstract, conceptual level. At the conceptual level interaction devices could, for example, be presented as different input and output channels. If a toolkit or more generally any API uses more abstract concepts, then more applications will be portable to different devices. This means that general portability level of applications depending on the toolkit will be much better than today.

5.3.4 Compound interactive controls

Because the problems of layouting semantic controls cannot be solved for a while, workarounds are needed in the short run. A *de facto* standard toolkit will most probably have interactive compound controls replacing the semantic controls. These kinds of higher-level user interface constructs have been used already in existing widget based toolkits, for example, in the form of standard dialogues like the file dialogue or the print dialogue. The toolkit provides some means to configure the compound control but concrete selection of which widgets and what kind layouting is used are implementation dependent issues. From the toolkit implementation point of view, compound controls are containers for different widgets. Every toolkit implementation must implement the compound controls in a way specified in the standard. The standard thus specifies the level of freedom that every implementation can have.

Interactive compound controls will solve the layouting problem by leaving the layouting completely to implementation. If the portability of applications and toolkits is totally based on the use of compound controls, then the power of expression is directly proportional to the number of compound

controls in the toolkit specification. This would mean that highly powerful toolkits that make only use of compound controls are also complex and large in footprint.

6. Conclusions

In this thesis I have studied the problem domain of user interface creation toolkits for portable network connected devices. The portable nature of these devices places great restrictions to the user interface capabilities. One of the main characteristics of the devices is that they have small screens compared to existing desktop environment displays. In this thesis Java and WAP were used as case studies or examples about how to use the ideas that were presented here in practise. Java provides good building blocks for the ideas presented in the thesis and WAP is a technology that already attempts to address some of the problems presented.

The appearance of different kinds of network accessible mobile devices or network appliances creates requirements for user interfaces that are not met by existing widget based user interface creation toolkits. End users want to use their applications whenever and wherever possible. The same applications need to be accessible in different situations, environments, and from different platforms and devices. This means that code mobility and accessibility are the two great challenges of user interface problem domain in the future. In this thesis I have presented many problems that exist in these problem domains.

The application portability requirements drawn from code mobility and diverse set of network appliances change the user interface design model. Application developers cannot have detailed control over user interface look and feel anymore. The main conclusion of this thesis is that user interface creation mechanisms that communicate the semantics down to the system level are clearly needed. There are a lot of alternatives how this could technically be handled and some ideas were given, such as the addition of a semantic control layer to the toolkit, use of abstract commands and associations between the user interface constructs to help solving the problems in layouting and presentation.

All these techniques need more detailed study before they can be used in user interface standards or products implementing the standards. However, compound controls were presented as some kind of a interim solution for problems that relate to layouting and presentation when using semantic control layer in toolkits.

The abstractions in the semantic control layer will provide better accessibility and portability, but it will also restrict the type of applications. The semantic control layer cannot address all application needs; for example, applications that use graphics or audio directly cannot be presented completely in an abstract way because of the pragmatics of interaction. This means that

traditional low-level access to graphics primitives must be allowed in a toolkit that needs to support these features. Applications that use the low-level API will be less portable and accessible. Input method abstractions for low-level API were discussed as a method to provide some application portability between devices that support either touchscreen only or keyboards only. The conclusion was that abstractions are needed even if they will not provide full-fledged semantics compared to direct use of input hardware or even hinder application usability when pragmatics are considered. Application programmers need to consider which is more important, portability and accessibility or direct control over the application usability, look and feel and layout.

For globally accessible applications, we need explicit toolkit level support for application semantics. I predict that in the following years there will be a slow evolution in user interface toolkits towards this goal.

References

[Abrams *et al.*, 1999] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuster, UIML: an appliance-independent XML user interface language. In: *Conference proceedings of the Eight International World Wide Web Conference*, 1999.

[Arnold *et al.*, 1999] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo and Ann Wollrath , *The Jini Specification*. Addison-Wesley, 1999.

[Blattner *et al.*, 1992] Meera M. Blattner, Ephraim P. Glinert, Joaquim A. Jorge and Gary R. Ormsby, Metawidgets: towards a theory of multimodal interface design. In: *Proc. of Sixteenth Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, 1992, 115-120.

[Buxton, 1983] William Buxton, Lexical and pragmatic considerations of input structures. *Computer Graphics* **17**, 1 (January 1983), 31-37.

[C-HTML, 1998] Compact HTML. February 1998. Available as *http://www.w3.org/TR/1998/NOTE-compactHTML-19980209/*.

[Davies *et al.*, 1995] Nigel Davies, Gordon S. Blair, Keith Cheverst and Adrian Friday, Supporting adaptive services in a heterogenous mobile environment. In: *Workshop on Mobile Computing Systems and Applications*, IEEE Computer Society Press, 1995, 153-157.

[Davis, 1999] Mark Davis, Abstraction. *JavaReport* **4**, 6 (June 1999), 80-89.

[Dix *et al.*, 1993] Alan Dix, Janet Finlay, Gregory Abowd, Russell Beale, *Human Computer Interaction*. Prentice Hall, 1993.

[Dix *et al.*, 1999] Alan Dix A, Devina Ramduny and Tom Rodden, Places to stay on the move, software architectures for mobile user interfaces, extended abstract. In: *Second Workshop on Human Computer Interaction with Mobile Devices*. Available also as *http://www.hiraeth.com/alan/topics/mobile*.

[ECMA-262, 1998] Standard ECMA-262, ECMAScript Language Specification, 2nd edition August, 1998. Available as *http://www.ecma.ch/stand/ECMA-262.htm*.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gosling *et al.*, 1996] James Gosling, Bill Joy and Guy Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[GSM MMI, 1998] GSM 02.30 Version 6.0.1 Release 1997: Digital Cellular Telecommunications System (Phase 2+); Man-Machine Interface (MMI) of the Mobile Station (MS). *European Telecommunication Standards Institute*, 1998.

[HDML, 1997] Proposal for a Hand-held Device Markup Language (HDML). May 1997. Available as *http://www.w3.org/TR/NOTE-Submission-HDML.html.*

[Hoque, 1997] Reaz Hoque, Java, JavaScript and plug-in interaction using client-side LiveConnect. Netscape Communications Corporation. Available as *http://developer.netscape.com/docs/technote/javascript/liveconnect/liveconnect_rh.html.*

[HTML 4.01, 1999] HTML 4.01 Specification, W3C Recommendation, 24 December 1999. Available as *http://www.w3.org/TR/html401.*

[Hyatt, 2000] Dave Hyatt, XPToolkit Architecture. Janurary 2000. Available as *http://www.mozilla.org/xpfe/xptoolkit/.*

[IFC Developer Central] IFC Developer Central. Netscape Communications Corporation. Available as *http://developer.netscape.com/docs/manuals/ifc/home.html.*

[ISO 8879, 1986] ISO 8879, Standard Generalized Markup Language (SGML).

[ITU-T E.161, 1995] ITU-T Recommendation E.161 (05/95), Arrangement of digits, letters and symbols on telephones and other devices that can be used for gaining access to a telephone network. *International Telecommunications Union*, May 1995.

[Jacob, 2000] Robert Jacob, User interfaces. In: Anthony Ralston, David Hemmendinger and Edwin Reilly (eds.), *Encyclopedia of Computer Science*, Fourth Edition. Macmillan, 2000. (in press)

[Java Glossary] Glossary of Java related terms, Sun Microsystems Inc. Available as *http://java.sun.com/docs/glossary.html.*

[JavaPhone, 1999] JavaPhone specification. Sun Microsystems, Inc. Available as *http://java.sun.com/products/javaphone/.*

[Johnson, 1992] Jeff Johnson, Selectors: going beyond user-interface widgets. In: *Conf. Proc. on Human Factors in Computing Systems*, 273-279.

[Jones and Marsden, 1997] Matt Jones, Gary Marsden, From the large screen to the small screen - retaining the designer's design for effective user interaction. In: *IEE Colloquium on Issues for Networked Interpersonal Communicators*, 3/1-4.

[JSR-0000037, 1999] Java Specification Request 37, JSR-0000037, Mobile Information Device Profile for J2ME. Available as *http://java.sun.com/aboutJava/communityprocess/jsr/jsr_037_mid.html.*

[Kamba *et al.*, 1996] Tomonari Kamba, Shawn A. Elson, Terry Harpold, Tim Stamper and Piyawadee Sukaviriya, Using small screen space more efficiently. In: *Proc. Conf. Human Factors in Computing Systems*, 383-390.

[Kawai *et al.*, 1996] Shiro Kawai, Hitoshi Aida, Tadao Saito, Designing interface toolkit with dynamic selectable modality. In: *Proc. of the Second International ACM/SIGCAPH Conference on Assistive Technologies*, 72-79.

[Keronen, 1999] Phone user interface API – requirements, email discussion with Seppo Keronen, 6. October 1999.

[KJava, 1999] KJava API Specification. In: *JavaOne, Sun's 1999 Worldwide Java Developer Conference*. Preliminary Release, June 15-18, 1999.

[Krasner and Pope, 1988] Glenn E. Krasner and Stephen T. Pope, A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, **1**, 3 (Aug./ Sept. 1988), 26-49.

[KVM, 1999] The K Virtual Machine (KVM), a white paper. Sun Microsystems, Inc., June 1999. Available as *http://java.sun.com/products/kvm/wp/*.

[Lindholm and Yellin, 1996] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Manifest Format, 1996] Manifest Format. Sun Microsystems, Inc., 1996. Available as *http://java.sun.com/products/jdk/1.2/docs/guide/jar/manifest.html*.

[McKay, 1997] Ross McKay, Platform independent FAQ. March 1997. Available as *http://www.zeta.org.au/%7Erosko/pigui.htm*.

[Miah and Alty, 1998] Tunu Miah and James L. Alty, Visual recognition of windows: effects of size variation and presentation styles. In: *Proc. of Australasian Computer Human Interaction Conference*, 72-79.

[Myers, 1995] Brad Myers, User interface software tools. In: *ACM Transactions on Computer-Human Interaction* **2**, 1 (March 1995), 64-103.

[Nokia WAP Toolkit, 1999] Developer's guide, Nokia WAP Toolkit 1.1. Nokia Corporation. Available as *http://www.forum.nokia.com*.

[Puerta, 1997] Angel Puerta, A model-based interface development environment. *IEEE Software* **14** (July/August 1997), 40-47.

[Pugh, 1999] William Pugh, Compressing Java class files. In: *Conf. on Programming Language Design and Implementation*, 247-258.

[Rekimoto, 1996] Jun Rekimoto, Tilting operations for small screen interfaces. In: *Proc. of the ACM Symposium on User Interface Software and Technology*, 167-168.

[Rochkind, 1992] Marc Rochkind, An extensible virtual toolkit (XVT) for portable GUI applications. In: *Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers*, 485-489.

[Shneiderman, 1988] Ben Shneiderman, Improving the accuracy of touch screens: an expreimental evaluation of three strategies. In: *Proc. Conf. Human Factors in Computing Systems*, 27-32.

[Shneiderman, 1991] Ben Shneiderman, Touch screens now offer compelling uses. *IEEE Software* **8**, 2 (March 1991), 93-94, 107.

[Smart Messaging, 1999] Smart messaging specification, revision 2.0.0. Nokia Mobile Phones Ltd., May 1999. Available as *http://www.forum.nokia.com.*

[Smith and Mosier, 1986] Sidney L. Smith and Jane N. Mosier, Guidelines for designing user interface software, Technical Report ESD-TR-86-278. Mitre, August 1986. Available as *ftp://archive.cis.ohio-state.edu/pub/hci/Guidelines.*

[Symbian] Symbian Ltd, *http://www.symbian.com/epoc/hardware/communicators/communicators.html.*

[Taivalsaari *et al.*, 1999] Antero Taivalsaari, Bill Bush and Doug Simon, The Spotless system: implementing a Java system for the Palm Connected Organizer. Sun Microsystems, Inc., February 1999.

[Tasker, 1999] Martin Tasker, EPOC overview: core, EPOC technical paper. Symbian Ltd., June 1999. Available as *http://www.symbian.com/epoc/papers/e5ocore/e5ocore.html.*

[W3C] World Wide Web Consortium, *http://www.w3.org/.*

[WAP Architecture, 1998] Wireless Application Protocol, Architecture Specification. April 1998. Available as *http://www.wapforum.org/.*

[WAP Forum Website] WAP Forum, *http://www.wapforum.org.*

[Wild, 1999] Rick Wild, An introduction to Waba. *Hand-held Systems* **7**, 1 (Jan./Feb. 1999), 13-15.

[WML 1.1, 1999] Wireless Application Protocol, Wireless Markup Language Specification. Version 1.1, June 1999. Available as *http://www.wapforum.org/.*

[WML Reference, 1999] WML reference, version 1.1. Nokia Corporation, September 1999. Available as *http://www.forum.nokia.com.*