RWTH AACHEN UNIVERSITY

Selexels:

Adapting User Interfaces to Mobile Input Devices

Master Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Mahsa Jenabi

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Wolfgang Prinz

Registration date: April 06th, 2006
Submission date: September 28th, 2006

Contents

Abstract ix

Acknowledgements xi

Conventions xiii

1 Introduction 1

1.1 Background 1

1.2 Motivation 4

2 Theory 7

2.1 Expressiveness of Input Devices 7

2.2 Selexels 10

2.3 Layout Manager 11

2.3.1 What is a Layout Manager 12

2.3.2 Without Layout Managers 12

2.3.3 Custom Layout Managers 12

2.4 Glass Pane 13

3 Related work **15**

3.1 Automatic Generation of UI 15

3.1.1 ICrafter 15

3.1.2 PUC 17

3.1.3 SUPPLE 19

3.1.4 Comparison With Selexels 20

3.2 Area Cursor 23

3.2.1 Comparison With Selexels 25

3.3 Interaction with Large Public Displays with Mobile Devices 25

3.3.1 Comparison With Selexels 26

4 Design **29**

4.1 Goals 29

4.2 Selexels Constraints 30

4.3 3-Layered Architecture of the Selexel Framework 30

4.3.1 Layer 1: Selexel Transparent Layer (Selexel Glass Pane) 30

4.3.2 Layer 2: Original User Interface 31

4.3.3 Layer 3: Selexel Layout Manager . . . 32

4.4 Diversity of Layout Managers 32

5 Implementation **37**

5.1 Implementation components 37

5.1.1 SelexelGlassPane 37

5.1.2 SelexelListener 38

5.1.3 SelexelLayout 38

5.1.4 SelexelFlowLayout 40

5.1.5 SelexelGridLayout 40

5.1.6 SelexelJPanel 42

5.2 Implementation Challenges 42

5.2.1 Original Location 44

5.2.2 Minimum Lines of Code for Adaptation 45

5.2.3 The Layout Manager of unit Selexel Containers (SelexelJPanel) 45

5.2.4 Layout Manager or Layout Manager 2 46

5.3 How to Adapt an already existing UI 46

5.4 How to write a new Selexel-based UI 47

5.5 How to implement a custom Layout Manager 48

5.6 How to implement your own Selexel-based Layout Manager 50

6 Design Process 53

6.1 Prototype 1: Fitts' Law 53

6.2 Prototype 2: Selexel-based SUPPLE Toolkit . 57

6.2.1 Why SUPPLE Toolkit 57

6.2.2 Challenges 58

6.3 Prototype 3: Custom Layout Manager and Transparent Layer 62

6.3.1 Challenges 63

6.4 Prototype 4: Selexel Layout Manager hierarchy 64

7 Evaluation 69

7.1 Experiment 1 69

7.1.1 Introduction 69

7.1.2 Experiment Design 70

7.1.3 Results 72

7.1.4 Discussion 73

7.2 Experiment 2 74

7.2.1 Introduction 74

7.2.2 Experiment Design 75

7.2.3 Results 75

8 Summary and future work 77

8.1 Summary and contributions 77

8.2 Future work 78

A Selexel Golden Rules 81

B Hello World Swing Code 83

C User Study 87

Bibliography **91**

Index **95**

Abstract

Selexels is a conceptual framework that guides a user interface (UI) designer to adapt a UI to input devices with low expressiveness: devices that have limited ability to convey intended meaning. By adjusting the interface, the user can enjoy a fluid and smooth interaction, without suffering from the technical weaknesses of the input device, such as low sampling rate or low resolution. We have evaluated this framework through some user studies using Fitts' Law.

As a proof of concept, we have implemented Selexel Toolkit. Selexel Toolkit can help UI programmers in two ways. The first way is when a programmer has already created a UI, working properly with the mouse and keyboard, but he wants to change the UI code, so that the same UI can be used with an input device, having low expressiveness, such as a mobile phone. In such a case, the Selexel Toolkit adapts the original UI to match the expressiveness of the input device. The second way is to give programmers a new Layout Manager, in addition to java standard Layout Managers, i.e., a new tool for laying out their UI in a different way with getting advantage of layout constraints that can also be useful in other application areas.

Selexel Toolkit performs these tasks with its 3-layered Architecture. The three layers are: Selexel Transparent Layer, UI, and Selexel Layout Manager.

Selexel Transparent Layer puts a Transparent Layer (Glass Pane) on top of the UI, in order to handle the Mouse input for the Selexel cursor. Additionally, It replaces the standard cursor with the rectangular-shaped Selexel cursor.

Selexel Layout Manager layouts the UI with respect to the low expressiveness constraints, in order to adapt the UI to the connected input device. This adaptation can be done dynamically, while changing the input device.

Acknowledgements

I have successfully finished this Master thesis, but not to forget, with the kind support of my lovely colleagues and friends and family.

I want to take the chance and say many thanks to Professor Jan Borchers, my first examiner, and Professor Wolfgang Prinz, my second examiner, for giving me a scientific support with my thesis.

Special thanks to Tico ballagas, who is a PhD candidate at InformatikX department and my supervisor for this thesis. He has patiently helped me through the whole improvement of the thesis with useful Tips.

Many thanks to my colleagues: Thorsten karrer for the nice Latex template he has prepared for the Master thesis, the research assistants and research students that helped me for reviewing the thesis and giving tips for programming and writing this thesis.

My lovely family that have supported me emotionally all through my thesis, in happy times and stressful days.

My kind friends who gave me hope and self-confidance to keep working on my thesis.

Thank you!

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in Canadian English.

Download links are set off in coloured boxes.

File: myFile[a]

[a] http://media.informatik.rwth-aachen.de/~ACCOUNT/thesis/folder/file_number.file

Chapter 1

Introduction

1.1 Background

Ubiquitous Computing

As Ubiquitous Computing (UbiComp) moves the computational resources beyond the desktop by blending them "into the fabric of our everyday lives" [Weiser, 1991], the need for new novel interaction techniques is inevitable. People use computers not just at home or at office, as in old days, but also on the go, or in the public areas such as bus stations, international exhibitions and airports. Traditional input devices, like mouse and keyboard can not be efficient for these post-desktop applications, because for example in on-the-go scenarios the portability of the devices is necessary (i.e., the user should be able to carry the device around) and the wireless capabilities of the device are important. In emergency scenarios like accidents, it is vital for the ambulance driver to get the location information of the patient quick on the way to the accident environment. In this scenario the driver is not at his office or at home to use his desktop device, instead he can use his car navigation system. Another alternative for the driver is to use his mobile phone to see the location on a map and also get the urgent location-based information, as [Hosokawa et al., 2005] has implemented. In order to study these new coming-up ubiquity challenges, researchers are designing and developing new devices. Different areas of applications have their own constraints, which need to be

considered, while designing new interaction techniques. As an example in public areas, speech modality performs poorly, because of the background noise of the passengers. The question is if one can develop some standard equivalents to the traditional mouse and keyboard for UbiComp environments.

Mobile phones

Mobile phones are one of the familiar and powerful devices that can be used for interaction in UbiComp environments. Mobile phones are the first truly pervasive computers that people use in everyday life. Calling friends, sending messages, managing personal data, or even playing games are the common things people do with their mobile phone. Mobile phones are with us almost everywhere, almost all the time. They also have numerous built-in sensors such as cameras and microphones. With these rich capabilities, mobile phones are powerful input devices that can be used to interact and control our environment, [Haro et al., 2005].

Mobile phones' limitations

Although people enjoy the portability advantages of the mobile phones, they may find their small screen inconvenient to work with. The small screen of mobile devices make them not so effective for some applications; for example, the web has become available on the mobile phones, but revealing the whole content of web pages on the small screen of mobile devices is an open question. Although some researchers, like [Baudisch et al., 2004], have studied the possible approaches, like interactively removing irrelevant content, to achieve it, there are still remaining problems while displaying detailed images or large UI components on the tiny screens [Roto et al., 2006]. One solution to this can be using a Large Public Display (LPD) for displaying the information in combination with the mobile phone.

Large Public Displays

LPDs are becoming more commonplace in public areas, like airports and bus stations. To date, LPDs are addressed to be a medium for displaying information to groups of people, not typically being interactive. However, interactions have been demonstrated that make use of LPDs for different application scenarios, such as BlueBoard inGoos et al. [2002]. BlueBoard is intended for two kinds of per-

sonal and in-group usages. Fast personal use is possible, while the user is walking up, check his calendar, and walk away. In addition, small groups can also use BlueBoard for sketching ideas, sharing information, or comparing notes. Making the LPDs interactive have the advantage of giving the right information, in the right time, to the right person. People can browse and search for the information they need.

Large Public Displays challenges

A challenge related to LPDs is finding a proper input device to interact with them. Therefore combining LPDs and mobile phones together can solve many problems that each of them has, when it is used separately. Sanneblad and Holmquist [2006] has introduced a novel interaction technique using a mobile device in combination of a large display. This technique is introduced for solving the problem

Figure 1.1: Sanneblad06: A user holds up a tablet PC in front of a large projected image to view details

of displaying a large map or detailed images. The large display is responsible for showing the large image. For viewing the detailed image of a special part, the user needs to hold a mobile device, such as a tablet PC in front of the place of his interest and the mobile device shows that part in higher resolution.
Of course such kinds of solutions can only be used in the public areas and Interactive Rooms that a LPD is available.

Sweep technique

In addition to Sannebald's work, there are other going research projects that are working on using a mobile device, like a mobile phone instead of a mouse to control the cursor on the screen. Ballagas et al. [2005], have introduced an interaction technique, called Sweep technique. Sweep technique allows a mobile phone with a built-in camera to

Figure 1.2: The sweep technique can be used to control a cursor on the large public display like an optical mouse

be used as an input device, instead of optical mouse during interacting with a LPD. For moving the cursor, the user should wave the mobile phone in the air and the built-in camera at the other side of the phone takes sequence pictures and by comparing these images can specify the relative motion with three degrees of freedom. With these capabilities, Sweep technique enables many direct manipulation interactions, such as cursor control, with LPDs.The connection between the LPD and the mobile phone is done per Bluetooth.

1.2 Motivation

Low expressiveness input devices

Some of the input devices have low expressiveness: their ability to convey intended meaning is limited (see low ex-

pressiveness devices in the next chapter). Therefore the expressiveness of the input device is lower than the resolution of the display. Consequently the users are not able to specify a pixel or a small region on the screen.The prototyped cell phone, used in sweep technique, is an example of a low expressiveness input device, Ballagas et al. [2005].

Eyes as input device

Another example for such kinds of devices is eyes as an input device. In the eye tracking research field, different possibilities of using eye movements, in order to interact with applications have been tested. Eyes are like a build-in input device in human's body. For using them in UbiComp environments users doesn't need to carry an extra input device with them. besides being intuitive, the direction of the eye gaze can reveal the users' attention in most cases.

Eye Tracking challenges

Although using Eyes as an input can be beneficial according to the mentioned features, using eyes instead of a mouse in order to control the cursor and select a UI component is challenging. Because eyes can fixate on a special target not more than about 600 milliseconds, Jacob [1994]. During a fixation eyes cannot be stable at one point, but instead they make small movements, called micro-saccades, Duchowski [2003]. These constraints of eyes make the accurate measurement of the gaze point mostly impossible. As mentioned in Isokoski [2000] small targets are harder to hit with the gaze and require careful calibration between the eye-tracker and the target; therefore an optimum UI for such a low expressiveness input device has bigger widgets, in order to make the selection task easier.

Matching the UI to input device

Our solution for this problem is to match the selection space of the application to the expressiveness of the input device. Without reengineering GUIs, performing the intended actions can be hard, frustrating or sometimes impossible for the users.

Dynamic adaptation of the UI to the current Input device

Another problem that emerges from the combination of mobile input devices and LPDs is that: different people are entering the interactive spaces, using their hand-held devices (e.g. a PDA or a mobile phone) combined with existing devices (e.g., LPD), and then leave; therefore, the LPD, which is located permanently in the space needs to connect and interact with all these different hand-held de-

vices, the users carry with. On one hand mobile devices have different sensors, and different computational characteristics, i.e., different sampling rates and sampling resolutions, on the other hand the applications running on the LPDs show always the same User Interface. In order to give all users the chance of having equally smooth interaction experience, the Graphical User Interface(GUI) of the LPD needs to be adapted to the current connected input device dynamically.

Chapter 2

Theory

This section includes the theoretical basis of this master thesis that understanding it is necessary, in order to understand the thesis.

2.1 Expressiveness of Input Devices

Input devices

As [Card, 1989] explained, Human-Computer interaction is different from Human-Human interaction in this case that: the interaction between human and a computer needs an artificial language, since human and computer are not from the same type; therefore he has modeled the interaction between human and computer as an interaction between at least three agents:

- a human
- a user dialogue machine
- an application

The human's input is taken, from an input device, and is mapped to the application's understandable events. As [Baecker and Buxton, 1987] have observed:

" basically, an input device is a transducer from the physical properties of the world into logical parameters of an application."

Based on this, [Card et al., 1991] has defined the key idea of "Primitive Movement Vocabulary" in modeling the language of Human-Computer Interaction, as follows:

Formally, input device is presented as a six-tuple,
(M, In, S, R, Out, W),
where
— M is a manipulation operator,
— In is the input domain,
— S is the current state of the device,
— R is a resolution function mapping from the input domain set to the output domain set,
— Out is the output domain set, and
— W is a general-purpose set of device properties that describe additional aspects of how a device works (perhaps using production systems).

Taxonomy for Input Devices

This model can be used as a taxonomy (i.e., classification) for input devices. Researchers have been working on defining taxonomies for input devices, in order to make an abstract specification of the device. Taxonomies make a clear and well-structured definition of the device in mind; and therefore make the evaluation of the devices easier. One can specify which features are important for a especial application and then compare the available devices just according to those features of interest.

Another good criteria for input device evaluation is expressiveness. [Card et al., 1991] has introduced the expressiveness for input devices, which shows how well the input device is able to convey the intended meaning. The problem with Expressiveness happens when the displaying preciseness mismatches the preciseness of the input device. For example, in a touch screen interaction, the user can select a widget on the screen by using his finger. What will happen if the selectable items, like buttons, are so small and placed so close together that the user is not able to select just one of them without hitting the other widgets in the neighborhood? In this example,

finger, as user's input device, is not able to express the intended meaning of the user; therefore the user has a hard time to interact with the system.
[Card et al., 1991] have described the expressiveness problem as following:

" In the design of input devices, an expressiveness problem arises when the number of elements in the *Out* set does not match the number of elements in the *In* set to which it is connected. If the projection of the *Out* set includes elements that are not in the *In* set, the user can specify illegal values; and if the *In* set includes values that are not in the projection, the user cannot specify legal values. "

In and *Out* sets are the parameters explained in the Primitive Movement Vocabulary (explained earlier). In our above example, *Out* set comes from the display resolution and *In* set from the input device (finger), which will select the total number of pixels below the finger, depending on the finger's size. A possible strategy here to solve the problem can be to adapt the selectable UI widgets' sizes big enough for the finger or at least enough distanced, so that a user with a big finger does not select more than one item inadvertently at the same time.

According to the definition of Expressiveness for input devices:

LOW EXPRESSIVENESS INPUT DEVICE:

An input device has low expressiveness, when it's ability to convey the intended meaning is low.

Definition: *Low Expressiveness Input Device*

As mentioned in the Introduction chapter, mobile phones, finger in touch screen, Eyes as an input device are some examples of low expressiveness input devices.

2.2 Selexels

As explained in the motivation section, the problem with using low expressiveness input devices can be solved by matching the selection resolution of the UI to the expressiveness of input device; keeping in mind that the UI resolution is independent from the display resolution. For example in the touch screen scenario in the previous section, the problem of selecting UI objects on the screen will be solved if the layout of the UI components respects the input device (finger) constraints; It would be enough if the UI components be laid out far enough from each other, so that the user do not activate other widgets inadvertently.

This goal is achieved by dividing the screen into atomic selectable elements (selexels) with a resolution that is independent of the pixel resolution of the screen. In

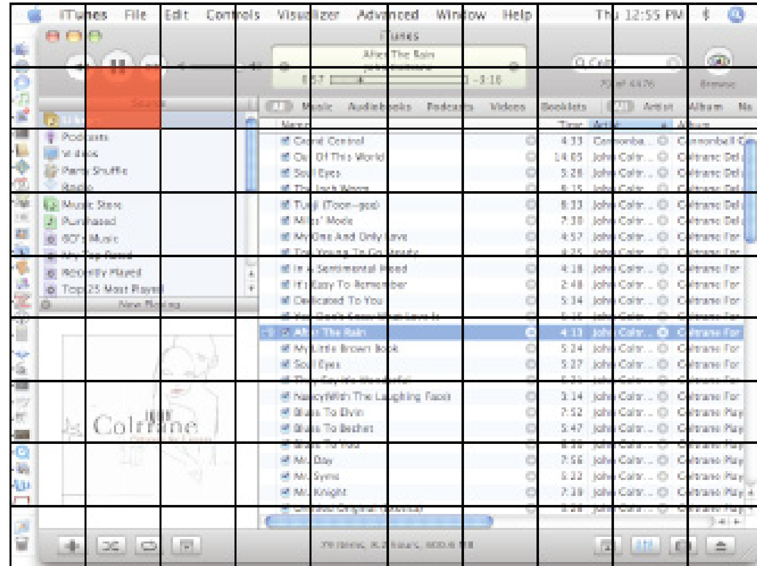

Figure 2.1: A sample selexels screen division over a typical desktop interface. It indicates that existing desktop interfaces may need to be modified to make selection with low precision pointing devices unambiguous.

other words the selectable units of the Application will change according to the maximum preciseness of the input device to select a target. The shape of Selexels is

rectangular and its size is computed according to the sampling rate and sampling resolution of the input device.

SAMPLING RATE OF INPUT DEVICES:

Sampling Rate = Number of Samples / Time (Sec)

Definition: *Sampling Rate of Input Devices*

SAMPLING RESOLUTION OF INPUT DEVICES:

Sampling Resolution = Information / Sample

Definition: *Sampling Resolution of Input Devices*

By separating selexels from pixels, we limit the range of motion in the interface to support a smooth and fluid user experience for the input device in use, while preserving the screen resolution and information capacity of the display. Selexels serves as a design tool to match the expressiveness of graphical user interfaces to input devices with low expressiveness. Using Selexel conceptual framework, we would describe a traditional desktop interface as a special case where the selection space is identical to the display space. Expressiveness, characterized by selexels (unitless), describes precision in selexel space; how many distinct positions, in selection space, can one express using this input device. Adaptation of the UI to selexel constraints seems inevitable.

2.3 Layout Manager

In this section, an introduction to Layout Managers is given. We will discuss why a Layout Manager is needed and how it can help the programmer to layout his UI. For more information about implementation details see the Implementation chapter. The information about Layout Managers in Java is taken from [Walrath et al., 2004].

2.3.1 What is a Layout Manager

A Layout Manager in Java is responsible for positioning and resizing the UI components inside their containers. It asks how much space each component needs. The size setting and locating for each component is done after checking the available size on the screen and also according to the especial policy the Layout Manager has. Sizing and placing policy is different for different Layout Managers, since they are designed for conceptually different applications.

2.3.2 Without Layout Managers

The question is if we can ignore Layout Managers. It is possible not to use Layout Managers, but it is usually better to use them in most applications. A Layout Manager helps adjusting to changeable font sizes and container sizes, e.g. while resizing the application windows. Adjusting to the especial Look and Feel is also easier if the UI components are not independent of it. From software-engineering-view, the reusability feature of Layout Managers are valuable, since they can be used in several applications.

Without a Layout Manager the programmer needs to do the precise positioning and sizing of all the components; it can be several lines of code.

2.3.3 Custom Layout Managers

There are 7 standard Layout Managers, included in java Swing Libraries. For laying out our GUI we can use one of these Layout Managers or use combinations of them. But what if these standard Layout Managers can not satisfy the application requirements? In such a situation programmers need to implement their own Layout Manager that respects their specific constraints. In our project for implementing the Selexel framework we have implemented our own Layout Manager, Selexel Layout Manager. For more informa-

tion about the concept behind the Selexel Layout Manager see Design chapter. You can find more information about how to implement a custom Layout Manager in Implementation chapter.

2.4 Glass Pane

In this chapter I will explain what Glass Pane in java is and how one conceptually can benefit from it. The implementation issues about it are explained in the Implementation chapter. More information about Glass Panes in Java can be found in [Walrath et al., 2004].

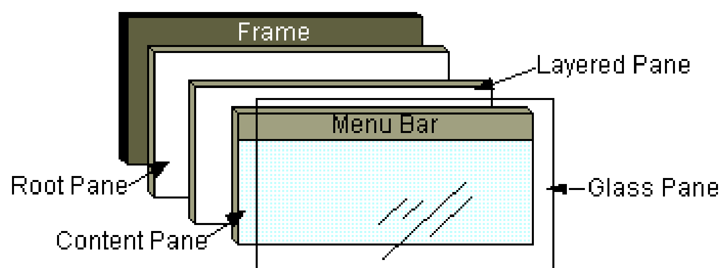

Figure 2.2: Different layers in java GUI (sun Tutorial)

Glass Pane

The Glass Pane in java is like a transparent layer coming on top of your created UI. It is invisible unless you render to it. Its Look and Feel in UI is exactly like its name, glass. Glass enables you to see whatever is behind it, without any visual obstacles, but it doesn't let you touch the objects behind it. If you touch it, you get to the glass surface. The Glass Pane is the same. You can see the UI components, drawn in behind layers, but when you want to click or manipulate the widgets nothing will happen since you are "touching the glass "; the Glass Pane blocks all the mouse input events unless you handle them specifically in your Glass Pane implementation.

Applications

The possible application for such a layer can be in the cases that you want to get the mouse input events and handle them differently, or in the case that the processor is busy with the user's request; you may want to change the shape

of the cursor to a busy- mode and block all the mouse events coming in. It can also be used in the situations that you already have some components drawn on the screen but you want to draw something on top of them.

We have used this concept in our project in order to change the shape and the motion progress of the cursor and also for handling the incoming mouse events different from standard Java. For achieving this goal we have implemented this concept as "Selexel Glass Pane". For getting more conceptual information about it see the Design chapter. About the detailed implementation issues take a look at Implementation chapter.

Chapter 3

Related work

"If I have seen further it is by standing on the shoulders of giants."

—Isaac Newton

3.1 Automatic Generation of UI

3.1.1 ICrafter

ICrafter[1] is a Service Framework for Ubiquitous Computing Environments, developed at Stanford University, [Ponnekanti et al., 2001].

In interactive workspaces, people enter the room, while bringing their handheld devices (such as mobile phones, Laptops, PDAs, etc.). These spaces are technology-rich, and include I/O devices (such as large wall mounted displays, microphones, speakers, etc.). People entering these workspaces usually intend to cooperate with others in a collaborative work (such as design reviews and brainstorming). The requirement for having an intuitive interaction in such workspaces is to have an intelligent infrastructure that supports heterogeneity of different appliances and

[1] http://graphics.stanford.edu/papers/icrafter_ubicomp01/

computers connecting all together. ICrafter is designed for interactive workspaces. It allows developers to deploy services and to create user interfaces to these services for various user appliances. By service they refer to a device (such as a light, projector, or a scanner), or an application (such as a web browser or Microsoft PowerPoint running on a large display) that provides useful functions to end-users. ICrafter facilitates users with flexible interaction with the services in their environment.

ICrafter gives three main contributions:

- It brings intelligence to the infrastructure, by supporting selection, generation, and adaptation of service UIs. This feature enables a better handling of resource-limited appliances.
- It enables on-the-fly aggregation of services.
- Created UIs are portable across workspaces, besides reflecting the context of the current space.

ICrafter has considered two types of adaptations, namely: appliance adaptation, and workspace adaptation. Workspaces are different according to the sets of devices they contain, and physical geometries they have.
The relevant part for Selexel project is the appliance adaptation part. Besides supporting different modalities, ICrafter supports the different appliances according to the resource criteria. ICrafter's solution for the resource-limited appliances is to add the component *Interface Manager* (IM), in addition to the service and appliance components, see figure 3.1. IM handles the UI selection, adaptation, and generation issues. The UI is selected based on the appliance and the number of services.

For workspace heterogeneity, ICrafter has a separated component, called *Context Memory*, in order to save the context information. As you can see in the picture 3.2 generators produce a UI with having the appliance description and getting the context information of the workspace.

The drawback ICrafter has is that the UI generators are

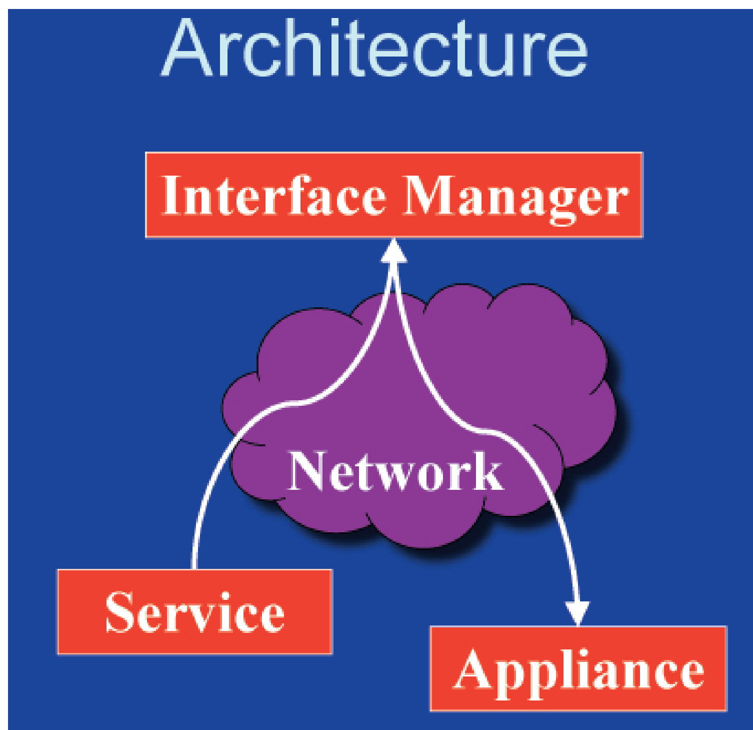

Figure 3.1: ICrafter Architecture. Appliances request UIs from the Interface Manager while supplying an appliance description. The Interface Manager first selects appropriate UI generators based on the requesting appliance and the services for which the UI was requested. Next, it executes the generators with access to the service descriptions, appliance description, and the context to generate the UI, [Ponnekanti et al., 2001].

hand-designed, i.e., ICrafter is using hand-crafted templates for generating UIs.

3.1.2 PUC

This project [2]has been done at Carnegie Mellon University, and studies about using a Hand-Held as a Personal Universal Controller (PUC). At home or office there are several devices (appliances, such as light switches, TV or stereo equipment) that people are interacting with everyday. Each of them comes usually with a remote controller.

[2]`http://www.pebbles.hcii.cmu.edu/puc/`

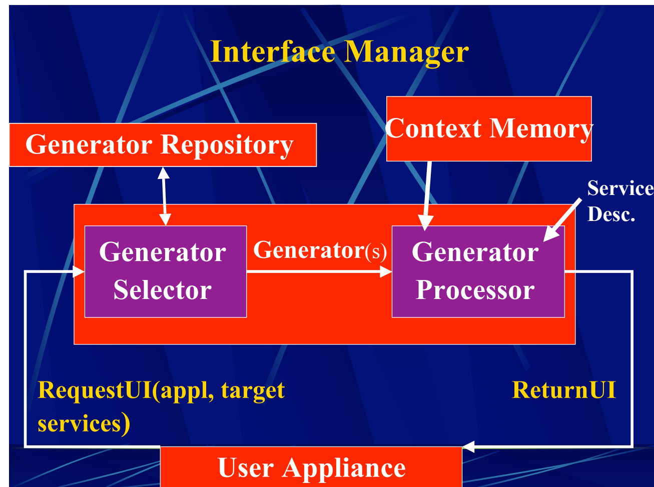

Figure 3.2: Interface Manager

Each of them supports its own functionalities and complexities. PUC is like a universal remote controller that can communicate with all these appliances, considering their supported functionalities. It achieves this goal by generating intermediate graphical or speech interfaces for each appliance the user want to interact with. After this connection is done, user's manipulation commands will go from the PUC to the appliance with using the two direction connection they have with each other.

The System uses a two way communication protocol [Nichols et al., 2002] . For translating from proprietary appliance protocols to the PUC protocol, the system includes adaptors. For describing the appliance functions, a specification language is used, with which the generators automatically can build interfaces. The language specification uses *dependency information*, which describes the availability of each function relative to the appliance's state. Dependency information is useful, since it allows the interface to provide feedback to the user about the availability of a function, such as "graying out" a button in a graphical interface, besides helping the interface generators to organize functions.

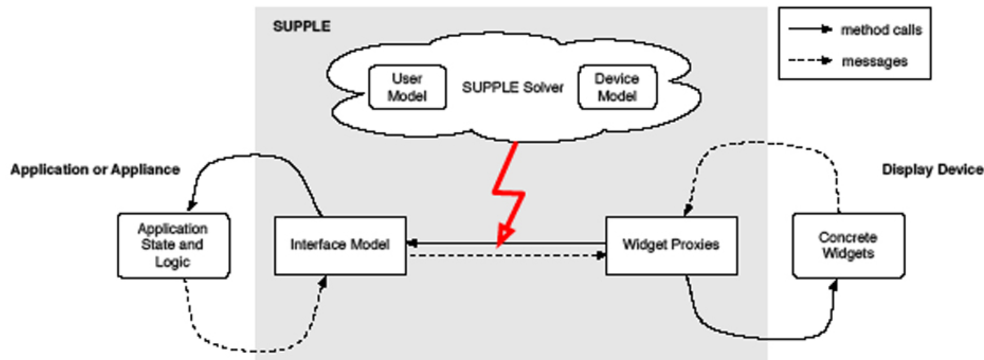

Figure 3.3: Supple's implementation: The interface model exposes the state variables and methods that should become accessible through the interface. The widget proxies generated by the device model are assigned to interface model elements by Supple's optimization algorithm.

Drawbacks

The drawback of PUC is that "Its rule-based rendering algorithms relies on specific domain knowledge and makes it inflexible even to the changes in the screen size of the device it runs on." Gajos et al. [2005]. Therefore PUC has made "some rough assumptions about the screen size and can not deal with situations that the most desirable rendition of the interface does not fit in the available area.", [Gajos and Weld, 2004].

3.1.3 SUPPLE

SUPPLE is a toolkit[3] for automatic generation of User Interfaces for ubiquitous applications, developed at University of Washington [Gajos et al., 2005]. The goal is to generate UIs automatically with considering the device constraints and also user's trace information.

Unlike other mentioned related work, which use templates or rule-based approaches to generate UIs, SUPPLE uses decision-theoretic, combinatorial optimization. SUPPLE enumerates all the possible layouts for a UI; after that it prunes some branches of the solution tree according to the device constraints.

[3] http://www.cs.washington.edu/ai/supple/

The rule-based approaches require hand-designed rule sets, but the cost function of SUPPLE can be quickly created according to the designer's responses to examples of concrete rendering of different interfaces. The optimization algorithm make it robust against screen size changes, which was a drawback for the previous mentioned systems. In order to generate concrete UIs, SUPPLE uses three inputs:

- User model, which includes trace information of user's activities . This model is independent of device and rendering.
- Functional specifications of the UI, which defines the types of data transfered between the user and the application.
- Device model (the relevant component to Selexels project) describes the supported widgets for each device and provides a *cost function*, which estimates the user's effort for manipulating these widgets with interaction methods supported by the device.

Figure 3.4 shows an example of functional specification, which is represented graphically for a stereo controller. The rendered GUI of this figure is shown in figure 3.5

One of the device constraints SUPPLE consider is the available screen size. In figure 6.4, you see different UI renditions for different devices. Besides different devices, also for the same device, resizing the windows will make SUPPLE rendering different UIs, as the rendition algorithm is considering the available size for each UI widget, see figure 3.6

3.1.4 Comparison With Selexels

All these Systems mentioned before are automatically generating UIs for Ubiquitous applications. Selexels is not a framework for UI generation, but for UI adaptation. As you have seen, other systems also have considered some adaptation criteria for their automatic UI generation, but these

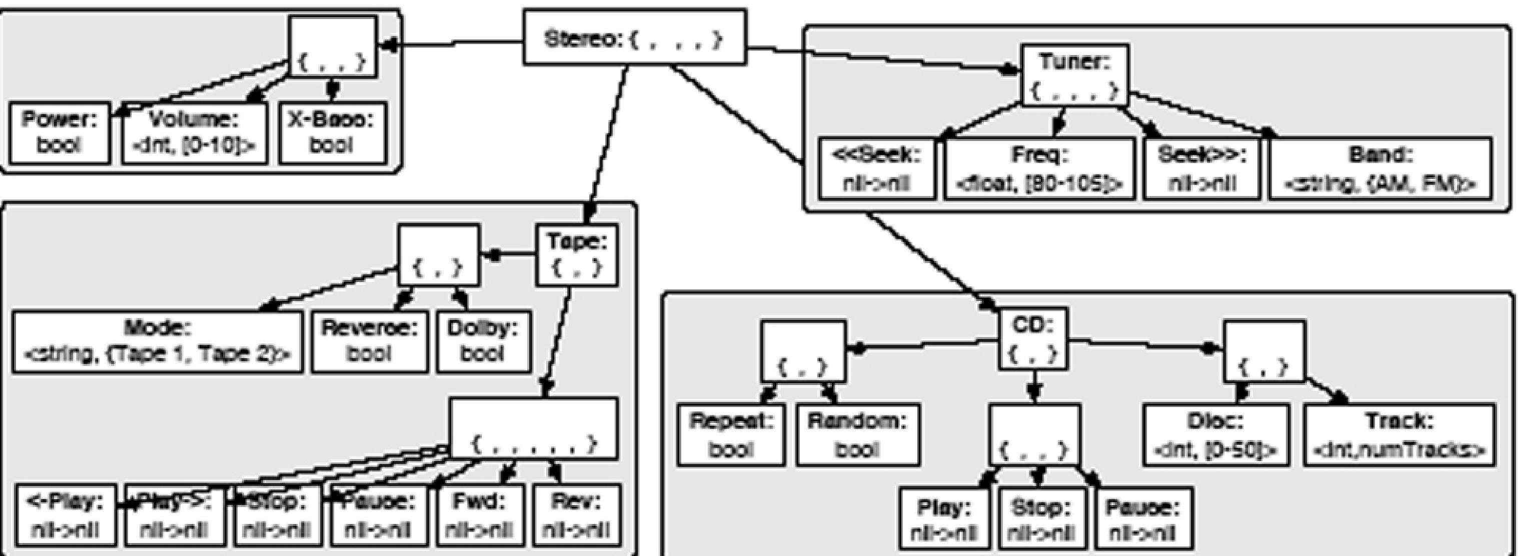

Figure 3.4: Graphical representation of the functional specification for a stereo controller. For clarity, different parts of the specification are grouped with gray shading.

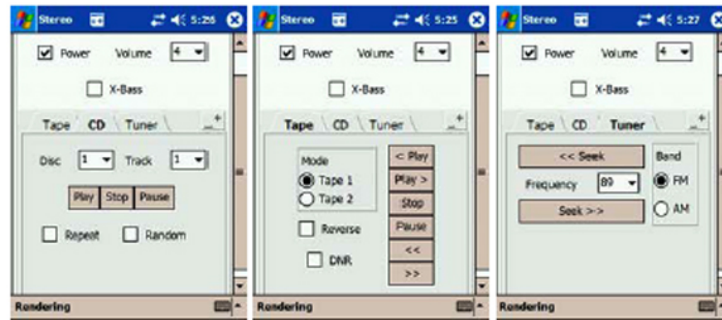

Figure 3.5: Three tab views of the stereo specification, rendered for a PDA.

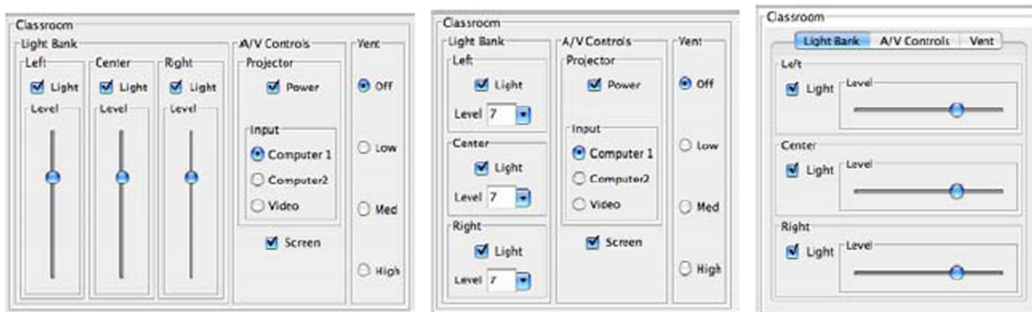

Figure 3.6: Supple optimally uses the available space and robustly degrades the quality of the rendered interface if presented with a device with a smaller screen size. This figure shows three renderings of a classroom controller on three devices with progressively narrower screens.

considerations are some rough ideas about the screen size (ICrafer and PUC), or in SUPPLE there are some considerations tin the case of Touch Panel device that the selectable widgets need to be big enough so that the user can click on them with his finger, but the constraint considerations remain just at this level. It is not adapting exactly according to the sampling rate and sampling resolution of the input device. In Selexels approach, we compute exactly how precise the input device can be and then according to that we compute the Selexel size and adapt the whole UI according to it.

3.2 Area Cursor

In standard GUIs usually a mouse-cursor is point-sized and for selecting a target on the screen this point cursor must go over the area of the target. Such target acquisition situations can be modeled with Fitts' law. The *area cursor* has an active selection region that spans a screen area, instead of a single point. Kabbash and Buxton [1995] showed that if we

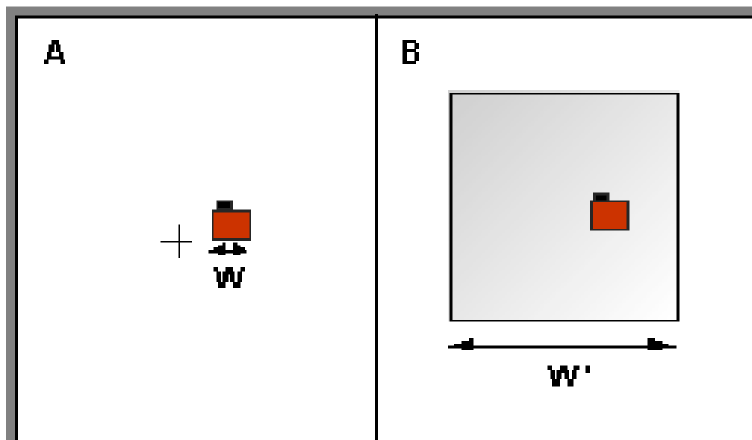

Figure 3.7: In (a) the target is selected using a standard "cross-hair" cursor. The difficulty of the task is limited by the size of the target (W). In (b), an area cursor with width W' surrounds the target to select it. The difficulty of this task is a function of W' .

have a point-sized target and want to reach it with an area cursor with width W, Fitts' law is still valid, see figure 3.7.

Area cursor faces an ambiguity problem when the the cursor area overlaps multiple targets.

Worden's *area cursor*: point +area cursor

A possible solution to this problem is proposed by Worden et al. [1997]. As you can see in figure 3.8 for solving this problem the area cursor also has a hotspot at its center. In the case of ambiguity this hotspot decides which target is under focus. This area cursor performs better than a standard one when the targets are far enough from each other, so that the area cursor doesn't overlap multiple targets. Of course in the case of targets standing close to each other the user still needs to use the centered hotspot and the performance will be equal to the standard cursor. Bubble cursor[Grossman and Balakrishnan, 2005] is another solution

Bubble cursor

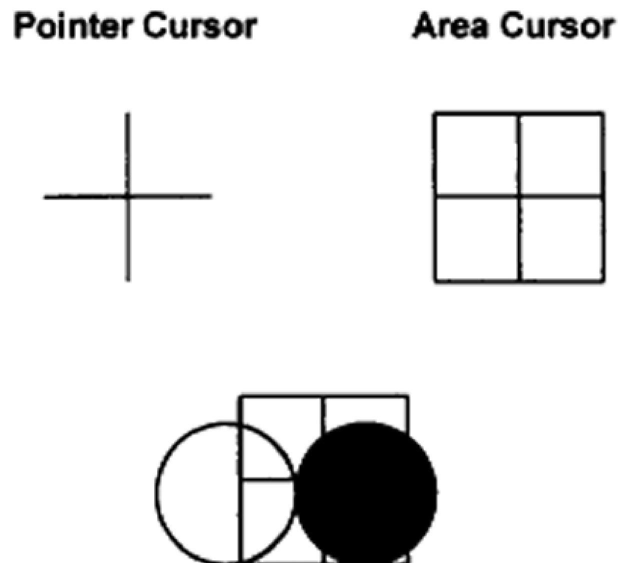

Figure 3.8: Two cursor types and activation pattern of area cursor with adaptive hot-spot when located over two icons. The selected icon is shown in reverse video.

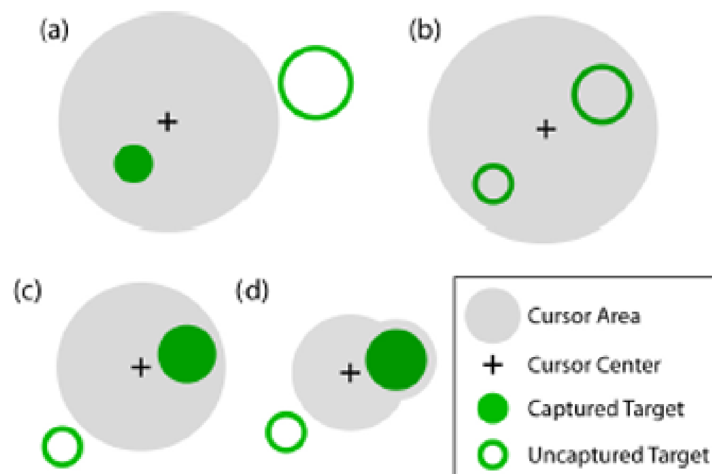

Figure 3.9: (a) Area cursors ease selection with larger hotspots than point cursors. (b) Isolating the intended target is difficult when the area cursor encompasses multiple possible targets. (c) The bubble cursor solves the problem in (b) by changing its size dynamically such that only the target closest to the cursor centre is selected. (d) The bubble cursor morphs to encompass a target when the basic circular cursor cannot completely do so without intersecting a neighboring target.

for the area cursor. The width of Bubble cursor is dynamically resizable. For solving the ambiguity problem Bubble reduces its size up to the level that it includes just one selectable object, see figure 3.9. This technique performs better than the previous solution in the case that the targets are closely packed.

3.2.1 Comparison With Selexels

All the cursor area solutions mentioned above are considering a wider area for the cursor, rather than one point, but the cursor motion still remains pixel-based. This pixel-based motion make them inappropriate for the low expressiveness input devices.
A selexel is a point in the selection space, i.e., the width of the cursor and the motion is based on the Selexel size. For making the GUI environment usable with the low expressiveness devices, we basically want to have bigger pixels, i.e., Selexels, which are accessible to the input device. When a pixel grows to Selexel, all the measurable units of the cursor environment from its size to its motion needs to be in Selexels unit (not in pixel unit anymore).

3.3 Interaction with Large Public Displays with Mobile Devices

The C-Blink [Miyaoku et al., 2004] system is one of the examples for these projects that uses a colored-screen mobile phone as the cursor on an LPD. The cursor is traced by a camera mounted at the top of LPD. If the user comes in an Interactive room and want to interact with such kind of LPD, he just needs to run an application on the mobile phone, which changes the hue of the mobile phone's screen. The displayed hue sequence encodes an ID and consequently interaction of multiple users is supported.

For controlling the cursor, the taken pictures by the camera will be traced for the signal and the result would be considered as an absolute position.

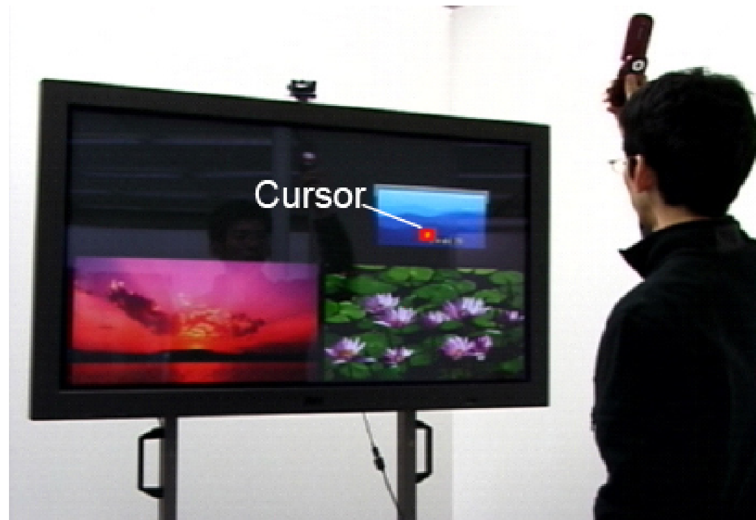

Figure 3.10: The system detects a C-Blink signal, and with increasing the number of large public displays in public areas and with increasing the number of mobile device consumers, researchers are thinking about the possible interaction methods between these two popular groups of high-tech devices.performs a process indicated by the signal.

3.3.1 Comparison With Selexels

C-Blink is also studying about using a low expressiveness input device(here a mobile phone) with LPDs. But the point of C-Blink research is based on having some novel interaction techniques in order to make using mobile devices with LPDs possible, since they are both becoming popular. C-Blink is considering special UIs that the targets are big enough for disambiguating the selection with a low expressiveness input device. They have used UIs that doesn't have this problem, and just focused on the novel interaction technique. Selexels framework is focusing on how the UI needs to be adaptive in the case that the input device has low expressiveness. A specific techniques for interaction between LPD and input mobile device is not under our focus in Selexels project.

In our scenarios and also user studies, we have assumed that the interaction technique is the Sweep technique, which is explained in the introduction chapter. Although

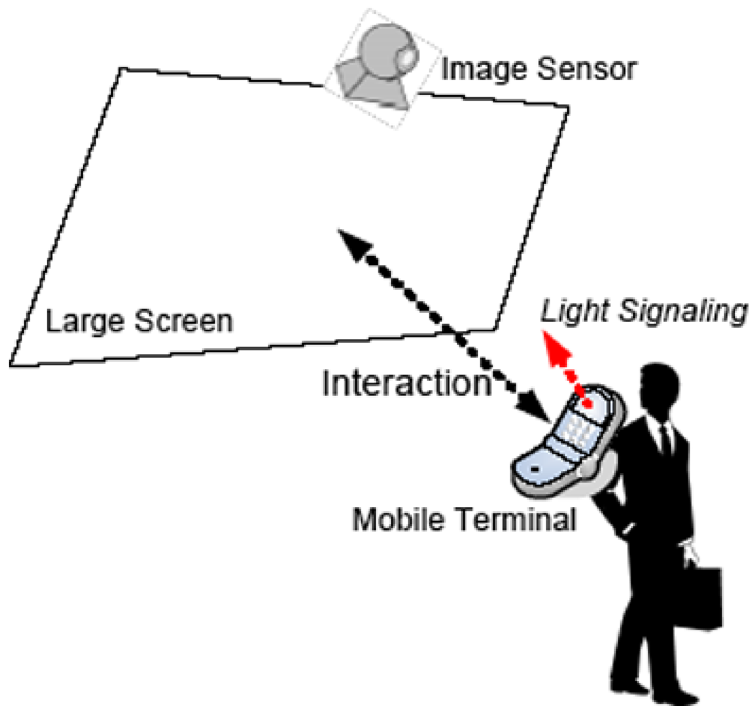

Figure 3.11: Large Screen Interaction with a mobile terminal by using a light signal marker

the Selexel concept is independent of the selected interaction technique.

Chapter 4

Design

In this section, I explain about the way we have designed our Selexels framework. The reasons that we have taken this approach are explained in the Design Process chapter. There you can find the earlier approaches we have taken and the challenges we had.

4.1 Goals

Given an already-created UI, which can work properly with high expressiveness input devices, such as mouse, our task is to adapt it according to the constraints of a low-expressiveness input device, such as a mobile phone. We need to compute how precise the input device is and according to it adapt the UI. Therefore we need to get the technical features of the input device (i.e, sampling rate and sampling resolution), compute the maximum Selexels size this device affords, and then adapt the UI according to the computed Selexels size.

4.2 Selexels Constraints

We have three general requirements for the Selexel-adapted UI:

- No more than one selectable item can be in the same Selexel. Otherwise when the user click on this Selexel, it is ambiguous, which of these selectable widgets must be activated.
- Selexel cursor needs to have a different shape from the standard cursor. It must be a rectangular shaped cursor with the size of one Selexel.
- The mouse input process is also different, since the standard cursor has just one hotspot, which is the exact pixel (i.e., point) it can specify. But for the Selexel cursor the hotspot extends to the whole rectangular region, which equals the whole number of pixels under the rectangle. Mouse click in the standard metaphor means clicking on one pixel, but in the case of Selexel cursor, it means clicking all the pixels lying under the Selexel rectangle.

For handling the first constraint we have implemented the Selexel Layout Manager. For the other two tasks we have implemented the Selexel Glass Pane. In general, we have divided the adaptation process to 3 layers, which is explained more in the following section.

4.3 3-Layered Architecture of the Selexel Framework

4.3.1 Layer 1: Selexel Transparent Layer (Selexel Glass Pane)

This layer is responsible for drawing the Selexel cursor, and processing the mouse input with the help of the program Selexel Listener. This Layer is like a Glass Layer comes on

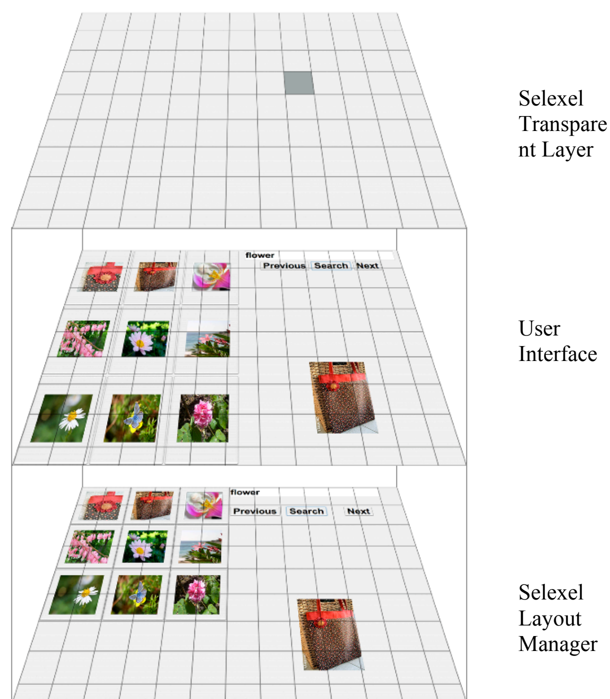

Figure 4.1: 3-Layered Selexels Framework Architecture

top of the UI and replaces the standard cursor with the Selexel (Rectangle-shaped) cursor and blocking the up coming mouse input and managing it according to the Selexel concept.

4.3.2 Layer 2: Original User Interface

This layer represents the User Interface and its components. This UI can be created using any kinds of java Standard Layout Managers or programmer's custom Layout Man-

ager. UI components can be any kinds of Java Swing components (other than the ones which have scroll functionality, such as a scroll bar or a scroll pane container. More information about exceptions can be found in Conclusion chapter, part "What doesn't work").

4.3.3 Layer 3: Selexel Layout Manager

This layer represents a custom Layout Manager that adapts the User Interface according to the Selexel limitations. It takes the original sizes and Locations of the UI components on the screen and change the locations if needed. It computes the Grid boundaries according to the Selexel Size, attained from the mobile input device per Bluetooth, and try to add the UI components of the second Layer one by one considering the Selexel Grid Constraints, which aligns the UI component along the Grid without allowing a Selexel to include more than one selectable widget. This layer tries to change the UI of the second layer as little as possible, i.e., the location of the components are changed just in the case that they conflict the Selexel constraints, mentioned in the previous section. If a UI component can not be placed in its original location, the Layout Manager needs to place it in another place that is free. The policy of placing the components in the case of overlapping is different for different Selexel Layout Managers we have implemented. The policy of each of these Layout Managers is explained in details in the following section.

4.4 Diversity of Layout Managers

Java Swing library has 7 common standard Layout Managers, which the programmers can use just one or a combination of different ones. For more information about Layout Managers see the Theory chapter.
These standard Layout Managers are Flow-, Border-, Card-, Grid-, GridBag-, Box-, and Spring Layout.

Each of these Layout Managers can be useful in some sce-

narios and maybe not so efficient for some other applications. The important thing for programmers is to understand the concept behind these Layout Managers and use them in the right place.
As mentioned in the previous section, each Layout Manager has its own policy for placing the components. For example, the FlowLayout is a good choice when you want to display some components compact in the same row, keeping their natural size. The GridLayout is useful in the scenarios that you want to display some components with the same size in rows and columns. You can find tips for choosing the right Layout Manager in [Walrath et al., 2004].

Selexel Layout Manager also needs to pay attention to these differences that Layout Managers have. Therefore,

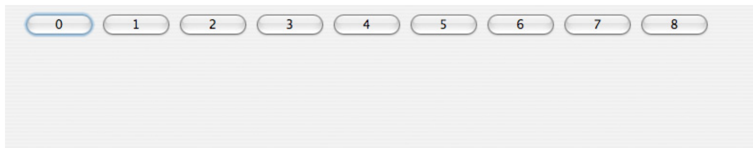

Figure 4.2: Original FlowLayout UI

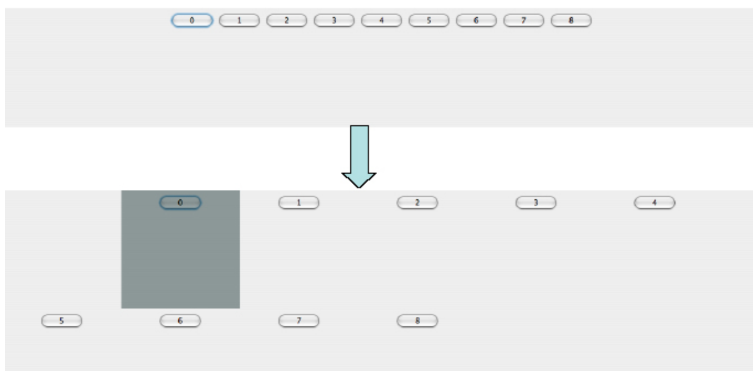

Figure 4.3: Selexel Flow Layout, taking the same policy as the Java FlowLayout, besides considering the Selexel constraints.

it checks which kind of Layout Manager the Container has used originally, and then take the same policy for placing the components, besides respecting the Selexel constraints. With this approach the purpose of the programmer of creating the UI can be highly preserved. As you can see in figure

4.3, the buttons' layout has been changed after adaptation. It may be expected that in this case the button number 0 get more to the left so that the buttons on the second row could still stay in the first row. That would be the technique the standard FlowLayout would do if you make the buttons bigger. This technique was not our case since we try to place the buttons, as the first priority, in the original location. As far as the component can be placed in the original location, we will not change its location. Further more the location of the components are changed just once, and will not be changed after relocating the next componnets. This approach keep the components as much as it can in their original location. More discussion about semantic preserving can be found in the Design Process chapter.

Selexel Layout Manager has a general policy for the UI containers that the programmer has set no Layout Manager, or an unknown/custom Layout Manager to the container.

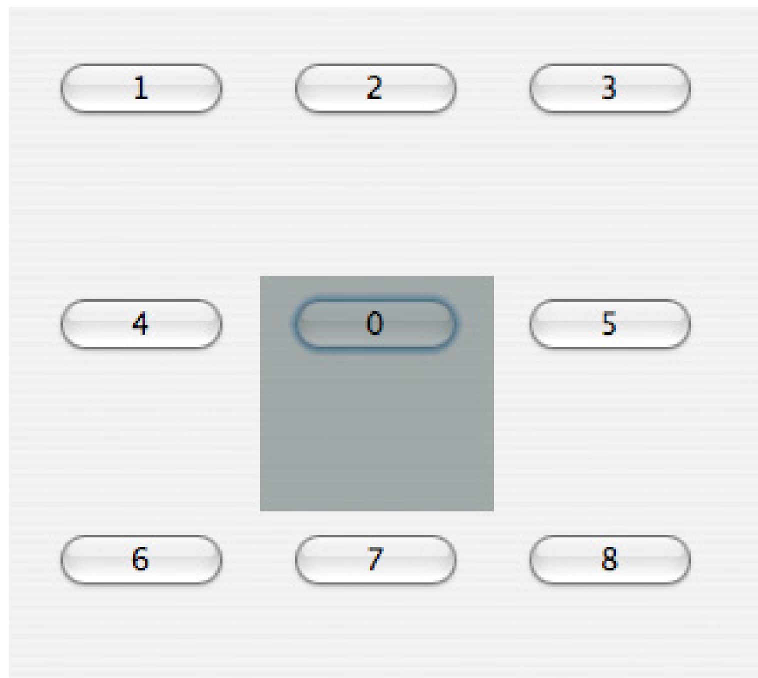

Figure 4.4: All 9 buttons have the same location as the button number 0; since after placing button 0 other buttons can not be placed there they are placed in the nearest neighborhood of the original location.

The policy here is to put the UI component in his nearest free neighborhood. Each Selexel Grid has 8 neighbors (the same as 8 different squares we have in a chess game plate). If none of these 8 neighbors are free the search for a free space will continue to the further level of neighborhood, which is 16 places this time.

Chapter 5

Implementation

In this chapter, the challenges we had while implementing the Selexel Toolkit are explained, and the reasoning behind the approaches we have taken is dicussed. After that the code a programmer needs to add to his code in order to adapt his created UI to the input device is shown. At the end comes a description of what a programmer needs to pay attention to if he wants to implement a Selexel-based Layout Manager.

5.1 Implementation components

In this section, I will introduce the different classes implemented. As explained in the Design part, we have a three-layered architecture for the Selexel framework, see figure 4.1.

5.1.1 SelexelGlassPane

This class is implementing a custom Glass Pane, which is responsible for drawing the Selexel cursor. It also receives the mouse input events and computes the corresponding Selexel, which includes the mouse point, and pass this information to the SelexelListener. This class is in the top

most layer (Selexel Transparent Layer) in figure 4.1.

5.1.2 SelexelListener

SelexelListener extends MouseInputAdapter. It overrides the actions regarding to the different mouse events, such as mouseMoved, mouseClicked, mouseDragged, etc.This class is called in the top most layer (Selexel Transparent Layer) in figure 4.1.

Its algorithm is as follows:

First it gets the cursor point, then it gets the top most component under that point. After that it computes the relative location of that component.

If the component is a:

- Selectable widget: then fire its action.
- Non-selectable widget: get its container and fire the actions of the all child components, only one of which is selectable.
- Container: fire the actions of all the child components

Since we are sure inside each SelexelJPanel there can be maximum one selectable widget, therefore maximum one widget can be fired on each mouse click.

5.1.3 SelexelLayout

This class is a custom Layout Manager, which has implemented the *LayoutManager2* interface. It is responsible for laying out the UI components with respecting the Selexel rules, mentioned in appendix 1.This class is in the third layer (Selexel Layout Manager) in figure 4.1.

The algorithm of SelexelLayout is as follows:

First of all we force the original Layout Manager, which can be a standard Layout Manager or a custom Layout Manager, to lay out the container, i.e., setting the sizes and locations of the container and all its child-components. With this step we know how the UI would look like originally, i.e., how the programmer wanted the UI to look like. The algorithm tries to put the components as close as possible to their original locations on the screen.

For all of the children of this container do:

Compute the number of Selexels they need, according to their original size.

If the child is a:

- Selectable widget: then search to find a location for it on the screen. This location needs to be free of any selectable component.
- Non-selectable widget: then search to find a location which is free of any Container.
- Container: then search for a location that is empty on the screen for this child.
  convert this child-container to a Selexels-based container (set its Layout Manager to SelexelLayoutManager)

As you see in the algorithm when a child is a container itself, its location will be specified and then its type is converted to the Selexel Layout type. This conversion function automatically converts the whole containers and subcontainers till it gets to the *Selexel unit container*. More information about this especial unit container is given in the SelexelJPanel section.

The difference in the implementation of this class and the SelexelFlowLayout and SelexelGridLayout is that: the way the algorithm finds a free location for a component is different. All these algorithms try to put the components in its original location as the first priority. If the original location is occupied, then these classes have different strategies

for finding a free space. The technique that SelexelLayout uses is the nearest neighborhood algorithm. It locates the components in the nearest free neighbor location. More information about this algorithm can be found in the *Design* section.

5.1.4 SelexelFlowLayout

This is a subclass of SelexelLayout. It is acting like the SelexelLayout, but just use another policy to add the components inside the cotainer. Instead of using the nearest neighborhood policy, SelexelFlowLayout adds the new components at the end of the row at the right side next to the previously added component, exactly like the FlowLayout in Java. Therefore, A free location is searched in the same row. If there were no empty space then the next row will be tested. This class is in the third layer (Selexel Layout Manager) in figure4.1. More information about this algorithm can be found in the *Design* section.

5.1.5 SelexelGridLayout

This is a subclass of SelexelLayout. It is acting like the SelexelLayout, but just use another policy to add the components inside the cotainer. Instead of using the nearest neighborhood policy, SelexelGridLayout computes the original row and column index of each component and try to put the components in the same index. But in this case the grid size is changed to the nearest coefficient of the Selexel size. If some of the components doesn't have space to be added, they will just be ignored (i.e., they will be set to invisible). This class is in the third layer (Selexel Layout Manager) in figure 4.1.

The algorithm is as follows:

The same as SelexelLayout, we force the original Layout Manager, which can be a standard Layout Manager or a custom Layout Manager, to lay out the container, i.e., setting the sizes and locations of the container and all its child-

components. With this step we know how the UI would look like originally, i.e., how the programmer wanted the UI to look like.

For all of the children of this container do:

- Compute the number of Selexels they need, according to their original size.
- Compute the location index of the child in the original layout, which means computing the grid index that includes this child; for example when a button is in the first row and second column the location index would be (1,2).
- Compute the Selexel grid, which is always an integer coefficient of the Selexels size, and is the nearest largest coefficient of the Selexels size.

If the child is a

- Selectable widget: then compute the new grid index for this component, which means the component remains in the same row and column, but just the exact location will be changed according to the Selexel grid. The reason behind this technique is to make sure that the component is remaining in the same index (i.e., row and column), since in the standard GridLayout usually the index location of the components is important.
- Non-selectable widget: If the container has more than one selectable widget in general, then the non-selectable widgets also need to be placed in the new grids. In the case that the total number of the selectable widgets is less than two, the layout will remain the same as it was originally, since the Selexels constraints are about selectable widgets, i.e., the non-selectable widgets are allowed to share Selexels.
- Container: then compute the new location through the grid index of the container.

convert this child-container to a Selexels-based container (set its Layout Manager to SelexelLayoutManager)

More information about this algorithm can be found in the *Design* section.

5.1.6 SelexelJPanel

This class extends the standard JPanel class in the Java Swing Toolkit. It represents the unit Selexel containers that the components are added to them. Such a special container class is needed in order to distinguish between the Java standard containers and the unit Selexel container.
This class is used in both first and third layers in figure 4.1.

5.2 Implementation Challenges

Goals

Our goal was to change the layout of the UI in such a way that it obeys the Selexels constraints. In this process we try to preserve the components' locations and sizes, and in the case that they need to be changed we try to resize and replace the components as close as possible to the original situation. Another aim was to make it easier to adapt an already existing UI to the Selexels rules, so that a UI can be adapted with adding the minimum number of code lines to the original program. Further more, it should be easy to adapt an already existing UI program, with having little knowledge about its implementation techniques; the advantage in this case would be when we intend to adapt a complex UI application which is written by someone else.

The implementation is done in Java language. The AWT and Swing libraries have been used. As mentioned in the Design Process chapter, Layout Managers are a good choice for forcing specific components' size and location settings in Java. The important plus point of the Layout Managers is their reusability. One needs to implement them once and can use them as often as he wants. There are some standard

Layout Managers in Java that programmers use their combination, in order to achieve their favorite layout.
In our project, we couldn't use any of the standard Layout Managers, since none of them have the exact policy we have in mind. The desired Layout Manager must ensure the Selexel rules explained in the Design Process chapter and also in appendix 2.

For creating a new Layout Manager there are two possible methods:

- subclassing a standard Layout Manager that its policy is close to our new Layout Manager.
- Writing our own custom Layout Manager.

Why not subclassing the standard Layouts

The first approach was not proper for our case, since the changes we needed to make to the already existing Layout Manager was so much, so that it was not worth anymore.

Why not GridBagLayout

For example the GridBagLayout seemed to be a good choice. GridBagLayout puts the components inside rows and columns. the cell size is specified according to the preferred size of the components added to the container. A component can occupy more than one cell, which is called its *display area*. The problem with this Layout Manager is that it is doing opposite tasks that we want to do, i.e., it gets the preferred size of the UI objects and then at the end decides how big each cell should be. But we set the size of each cell first and set the size of the widgets accordingly. This paradox of the size setting make the GridBagLayout a difficult layout to adapt to the selexel paradigm.
With all these drawbacks, still we tried to subclass the GridBagLayout and this approach failed. The reason was that we needed to override the function *getLayoutDimensions()* of the GridBagLayout to have the same width and height for all the cells and this size equals to the selexel size, but unexpectedly *GridBagLayoutInfo*, which includes the information regarding to the number of cells horizontally and vertically, is just visible to the GridBagLayout itself and not to any subclasses of it. Therefore we couldn't make the changes we wanted even by overriding the important functions.

Why not GridLayout

Another example is the GridLayout. There, the size of all the cells are the same and are set to the preferred size of the largest width and height among the components added. Such a cell size does not necessarily equals the Selexel size on one hand, and on the other hand, it must allow the components to occupy more than one cell. These policies make the GridLayout not a desirable choice for sub classing.

Although by subclassing these standard Layout Managers one can easily prevent them to do something undesirable, by overriding their functions, but overriding all these functionality make us thinking, why we must sub class a class which has so many contradictory features. Why not implementing our own Layout Manager instead; there we can be sure that the super class is not doing something unacceptable and is not changing our set values in between.

5.2.1 Original Location

Our goal is to keep the size and the location of the UI widgets as close as possible to their original size and location. A challenge that we have here is how to know where the original locations are. During the run time although the Layout Manager is the original one in the beginning of the program, but at the end we set the Layout Manager to the SelexelLayoutManager. When SLM get the control of the program, of course the UI is not created on the screen yet; consequently the locations of the components on the screen has not been set by the original Layout Manager.

The *doLayout()* trick

Our solution to this problem was to call the *doLayout()* function of the Layout Manager. This function forces the original Layout Manager to compute the sizes and locations of the UI components inside their containers, but it will not draw it on the screen. After calling this function all the components' sizes and locations are set. Therefore if you call the function *getLocation()* it will return the original location of the component; but this call would return just 0 before calling the *doLayout()* function. With this trick SLM can get the locations information in its constructor before calculating the Selexel grid.

5.2.2 Minimum Lines of Code for Adaptation

We wanted to make the coding task as minimum as possible. Imagine this scenario that Bob has implemented his UI, which is working properly with the high expressiveness input devices. He has recently heard about the Selexel concept and wants to adapt his UI according to the connected low expressiveness input device. He should be able to do this adaptation in his code as easy as possible. Perhaps he can not remember anymore how exactly he has implemented his UI. Therefore it should be possible to adapt the UI even though one doesn't have detailed information about what is done where. One of the approaches we have taken, in order to achieve this goal is that Bob just needs to set the Layout Manager of the Content Pane (Mother Container) of his UI to SLM, without changing all the Layout Managers used inside the UI to SLM. In this case SLM goes recursively down to all the sub containers and convert their Layout Manager to SLM, i.e., translate all the containers to SLM containers.

5.2.3 The Layout Manager of unit Selexel Containers (SelexelJPanel)

As mentioned before, all the UI components are inside their own Selexel Container, which is SelexelJPanel in our case. The question here is: what is the Layout Manager of this container? It can not be an SLM itself, since it is one unit of presentation and can not be divided anymore, i.e., an atomic division of the UI. If we set its Layout Manager to SLM, it tries recursively to divide the UI to the Selexel unit, and since it is already a unit of Selexels the algorithm will goes in an infinite loop. It is basically like defining something with itself. It never ends up, since there is no termination condition for the recursion algorithm.

The solution is to specify its Layout Manager to one of the standard Layout Managers. Our approach was to choose the Flow Layout Manager. The reason for this decision was that this Layout Manager is the simplest and most intuitive Layout Manager. If we don't specify any Layout Manager

for the Selexel Units, the components may draw on each other; the same problem with having no Layout manager for a UI repeats. So we would need precise sizing and locating of all the components.

5.2.4 Layout Manager or Layout Manager 2

If you see the section about implementing a custom Layout Manager, you see two possibilities for such an implementation. You can either implement the LayoutManager Interface or the LayoutManager2 interface. In our project we needed to implement the LayoutManager2, since we wanted to override the function *addLayoutComponent(Component comp, Object constraints)*, which doesn't exists for the LayoutManager interface. LayoutManager2 is basically extending the LayoutManager, i.e., it has the functions of LayoutManager, but the other way around is not valid.

5.3 How to Adapt an already existing UI

If you have a program, creating a UI, you can adapt it to the Selexel constraints with adding two settings at the end of your program. Any UI application has a function which is adding the components to their containers. For example consider Hello world application in Appendix 2. This code includes every parts that a typical Swing program needs to include. The application is in full-screen mode.

For adapting the UI we just need to add the code below to the *createAndShowGUI()* function, after all the code for adding the components is finished.

As you see in the code, just two settings are needed; one is for setting the Layout Manager to SLM and the other one is to set the Glass Pane of the Content Pane to our Selexel Glass Pane. SLM will translate the Layout Manager to the Selexel Layout Manager, and the Glass Pane setting will

```
Dimension selexelsize = new Dimension(100,100);
SelexelGlassPane myGlassPane;
contentpane.setLayout(new SelexelLayout(selexelsize,frame,false));
myGlassPane = new SelexelGlassPane(selexelsize,contentpane);
frame.setGlassPane(myGlassPane);
myGlassPane.setVisible(true);
```

draw the Selexel cursor on the screen and hide the actual cursor (standard arrow-shaped cursor) and process the Selexel cursor events.

If the input device be changed during the run time dynamically, the *setSelexelSize()* function should be called with the new computed size of Selexels. After adding the adaptation source code, the code will look like the second program in appendix 2.

5.4 How to write a new Selexel-based UI

For writing a new UI application with Java, there are few changes from the normal Java application development process. The only difference is that instead of setting the Layout Managers of the containers to the Java standard ones, we set it to the Selexel-based one, which is basically adding the term "Selexel" in the beginning of its name. For example for the FlowLayout we set the Layout Manager to SelexelFlowLayout.

Another way, which I recommend, is to write your own UI application in the traditional way you have always done and then add the code for adaptation, as explained in section 5.3—"How to Adapt an already existing UI". In this case you will not set the Containers' Layout Manager one by one to the Selexel-based version, but you just set the Content Pane's Layout Manager to SLM at the end. It will then recursively change the Layout Managers automatically and the result will be the same.

5.5 How to implement a custom Layout Manager

In this section I will explain how one can implement a custom Layout Manager. This information is taken from [Walrath et al., 2004].

Before implementing your custom Layout Manager, you should make sure what you want to do is not doable or can not be done efficiently by Java standard Layout Managers, or the Layout Managers that can be found in the Internet. In most cases, you can lay out your UI by using a good combination of standard Layout Managers.

If it didn't work then start implementing your own as following:

You need to implement one of the interfaces: LayoutManager or its sub-interface LayoutManager2. In any case you need to implement at least the following 5 methods:

void *addLayoutComponent(String, Component)*
Adding a component to the container and specifying a string for it.

void *removeLayoutComponent(Component)*
removing a component from the container. Many layout managers do nothing in this method, relying instead on querying the container for its components, using the Container method getComponents.

Dimension *preferredLayoutSize(Container)*
Calculate and return the ideal size of the container, assuming that the components it contains will be at or above their preferred sizes. This method must take into account the container's internal borders, which are returned by the getInsets method.

Dimension *minimumLayoutSize(Container)*
Calculate and return the minimum size of the container, assuming that the components it contains will be at or above their minimum sizes. This method must take into account

the container's internal borders, which are returned by the getInsets method.

void *layoutContainer(Container)*
It doesn't draw components. It simply invokes each component's setSize, setLocation, and setBounds methods to set the component's size and position.
This method must take into account the container's internal borders, which are returned by the getInsets method. If appropriate, it should also take the container's orientation (returned by the getComponentOrientation method) into account. You can't assume that the preferredLayoutSize or minimumLayoutSize method will be called before layoutContainer is called.

If you wish to support component constraints, maximum sizes, or alignment, then your layout manager should implement the LayoutManager2 interface. That interface adds five methods to those required by LayoutManager:

- addLayoutComponent(Component, Object)
- getLayoutAlignmentX(Container)
- getLayoutAlignmentY(Container)
- invalidateLayout(Container)
- maximumLayoutSize(Container)

As you can see the *addLayoutComponent* function exists for both interfaces, but with different arguments; instead of *String*, LayoutManager2 uses the argument *Objects*. This argument includes the constraints (features) the component has. It is a type of object, which means any subtype of objects can be used. We needed to implement the LayoutManager2 interface for Selexels implementation, since we wanted to let the programer adding the components to their containers by specifying their ratio locations. For example, if a programer wants to add component *button1* to its container (*container1*) in a way that the component is standing at the center of the container, he writes:

container1.add(button1,0.5F,0.5F)

The ratio is calculated according to the container and can be a float number from 0 to 1.

5.6 How to implement your own Selexel-based Layout Manager

In order to implement your own Selexel-based Layout Managers you need to do the following steps:

- Design issues: Think deeply, which policy the Layout Manager should have for relocating and resizing the components.
- Implementation issues: Subclass the SLM, i.e., extend the SelexelLayout
- Check which functions need some changes, i.e., override. For this step it is necessary to understand what each of the functions in the SelexelLayout is doing.
- Implement just the functions that need to be changed
- Done!

As an example consider the SelexelFlowLayout and SelexelGridLayout that are already implemented. They have also done the steps above.

The functions most probably need to be overridden are the functions below:

- layoutContainerChild
- layoutNonSelectableChild
- layoutSelectableChild
- preferredLayoutSize

The layout functions are responsible for relocating and resizing the three groups of UI components: Containers (have

children inside, e.g., a frame), Selectables (e.g., a button), Non Selectables (single widget that is not selectable, e.g., a label)

and the preferredLayoutSize, which is responsible for computing the preferred size of the container.

In some cases, such as in the SelexelFlowLayout the layout functions were doing what we wanted, but the priorities for finding another location in the case that the original location is occupied was different, therefore we just needed to override the Find functions for each case as below:

- FindSelectableFreeSpace
- FindEmptySpace
- FindContainerFreeSpace

What finally needs to be done is to make SLM consider the newly created Layout Manager as one of the Layout Managers it needs to call during its translation. If the created Layout Manager is called *SelexelXLayout*, the code below must be added to the *ConvertLayout* function, which is inside SLM:

```
 if((parent.getLayout() instanceof XLayout)){

parent.setLayout(new SelexelXLayout(selexelSize,parent,true));

}
```

These are the steps for writing a Selexel-based version for a standard Layout Manager (e.g., BorderLayoutManager). If the Selexel-based Layout Manager is something different, it needs to be specified inside the if condition, in which situations SLM needs to convert the container's Layout to the new Layout. This feature is used while adapting a Content Pane, including some container children, which each have their own different Layout Manager. Of course this last step can be ignored and the new Layout Manager can be used as a stand alone Layout Manager.
After doing the above steps, the new Selexel-based Layout Manager is ready.

Chapter 6

Design Process

"We can't solve problems by using the same kind of thinking we used when we created them."

—Albert Einstein

In this chapter, I will go through our design process: explaining about the earlier prototypes that we have designed and implemented, the challenges we had, and our solutions for solving the faced problems. This design process is done according to the DIA Cycle. DIA is abbreviation for Design, Implement, Analyze. Nielsen has described this iterative way of designing UIs in [Nielsen, 1993]. UIs will be refined iteratively over several versions (i.e., prototypes). In each of the iterations the produced prototype get more matured and the users' feedbacks to the prototype gets more concrete. We have had 4 different prototypes and the 4th one is our final design, which is explained in the Design chapter.

6.1 Prototype 1: Fitts' Law

As mentioned in the Motivation chapter, the low expressiveness input devices are unable to specify the intended meaning of the users. Our solution to this problem (as explained in Theory, part Selexels) is to adapt the resolution of the UI to the expressiveness of the input device. We have

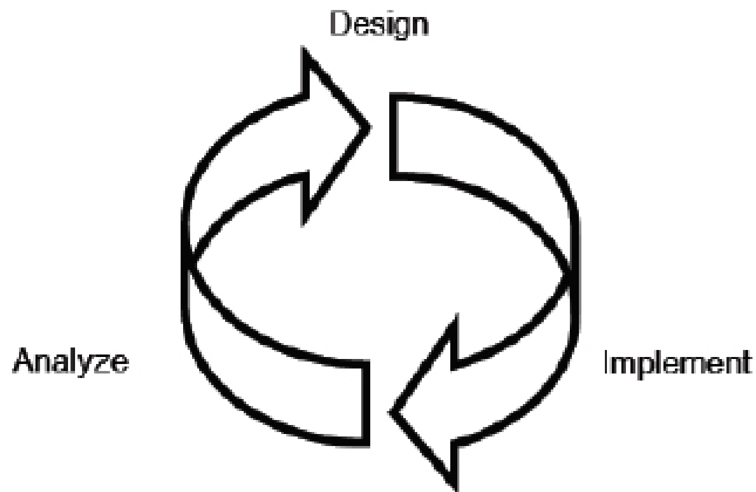

Figure 6.1: Iterative Design- Implement- Analyze introduced by [Nielsen, 1993]

divided the screen space to a Grid of same-sized rectangles, i.e, Selexels. These regions are considered as the selectable unit instead of Pixel (inch per dot), which is a standard for GUIs. With this modification all the input devices are able to work with UI. The difference will be just the way the UI looks like (i.e., the layout of the UI) and the way the mouse will be interpreted (from a hotspot mouse to an area cursor).

In the Selexel approach, the screen space is divided to *equal-sized* regions, i.e., Selexels. It means, as far as the input device isn't changed, or it is changed, but the new input device has the same technical features (i.e., sampling rate and sampling resolution), the size of the Selexels remains unchanged. According to the performance improvement, one may suggest having different Selexel sizes for different UIs/ Dialogues; which means, besides considering the maximum preciseness of the input device, also take the minimum necessary preciseness for that specific UI (dialogue) in to account and set the Selexel size as the minimum of these two. As an example, see figure 6.2, we have a photo that the user can zoom in or out. In this simple UI, the preciseness we need for interacting needs to be enough for selecting the buttons, which have the minimum size in

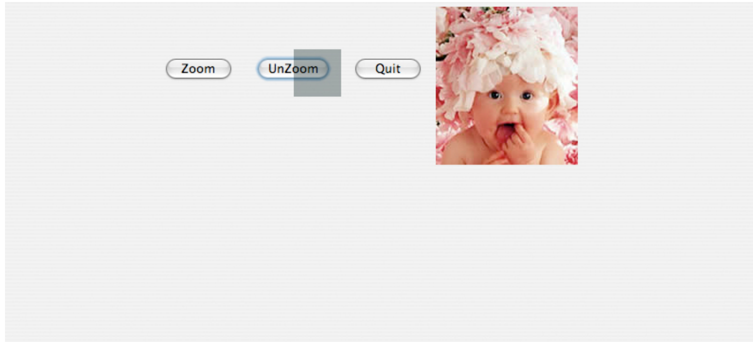

Figure 6.2: An example of an input device that has a higher expressiveness than UI needs.

compare to the other widgets. Although the input device may afford higher resolutions, it is really not needed for this UI. The size of the gray cursor show you the Selexel size in this case and therefore the maximum preciseness of the input device. For this UI, it would be enough to make the cursor as big as the size of the smallest selectable widget. There the user could still choose all the selectable widgets with enough expressiveness.

The problem with this approach is having variety of Selexel sizes for different UIs in the same application. It makes the computation unit of the UI, i.e., Selexels, inconsistent. Consequently, the user can not learn the interaction behavior from his cursor movement, since this Selexel size is changing often for each dialogue. It causes user confusedness and can not help smoothening the user interaction. With having the same Selexel size, users are able to play around with the device and learn how they need to react and move the device in order to perform their intended tasks. Therefore we define the first Selexel rule as:

SELEXEL RULE 1: SELEXEL SIZE RULE:
Selexel size is computed just according to the sampling rate and sampling resolution of the input device. As long as the sampling rate and sampling resolution of the input device are unchanged, the selexel size also remains unchanged.

Definition:
Selexel Rule 1:
Selexel Size Rule

For having an idea, how different input devices can work with this concept, we have tested different input devices (gyro mouse, mouse, joystick and mobile phone, which is used as a cursor controller in sweep technique, see Sweep Technique.
The test application was the horizontal tapping test explained in [ISO, 2000]. In this test the UI includes two

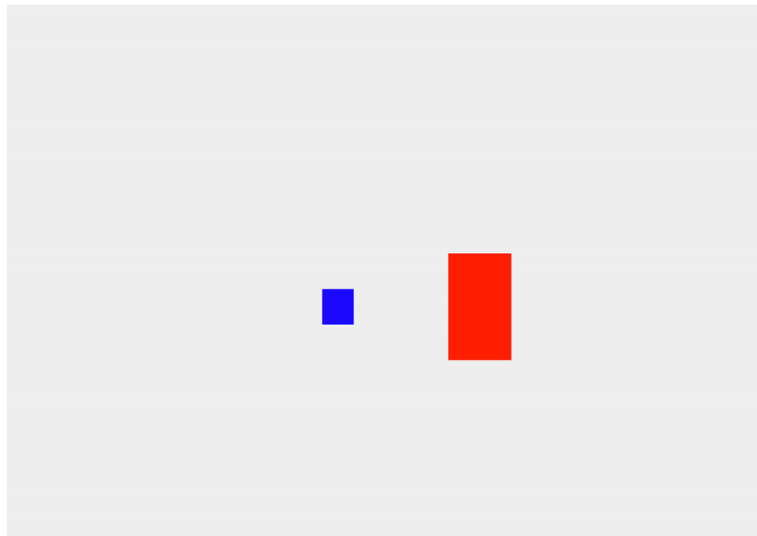

Figure 6.3: Screen shots of the horizontal tapping test, target with the red color and cursor in blue.

targets, but showing just one of them at a time. The user needs to move the cursor, in order to reach the target, and click on it. The result of this user study is explained in Evaluation chapter.

This tapping test was basically just a Fitts' law application and not a real GUI with User Interface widgets. Therefore as the next improvement, we thought of implementing an example UI, which looks more similar to the real UIs, people use in their everyday experience with their computer. The goal was to show how such an adaptable UI can improve the users' interaction. This goal made us start working on the second prototype.

6.2 Prototype 2: Selexel-based SUPPLE Toolkit

The purpose was to implement a real-life UI example that can adapt itself dynamically to the expressiveness of connected input device. For achieving this goal, we decided to extend the SUPPLE Toolkit (see Related Work) which is a Toolkit for automatic generation of UI for different Applications. In the following section you can see our reasons for making this decision.

6.2.1 Why SUPPLE Toolkit

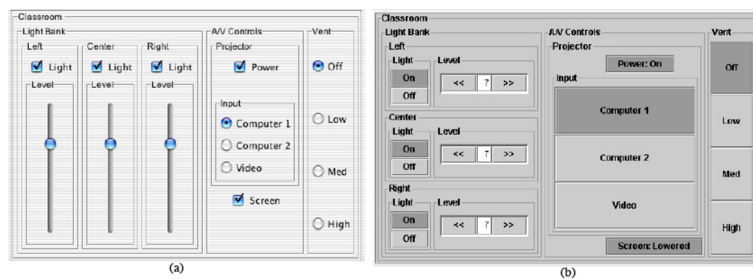

Figure 6.4: The classroom interface rendered for two devices with the same size: (a) a pointer-based device (b) a touch-panel device,[Gajos and Weld, 2004]

SUPPLE Toolkit is automatically generating UIs for different applications and different input devices. It is choosing the optimum UI rendition that fits the available free space on the screen. It is already handling so many problems that come up with automatic generation, such as using tabs in the case of having no more free space, which is the case also for our framework, happens even more often in our case. For example, SUPPLE adapts the UI accordingly, when the input device is a pointer device, such as a mouse, or when it is a touch panel, see figure 6.4 including two screen shots generated automatically by SUPPLE.
SUPPLE stores the constraints related to each of these devices; therefore, it make sense to add the Selexel constraints to SUPPLE, in order to have an automatic generated UI respecting Selexels limitations. In the following, you see the

challenges we have for this adaptation.

6.2.2 Challenges

UI components, in standard UIs, may stand too tight together or be so small, and therefore require a high expressiveness input device to work with. In such cases, in order to adapt the UI, we need to solve the overlapping problem.

Selexel rule 2 says:

Definition:
Selexel Rule 2: Rule of One Selectable Item

SELEXEL RULE 2: RULE OF ONE SELECTABLE ITEM:
No more than one selectable widget can be in the same Selexel.

By Selectable Widget we refer to:

Definition:
Selectable Item/ Widget

SELECTABLE ITEM/ WIDGET:
A widget or UI component is selectable, if it contains a set of items for which zero or more can be selected; e.g., a button, an editable text field, e.t.c.

The possible solutions to achieve Selexel rule 2(i.e., rule of one selectable item) can be:

- Set the widget size as an integer coefficient of the selexel size.
- The distance between two selectable widgets must be at least 1 selexel.
- Without changing the size of the widget, just make sure that in each selexel there are maximum one selectable widget.

The first and second solutions solve the overlapping problem, but they don't respect the Selexel rule 3 (i.e., the rule of Selexel alignment). The Selexel rule 3 says:

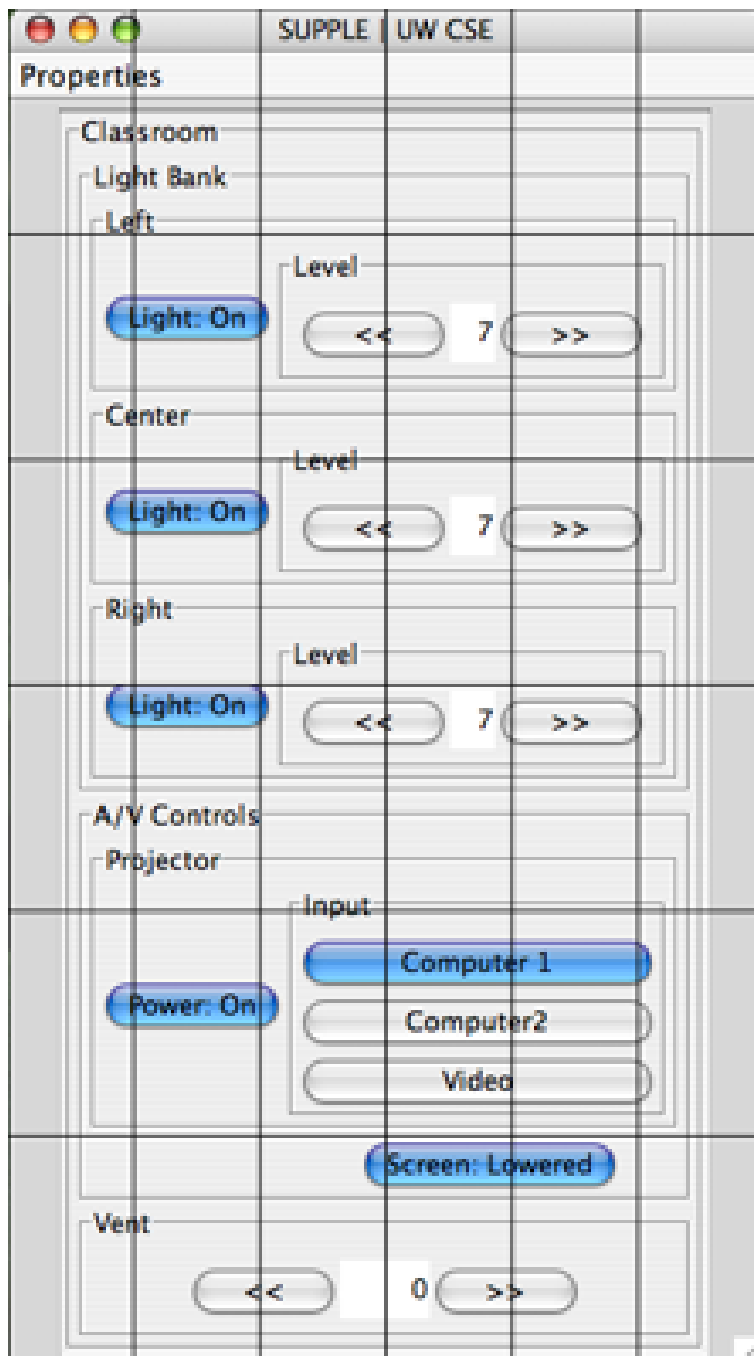

Figure 6.5: This is a screen shot of a UI automatically generated by SUPPLE. The Selexel Grid on top of the UI shows that the adaptation of the UI with the Selexel grid is necessary, since selectable widgets are overlapping in shared Selexels and make the selection task ambiguous.

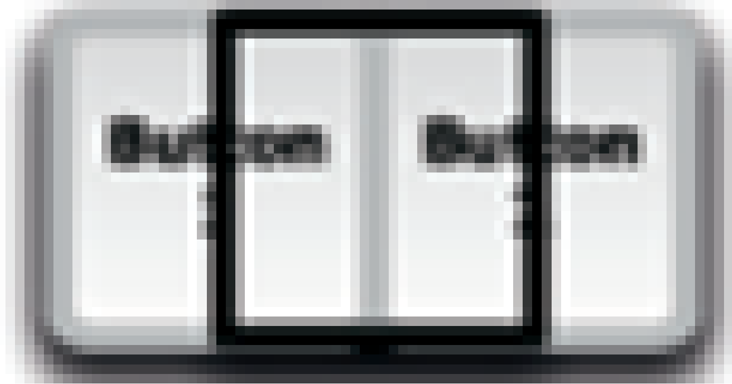

Figure 6.6: This figure demonstrates the alignment problem occurs while using the first solution for solving the overlapping problem.

Definition:
Selexel Rule 3:
Selexel Alignment

Selexel Rule 3: Selexel Alignment:
The widget placement should be aligned with the Selexel grid.

In the first two solutions, we make sure that the widgets are far enough or are big enough, but as far as they are not aligned according to the Selexel grid they may still overlap in the same grid, see figure 6.6.

Therefore, we took the third approach, and decided not to change the size of the selectable widgets, but make sure that in each Selexel maximum one selectable widget is placed. For overcoming the alignment problem, we considered a Selexel grid on the UI and calculated the borders while placing the components. With this approach we ensure the Selexel rule 3 (Selexel alignment).

SUPPLE Toolkit defines a UI generation as a "constrained decision-theoretic optimization problem", [Gajos and Weld, 2004]. The optimum UI is generated for a specific user using a specific input device, by considering the constraints of the input device. First of all a tree of possible UI rendition is generated and then the algorithm starts pruning some of the solutions according to the constraints. Finally among all the remaining solutions one with the minimum cost will be taken as the final decision. By minimum

cost, Gajos and Weld [2004] means the minimum effort the user needs to put for manipulating the widgets of the UI.
For designing the Selexel-based SUPPLE we considered two approaches:

- Take the optimized UI from SUPPLE Toolkit and shift the selectable items.
- Prune the solutions and let the ones that obey our constraints pass

By taking the first approach, we can get to a solution, which is respecting Selexel constraints, but there is no guarantee anymore that the final UI keeps optimum. Briefly say, it is like generating our own UI without using SUPPLE. In this case SUPPLE optimization algorithm can not help us anymore since we are basically changing the UI.

Taking the second approach makes more sense. There we put our Selexel constraints as some additional constraints and then SUPPLE optimization algorithm can find the optimum solution accordingly. Another advantage of this approach will be in the case of not having enough space on the screen for the last added components. In this case SUPPLE have its solution; when there are not enough available space, SUPPLE use tabs in order to place all the components on the screen.

For adding our constraints to SUPPLE we decided to consider two different sizes for each selectable component: the virtual and actual size. The actual size of the component is its current size and the virtual size is basically the size of its bounding box, which is calculated according to the number of Selexels this component takes in the grid; therefore it is always a coefficient of the Selexel size.
The advantage of this approach is that: besides keeping the actual size of the component, as it was originally, it makes the necessary distance between the selectable components, in order to respect the Selexel rule 2 (i.e., the rule of one selectable item). It respects the Selexel rule 3 (rule of Selexel alignment), by making the grid squares to be the bounding box of the components, i.e., the bounding box is aligning the Selexel grid.

faced problem

To implement this approach in the first step, we needed to calculate the Selexel grid and the borders. In the second step the location of each of the UI widgets must be calculated and set the locations to the new locations. By implementing these steps we got to a point that, although the calculated locations were correct, the UI widgets were placed in the wrong positions. After tracing, we found the origin of this problem: Layout Managers!
As explained in details in the Theory chapter, Layout Managers place and size the components on the screen. They have their own policy and they follow their policy even by ignoring the programmers settings. Although we have set the new locations, the Layout Managers used in the SUPPLE Toolkit didn't allow us to position the components precisely.

Two approaches can be taken for solving this problem:

- Ignoring the Layout Managers
- Implementing our own Layout Manager

The first approach doesn't make sense, since by ignoring Layout Managers, programmers need to do the exact positioning of all the components, which can be many lines of code.

Solution

We decided to implement a custom layout manager that places the UI widgets with respecting Selexel rules. The advantage of this approach is having a reusable tool that every programmer can use it in order to have Selexel-based UI. This idea leads us to design our third prototype.

6.3 Prototype 3: Custom Layout Manager and Transparent Layer

As explained in the previous prototype, we take advantage of Layout Managers, in order to layout our UI with respect to Selexel constraints. We called this custom Layout Manager Selexel Layout Manager (SLM).

SLM's algorithm step by step is as following:

- SLM takes the original locations and sizes of the UI components
- It computes the Selexel grid borders according to the Selexel size; each cell of the grid is a container itself with the size of one Selexel.
- For all the containers, it recursively checks if the corresponding Layout Manager is a SLM; if not it will convert it to a SLM.

For converting to an SLM the following steps are done:

- If the current widget is Selectable then put it inside a Selexel that is free of any selectable widget.
- If the current widget is a non-selectable widget or a container no specific Selexel constraint exists; i.e., act as standard Layout Managers

This approach works well for adapting already created UIs (having their own Standard Layout Managers), and also for starting to create a UI with this Layout Manager. It tries to keep the location and size features of the components and perform minimum changes necessary for adaptation.

6.3.1 Challenges

With having a Selexel grid for our layout and alignment, we may need to put the widgets further from each other, in comparison to their original distance together. Consequently, we may face a situation, which the original location of a widget is occupied by another widget added before. The question is where to put this widget. Our solution to this problem was to put the widget in the first free nearest neighborhood. This method is explained more in the Design chapter; see figure **??**.

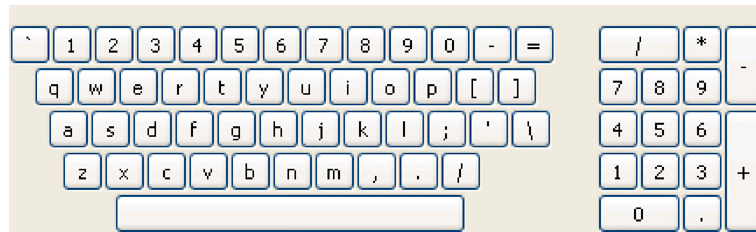

Figure 6.7: An example of a standard layout. This layout can not be changed, while adaptation.

Faced Problems

The problem we faced, was the semantic meaning of the UI, i.e., each programmer, while implementing a UI, has his own intentions and purposes, which SLM can not know about it. What SLM does is basically adding the UI components to the Selexel grid, while ensuring the Selexel rules. If a widget can not be placed in its original location, it will be placed in its nearest neighborhood, which is a good guess, but not always exactly how the programmer intended it to be. To demonstrate this problem, lets look at the example UI in figure 6.7. This is a virtual keyboard. The Layout of a keyboard is standard and should not be changed because of adaptation purposes. SLM will not keep the layout as it is, by using the nearest neighborhood algorithm.

Our solution to this problem was to take the original Layout Managers, the programmer has used in to account, and guess the intended positioning of the components accordingly. This approach formed our 4th (last) prototype.

6.4 Prototype 4: Selexel Layout Manager hierarchy

In our last prototype, we tried to have a closer guess about the location decision. Although prototype 3 was already using the nearest neighborhood, in order to relocate the components close to the intended location, it is still not clear at this point, which of the neighbors should have priority to others. In some cases, e.g., in the Grid Layout it makes more sense to scale the whole layout and even change the size of the components, since when a program-

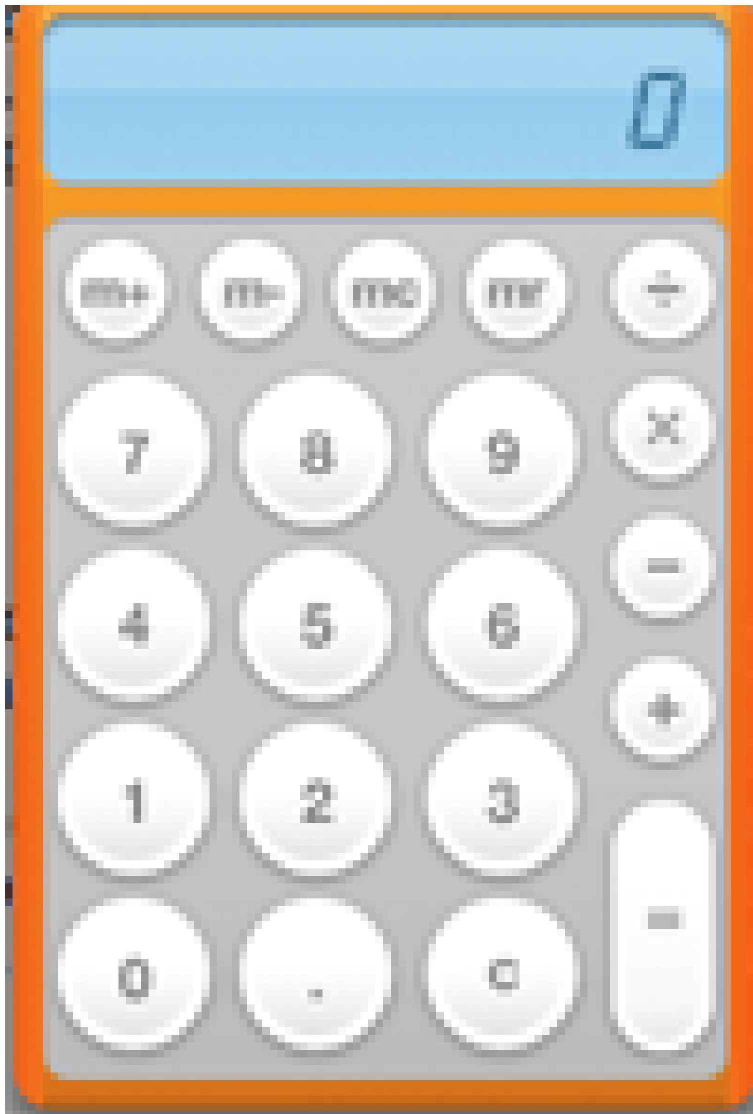

Figure 6.8: calculator layout.

mer use this Layout Manager, in order to lay out his UI, it means he intended to make some components in rows and columns with the same size. In such cases, when the components needs to stand further from each other, it is better to make the size of the components bigger, so that after putting them inside the Selexel grid they still stand closely together and make the UI looking less different. In different application scenarios the UI designer might want to present

a different layout, which needs some semantic information about the context of the UI.

Our solution for this problem was to write different Selexel-based version of the standard Layout Managers, such as Flow Layout, Grid Layout, Border Layout, etc. Each of these versions are acting according to the original Layout Managers they are based on, besides respecting the Selexel rules. Selexel Flow Layout Manager, e.g., is a Selexel-based Flow Layout Manager that besides ensuring the Selexel constraints, is following the Flow Layout Manager policy for putting the components on the screen. Flow Layout manager in Java doesn't use the nearest neighborhood algorithm we used in the SLM, but just put the added component on the right side, next to the previous component; see figures 4.2 and 4.3.

For achieving this goal we have sub classed the SLM to dif-

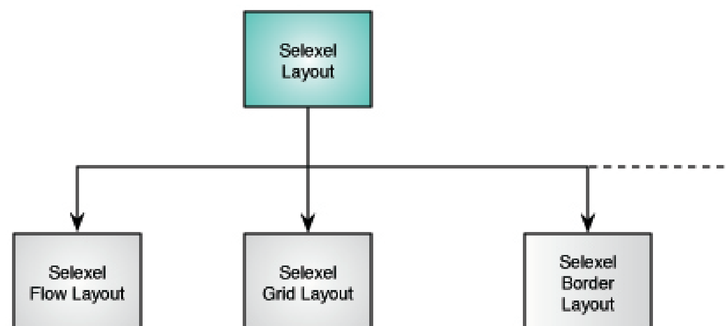

Figure 6.9: The hierarchy of Selexel Layout Manger classification

ferent Layout Managers; see 6.9

This approach help us to have a good guess for location and size decisions, but as showed in figure 6.7 there are some more complicated situations that the layout needs to remain in the frame of the same concept, and in this case the semantic information is missing to the system.

The way this prototype works is similar to prototype 3. The difference is in the Convert part. The Algorithm checks which Layout Manager type the container has originally

and convert the container to the Selexel-based version of it. For example if the original Layout Manager is Flow Layout then it will be converted to Selexel Flow Layout. In the cases that the Layout Manager is a customized one, implemented by the programmer, the nearest neighborhood algorithm is taken in order to make the best guess of the widgets' locations. The convert function can be imagine as a translator from the original Layout Manager to the corresponding Selexel-based one.

For the Multi User Applications rule number 4 says:

SELEXEL RULE 4: MULTI USER RULE:
Get the maximum size of the Selexel in the case of multi-user applications.

Definition:
Selexel Rule 4: Multi User Rule

In this case the shared UI will be adapted according to the sampling rate and sampling resolution of the weakest input device; therefore, all the users with all kind of input device power can fluidly interact with the application without having any advantage because of having an input device with better technical specifications.

More Information about our final prototype is given in the Design chapter.

Chapter 7

Evaluation

"Anyone who has never made a mistake has never tried anything new."

—Albert Einstein

7.1 Experiment 1

7.1.1 Introduction

This experiment was a preliminary evaluation of Selexel framework, comparing mobile phone (using Sweep technique) with some standard input devices, namely: joystick, gyro mouse, and mouse.

As mentioned before, researchers are working on novel interaction devices for Ubiquitous Computing applications. These input devices need to be tested, while they are in the prototype phases of development. Fair evaluation of these new techniques against the standard input techniques is difficult, since the standard input devices are more established. The prototype input devices, such as Sweep mobile phone, usually suffer from low expressiveness. In order to fairly compare Sweep mobile phone with other input devices, we need to test them under equal conditions, i.e., we

need to test them all under low expressiveness situation.

With running this experiment we wanted to test if the mobile phone performs better than other selected input devices.

7.1.2 Experiment Design

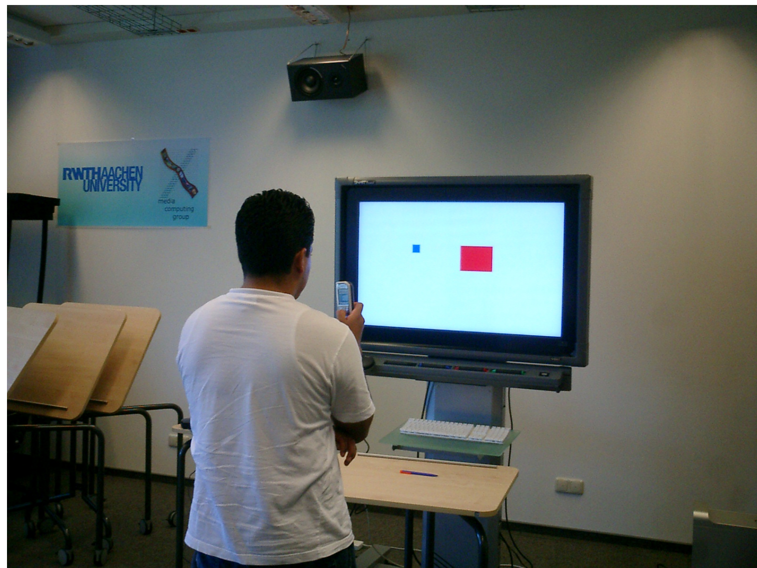

Figure 7.1: User is doing the tapping test with a mobile phone which was running the Sweep technique.

Within-group design

This user study is done with a within-group design. In a within-group experiment design, all of the users test all the systems, unlike the between-group design, that each user just test one system, [Nielsen, 1993]. In this experiment we let all the users to do the experiment with all the selected input devices.

Technical Details

The test application was the horizontal tapping test, mentioned in ISO [2000]. As you can see in picture 7.1, the output device was a large public display (a 40-inch NEC LCD screen) and the users needed to use 4 different input devices in order to interact with the application. The task was to reach the target on the screen, which was a red-colored rectangle, and clicking on it. The application was hiding the

cursor and showing a rectangular-shaped Selexel cursor on the screen. After clicking the target, it disappears and the second target will appear on the other side of the screen.

The input devices included a Logitech MX900 Bluetooth optical mouse, a Gyration GyroRemote gyromouse, a Logitech Freedom 2.4 cordless joystick, and the Sweep technique running on a Nokia 7610 mobile phone.

Figure 7.2: The 4 different input devices we used for Experiment 1.

In the test application, changing the level of expressiveness was simulated by varying the display refresh rate. In order to store all the samples, we have buffered all the device motions, till the display was refreshed. Consequently the sampling resolution for a single sample was effectively increased, but the frequency of the samples were decreased. This allowed for a consistent feel of the sensitivity of the device in terms of screen distance across the different selexel resolutions. For the case of low expressiveness in this experiment the Selexel resolution was 20x15, and the display refresh interval equaled 80 ms.

Three different Index of Difficulties (ID) are considered with values of 1.7, 2.7, and 3.7. The data set can be found

in appendix 3. This provided a total of 12 (3 IDs * 4 input devices) different tapping tests for each user. The tests were grouped first by device, and then the users would complete all indices of difficulty for each device.

The order of devices, and index of difficulty for each user and each condition was alternated to minimize learning effects. At the beginning of each new device, users were given a couple of minutes to practice with the technique until they felt comfortable. We encouraged the users to do the task as quickly and accurately as possible. To motivate our participants, we offered a prize for the best overall score where points would be added for speed and reduced for errors. In order to simulate a large public display interaction, we asked users to stand for all interactions except with the mouse. The interaction took place while standing (or sitting in the case of the mouse) one meter away from the screen.

Totally 16 users participated in the study, 19% female. Age of the participants ranged from 22 to 29.
93% of the participants claimed to use the mouse every day, and the remaining participants reported a high frequency of use. 31% had never used a joystick, 38% less than once a month, and 25% once a month or more. None of our users had ever used a gyromouse. 93% had never used the Sweep technique, and the remaining had used it only once in an open house demonstration earlier in the year.

7.1.3 Results

The results for the tapping test are depicted graphically in figure 7.3

The results for the linear regression analysis are as follows:

$$Mouse : MT = 0.224 + 0.280ID(R^2 = 1.00) \tag{7.1}$$
$$Joystick : MT = 0.0294 + 0.644ID(R^2 = .996) \tag{7.2}$$
$$Gyromouse : MT = 0.0036 + 0.484ID(R^2 = .992) \tag{7.3}$$
$$Sweep : MT = 0.235 + 1.14ID(R^2 = 1.00) \tag{7.4}$$

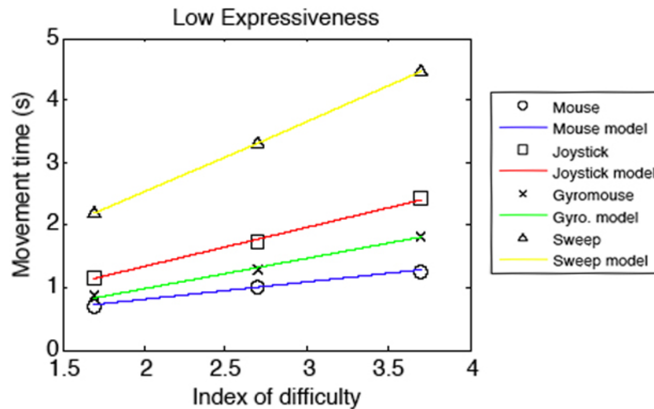

Figure 7.3: Graphs of the results from the user test showing the fit of the models with actual data.

7.1.4 Discussion

The Sweep technique still performs far worse than the other input techniques in terms of task performance, even though we have examined them in a way that is relatively independent of resolution and sample rate. That means that other factors might be at play that are affecting relative performance.We have identified the following possibilities:

- **Error rate**: The motion detection algorithm used in the Sweep technique is not perfect and does make errors in the direction and magnitude of the movement detected. This can impede progress towards the intended target. This factor is theoretically easy to counter balance in an evaluation by intentionally incorporating errors into the input of other input techniques. However, it is very difficult to accurately characterize the error rate of the system.
- **Variance of lag**: The Bluetooth profile used on the mobile phone (SPP) only supports "guaranteed delivery", meaning that if a packet is lost or corrupted it must be resent, slowing down the data transmission. Although the average sample interval of the Sweep technique was faster than 80ms, the variance of the lag could push the sample interval above that threshold. In contrast, the Bluetooth profile used by the

mouse (HID) is optimized for input devices and does not necessarily guarantee delivery of packets. Variance of delay has been shown to increase task performance time [Park and Kenyon, 1999], but no model has been developed to isolate its effect.

- **Ergonomics**: The mobile phone has not been designed with the Sweep interaction in mind. The shape and position of buttons on the gyromouse, for example, make it much more appropriate for these types of pointing tasks. With the gyromouse, the clutch is located underneath the device to make it feel like a trigger, learning from the stability considerations that go into weapons design. It would be appropriate to consider alternative form factors of the mobile phone with these interactions in mind.

Clearly task performance time is not the sole indicator that should be considered in judging a device, but it is an important one. Had the task performance time of the Sweep technique been closer to that of the rest of the devices, it would have been beneficial to examine the technique under several different lag conditions and perform a regression analysis to determine the device specific constants for the MacKenzie and Ware lag model, [Mackenzie and Ware, 1993].

7.2 Experiment 2

7.2.1 Introduction

As explained in Related Work (Area cursor), a Selexel cursor is like an area cursor in pixel space, but in a Selexel space,which is the space of our purpose, it is acting like a point cursor. According to the previous analysis of area cursors[Worden et al., 1997], the index of difficulty decreases while selecting smaller targets with area cursors. A Selexel-wise motion may impede or annoy users, since users expect a pixel-wise motion. Therefore, it is important to determine empirically whether Fitts' law holds for Selexel cursors.

7.2.2 Experiment Design

This experiment has used the same test application as experiment 1. The experiment design was within-group like the previous experiment, i.e., all the users have tried all the test conditions. Our goal for this user study was to show that pointing in the selection space can be modeled using Fitts' Law. The input device was just the bluetooth mouse from the previous experiment and the output device was an Apple Cinema Display 23".
20 participants have taken part in this user study. We had 6 different levels of expressiveness. A special test program allowed us to simulate different levels of expressiveness by varying the selexel resolution and the display refresh rate. Six different lndex of difficulties (ID) were considered, which was the result of 2 different distances between targets and 3 different target widths. Therefore we had 36 (6 expressiveness * 6 ID) combinations that the users needed to do. The detailed data set can be found in appendix 3.

7.2.3 Results

The result of this user study is graphically shown in figure 7.4 The results approved that pointing in the selection space can be modeled by Fitts' Law. More information will be published in CHI 2007, [Ballagas, 2007].

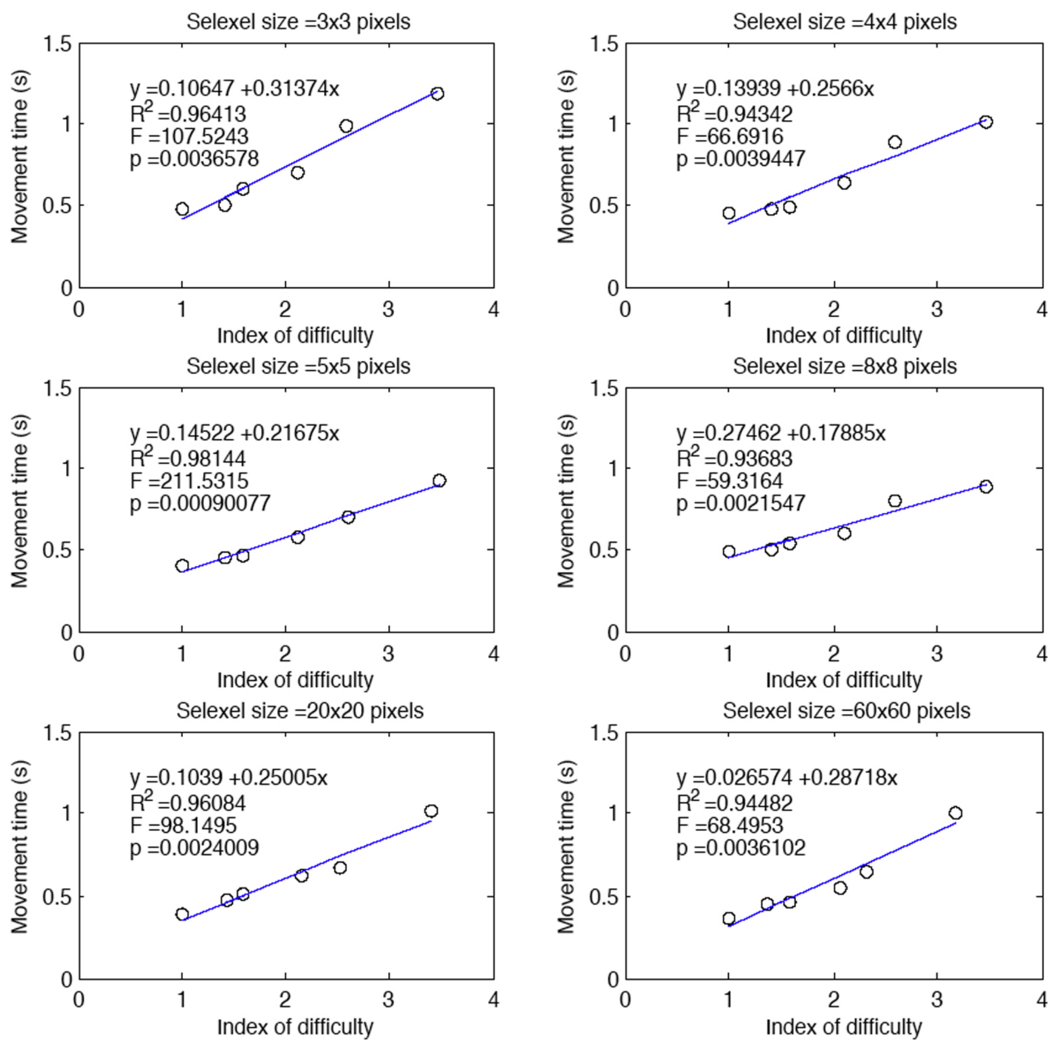

Figure 7.4: Graphs of the results from the user test showing that pointing in Selection space can be be modeled using Fitts' Law.

Chapter 8

Summary and future work

"Imagination is more important than knowledge..."

—Albert Einstein

8.1 Summary and contributions

Selexel framework is a conceptual framework that allows the user interface to be tailored to match the expressiveness of the input device, without sacrificing the screen resolution which is important to preserve the information capacity of the display. We have shown how this framework can be used as a design tool for applications intended to be used with low-expressiveness input by matching the selection space of the application to the expressiveness of the input device. By this adaptation all the users with different input devices can enjoy an equally-smooth experience of interaction.

Our evaluation shows that pointing under the selexel framework can be modeled using Fitts' Law, regardless of the selexel resolution.

Selexel Toolkit is an implemented prototype of Selexel software framework. It allows rapid adaptation of Java Swing applications to the attached low expressiveness input device.
The main feature of this Layer-based approach is reusability. The GUI programmers can adapt their own already-created UIs, just by adding a few lines of code. Selexel Toolkit tries to keep the original size and the location of the UI components, as far as they respect the Selexel constraints.

The Toolkit makes a good guess for placing the components in the case of Flow- and Grid Layout Managers. An approximate location decision is given,by using the nearest neighborhood algorithm, for other Java standard Layout Managers, and also for programmers' custom Layout Managers.

8.2 Future work

Although the Selexel framework has achieved its main goal, there are some improvement possibilities that can make the framework more robust against different application scenarios.

The problems the current system has and the possible solutions for them are as below:

- **Lack of enough space on the screen**:

  With wider Selexel sizes, the Toolkit will make the selectable widgets further and further from each other, and the probability of not having enough space on the screen to show all the widgets, gets higher.

  A solution can be to add the extra widgets inside different Tabs, or make a Scroll bar for the GUI so that the user can access all the widgets.

- **Semantic UI adaptation**:

Although the Toolkit is adapting according to the specific Layout Managers the programmer has used, but the exact information about the UI context and the purpose behind the UI is not visible to the Toolkit. This lack of semantics can cause some meaningless, adapted UIs that can not achieve the intended purpose of the programmer anymore.
There are two different approaches to solve this problem:

– Automatic adaptation:

  In this case, we just want to allow the automatic adaptation of the UI, without programmers' help.
  The solution in this case can be to continue the approach we have taken, and implement Selexel versions of other standard Layout Managers, e.g., BorderLayout

– User configurable:

  The programmer is allowed to help the Selexel Toolkit with understanding the context and logical information behind the UI.
  An Intelligent Selexel Layout Manager with policy configuration files can solve the problem. The programmer can give his priorities to the system by adjusting the policy configuration files and the Selexel Toolkit can change its placing and resizing policy accordingly.

- **Selexel Toolkit is just able to handle the full screen Java Swing applications**:

  In the future one can extend it to be used also for resizable window applications.
  Having resizable window application is challenging, since when one resizes or moves the window, the cursor alignment would be hard to preserve.

- **Selexel Toolkit doesn't work with the widgets that are instance of Adjustable Interface**:

Adjustable Interface is the interface for objects which have an adjustable numeric value contained within a bounded range of values. Subclasses of this interface are JScrollBar and ScrollBar, namely the widgets that have the scrolling abilities, such as a ScrollPane.
The reason is that: firstly, the containers with this feature have a special Layout Manager called ScrollPaneLayout, which can not be converted to other kinds of Layout Managers. Therefore our attempt to convert it to the SelexelLayoutManager failed.

Secondly, the scroll-abled widgets needed to be treated especially, in order to obey the Selexel constraints. More precisely, all the Selexels that the scroll bar is occupying must be free of any other selectable widgets, since during the scrolling task the moving part of the scroll bar, which is a selectable item itself, is passing by all these Selexels.
A solution to this problem is subclassing the ScrollPaneLayout to SelexelScrollPaneLayout and make the changes needed to this subclass.

- **The cursor is currently a simple gray rectangle**:

  Adding some cursor effects, such as, changing its shape in the waiting mode can give the users more feedback about the current status of the application.

- **More powerful programming language**:

  While implementing the Selexel versions of standard Layout Manager, the need of using another language for this purpose was felt. The best way for implementing the Selexel-based versions of Java standard Layout Managers would be to extend both SelexelLayoutManager and the original Layout Manager, which is not possible in Java programming.

Appendix A

Selexel Golden Rules

The Selexels rules are briefly presented in this appendix, in order to facilitate a quick look, when this thesis refers to them in some chapters.

Selexel Rule 1: Selexel Size Rule

Selexel size is computed just according to the sampling rate and sampling resolution of the input device. As long as the sampling rate and sampling resolution of the input device are unchanged, the selexel size also remains unchanged.

Selexel Rule 2: Rule of One Selectable Item

No more than one selectable widget can be in the same Selexel.

Selexel Rule 3: Selexel Alignment

The widget placement should be aligned with the Selexel grid.

Selexel Rule 4: Multi User Rule

Get the maximum size of the Selexel in the case of multi-user applications.

Appendix B

Hello World Swing Code

```java
import java.awt.*;
import javax.swing.*;

public class HelloWorldSwing {
    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private static GraphicsDevice device;
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setLayout(new FlowLayout());
        JLabel label = new JLabel("Hello World");
        JButton button = new JButton("Welcom");
        Container contentpane=frame.getContentPane();
        frame.getContentPane().add(label);
        frame.getContentPane().add(button);
        frame.setUndecorated(true);
        frame.setResizable(false);
        //Display the window.
        device.setFullScreenWindow(frame);
        frame.validate();
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
                device = env.getDefaultScreenDevice();
                createAndShowGUI();
            }
        });
    }
}
```

Figure B.1: Hello World Swing before adaptation.

```java
import java.awt.*;
import javax.swing.*;
public class HelloWorldSwing {
    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private static GraphicsDevice device;
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setLayout(new FlowLayout());
        JLabel label = new JLabel("Hello World");
        JButton button = new JButton("Welcom");
        Container contentpane=frame.getContentPane();
        frame.getContentPane().add(label);
        frame.getContentPane().add(button);
//      adaptation Code//////////////////////////////////////////////////
        Dimension selexelsize = new Dimension(100,100);
        SelexelGlassPane myGlassPane;
        contentpane.setLayout(new SelexelLayout(selexelsize,frame,false));
        myGlassPane = new SelexelGlassPane(selexelsize,contentpane);
        frame.setGlassPane(myGlassPane);
        myGlassPane.setVisible(true);
        ///////////////////////////////////////////////////////////////////
        frame.setUndecorated(true);
        frame.setResizable(false);
        //Display the window.
        device.setFullScreenWindow(frame);
        frame.validate();
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
                device = env.getDefaultScreenDevice();
                createAndShowGUI();
            }
        });
    }
}
```

Figure B.2: Hello World Swing after adaptation.

Appendix C

User Study

You can find the data used in the experiment number 2 below.

Expressiveness	Number of Selexels	Latency	Selexel Size(pixels)
E1 (highest)	640*400	12	3*3
E2	480*300	18	4*4
E3	384*240	27	5*5
E4	240*150	40	8*8
E5	96*60	60	20*20
E6 (lowest)	32*20	90	60*60

Figure C.1: This table shows the data used for the experiment number 2.

One of the UI examples implemented was an application which can search in Flickr[1] data base for finding the entered keywords. As you can see also in the pictures the user can select a picture on the screen, and the selected picture will be shown on the right side of the screen in its original size. This application has used SelexelFlowLayout and also SelexelGridLayout for laying out the picture album in the left side of the screen.

[1] www.flickr.com

Screen size = 1920*1200
D1= ½ Screen Width =960 Pixels
D2= ¼ screen width = 480 Pixels
W1= 0.05 * Screen Height= 96 Pixels
W2= 0.15* Screen Height= 288 Pixels
W3=0.25* Screen Height =480 Pixels
We have 6 combinations, i.e., 6 IDs.
ID1 [D1, w1] = 3.47
ID2 [D2, w1] = 2.59
ID3 [D1, w2] = 2.12
ID4 [D2, w2] = 1.42
ID5 [D1, w3] = 1.59
ID6 [D2, w3] = 1.00

Figure C.2: The data used for the experiment number 2.

Figure C.3: A picture of a user interacting by Sweep mobile phone with the large public display. The UI example used here is the Flickr application.

Figure C.4: A picture of a user interacting by Sweep mobile phone with the large public display. The UI example used here is the Flickr application.

Bibliography

R. M. Baecker and W. A. S. Buxton. *Human-computer interaction: a multidisciplinary approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1987.

R. Ballagas, M. Rohs, J. G. Sheridan, and J. Borchers. Sweep and point & shoot: Phonecam-based interactions for large public displays. *CHI '05': Extended abstracts of the 2005 conference on human factors and computing systems.*, 2005.

Tico Ballagas. Selexels: a conceptual framework for pointing devices with low expressiveness. CHI 2007, 2007.

Patrick Baudisch, Xing Xie, Chong Wang, and Wei-Ying Ma. Collapse-to-zoom: viewing web pages on small screen devices by interactively removing irrelevant content. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 91–94. ACM Press, 2004.

S. K. Card. *Intelligent Interfaces: Theory, Research and Design*. Elsvier Science Publishers B.V., North-Holand, Amesterdam, 1989.

Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. A morphological analysis of the design space of input devices a morphological analysis of the design space of input devicesa morphological analysis of the design space of input devices. *ACM Transactions on Information Systems (TOIS)*, 9(Issue 2):99–122, April 1991.

Andrew T. Duchowski. *Eye tracking methodology: theory and practice*. Springer, 2003.

Krzysztof Gajos and Daniel S. Weld. Supple: automatically generating user interfaces. In *IUI '04: Proceedings of*

the 9th international conference on Intelligent user interface, pages 93–100. ACM Press, 2004.

Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld. Fast and robust interface generation for ubiquitous applications. In *UbiComp 2005: Ubiquitous Computing*, 2005.

G. Goos, J. Hartmanis, and J. van Leeuwen, editors. *Social Aspects of Using Large Public Interactive Displays for Collaboration*, 2002. Springer.

Tovi Grossman and Ravin Balakrishnan. The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 281–290. ACM Press, 2005.

Antonio Haro, Koichi Mori, Vidya Setlur, and Tolga Capin. Mobile camera-based adaptive viewing. In *Proceedings of the 4th international conference on Mobile and ubiquitous multimedia*, pages 78–83. ACM Press, 2005.

Yoshihide Hosokawa, Naoki Kimura, and Naohisa Takahashi. An implementation method of a location-based active map transformation system. In *Proceedings of the 6th international conference on Mobile data management*, pages 13–21. ACM Press, 2005.

ISO. Ergonomic requirements for office work with visual display terminals (vdts) - requirements for non-keyboard input devices. *ISO 9241-1*, 2000.

Poika Isokoski. Text input methods for eye trackers using off-screen targets. In *Proceedings of the symposium on eye tracking research and applications*. University of Tampere, November 2000.

Robert J.K. Jacob. Eye tracking in advanced interface design. In *Advanced Interface Design and Virtual Environments*, pages 258–288. Oxford University Press, 1994.

P. Kabbash and W. Buxton. The "prince" technique: Fitts' law and selection using area cursors the "prince" technique: Fitts' law and selection using area cursthe "prince" technique: Fitts' law and selection using area

cursors ors. In *Proceedings of CHI'95*, pages 273–279. ACM Press/Addison-Wesley, 1995.

I. Scott Mackenzie and Colin Ware. Lag as a determinant of human performance in interactive systems. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 488–493, Amsterdam, The Netherlands, 1993. ACM Press.

Kento Miyaoku, Suguru Higashino, and Yoshinobu Tonomura. C-blink: a hue-difference-based light signal marker for large screen interaction via any mobile terminal. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 147–156, 2004.

Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 161–170. ACM Press, 2002.

Jakob Nielsen. Iterative user-interface design. *IEEE Computer*, 26(11), 1993.

K. S. Park and R. V. Kenyon. Effects of network characteristics on human performance in a collaborative virtual environment. In *VR '99: Proceedings of the IEEE Virtual Reality.*, Washington, DC, USA, 1999. IEEE Computer Socienty.

Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. Icrafter: A service framework for ubiquitous computing environments. In *Ubicomp 2001: Ubiquitous Computing*, 2001.

Virpi Roto, Andrei Popescu, Antti Koivisto, and Elina Vartiainen. Minimap – a web page visualization method for mobile phones. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 17, pages 35–44. ACM Press, 2006.

Johan Sanneblad and Lars Erik Holmquist. Ubiquitous graphics: Combining hand-held and wall-size displays

to interact with large images. In *Proceedings of the working conference on Advanced visual interfaces*, pages 373–377. ACM Press, 2006.

Von Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour. *The Jfc Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Professional, 2004.

M. Weiser. The computer for the 21st century. *Scientific American*, 265(94-104), 1991.

Aileen Worden, Nef Walker, Krishna Bharat, and Scott Hudson. Making computers easier for older adults to use: area cursors and sticky icons. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 266–271. ACM Press, 1997.

Index

Area cursor 23

Bubble cursor 23

Ergonomics 74
evaluation 69
Experiment Design 75
EyeTracking 5

Flow Layout 33
future work 78

Gyro mouse 71

iCrafter 16
Implementation 37

Large Public Displays 2
Low expressiveness input device 9

nearest neighborhood 34

Personal Universal Controller 17
Primitive Movement Vocabulary 8

Sampling rate 11
scroll bar 80
Selexel Glass Pane 37
Selexel Golden Rules 81
SelexelLayout 38
SelexelListener 38
Selexels 10
Semantic UI adaptation 79
SUPPLE 19
Sweep technique 4

Taxonomy for Input Devices 8
Technical Details 70

Ubiquitous Computing 1

Typeset October 5, 2006