Design guidelines for mobile applications

Erik G. Nilsson

SINTEF ICT

June 2008

Abstract

This document contains a set of design guidelines that are meant as a means for facilitating development of more user friendly applications on mobile devices (PDAs/SmartPhones).

The design guidelines give practical advices for how to solve various problems that arise when designing user interfaces on mobile devices. Firstly, some important choices are presented, like equipment types, platforms, etc. Then, a number of problems are presented in more detail. These problems are grouped in three main problem areas:

- Utilizing screen space
- Interaction mechanisms
- Design at large

Each problem is presented on a “design pattern” format where background for the problem, the problem itself, and possible solutions for the problem is discussed. In the presentation of possible solutions, pros and cons of different solutions are discussed, and examples of good solutions are given where appropriate. Finally, some basic differences between applications on mobile devices and applications on PCs are presented for developers that are “new” to mobile devices, as well as some thoughts on and links to platform styleguides and some thoughts on exploiting the fact that the user is mobile when designing mobile applications.

TABLE OF CONTENTS

1 Introduction

This report is one of the main results from the research project “UMBRA – Development of user friendly mobile applications”. The UMBRA project has been conducted in 2004 and 2005, and has been funded by the Norwegian Research Council.

The participants in UMBRA have been:

- PSI Systems International – Prime partner, a company that develops mobile solutions for commodity trade
- Antares – a consulting company developing mobile applications
- Giant Leap Technologies – a company that develops applications and middleware solutions for mobile applications
- IT Liberator – a company that develops applications, database management systems and development tools for mobile applications
- MediCom – a company that develops mobile solutions for the health sector
- Mobile Mind – a consulting and advisory company for mobile solutions
- SINTEF ICT – Research partner

This document represents the third version of these guidelines. In addition to the report, the guidelines are also available as a simple web application.

This document contains a set of design guidelines that are meant as a means for facilitating development of more user friendly applications on mobile devices (PDAs/SmartPhones).

The design guidelines give practical advices for how to solve various problems that arise when designing user interfaces on mobile devices. Firstly, some important choices are presented, like equipment types, platforms, etc. Then, a number of problems are presented in more detail. These problems are grouped in three main problem areas:

1) Utilizing screen space
2) Interaction mechanisms
3) Design at large

Each problem is presented on a “design pattern” format where background for the problem, the problem itself, and possible solutions for the problem is discussed. In the presentation of possible solutions, pros and cons of different solutions are discussed, and examples of good solutions are given where appropriate. Finally, some basic differences between applications on mobile devices and applications on PCs are presented for developers that are “new” to mobile devices, as well as some thoughts on and links to platform styleguides and some thoughts on exploiting the fact that the user is mobile when designing mobile applications.

The “sources” for the problems addressed in these guidelines are problems identified in the requirements elicitation phase of the project. The problem stem from the pilot projects conducted by the project partners, from general experience gathered by the project participants when developing mobile applications, from a literature survey, and from experiences with using different applications on mobile devices.

After the problem structure was identified as part of the first version of these guidelines, the industrial partners prioritized the most important problems. This showed that the following problems where considered most important:

- Problem 1.1.o.1 – Principles for grouping information
- Problem 1.1.o.2 – Mechanisms for grouping information
- Problem 2.1.p.1 – Mechanisms for entering text
- Problem 2.1.p.4 – Multi modal interaction – stylus, scanner, RF-ID, different types of keyboards, voice control
- Problem 3.1.p.1 – Design that both supports branding, is aesthetic, and utilize screen space optimally

These partners have also given special contributions in identifying possible solutions to these problems.

2 Important choices

Before handling the specific problems identified, we will present some important choices that influence the user interface when developing mobile applications. The presentation will also to some extent define the scope of this work.

2.1 Type of device

There is a distinction between users moving between devices and user moving with devices. Although both may be considered being mobile, only the latter is considered in this report. A user bring with him a device may use any computer that is possible to carry (we also exclude use of stationary computers in cars etc.). A rough categorization of computers or devices with computing abilities is the following:

- Portable PC (incl. ultra portable PCs)
- Tablet PC
- Personal Digital Assistant (PDA)
- SmartPhone
- Other mobile phone

In this report, we focus mainly on PDAs and SmartPhones, but some of the problems handled may also be relevant when designing UIs for other mobile phones and tablet PCs, in some cases even for Portable PCs.

In the following we give some characteristics of these device types.

By a portable PC (incl. ultra portable PCs) we mean a PC running a standard version of the operating system (usually Windows XP). These devices have a full size keyboard and support mouse as a pointing device. Some potable PCs have touch screens that may be operated by a stylus, but this is a supplement to the mouse, keeping the characteristics of a mouse operated environment (e.g. having a mouse pointer, supporting double click and supporting tool tips in a “normal” way).

By a tablet PC we mean a PC running a version of the operating system tailored for tablet PCs (usually Windows XP Tablet Edition). These devices may or may not have a keyboard (the keyboard may often be “hidden” when the user is walking around), but are tailored for usage without a keyboard. Modus operandi for these devices is operating through a “pure” screen based dialog where text is entered through some on-screen text entry mechanism(s), and controlled by a stylus. The Tablet Edition of Windows uses the “mouse” model of interaction with a visual pointer. Compared to a PDA, these devices share most of the characteristics regarding interaction mechanisms, although the tablet PCs focus more on handwriting. Thus, most of the material handling this in this document also applies to tablet PCs. As the screen size is larger on a tablet PC, the material on screen size are less relevant.

By PDA we mean a small size computer whose main function is to support mobile tasks. The main common functions of these devices are Personal Information Management (PIM) applications (calendar, tasks, notes, email, etc.). It has become increasingly common that these devices also include camera and phone. Most devices have touch screen (operated by a stylus), and no or limited keyboard. If a keyboard is available, it is either a telephone type keyboard or a keyboard with very small keys. The screen resolution is usually in area of ¼ VGA, but recently, devices with VGA (640x480) resolution have become available. The most common pointing model is that there is no visual mouse pointer on the screen, and that there is no distinction between click and double click (i.e. click (called tap) means double click). Also, as there is no mouse pointer on the screen, there is no mouse move event, which is a severe hindrance for utilizing tool tips in a natural way. In addition to tap, the gesture “tap and hold” is supported, which in most cases plays the same role as right click on a mouse.

By SmartPhone we mean a high end phones with PDA functionality. These devices share most characteristics with PDAs, and as the devices evolve, the distinction becomes increasingly superficial. Usually, SmartPhones are considered to be less powerful, often smaller with less memory and weaker processor (and thus often longer battery life). As there are two versions of the PocketPC/Windows Mobile operating system, one for SmartPhones and one for PDAs (with the main difference being screen size requirements and requirements for using stylus), we have kept the distinction. As a rule of thumb, a SmartPhone is a phone with PDA functionality, while a PDA with phone is, well, a PDA with phoning capabilities. Although we have made a distinction here, the material in presented below applies both to PDAs and SmartPhones.

By Other Mobile Phones we mean low end phones with no or limited PDA functionality. Again, the distinction is not very clear as almost all phones have some PIM support, but these devices have smaller screens, seldom touch screens, have telephone keyboard, and are less powerful. As they are seldom a target platform for professional applications, this type of devices is not focused in the material below. Despite this, many of the problems described also apply for these devices, in many cases even to a larger degree. The solutions, though, may not.

2.2 Platform

There are four main platforms for PDAs / SmartPhones:

- Linux (http://www.linuxdevices.com/)
- Palm (http://www.palm.com/)
- Windows Mobile/PocketPC/Windows CE (http://www.microsoft.com/windowsmobile/default.mspx)
- Symbian (http://www.symbian.com/)
- Apple OS X – an optimized, full version of the Mac OS X operating system (without unnecessary components) will run on the iPhone that will be released during 2007 (http://www.apple.com/)

The material in this report is to some extent based on experiences from the PockePC platform, but most of the content is general, and should be independent of the chosen platform. In the cases where functionality that is platform specific or varies between platforms, this is noted in the text. Most of the contents is based on experiences from PocketPC 2003 SE. The new version of the PocketPC operating system is now called Windows Mobile 5.0. There are some changes in the UI in this new version, but few of these influence the details in these guidelines. Still, we prefer to denote this platform PocketPC in the guidelines in case there are small discrepancies. For functionality that is special for the new version, the term Windows Mobile 5.0 is used in the text.

We do not find it appropriate to recommendations regarding choice of platform in this report. In addition to the characteristics of the different platforms given above, which may be used as basis for prioritizing in a development project, we provide some main trends regarding how the different platforms are used.

So far, Linux-based devices have limited deployment, but may be increasing its marked share. The Palm platform is mainly used for fairly simple PDAs that are to be used mainly as personal organizers. The new models with built-in telephone move the platform mainly towards the Symbian platform. The PocketPC platform has had its main success as platform for professional applications. This is partly due to a fairly accessibly development environment providing an almost seamless transformation from PC development to PDA development, partly due to powerful devices that are able to support applications requiring much resources to run. The second fact has also made these devices fairly large and expensive, which have made them less attractive as pure SmartPhones (for the devices containing telephones) or personal organizers. The Symbian platform has mainly had its success in the SmartPhones segment, i.e. equipment that mainly are phones, but with personal organizer and other PDA functionality. A general trend is that the assortment of devices within each platform is increasing, and that the variation within each platform is increasing, thus decreasing the difference between the platforms.

2.3 User interface style

There are two main choices of user interface style when developing PDA/SmartPhone applications, with a third option which seldom is relevant for most practical cases:

- Web-based User Interface (WUI)
- Graphical User Interface (GUI)
- Terminal server based solution

A WUI is characterized by the fact that is can be run in a web browser. In this context, we define this to http-based solutions (i.e. we exclude wap). The pros of a WUI-based solution are that the deployment of the solutions is easy, and that the resource requirements for the devices are modest. The cons of a WUI-based solution are that the application usually requires that the device is always online, and that the repertoire of available UI functionality is limited. In fact on most PDA / SmartPhone browsers, the possibilities are so limited that one may say that the difference in available UI functionality between WUI and GUI is larger on PDA / SmartPhones than on desktop PCs (even though the GUIs also are more limited on the small devices).

A GUI is usually a standalone application. The pros of a GUI-based solution are that it offers the developer more functionality and control of the application, thus it is easier to realize specialized solutions, and that it is more flexible with regards to how the application communicates with servers (on-line, synchronization, combination). The cons of a GUI-based solution are that it usually requires more powerful devices and that deployment is more demanding, specially upgrading an existing application (first time installation is often provided when the equipment is delivered).

Partly as a curiosum, it should be mentioned that it is also possible to use a terminal-server based solution on PDAs. It is available on most platforms. The most known provider of technology for such solutions is probably Citrix. Using a terminal-server based solution, the whole application runs on a server, with only the user interface shown on the client. This means that once the generic client is installed, nothing is required on the client to run an application. It also means that the device must be online all the time. How practical using such a solution is depends on the degree to which the application (on the terminal server) is tailored to the screen size of the device that runs the client. Running a standard PC application through a terminal-server solution on a PDA is possible, but most users would only use it in situations where no other options exist and running the application is of vital importance. Also, controlling an application that is intended to be controlled by a mouse using a stylus also causes special challenges for the user.

Looking at the same choices for desktop solutions, a WUI is least vulnerable to requirements to screen size on a computer to run an application. As the difference in screen size between a desktop and stationary computer is as large as it is, the same is not the case for mobile application. I.e. it is usually necessary to tailor a UI to a small screen in all the three solutions.

2.4 Connection to server

Although primarily a technical question, how a mobile client program is connected to a central server also influence the user interface of the system. A presumption for this section is that almost all professional applications need to have some sort of communication with a back-end system storing the data on which the application operates.

The main choices for connection to server are using a stand-alone or a server-based solution. Some solutions may follow a stand-alone model only, but few solutions follow a pure server-based solution. Most solutions probably use some sort of hybrid solution. Despite this, we describe the “pure” models to see the characteristics of extremes.

A stand-alone solution in this context is a solution that works in all mobile situations without requiring an online connection to the server. As we assume that there is a back-end system, this type of solution requires a synchronization solution. This means that data is transferred to the mobile client when the user connects, and the user manipulates this information in a non-connected state, transferring the information the user has entered the next time the client is connected. The main pros of this solution are that the client does not need to be online all the time, and thus may be operated in environments without available connections, that the communication costs are reduced, and that the operational model is fairly easy to understand for the user, i.e. the user does not need to know anything about the connection state. The cons of this solution are that it requires more storage and usually also more processing capacities on the client, as the client needs to be self-supplied, that there needs to be developed a synchronization solution, and that the information is seldom up to date, neither on the client nor the server.

A server-based solution in this context is a solution where the client is always connected to the server through some sort of wireless network connection. The pros of this solution are that the client may be simpler, both regarding processing powers and client software (ideal for WUI-and terminal server-based solutions), and that the information is always updated both on the client and the server. In addition, a pure server-based solution does not need any synchronization solution, but as pure server-based solutions seldom work very well in practice, synchronization solutions are seldom avoidable. The cons of this solution are that there always needs to be available wireless network connection for the solution to work, increased communication costs, and that the user must have a basic understanding of the connection state (including how to know that and what to do when the connection is lost) and in cases where different communication means are used (e.g. both WLAN and GPRS) also which type of connection that is used.

2.5 Type of application

Regarding the fairly general title of this section, we focus on one aspect of the application to be developed, i.e. whether it is transaction/data collection oriented or it is retrieval oriented. This is seldom or never a choice when developing a system, it is given by the requirements to the application. It should also be mentioned that although most systems combine transaction and retrieval functionality, there is usually a focus on one of them. The reason to make a distinction is that there are different challenges when designing the two.

For applications (or rather the parts of the applications) that focus on transactions, the main challenge is handling input. Thus the main problem area 1 is most relevant for these applications. Also some of the issues in problem area 3 are highly relevant.

For applications focusing on retrieval, the main challenge is presenting information in a reasonable way so that the user is able to identify the desired information. Thus the main problem area 2 is most relevant for these applications.

3 Main problem areas

The design guidelines presented in this document follow a given structure. On the top level, they are grouped into three main problem areas:

1. Utilizing screen space
2. Interaction mechanisms
3. Design at large

Within each of these three main problem areas, a small number of problem areas are defined. Within each of these problem areas, a number of problems are identified. These problems are grouped according to whether they are identified from the pilot applications or from other sources. So the structure is this:

- Main problem area (1-3)
  - Problem area (1-n)
    - Problems identified from the pilots
      - Problem
      - Problem
      - …
    - Problems identified from other sources
      - Problem
      - Problem
      - …

The actual problems on the leaf level are identified according to their placement in the structure:

<main problem area>.<problem area>.<p or o>.<problem number>

The third item in the identifier is given a “p” if the problem stems from the pilots or an “o” if the problem stems from other sources.

In this part the problem structure, with the problems and their identifiers are presented. A brief description of the main problem areas and problem areas are given in this part, but no further description of the problems is given here. In the next part (*Problems*), all the problems on the leaf level are presented in a flat structure. To link back to the structure in this part, both the problem identifier, and references to the main problem area, problem area and source are given in the *Problems* part.

3.1 Utilizing screen space

This main problem area focus on problems connected to the limitations regarding screen space on mobile equipment.

3.1.1 Screen space in general

This problem area addresses problems connected to different layout challenges on small screens.

3.1.1.1 Problems from the pilots

- Problem 1.1.p.1 – Presenting elements in lists

3.1.1.2 Other problems

- Problem 1.1.o.1 – Principles for grouping information
- Problem 1.1.o.2 – Mechanisms for grouping information
- Problem 1.1.o.3 – Mechanisms for packing information
- Problem 1.1.o.4 – Horizontal scrolling

3.1.2 Flexible user interfaces

This problem area addresses problems connected changing the layout dynamically at runtime because the either the information that should be presented and/or the environment in which to present the information change.

3.1.2.1 Problems from the pilots

- Problem 1.2.p.1 – Presentation (which types of data) based on models or data – how to do this on a small screen
- Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden

3.1.2.2 Other problems

- Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode
- Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size

3.2 Interaction mechanisms

This main problem area focus on problems connected to the limitations regarding interaction mechanism on mobile equipment.

3.2.1 Handling input

This problem area addresses problems connected to entering information more efficient and/or with less probability for entering incorrect information.

3.2.1.1 Problems from the pilots

- Problem 2.1.p.1 – Mechanisms for entering text
- Problem 2.1.p.2 – Order entry – needs to be fast even if it contains large amounts of data and there are a number of rules
- Problem 2.1.p.3 – Numerical keyboard (more general: mechanisms for entering numerical data
- Problem 2.1.p.4 – Multi modal interaction – stylus, scanner, RF-ID, different types of keyboards, voice control
- Problem 2.1.p.5 – Controlling the input cursor from an application

3.2.2 Not using the stylus

This problem area addresses problems connected to entering information in situation when it is not possible / convenient / desired for the user to use a stylus.

3.2.2.1 Problems from the pilots

- Problem 2.2.p.1 – Interacting with applications without using stylus
- Problem 2.2.p.2 – Retrieving data from a database without using keyboard

3.3 Design at large

This main problem area focus on problems connected to design principles for user interfaces on mobile equipment.

3.3.1 Guidelines

This problem area addresses design guidelines on different levels of generality.

3.3.1.1 Problems from the pilots

- Problem 3.1.p.1 – Design that both supports branding, is aesthetic, and utilize screen space optimally
- Problem 3.1.p.2 – Solutions for searching large amounts of data, e.g. multi-step solutions combining reliable identification and optimal screen space utilization (also horizontally)
- Problem 3.1.p.3 – Visually coding of entry fields to mark editability (must, may, may not)
- Problem 3.1.p.4 – Standard solutions vs. usable tailored solutions – how to choose

3.3.2 "Difficult to understand"

This problem area addresses problems connected to providing understanding of what is happening when a mobile application performs functionality that can be difficult to understand for the end user – usually functionality that is specific for mobile applications.

3.3.2.1 Problems from the pilots

- Problem 3.2.p.1 – User interaction for/during synchronization solutions
- Problem 3.2.p.2 – User interaction for log-on/log-off
- Problem 3.2.p.3 – User interaction during waiting for long-lasting operations to complete

4 Problems

In the previous chapter we presented the problem structure, with main problem areas, problem areas and individual problems grouped by their source. In this chapter, we present the problems in a “flat” structure, i.e. one section for each problem.

Each problem is presented on a “design pattern” format where background for the problem, the problem itself, and possible solutions for the problem is discussed. In the presentation of possible solutions, pros and cons of different solutions are discussed, and examples of good solutions are given where appropriate.

The individual problems are more or less closely connected (also across the problem area structure). Such connections are mentioned numerous places in the description below, e.g. to avoid describing the same solution a number of places. These connections are placed as cross references in the document, and should work as links in an electronic version of the document.

4.1 Problem 1.1.p.1 – Presenting elements in lists

Main problem area: Utilizing screen space
Problem area: Screen space in general
Main source for problem: Pilot

4.1.1 Background

In this context, we define a *list* to be a tabular presentation of similar information in a vertically oriented list. By *similar* we mean that the items presented in a list are instances of the same model or share some base class(es) in overlapping models (possibly with some derived attributes from related classes). Lists usually have more than one column, and may also use more than one line to present one instance.

Lists are a very common user interface component in most user interfaces. When comparing user interfaces on small screens with traditional GUIs on larger screens, lists are used even more frequently on the former. There may be a number of reasons for this, the two most important are probably

- Some of the UI components that may play the same role as a list (like a tree view) are not available or are considered more difficult to use on a small screen
- Lists are more space efficient than some of the alternative UI components

There are very few applications on mobile equipment that do not utilize lists, often as a very prominent presentation form, but sometimes only for more specialized functions.

Lists are usually used for presenting data, more seldom for offering functions (i.e. application specific functions – the lists themselves possess some generic functionality (select, open – which in itself may have an application specific interpretation) and may be augmented by specific functions on popup menus etc.), the lists present information and usually offer various mechanisms for navigating or manipulating the information presented.

4.1.2 Problems

Even though lists are a compact presentation form, the main challenge when using lists on small screens is handling the space restrictions.

Given the most common case – i.e. that the lines in the list represent instances stemming from the same model – handling space restrictions is at a certain level of abstraction a question of handling projections and selection of the extension of the model (the total number of instances). Projection is about which attributes that are to be shown (and in this context the sequence of the attributes is of prime importance). The selection is about which instances that are to be shown (sequence is also important for this). The main goal is often to avoid scrolling the data to find the desired information. If the projection is wrong (compared to the user's need), the user will not find the desired information, or horizontal scrolling is needed to locate it. If the selection is wrong, the user will not find the desired instance(s), or vertical scrolling is needed to identify it/them. As a rule of thumb, horizontal scrolling is worse than vertical, but both should be avoided – i.a. because operating a scroll bar using a stylus is difficult. The main reason why horizontal scrolling is "worse" than vertical is that the information on the same line usually has a closer connection than information on different lines.

Another problem related to presenting elements in lists is which functionality to offer in the lists, and which standard(s) to relate to. Lists on desktop computers make a clear distinction between *select* and *open*. Usually, selection is related to single click, whilst open is related to double click

(or selection followed by a menu or button choice). Using a stylus on a mobile platform, it is not common to make a distinction between single and double click. Thus, if both select and open is needed, it must be decided whether clicking with the stylus follows the desktop model and means select (which leads to the need for some other – usually less obvious – way to open) or follows the common model on PDA which is to open an item when the user clicks on it with the stylus (which makes selection impossible or very cumbersome). The latter causes a special problem when combined with vertical scroll bars: when dragging a scroll bar downwards along a list the stylus easily "slips" out of the scroll bar area and into the list causing an item to be opened. Returning to the list sometimes even resets the position in the list to the top so that the scrolling process must be restarted.

This problem is handled in some applications on the Symbian platform using a combined solution where clicking the text in a list opens it and clicking an icon or check box in the list causes selection.

A different functionality problem is how to handle hierarchies (usually of the same type of data, e.g. files in different folders). On desktop platforms, this is often handled using separate UI mechanisms on a different part of the screen (usually a tree view control). Using a separate (and usually fairly large) part of the dialog to handle hierarchies is seldom an optimal solution on a small screen. Using different techniques (see the description of **Error! Reference source not found.**) usually is less intuitive, and often also more cumbersome.

An observation concerning select vs. open is that using the PDA standard (click is open) is that it mainly makes selection difficult when using the stylus. When using HW buttons (i.e. usually a four-way navigation pad – or even a keyboard) selection (and open) is usually easy.

Probably the best way to avoid horizontal scrolling is to **restrict the number of attributes presented in the list** to the available space. Although quite obvious and easy to agree with, this solution can be very difficult or impossible to apply in practice, mainly because the user may need more information than there is room for in order to perform the task the list is supporting. Also, a prerequisite for this solution to work is that all users need the same information from the list.

A related solution when it is not possible to restrict the number of attributes is to **optimize the sequence and size of the attributes shown** in the list so that the user very seldom need to access the data that is to the right of the visible area. As for restricting attributes, this solution requires that all user need the same information from the list.

Another way of avoiding horizontal scrolling (i.e. making it easier to match a preferred projection with screen size) is to use **more than one line per item in the list**. Visually, it works very well – it is usually not a problem to see which information that is connected to which instance. On the other hand, having more than one line per instance makes it difficult to use column headings. Showing e.g. a number of different dates or time values connected to each instance is then difficult unless information about which attribute that is shown is added. This problem may also be partly handled by showing details about selected instances together with the list (see below). Another problem with multi-line list is that there is room for fewer instances on the screen at the same time, thus causing the need for more vertical scrolling. As the screen space is restricted, this trade-off is quite obvious. Prioritizing getting rid of horizontal scrolling will generally be a preferred solution when there are few instances, or when the instances are visited in the same sequence as they are presented, so that the user does not have to use the vertical scrolling for looking up instances.

A forth way of avoiding horizontal scrolling is to make the **size and sequence of attributes shown flexible**. This can be done by letting the user use the table headings to adjust the width of each attribute and drag and drop to read just the column sequence. This might make it possible for the user to fit the attributes that are needed for each instance within the width of the screen. The application must remember the setting for this type of functionality to be useful, and it must be possible to reset to standard size and sequence. Generally, this type of functionality must be considered to be used mainly by experience users.

A fifth way of decreasing the need for horizontal scrolling is to supply **additional information** about an instance when it is selected (this solution requires that click means select). The natural way of doing this is to use a part of the dialog for presenting the instance in a forms based view. Additional information may also be provided in a pop-up field when the user performs a tap-and-hold operation on a line in the list. This will block for using a pop-up menu in the list, but requires less screen space. The main problem with reserving a part of the dialog for a forms-based presentation of the instance is indeed that it occupies screen space, and increases the need for vertical scrolling (at least if the screen orientation is portrait). It is also a restriction to the applicability of this solutions, as the size of the presentation view increases with the number of attributes the instance have. But this is also one of the main factors that cause the need for horizontal scrolling. As for the list itself, it should be mentioned that it is not necessary to show all attributes in such a “preview” part of the screen.

Part of the problem connected to scrolling in lists (both vertically and horizontally) is operating scroll bars with a stylus. One way of solving this is **avoiding scroll bars but keeping the scrolling functionality**. To render this possible, some other mechanism than traditional scroll bars must be used. In some applications, larger scroll bars are used. Another solution is to use large arrows, e.g. in the bottom of the list to scroll up and down. A problem with this solution is that it does not support rapid scrolling using the elevator in the scroll bar. It also lacks the additional information the elevator gives about which (and how large) part of the extension that is shown in the current view. Jet another solution is to use part of the list as scrolling control[1]. In this solution, parts of the list (i.e. the right half) act as scroll control. Clicking on the upper right quarter scrolls the list upwards, while clicking on the lower right quarter scrolls the list downwards. The pro of this solution is that it is easy to operate, even without using a stylus. The cons are the same as for replacing the scroll bar with arrows.

In the problem part above, we discussed that the standard functionality when clicking on items in lists are different between PDA and PC platforms (open on PDAs and select on PCs). On PDA, this causes a problem only if there is a need for having both select and open as functions in the list. If there is no need for selecting elements, the list items may as well be opened when selected (with some inherent problems, though). If there is indeed a need for selection, e.g. to show details about the selected instance in another part of the screen, using click as select is preferable. This causes a need for another operation for opening the instance (if needed). We will not recommend double click, both because it is difficult to perform a double click correctly on a PDA, and because it is seldom used in other applications. Rather, tap-and-hold, push buttons and menu items should be used. It should be marked that the situations where select is needed, usually have less needs for opening the item than the situations where click to open is most natural. The most relevant situation for selection is a list + detail view where details about the selected item are shown elsewhere in the window. It is natural to put a bush button for showing more details in this part.

[1] This solution is adapted from Giant Leap Technology

4.2 Problem 1.1.o.1 – Principles for grouping information

Main problem area: Utilizing screen space
Problem area: Screen space in general
Main source for problem: Other

4.2.1 Background

Grouping information is a central means to handle the general screen space problem. In these guidelines, we have divided the grouping problem in two: principles and mechanisms. In this problem, we handle the principles, while we handle the mechanisms in Problem 1.1.o.2 – Mechanisms for grouping information. The main distinction is that the principles focus on how to group information independently of the actual mechanisms for doing the grouping. Thus, the problem description on principles is more abstract than the one on mechanisms.

Grouping information is mainly about deciding which information that should be presented together. In this context we focus on grouping information as a means for splitting information into different parts that are to be shown in different parts of the UI. This means that we do not handle grouping of information within a single view, e.g. using borders. Doing this in a mobile UI is quite similar to desktop solutions.

4.2.2 Problem

Probably, grouping information according in some way to the underlying information model is the most natural principle to follow when grouping information on a PDA, but it is not the only one possible. Below, we will investigate some alternatives.

Given that the information is grouped according to the information model the application is built on, there are still a number of choices. The major grouping principle in all data oriented applications is to group according to different classes/entities/concepts in the information model, e.g. by having one window for each major concept in the application, and some windows for handling important connections (typically parent/child-windows). This type of grouping is of course also natural to use on applications utilizing small screens. When using small screens it is in addition also often a need to group information connected to one concept. As for lists (see Problem 1.1.p.1 – Presenting elements in lists), this kind of grouping is also connected to projections and selections. I.e. projections are applied if different sets of attributes are presented together in different groups, while different selections are applied if different sets of instances are presented together in different groups. Generally, projections are probably used more often than selection as grouping principles. See Solution(s) for pros and cons connected to the grouping principles.

In addition to grouping information according to information model, grouping information according to the information needed to perform different tasks is also fairly common as a general user interface design principle. Such user interfaces are often referred to as task-oriented or task-driven. Of course, there is often a connection between the information needed to perform a task and how information is grouped in the information model – but not necessarily 100%. Generally, it is highly relevant to organize the information according to tasks also on a PDA (see Solution(s)).

Although data- and task-oriented user interface designs are most common, also other principles may be applied. One reason to choose a different grouping may be to make the solution similar to

a PC application that the user is accustomed to using. Even if the application is task oriented on a PC, the same need not be the case on a PDA.

4.2.3 Solution(s)

The main advantage of having a data oriented user interface is that the user interface becomes quite generic (depending on the available operations), and thus may support different (but usually related) tasks, and may fit more users than a task oriented one. The main disadvantage is that by being generic such a solution tend to be suboptimal for everyone.

The main advantage of having a task oriented user interface is that it fits the tasks of the users well. On most mobile equipment, only one dialog may be shown at the time. This also supports being task oriented, as it will reduce the need for switching between dialogs while performing a specific task. The main disadvantage is that to user interface will be more difficult to use if the tasks differ from the ones the UI is designed for. This may often be the case if the application is used by a different user group. Task oriented principles are incorporated in the PocketPC version of the MS Office applications. To avoid the need for switching between applications, the PocketPC Office applications have built-in functionality for file handling. These “mini file explorers” only let the user manage the documents that are connected to the application, and also offer special functionality for these document types (functionality that would not be as natural to have in a generic file manager). One of the advantages of this solution (in addition to avoid switching between applications) is that there are fewer files to deal with, which makes it easier to get an overview. In addition to grouping information according to tasks, it is often a good idea to make this grouping explicit, i.e. make the main work processes (tasks on a coarse level) – typically up to six – visible in the user interface as main choices.

In cases where the set of tasks differ much between users or roles, it is a solution to have a framework for the application that may be adapted to the different users/roles. This can e.g. be achieved through having a configuration file controlling which UIs that should be presented in which way to the different users/roles. This could also include which attributes to include, but this will usually make it more difficult to implement (see also Problem 1.2.p.1 – Presentation (which types of data) based on models or data – how to do this on a small screen).

It should be mentioned that a task oriented UI may well also be quite data oriented (and thus also the other way around), given that the tasks fit well to the information model on which the application is built. It is also important to mention that data or task oriented UIs is not an either/or choice. It is quite possible to have UIs that utilize both principles.

Further to this, the “ideal” solution is often to base the grouping on frequency of use, i.e. to give the information that is most often used a prominent place in the UI, and to hide information that is more seldom used. One may argue that this leads to a task oriented UI, but this is not necessarily always the case.

The solutions presented above focus on grouping based on projections. It should be mentioned that grouping based on selection, i.e. filtering information that is not relevant, is an important mechanism for making it easier to identify the most relevant information. In many cases, such filtering may be done automatically based on the identity of the user. If the user needs to see information that is not “his”, then there should be special mechanisms for obtaining this information.

4.3 Problem 1.1.o.2 – Mechanisms for grouping information

Main problem area: Utilizing screen space
Problem area: Screen space in general
Main source for problem: Other

4.3.1 Background

Grouping information is a central means to handle the general screen space problem. In these guidelines, we have divided the grouping problem in two: principles and mechanisms. In this problem, we handle the actual mechanisms for achieving the grouping, to some degree independently of the principles described in Problem 1.1.o.1 – Principles for grouping information.

4.3.2 Problem

When choosing mechanism(s) to use for grouping information, it is important to have a clear picture of the need for grouping (cf. Problem 1.1.o.1 – Principles for grouping information), and if the grouping is data oriented, not to mix up grouping between main concepts and within one concept. It is also necessary to know if the grouping represents a projection (most common) or a selection (e.g. some filtering functionality).

Before choosing a relevant mechanism, it is important to have an opinion on the number of levels that is needed for the grouping. First of all, is the grouping really needed, or could a mechanism like scrolling be used instead. This could be the case when there is only slightly too much information to fit on the screen. If grouping is indeed needed or desired, is one level sufficient, or should the grouping use more than one level? Independently of the number of levels, it is also necessary to consider the number of groups on each level.

When doing these considerations, it is of prime importance to consider the ways the UI is to be used. One of the most important aspects is to consider the frequency of use for different part of the information that is to be present in the application.

4.3.3 Solution(s)

One of the most common mechanisms for grouping information is using **tab folders**. This is a mechanism that makes it possible to enlarge a dialog, and thus it is very well suited for small screens – and it is therefore also much used in mobile UIs. But it should be noted that like most grouping mechanisms, the tab folders themselves take up screen space, and on a small screen there is room only for a limited number of tabs. This number is depending on the labels on the tabs, and in case of “overflow”, scrolling mechanisms are usually used. This should be avoided. The alternative solution used on desktop computers is to use more than one row of tabs, but this is not supported on PDAs, at least not on the PocketPC platform.

A variant of tab folders is to have **blocks of information that may be collapsed and expanded**. This must usually be combined with scrolling, as the amount of information may be large if the user chooses to show all information simultaneously. The benefit of this solution compared to tab folders is that the user has more control over which information that should be shown together. The cons are that such UIs easily become untidy and may be perceived as unstable by the user.

An alternative mechanism for grouping information is using **different dialogs with intuitive navigation**. By this we mean that having different dialogs in an application is also a grouping mechanism, and by having an intuitive way of navigating between the dialogs, this may work

perfectly. By intuitive navigation we mean both the whole dialog structure, i.e. which navigation paths that exist between the dialogs, and how a new dialog is started for the current, how and where the user returns, i.e. what the user must do to utilize the available navigation paths. Compared to tab folders, this solution offers more room for presenting information in each dialog, but maintains less context information when the user navigates between the groups of information.

A special variant of using different dialogs and focusing on navigation is using **wizard based UIs**. This type of UIs applies a very process oriented approach, with just one (or a few) predefined paths between the dialogs. Thus they may be considered being extremely task oriented, which indeed is the case given that the process implemented in the wizard actually matches the task of the user. Generally, it is considered that wizard based UIs are best fitted for inexperienced users, as they usually eliminate many error sources. They are often viewed as too elaborate and rigid by experienced users. Thus, some applications offer wizards as an alternative to other dialogs. It should be mentioned though that for some tasks, a wizard style UI may be the best solution for all users. The prime characteristic of these tasks is that they are very step-by-step-oriented, i.e. information *must* be entered in a special sequence, either to avoid errors or to avoid a lot input control.

A mechanism that it not too much utilized in mobile applications is the **tree view control**. There are probably many good reasons for this, e.g. that it is difficult to operate using a stylus and virtually unusable for finger navigation. It should though be mentioned that tree view controls are indeed available on PDAs, and may be used in special cases, probably mainly for experienced users in an indoor environment requiring flexible solutions. The pro of a tree view control is that it offers quick access to information both of different types and information of the same type on different levels in a hierarchy.

It should also be mentioned that the different mechanisms discussed above may of course be used in combination in a given application.

Regarding what the different mechanisms are best suited for, this has already been covered above for wizard based UIs and partly for tree view control. It is difficult to give absolute rules for when to use tab folders and when to use separate windows, but as a rule of thumb separate windows should be used to split between different main concepts/classes while tab folders should be used to separate between different aspects of a single concept/class.

As mentioned above, how frequent some information is used should play a major role when choosing which information that should be presented together. It is also important when choosing the sequence of e.g. tab folders. In a window presenting information about a class with a large number of attributes, the most commonly used ones should be presented in the part of the window that is always visible. The second most commonly used information should be presented in the tab folder that is shown by default, and so on for the rest of the tabs. This principle must of course be balanced against the needs to present information that has natural connections together.

In addition to promoting type of information based on frequency of use, it is also possible to promote instances based on how often they are accessed. E.g. by keeping an easily reachable list of the instances last accessed may spare the user from a lot of work. Also having flexible sorting mechanisms is a useful means to group instances.

4.4 Problem 1.1.o.3 – Mechanisms for packing information

Main problem area: Utilizing screen space
Problem area: Screen space in general
Main source for problem: Other

4.4.1 Background

In addition to the issues related to grouping treated above, packing may also be used to exploit the screen space more efficiently. Packing information is about putting more information into one dialog than what is usually done. This usually either causes elements or the amount of white space to become smaller, but more radical solutions (see below) may succeed in packing information without making the visual elements smaller. Packing in this context is primarily about packing which attributes that are possible to show, not being able to show a larger number of instances (although visual solution may facilitate the latter – as may also filtering mechanisms).

4.4.2 Problem

Information packing may be used both as an alternative and a supplement to grouping (see Problem 1.1.o.1 – Principles for grouping information and Problem 1.1.o.2 – Mechanisms for grouping information), but as indicated above, packing reduces the need for grouping.

Some packing mechanisms – specially using smaller elements and reducing white space, but also using alternative UI controls – may be in conflict with standards for UI design on a given platform. By conflicting the resulting UI will easily get usability issues. Specially, readability and ease of learning may be decreased when packing, but on the other hand, the interface may be faster to use, as less navigation is needed. If the UI controls are smaller, they are also usually more difficult to operate.

4.4.3 Solution(s)

An obvious solution in order to pack information is to **reducing white space**. This is obtained by putting the UI controls closer together than what is expected. The benefit from doing this is mainly that there is room for more information in the window without reducing readability of each control. The main disadvantage is that the UI easily will become crowded, untidy and "ugly". Related to this, it will usually also not be according to the look and feel standard of the platform. This solution is very simple (and inexpensive) to apply.

One simple way of reducing the size of the elements in a UI is to **reduce font size** or **use a more compact font**. This may increase the amount of information that fits on a screen, possibly sacrificing readability and conformance with standards. Also this solution is very simple (and inexpensive) to apply.

In the same way as reducing the font size, it is also possible to make **alternative UI controls**. This may involve significantly more development work, and thus becomes more expensive than just changing fonts. As with reducing font size, it may also compromise readability and standards. On the other hand, this solution may also be combined with implementing a company brand (see Problem 3.1.p.1 – Design that both supports branding, is aesthetic, and utilize screen space optimally).

A completely different approach is to packing information is to use offer **zooming** functionality. This is seldom used on other than graphical elements in a screen, but is indeed a possibility to use on the whole UI – even on a forms based one. Usually, it must be combined with scrolling – the

whole idea of zooming in this context is to have UIs that initially are larger than the physical screen. The Citrix terminal server client for PocketPC is an example of an application offering this. Zooming may well be combined with a panning solution to facilitate the scrolling functionality. One way of realizing this is to have a miniature version of the screen as a panning area, or having a special panning mode. Panning has a number of advantages to scrollbars in the case of zooming, i.a. that there is more room for the information, that scrolling in both directions at the same time is possible, and that dragging the UI directly is more intuitive than manipulating it indirectly through scrollbars. Panning is best suited for moving the UI short distances, but may in fact be a supplement to scrolling.

Independently of whether zooming is used, **modeless panning** without a designated panning area is a possible (though not usual) solution. This may be realized by letting a drag operation on non-active parts (the background) of a UI cause the panning. An example of using this solution in a commercial product is the ClearView PDF Reader on the PocketPC platform. This panning solution should preferably be used as a supplement to scroll bars. If it is not, other visual indications of where the UI may be moved must be added.

A more radical solution to packing information is to use **visualization**. In this context we include various ways of having a more visual presentation than what is common in data base / forms based applications. This includes finding visual metaphors through which the information is presented (and manipulated) and using different information visualization techniques. Examples of the first are to use a map to present geographical information or a thermometer to present a temperature. Examples of the latter is to use a bar graph instead of a grid of numbers or to present information as icons on a static background, where both location and visual attributes are given by information from the application. Visual solutions often fit well with drag and drop as a manipulation mechanism. Such solutions may be both easier and more fun to use than traditional ones, and may also give the user better overview of information. The main problems with this solution are that such systems may be expensive to implement, and that it is challenging to find an appropriate visualization, especially if the information in the application is abstract.

An important mechanism to use in solutions utilizing visualization is **focus & context**, i.e. to present overview and details at the same time. Using fish-eye techniques is one example of this, i.e. a visualization where the elements that the user focus on is magnified, and the further the information is from the focus point, the smaller it is (in fish-eye techniques, the magnification is not linear, rather logarithmic). Also other techniques may be used, as long as the principle of showing both details about the element in focus and the context in which it operates is maintained. Focus & context is of special importance when designing visual solutions, but may also be applied in forms based solution. Having a list + details presentation may utilize this principle if the data in the other items in the list give the user helpful information. Choice and sequence of attributes to show and sorting of the items in the list are important means to make the other items useful. Also showing derived attributes from “owner” classes of 1:n relationships in the “member” window may be viewed as a simple utilization of focus & context.

4.5 Problem 1.1.o.4 – Horizontal scrolling

Main problem area: Utilizing screen space
Problem area: Screen space in general
Main source for problem: Other

4.5.1 Background

As indicated in Problem 1.1.p.1 – Presenting elements in lists, scrolling should generally be avoided, and horizontal scrolling is usually worse than vertical scrolling. One of the reasons for this is that information on the same line usually is closer connected than information on different lines. This means that the user looses more context information when scrolling horizontally than vertically.

4.5.2 Problem

Using the selection and projection analogy once more, avoiding or handling horizontal scrolling (especially for lists) is often a question of projection, i.e. which attributes that are to be shown and in which sequence. If the projection is wrong (compared to the user's need), horizontal scrolling is often needed.

Horizontal scrolling should be avoided in all UIs, but is probably worse on PDAs than on larger displays, partly because the amount of context information is larger when the screen is larger. Also, on a larger screen, it is usually possible to make the window larger to decrease the need for horizontal scrolling. And even worse, because the screen is smaller, the need for horizontal scrolling occurs more often.

4.5.3 Solution(s)

This problem has much in common with Problem 1.1.p.1 – Presenting elements in lists and Problem 1.1.o.3 – Mechanisms for packing information. Thus, some of the same solutions may be used. From the list problem, optimizing the sequence and size of the attributes shown also apply in this more general problem. The same is the case for avoiding scroll bars but keeping the scrolling functionality, although this may require different realizations in the general case. From the packing problem (Problem 1.1.o.3 – Mechanisms for packing information), all solutions naturally also contribute to reducing the need for horizontal scrolling.

Looking specially at avoiding horizontal scrolling, **changing the layout** may solve this without using any of the packing solutions. This will usually increase the need for vertical scrolling.

A variant of changing the layout is to **change the screen orientation.** If this is an option on the platform, it will by default reduce the need for horizontal (and increase the need for vertical) scrolling. An important choice if this solution is used is whether the user should be given the opportunity to switch between landscape and portrait, or if only landscape should be available. The first choice imposes a number of new problems described in Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode. The latter choice is easier to realize, but offers less flexibility for the user, and may reduce which devices/versions of the operating system that may be used to run the application.

Changing the screen orientation may be considered being a fairly simple way of changing the layout of a UI, which in many cases will not cause any layout changes at all. In other cases it may cause a need for moving or regrouping some fields. This is an example of what we call a **simple redesign** – usually restricted to changing the layout of a given set of fields. This is an inexpensive

solution to the problem of horizontal scrolling, but may not be sufficient in some cases, either because the UI becomes too cluttered or requires very much vertical scrolling to avoid the horizontal one.

A slightly more radical solution is to perform a of the UI. By this we mean to do more than just a layout change, but keeping the style (usually forms based). In the case of a forms-based UI this typically implies changing the controls that are used in the UI to more space conservative ones or to use controls that may replace a number of other controls. A medium redesign may also imply redesigning which information that is to be presented in which window (often resulting in a larger number of windows). This is also a fairly inexpensive solution (depending on the degree of changes), which may work in more cases than just a simple redesign.

The most radical solution is to perform a **major redesign** of the UI. By this we mean to change the style of the UI, e.g. from a forms based UI to one using a more visual metaphor. This is usually more difficult and may be much more expensive than a simple or medium redesign, but may also result in a much more user friendly solution. Such changes may not only reduce the need for horizontal scrolling in a single window, it may reduce the total number of windows in the application.

4.6 Problem 1.2.p.1 – Presentation (which types of data) based on models or data – how to do this on a small screen

Main problem area: Utilizing screen space
Problem area: Flexible user interfaces
Main source for problem: Pilot

4.6.1 Background

Traditionally, it is considered that there should be a distinction between the model of a system and the system. I.e. the model of the system is depicted in the development phase, and is considered to be fairly stable, so that changes in the model over time can be handled together with bug fixing as a normal maintenance process. This model is considered being clearly separated from the data that is entered into the system, which of course changes all the time.

This traditional model is not always sufficient. In some cases, also (parts of) the model needs to be considered as data – or the viewed in a different way, some of the data stored in the system act as a model for how the system is to behave. A simple example where the distinction is not practical is in situations where a user organization needs to define additional attributes to be handled by various classes of a system. To be able to do this without changing the system, the definition of the attributes must be part of the data stored in the system. This causes a need for mechanisms to handle dynamic layout of a UI based on the data about these attributes.

Defining additional attributes (and other model changes available at run time) is usually done by some type of system administrator or super user, not by the end users. Another variant of this problem is the case when behaviour of the system is changed based on the *data* that the end user enters. A simple example of this is when there is an “other” choice available in a radio button letting the user enter a value that is not in the normal radio button set. The field for entering this additional value is usually hidden or disabled until the “other” choice is clicked. In more complicated cases, a number of attributes or a whole part of the UI may change depending on the data entered by the user (usually additional parts are added when some choice is made in the UI, or a specific value is entered in a field). This may also cause a need for a dynamic layout.

4.6.2 Problem

From a UI viewpoint, the first sub problem – having part of the model as data – is the most challenging. In most cases this causes a need for having dynamic layout of the UI for the part of the model defined as data. By this we mean that the program needs to have an algorithm that is able to make a reasonable layout of the UI for the attributes that are defined as part of the data. To make a good layout algorithm is not easy, and it is specially challenging on a small screen.

The second sub problem – changing content in the UI based on data entered by the user – is usually easier to handle, especially if the changes are limited. Even if the changes are not limited, they are usually known at design time. This means that there is usually not a need for a layout algorithm. Thus the challenge in this case is not making the layout, but how it is perceived by the user. If the UI changes often, or large parts of it changes, the user may be confused.

4.6.3 Solution(s)

Handling the first sub problem in a general case is challenging. Building a full-fledged layout algorithm is often beyond the scope of an ordinary business application, so this approach is probably more relevant for standard applications for specific trades (that will be sold to a number

of customers with varying needs) than for applications tailored for one organization. On the other hand, if the attributes to use in a specific part of an application changes often, it may also be relevant for more specific applications. See also Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size.

To outline a layout algorithm is outside the scope of these guidelines, but what is important is to have basic knowledge of the approximate number of attributes that are to be included, and to have available mechanisms for the user entering the attributes to group the attributes if there is not room in one screen. It is also important to collect enough meta information about the attributes to be able to generate a reasonable UI. In addition to name, label and data type, number of characters and domain information should be collected. Regarding choice of UI control for such attributes, there are three main strategies. The easiest is to use ordinary text entry fields for all. This will cause a simple, but dull and not very usable UI, but it makes the layout algorithm simpler. A different strategy is to let the user entering the data about the attribute also choose which control type that should be used. The risk with this strategy is that the user makes silly choices, making a bad UI. The most advanced strategy is to have rules that choose UI controls based on the meta information about the attributes. The two last strategies could be combined in the way that the user makes the choices, but the system restricts the available choices of UI controls based on the meta information.

Handling the second sub problem is easier than the first, but giving general advices is not very easy. Usability experts do not agree whether a UI should be stable in the sense that all available choices should always be visible (but disabled when not relevant) or should be as simple as possible (minimal information as a start), and supply additional choices based on user actions. The first strategy benefits from stability, but may be difficult to navigate for an inexperienced user. The second strategy benefits from simplicity, but may be perceived as being unstable and unpredictable. An important basis for choosing one of the strategies is the amount of users that use the different parts. Parts used by just a few users may be hidden, while parts used by most users (but not very often) may be disabled. But even parts used by all users may be considered hidden if they are used very seldom. In not obvious cases, experiments and/or user tests may be needed to find the best solution. See also Problem 3.1.p.3 – Visually coding of entry fields to mark editability (must, may, may not).

4.7 Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden

Main problem area: Utilizing screen space
Problem area: Flexible user interfaces
Main source for problem: Pilot

4.7.1 Background

On PDAs without keyboard, a common solution for entering text is to show a software keyboard on the bottom of the screen where the user can enter text using the stylus. The area in which the keyboard is shown may already be used by the application. This means that the application have less room for its “normal” interaction.

4.7.2 Problem

The main problem when designing a user interface that should be able to handle that the software keyboard comes and goes is to find out how to resize the dialogs. Resizing may just imply adding a scroll bar (if a scroll bar is already be present it only needs to be adjusted), but often some other adjustments are needed. If no adjustments are done, some parts of the dialogs will not be visible. If the keyboard is not automatically hidden when text entering fields loose focus, the user may need to manually remove the keyboard before continuing to work with the application.

The severity of this problem is depending on the type/style of the user interface. If the UI only contains an arbitrary text, adding or adjusting the scrollbar is a sufficient solution. A forms-based UI may be much more difficult to resize. The same may be the case for a more visual presentation, but this is depending on whether the visualization is tailored for the screen size or not (see Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size). Handling tab folders and buttons that are usually placed on the bottom of the screen is also a challenge.

In a forms-based UI, it may also be the case that resizing a dialog (e.g. by adding scroll bars) may cause the element that has focus to be placed outside the visual part of the dialog. This may cause the need for doing program controlled scrolling of the UI.

Related to the focus problem is a more general problem of how a user normally should traverse fields in a forms-based UI on a PDA. Should the tab or return keys be used, should there be a special next field button, should the UI advance automatically, or should the user be forced to click explicitly in each field using the stylus?

4.7.3 Solution(s)

The solutions presented in this problem focus on forms based UIs, because it is mainly for this type of UIs that this problem occurs.

The most obvious and most simple solution to this problem is (as indicated above) to **add or adjust scroll bars** when the keyboard appears. As indicated above, this solution is not optimal, especially not for forms based UIs, e.g. there may be a need to auto scroll to keep the field having focus visible. The other solutions presented below are solutions where the need for adding scroll bars are removed or reduced. A minor – but still annoying – effect of adding vertical scroll bar when the keyboard is shown is that adding a vertical scroll bar makes the screen width smaller, which in turn may cause the need for adding a horizontal scroll bar as well.

In some cases not solving the problem may be OK. By not solving the problem we mean **letting the keyboard cover part of the UI**. How "bad" this solution really turns out to be is of course depending on what is placed on the part of the screen that will be covered by the keyboard. If this part is just occupied by output fields, the solution may work fine as long as the keyboard is removed when not needed (preferably automatically). If this part of the screen contains important input fields the solution is useless. There will always be cases of doubt, like when there are a number of command buttons where the keyboard is shown, and it is difficult to remove the keyboard automatically (e.g. because all the other fields in the screen are input fields). In this case the user must remove the keyboard manually to reveal the buttons. In the best cases this is annoying, in the worst cases it may cause the user never to detect the buttons (i.a. because they are covered when the window is shown). The button hiding problem may be even worse it the keyboard covers tab folder in the same position.

Another simple, but seldom very practical solution is to **just use the part of the screen that will not be covered by the keyboard**. In practice, what this solution does is to reduce the size of the screen. This solution is quite common on dialog boxes, often combined with tab folders to have room for more information – but is seldom practical for normal windows.

A more advanced – but still fairly easily implemented solution is to **use one large UI control as a buffer**. By this we mean that when the keyboard is added, one of the controls is reduced vertically to be just as much smaller as the size of the keyboard. Controls that may be used for this is primarily list boxes and multi line text boxes. For this solution to be feasible there needs to be such a large control in the UI. For list boxes, this is often the case.

Yet another fairly simple solution is **having two variants** of the UI, one that use all the screen space, and one that makes room for the keyboard. This solution will often boil down to adding scroll bars, but may also be done using separate layout in the two variants. The main disadvantage with the solution is – in addition to added development work – that the user may be confused if the UI changes significantly when the keyboard is added. It is also restricted to situations where no other dynamic changes in screen layout (like changing screen orientation) occur.

The buffer solution may also be used with two or more large UI controls sharing the amount of size reduction to be applied. Generalized, this solution ends up as **dynamic resizing** of the controls in the UI. This may be done using two different approaches. The first is to decide a resizing rule for each window and apply that as tailored code for each window. The second is to have a general layout adjustment algorithm doing it for all windows. The first solution is easier to implement for each window, but requires the same work to be done over again for each window. The second solution is more difficult and requires much more work (as a starting point), but once it is done it does not require any additional work for applying it to new windows (and even new applications). Still, the investment needed for adding a layout adjustment algorithm will seldom be viable just to handle the keyboard problem. In this context it is important to mention that this problem is a specialization of the more general problem of having a dynamic layout of windows (see e.g. Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode, and Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size). That means that in cases where the more general problem is solved, this problem is automatically handled as a bonus.

4.8 Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode

Main problem area: Utilizing screen space
Problem area: Flexible user interfaces
Main source for problem: Other

4.8.1 Background

Switching between portrait and landscape mode is functionality that traditionally has been available as tailored functionality in specific applications, where the developers have found it useful. At least on the PocketPC platform, using only landscape mode as an alternative to portrait has been quite common in games, but applications supporting both have been rare – at least for the application as such, it is more common for special features like showing a picture in full screen mode.

With the arrival of the PocketPC 2003 SE version, switching between landscape and portrait has become standard functionality in the operating system, and all included applications support the switching. When activated, applications that do not support landscape mode will force the PDA to switch to portrait mode. The support for switching screen direction is also included in the new Windows Mobile 5 (WM5). Also, some new devices supporting WM5 (like QTek 9000 and QTek 9100) use landscape mode as standard when the built-in HW keyboard is used. This facilitates having a larger keyboard than previous models. On the 9000 model which has VGA screen resolution (640x480), landscape mode does not pose usability problem in most applications. Symbian does not support landscape mode in the operating system, while Palm does it for selected models.

This problem is primarily a challenge when designing forms based applications. Document based ones or visual solutions (like a map based application) are usually much simpler to adapt to changes in screen direction.

4.8.2 Problem

As the switching functionality is part of the standard functionality in PocketPC, it is expected that applications support it. Of course, this it a question of cost/benefit whether supporting all features in the operating system is worth the costs it will imply. Thus, a developer organization should have a clear notion of what the benefits of supporting landscape mode (in addition to portrait mode) have for the user.

Generally, the main benefit – from a user's point of view – by supporting landscape mode is that the interface facilitates more information horizontally without having to scroll. As mentioned above, vertical scrolling (which will be used more in landscape mode) is not as bad as horizontal scrolling. On the other hand, still from a user's point of view, there is a problem with landscape mode and using SW keyboard, as the relative amount of screen space occupied by the keyboard is larger in landscape than in portrait mode (on the PocketPC platform).

Switching to a developer's point of view, the main problem with introducing landscape mode is added development work. What the added development work will imply is depending on how the problem is handled (see Solution(s)).

To utilize swapping the UI, it is really needed to do both horizontal and vertical resizing of the user interface. Vertical resizing may be avoided by adding a scroll bar (see Problem 1.2.p.2 –

Handling crowded dialogs when software keyboard is shown and hidden). In any case, it should be avoided ending up with a common user interface that is a small square (with sides equal to the screen width in portrait mode) that fits in both modes. A solution with a large square (with sides equal to the screen height in portrait mode) that needs horizontal scrolling in portrait mode and vertical scrolling in landscape mode should also be avoided.

4.8.3 Solution(s)

When it comes to the solution space, this problem has much in common with Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden. Solutions involving scrollbars are less suitable for switching orientation. The same is the case for using only the common screen estate (as discussed in the problem part above). The solution of ignoring the problem is not at all applicable, as a UI with parts of it never visible is not usable at all. On the other hand having dynamic layout is very relevant – both the variant where it is tailored to each window and a general layout mechanism.

A solution that is more relevant for this problem is **having two versions of the UI.** This is a special case of having dynamic layout tailored for each window. The reason why this is more relevant when switching UI direction is that the total available screen estate is the same in both variants, so that optimizing “by hand” for portrait and landscape mode makes more sense than optimizing for the same direction with and without keyboard. The latter usually requires resizing controls, while switching direction may be best solved using a different layout between the controls. This solution is better suited for this problem than for solving Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden, where it is also mentioned.

Having a general layout mechanism for this problem is more complicated than for the keyboard problem. The reason for this is in fact the same as for why having two versions is more relevant for this problem. It is more difficult to make an algorithm that analyzes the need for layout changes when both the width and height changes than when just the height changes.

Looking at this problem from a general point of view, it is a special case of having a UI that supports resizing on a desktop platform. In addition to using scroll bars to handle this more general problem, it may be handled using so-called **struts and straps**. This means general mechanisms that maintain a set of constraints on how different controls should change when the shape of the UI changes. Such constraints may be max and min size of controls, max and mean space between controls, which side the UI the controls should stick to, etc. These mechanisms are able to handle more complex changes than switching between two screen orientations of the same screen size, but may still be applicable. On desktop platforms, struts and straps are usually supported as general properties on the UI controls combined with invisible controls (e.g. virtual glue) and layout managers negotiating with the controls which size and position they should use after a layout change. The concepts and model just presented is adapted from the available mechanisms in the Java Swing set UI components. On the Windows .net framework, similar but less advanced mechanisms are used, i.e. the controls may be either be anchored to their parent component in any combination of their four sides or they can be docked to one side (or filling from the centre) of the parent. Unfortunately, neither the Swing set nor the .net mechanisms are available on the PDA versions of these frameworks. This means that until such mechanisms will be available, developers may use them as an inspiration for self-made layout mechanisms.

It should also be mentioned that the combination of having a UI that switch between portrait and landscape mode and handling the software keyboard in reality gives four different screen sizes to adapt to. A generalization of this is treated in Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size.

4.9 Problem 1.2.o.2 – User interfaces that are able to run on equipment with different screen size

Main problem area: Utilizing screen space
Problem area: Flexible user interfaces
Main source for problem: Other

4.9.1 Background

Some applications and services must be available on different types of equipment. Of this is the case when the organization developing the application / offering the service does not have control of the types of equipment that are exploited by their users. Such systems usually have a flavour of being for the "general public". For specialized systems for one organization or a limited set of users it is often less expensive to buy new equipment to the users than to adapt the system to a number of arbitrary PDAs. The problem may also be present for "standard systems" for a given business sector. For "general public" and standard systems the actual deployment is sometimes done applying a compromising strategy in between, i.e. adapting the system to a small number of equipment types that together cover a large percentage of the potential users.

Developing systems that are to run on equipment types with differences poses a number of problems. The problem focused here is handling user interface design when the screen size on the different equipments is different.

This problem is a generalization of the problems connected to SW keyboard (see Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden) and UIs that are able to flip (see Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode).

As with Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode, this problem is primarily a challenge when designing forms based applications. Document based ones or visual solutions (like a map based application) are usually much simpler to adapt to different screen sizes (and aspect ratios).

4.9.2 Problem

As for the flip problem, the main problem with UIs that are to run on different screen sizes is added development work. Although related, the problem may both be more and less diverse in the way that the differences between the relevant screens may both be larger and smaller than between portrait and landscape mode of a given screen. E.g. it is important whether all screens have the same orientation, or there is a mix. What the added development work will imply is depending on how the problem is handled (see Solution(s)). The goal given this problem will often be to find a solution that makes it possible to avoid having one edition of each window for each screen size supported. Here, we have restricted the problem to handle different screen sizes. If an application in addition supports different platforms, the set of available UI components may differ in addition. This makes the problem more challenging, but some of the approaches described below are able to handle this.

4.9.3 Solution(s)

In Problem 1.2.p.2 – Handling crowded dialogs when software keyboard is shown and hidden and Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode balance between investing in specialized versions for each variant the UI should adapt to and

implementing general solutions that are able to adapt any UI to the different resolutions. When generalizing this problem, an additional dimension occurs, i.e. whether the general mechanisms should be able to adapt the UI to any form or just to a set of predefined resolutions. The ability to adapt to “any” form (possibly with some constraints) adds an extra level of complexity for an automatic layout mechanism – but once one decides to make a general layout mechanism, doing it halfway may seem a bit odd. But as with many such issues it boils down to a cost/benefit assessment.

Both a general layout mechanism and the strut and straps solution presented in Problem 1.2.o.1 – User interfaces that facilitate switching between portrait and landscape mode are only relevant for this problem if the variations in screen sizes are modest. If the differences are large, also changing the contents to be placed in each window may be needed to exploit the possibilities on each platform. Making a general solution for this requires an even more fundamental solution involving some kind of model reasoning. It should be mentioned that such a solution indeed requires a layout mechanism to decide the layout once it has been decided which controls to include on the UI. This means that the additional functionality needed is the one being able to make this type of decisions. As for a layout mechanism, also a model-based reasoning mechanism may be made application specific or generic.

4.10 Problem 2.1.p.1 – Mechanisms for entering text

Main problem area: Interaction mechanisms
Problem area: Handling input
Main source for problem: Pilot

4.10.1 Background

A central design criterion for a PDA is size, i.e. it should be small enough to fit in its user's pocket. As for screen size, this is in conflict with certain usability criteria. In the case of mechanisms for entering text, the size of the equipment is in direct contrast with having a normal size keyboard available. This makes entering text at the best different, and usually much more difficult than using a keyboard on a stationary PC.

The main mechanisms for entering text on PDAs may be split in two axes with two variants in each. Along one axis, there is a distinction between keyboards and stokes-based input. Keyboards are either SW keyboards or small physical keyboards. Strokes-based input is usually performed in a designated area, either on the screen (similar to SW keyboards) or on a pressure sensitive area outside the screen. Some strokes-based input (like the Transcriber on the PocketPC platform) can be used anywhere on the screen.

Common for all built-in physical keyboards is that they are so small that the user must enter text with one or two finger. Common for SW keyboards and all strokes-based input is that it is difficult to operate them without using the stylus. Entering national character like the special Nordic letters (often referred to as accented characters) often requires special efforts by the user. This is particularly the case for physical keyboards and strokes-based mechanisms.

4.10.2 Problem

When designing a PDA application where the user must enter some text, there are two related problems to solve: how to avoid that the user must enter text and how to make it easier for the user to enter text. These two approaches are so interconnected that it is difficult to make an accurate distinction between them, especially when considering solutions (see below). The main goal in most cases is that it as much as possible should be avoided that the user is forced to use the generic text entering mechanisms (keyboard/strokes-based input).

4.10.3 Solution(s)

An obvious solution for making it easier to enter text is to use **auto complete**. This is a mechanism that tries to guess what the user is about to write and suggests this by filling in the suggested text ahead of the writing of the user. On the PocketPC platform one form of auto complete is included in all the generic input mechanisms, i.e. a pop-up list of possible completion(s) of the word being written. To apply the suggestion, the user must actively choose it. This mechanism is also adaptive, in the way that it remembers words written earlier in the same session with the given document. Except for this, it only suggests words from a dictionary in the language of the operating system. This means that while writing Norwegian using an English version of PocketPC, the mechanism is of modest help. Such generic mechanism is not available in Palm OS and Symbian. Some applications (regardless of platform) implement auto complete in certain fields. E.g. this is common when writing an URL in most web browsers on most platforms. It is also common when writing names in an email client. Common for such solutions is that they are always adaptive, i.e. they use the history of values used earlier to suggest the new ones. This type of mechanism may be very helpful, especially when long and/or not too intuitive values should be entered.

A related solution to auto complete is to use **predefined values**. By this we mean having a list of all (or usually the most common) texts to enter in a field. The values may be accessed from a menu (especially if the values may be used in different fields), or from a combo box (combination of dropdown list box and text field). If the number of values is small and predefined, using radio buttons to choose the value is also an option. If the number of possible values is large and not restricted to predefined values, only the most common ones should be presented. In this case, the list should not be longer than what fits on the screen (to avoid scrolling) and may well be adaptive to the actual values entered/chosen by the user.

Presenting predefined values as a set of radio buttons fits better as an example of a different solution, i.e. **alternative input mechanisms**. By this we mean using UI controls that do not require using a keyboard or keyboard replacement. In addition to radio buttons, (dropdown) list boxes, check boxes, spinners (input field with up/down arrows to browse through values (called updown in .net compact framework)), sliders (called trackbar in .net compact framework), and menus are the most common controls for entering values without having to type, but in some cases, buttons may also be used. Most of these mechanisms require that there are some sort of restrictions on the domain of the attributes that should be entered through the mechanism.

The alternative input mechanisms just discussed are generic mechanisms that can be used to avoid having to use a keyboard. It is also possible to make **specialized input mechanisms**. By this we do not primarily consider self made UI controls, but rather using (a combination of) existing controls in a new way to implement a creative solution. An example of this approach is the mechanism used in an application for service technicians implemented by IT Liberator, where the user may write common fault description in a natural language like syntax by choosing from four drop down list with commonly used nouns, verbs and preposition expressions. Having a restricted number of values in each dropdown list still facilitates entering a very large number of possible sentences in a simple way. Such mechanisms are usually application domain specific – at least the values to choose from.

Most of the solutions presented in this problem exploit a general principle that may be considered a solution in itself, i.e. **exploiting domain knowledge**. Taking advantage of knowledge about the domain area is often a question of restricting the set of possible values to enter. As soon as this number is at a certain low enough level, some of the solutions above may be chosen. In the cases where it is not possible to restrict the set of legal values, identifying the most common ones (i.e. the ones that it is most likely that the user will enter) may facilitate using the same or similar mechanisms.

A solution in the same category as exploiting domain knowledge is to have **dynamic behaviour based on actual use**. By this we mean that the application tracks how it is used (in this context mainly which values that are entered) and adapt the values presented in a given input mechanism based on this. To make solutions based on this principle more user-friendly and predictable, it may be combined with having special functionality in the application letting the user adjust which values that should used (like the “edit my texts” functionality in the messaging application on the PocketPC platform).

The solutions presented so far has all focused on making text entry from a keyboard easier or replacing it with various on-screen mechanisms. An alternative is to try to collect the data from some other source than user interaction, usually by **exploiting contextual data**. This principle is based on an assumption that the context in which a mobile user operates changes more rapidly than it does for a stationary user. The idea is that if an application is able to obtain knowledge about the context, this knowledge may be used to make the application more user-friendly.

Regarding entering text, being more user-friendly means that the application obtains data that the user would have had to enter if the application did not have this ability. A very simple example (that also applies for a stationary user) is that the date and time of a given event is entered automatically from a system clock instead of being typed by the user. Another example is using an rf-id sensor to identify a piece of equipment that is to be inspected so that the user is relieved from enter a long and cryptic equipment id. This topic is discussed more exhaustively in section 5.2 (Exploiting that the user is mobile).

A related alternative to exploiting contextual data is to use **multimodal input**. Although this by definition involves user interaction, the principle is related in the sense that the goal is to avoid both keyboard and screen based interaction. It is also related because mechanisms for multimodal input are sometimes used as a means for obtaining contextual data. E.g. using an rf-id sensor is primarily a mechanism for obtaining contextual data, while a bar code reader is primarily an input mechanism. This is spite the fact that what really happens on a fairly low level of abstraction is identical. By multimodal input we partly mean using alternative input mechanisms to keyboard, hardware buttons and screen interaction (e.g. bar code reader and voice input), and partly using more than one input mechanism at the same time. The former is easier to exploit in an application as it is usually possible to rely on mechanisms outside the application for obtaining the input. The latter often requires some kind of interpretation from the application. Multimodal input is discussed more exhaustively in Problem 2.1.p.4 – Multi modal interaction – stylus, scanner, RF-ID, different types of keyboards, voice control.

An alternative for situations where large amounts of data should be entered is **using a voice recorder**, and transcribing it manually in a stationary context later. In some cases, this may be combined with speech-to-text translation on a stationary computer, specially if the text is regarding a limited application domain.

A solution not discussed so far is to include external (nearly) full-size keyboards. Although such keyboard may solve the problem, they are of limited use. The reason for this is that once the user must connect an external keyboard, he must also sit down (or at least find a fairly horizontal area to place the keyboard). This severely limits the mobility of the user.

In earlier versions of operating systems for PDAs, only strokes and SW keyboards with QWERTY layout were available as generic mechanisms. In addition, some applications offered tailored keyboards like a keypad for phone or pin code entry. A general trend has been that the number of available mechanisms has increased. One the one hand by offering HW keyboards on an increasing number of devices, and on the second hand by offering different SW solutions for entering text. In addition to strokes and QWERTY keyboard, transcriber (writing whole words anywhere on the screen), and T9 have become standard. Furthermore, some applications also offers alternative SW keyboards like alphabetic ones and T9 for applications also on OSs not supporting it as a generic mechanism. An important reason for this development is that it offers the user choices for how to type. This is important, as different users prefer different solutions, both based on personal preferences and prior experiences.

An example of extreme flexibility is PSIs dynamic keyboard that is configured using an XML file. In principle, this opens for letting each user design his own keyboard.

An interesting development that may become a standard keyboard later is the SHARK keyboard from IBM. It offers strokes on a SW keyboard. By combining this with keyboard layout that it optimized to make writing common words easier (of course language dependent), writing speed may be considerably increased.

4.11 Problem 2.1.p.2 – Order entry – needs to be fast even if it contains large amounts of data and there are a number of rules

Main problem area: Interaction mechanisms
Problem area: Handling input
Main source for problem: Pilot

4.11.1 Background

This problem has been brought up as partly as a typical example covering many of the general problems of forms based user interfaces on PDAs, and partly as a specific problem as such (because order entry is a (central) part of many applications).

What makes order entry interesting is partly that it involves entering “large” amounts of data and partly that there usually are a number of rules for how the data should be entered to be valid. Also, order entry is often done quite frequently, which may make a small efficiency gain while entering each order sum up to a much larger total gain. Furthermore, the user situation is often such that the user needs to enter the data quickly (e.g. because he communicates with a customer that does not have the time to wait while the user fumbles).

4.11.2 Problem

The main problem with order entry is to make it efficient. In this context that implies both that order entry should go fast, and that the chances of making errors are small. Making order entry efficient is mainly about avoiding having to enter text, i.e. finding data that is already available instead of entering data manually. It should also be mentioned that also the process of finding data must be performed efficiently.

Handling rules is to a large extent a question of not making it possible to break the rules – not about checking that the rules are obeyed.

4.11.3 Solution(s)

The most important – but least operational – issue when solving this problem is to make the UI optimal with regards to the task being performed. The tricks for making a solution efficient are often connected to the information that is available and required to solve the task. How to do this often boils down to individual problems addressed in other problem areas.

One important issue is to minimize the needs for text entry. This is handled in Problem 2.1.p.1 – Mechanisms for entering text, Problem 2.1.p.3 – Numerical keyboard (more general: mechanisms for entering numerical data), and Problem 2.1.p.4 – Multi modal interaction – stylus, scanner, RF-ID, different types of keyboards, voice control.

Another important issue is searching efficiently or using other mechanisms for identifying the relevant instance(s). This is handled in Problem 3.1.p.2 – Solutions for searching large amounts of data, e.g. multi-step solutions combining reliable identification and optimal screen space utilization (also horizontally) and to some extent in Problem 2.1.p.5 – Controlling the input cursor from an application.

A third principle to use is letting the system provide information automatically whenever possible. In the cases where the information that can be provided automatically is unique, this is always a benefit for the user. But the principle may also be used in situations where the information is not unique. In this case, there is a trade-off between the benefit of the system “guessing” a right value

and the extra work involved for the user if the system provides a wrong value. Also, providing non-unique information automatically is a source for errors, as the user may assume that the information is correct and not review it.

A related solution to providing information automatically is to use auto complete mechanisms when entering text. This is handled in Problem 2.1.p.1 – Mechanisms for entering text. The same is the case for using reasonable default values. This is also handled in Problem 2.1.p.3 – Numerical keyboard (more general: mechanisms for entering numerical data).

A final principle that makes order entry more efficient is to jump automatically between fields in a natural sequence (i.e. a sequence that fits the user task). This is handled in Problem 2.1.p.5 – Controlling the input cursor from an application.

Most of the solutions/principles referred to or discussed may be enhanced if the system learns from user actions, i.e. keep track of the most common values to enter, or the most common way of using a dialog, and adapts it to these patterns. For auto complete, default values, and automatic information this can be achieved quite easily. Adapting to other usage patterns like visiting sequence between fields may be more difficult to achieve.

4.12 Problem 2.1.p.3 – Numerical keyboard (more general: mechanisms for entering numerical data)

Main problem area: Interaction mechanisms
Problem area: Handling input
Main source for problem: Pilot

4.12.1 Background

As discussed in Problem 2.1.p.1 – Mechanisms for entering text, knowledge about the domain or other knowledge reducing the potential values to be entered should be exploited. Having special mechanisms for entering numerical data is an example of this. The problem ranges from having generic mechanisms for entering numbers to special mechanisms handling limited sets of numbers. The latter will in some cases resemble the general problem of entering text more than a general number entering mechanism.

There reason for having specific mechanisms for entering numbers is usually to reduce screen space occupied by a software keyboard (fewer keys needed), to reduce the number of errors, to avoid having input control and/or to make the keys easier to hit. Avoiding input control could in some cases be difficult to achieve, as the user may use other generic text entry mechanisms allowing arbitrary input as an alternative to the keyboard only allowing numbers. To avoid this, the generic mechanisms must be disabled or non-numeric values must be blocked. This solution is used in the phone application on the PocketPC platform, where the software keyboard is disabled and values from hardware keyboards are filtered.

4.12.2 Problem

Which mechanism(s) to use for entering numerical data is depending on the goal for using a special mechanism. If the goal is to reduce screen space occupation (and possibly reducing the need for resizing the screen) having a small numerical keyboard is an obvious solution. If the goal is to reduce the number of errors or to avoid having input control, any solution reducing input to numbers is applicable. On the other hand, if the goal is to make the keys easier to hit, having a large keyboard – possibly with a layout similar to the numerical keyboard area that is usually located to the right on a traditional PC keyboard is the most obvious solution. The main problem is balancing the goals if it is desired to have more than one fulfilled.

Regarding the goal of reducing errors, it should be marked that it is more difficult to control a number than a text visually, especially if the number contains more than a few digits. Moreover, the consequences of entering a wrong number (e.g. one extra digit in an amount to deposit or one wrong digit in an account number) are in many cases much larger than doing a similar error in an arbitrary text.

4.12.3 Solution(s)

As mentioned in Problem, the solution to this problem is to a large extent depending on the rationale for using a special numerical entry mechanism. Here we will present different solutions and present their pros and cons to indicate the benefits.

One common solution in general purpose applications is to have a **large numerical keyboard**. E.g. the phone application on the PocketPC platform uses this. This solution is very easy to use, even without using a stylus, as the buttons are large with big labels. The main disadvantage with this solution is that it requires much screen space. This means that it is best suited for situations

where the user should enter one number only (e.g. a pin code). It may also work well in single steps of a wizard based UI.

A different solution is to have a **small numerical keyboard**. The main advantage with a small numerical keyboard is that it requires few buttons and may be fairly small, and thus leave more space to show and/or receive information. Typical layouts for small keyboards are either almost quadratic ones or to have all keys in one row or column. The first solution is faster to use and less error prone than the second, but is usually more obtrusive for the rest of contents of the window. A small keyboard is difficult to operate without using a stylus. This solution is best suited for situations where the user uses a stylus to enter a number of numbers either just numbers or numbers in combination with other types of input fields. It is seldom worthwhile to have a special numerical keyboard in a window where the user enters just one (small) number, but a number of alphanumerical values.

Solutions based on having numerical keyboards facilitate both increased speed and reduced number of errors. In some cases they also eliminate the need for value control, but in most cases there is a need to do value control also when using numerical keyboards, e.g. if the value should be within a certain range. In many cases possible values for a numerical field can be divided into three sets: illegal values, legal and likely values, legal but unlikely values. One way of reducing the number of errors (independent of the input mechanism) is to **check for unlikely values**. I.e. if the user enters an unlikely value, a warning should be given. It should be possible to turn off such a test. This can be combined with using **default values** for fields where the value seldom is different from the default value. A default value also signalizes an example of a likely value. These solutions are examples of exploiting domain knowledge in the UI design. Exploiting domain knowledge in this context means that the UI has more knowledge about the value than that it is a number. This principle applies also for alphanumerical data, but checking for likely values may be more difficult if the values are not numerical.

Using default values may very well be combined with a spinner control, which then will be an example of a solution based on **adjusting instead of entering values**. This solution is based on an assumption that it is easier and less error prone to let the user adjust a likely value than to enter one from scratch. This solution is especially well suited if the number of digits in the numerical value is large and the variation between the likely values is small.

4.13 Problem 2.1.p.4 – Multi modal interaction – stylus, scanner, RF-ID, different types of keyboards, voice control

Main problem area: Interaction mechanisms
Problem area: Handling input
Main source for problem: Pilot

4.13.1 Background

Many of the problems in this problem area (handling input) are connected to using various types of keyboards. One way of solving these problems is to use something else than a keyboard for entering information.

The concept *multimodal* indicates that more than one mechanism is used for controlling a PDA. It covers both the case where different modalities are available, but only one is used at a time, and the case where a number of modalities are exploited in parallel. Although the latter can be viewed as "true" multimodality, we focus mostly on alternative modalities here (as this is more realistic to use in applications today).

As for specialized SW keyboards (see above), the main goal for using alternative modalities is to gain efficiency and to reduce the number of errors. E.g., it is obvious that it is a better solution to use a bar code scanner to read a 20 digit identifier instead of entering it using a stylus.

In addition to using stylus and HW keyboards (when available), the most common alternative input mechanism to use is a bar code reader (scanner). RF-ID is not so widespread, but has gained much attention lately. Voice control has since long been considered "promising" on PCs, but is only to a limited extent available on PDAs. Even though some general systems for controlling PDAs exist (like Microsoft's Voice Control for PocketPC), very few generic tools (SDKs) for implementing voice control on PDAs exist.

4.13.2 Problem

One of the problems connected to using alternative modalities is the technological challenges it imposes. These challenges are both connected to making the necessary extra equipment work, interfacing the equipment from the application (using SDKs etc.), and designing UIs for the alternative modalities. In this document, we will focus on the latter.

Another problem is that using alternative modalities usually requires that special hardware is available on the equipment to use. This limits the choice of equipment, and also makes it more difficult to change equipment (both type and instances).

As mentioned above, the maturity level is varying between different alternative modalities. Lack of maturity is most pronounced for voice control, but also implementing a RF-ID solution may pose challenges on some equipment types.

4.13.3 Solution(s)

As just stated, the major problems connected to using alternative modalities are technical ones. In these UI design guidelines, we will not focus on solving these. Instead, we will present some advices on which situations the different modalities (including traditional ones) are best suited (given that the technical challenges are overcome).

First we look at the traditional modalities for controlling a PDA, i.e. stylus, hardware buttons and hardware keyboards. Common for all these is that there they work de facto, maybe with the exception of an external keyboard that may involve some hassle to make work.

The main advantage of using a **stylus** is its versatility. It can be used to control almost all functions of a PDA with touch sensitive screen. It is also very inexpensive, and may be replaced by any fairly sharp object (including a finger(nail)). One disadvantage is that it is not well suited for entering arbitrary text unless some special mechanisms are applied (see Problem 2.1.p.1 – Mechanisms for entering text). There are also some usage situations where the stylus is not so well suited, partly in situations where the user wear gloves and partly in situations where the user needs to use one or two hands to perform the primary task.

The main advantage of using **hardware buttons** is that they reduce the chance of making errors. The buttons either have clear, predefined functions on system or application level, or performs predefined actions on the UI control that has the focus (typically a four-way navigation pad, usually with the possibility to perform select). To some extent, HW buttons also support one-hand use of a PDA – depending on the size of the PDA and the placement of the buttons. Although the HW buttons have clearly defined functions and actions, what these functions and actions are is not always obvious for an inexperienced user. And even if it is, operating e.g. a four-way navigation pad with select function is not trivial, usually depending on experience with each PDA model or brand. Operating a PDA using only hardware buttons is usually not possible, or at best very cumbersome. For entering text in a text entry field, HW buttons are to no help.

The main advantage of using a **hardware keyboard** is that it eases text entry. It usually also supports navigation between fields in a forms based UI. How well a HW keyboard is suited for entering more than small amounts of texts is depending on the size of the keyboard. Most built-in HW keyboards are very small and best fitted for one-finger text entry, which slows down typing speed. Also, such keyboards usually only use standard placement of alphabetic characters, i.e. other characters (in some cases including numbers) are placed in non-standard positions and/or require setting the keyboard in a special keys entry mode. New models like the Qtek 9000 offer a fairly large built-in keyboard, but still using some non-standard placement of characters and special characters mode. Most built-in HW keyboards may be utilized in a true mobile usage situation. To utilize a full-size keyboard with a PDA, external keyboard is needed. Some models supply foldable keyboards that also may be used as a placeholder for the PDA. This supports typing fairly well, but not in a true mobile usage situation. An external keyboard also is an extra piece of hardware to carry around, lose, etc.

The main advantage of using a **bar code reader** is that it eases identification of objects or places without having to type a long identifier or using a search function. It also is fairly simple to produce the bar codes (just a bar code printer is needed), so tagging the environment with bar codes is simple and inexpensive. To utilize bar codes in a PDA application is primarily a hardware challenge, usually involving off the shelf equipment. Quite a few industrial PDAs have a built-in bar code reader. This makes the equipment larger and heavier and poses extra challenges to battery capacity. There also exist stand-alone bar code readers that may be used together with standard PDAs, either wired through a serial connection or wireless through Bluetooth. To an application, a bar code reader just delivers a number. The main problem with using bar code readers is the extra HW needs, both regarding cost, weight and restrictions on which equipment that may be used. Of course, an application that is built for use with a bar code reader may also be used without it, but then the user must type the identifiers using a less suited mechanism.

It should also me shortly mentioned that a camera may be used as bar code reader (better for matrix codes than for traditional bar codes), but this requires decoding software either on the PDA

or on a server that the PDA communicates with. Using a bar code reader facilitates one-hand operation of the PDA, at least when using a PDA with built-in bar code reader. For an example of a service utilizing camera to read two dimensional bar codes see www.cybstickers.no.

The advantages of using **RF-ID based solutions** are the same as for bar code reader. It eases identification and reduces the need for typing identifiers/performing search. Tagging the environment is more complicated and expensive with RF-ID tags than with bar codes, but this is partly a question of how large volumes the RF-ID tags are produced in. Extra equipment on the PDA is less widespread than bar code readers, but is also much lighter. Another difference for an application is that while a bar code reader is triggered by the user and just produces a number to the application (which may be ignorant to the source of the number), an RF-ID reader must be triggered by the application – thus requiring more development efforts.

The advantage of using **voice control** (usually used together with some speech output solution) is that it utilizes hands-free operation of a mobile application. The main challenge with using voice control is to find a voice control toolkit that works on a given PDA platform. Running the voice control SW on the PDA is required unless it can be guaranteed that the PDA is always connected. If this is the case, a server based solution is an option. There is also a challenge to design and implement a good voice control dialog. Usually, voice control is used for issuing commands, and maybe choosing between a limited set of input values. So far, dictation systems work poorly even on desktop systems, so trying this on a PDA is not a good idea. Thus, for entering text, typing must be used (unless saving speech data as the value is an option). This means that only a limited set of applications can be entirely controlled by voice. Usually, voice control is only needed in a limited part of an application – i.e. the parts supporting tasks where the user's hands are occupied. Closely related to voice control is using audio feedback in applications. Strictly speaking, this has to do with multi media, not alternative modalities. Still, this is a useful mechanism for attracting the user's attention, and may also be used for confirming that operations have been completed. E.g. the SMS system on the PocketPC phone edition platform uses sound both to draw attention and to confirm that operations have completed, using different sounds for different messages. This technique is called earcons, to indicate that the sound play a similar role as icons in a purely visual UI.

For some of the modalities like the built-in ones and bar code reader, using them requires no special actions by the user (other than using the modalities). Others require that the user activates the modality. This means that the modality choice must have its own UI.

This problem is closely connected to obtaining information automatically through sensors (of which RF-ID is an example). Doing this is primarily a technical challenge, which – if it succeeds – is a benefit for the users. Designing the UI for such mechanisms is often a question of doing nothing, but there may be a question of whether the user should be informed that an information is obtained automatically or not, including whether the obtained information should be shown or not. It is not possible to give a general answer to this, as it depends on the user's task, and how the automatically obtainable information fits to it, e.g. if there is a need for the user to check whether it is the right information that is obtained, or to assess whether the obtainable information is relevant for the application/the user or not.

4.14 Problem 2.1.p.5 – Controlling the input cursor from an application

Main problem area: Interaction mechanisms
Problem area: Handling input
Main source for problem: Pilot

4.14.1 Background

A central characteristic of an event-based system (like MS Windows on a PC or a given operating system environment on a PDA) is that the main modus operandi is that the system reacts to user actions – i.e. the user controls the dialog, not the system. Despite this main principle, there are cases where it is relevant to let the application control some of the dialog structure. A common way of implementing this is to use a wizard-based UI style. A less imposing way to control (some of) the dialog is to let the application control the input cursor. This is done in all forms-based UIs on PCs, i.e. when the user presses the TAB key the application decides where to move the cursor. On a PDA operated primarily using a stylus, pressing the TAB key is seldom a relevant user task. Despite this, there may be cases where it is needed or relevant to force or guide the user by moving the cursor automatically.

When looking closer into this problem, it is necessary to make a division between guiding the user and forcing the user. Guiding the user should be done to make it easier for the user to visit the relevant parts of the UI in a natural sequence. Forcing the user should only be done when there is only one relevant or legal sequence in the UI.

4.14.2 Problem

The idea of guiding the user is based on an assumption that it is possible to predict what the user wants to do. Such a prediction is then operationalized in a tab sequence. Given that such a sequence exists and is indeed helpful for the user, it is still a challenge to find out when the user wants to visit the next field (unless the user uses the tab key on a HW or SW keyboard).

Forcing the user to use a specific sequence is very difficult to accomplish unless all fields but the one to use is disabled. On the other hand, restricting the user to a limited set of fields (specially only one) is not a natural design of a forms-based UI.

4.14.3 Solution(s)

Our main recommendation for both of the sub-problems is to make sure that there is a **connection between the users' tasks and usage patterns and the intended functionality** (this should indeed always be the case in a UI, but is of special importance to stress in cases where the application takes control of the dialog in an application). If this is not the case, the UI will be perceived as hostile and rigid.

As mentioned above, forcing a specific sequence of visiting the fields in a UI should be avoided in a forms-based UI. If there indeed is a need for a specific sequence, a wizard based UI should be used. If this is not appropriate, the UI and/or application logic should be changed so that a forces sequence is not needed. A possible exception is to force the user to enter the key field before the rest of the fields (possibly also locking the key field when the other fields are activated). Generally, enabling and disabling fields is a better solution than forcing the cursor to a specific field even when the user clicks at another.

Having a tab sequence as a guide for the user is always a good idea, but as mentioned above, the benefit is limited as the user must use a tab key on some kind of keyboard to utilize the sequence.

If it is important for the user to receive this kind of support, one possible solution is to **move automatically between the fields** wherever possible. This is possible in fields where there is a fixed or maximum number of characters. The disadvantage of this solution is that the user must move manually back to the previous field if he made an error in the last character typed in the previous field.

Another solution is to have a **“next field” button**. If this is desired by the user, using a hardware button for this is an applicable solution. Having a designated next field button on the screen is also possible, but is not recommended except for very special cases.

4.15 Problem 2.2.p.1 – Interacting with applications without using stylus

Main problem area: Interaction mechanisms
Problem area: Not using the stylus
Main source for problem: Pilot

4.15.1 Background

Some PDAs and SmartPhones are designed to be controlled without a stylus. These devices usually do not have a touch-sensitive screen. Other devices are designed to be used both with and without a stylus, i.e. there are generic mechanisms for controlling all UI components on the device using a navigation wheel, arrow keys etc. For the first type of devices, this problem is not applicable (as it is already solved). The applicability of the problem for the second type of devices is depending on how good the built-in support for non-stylus use is (this is partly supported on the OS level and partly on the UI control level). For most PocketPC and Palm PDAs, it is definitely relevant, but devices like Sony Ericsson P800 and P900 is more in a border zone.

For some users it is not practical to use the stylus. This could be because the user wears gloves, because it too cold to fumble with the stylus, because the user has only one hand available, because the user has lost the stylus, or because the user prefers not using it.

There are two main strategies for using a PDA that is primarily designed for stylus use without a stylus. One is to use something else (usually a finger) instead of the stylus, the other is to use hardware keys. When looking closer at this problem, it is necessary to make a distinction between these two strategies (even though an actual user may choose to use a combination of these two). It should also be mentioned that there is a big difference between controlling the PDA by using the fingertip and by using the fingernail. Once gotten used to, the latter is easier. The considerations below are mainly given on a worst case basis, i.e. operation using fingertip or gloves.

4.15.2 Problem

The most obvious problem that occurs when using the finger instead of a stylus to control a PDA is that the precision when pointing is coarser. Combined with the fact that a finger in addition conceal the UI more than a stylus, makes it even more difficult to hit small details using a finger than with a stylus. This problem gets even worse if the user uses gloves of some kind. The most obvious solution, i.e. making the controls bigger easily increase screen space problem present in most PDA applications.

For a user only using hardware keys, the main problem is to find relevant and intuitive support for the available hardware keys. As mentioned above, there is usually some generic support for using HW keys from the platform, but this may vary depending on which UI components that are used. In addition, it may be very relevant to use the HW keys to navigate the UI as such, a type of functionality that is usually not supported generically.

When giving support for using an application without stylus (independent of the two strategies), it should also be considered whether there are special part(s) of an application that are more relevant to use without stylus than other.

4.15.3 Solution(s)

Solving the sub-problem of controlling the PDA using a finger instead of a stylus is partly a question of choosing, partly a question of adapting and partly a question of making UI

components (controls). These three levels of solving the problem are proportional regarding cost vs. potential benefits.

Just **choosing most appropriate UI components** as well as **simple adaptation of UI components** has no additional costs connected to the components, but may not facilitate an "optimal" solution. By simple adaptation of UI components we mean adjusting the components' properties to make them more fit for finger use. Exactly which components that work best for finger control vary between the different PDA platforms. For the PocketPC platform the following table give some characteristics of the standard components (i.e. the components that are designed for user interaction) with respect to "finger friendliness":

Component	**Appropriateness for finger navigation**
Button	Standard Button size is a bit small, but given a bigger size, Buttons are OK for finger use.
TextBox	For entering text, it is sufficient to click on a TextBox – the rest of the interaction is done through some kind of text entry mechanism (see above). The latter may be far from trivial using the fingers, but this is outside the scope of this problem. Clicking on a TextBox is feasible using the finger when it has standard size, increasing the size will make it easier. Increasing the height may only be done by increasing the font size (unless it is multiline). If the TextBox has a value already, selecting this value when the TextBox gets the focus will ease finger use. Changing the text in the TextBox – e.g. by selecting and changing three characters in the middle of the text – is not trivial using just the fingers. How difficult it is depends on the font size used, but increasing the font size too much may easily result in a number of other usability problems.
CheckBox	CheckBoxes are not too difficult to operate with fingers, depending on the distance to other CheckBoxes (and other UI controls). To trigger a CheckBox, not only the tick box, but the whole control (including the text and any additional space around the text) may be clicked. Increasing the size and/or the font size will not increase the size of the tick box. Thus, given large enough size and distance to other components, CheckBoxes are easy to control using fingers.
RadioButton	RadioButtons have the same characteristics with regards to finger friendliness as CheckBoxes. As RadioButtons always appear in groups, the distance/size requirements are especially important.
DataGrid	The finger friendliness of the control has not been investigated in depth, but the standard size of cells in the grid is fairly small, and it does not seem to be an easy way of making the cells larger.
ListBox	A standard size ListBox is only partly suited for finger use, as both the elements in the list and possible scroll bars are fairly small. Increasing the font size will make the elements in the list larger, but also makes it more likely that there is a need for using scroll bars (that do not increase in size).
ComboBox	ComboBoxes have approximately the same characteristics with regards to finger friendliness as ListBox.
ListView	Using icons and LargeIcons as View, ListView may be appropriate to control with fingers (even though the control as such probably has limited applicability in many applications).
TreeView	A TreeView is difficult to control using fingers, and it does not seem like it is possible to make it more appropriate by adjusting its properties.
TabControl	The tabs in a TabControl are not too difficult to operate with fingers,

	depending a bit on the size. The size of each tab is partly dependent on the length of the text on the tab, and partly on the font size. But as discussed in Problem 1.1.o.2 – Mechanisms for grouping information, all tabs should fit on the screen to avoid having to use the scrolling features of the TabControl (which is not easy to operate using fingers). So there is a clear trade-off that need to be balanced.
ScrollBar	ScrollBars are notoriously difficult to use on a PDA, even with stylus (see Problem 1.1.o.4 – Horizontal scrolling). The ScrollBars size may be increased to enhance their suitability – but then of course leaving less space for other components. Using alternative scrolling mechanisms (see Problem 1.1.o.4 – Horizontal scrolling) should be considered as an alternative.
UpDown	There are two types of UpDown controls, one that can adjust numbers (spinner) and one that can adjust an arbitrary domain. These controls are identical with regards to finger friendliness. In their default size, they are almost impossible to operate using fingers, and there is no apparent way of adjusting their size or fontsize.
TrackBar	TrackBar are not specifically easy to operate using fingers, and there are no obvious ways of adjusting their properties to make them more suited. If there is a choice of direction, horizontal TrackBars are slightly easier to operate using fingers.
MenuItem	MenuItems (i.e. members of the pull-up menu on the bottom of the screen) are not too difficult to operate using the finger although the choices are fairly small. There are no ways for the application to change the size of its MenuItems.
ContextMenu	Items in a ContextMenu are the same UI controls as MenuItems in a main menu, and are used in the same way – but triggering a ContextMenu is more difficult than a main menu using the finger, as the user must hit the control to which the ContextMenu is connected.

Above we used the term simple adaptation of UI components to denote adaptation based on adjusting the properties of the components. This indicates that there may be an **advanced adaptation of UI components**. By this we mean adaptations that require programming. Typically, this is done by making subclasses of built-in UI controls. Both the possibilities for doing this, the effects it may have, and how difficult it is to do this type of adaptation varies between platforms, development tools and available libraries. Thus, it is difficult to give general considerations about this, but when tailoring an existing UI control is an option, this is usually a less expensive effort than implementing a self-made UI control from scratch – of course depending on how much of the existing functionality that is to be kept. The possibilities may be just as good as when starting from scratch – but naturally, the more tailoring the more development work.

In cases where advanced adaptation of existing components is not possible or feasible for the required custom UI controls, **developing custom UI controls** is an option. The benefit of doing this is that it will give full control of the appearance and behaviour of the control. The main disadvantage is the costs involved.

For a user only using hardware keys, the main problem is to find relevant and intuitive support for the available hardware keys (in addition to the generic support for using HW keys built into the UI components). There are no general rules for how to do this, but the easiest solution is often to use the navigation pad for controlling directional movement (mark that this is often already built into the various UI controls). Finding intuitive mechanisms for navigating the UI as such is more

difficult, partly because the buttons have icons on them indicating their main use on the platform. Thus, solutions utilizing specially tailored use of HW buttons for navigating the application may require special training for the users. Also related to this sub problem is the issue of how to implement support for HW buttons. This may be done on the application level (best suited for navigating between different parts of an application), on the window level (best suited for navigating between controls on in a window) or on the UI control level (best suited for UI control level support).

4.16 Problem 2.2.p.2 – Retrieving data from a database without using keyboard

Main problem area: Interaction mechanisms
Problem area: Not using the stylus
Main source for problem: Pilot

4.16.1 Background

This problem is related to the general problem of entering text (see Problem 2.1.p.1 – Mechanisms for entering text), but is much more restricted. The problem addresses database retrieval, which is many cases is done in a "very mobile" context, a user situation where it is difficult to use both a SW and HW keyboard. It is also a special problem as database retrieval usually requires typing.

4.16.2 Problem

As indicated above, database retrieval often includes typing a query (usually expressed as a search string in a search dialog). A simple alternative approach is to show all instances and let the user scroll for the right instance. A related (and usually better) approach is to use filtering instead of search, i.e. instead of finding information that match a certain criterion, all information is initially shown, and the user is given mechanisms for telling which information that should be excluded. Designing an efficient filtering mechanism is difficult, and retrieving all instances (to have a starting point for filtering) may be very resource demanding.

4.16.3 Solution(s)

Finding solutions to this problem is to a large extent about finding ways the user may use to avoid having to type. In this sense it is a special case of Problem 2.1.p.1 – Mechanisms for entering text, and thus many of the solutions presented there may be applied for this problem. But as the problem is more restricted, it is possible to design more specialized solutions for this problem.

One possible solution is to have a set of **stored queries** that the user may choose from. One of the main challenges with this solution is to have the right amount of queries. Too few queries makes it unlikely that a suitable query exist, too many causes finding the right query a challenging task to perform without using keyboard. Also, this solution only works if there is some patterns in which instances that the user wants to identify. Stored queries may both be predefined by developers or super-users, or be specified by the user. The latter case requires a specific UI for defining queries to be stored and/or a function that lets the user save the result of a search operation as a stored query. For a given application, there may be different sets of stored queries, i.e. possibly one set for each of the windows where it is possible to search for occurrences. In this case, a function for invoking stored queries should be available in each of these windows, not as an application level function.

In cases where there are no patterns in which instances the user wants to identify, a solution based on **narrowing possible instances** may be used. By this we mean mechanisms that let the user identify a small set of instances in a few steps of selection. One obvious way of using this solution to identify a person is to have a set of buttons for selecting starting letter, and when e.g. "L-O" is chosen, a similar set of buttons grouping actual interval of names is shown. This is repeated until a sufficient low number of instances to be presented in a list is identified. Doing it this way will make it possible to narrow from 1 million to 25 instances in three clicks if there are 20 buttons.

As mentioned above, a solution based on filtering is not practical to implement if the amount of data is large. A hybrid solution exploiting the UI benefits of filtering with the implementation benefits of search is to **use filtering mechanisms for search**. One example of a filtering

mechanism is a set of check boxes giving the possible values of a small set domain type attribute. Checking/un-checking the boxes narrows or widens the set of instances to be shown in a true filtering solution. The same may be done with intervals for attributes that do not have a small set of values in their domain. For a numeric value, a range slider or two sliders may be used to set the desired interval to view. In a filtering solution, response on the filtering operations should be instant. In a hybrid solution, only the UI mechanisms are used, and there must be an explicit operation to perform a search based on the filter settings. Based on analyses of the stored data, it may even be possible to estimate the number of instances that will be identified by a given filter setting – given that the contents of the database is fairly stable and/or contains comparable data over time.

4.17 Problem 3.1.p.1 – Design that both supports branding, is aesthetic, and utilize screen space optimally

Main problem area: Design at large
Problem area: Guidelines
Main source for problem: Pilot

4.17.1 Background

Most organizations developing applications want to apply their own brand to the products. For service providers, this is important to communicate who is providing the service to the users. For application developers the branding may either be connected to the developer organization or the user organization – which may both be the same – or even a combination, where the main branding is from the developer organization, but that e.g. the logo of the user organization is used in the UI.

Branding may be done using different means. A simple branding mechanism is to include the company logo various places in the UI. A more extensive branding mechanism is to also use colour and other visual elements from the graphical profile of the company in all or most UI controls and backgrounds. This is quite common for web pages, but more unusual in applications. A more subtle branding mechanism is to have a company specific UI design, i.e. a standard way of designing UIs that is specific for the organization, and that is easy to recognize. As this type of branding usually is less visual than a graphical profile, such a brand needs to be established to "work". I.e. it will probably work as a brand after some time, and maybe first for the second and third product following the style. It also needs to be very usable to work as a "positive" brand. Of course, these different means may be combined.

4.17.2 Problem

A commonly acknowledged problem connected to branding of UIs is how to combine branding with following standard look and feel. If the branding includes a graphical profile, the UI controls will evidently look different from the look and feel standard for the platform. This may be considered a usability issue, i.a. that the UI may be considered more difficult to learn.

An important issue in usability is affordance. By affordance, it is meant to which degree the design of an object communicates how it is to be used. Certain visual aspects of UI controls – often added for aesthetic reasons may decrease the affordance. An example of this on the PC platform is the changed appearance of buttons on standard toolbars. The last years these buttons have been changed from having an embossed look to being flat (and given a visual marking when the mouse pointer enters). These "modern" buttons are less intuitive than the traditional ones, but are nicer to look at.

Earlier in these design guidelines, issues concerning utilization of screen space have been discussed. In most cases, branding occupies screen space. This is the case both for added visual elements, and if the various controls are given a special look.

The question of which organization that should be branded in the UI is to a large extent an organizational issue, but a relevant problem to address here is how to combine branding of different companies. The most relevant approach to make this possible is to use different branding means for branding the different organizations.

Independent of the problems above, there is also a problem that any branding of a UI has a development cost. This cost is both connected to designing how the brand should be achieved in the UI, and to implementing the necessary UI components that realize the design. “Implementing” in this setting is often adapting and tailoring existing UI controls (which is fairly inexpensive), but may include building controls from scratch (which may be quite expensive).

4.17.3 Solution(s)

One way of combining branding with standards is to **brand the standard.** By this we mean to add branding elements to the platform standard instead for building the elements that make up a brand from scratch. It is difficult to give general advices of how this may be achieved, but one of the main principles is to use subtle means, like changing background colours or adding a pattern or an abstract image as part of some controls and/or backgrounds. Also, using a specific font together with standard controls may be a good branding mechanism. The main problem with this solution is that the branding may be difficult to recognize, so designing this type of solution may be very challenging. On the other hand, implementing it does not need to be too costly.

Branding the standard will often also solve the problem that branding occupies additional screen space. Keeping the visual look close to the standard usually maintains the screen space requirements from the standard. Generalizing the principle also to cover branding that are further from the standards, **branding the controls** usually will take up less screen space than adding additional purely visual elements (like icons and advanced borders) as the main branding means. In addition to being a challenging design effort, doing more “radical” branding of UI controls may also be quite expensive to implement. Using purely visual elements as branding means may be much less expensive to implement. Also if the principle of not “wasting” screen space on branding, a possible (but not easy) solution is to put a requirement on the design that is shall not require more screen space than the platform look and feel standard.

As mentioned above, the question of which organization to brand is primarily an organizational issue, but in many cases at least the visual branding should normally be connected to the user organization. I.a. this may increase the users’ ownership to an application. This main principle is based on an assumption that the application supports some important mission critical process in the user organization. It is not the case for generic applications like word processors (for such more generic applications, adding a logo etc. for the user organization may be a way of combining brands). One way to combine the branding is to use visual branding of the user organization, and use other design means like layout, icon design, organization of menu items, etc. to brand the developer organization. As mentioned above, this type of branding will only work when users apply more than one application from the given developers organization. The strength of this type of branding can be seen on the desktop platform where the Microsoft way of designing applications (using standard components) has become so well established that most application developers try to imitate it.

Independent of the solutions above, an important principle to follow is that once a branding effort has been chosen, it should be used consequently throughout the application – not just on the main window or the windows that are used most often.

4.18 Problem 3.1.p.2 – Solutions for searching large amounts of data, e.g. multi-step solutions combining reliable identification and optimal screen space utilization (also horizontally)

Main problem area: Design at large
Problem area: Guidelines
Main source for problem: Pilot

4.18.1 Background

This problem may be viewed as an instantiation of various more specific problems above applied for searching in large amounts of data. Searching in large amounts of data is of special importance on mobile platform i.a. because of the identification problem and because it is seldom practical to store all the data on the PDA, so smart client/server solutions are also needed.

4.18.2 Problem

A very common user situation is the need for identifying one or a few of a set of instances. If the amount of data is limited, basing the solution on a list box variant usually works. This is seldom the case if the amount of data is large, partly because it is not practical to store the data on the PDA, and partly because it is not practical to find a particular instance in a list box if the number of instances is very large. Thus some sort of mechanism for reducing the number of instances is needed.

A main choice for this is either searching or filtering (see Problem 2.2.p.1 – Interacting with applications without using stylus and Problem 2.2.p.2 – Retrieving data from a database without using keyboard). When there are large amounts of data, filtering is difficult to realize because it usually requires cashing of data. Thus searching often remain the only alternative, even though filtering in many cases results in a solution that is more usable.

One of the main problems with search is that the user must have fairly good domain knowledge to enter reasonable search criteria, i.e. so that the search doesn't either give an empty answer or a very large number of instances. Also, when large amounts of data are involved, a search may take quite long time, especially if the result set is large (large amounts of result data must be transferred to the PDA). One important issue to ensure easy identification of the right instance(s), is the available search fields, both the number and which fields that are available.

In some applications it is especially important that the right instance is identified, and that the user is aware of this. Usually, this influences the result presentation more than the searching mechanisms.

4.18.3 Solution(s)

In Problem 2.2.p.2 – Retrieving data from a database without using keyboard, we presented a solution using filtering mechanisms for search. This is a good example of a more general solution which we may denote **multi-step search.** By this we mean solutions where a number of steps are used to identify the right instance. Even though this is more time consuming than normal search with a unique identifier, it may be much faster than trial and failure based on empty results or transferring large amounts of data as a result of a too wide search. This solution may either use a set of predefined steps (similar to a wizard based solution) where the user enters different types of data needed for identifying the right instance(s), or use an undefined number of steps using the same mechanism. The name search presented in Problem 2.2.p.2 – Retrieving data from a database without using keyboard is an example of the latter followed by the former, where the

available set of names is narrowed in a number of similar steps until the amount of instances is small enough to transfer to a list box. An example of using predefined steps is a browser type solution where a fixed number of list boxes are used to identify instances based on different attributes. The first list box may e.g. be the region in which the user lives, the second the municipality, and the third the street name. In this solution the lists need to be populated dynamically based on the choice in the prior list. A different variant could be to use attributes that are independent of each other but that each one has a limited set of values that are predefined or transferred to the PDA fast, e.g. type of subscription, region, occupation and hair colour. Although each choice in each list box maybe identifies tens of thousands of instances, the combination may identify just a few.

This problem is based on the assumption that it is not possible to cache all unique values of an identifier in the client. Despite this, some **intelligent caching/pre-fetching** may be possible. By this we mean to use a combination of previous search results and caching values that are stored and pre-fetching possible values based on what the user types. E.g. the application may store the values that the user searches on most often (if such values exist) and use auto-complete if the user starts typing such values; or if the user types a “z” or another letter with few instances, all identifiers starting with this letter may be transferred in parallel with the user’s continued typing. For pre-fetching to be a practical solution, a fast communication channel to the server is a presumption. In cases where it is possible to transfer all instances identified at some stage in a search process, an alternative to auto-complete is to use auto-select in a list box based on a search criterion. This could be a solution where the user starts to type, and after having written a number of characters a list box under the search field is populated with some hundred values. If the user decides to continue typing, the list is not further reduced – instead the first instance starting with the right characters is selected and shown in middle position in the list box. The benefit of this variant compared to auto-complete is that the user can see a number of possible values at the same time, and may even find the right one if the last letter typed was wrong. This solution is similar to how some dictionary applications (both on desktop and PDA platforms) are designed.

One of the sub-problems of searching is that the result set is too large. This may result in a situation where the user must wait very long to have the result presented. To avoid this, there should be some intelligence in the search mechanism so that transferring data in such situations is delayed – e.g. so that the application can issue a warning and let the user decide whether the data should be transferred anyway. A similar more dynamic solution is to show a dialog counting the instances while they are transferred – with a cancel button to abort the transfer operation.

4.19 Problem 3.1.p.3 – Visually coding of entry fields to mark editability (must, may, may not)

Main problem area: Design at large
Problem area: Guidelines
Main source for problem: Pilot

4.19.1 Background

In forms-based UIs, all fields can be categorized according to what the user can/must do with the field. The usual values are must, may and may not. “May not” can be divided into fields that are never relevant to edit (e.g. post office name when the ZIP code is given), and fields that are temporarily not editable (e.g. a field for giving an “other” value when a given set of alternatives is presented). Permanently “may not” is often marked with grey background colour (and with borders to mark that it is not a label). Temporarily “may not” is often marked with grey text but ordinary editable background colour. In some contexts, such text may be difficult to read. Some also choose to hide “may not” fields that are not relevant in a given context. (The distinction between permanent and temporarily “may not” is only available for UI controls including a text entry field).

Visual marking of “may not” field is usually part of a look and feel standard and is supported in the UI controls of a given platform. This is not the case for “must” fields. Still, some find it adequate to have a special marking of these fields. This is more common (though not standardized) in web pages than in GUI applications. If no visual marking is given, users must be notified in an error message if the value is not entered (in fact also if a marking is given).

4.19.2 Problem

Visual coding of “may not” fields is considered a must in UI design. How the fields should be marked, or whether they should be hidden instead is not that obvious. Visual coding of “must” fields is a question of balancing what the user needs to know about the domain and what the user needs to know about the design of the application. Thus, visual marking of “must” fields is most relevant if the users have limited domain knowledge (and are trained in what the visual markings means). Also, a special colour for marking “must” fields may cause the UI to look more untidy.

Given that visual marking of “must” fields is desired, it is not obvious how the marking should be done. See Solution(s) for some possibilities.

4.19.3 Solution(s)

Visual marking of **“may not” fields** is discussed in Background. If fields that are temporarily unavailable are not easy to read, using normal fields combined with e.g. an audio signal if the user tries to tap the field is an option – also switching to a different background colour is a possible alternative. Whether such fields should be hidden instead is not an easy issue to give general advices about. Hiding is an option only in situations where the field is not relevant, and any possible value in the field is not to any benefit for the user. The main benefit of hiding such fields is to avoid visual noise for the users – so that they can focus the important information. The main benefit of disabling the fields is that the UI is more stable, and does not change in ways that may confuse the user. Permanently “may not” fields should always be visible. One problem with the standard visual presentation of this on the PocketPC platform is that such fields (i.e. text entry fields with the property ReadOnly set to true) are very similar to command buttons (text alignment is the only difference). If this confuses the user, different visual coding (e.g. setting the field to disabled instead) could be considered.

Regarding visual marking of “must” fields, several solutions may be used. One option is to use **bold borders** on such fields. This gives an indication that these fields are of special importance. .net compact framework does not support changing border thickness as an available property. Another solution is to have **bold labels** for the “must” fields. This also gives a suggestion that there is something special about the field, but in a more subtle way, as it is not the entry field that is marked, but it may be easier to see the difference between “must” and “may” fields using bold labels than using bold borders. Bold labels are supported in the .net compact framework. A third solution is using an **asterisk with the label**. This is a convention that has been fairly established for web forms – so if the users have web forms experience, they will probably understand this solution. The solution has many of the same characteristics as using bold labels. A forth solution is using a designated **background colour on the entry field**. This is the most “visual” of the solutions, and thus easiest to notice for the user, but also the one that creates most visual noise. To make this solution as “quiet” as possible, a fairly light colour (e.g. light yellow) should be used. Background colour is supported for text entry fields in the .net compact framework.

4.20 Problem 3.1.p.4 – Standard solutions vs. usable tailored solutions – how to choose

Main problem area: Design at large
Problem area: Guidelines
Main source for problem: Pilot

4.20.1 Background

This problem covers at least two different sub-problems. One is to which degree an application should conform to standards, especially when a non-standard solution is considered more usable. The other sub-problem is whether to use a standard solution e.g. for a given trade, or to develop a tailored one.

4.20.2 Problem

Generally for this problem, it is important to know the users. What are their knowledge, experience and how much/which training will they be given. For knowledge and experience, both general ICT knowledge/experience, PDA knowledge/experience and domain knowledge/experience may be relevant.

The sub-problem connected to the degree to which an application should conform to standards is quite wide and varied, and very difficult to give good, general solutions to.

The sub-problem connected to standard solutions vs. tailored ones is more restricted, and is to a large extent a trade-off between costs and having a solution that fits the user needs very well.

4.20.3 Solution(s)

First we consider the sub-problem of conforming to standards or not. Generally, there should be good reasons for going outside the established standard on a platform. If the users have wide experience on the platform that the application runs on, deviating from the standard should be avoided. If the users do not know the platform, deviation may be easier to handle. Another general principle is that there should be a really good reason for not following the platform standard – preferably based on a usability judgement. The solution must be easier to learn, faster to use, more entertaining, or have some other usability gain. It should also be mentioned that implementing a solution that is different from the standard usually is more expensive than a standard one, cf. the problem on branding above (see Problem 3.1.p.1 – Design that both supports branding, is aesthetic, and utilize screen space optimally).

Considering the sub-problem of using a standard application vs. a tailored one, using a standard solution (e.g. for a given trade) is usually less expensive than developing one from scratch, but a tailored one may fit much better to the users tasks and needs. Some standard solutions also offer tailoring facilities, but this is often connected to interfacing the application to a back-end server, etc. more seldom to the issues that are important for the usability of the system. It should also be mentioned that the two approaches may be combined, i.e. to base an application on a standard solution, which is enhanced with newly developed functionality (i.e. more than just tailoring the existing functionality of the standard system). In this way the benefits from both approaches may be exploited.

4.21 Problem 3.2.p.1 – User interaction for/during synchronization solutions

Main problem area: Design at large
Problem area: "Difficult to understand"
Main source for problem: Pilot

4.21.1 Background

In this problem we focus on UIs for synchronization solutions. It should though be mentioned that other parts of a synchronization solution than the UI influence how it is perceived by the user, including how usable it is. Some options are discussed in the section on Connection to server in the chapter on Important choices.

Many will claim that the best UI for a synchronization solution is no UI, i.e. that the synchronization is handled transparently for the user. For some solutions it is indeed possible to realize a transparent synchronization solution, but in many cases it is not possible, mainly for technological and/or economical reasons.

A UI for synchronization may be user or system driven. By user driven it is meant that the synchronization is started and controlled by the user. By system driven it is meant that the synchronization is started by the system, but that the user either is informed, or must take some kind of control during the process.

4.21.2 Problem

If there needs to be a UI for synchronization solution, one of the main problems is to give the user a minimal understanding of what is happening and what the user's role is. This includes the choice of a user or system driven solution. If a user driven solution is chosen, the user needs an understanding of when and how often the synchronization should be done – and it is of prime importance that the application is able to determine and alert the user if a needed synchronization has not been performed, as the consequences of not synchronizing may be severe.

During the synchronization it is not obvious what information that should be given to the user. Usually, it is not possible to do anything meaningful on the PDA while it is synchronizing, but the user will usually want to start using the PDA as soon as the synchronization is finished.

After the synchronization has finished, the user needs to know whether the synchronization was successful, and if it was not, possibly what went wrong, but at least what the user should do to fix the problem. Even a transparent synchronization solution may need UI for handling certain error situations, or at least prevent the user from using functions that rely on a synchronization that has not been performed.

4.21.3 Solution(s)

As discussed above, avoiding having some kind of UI for synchronization is not possible. In most cases, it should be a goal to **minimize the user interaction**. E.g. if an application has or is planned to have a user driven synchronization solution, check that this is indeed needed. For the minimal UI, it is important to make it non-technical and to the point. This means using terms that are connected to the user's tasks – e.g. call the function "collect orders from the main office" rather than "synchronize order database".

During synchronization, the user should primarily need just a little more information than an hourglass. Instead of flashing data being transferred or giving technical information about which

tables that are being synchronized, it is usually sufficient to just **show a progress bar** (possibly supplied with information about estimated time left). If technical information about the synchronization process is needed in communication with the IT department in error situations, this information should be written on a log file, and a command for opening/sending this file should be available somewhere (deep down) in the menu system.

4.22 Problem 3.2.p.2 – User interaction for log-on/log-off

Main problem area: Design at large
Problem area: "Difficult to understand"
Main source for problem: Pilot

4.22.1 Background

During the expert usability evaluations in UMBRA in 2004, surprisingly much time and involvement was given to the evaluation of the log-on screens. This may partly be caused by the fact that the evaluation started with these screens, but also shows that there are certain problems connected it. It is also being considered a harder problem to solve on PDA than on a PC.

4.22.2 Problem

A major problem with log-on on a PDA (especially when it is operated using a stylus) is entering password. Because the keys on the SW keyboard are small, it is easy to miss – and when the feedback is just an asterisk or a black circle, the user does not get to know about the error until the password has been checked. Using strokes makes it even more difficult, as the SW keyboard – as oppose to the strokes interpreter – at least gives visual feedback for a very short time.

When finding more usable alternatives to using the standard input mechanisms, one will very easily run into a trade-off between security and user-friendliness. E.g. adding more explicit user feedback on entered values usually makes it easier for a spectator to find out which password that has been entered. Another problem with tailored solutions is that they usually restrict the number of choices for each character to be entered. This makes it easier to crack the password.

PDA solutions are sometimes used as front-ends towards a number of PC systems. When this is the case, the user may have to enter a number of passwords during a session. Using a common log-on will often compromise security, e.g. because the passwords for the various underlying systems need to be stored.

4.22.3 Solution(s)

As mentioned above, finding good password entering solutions is a trade-off between user-friendliness and security. Thus, the applicability of different solutions depends on the requirements for security.

If the needs for security are low, a solution based on a **large keyboard with few buttons** (e.g. numerical pin codes) should fit well. As an alternative to numbers, iconic symbols may be used. In such a solution, it is advisable to give visual feedback a small second on which button that has been pressed.

If the needs for security are medium, a similar solution to the one just described, only with a **larger number of buttons** should fit well. This means that a pin code solution is not applicable, but iconic symbols may still be used. The same is the case with a reduced keyboard showing only the most commonly used letters. Some kind of feedback on which button that has been pressed should still be available, but if it is visual, it should be as short as possible (but long enough to be observable by the user). Audio feedback is an alternative. An alternative to increasing the number of buttons to increase security is to increase the number of characters in the password.

If the needs for security are high, the most obvious alternative to using the standard keyboard is to have a **full screen keyboard** for entering the password. If the number of buttons becomes large,

using iconic symbols is a less applicable solution, both because it may be difficult to find the right symbols, and because this makes it less natural to supply a shift button to make the number of available choices larger. In such cases, visual feedback outside of the time the user presses the keys should be avoided. Thus, having a feedback field showing some neutral characters becomes very important so the user knows how many character he has entered.

4.23 Problem 3.2.p.3 – User interaction during waiting for long-lasting operations to complete

Main problem area: Design at large
Problem area: "Difficult to understand"
Main source for problem: Pilot

4.23.1 Background

This problem may be viewed as a generalization of parts of the synchronization problem described above. One of the things that make this problem special on a PDA is that the necessary information for showing a reasonable progress indication may not be available on the PDA. A related issue is that it may be more difficult to predict the duration, e.g. when data is transported using a wireless connection with varying bandwidth. Also, obtaining the necessary information for showing progress – and partly also the process of showing it – may require more overhead on a PDA than on a PC. As it usually is impossible, difficult or inconvenient to use other applications than the one performing the long-lasting operation on the PDA, long-lasting operations will probably make PDA users more impatient than PC. This increases the need for good feedback solutions.

4.23.2 Problem

The issues just presented above may make it difficult to use the established mechanisms for showing progress for long-lasting operations. Sometimes it may be a trade-off between giving information and not delaying the long-lasting operation even more. Balancing this trade-off is probably the main problem when trying to find an optimal solution for user feedback during long-lasting operations.

4.23.3 Solution(s)

When faced with a problem like the current where there are no optimal solutions ("damned if you do and damned if you don't"), a possible approach is to try to eliminate the problem instead of solving it. In this case it means trying to **avoid or bypass the long-lasting operation**. There are of course lots of situations where this is not possible, because an operation requires transferring large amounts of data or very much computation. Still, trying to find smart implementations may speed up the application. When there is a need to transfer large amounts of data, a smart caching solution both pre-fetching that the user may start working with before it is asked for, and continuing to transfer data while the user works with it – even with reduced performance – may be a much better solution than waiting for the user. If there is a need for a very demanding computation involving modest amounts of data, doing the computation on a server computer instead of on the PDA may enhance performance significantly. This is of course only possible if the application is on-line.

Given that it indeed is not avoidable that the user has to wait, information may be given on three levels:

1. Inform the user that something is happening. The normal way of doing this is providing a wait cursor, preferably supplied with a message saying "please wait…". This solution is easy and inexpensive to implement, and it does not decrease performance (and thus prolong the waiting) significantly. On the other hand, it is not very informative for the user, and will not help him decide whether he should make a cup of tea while waiting or not.
2. Inform the user that something is happening, and indicate progress. This is normally done with a counter and/or a slider/gauge that shows the percentage of time spent. If it in

addition is possible to estimate in actual time values the time left, this is a benefit for the user. This solution is more costly to implement than a static wait message, and it requires more computation, and will thus decrease performance slightly. On the other hand, it gives the user more useful information. The main problem with the solution is that it is not always possible to implement, as it requires that it is indeed possible to estimate the progress. For some operations, this is not possible – or at best very difficult/giving inaccurate values.

3. Inform the user about what is happing (in addition to indicating progress). This may be done as a scrolling text that the user can browse back in, just a small list showing the latest events or as single text changing as events happen. Independent of how the information is shown, it should be presented in a way that is comprehensible by the user – i.e. related to user concepts and user tasks. Showing detailed information is more costly to implement than the two other solutions, and it may also decrease performance more severely (of course depending on how easy it is to obtain and present the information to present, and how often it is presented). On the other hand, it gives the user more (and hopefully more useful) information. Again it may be a problem to obtain the necessary information to give this kind of feedback.

5 Other issues

5.1 Main differences between UIs on mobile and stationary equipment

In this section we present core differences between user interfaces on mobile and stationary equipment in general, with focus on exploitation of contextual information when designing mobile user interfaces.

Designing user interfaces for mobile equipment is usually considered being problematic compared to designing user interfaces for stationary equipment. There are a number of reasons for this. The screen size is smaller, and the interaction mechanisms are less rich on mobile equipment; this includes both the number of available user interface components and available modalities (e.g. keyboard is not available on many mobile units). The mobile equipment is used in more demanding environments (e.g. challenging light and sound environments), and in user situations where it is difficult to use computer equipment (e.g. because the user is wearing large gloves or because the user's hands are occupied with the current task).

These problems are important and must be handled when designing mobile user interfaces. But we find it even more important to address the *opportunities* that rise from the knowledge that the user is mobile. The context in which a mobile user operates changes much more frequently than it does for a stationary user. An important challenge when designing mobile user interfaces is to exploit knowledge about these changes in the user's context – and to use this knowledge to enhance the user experience.

One of the most evident differences between user interfaces on mobile and stationary equipment is the difference in screen size (still most PDAs usually have screen resolution of ¼ VGA or less). Evidently, this causes a number of restrictions on how much information it is possible to present on a mobile user interface. The consequences of this vary depending on the actual application or service which is to be designed. But generally, an actual dialog must either be simplified, or it must be divided into a number of smaller parts (either accessed through some tab folder type mechanism or as a number of separate dialogs).

Another common consequence of the restricted screen real estate available is that user interfaces on most mobile equipment do not offer the ability to have more than one dialog available at a time. Usually, simple dialog boxes may be presented in separate pop-up windows, but even dialog boxes are presented in "full screen" mode, and are thus usually also system modal. In the same way, "normal windows" may also be shown only one at a time. This gives consequences both for how a single application or service is designed, and for how the user works with more than one application/service in parallel. This is actually quite paradoxical, as the mobile use with its flux of changing context ought to make it easier to switch tasks! The one-dialog-at-a-time principle causes the use of mobile user interfaces to be more linear or sequential compared to stationary user interfaces where it is easier to work with different dialogs and applications in parallel. These restrictions on mobile user interface design make it even more important to design task-based user interfaces. One may say that this causes the mobile user interfaces to be a little more like character-based terminal user interfaces than the stationary ones.

Partly as a consequence of the more linear way of using mobile user interfaces, new principles for structuring the user interface of an application / a service have emerged. E.g. on the PocketPC platform, many applications have adopted a design paradigm where the application initially presents all available data files (often documents) in a list. This list may be filtered to cover e.g. only files in one catalogue, and the user may rename, move and delete the files in the list. Tapping

a file in the list opens it for editing. The open file must be closed to be able to access the file list again. This scheme makes it impossible to work with more than one file (document) at a time, and restricts the needs for a general file manager, as most of file manager functions are available in a tailored way inside the application.

In addition to the screen size, the limited repertoire of available interaction mechanisms is the most evident difference between stationary and mobile equipment. At the first glance, the absence of keyboard and mouse on most equipment is most striking. Some PDAs have a miniature keyboard, or a foldable keyboard as auxiliary equipment, but in these cases the user has to sit down to be able to utilize these – which makes the usage situation less mobile. Some mobile devices have a tracking ball or touch pad as a mouse equivalent, but the far most common solution is to have a pen for selecting etc.

The absence of a physical keyboard makes it necessary to have different means for entering text. Usually, there is a choice between a software keyboard on the screen or a special area for writing single letters using special strokes. Some systems also let the user write whole words at arbitrary positions on the screen.

Either way, using a clip-on miniature keyboard or using strokes or a software keyboard, makes it inconvenient to enter more than small amounts of text. This makes it important to design the mobile application so that it is not necessary to enter large amounts of text.

The most common mouse equivalent is to use a pen. In actual use, the differences between pen based and mouse based interaction are larger than they seem. The traditional difference between select and open on a mouse based user interface has to be implemented differently on a pen based one because fewer event types are available. In mouse-based user interfaces, mouse position is often used to inform the user through cursor changes, tool tips etc. Movement is not recognized in a pen-based interface, so this type of information must be made available through other means. Furthermore, drag and drop is clumsier using pen than using mouse. On the surface, pen based and mouse based interaction may look quite similar. But in actual use, the differences are larger than they seem. Technically, the main difference is the event types offered. While a mouse offers both *mouseDown* and *mouseUp* events and distinguishes between left and right mouse button and *single*- vs. *double-click*, a pen offers only *tap* and *tapAndHold*. This means that e.g. the traditional difference between selection and open on a mouse based user interface must be implemented differently on a pen based one. Also, drag and drop is clumsier using pen than using mouse.

Also, the repertoire of user interface controls on most PDAs is more limited than on stationary platforms, but these differences are less significant than those of screen size and available modalities.

To some extent, the performance difference between mobile and stationary equipment also influences the user interface. But the main consequence of the lack of processor power, battery life, and memory on mobile equipment is that certain types of applications and services are less applicable.

A user choosing to use one or more services while being in a mobile setting usually does this intentionally, i.e. there is a connection between his/her mobility, the tasks s/he is performing and his/her need to access the service(s). Another important characteristic of a user exploiting mobile equipment is that his/her main task is usually not to use a computer, i.e. the main focus is primarily in the physical world, not on the computer screen. The computer's role is then to give *secondary* task support. In many cases this situation is augmented with the fact that the user's hands are occupied by the primary task performance. This introduces a need for using other

modalities than mouse and keyboard as main communication means. Speech interfaces will undoubtedly become important, but it will also be important to have user interfaces utilising multiple modalities at the same time, e.g. giving speech and gesture commands simultaneously ("move *that* object over *there*"). Using audio in a mobile user interface is elaborated in the end of this section.

5.2 Exploiting that the user is mobile

These problems are important and must be handled when designing mobile user interfaces. But we find it even more important to address the *opportunities* that rise from the knowledge that the user is mobile. Compared to a stationary user, the context in which a mobile user operates changes much more rapidly. An important challenge when designing mobile user interfaces is to exploit knowledge about these changes in the user's context – and to use this knowledge to enhance the user experience. The context changes are multidimensional – and sometimes rapid – and comprise position, light, sound, network connectivity, and possibly biometrics. Detecting such changes, and maybe even comparing the traces of them from different users, may be exploited for providing and presenting services in a better way to the user. In mobile informatics there is still no common coherent approach to deal with such issues. Reflecting and utilising contextual information in a mobile user interface is different and more important than in a stationary one. This causes the need for new types of user interface designs, covering the contextual aspects, and ways for the UI to adapt to changing contexts. This reveals a need for something more than just down-scaled versions of desktop-UIs, which is the case for many applications and services available on PDAs and other types of mobile, general purpose computers today.

Which types of changes that should be applied to user interfaces when the context changes, may vary. Different contextual parameters are used to adapt *which* information to present and *how* it is presented. E.g. in a context aware airport service, the main features of the service could be adapted to whether the user is departing, arriving or in transfer. Within each of these main modes, details may be adapted. E.g. information about check in counters should only be shown before the user passes the security control. If the user is handicapped, different toilets should be shown on the map of the airport than if the user is not. Information about tax free shopping should only be shown for international travellers. On a lower level of granularity, it is relevant to let different aspects the icons representing PoIs (e.g. restaurants) change depending on properties of the restaurant and available information – like number of stars, type of restaurant, price category, whether the menu is available electronically through the service, etc.

Furthermore, applications and services may exploit context for different purposes. On one hand there is a need to augment existing applications and services with context information, so that they may support the user in a better way than before. On the other hand, it is increasingly important to develop *new* types of services and applications and services that are enabled or driven by available contextual information. By this we mean applications and services that it usually does not make sense to offer to a stationary user, mainly because the context does not change very much, e.g. presenting a you-are-now-here dot on a map to a user sitting in the office.

Looking specifically at map-based user interfaces, experiments with applications have shown that the most striking differences are between the user interface for finding information. A stationary UI needs much focus on the filtering mechanisms as well as much functionality for doing geographical filtering (zooming and panning) – functionality that is not needed in a mobile context aware variant. This difference – the need for filtering information geographically and thematically in the stationary version and the mobile version with most focus on information close to the user's current position (and thus with limited or no need for filtering mechanisms) – reflects one of the most important differences between using map-based (and other type of) applications in a stationary and mobile context. Although it would not be easy to implement filtering mechanisms

on the small screen, it is our claim that the mechanisms are not needed because the user's position constitutes so much filtering that the thematic filtering becomes superfluous. That is to say that in the mobile case the user's position is the context, whereas the context has to be "created" in the stationary case.

User interfaces for entering information (typically points of interest (PoI)) tend to be different too, but in this case more on the amount and type of information than how it is handled. While a stationary UI requires quite detailed information, with much focus on thematic and semantic information, a mobile UI usually require less information. This difference has a strong connection to the different ways the two variants are used to browse information. For flexible filtering mechanisms in a stationary UI to be effective, the relevant semantic information must be entered for each PoI. It is also our claim that this is the *main* reason for adding this type of information in a structural way. Once an annotation is identified, accessing it might as well be through a textual description. Of course, a mobile user could generally benefit from having more semantic information about each PoI, and a user adding a PoI may well include this type of information as part of the textual description or the audio part.

Mobile and stationary users also tend to use media differently. In a stationary UI, all information may well be alphanumeric, while a mobile one has larger benefits from applying pictures and sound. Experiences with mobile contextual UIs show that the multimedia information is very valuable, especially the audible part of a PoI. In fact it has turned out that when using a specific mobile application, walking around on-site, sound is *the* most important part. The reason for this is partly that it is the least intrusive part for a mobile user, and partly because sound facilitates hands-free use when walking around. When using the application in a different location than where the annotations were added, the picture and text is much more important. The user benefits from having pictures as part of a PoI in a stationary UI also, but sound would in many cases likely be more annoying than useful. When sitting down, most users read faster than people talk, so having to wait for a person to say something that the user might as well read is often frustrating. For a mobile user the situation is quite different. To be able to read, the user has to stop, while listening is possible when s/he is walking anyway. This argumentation holds as long as the sound contains equivalent information to the alphanumeric. In both UI types, it would anyway be valuable to use sound for other purposes. E.g. to add a sound sample to a concert PoI is very relevant. Also for pictures, the utilization is different depending on how the picture relates to the PoI. The considerations above about pictures in on-site and off-site use of a mobile UI hold as long as the pictures show the position of the annotation. E.g. having a picture taken on the position of the annotation some time in the past may be very relevant also for a mobile user.

The rapid context changes in a mobile setting cause the need for flexible user interfaces that are multitasking and possibly exploiting multiple modalities. This need is poorly met by standard PDA interfaces focusing on one task at a time (because of the limited screen size). An alternative is to have extremely context adaptive user interfaces – but this is probably not possible to construct. To solve this trade-off between having small and light equipment and having access to different information and services in a flexible way (both are needs that are caused by the user being mobile) – there is a need for new interaction mechanisms. One way of approaching this challenge is to use alternative modalities, like audio and speech. Audio output is briefly discussed above. Speech input is a modality that fits very well to many of the challenges facing a mobile user context. The main problem with using speech on mobile computers is limitations in the equipment. Handling speech recognition on stationary computers is challenging, but using it to control various programs in calm environments works fairly well. Dictation requires a restricted application domain or extensive training. In a mobile user context, both commands and dictation is relevant. But as the mobile units are less powerful than stationary ones, few mobile units are able to run very powerful speech recognition software. Furthermore, the mobile user context is

often not calm, rather it tends to be noisy, a fact that causes the speech recognition task to be more demanding. To handle the lacking processing and memory capabilities of the mobile unit, server based speech recognition may be used. I.e. instead of doing the speech recognition on the mobile unit, the spoken input is sent to a server using wireless communication. The response is sent back to the mobile computer. This solution requires a stable and reliable wireless connection with sufficient bandwidth to work.