LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department "Institut für Informatik"
Lehr- und Forschungseinheit Medieninformatik
Prof. Dr. Heinrich Hußmann

Diploma Thesis

SeCuUI: Secure and Fast Data Submission to Public Terminals Using an Auto-Complete Mechanism

Max-Emanuel Maurer
max@max-maurer.de

Bearbeitungszeitraum: 1. 2. 2009 bis 30. 6. 2009
Betreuer: Dipl. Medieninf. Alexander De Luca
Verantw. Hochschullehrer: Prof. Dr. Heinrich Hußmann

Abstract

Security at public terminals is an often discussed issue today. Many different ways exists to steal private data from someone using such a terminal. This thesis tries to present a new approach to this problem, using the mobile device of the user to enhance input security. SeCuUI – Secure Custom User Interface – is a software suite consisting of a client software and a framework to develop server applications. The client can be used with any public terminal running a software based on the SeCuUI framework. Using a device protected from modifications in the close proximity of the user makes it much harder to spy on any data she enters. To speed up the process the client remembers previously entered values and proposes them in future sessions. The framework and the client have both been evaluated in a user study using a demo-server based on the framework. This thesis contains explanations to all work done on the software as well as information on the planning, conduct and outcome of the user study.

Zusammenfassung

Die Sicherheit an öffentlichen Displays ist heutzutage ein oft diskutiertes Thema. Es gibt eine Vielzahl von Wegen an die privaten Daten der Nutzer eines solchen Gerätes zu gelangen. Die Diplomarbeit präsentiert einen neuen Ansatz die Sicherheit der Benutzer bei der Eingabe mit Hilfe des eigenen mobilen Endgeräts zu verbessern. SeCuUI – Secure Custom User Interface – ist ein Softwarepaket, bestehend aus einem Client für mobile Endgeräte und einem Framework zur Entwicklung von Serveranwendungen. Die Clientanwendung kann an jedem öffentlichen Display genutzt werden, welches unter Zuhilfenahme des Frameworks entstand. Das mobile Gerät bietet durch seinen Schutz vor Modifikationen und seiner räumlichen Nähe zum Benutzer eine bedeutend größere Sicherheit vor eventuellen Angreifern. Die Clientanwendung merkt sich dabei zuvor eingegebene Werte und präsentiert diese dem Benutzer bei zukünftigten Eingaben. Sowohl Framework als auch Client wurden in einer Benutzerstudie mit Hilfe eines zuvor erstellen Demoservers evaluiert. Diese Arbeit befasst sich mit Erklärungen zur gesamten Arbeit an den Softwarekomponenten, sowie mit der Planung, der Durchführung und den Ergebnissen der Benutzerstudie.

Thesis Topic

LMU München
Institut für Informatik
LFE Medieninformatik

Topic for a Diploma Thesis in Media Informatics

Student: **Max-Emanuel Maurer**
Matriculation-Nr.: **2113273**

Title: SeCuUI: Secure and fast data submission to public terminals using an auto-complete mechanism

When interacting with public terminals, users are exposed to several privacy threats. Private data can be lost or stolen by passersby, onlookers or any kind of attackers.
In this thesis, this problem is handled by moving input from the public terminal to the users' mobile device. To deal with the problem of reduced input speed, the system includes an automatic form filling functionality that assists the user in filling out forms. Based on previous work, the student will implement a system (including the appropriate APIs) that enables service providers to easily create and deploy such privacy enhanced services. An exemplary service will be implemented and evaluated in an user study.

Responsible Professor: **Prof. Dr. Heinrich Hußmann**
Supervisor: **Dipl. Medieninf. Alexander De Luca**

Start: **February 01 2009**
End: **June 30 2009**

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, 25. Juni 2009

. . . . . .

Contents

1 Introduction

1.1 Motivation

Over the last years the number and variety of vending machines has increased. When the first automatic teller machine (ATM) was installed in 1939 in front of a New Yorker bank, it was dismounted six months later because nobody used it [21]. But with the first electronic ATM installed in London in 1967 [22] the ATM concept finally took off on its triumphal course.

Until today the number of ATMs worldwide increased to over 1.7 million. But it is not only automated teller machines people use these days. Nearly everything that was once offered by someone behind a counter, is also available today at a certain machine.

This gives rise to frauds or misuse such as stealing money or private data. The most common concept to protect people using ATM machines today is a secret 4-digit number called PIN (personal identification number) that has already been introduced together with the first ATM in 1967. Today a large number of attacks exists to get hold of the card data and the according PIN number [14].

Over the last years many different methods have been proposed to make the input at those machines safer. This related work is covered in chapter 2.

Especially the variety of different modifications that can be made unnoticed at such a machine is a problem. This is why SeCuUI tries to move user input closer to the user's proximity by using a mobile device most people carry with them today. These days nearly everyone possesses a mobile device, like a mobile phone or anything similar that can be used for external data entry. The number of mobile phones worldwide breached the 4 billion mark at the end of 2008 [13]. In 2008, 60.1 million mobile devices were in possession in Germany [36].

1.2 Goals

The main goal of this thesis was to find a method to increase input security at public terminals by using a mobile device many people carry with them today. To fulfill that goal a client software running on a customers mobile device and a framework helping software developers to build applications for mobile terminals, was developed. A special focus of the thesis was placed on optimizing non-functional requirements like speed, security and simplicity. To test the software a user study was conducted to find out how people would react to such a system. When building the applications, modularity was very important. Modularizing the system made it possible to enhance it in the future. In general one can break down the goals of this work to a list of nine items:

1. Review other systems and research work that has been done so far and find out about benefits and disadvantages of those.

2. Create a client application capable of running on most mobile devices and being able to allow the user to connect to possibly any public terminal she wants to use.

3. Create a corresponding application framework for developers and enable them to develop applications for this certain client application most easily.

4. To be able to maintain the biggest flexibility throughout the development process, important parts of the program should be created in a modular way. Especially thinking of possible hardware components involved. The possibility for connection modules and connection types should be given.

5. When developing a server application using the framework users normally should not be forced to use the mobile device. Anyone not having a mobile device or anyone who does not want to use the system in a secure way should still be able to do so even if having less security through this.

6. Develop a test server that demonstrates how to use the framework and create an application with it. This application should also be used to evaluate the whole concept.

7. Try to bring the idea of an auto-completion concept, used with most internet browsers today, to the mobile device application, to make it a personal data safe. While this should increase input speed a lot, it is important to not let the client application give out any security relevant data to a server application without the user's knowledge.

8. Conduct a user study evaluating different aspects of the test-server application and the mobile client. Finally the study should try to prove the acceptance and relevance of such a system.

9. Finally analyze the results of the study and insights gained during development of the applications and give a conclusion and a future outlook to this field of research.

1.3 Structure of this Document

In this chapter the introduction and motivation to the topic was explained. The rest of the document is structured as follows: Chapter 2 gives an overview over different technologies that have been proposed to make input on public terminals more secure. Chapter 4 gives an insight on the different hardware and software components that have been used to create SeCuUI. The application itself consists of two big parts: First a framework that can be used by programmers to build their own server applications, which is explained in chapter 5. Second a client application featured in chapter 6 that is used on the mobile device to connect to the different servers. The complete system has been evaluated in a user study that is explained in chapter 7. Finally a conclusion can be found in chapter 8.

2 Related Work

In this section the different related work is described that has been published in the field of data submission on public terminals. One can distinguish three different groups of work. Much of the work done in the field just focuses on the most important part of data entry, namely entering passwords or PINs. Some of this work is described in section 2.1. Another group of work tries to delegate some part of the user interface and input to a mobile device (see section 2.2). A third group of related work tries not to rely on common input and output mechanism and uses other means like biometrics for legitimation. This group is discussed in section 2.3. Since SeCuUI needs to establish a connection between a mobile device and a public terminal some possible connection methods are presented in 2.4. At the end of this chapter – in section 2.5 – it is stated in how far SeCuUI differs from the work done so far.

2.1 Password Entry Methods

The most primitive attack on a password or PIN is the so called “shoulder surfing”. This means that someone is just standing beside the person, who is entering her password and tries to spot the different letters – or digits – she is entering.

Two of the following principles try to make the password entry process more complex by using a cognitive trapdoor game. Although such a method makes entering a password a little bit more complicated, it gets nearly impossible for someone watching the scene to guess the right password. In fact someone who places a camera somewhere near the PIN-pad to record what was entered, will still be able to reconstruct the password. Like this it would be possible to play back every second of the entry process over and over again. As two examples the Black and White PIN Pad (see section 2.1.1) and the Spy-Resistant Keyboard (see section 2.1.2) are presented below.

Another way of changing the password entry process is to use graphical passwords instead of passwords consisting of letters and digits. One form of graphical password techniques present the user with a set of images from which she has to pick the ones she previously selected. Other approaches require the user to reproduce something that was specified earlier (e.g. a sketch). A good example for the recognition approach is the Shoulder-Surfing Resistant Password Scheme described in section 2.1.4. As an example for the recall-based approach the pressure-based graphical password is explained in section 2.1.3. Suo et al. present a good survey on all the different methods in their paper [40].

A last group of password entry methods uses the tactile channel to submit additional information to the user without others getting to know it. VibraPass – described in section 2.1.5 – uses mobile phone vibrations to signal the user she should tell a lie. The authentication via tactile PIN entry uses pins raising out of a surface. This approach is described in section 2.1.6. A similar approach uses a mixture of image recognition and a rotating trackball to permute the order of the answers (see section 2.1.7).

2.1.1 Black and White PIN Pad

Volker Roth et al.[32] developed a virtual PIN pad on a touch screen that uses a kind of indirect input for numbers. The displayed keys from 0 to 9 are randomly colored half of them black the other half white. To enter a PIN the user just tells the machine whether the current digit is colored black or white. After multiple inputs for the same number the ATM machine can determine the right digit by intersecting the different colored sets. An example for one state of the PIN pad can be found in figure 2.1.

To make it even harder for attackers, the system is able to display several rounds in a row before giving the user the possibility to enter something. She then has to remember the sequence of colors in the background of the digit.

To create a certain resilience against recording with a camera, Roth et al. propose to play all but one round of the trapdoor game. With this the ATM as well as a possible attacker are left with two different possibilities per digit. Now someone recording the whole scene gets 16 possible PINs in case of a 4-digit PIN for example. On one hand this makes it harder to enter the right PIN out of the possible PINs on the other hand it gets easier to guess a PIN that will be accepted though it is incorrect.

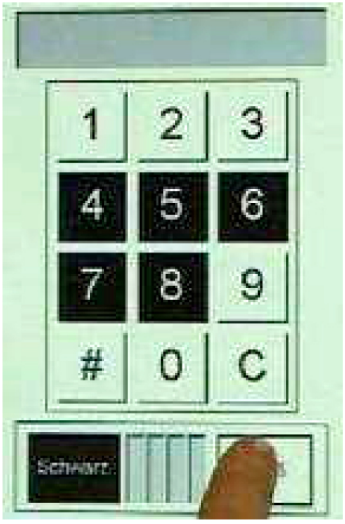

Figure 2.1: An example of the Black and White PIN Pad [32].

2.1.2 Spy-Resistant Keyboard

Another similar approach presented by Tan et. al. is the Spy-Resistant Keyboard [41]. They present an on-screen method that is suitable for entering full passwords. The system works by similar means than the black and white PIN pad (see section 2.1.1). On-screen the spy-resistant keyboard is represented by multiple lines of buttons. Each button randomly has several characters on it. One of them is highlighted by an underscore. This is called the “shift state”. To enter a character the user first looks for the button the character has been randomly placed on. Then she changes the shift state of all buttons until the respective character is underlined. Now the user can pick and drag an “interactor” towards the correct character tile. When she starts dragging, all characters are removed from the screen and nothing is shown on the tiles. Only the user now still nows where to drop the interactor. For an example of the spy-resistant keyboard see figure 2.2.

2.1.3 Pressure-Based Graphical Password

Malek et al. [20] presented the Pressure-Based Graphical Password. Instead of entering information via a keyboard the user’s password is a certain drawing on a grid with eight rows and eight columns. On a touch screen the user connects some of the dots to lines and draws his password shape to the screen. Additionally to shape the pressure that is used, while drawing the lines, is measured. This information cannot be seen by someone watching the process. Figure 2.3 shows an example of such a “passgraph”. When the drawing is displayed on the screen the different pressure is not shown.

2.1.4 Shoulder-Surfing Resistant Password Scheme

Sobrado and Birget created a system based on recognition rather than on recall [44]. They call it a “convex hull click scheme”. The system is safe against shoulder surfing or filming the input

Figure 2.2: The Spy-Resistant Keyboard by Tan et. al. [41].

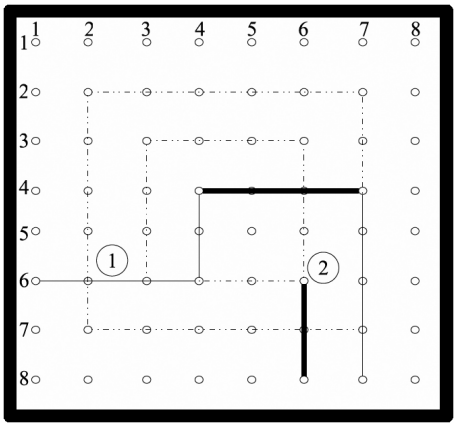

Figure 2.3: Example of a passgraph used by the Pressure-Based Graphical Password [20].

since users never point directly to the tokens that form their password. When entering a password with their system it displays several image-icons. Some of those icons have a special meaning to the user as they have been preselected by her. Those icons are called “pass-icons”. To enter a password the computer randomly lays out a large number of image-icons. Between them three or more pass-icons are hidden. The user now simply needs to click somewhere inside the area defined by her pass-icons. An example for such a convex hull created by three pass-icons is shown in figure 2.4.

2.1.5 VibraPass

In 2009 De Luca et al. presented a password input method that uses the user’s mobile phone to provide her with secret hints using the phones vibration alert [8]. The user connects his mobile phone to the terminal before entering the PIN. Before entering a digit the phone either vibrates or not. On vibration of the phone the user is not meant to enter the current correct digit of his PIN. Instead, she should enter a lie that is then discarded by the system. Someone filming the entry process does not know which digit belongs to the real password and which one was a lie.

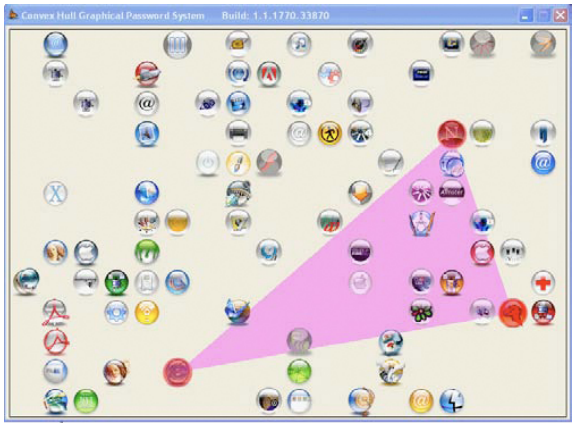

Figure 2.4: A convex hull created by three pass-icons [44].

2.1.6 Tactile PIN Entry

The tactile PIN entry concept presented by Roth and Deyle [10] works with eight pins mounted beneath a board. The pins can be raised by an electromagnet. They are arranged in a way that it is possible to put a finger on each of the pins. The thumbs are used to trigger two different buttons. A password now consists of a specific sequence of the fingers lying on the pins. To enter this sequence the computer raises a set of pins and the user has to tell whether the pin below the current "password finger" is raised. She does this by using one of the buttons she has beneath her thumbs – one of them meaning "yes" the other "no" –. The computer can calculate which PIN is meant by comparing the raised or lowered sets of pins. For an example of the system see figure 2.5.

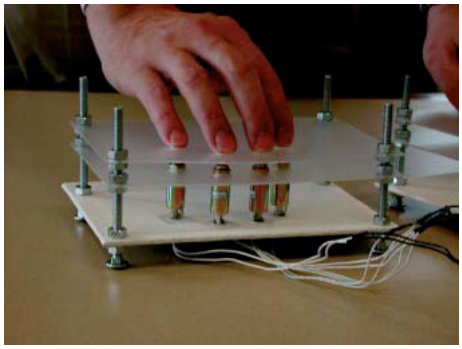

Figure 2.5: The tactile PIN entry system [10].

2.1.7 Undercover

The "Undercover" prototype by Sasamoto et al. [33] uses a combination of an image based recognition approach and an additional trackball system installed in front of the system. On a computer display a row of images is displayed and the user has to say which of those images belongs to an image portfolio she has selected as her password. Entering this immediately on a keyboard would show an attacker which of the images was selected and he could reproduce this in the future. To make it safer to enter the choice Sasamoto et al. built a device that has a trackball that is able to rotate in one of four directions. As a fifth state it can also vibrate. The user covers this trackball with her hand – hence the name "Undercover" – and depending on the direction of rotation she changes the input order when pressing one of five buttons to indicate which picture belongs to his portfolio. A picture of the device is shown in figure 2.6.

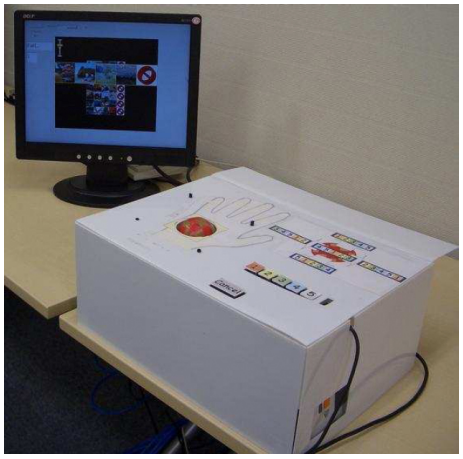

Figure 2.6: Undercover: Depending on the trackball, the user has to press different buttons [33].

2.2 Mobile User Interfaces

Ever since mobile devices were able to access networks, researchers tried to use them to interact with bigger terminals. Such connections can also be used to enhance the security for people using those terminals in public. One solution for this has been presented by Sharp et al. and is described in section 2.2.1. Other researchers not always presented a specific software solution with their work but thought about general problems. Hutchings and Pierce [18] conducted a study on how to divide interfaces best onto several devices. Together with Mahaney, Pierce even created another paper that shows several strategies in this field [30]. They describe possible methods to annex devices, divide interfaces, grant access and finally present a possible software architecture. Claycomb and Shin [4] focus on the security of such systems. For example, how to establish a secure connection between a mobile device and a public terminal via a colored two-dimensional barcode. Berger et al. [1] built a demonstration with a device called WatchPad that is more or less a display worn on the user's wrist. This display is used to display specific security related words or names that have earlier been blurred on a bigger screen. Myers [25] uses a Palm handheld device to extend the computers functionality to it. Like keyboard and mouse, the handheld is used as a third device controlling application specific behavior. As applications he presents a "slideshow commander", an application displaying the screen contents, or an application called "shortcutter" that displays different sets of buttons to control system applications.

2.2.1 Secure Mobile Computing

Sharp et al. [34] present a working system based on thin-client technology. First the mobile device is connected to the public terminal. After that several security related modifications can be enabled. The screen contents on the terminal screen can be blurred or modified in a different way using image processing filters. The part around the mouse cursor is then displayed unmodified on the mobile device. The input controls can also be deactivated or restricted on the terminal. Like this it is still possible to point using the mouse to reach a location on the screen but mouse clicks can only be done with the mobile device. Figure 2.7 shows an example of a blurred screen content being revealed on the mobile device.

2.3 Other Approaches

One can differ between three different kinds of authentication techniques [41].

1. **Token-based authentication** requires the user to own a special token. This could be a card with a magnetic stripe, a key or something else.

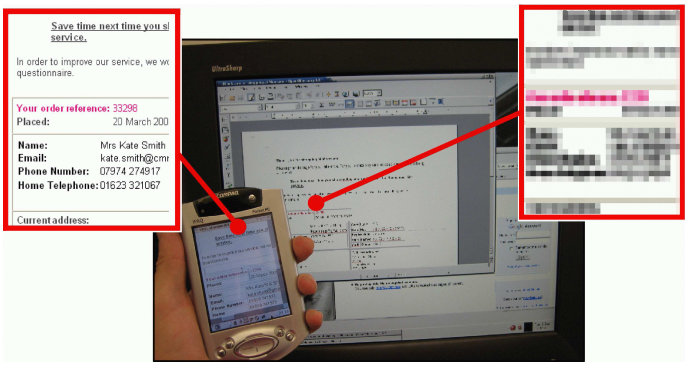

Figure 2.7: Secure Mobile Computing: Blurred screen contents only visible on the mobile device [34].

2. **Knowledge-based approaches** ask the user for a secret that only she knows. Normally this is a password or PIN.

3. The last group of authentication techniques relies on the uniqueness of the human body. With **biometric authentication** the user's finger or retina is scanned to proof his eligibility.

This section contains three different biometric approaches: Coventry et al. focussed especially on the biometric verification at ATM interfaces. Their work is summarized in section 2.3.1. Kumar et al. tried to use eye-gaze to operate an on-screen keyboard (see section 2.3.2). Gazing was also researched in several cases by Drewes et al. (see section 2.3.3).

2.3.1 Biometric Verification at ATM Interfaces

In 2003 Coventry et al. published the results of a big study they conducted to understand the usability of biometric authentication systems [5]. They conducted focus groups and surveys and found out that users perceive no need for biometrics at an ATM interface. They are even scared about eventual health risks of such devices. Finally they did a 6-month field trial and established an ATM with iris verification at a mayor bank in the UK. Over ninety percent of the participants were satisfied with the system after they used it for some time.

2.3.2 Gaze-Based Password Entry

Kumar et al. tried to attend to the problem of shoulder surfing by creating an on-screen keyboard that is controlled with the user's eye-movement [19]. A screenshot of the system can be found in figure 2.8. They calculated that the size of a target on a 1280x1024 pixels screen with 96 dpi should be at least 66 pixels and chose buttons with 84 pixel size for their experiment. They also tried different trigger methods. For a dwell-based method the user had to stare for some time on a point until it was "clicked". With a "point-and-shoot" method the user triggers the click manually with a special key (e.g. the spacebar). They found out that a dwell-based approach produces significantly less errors.

2.3.3 Eye-Gaze Interaction for PIN-Entry

A group of researches around Drewes did some research work on using eye-gaze for several things including PIN-entry. Two studies dealt with using gaze-gestures to control terminals or mobile phones [11] [12]. Another approach used gestures denoting digits to enter numeric PINs (see figure 2.9). For gestures like this, a cheap eye tracker build out of a normal webcam and an infrared LED was already enough [7].

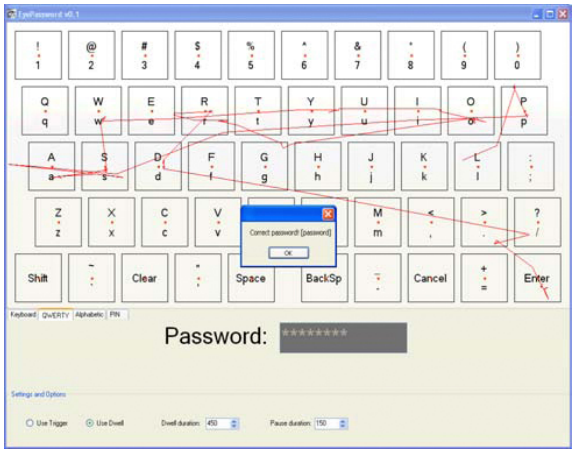

Figure 2.8: A gaze-pattern produced with the Gaze-Based Keyboard [34].

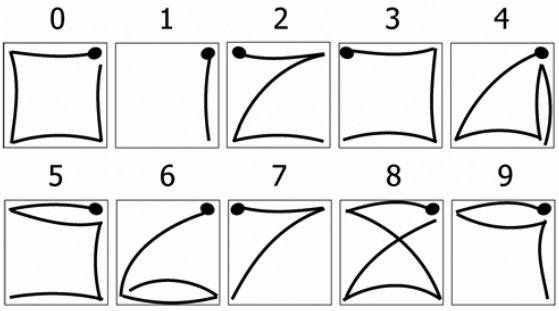

Figure 2.9: Entering a numeric PIN using gaze-gestures [7].

2.4 Connection Methods

When thinking about connecting a mobile device to a public terminal it is also important to think about how to establish that connection. For SeCuUI five different possibilities have been considered:

- Usually when connecting bluetooth devices today the user performs a scan of the bluetooth environment and is presented with a list of the available bluetooth devices in her neighborhood. From this she has to select the desired device and perform a handshake procedure. For SeCuUI such a procedure would have been to complicated and would have taken too long.

- Another possible method would have been to replace the standard names or network entries in such a discovery list by images representing the device. A terminal could constantly acquire those ID-images automatically and then display them whenever a user wants to connect.

- A third idea would be to read out the bluetooth address of an NFC-tag. Such a near field communication tag can store small amounts of data. With this the user would touch the tag with his mobile device and the connection procedure would start. A problem with this is, that most mobile devices produced today are unable to read those NFC-tags. And a tag like this would increase the possibility for fraud since it could be exchanged with another one that would then redirect the connection to a malicious device.

- The idea of a button press connection called SyncTap is explained in section 2.4.1 below.

- The last idea that was finally adopted for the prototype application, is the idea of using two-dimensional barcodes. This is explained in more detail in section 2.4.2.
- A connection technique presented by Holmquist et al. [16] was quickly discarded because it involved shaking the respective devices together.

2.4.1 SyncTap Connection

SyncTap is a connection method introduced by Rekimoto et al. [31]. Its basic idea is to quickly create a network connection between two devices in physical proximity. To perform the connection the users just needs to press a "SyncTap"-button simultaneously on both devices. The press-release time is broadcasted to all devices in range and the two intended devices find themselves automatically by comparing those values. Collisions of multiple devices with the same parameters would need to be discarded. This method is very easy but for the use in SeCuUI it has two disadvantages: None of the currently sold devices has a "SyncTap" button. So another button on the devices would have to be used. Another problem with this connection method is, that a button mounted to the public terminal could again be modified by an attacker to redirect the connection process to a malicious device.

2.4.2 QR-Code Connection

The connection method chosen for the prototype of SeCuUI is a connection method based on QR-Codes by DensoWave [9]. QR-Codes are two-dimensional codes that consist of a matrix of black or white dots. "QR" stands for "quick response" because the codes have been designed to be quickly decoded. Depending on how much information needs to be stored, QR-Codes can be produced for nearly any data capacity. When creating a QR-Code different levels of error correction can be applied. Figure 2.10 shows a QR-Code with its different sections used to recognize the code correctly [47].

When establishing a connection via a QR-Code the network address of the public terminal is encoded in the barcode and then displayed on the public terminal. The user takes a picture of this barcode which is decoded on her mobile device. After that the connection can be established.

Claycomb et al. used a colored two-dimensional tag called UbiColor to transfer connection information [4]. A colored tag on the one hand can store more data on one single pixel but on the other hand reading it out from different monitors with different types of cameras gets harder due to the variety of different devices that exist.

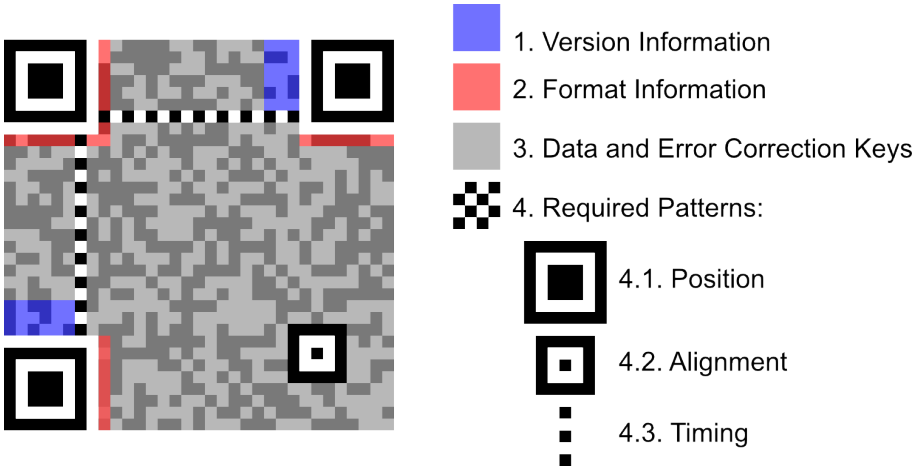

Figure 2.10: Example of a QR-Code [47].

2.5 Separation of SeCuUI

The research in the field of public terminals that has been done so far follows either a macroscopic or microscopic approach. The different password entry methods described in section 2.1 try to make the entry of a single password or PIN safer. In some cases it is possible to make the password entry process complex enough that shoulder surfers are unable to recognize what was entered. In other cases the password entry process can even be filmed without getting to know the password. All those approaches raise the complexity of the input but can only serve one single purpose. SeCuUI offers a more secure input alternative for every kind of input but serves also as an input method for passwords and PINs while reducing the possibility for shoulder surfing or camera recording of the password to a minimum.

A macroscopic approach like the secure mobile computing described in section 2.2.1 is able to make controlling of complete terminals more securely. SeCuUI in contrast focusses just on the possibility to fill out forms on public terminals and to control their behavior. Like this it is possible to show all functional elements of the server while still rearranging them in a manner suited for the mobile device.

Also research on biometrics tried to make entering password or controlling terminals more securely. Though this is a promising field of research additional hardware is needed to use such systems in public. SeCuUI just relies on hardware that most devices already have or that can be easily and cheaply refitted.

What makes SeCuUI so promising is that it is a happy medium between the macroscopic and the microscopic approaches. SeCuUI is based on an approach called PocketPIN that has been researched at the department [6]. PocketPIN already followed the idea of entering passwords with a mobile device. A small sample application had been built and a user study with this application was conducted. A major difference of this prototype was that it was less modular and that participants where offered the possibility to preselect the values they wanted to enter in a secure way. The study showed that this preselection phase costs the user a lot of time.

3 SeCuUI

This section gives a general overview about SeCuUI and what it is. Section 3.1 demonstrates how easily a user can use the system to make her input on public terminals safer. SeCuUI is made out of two big parts, a framework and a client application. Each system is explained in an extra chapter. For details on the framework refer to chapter 5. The client application is covered in chapter 6.

This chapter itself covers a non-technical introduction to the overall concept of SeCuUI. Section 3.1 is concerned with the question "What is SeCuUI?". It provides an overview on the concept of the application suite. Section 3.2 shows the three easy steps a user needs to perform when using SeCuUI for terminal input. Section 3.3 introduces the XML user interface concept, which is essential using the SeCuUI framework and section 3.4 gives a first glance on the special auto-complete feature used to speed up the user's input actions.

3.1 What is SeCuUI?

SeCuUI stands for **Se**cure **Cu**stom **U**ser **I**nterface. The idea behind SeCuUI basically is to make input at public terminals more secure using the user's mobile device. A public terminal in this case could be any type of vending machine, an ATM or another machine the user needs to input data to. Normally at least some of this data is privacy related. This means the user either has to enter a password, PIN or some credit card information that could be valuable to someone else. Using a mobile device like the user's phone to do this would reduce the possibility for fraud. Most of the attacks on public terminals rely either on the idea to modify the terminal somehow to capture what the user enters or simply on watching the user from nearby. Modifications to the terminal can be done with additional key-pads that log what is entered or a miniature camera placed above the keypad to record what is typed.

Using the mobile device of the user to enter data removes the necessity of using the integrated input method of such a public terminal. Hence cameras or additionally mounted keyboards would record nothing valuable. This kind of attacks also relies on the fact that the public terminal normally is unattended for some time, an attacker can use to install his equipment. The mobile device of the user is carried always with him and therefore it is nearly impossible to modify this device.

Another important advantage of using a mobile device is that its properties make it ideal for the input of secure data. Mobile devices are always hold in close proximity to the user with their display and input facing to her. The usually small display size – compared to a public terminal's screen – even makes it harder to spot any information intended for the user.

Like this, one can see quickly that a mobile device offers many advantages when it comes to entering secure data. Besides the possibility of input the phone also offers the possibility for a more secure output of information. Most ATM machines for example offer the possibility to display the user's bank account balance. This data displayed on the public screen is clearly visible for anyone standing beside the user. With SeCuUI this data can be hidden on the public terminal and instead be displayed at the user's device screen.

To achieve all those advantages this thesis presents the SeCuUI software suite consisting of two parts. An application the user can install at his mobile device to input data (the client-application) and a matching developer framework. This framework makes is possible to write applications for public terminals that can handle the mobile phone's input.

A possible refutation for this system could be the fact that there are still people using public terminals that do not want to deal with a new kind of technology, do not own a mobile device or even do not worry about their privacy at those systems. To avoid those problems building an application with the SeCuUI framework normally results in an application that does not force the user to use a mobile device. She can connect one every time she worries about her security, or use the old-fashioned way when she does not want to. In certain cases the programmer is able to force the usage of a mobile device when thinking this is absolutely necessary.

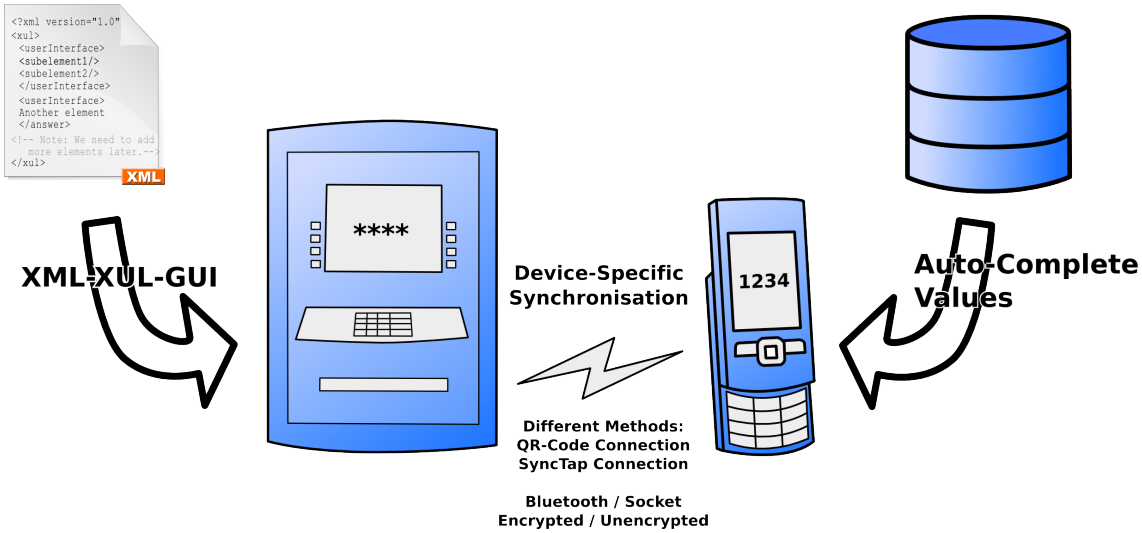

Figure 3.1: Schematic overview of SeCuUI

Looking at figure 3.1 the idea of SeCuUI is depicted. To connect a mobile device SeCuUI offers a very modular way of different methods. Like this the connection procedure can adapt to eventual hardware requirements of the mobile device or the public terminal. In addition to the type of connection the way data is transferred – e.g. using bluetooth or a socket connection – and the encryption can be chosen. For example an optional two way encryption can be used in case the user's device has the necessary computing power.

Two other things make SeCuUI special. User interfaces displayed on the public terminal are not only sent to the mobile device in a way to better fit on the small screen, they are also dynamically created out of XML files on the server. Like this, additional parameters, like where to show which value, can be directly incorporated during the creation of the user interface. Section 3.3 gives a first introduction to this.

The second thing making SeCuUI not only more secure but also faster is the fact that SeCuUI reuses previously entered data. This data is stored on the mobile device and accessed every time the user connects to a public terminal. To reduce the number of displayed items for each user interface component a data type is specified, which has to match one of the values stored on the device. A first introduction to this is found in section 3.4.

How easily a server application for SeCuUI can be developed using the framework and how the framework works on the whole is explained further down this document in chapter 5. For the user of the client application – described in chapter 6 – it is even easier to use SeCuUI. She just has to follow thre easy steps.

3.2 The three Steps

For a user the basic process on a SeCuUI capable public terminal consists of three steps:

1. The user launches a specific application on her mobile device she previously installed. This application is a simple and small Java ME application an thus runs on nearly every of today's mobile devices. The user then selects an adequate connection method from a list. If his mobile device is equipped with a camera she can for example use the QR-Code connection (see section 6.2). Using that connection method the user simply takes a photo of a two-dimensional barcode displayed on the public display. The mobile device analyzes the picture and establishes a connection to the public terminal.

2. After the connection has been established successfully. The user interface elements of the

public terminal are synchronized with the device. Doing this the client does not show visually the same as the public terminal by transferring image data. Instead it displays device specific elements with the same functionality. Not necessarily all elements are synchronized. After the connection was established the public terminal decides, which elements are synchronized and how they should behave on the user's client.

3. Once the first synchronization finishes the user may control the user interface with his mobile device, changing values of the different fields or pressing one of the buttons on the remote device. Furthermore the user is not forced to do everything using his mobile device once connected. For some of the fields he can stick to the public terminal itself to enter those values. Every change is immediately synchronized with the other device what makes it possible to switch the input device at any time. Again the public terminal is in control how much freedom the user has. It may block some of the fields on the public terminal after a mobile device was connected or at least hide the contents of those fields by replacing them with asterisks.

3.3 XML User Interfaces

Separating user interface and data functionality has always been a good principle when programming applications. Not only the Model-View-Controller as an architectural pattern [46] proposes the separation of application logic and user interface, also many development APIs do so today. To perform this separation the SeCuUI framework relies on XUL. XUL is called the XML User Interface Language and was developed by the Mozilla Foundation [23]. With XUL it is not only possible to specify user intefaces using an XML language it is even possible to describe certain functionality. Another point why XUL is that well suited for the use with SeCuUI is the fact that as an XML language it is easily expandable. Like this it is possible to even control and not only define the user interface components that will be used on the public terminal and that can be possibly synchronized with the mobile device. It is also possible to define the behavior of those components in the system together with the component description itself.

This is achieved by introducing three different XML attributes that extend the normal XUL language. One to define where the content can be input called `security`. One defining where the content is visible called `asterisk` and a last one defining the type of content used with the auto-complete function. This attribute is called `dataType`. The technical side and further details on XUL are described in section 5.2.

3.4 Auto-Complete Using the Mobile Device

Entering data with a mobile device makes input to public terminals safer. But using a small keypad to enter text costs time. So does the connection process one has to go through before being able to use it. To reduce this additionally needed time to a minimum SeCuUI makes use of another concept called auto-complete. Today this is mostly used with internet browsers where people have to fill in the same terms over and over again. Since the HTML-standard does not officially mention this feature it is usually performed using the names that are assigned to input components in HTML. Sometimes it is even possible to produce a website-spanning effect in case website-authors used the same "name"-attributes for the same type of field.

To make this more effective SeCuUI uses a special `dataType`-attribute – just mentioned above – that defines the type of data for a certain field independently of its name. A catalog of predefined data types can be used by any programmer to conform to the standards. But as in HTML the attribute basically can be assigned any value making it possible to create also company specific behavior.

Another important thing is to keep the user's secret data still as private as possible when it is saved on the mobile device. One important thing is that the public terminal is never able to read

the stored values by itself. Whenever the user has stored matching auto-complete entries on her phone they are only displayed to her and she has to decide whether they are used as an input value or not. To protect the saved values from theft it would be possible to save them encrypted by a user password and only load them whenever the correct user password was entered. Like this the mobile device of the user becomes her personal data safe.

Technically the auto-complete function of SeCuUI is partially implemented in the framework, which is described in section 5.3. The bigger part of the implementation is done on the client side. How this works is explained in section 6.1.

4 Utilized Hardware and Software

This chapter covers details on the different hardware and software technology used for the prototypes of SeCuUI. SeCuUI consists of a framework which allows to build server applications (see section 5) and a client application identical for all created server applications (see section 6). The framework and the prototype server application have been programmed in Java SE. The client application has been programmed in Java ME using the MIDP 2.0 profile. Some parts of the code use third-party software. To communicate with the bluetooth stack of the server systems the BlueCove 2.1.0 library has been used (see section 4.1). The user interfaces used inside the framework are created reading out XUL files. The XUL standard and the modifications made to it are described later on in section 5.2. In this chaper the library SwiXml used to parse the XUL files is explained (see section 4.2). Section 4.3 describes XParse-J an XML lightweight-library that was used to parse the small XML files on the mobile device. For the framework the more sophisticated JDOM was used (see section 4.4). When developing SeCuUI an encrypted connection method was created too and can be optionally used. To encrypt and decrypt the network messages the BouncyCastle API was used. See section 4.5 for an explanation on this. For the decoding of the pictures taken of the two-dimensional barcodes the Google ZXing library was used (details in section 4.6).

All the tests of the mobile client were performed with a Nokia N80 device that is briefly illustrated in section 4.7.

4.1 BlueCove

BlueCove is a Java library that helps to communicate with different bluetooth stacks. It supports the most important platforms as there are implementations for Mac OS X, WIDCOMM, BlueSoleil, Microsoft Windows and Linux [2]. BlueCove is an implementation of the JSR 82 specification that has been developed under the Java community process [45]. With BlueCove it is possible to get access to the operating systems bluetooth stack and perform several actions: After acquiring the local device it is possible to start a remote device discovery, a services search or to act as an OBEX Put Client/Server.

The most important Classes of BlueCove are:

- The `LocalDevice`-class allows to get an instance of the local bluetooth device with its static method `getLocalDevice`.
- The `Connector`-class offers a static method to a `StreamConnectionNotifier` from which it is possible to get the next incoming connection with the method `acceptAndOpen`.
- To create a client side connection on a mobile device – communicating with BlueCove on the server side – with Java ME, the `Connector`-class in the `javax.microedition.io`-package is used.

4.2 SwiXml

SwiXml is a GUI generator for Java applications. The GUI is created during runtime, generating it out of XML code. XML documents can be created on runtime or previously created GUIs can be loaded dynamically depending on which type of GUI is needed [28].

For SeCuUI SwiXML was used to parse the XUL-XML-GUI files in a first run to render the server side GUI elements. To incorporate the additional SeCuUI specific XUL language parameters a second run on the same XML file is needed. More about this is explained in section 5.2.

4.3 Xparse-J

Xparse-J is a Java-port of Xparse itself. This is a very small JavaScript library to parse XML documents. After parsing a document with Xparse-J there are only two important Classes that are needed to explore the complete XML tree. The `Node`-class represents a node in the XML tree storing the attributes in a Hashtable. A `JSArray` named `contents` holds the children of a node. `JSArray` is a helper-class that behaves like the normal `Array`-class in JavaScript. Xparse-J is also capable of interpreting XPath expressions [3].

For SeCuUI all messages exchanged between client and server are small XML documents. Xparse-J is used to parse these on the mobile device to retain best performance while having only little code.

4.4 JDOM

For parsing the XML documents on the server side the well-known JDOM-API has been used. Due to Sun's trademark policies JDOM is not an acronym, so one has to guess what was meant with those four letters. To work with an XML document using JDOM the `SAXBuilder`-classes `build`-method parses a complete file and returns an instance of a `Document`-class. The document itself has a `getRootElement`-method that returns an instance of an `Element`-class. The elements represent the nodes in the XML tree. Children of the elements are accessed using the `List`-interface [17].

4.5 BouncyCastle

The BouncyCastle Crypto APIs offer cryptographic APIs for many different cases. Their lightweight cryptography API for Java not only works with Java SE but also works with Java ME, which had been important for this thesis. It is also available for C#. BouncyCastle is capable of many different and well-known cryptographic algorithms. Examples are the Advanced Encryption Standard (AES), the Data Encryption Standard (DES) or the asymmetric crypto system RSA [42].

SeCuUI offers the possibility to encrypt the complete data transferred between the mobile device and the public terminal. Two `RSAEngine`-instances are used to get a complete two-way asymmetric encryption.

4.6 Google ZXing

The ZXing or Zebra Crossing library is intended to process images showing different types of barcodes. The library can be used for different Java versions as well as for Android or iPhone development. The library is able to decode the following barcode systems: UPC-A, UPC-E, EAN-8, EAN-13, Code 39, Code 128, QR-Code, Data Matrix and ITF. For each of those barcode systems a decoding-class exists that implements the `decode`-function defined in the `Reader`-interface. The decode function is passed a `MonochromeBitmapSouce` and the decode function returns an instance of a `Result`-class [15].

Since SeCuUI uses QR-Codes (see section 2.4.2) the `QRCodeReader`-class coming with Zebra Crossing has been used to decode the displayed information.

4.7 Nokia N80

For the development of SeCuUI the Nokia N80 phone was used. An image of the phone can be seen in figure 4.1. Due to its technical specifications the phone was suitable for the tests with the mobile device client software. The phone is capable of Java, which made it possible to run the Java ME application on it. It has an integrated 3 megapixel camera with close-up mode. As operating

Figure 4.1: Picture of the Nokia phone N80 [26].

system the phone uses Symbian OS. It comes with a 352x416 pixel color display. To control our application the 4-way scroll key with center select, the two softkeys and the keypad that can be slid open [27] were used.

5 SeCuUI Framework

Applications running on public terminals always are very specific. But in most cases they need to acquire some data of a customer. After processing this data the client receives something in return. This can be money, a train ticket or some kind of information. To give programmers the largest degree of freedom when developing such an application SeCuUI offers programmers a framework they can use to develop own applications conforming to the SeCuUI standard. Applications created with this framework are automatically able to be connected by mobile devices. This section shows the structure of the framework and how to use it. First – in section 5.1 – it is explained how a programmer would use the framework in general. The XML User Interface Language XUL is important to the framework as well. All user interface elements of an application need to be modeled in this language to work properly with SeCuUI. Details on XUL are given in section 5.2. Section 5.3 explains the auto-complete function that is used with SeCuUI from the framework perspective. Section 5.4 explains the different parts of the framework in detail. A simple server application that is already able to communicate with the client would just take a few lines of code. Such an example can be found in section 5.5.

5.1 Working with the Framework

When working with the framework developers build a normal Java application as they would do without SeCuUI. When designing user interface elements of the application that shall be controllable with the help of a mobile device the programmers need to design them using XUL. The SeCuUI framework is able to render those files and passes a rendered component back to the application. The user interface can be placed anywhere the programmer wants. When rendering the components the framework keeps track of the different items making it possible to sync them at anytime to a mobile device.

To make it possible to block the user from filling in some fields at the public terminal additional attributes have been added to the XUL tags. One to control on which device the user may fill in text, another to control if the value of a label or textfield is visible and a third one controlling the behaviour of the auto-complete feature. This feature is described in section 5.3. The three attributes and their possible values are explained in section 5.2.1.

To instantiate a connection the programmer can either create an own interface element or he just places a predefined connection button that comes with the framework somewhere in his application. This button is offered by the framework, too. When clicking on the connection button a separate window opens and shows different connection methods. The programmer can select and configure all those connection methods but he does not need to take care of programming those features by himself.

While being connected the framework takes care of all synchronization issues. With the use of different listeners the programmer can keep track of what is happening inside the framework. If he does not want to use those listeners he can still refer to the classic listeners of the user interface components displayed on the public terminal.

5.2 XUL

Normally Java user interfaces are created by assembling instances of objects onto a form by code. Doing this is not very flexible and the classes used to do this do not allow to specify additional parameters needed for the SeCuUI behavior. The three special parameters used for the SeCuUI functionality are explained in section 5.2.1. To make the user interface creation more flexible and more expandable SeCuUI uses the XUL language to describe user interfaces.

XUL stands for XML User Interface Language and it was invented by the Mozilla Foundation [23]. Mozilla uses this language for their own platform. With XUL it is possible to design

whole applications. XUL is based on XML and like most XML languages it is platform neutral not specifying an operating system specific design. XUL separates the presentation from the application layer (similar to the Model-View-Controller pattern). Also different text for different languages is possible. Like this, one can easily localize applications**??**. A simple example for a XUL file is shown in figure 5.1.

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
3 <window id="findfile-window"
4         title="Find Files"
5         orient="horizontal"
6         xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
7
8   <button id="find-button" label="Find"/>
9   <button id="cancel-button" label="Cancel"/>
10
11 </window>
```

Figure 5.1: A simple example for a XUL file [24].

The example creates a window with two buttons. Each XML tag represents a component. The nesting of the tags shows, which component is placed inside which other container. The window-tag in this example creates a window with the title “Find files”. Inside the windows two buttons are placed. One button showing “Find” and another showing “Cancel” as its label. Using the id-attribute each component is assigned a distinctive name that can be used later to access its properties.

SeCuUI does not use XUL to build whole applications. But since XUL is able to model all kinds of user interface components and their layout on the screen it is possible to specify Java interfaces with it, too. SwiXml is a Java API that is able to read those XML files and create the according Java user interface. SwiXml has already been introduced in section 4.2. Since XUL is able to model a lot more things than needed for user interfaces in Java the SwiXml developers maintain a list of the tags that can be processed [29].

5.2.1 Modifications Made to XUL

To control the security-specific behavior of the different input components SeCuUI extends the XUL language with three new attributes.

1. **The `asterisk`-attribute:** This attribute can be added to labels, textfields or buttons and controls in which case the text of those components is visible. A typical use case for this attribute would be that someone checking his bank account balance does not want to see this information showing up on the big screen of the public terminal. But certainly it should show up if the user has not connected his mobile device to the terminal. This behavior can not be created by using any standard Java component. Because of this wrapper components have been created. They are part of the ui-package of the framework that is described in section 5.4.2. The `asterisk`-attribute therefore can have four different values:

   - `ASTERISK_NEVER`: Asterisks are never displayed. Neither on the public terminal nor on the mobile device.
   - `ASTERISK_LOCAL`: On the terminal side the fields content is never visible regardless whether the mobile device is connected or not. When a device gets connected the contents are shown on the display of the mobile device.
   - `ASTERISK_OPTIONAL`: Using this option would result in the behavior spoken of in the example above. The value of this component is displayed as long as there is no

mobile device connected. Connecting a mobile device hides this value. When a user connected in an earlier stage of the application the contents are certainly hidden before ever displaying anything personal.

- `ASTERISK_BOTH`: This option simply shows asterisks on both devices, such that the value can not be read anywhere.

2. **The `security`-attribute:** When the user connects her mobile device to a public terminal she can by default choose for herself where to fill out the different fields. Even the connection to the public terminal itself is not mandatory by default. But perhaps a programmer wants to force someone, who has established a connection with the public terminal, to enter his PIN with his mobile phone. This is what the `security`-attribute is used for. This attribute works on any input components or any components that can be modified by the user. Examples for such components are textfields, checkboxes and buttons. To use the `security`-attribute on labels makes no sense. To hide the contents of a label the `asterisk`-attribute is used. Just like the `asterisk`-attribute the `security`-attribute can also be set to four possible values:

   - `SECURITY_INSECURE`: Insecure components are shown only at the public terminal. They are not sent to the mobile device and can not be filled in there because they do not even appear.
   - `SECURITY_BOTH`: Components with this `security`-attribute are displayed on both devices. The user can selected where to fill in the text for this field or where to use the button with this attribute value.
   - `SECURITY_SECURE_OPTIONAL`: A secure-optional field behaves the same way the insecure or both field would do as long as no device is connected to the terminal. The user can always modify the value of those components. As soon as the user connects her mobile device to the terminal all `SECURE_OPTIONAL` fields are disabled at the terminal. This makes it possible to enhance the security for users that have a mobile device with them by forcing them to use it for the input while users carrying no device with them still are able to use the more insecure solution.
   - `SECURITY_SECURE_FORCED`: Whenever one field on a screen is assigned this value, it will affect the whole application. This option forces the user to fill in this field with his mobile phone. Without having one, the user will not be able to complete the form. Because of this, as soon as one component has this `security`-attribute all fields are blocked until a connection to a mobile device has been established. After the user connects to the terminal this field will remain disabled on the terminal but he can fill out the field using the connected mobile phone. All other fields are set to the state according to their own `security`-attribute.

3. **The `dataType`-attribute:** SeCuUI remembers previously made entries of the user and offers a list of those when connecting the same device somewhere else. Due to security-reasons this auto-complete feature is implemented on the client side and is explained in section 6.1. A short introduction to this feature from the programmers point of view is given in section 5.3. Most important for this feature to function properly is that the client knows which type of value the user is asked to enter in which field. This is what the `dataType`-attribute is meant for. The `dataType`-attribute holds a value denoting which type of content should be entered in the field. SeCuUI gives a standardized list of possible types that is included in the appendix as figure A.3. Two examples of such values would be `TYPE_ADDRESS` or `TYPE_VISA_CARD_NUMBER`. These values can be used by any programmer that develops a SeCuUI-based application and they ensure that even applications of different companies can bring up the same auto-complete values on the user's device. In fact such a list can

never be complete, this is why any value given to this attribute that is not part of the list is also accepted and then denotes a company specific value. For future changes all values beginning with "TYPE_" are reserved for future use.

5.3 Auto-Complete Feature

When working with the framework the programmers do not have to care much about the auto-complete feature. When designing the front-end using XUL (see section 5.2.1) it is just necessary to provide the correct `dataType`-attribute values in the XML file. This attribute is parsed and transferred to the client application. To enhance privacy and security for the user, the framework itself offers no possibility to get access to the saved values stored on the user's phone. Only if a user selects a specific value, she wants to enter into the form, it will be synchronized with the server application. Programmers do have one possibility to modify the auto-complete behavior of the client: When creating a new instance of the framework they can disable the auto-complete function. When a connection to a mobile device is established with auto-complete disabled the user will be forced to enter everything manually even when auto-complete values for the according type of data are stored on her phone. A more detailed look on the auto-complete function and how it works on the client side is given in section 6.1.

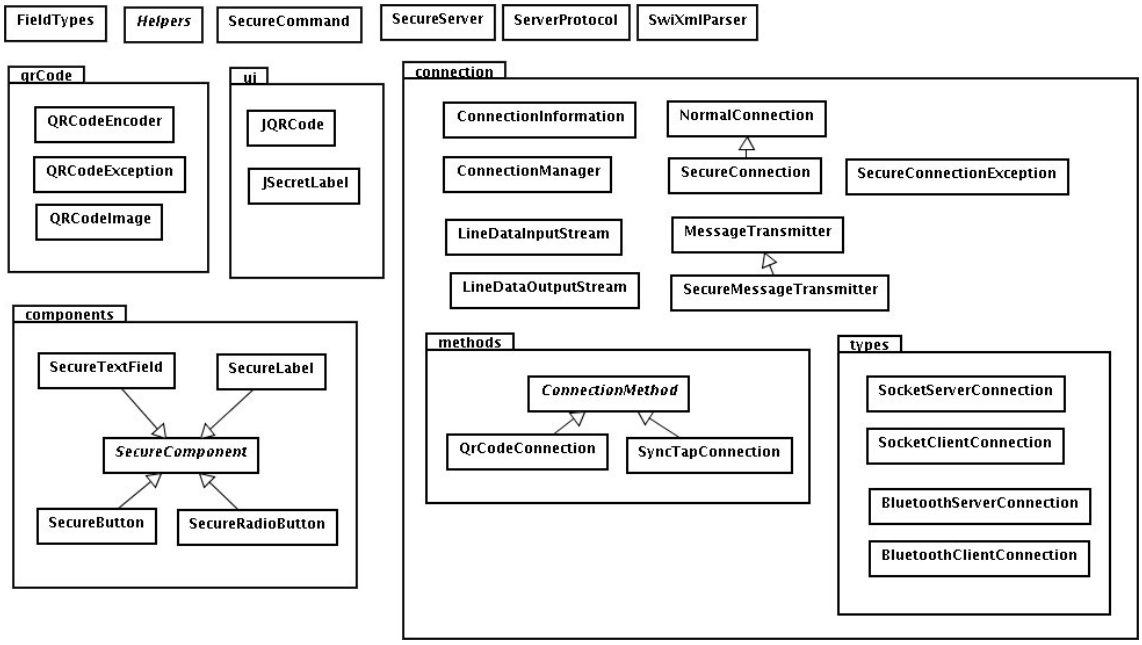

Figure 5.2: A UML package diagram showing the framework structure.

5.4 Framework Components

This section explains the different framework components the SeCuUI framework consists of. Figure 5.2 shows the package diagram of the framework. For programmers working with it, only the classes of the framework-package (the "main" package) itself matter. All other classes in sub-packages are used by the framework-classes for different reasons. In the following, the different packages and their classes are explained. A complete UML diagram containing all classes with respective methods and variables is located at the end of this document (see figure A.1).

5.4.1 framework-package

The framework-package contains the classes programmers need to bring SeCuUI-functionality to their software. To create a SeCuUI server-application using the framework is quite easy. Everything starts with an instance of the `SecureServer`-class. A UML diagram of this class is shown in figure 5.3.

Figure 5.3: The SecureServer-class.

The **SecureServer**-class (see figure 5.3) uses three different constructors. When creating a new `SecureServer` instance the programmer can specify if this secure server uses the auto-complete feature or not. With the second value he can define, which connection method the server uses as the primary connection method. Available connection methods are denoted by public strings contained in the `SecureServer`-class. The connection-method-system works with modules. So far only one working module is included in the framework. The "QR-Code connection" denoted by `CONNECTION_QR_CODE`. To test the module system another dummy connection is included named `CONNECTION_SYNC_TAP`. When the user later initiates a connection a dialog-window appears. This window can contain multiple connection methods at once. To add more than one connection-type the method `addConnectionMethod` can be used.

The next step after creating the server is to parse a XUL file containing XML code of a user interface. XUL has been explained in section 5.2. To do that different methods exist. The `parseXul`-method accepts a simple String denoting a file name, a `File`-object referring to a specific file or an XML document parsed into an `org.jdom.Document`-class. Passing one of those three parameters to the method makes SeCuUI parse this XML structure and create the corresponding Swing-architecture. This architecture is then returned as one component. In case the XUL document specifies a complete frame instead of a plain set of components, only the inner portion of the frame is returned. The set of components that has been parsed last can be always accessed

by using the `getCurrentComponent`-method. This component can be placed anywhere inside an existing window of the programmers application. New XUL files can be parsed at any time even when the user is connected. The parsed contents will then be resynchronized with what the user just entered.

To control the behavior of the different single components included in the XUL file they can be referenced by their XML ID assigned to them. The method `getIds` returns a `Vector` containing all IDs that have been successfully parsed. To get a container according to one of those IDs the `getContainerById` method is used. Like this it is possible for the programmer to add EventListeners to buttons or other components, they just added with XUL.

A more advanced method is to use the `getSecureComponentById`-method. All Containers are enclosed in a `SecureComponent`-class that stores additional parameters like `dataType` and `security`-attributes. More about this class is explained in section 5.4.5.

To initiate a connection with a mobile device the programmer has two possibilities. One of them is to use a predefined connection button coming with the SeCuUI framework. Using the `getToggleButton`-method the programmer can place a `JButton`-component anywhere in his own user interface. This button already provides all functionality needed for SeCuUI. It changes state text and icon, depending on the current connection state and displays a connection manager whenever the button is pressed. When a connection is established the same button is used to terminate that connection again. In case the programmer does not want to use the prefab-button he can design the functionality using his own button. The method `getConnectionEstablished` can be used to determine whether a connection is currently established. To establish a connection and display the connection manager the blocking method `getConnection` is used. This method returns either true or false whether a connection has been established successfully or not. The methods `addConnectionStateListener` and `removeConnectionStateListener` can be used to be informed whenever the connection state changes. When acting as a `ConnectionStateListener` the application is also informed whenever an update to a `SecureComponent` happens. Therefore the `componentUpdated`-method is used. To disconnect an existing connection manually, the programmer has to use the `disconnect`-method.

The **ConnectionStateListener**-interface defines three methods. A class implementing this interface can be registered with a `SecureServer` using the `addConnectionStateListener`-method. After the registration the class will be notified about state changes via `connectionLost` or `connectionEstablished` and about changes to a `SecureComponent` via the `componentChanged`-method.

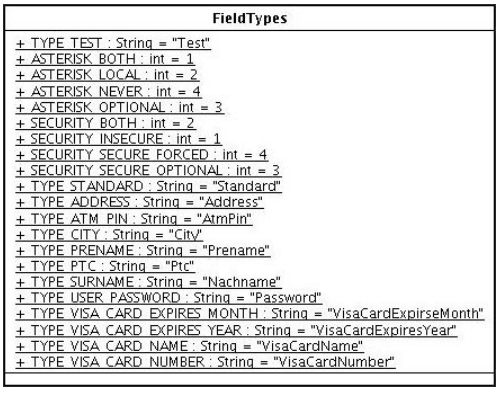

Figure 5.4: The FieldTypes-class.

The **FieldTypes**-class (see figure 5.4) in the framework package represents the different possible types the `dataType`-attribute can have. This class is never instantiated.

The **Helpers**-class (see figure 5.5) contains different static methods that can be used by any of the framework classes.

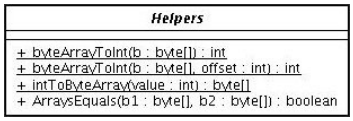

Figure 5.5: The Helpers-class.

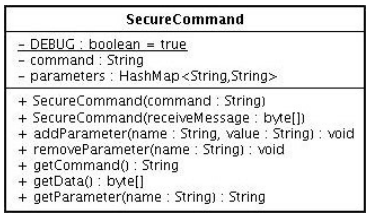

Figure 5.6: The SecureCommand-class.

The **SecureCommand**-class (see figure 5.6) is so far not necessary for anyone programming a SeCuUI server application using the framework. It is a representation of commands that can be sent between the client and the server application. A command consists of a command name and a list of parameters. The parameters always have a name and a value. When sending a command it is converted to an XML structure, which is then transformed into a simple byte-array. The methods `getData` returns the byte-array for a command. To decode an incoming byte-array back into a valid `SecureCommand` a class-constructor is available. A second constructor takes a string as parameter. This method creates a `SecureCommand` having the parameter value as command-name and an empty list of parameters.

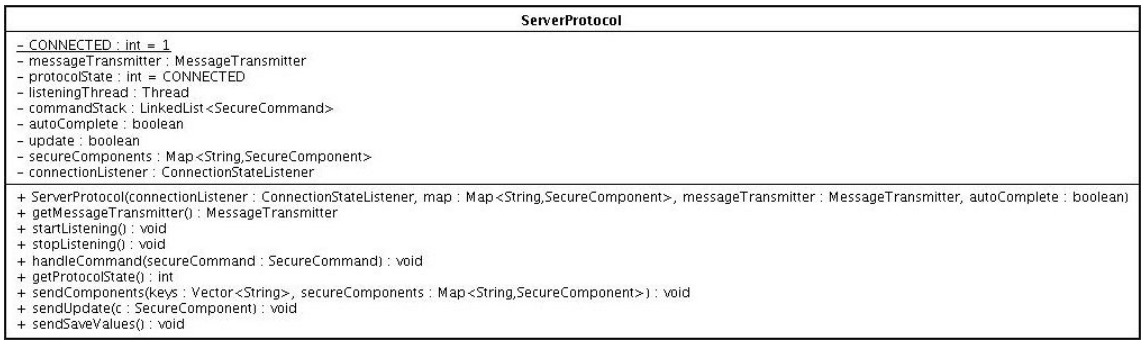

Figure 5.7: The ServerProtocol-class.

The **ServerProtocol**-class (see figure 5.7) handles the communication between the server and the client. This process cannot be influenced from outside the framework and works automatically. The `ServerProtocol`-class is not responsible for the handshake procedure between the two devices. This handshake procedure is done by the connection method used. The server uses a `MessageTransmitter`-class (see section 5.4.4) to send and receive messages. Messages are not sent in a blocking synchronous way. They are appended to a message queue that is processed by an extra thread. Like this, sending a message does not block the complete framework until it has been delivered. When adding new messages to the queue this is done in an intelligent way. For example: Whenever a new update message for a specific component arrives while another update message for this component is still pending only the newer one will be delivered. This saves bandwidth and ensures that always the newest status is shown on the mobile device. Every time a new character is typed into a text field a new update message is generated. Sending each of those changes to the mobile device would take much too long and block the rest of the application. This is why the message queue needs to be managed in a more intelligent way. Whenever a command is

received by the protocol its contents are analyzed by the `handleCommand` method and depending on the command and its contents the according action on the server side is performed.

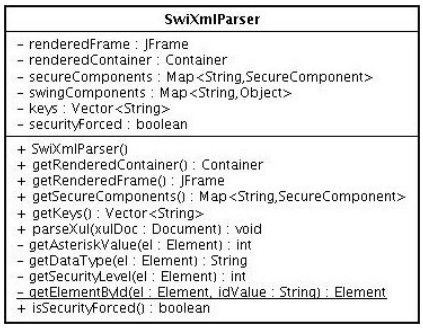

Figure 5.8: The SwiXmlParser-class.

The last class inside the `framework`-package is the **SwiXmlParser**-class (see figure 5.8). It is used by the `SecureServer`-class to parse XUL files and to manage the different `SecureComponents` – explained in section 5.4.5 – resulting out of these files. The most important method of this class is the `parseXul`-method. It processes the XUL code in three steps. In the first step the contents of the XUL file are rendered using the SwiXml-API (section 4.2), hence the name `SwiXmlParser`. The second step parses the document again looking for the three extended attributes `dataType`, `security` and `asterisk`. Together with the component belonging to this tag everything is stored inside a `SecureComponent`-object. The third step is used to filter the label-components that belong to a textfield-component. Those labels are not stored as a separate `SecureComponent`, instead they are linked to their corresponding `SecureComponent`. This is done to keep the textfield and its according label semantically together whenever the components are synched.

The `SwiXmlParser`-class maintains the IDs of the different components parsed in a list according to their appearance in the XML file. For each key an entry in a hash-map points to the according `SecureComponent`. A small boolean value named `securityForced` informs the framework whether there is at least one component in this set of components that forces the use of a mobile device.

5.4.2 ui-package

The `ui`-package (user interface-package) contains only two classes. These classes are additional user interface elements that are not part of the standard Java SE distribution.

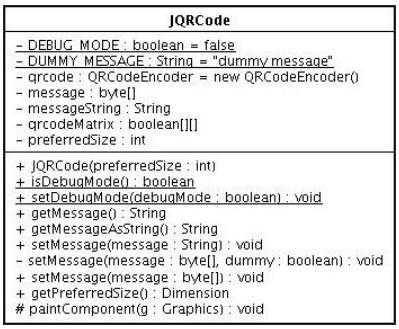

Figure 5.9: The JQRCode-class.

The first of those two classes is the `JQRcode`-class (see figure 5.9). This class is used to display a message-string as a 2D-barcode or QR-Code. When an instance of the class is created it shows

a barcode displaying the message “dummy message”. The complete barcode is marked by a red cross to indicate that this is not a correct barcode. Whenever the `setMessage`-method is called the component is updated with the new message and the red cross is disabled.

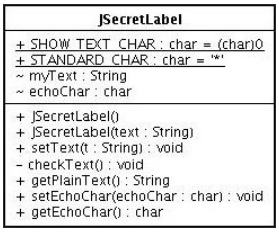

Figure 5.10: The JSecretLabel-class.

The other user interface class needed for SeCuUI is the `JSecretLabel`-class (see figure 5.10). SeCuUI offers the possibility to hide text on the public terminal and to replace it with a certain echo-character. For text-fields this behavior can be achieved by using a `JPasswordField` instead of a normal `JTextField`. The `JPasswordField` does not necessarily show what contents the field has. So basically it handles two strings of data. One containing the correct word and another displaying a certain password character for each char contained in the text string. The `JSecretLabel`-class models the same behavior for a `JLabel`. It also stores a different text value than the one that is displayed. By setting an echo-char the text is made invisible and replaced with a number of echo-chars according to the content’s length. Setting the echo-char to `null` causes the text to get visible again.

5.4.3 qrCode-package

The `qrCode`-package contains classes that are related to the QR-Code connection or the drawing of QR-Codes. Those classes are based on the .NET QR-Code library found at [43] and have been ported to Java from Bernhard Frauendienst for the use in [6]. Since those classes are used more or less like a library they are not explained in detail. The package itself consists of three classes:

- The `QRCodeException` is a special exception that can be thrown by the classes of this package.
- QR-Codes have different error correction values or modes. An alpha-numeric QR-Code reduces the bits used to store one character. The `QRCodeEncoder`-class basically is used to convert data to a byte-array that is afterwards converted to a two-dimensional boolean-Array representing the image. The most important method to do this is the `calQrcode`-method.
- A boolean-array with two dimensions, as it is produced by the `QRCodeEncoder`-class, is passed to the `QRCodeImage`-class. This class has static methods to draw a black-and-white pattern in a specific size on a given `JComponent`.

5.4.4 connection-package

The SeCuUI framework does not only allow one specific connection possibility. The whole connection process is modular. This means that the method how the two devices perform their handshake and transfer data may be chosen (e.g. connection by using a QR-Code). Additionally the type of connection can be chosen (e.g. bluetooth connection) and the way data is transferred on this connection may be chosen, too (encrypted or plain). To realize all this, a bunch of classes and interfaces is needed. The connection methods contained in the package `connection.methods` are explained at the end of this section. The rest of this package contains classes to make the connection process a modular as possible.

To get a better understanding of how connections in SeCuUI work, one can have a look of two connection stacks. The connection stack shown in figure 5.11 shows a hirachy of classes that are used to initiate a connection. After a successful initiation this stack is completely discarded and a `MesssageTransmitter`-object is used no matter in which way the device connection was chosen. This stack is called the communication stack and can be found in figure 5.12.

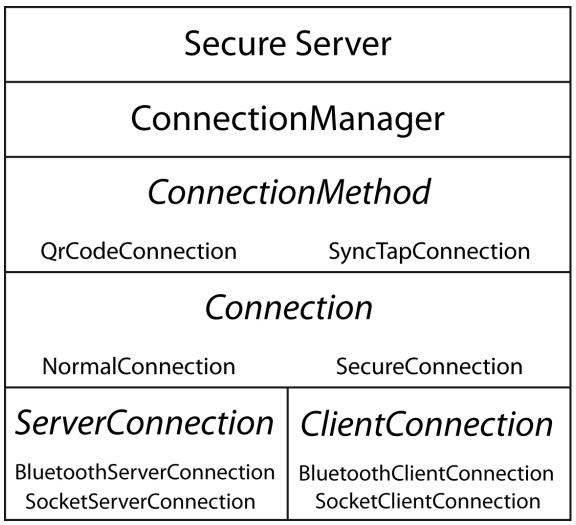

Figure 5.11: The SeCuUI connection-stack.

During a connection process the connection stack starts at the main class the `SecureServer`-class. This class initiates a modal dialog called the `ConnectionManager`. The connection manager creates a visual instance of an arbitrary number of `ConnectionMethods`. Currently SeCuUI supports a working `QrCodeConnection` and a dummy `SyncTapConnection`. The connection method can either use an unencrypted `NormalConnection` or an encrypted `SecureConnection`. Those classes handle the handshake process independently from the way data takes. To register a port at the local terminal first a `ServerConnection` is initialized. This can be either a `BluetoothServerConnection` using bluetooth or a normal socket connection called `SocketServer`-connection. They finally wait for an incoming connection and if a user connects a `ClientConnection`-object, according to the type of server connection, is created and passed back up to the connection-class. After a successful handshake the connection manager is closed and all that lasts is a `MessageTransmitter`-object encapsulated in a `ConnectionInformation`-object.

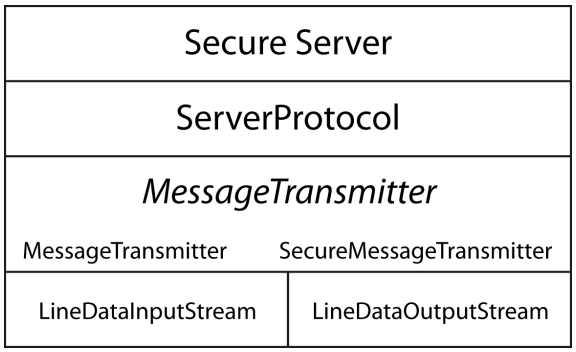

Figure 5.12: The SeCuUI communication-stack.

For the normal communication process (see the communication stack in figure 5.12) the secure server manages an extra `ServerProtocol`. This protocol takes a `MessageTransmitter` and uses it to communicate with the client. The message transmitter contains methods to send and receive

messages to the client that has been selected using the connection stack. For secure connections a `SecureMessageTransmitter` exists that is additionally able to encrypt and decrypt the messages using the public keys shared during handshake. As a basis the `MessageTransmitter` uses a `LineDataInputStream` and `LineDataOutputStream`-class, that is able to send packages or lines instead of single bytes over the network. All of the classes used for both stacks are explained in the following:

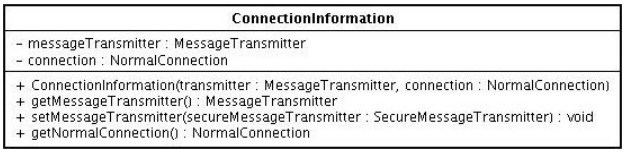

Figure 5.13: The ConnectionInformation-class.

The first important class here is the **ConnectionInformation**-class (see figure 5.13). Whatever connection method or type is chosen for a connection at the end of the handshake phase a `ConnectionInformation`-instance is passed to the framework. The class basically serves as a container for two elements. A `MessageTransmitter` and a `NormalConnection`. Both classes and their functions are described later in this section.

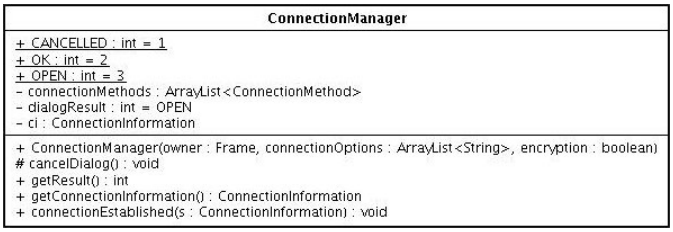

Figure 5.14: The ConnectionManager-class.

The **ConnectionManager** (see figure 5.14) is the visual interface that is used to establish any connection. The class is a subclass of `JDialog` and therefore blocks the program-execution while it is visible. Like this a `ConnectionManager`-instance is shown and when the window disappears a connection has been either established or not. The constructor-method of the connection manager accepts a `JFrame`, which will be the dialogs owner, and a list of so called "connection options". When the dialog window shows up it can contain multiple different connection methods for the user. Each of those connection options is a subclass of the abstract class `ConnectionMethod` explained below. Each of those connection options offers the connection manager, a visual component it can place inside its dialog window. A third boolean parameter to the constructor indicates whether the connection should be encrypted or not. When the dialog is dismissed it returns its status similar as Java's `JFileDialog` via the `getResult`-method. The dialog result can either be `OK`, `OPEN` or `CANCELLED`. When the dialog returned an `OK`-result the connection information for the established connection can be retrieved as a `ConnectionInformation`-object using the `getConnectionInformation`-miethod.

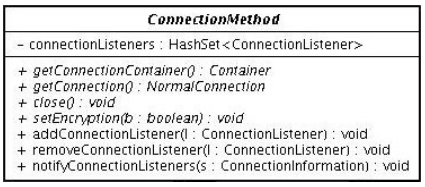

Figure 5.15: The ConnectionMethod-class.

Each different connection method is based on the abstract class **ConnectionMethod** (see figure 5.15), which already contains methods to manage a list of `ConnectionListeners` and is able to notify those as soon as a connection has been established. So far the `ConnectionManager` is the only class that is registered as a listener to those methods. It also defines four abstract methods that need to be implemented by subclasses. `getConnectionContainer` returns a visual instance of the connection method that can be placed inside the connection manager window. `close` is used to free up connections and memory that has been used before when dismissing the dialog. `getSecureConnection` returns the connection instance after the listeners have been notified that the connection has just been established. Optionally it is possible to set the encryption of the connection by calling the `setEncryption`-method. For this work, two different subclasses of this interface have been created. The `QrCodeConnection` displays a two-dimensional barcode based on the bluetooth address of a socket that is created on the server side. The `SyncTapConnection` is a dummy connection. It just gives and idea of how a second connection could look but it does not implement any functionality.

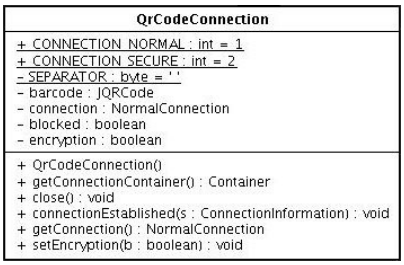

Figure 5.16: The QrCodeConnection-class.

The **QrCodeConnection** (see figure 5.16) implements the four abstract functions defined by the `ConnectionMethod`-class. Depending on the encryption-state set when creating the display component with the `getConnectionContainer`-method either a `NormalConnection` or a `SecureConnection` is instantiated and the corresponding URL to the server is displayed as a barcode. The process now waits for an incoming connection and then notifies the connection manager.

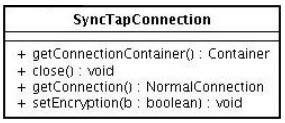

Figure 5.17: The SyncTapConnection-class.

The **SyncTypConnection** (see figure 5.17) is a dummy-class representing another connection method. Its `getConnectionContainer` method returns a simple button showing the text "SyncTap!" but the button has no effect.

The **NormalConnection** (see figure 5.18) is used to create a server socket and to wait for an incoming connection on that socket. In contrast to the `SecureConnection` the normal connection uses no handshake procedure except the normal handshaking done by the socket itself. As a socket this connection uses the `BluetoothServerConnection` but it would be also possible to use the `SocketServerConnection`, which uses a normal TCP socket, instead of a bluetooth device. Both connections are based on the interface `ServerConnection` so one could imagine other types of server connections. The interface declares a method named `getNextConnection`, which returns an incoming connection based on the `ClientConnection` interface. According to the server connections, there is also a `BluetoothClientConnection` and a `SocketClientConnection`. When the server connection returns the client connection, an input- and an output-stream for this connection is created. Those streams are then put together in a `MessageTransmitter`-object.

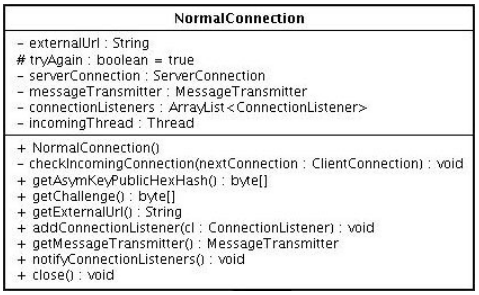

Figure 5.18: The NormalConnection-class.

Having those two streams, normally the handshake procedure would begin. As mentioned above the `NormalConnection` does not perform an additional handshake so all listeners of this connection are notified immediately.

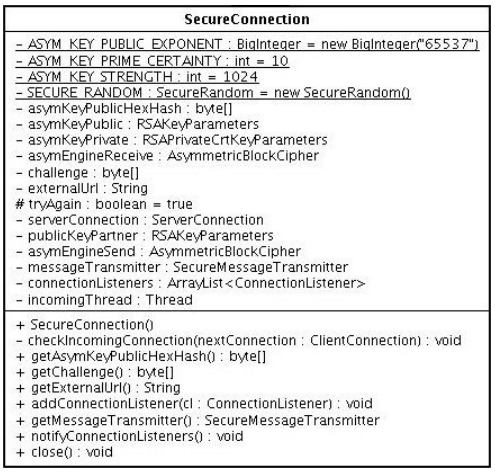

Figure 5.19: The SecureConnection-class.

The **SecureConnection** (see figure 5.19) works similar to the `NormalConnection`. Instead of the `MessageTransmitter` used with the NormalConnection a special `SecureMessageTransmitter` encrypts and decrypts the incoming and outgoing messages. To be able to encrypt data the class is provided with an asymmetric encryption engine based on RSA. The `SecureConnection` uses the same `BluetoothServerConnection` and the same input- and output-streams are passed to the message transmitter. This type of connection does perform a handshake which is done as follows: The client receives the public key of the server unencrypted. Together with the connection method – could be the 2D barcode – the client application is provided with a challenge and a hash value of the public key. With this hash value the client is now able to check if the public key sent is correct by calculating the same hash value out of the sent key again. The challenge is now encoded by the client with the public key he just checked. At the server the encrypted challenge is decrypted and compared with the real challenge that was displayed. If it does not match a `SecureConnectionException` is thrown. In case it matches the public key of the client is received – again encrypted. Now the server creates an encrypter for this public key and hands it over to the message transmitter, which is now able to encrypt and decrypt in both directions. After that procedure the handshake has finished and the message transmitter class can be used exactly the same way it would be used without an encryption.

The **MessageTransmitter**-class (see figure 5.20) is used together with the `NormalConnection`. It stores two streams, one to read incoming data and one to write

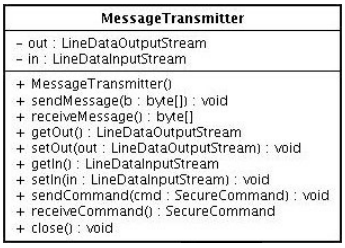

Figure 5.20: The MessageTransmitter-class.

outgoing data to. Additionally the two methods `sendCommand` and `receiveCommand` allow to send and receive `SecureCommand`-instances out of the streams. A `close`-method is used to dismiss the two streams.

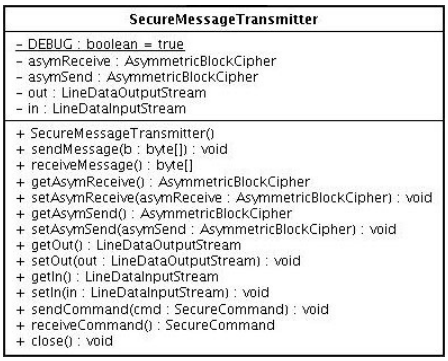

Figure 5.21: The SecureMessageTransmitter-class.

The **SecureMessageTransmitter**-class (see figure 5.21) is a subclass of `MessageTransmitter` in addition to the two streams, it can carry two `AsymmetricBlockCiphers`, which are part of the BouncyCastle Crypto-API explained in section 4.5. One of them is used to encrypt outgoing messages, one is used to decrypt incoming messages. Due to the type of encryption used – RSA in this case – the byte-stream is divided into packages before encryption and reassembled afterwards. For a class using one of both transmitters it makes no difference which one it is using.

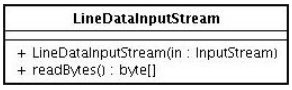

Figure 5.22: The LineDataInputStream-class.

The **LineDataInputStream** (see figure 5.22) and **LineDataOutputStream** (see figure 5.23) both extend the `DataInput-` respectively the `DataOutputStream`-class. Like this, the message length or the length of a "line" is sent to or received from the stream before the rest of the bytes. The `LineData`-classes make sure the right portion of data is received. This makes it possible to send messages with different lengths.

The **SecureConnectionException** (see figure 5.24) is part of the `SecureConnection` and is thrown when an error during the handshake-phase occurs.

On the lowest layer of this connection stack the actual connection-classes, contained in the `connection.types`-package, are situated. First a **ServerConnection**-class is needed. Each one of the connections is based on the `ServerConnection`-interfaces that specifies three methods:

- The `getExternalUrl`-method returns an URL on which the server-connection is waiting for an incoming connection. Usually this URL consists of protocol, adress and port.

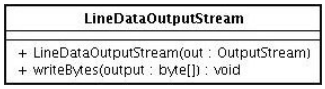

Figure 5.23: The LineDataOutputStream-class.

Figure 5.24: The SecureConnectionException-class.

- The `close`-method is used to free up sockets and objects that have been created before unlinking the object.
- To acquire a new client-connection the blocking method `getNextConnection` is used. This method returns a `ClientConnection`-object after a connection has been established.

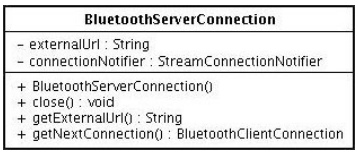

Figure 5.25: The BluetoothServerConnection-class.

Momentarily the **BluetoothServerConnection**-class (see figure 5.25) is used all over the framework. The `SocketServerConnection` – explained in the next section – was used during the testing phase. The framework itself so far offers no method to configure which type of connection shall be used. One possibility would be to pass the connection-type with the constructor of the `SecureServer`-class (see section 5.4.1). Another possibility would be to just create different connection methods and indicate the connection use inside their name. Instead of `QR_CODE_CONNECTION` there could be a `QR_CODE_BLUETOOTH_CONNECTION` and a `QR_CODE_SOCKET_CONNECTION`. The implementation of the `BluetoothServerConnection` is as follows: The constructor opens the local bluetooth device and determines its local address. It also creates a listener object for incoming connections. An example for such a bluetooth-address is: `btspp://00236CA0E9E9:1`. This address is now available by calling the `getExternalUrl`-method. Calling the `getNextConnection`-method tells the listener to return the next incoming connection. After this happened the input- and output-stream of the connection are passed to a new `BluetoothClientConnection`-object that is returned. Whenever the `close`-method is called the listener for incoming connection is terminated and the resources are freed up.

The **SocketServerConnection** (see figure 5.26) works similar to the `BluetoothServerConnection` but it just uses the normal `ServerSocket`-class of the `java.net`-package. This is why a connection URL is something like `socket://192.168.0.5:1024`. The `close`-method closes the socket and the `getNextConnection`-method accepts a new connection from the socket and returns a `SocketClientConnection` with the two streams of this connection.

Both client connections returned by each of the server connections are based on the **ClientConnection** interface. It defines that a `ClientConnection` needs to return an input stream and an output-stream to communicate with the client. A `close`-method is called before releasing the client connection.

For the client connection there is no difference between the **BluetoothClientConnection** (see figure 5.27) and the **SocketClientConnection** (see figure 5.28). Both take the normal input- and output-streams and use them to instantiate a `LineDataInputStream` or a `LineDataOutputStream`. Calling the `close`-method, both streams are closed.

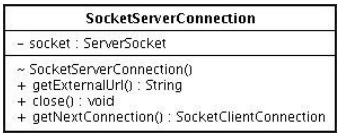

Figure 5.26: The SocketServerConnection-class.

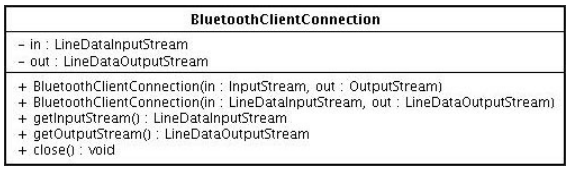

Figure 5.27: The BluetoothClientConnection-class.

5.4.5 components-package

The components-package contains different classes that each represent one component that can be synchronized with a connected mobile device. All those components are based on the abstract `SecureComponent`-class. A `SecureComponentListener`-interface exists for changes that happen to these components. In addition, for the component types button, label, text-field and radio button an extra `SecureComponent`-subclass exists.

The abstract class **SecureComponent** (see figure 5.29) serves as a draft for all the different secure components. A `SecureComponent` is the representation of an interface element having additional SeCuUI specific parameters. Basically this class already offers a big number of functions. The most important method in this class is the static method `createSecureComponent`. It returns, depending on a given component, a correct instance of one of the subclasses of `SecureComponent`. Other methods are setters and getters to the following security related values used by SeCuUI. The `asterisk`-value states where the text of this element will be visible, the `dataType`-value says which type of value is contained in this secure component and finally the `security`-value defines where the element can be used. More about these three properties is explained in section 5.2.1. To get notified whenever a `SecureComponent` changes the class maintains a list of listeners. A pointer to the server-side graphical representation of this component is stored and can be retrieved using the `getContainer`-method. Three methods have to be specified according to which component-type the class represents. Those three abstract methods are:

- `getCommandAdd` returns a `SecureCommand` that is used to add this component to the remote interfaces on the mobile device.
- `getCommandUpdate` returns the `SecureCommand` needed in case of an update of the component to re-sync the two components on either side.
- `getValue` forces the component to return a string representation of this component that can be used for different things but mainly for debugging.

In the following each of the subclasses of `SecureComponent` is described.

The **SecureButton** (see figure 5.30) represent a button component on the user interface usually represented as a `JButton` swing-component. The class defines the three abstract methods needed, to add and update the remote component, as well as it returns the button's caption as a string for the `getValue`-method. When this class is instantiated, it is registered as a listener for this button to receive changes of the button's properties. Clicking this button is not transferred to the client because all computational logic of a framework-application is done on the server side.

The **SecureLabel**-class (see figure 5.31) replaces an incoming label component on initialization of the class. This is needed to be able to apply the different asterisk-options. The new label

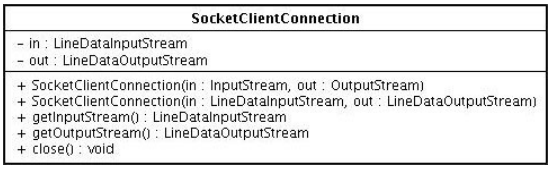

Figure 5.28: The SocketClientConnection-class.

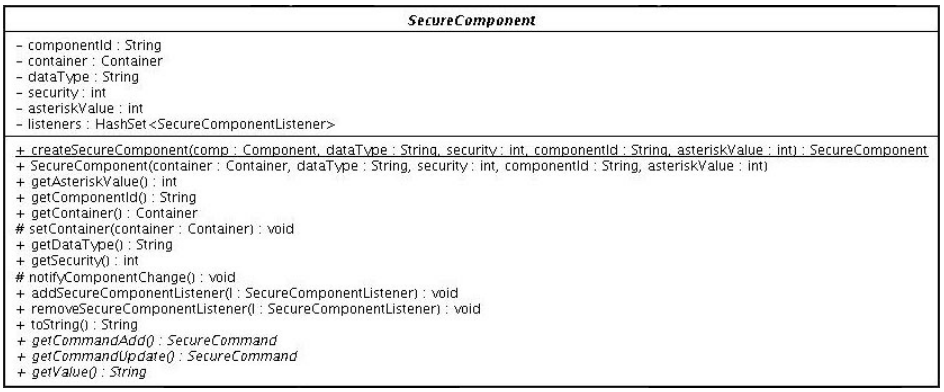

Figure 5.29: The SecureComponent-class.

class `JSecretLabel` is explained in section 5.4.2. Whenever the contents of the label change, all component listeners are notified. The `getValue`-function of this component always returns the plain text of the label even if the text is currently invisible due to the asterisk-setting.

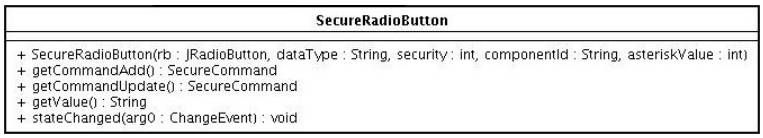

Figure 5.32: The SecureRadioButton-class.

The **SecureRadioButton** (see figure 5.32) represents one radio button as a member of a group. In Java SE radio buttons are managed putting them into an extra `ButtonGroup`-instance. For SeCuUI grouping is done as in `HTML` by assigning a group name to each button. Buttons having the same group name belong together. This group name is sent together with the add command. Updates contain not only the caption of the radio button, but also its state.

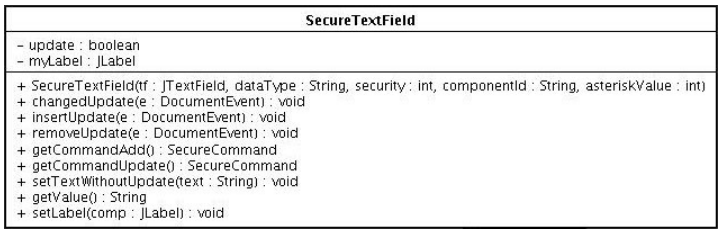

Figure 5.33: The SecureTextField-class.

Whenever a **SecureTextField** (see figure 5.33) is created, the according TextField-component is replaced by a `JPassword` component. This component allows it to set an echo-char that hides the text on the screen. Having all `TextFields` replaced with `JPassword`-fields makes it possible to apply the `asterisk`-value. Changes to the text are synchronized using an update command. A

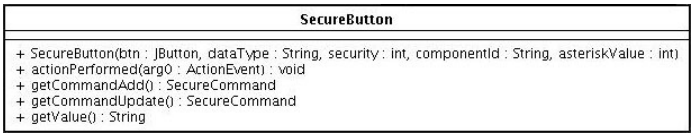

Figure 5.30: The SecureButton-class.

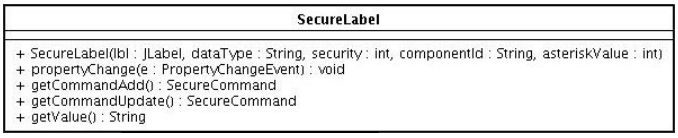

Figure 5.31: The SecureLabel-class.

`SecureTextField` can optionally carry a pointer to another `JLabel`-component. The text of such a label would then be shown together with the text field on the mobile device.

5.5 Building an application with the SeCuUI framework

This section shows a basic example of a minimum Java application that uses the SeCuUI framework. The whole complexity of the framework is more or less transparent to any programmer working with it. Basically the `SecureServer`-class is all the programmer needs.

```
1  JFrame frame = new JFrame("SeCuUI Example");
2  JPanel appPanel = new JPanel();
3  appPanel.setLayout(new BorderLayout());
4  SecureServer server = new SecureServer(true, SecureServer.
       CONNECTION_QR_CODE);
5  server.parseXul("xulFile.xml");
6  appPanel.add(server.getCurrentComponent(), BorderLayout.CENTER);
7  appPanel.add(server.getToggleButton(), BorderLayout.SOUTH);
8  frame.setContentPane(appPanel);
9  frame.pack();
10 frame.setVisible(true);
```

Figure 5.34: Simple example code for a working SeCuUI server.

Figure 5.34 shows a simple example. This code placed inside a main method of a Java application would already create a working server application. The code creates a `JFrame` window and a `JPanel` to put the components in. After that a `SecureServer`-instance is created. The first parameter activates `autoComplete` for this server, the second parameter defines the QR-Code connection as a standard connection. After the object has been created a XUL-XML-file is parsed that contains a simple set of components. The contents of the XUL file are shown in figure 5.35. After the code has been parsed, the parsed component is placed inside the panels center region. To toggle the connection the button provided by the framework is used and added to the lower region of the panel. After that, the panel is added to the frame and the frame is displayed. The running server would look something like the image shown in figure 5.36.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <panel>
3 <vbox>
4         <hbox Border="EmptyBorder(5,0,5,0)">
5                 <label id="lbl_test" LabelFor="tf_test" text="Testeingabe:"
                      HorizontalAlignment="RIGHT" PreferredSize="150,30" Border="
                      EmptyBorder(5,5,5,5)" Font="Arial-BOLD-16"/>
6                 <textfield id="tf_test" columns="20" text="" security="
                      SECURITY_BOTH" dataType="TYPE_TEST" asterisk="
                      ASTERISK_NEVER" Font="Arial-BOLD-16"/>
7         </hbox>
8         <hbox Border="EmptyBorder(10,0,10,0)">
9                 <button text="Weiter" action="submit" id="btn_next_2"/>
10        </hbox>
11 </vbox>
12 </panel>
```

Figure 5.35: Example XUL file that can be used inside the server Example.

Figure 5.36: A picture of a simple SeCuUI server running.

6 Client Application

This section describes the client application of SeCuUI. In contrast to the different server applications that are built using the framework described in section 5, there is only one single client application for SeCuUI. This client application is able to communicate with all different server applications that can be possibly created. Like this a user needs to install just one client application and after that she is able to communicate with all different kinds of servers based on SeCuUI. Another advantage of this method is that the application has a single data storage for the auto-complete feature for all of the servers. More details on the auto-complete feature are explained in section 6.1. The client itself thus has to be able to connect using all different connection methods. For this work the QR-Code connection is the only connection fully implemented. A client-side look on this connection method is explained in section 6.2. The different classes and their packages are explained in section 6.4. The client application is coded using Java ME. To better understand the explanation of the code, section 6.3 features a short introduction to Java ME and explains how the storage of data used for the auto-completion works. Afterwards a typical usage scenario of the client application is shown in section 6.5.

The SeCuUI client application is a little less modular than the framework. It must be able to communicate with all different servers that can be created using the framework. For this reason all possible connection methods and types must be supported by this application even if they are not needed for one specific server. Looking on one special case this means that someone installing the SeCuUI client on his mobile phone produces some overhead first, but as soon as the user uses more than one SeCuUI-based system the advantages of this architecture become visible. For all different kinds of public terminals it now suffices to start one single application and because all the public terminals share a common code-base even the stored auto-complete data is shared among different terminals and companies. A big advantage for the user.

6.1 Auto-Complete Feature

As mentioned in section 5.3, the auto-complete feature is nearly completely part of the client application. This is done to ensure security of the saved data on the mobile device. All the auto-complete values are stored in the mobile device's memory. Any server application is not able to access this memory or the entries stored in there. When the client application starts, the memory is parsed and the different auto-complete values are loaded. Whenever a form is transferred to the mobile device, the server sends the type of data required in the different fields. This is done by a parameter denoting the semantic type of the content. Whilst certain types of standard parameters exist (see figure A.3) the servers can extend this list by using not documented types.

Receiving the type of data, the client looks for data entries of this type of data that have been entered anytime before. A list with the matching entries is automatically displayed below the standard entry field. The server application is not informed of the existence of any matching entries. Only when the user selects a value out of this list of values it is entered in the form field and synchronized with the server application. Every time the connection is closed or a new form is sent to the device, the current list of auto-complete values is saved.

In case the programmer of a server application does not want that the user is able to use the values stored on his phone or if he does not want the entered values to saved permanently he can disable the auto-complete feature on the server side. This decision is broadcasted with each form to the mobile device and any eventually matching auto-complete values are hidden. The actual implementation of the auto-complete feature is explained in detail in section 6.4.

The user of the client application has the possibility to manage the elements stored on his device. This is necessary in case he entered a wrong value or a stored value is not needed anymore (e.g. an old address). In the main menu of the client application the user can choose between the options "Show record store" or "Delete record store". With the second option all entries are

deleted at once. The first option first displays a list of groups. Each group contains the auto-complete values for one specific data type. Selecting a group shows the different entries saved for this group. The user is then able to delete a certain entry from the mobile device's memory.

6.2 QR-Code Connection

The QR-Code connection is used to easily connect the public terminal and the mobile device in a secure way. Therefore a server application displays a two-dimensional barcode. How this barcode is created has already been explained in the framework section. On the client side the process is a little bit more complex. The first thing is, each device that should be able to use the QR-Code connection needs an integrated camera to take a photo of the barcode. During the user study conducted with this work it turned out that 71 percent of the participants already have a camera integrated in their mobile devices (see section 7.7.3). When using the QR-Code connection the camera is used inside the SeCuUI client application to take a photo of a potential barcode. This picture is then processed and in case a barcode is found the contents are checked. To process the image and find any containing barcodes the Google "Zebra Crossing" library for Java ME is used (see section 4.6).

The barcode displayed contains the network address of the server together with a port number at which the client may connect to the server. In case of a secure connection additionally a hash-value of the server's public key and a challenge are included in the barcode. The hash-value is used to check the incoming public key of the server during the handshake-phase and the server expects the client to send the challenge-value back encrypted. After the according handshake-procedure a normal network connection is established and is used for data transfer.

6.3 Java ME

The SeCuUI client application has been developed completely using Java ME. Java ME stands for "Java Micro Edition" and offers a reduced set of classes for the development on mobile devices. Applications developed with Java ME run on most of today's mobile devices though they all have different operating systems. The Java Virtual Machine exists for nearly any platform. This is the main reason why the SeCuUI client was made using this programming language. Another point is that Java ME offers easy and reliable network access, which is crucial for an application like this.

For Java ME different configurations and profiles are defined. A configuration defines a set of requirements devices have to fulfill to run an application according to this configuration. For SeCuUI the "Connected Limited Device Configuration" (CLDC) was used. Together with this configuration a profile defines the API or the set of classes the programmer can use to build his application. The SeCuUI client is a MIDlet and hence uses the most common "Mobile Information Device Profile" (MIDP) [37].

Looking at the MIDP-API [38] one can see that every MIDP-application extends the `MIDlet`-class. Since this class is abstract the programmer is forced to write some methods to get the application running. The most important one here is: `startApp`. This method is the equivalent to the `main`-method in a basic Java Standard Edition-application. To produce visible output a MIDlet has to acquire a `Display`-object. This can be retrieved using the static-method `getDisplay`-contained in the `Display`-class. This method takes the MIDlet instance as a parameter. All objects that can be displayed on a display are based on the `Displayable`-class. They can be made visible using the `setCurrent`-method of the `Display`-class. In most cases the element that is attached to the display is a `Form`- or an `Alert`-element. Forms can then have different elements inside their form body. To load and save data the MIDP-API does not allow a direct access to the device's storage memory, instead `RecordStore`-objects can be used. They are managed by the Java Virtual Machine running on the device. A direct memory access is impossible. More about the record stores is explained in section 6.3.1.

6.3.1 Java ME Record Store

In Java ME file access is done using a record store. The `RecordStore`-class is part of the MIDP-Profile-API [39]. A record store is used to save data permanently between different sessions of a MIDlet and as its name says it is able to store multiple records. `RecordStores` can be shared among different MIDlets. A static method `openRecordStore` inside the `RecordStore`-class is able to open a record store with a given name. The different records are all stored with an ID determined by the operating system and contain a byte-array each. Changes to a record store entry are immediately persistent throughout the mobile device.

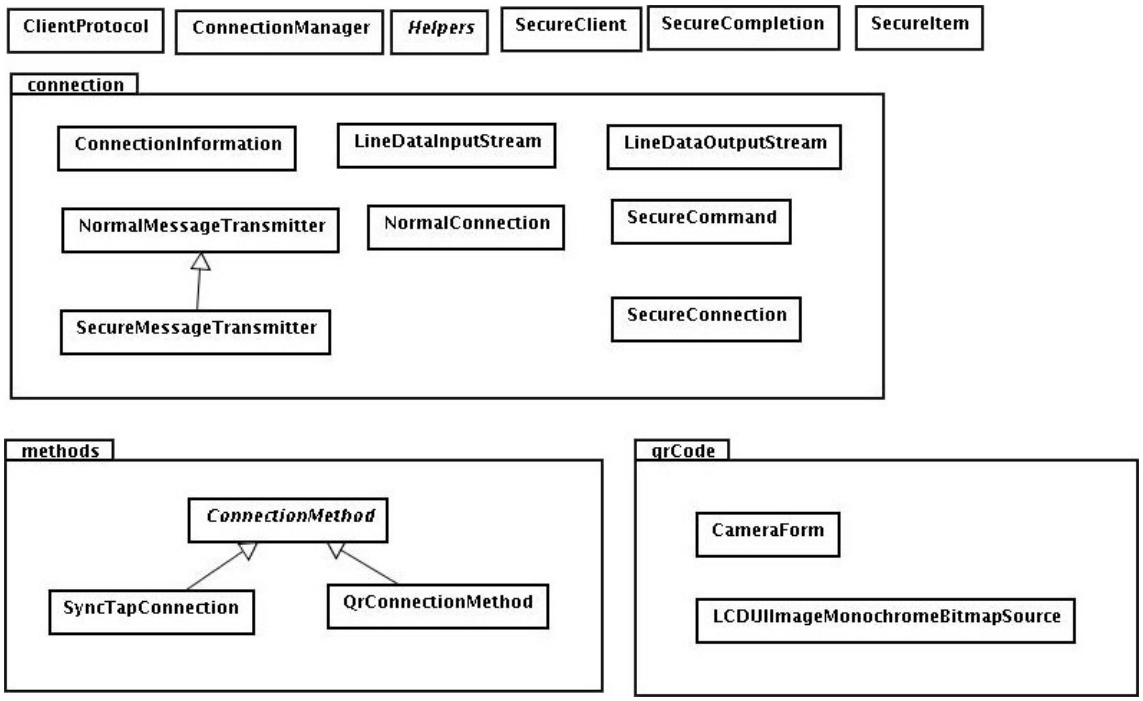

Figure 6.1: The package structure of the SeCuUI client.

6.4 Client Components

In this section the different classes the client application consists of are described. For most of the classes explained in the framework chapter of this document (see chapter 5) the client offers a matching counterpart. The client is divided into different packages, too. The client-package (see section 6.4.1) itself contains the main classes such as `SecureClient`-class that is launched when the client starts. It also contains the globally used `Helpers`-class. The following packages are sub-packages of the client-package. The connection-package contains classes handling incoming connections and messages. These classes are described in section 6.4.2. The methods-package, explained in section 6.4.3, contains classes handling the different connection methods. As in the framework, this package contains an abstract class all connection methods are based on. So far, this is a working QR-Code class as well as a dummy implementation of a SyncTap connection. Finally, in section 6.4.4, the different classes of the qrCode-package that are used together with the QR-Code connection are explained. Figure 6.1 shows how the different classes of the client application are arranged in their packages. A fully expanded UML diagram can be found in the appendix (see figure A.2).

6.4.1 client-package

The client-package is the main package of the SeCuUI client. It contains the most important classes used during execution. Especially the main MIDlet-class `SecureClient` is included in this package. This and the other classes contained in this package are described here.

Figure 6.2: The SecureClient-class.

The most important class of the whole client application is the main class **SecureClient** (see figure 6.2). Since this class is executed on the mobile device it extends the `MIDlet`-class coming with Java ME. The class also implements the `Showable`-interface. How this interface works and why it was introduced is explained further down this section. When the application is initialized the `startApp`-method is called. This method initializes a vector to store the incoming secure components. The auto-completion feature is handled by another class called `SecureSwingCompletion`. This class is also initialized at start. As a last step the display is allocated and a new `ConnectionManager`-instance is displayed on the screen.

The `SecureClient`-class is then notified when a connection is established and creates a new `ClientProtocol`-instance. The protocol then waits for incoming events. The class also has several methods to access the secure items that are stored using this class. `addSecureItem`, `removeAllSecureItems`, `getSecureItemWithId` are only some of the methods this class offers to affect those items. Another task of this class is to access the phone's memory. With Java ME access to the phone's memory is done via so called `RecordStores`. More about this was explained in section 6.3.1.

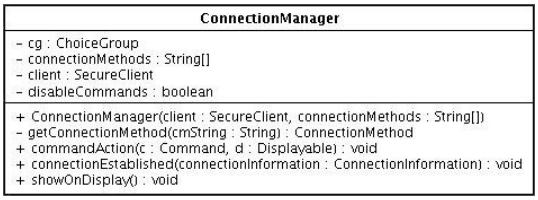

Figure 6.3: The ConnectionManager-class.

The **ConnectionManager**-class (see figure 6.3) of the SeCuUI client does not only handle the different connections. It also is sort of the main menu to the application. The connection manager

is the first thing that is displayed after the start of the application and it also contains the option to access the management of the auto-complete entries. The `ConnectionManager` extends the Form class of Java ME. Its constructor is called with a list of different connection methods that shall be displayed. For the tests of the prototype, this list consisted just of a link to the QR-Code connection. The second parameter passed to the `ConnectionManager`-class on instantiation is a pointer to the `SecureClient` application. Similar to framework the connection manager creates an instance of each connection method and adds their names to a list. The last two entries of this list are always fixed denoting: "Show record store" and "Delete record store". Choosing one of those entries calls the appropriate method of the client-class. Any other choice results in showing a displayable object of the selected connection-method. With each instantiated connection method the connection manager is registered as a listener. After the class is notified that a connection has been established, the event is passed to the client application.

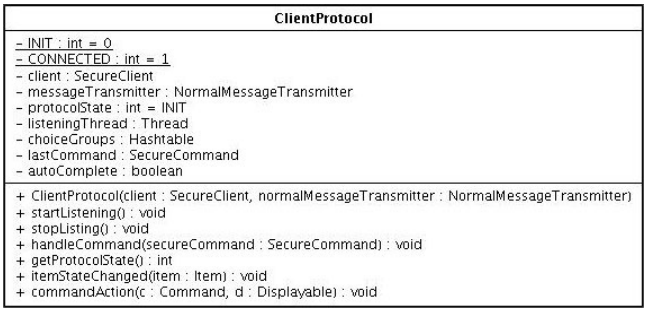

Figure 6.4: The ClientProtocol-class.

The complete communication between a server and the client – besides the handshake procedure of the connection method – is handled in the **ClientProtocol**-class (see figure 6.4). The `ClientProtocol` again is the counter instance to the `ServerProtocol`-class in the framework. To simplify things a little bit on the client side, there is no `ConnectionInformation`-class here. Instead a `MessageTransmitter` is passed to the `ClientProtocol` on instantiation. The public method `startListening` creates a thread that waits for incoming commands of the message transmitter. Those commands are then handled by the `handleCommand`-method. Depending on the current protocol state, different commands are accepted. Those commands are:

- **CLEAR_FORM**: This command resets the complete form that is currently shown. Depending on the setting of the auto-complete value the current values of the fields are stored to the phones memory or discarded. All secure items that have been transferred to the client are also removed from memory.

- **ADD_-commands**: Depending on the type of component different add-commands can be transferred. Most important is that each of the add commands sends an ID as a parameter. This ID is used to determine the right component whenever future commands like update commands arrive. `ADD_LABEL` adds a simple text string to the form showing a text that comes with this add command. The `ADD_RADIO_BUTTON` command adds a radio button with text to the form. Each radio button is interconnected with other radio buttons of the same group using a `groupName`-parameter. The `ADD_TEXTFIELD`-command adds an input text field to the form. Optionally this text field can have a label in front of it hinting on what has to be entered in this field. In case the auto-complete feature is turned on, the application now searches for any previously stored values for the type of data submitted with this add-command. If there are any, an additional drop-down-component is just placed below this text-field and the different possibilities are added to this list. In case an entry of the list is selected the text-field is updated with that value. The `ADD_BUTTON` command creates a new entry in the list of different menu-options that can usually be accessed by a soft-key near

the display. Pressing this menu-option results in programmatically pressing the according button on the server side.

- **UPDATE**: The update-command is identical for all components on the client side. It carries the ID of the component that should be updated and the new value for this component. Depending on the type the component according to the ID, either the label-text, the text-fields value, the caption of the entry in the hot-key-area or the selected radio button is changed.
- **SAVE_VALUES**: This command may be issued by the framework to make the client write the currently entered values to the mobile device's memory.

Another important method of the `ClientProtocol`-class is the `itemStateChanged`-method. It is called whenever the forms contents change together with the item that just changed. After that an update command for this item is created and sent to the server using the message transmitter.

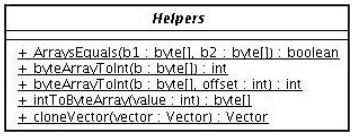

Figure 6.5: The Helpers-class.

The **Helpers**-class (see figure 6.5) contains some static functions that are used throughout the client application. For example a function comparing two byte-arrays or a function to clone a vector.

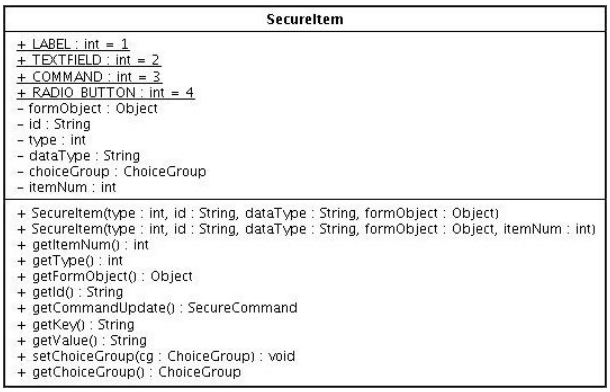

Figure 6.6: The SecureItem-class.

The **SecureItem**-class (see figure 6.6) represents one form component that has been transferred from the server to the client. On the server side this class is called `SecureComponent`. The clients class name to this functionality differs because the client has only this one class for all the different types of components. The server framework uses different classes for each display component. To distinguish the different component types anyway, the class contains different static integer values, one for each component type. This value is passed to the constructor together with an ID for this component, a data-type and the object that has been added to the form and that belongs to this item. The `getUpdateCommand`-method returns a different `SecureCommand` instance depending on which type this item currently represents.

The complete management of the auto-complete values is done by the **SecureCompletion**-class (see figure 6.7). This class is a subclass of `Form` because it does not only load and save the auto-complete values but it serves also as an interface to modify the values. The auto-complete values are stored in the system memory of the device in an XML structure. When the class is instantiated the `RecordStore` containing this XML document is opened. More information about

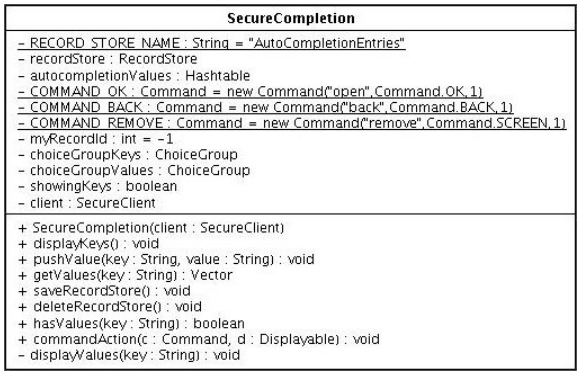

Figure 6.7: The SecureCompletion-class.

the record store principle is given in section 6.3.1. The XML is then parsed and each key-value pair is changed into a java object structure based on vector objects. The `saveRecordStore` reverts this process and saves the currently stored values to the memory. The record store may also be deleted from the phone's memory by using the `deleteRecordStore`-method. When the form is shown on the phone's display it first displays a list of all keys that are currently stored on the phone. The user can select one of those keys and a list of entries for this key is displayed. Selecting one of those keywords, she may delete it. When the last value of a certain key is deleted the key disappears from the list, too.

Java ME attaches everything that should be currently visible to the the display. But in contrast to the normal window environment one is accustomed with, only one element can be displayed at a time. Since the display object is given to the main class of the MIDlet it is hard to show forms coming from other classes. To make this easier the **Showable** interface was introduced for this client application. Any class implementing the `Showable`-interface can be made visible by calling a `showOnDisplay`-method. This is a far more convenient way of displaying different components. This interface is implemented by the classes: `ConnectionManager`, `SecureSwingClient`, `ConnectionMethod`, `QrConnectionMethod`, `SyncTapConnection` and `CameraForm`.

6.4.2 connection-package

The connection-package contains the different classes that handle the data during the connection process. Due to the fact that the client software is able to use different connection methods different types of classes exist in this package doing more or less the same work. The `NormalConnection` and the `NormalMessageTransmitter` are used for an unencrypted connection. The `SecureConnection` and `SecureMessageTransmitter`-classes instead encrypt and decrypt the data passed between the client and the server.

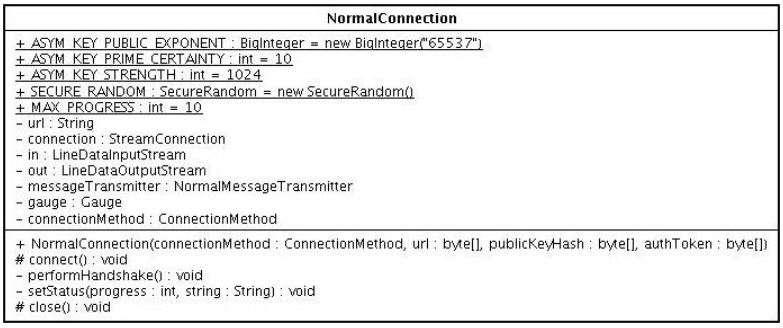

Figure 6.8: The NormalConnection-class.

The **NormalConnection** (see figure 6.8) extends the Java ME-class `Alert`. Like this, progress during the handshake process can be shown to the user. The `NormalConnection` is instantiated passing a connection method, a URL, a public key hash and an auth-token to the constructor. The public key hash and the auth-token are only used with encrypted connections and are therefore just discarded by this class. The URL contains the URL where the server application should be reached. During the instantiation the `connect`-method is called. It starts a thread that tries to open the specified URL and creates a `MessageTransmitter`-object out of the input- and output-stream of the opened connection. After this was completed successfully the handshake is performed. The `NormalConnection` does no separate handshake and hence the connection process ends and the objects waiting for this connection are notified.

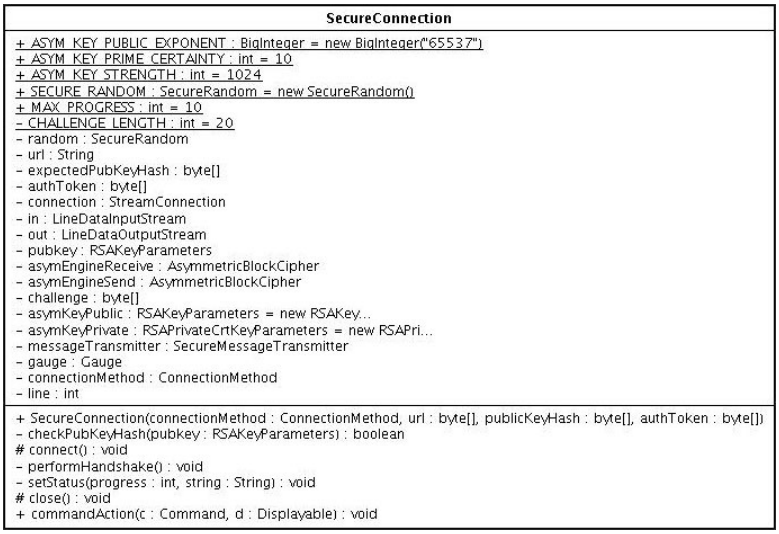

Figure 6.9: The SecureConnection-class.

The **SecureConnection** (see figure 6.9) works similar to the normal-connection. Except that it uses the public-key-hash and the authorization token. After the connection is open the handshake procedure starts. The server firsts sends its public key to the client and it is checked whether it conforms with the hash of the public key. After that the authorization token is encrypted using this public key and sent back to the server. It can now check that the encrypted token was the same it issued previously. The next step is that the client sends its own public key encrypted to the server, which can decrypt the key after validating that the token was correct. Now that both sides know the public key of the respective other side they are able to communicate in a secure way. This means the handshake process was successful and the objects waiting for this connection are notified.

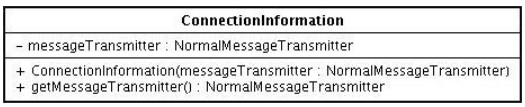

Figure 6.10: The ConnectionInformation-class.

Whenever a connection was established the listening classes are informed with a **ConnectionInformation**-object (see figure 6.10). This object contains a message transmitter that can be used to send or receive messages without having to worry about connection types or encryption. Depending on the type of connection, either a `NormalMessageTransmitter` or a `SecureMessageTransmitter` is contained in the `ConnectionInformation`. The `SecureMessageTransmitter`-class is a subclass of the normal transmitter and so it can handle the exact same method calls as the parent class.

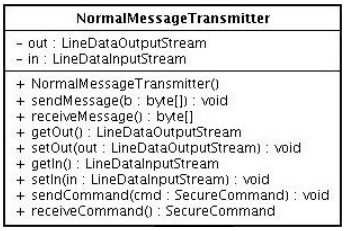

Figure 6.11: The NormalMessageTransmitter-class.

The **NormalMessageTransmitter** (see figure 6.11) uses a `LineDataInputStream` and a `LineDataOutputStream` to receive or send data. The message transmitter is able to send a pure byte message using the `sendMessage`-method or it can receive one using the `receiveMessage`-method. It can also handle `SecureComammands` and either send them or receive them from the input stream.

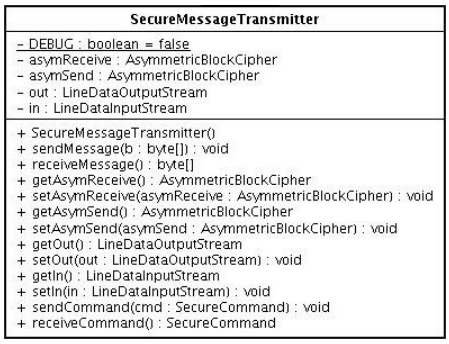

Figure 6.12: The SecureMessageTransmitter-class.

The **SecureMessageTransmitter** (see figure 6.12) however additionally uses two asymmetric block ciphers for encryption of data that is going to be sent and for the decryption of incoming data. For the used encryption the encrypted blocks may have only a certain byte-size. Because of this a message is split into several byte-blocks of the maximum block size and the first message encoded contains the number of those blocks that form one unique message. This process is done using the `sendMessage`-method just as with the `NormalMessageTransmitter`. The `receiveMessage`-method works the same first reading and decrypting the number of blocks belonging to the encrypted message and then receiving all those split-blocks and putting them back together. Certainly the `SecureMessageTransmitter` is also able to send and receive `SecureCommands` using the `sendCommand` and `receiveCommand`-methods.

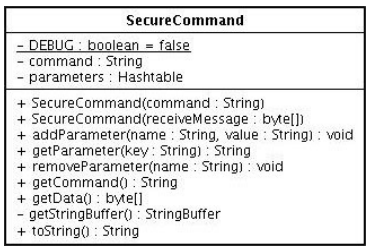

Figure 6.13: The SecureCommand-class.

A `SecureCommand` (see figure 6.13) is a representation of message that is passed between client and server. The command contains a command name and a indefinite list of key-value pairs representing parameters to this command. The parameters are stored using a hashtable. Before

the command is sent it is transformed into an XML structure. The `getData`-method does this and returns then a byte-array containing the XML tree of this command. A `SecureCommand` can be either created with just the commands name as a string or by using a byte-array that has recently been received that is then parsed to a new `SecureCommand`-instance.

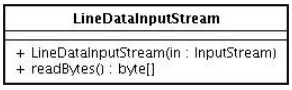

Figure 6.14: The LineDataInputStream-class.

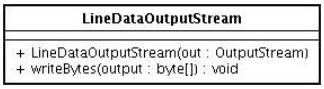

Figure 6.15: The LineDataOutputStream-class.

To make all this work, two more classes are needed. The normal input and output-streams used when networking with Java or Java ME are unable to sent or receive a complete message (a certain amount of bytes forming one group). Therefore the `LineDataInputStream` (see figure 6.14) and `LineDataOutputStream`-classes (see figure 6.15) are used. They prepend an integer with the number of bytes the message has before sending it. The other side may now read out one integer from the stream and immediately knows how many more bytes following this integer will form the next message. The **LineDataInputStream**-class therefore has a `readBytes` method, the **LineDataOutputStream** uses a `writeBytes`-method.

6.4.3 methods-package

The methods-package contains the different connection methods. For each connection method that is defined by the SeCuUI framework and could therefore be used by a server application the client needs to have a respective connection method, too. Connection methods for the client application are based on the abstract class `ConnectionMethod`. There exist two subclasses to this class that implement its abstract methods. The `QrConnectionMethod` is a fully working counterpart to the QR-Code connection used with the framework. The `SyncTapConnection`-subclass is only a prototype showing how additional connection methods could eventually work.

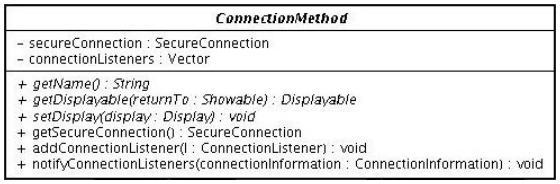

Figure 6.16: The ConnectionMethod-class.

The **ConnectionMethod**-class (see figure 6.16) is an abstract class serving as basis for the different connection methods that can be used with SeCuUI. The class already contains the functionality to manage a list of connection listeners and to notify those listeners in case of a successfully established connection using the `notifyConnectionListeners`-method. The class defines three abstract methods: The `getName`-method simply returns a string denoting the name of this connection method. This is used because in contrast to the server applications the connection methods of the client do not have an immediate graphical representation. Instead, when running the client application, a text-based list of possible connection methods is displayed. These texts are generated calling the `getName`-method on the different connection-method classes. A `setDisplay`-method

is called every time before the `getDisplayable`-method is called. Like this the connection method can grab hold of the display during the connection phase. The `getDisplayable`-method itself should return a displayable component that is used to execute the designated connection method.

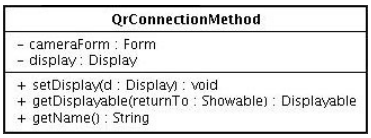

Figure 6.17: The QrConnectionMethod-class.

The **QrCodeConnection**-class (see figure 6.17) implements the three abstract methods to create two-dimensional barcode connections. Hence the string returned by the `getName`-method denotes "2D-Barcode". Calling the `getDisplayable`-method instantiates a new instance of the `CameraForm`-class. This class is described later in this document. In general it starts the phone's camera, takes a picture, decodes it and starts a connection with the information received from the barcode information.

Figure 6.18: The SyncTapConnection-class.

The **SyncTapConnection**-class (see figure 6.18) serves only as a dummy class for a second connection method. The `getName`-method returns the name "SyncTap-Verbindung" for this connection method, but calling the `getDisplayable`-method has no effect at all and simply returns `null`.

6.4.4 qrCode-package

The classes inside this package are used to establish connections using a 2D-barcode. The `CameraForm`-class is created in the `QrConnectionMethod`-class and for the decoding of the pictures taken by the user the `LCDUIImageMonochromeBitmapSource`-class is needed.

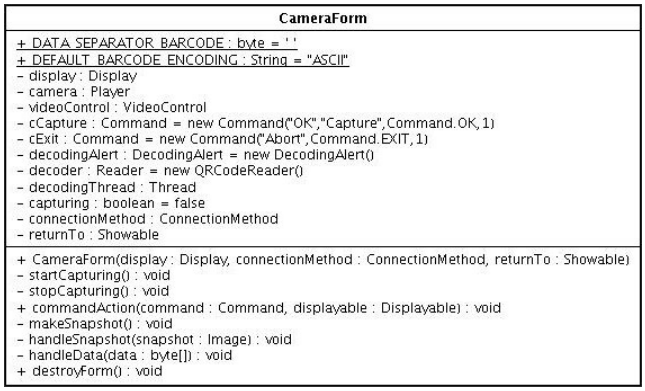

Figure 6.19: The CameraForm-class.

Whenever the user selects to establish a connection using a 2D-barcode, the **CameraForm** (see figure 6.19) is instantiated and an instance of it is displayed. The `CameraForm` extends the Java ME `Form`-class and adds an exit- and a capture-command to the soft-key-area of the display.

Inside the form the camera of the device is realized and the user is now able to point to a 2D-barcode and take a picture of it. Pressing the capture-button the `makeSnapshot`-method is called which stops the process of capturing an image and starts a thread to decode the image taken. Out of the snapshot an `LCDUIImageMonochromeBitmapSource` is created that serves as an input to the Google ZXing-API described in section 4.6. This API-tries to find and decode a QR-barcode in that image. If the decoding fails, the camera is put back on and the user has the possibility to take another picture. Otherwise the decoded byte-array is passed to the `handleData`-method. This method resolves the servers network address and optionally a public key hash as well as a authentication token from the byte-array. With this information a new connection method-instance is created that takes now control over the display and shows the status of the handshake process. After everything is completed successfully the connection-method-listeners are notified that the connection has been established.

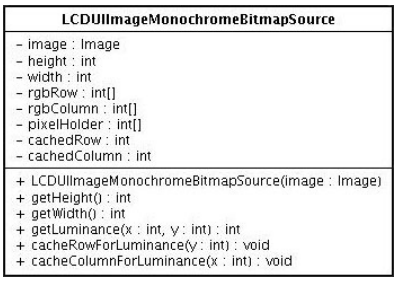

Figure 6.20: The LCDUIImageMonochromeBitmapSource-class.

Although the **LCDUIImageMonochromBitmapsSource**-class (see figure 6.20) is part of the `qrCode`-package the code of this class is not part of this thesis but has been created by the Google ZXing-Team. This class has been written as an extension for the `BaseMonochromeBitmapSource` that is the correct class for decoding an image with ZXing. Since this class is abstract it does not work together with images captured with Java ME. The class used here is not part of the original ZXing-distrbution but is used with an example client running with Java ME. For better understanding it has been placed in this package but the whole code including Googles copyright notice is untouched. The class offers certain methods to get information about the image such as width and height but also lunimance information at a certain point of the image (`getLuminance`).

6.5 Using the Client-Application

To get a more practical view on how the client application works this chapter shows two use-cases of the client application. The first use case “connection and entering data” in section 6.5.1 shows a basic example of a complete execution sequence using the mobile device to enter data. Establishing a connection with the public terminal and entering data using the auto-complete feature. Section 6.5.2 shows how the client application is used to manage the values stored for auto-completion.

Figure 6.21: An image series showing how to use the SeCuUI client.

6.5.1 Connection and Entering Data

After the client application has been started, the different connection-methods and the possibilities to modify the values stored on the mobile device are displayed. Selecting a connection-method – in this case the QR-Code connection – displays a dialog to establish such a connection. The user takes a picture of the two-dimensional barcode displayed on the screen of the public terminal and waits for the device to analyze this picture. In case the picture was not successfully analyzed he can take another one. After that a connection with the public terminal is established and all the form elements are synchronized to the mobile device. In this example the first form consists of a choice of radio buttons for choosing one of three different movies. The user selects one of them and then uses the display command “Weiter”. This command is represented as button on the public terminal. The button causes a new form to be loaded that displays the amount the user has to pay. The next page contains input elements for the user’s name and address. In case there are some auto-completion values stored on the device and they match the current data-type a drop-down-field with auto-complete items is automatically added behind corresponding form entries. The user can now fill in the field or select an entry out of the list of auto-complete-values, which is then entered into the field on both sides. In this example he uses the last name “Maier” out of this list. Since both devices are always synced the connection can be closed at any time.

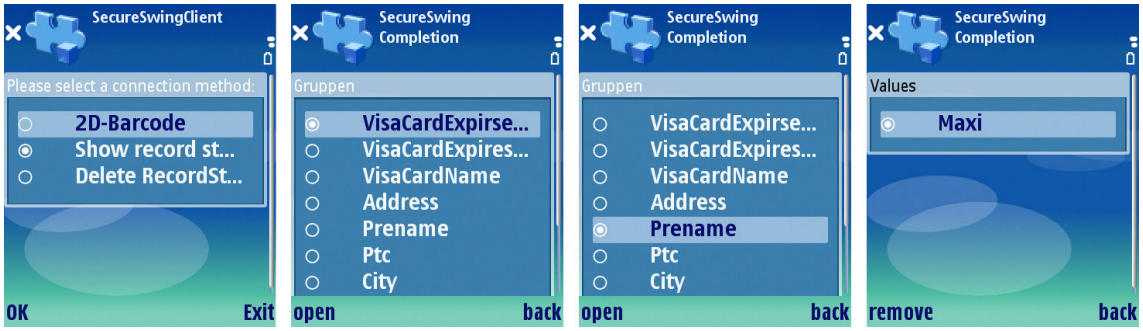

Figure 6.22: An image series showing how to manage the auto-complete values.

6.5.2 Removing an Auto-Complete Entry

Another use case is to modify the different data entries that are stored on the phone and that are used for the auto-completion process. Besides the different connection methods the user can select to show the record store in the main menu. After selecting this, a list of the different field categories is displayed. She selects one of those categories and presses “Open” using the left soft key. After that a list of the different values stored for this category is displayed. She can select one entry and delete it using the “remove”-button. The “back”-button always takes the user back one step. Changes to the items are immediately saved to the phone’s memory.

7 Evaluation

The SeCuUI framework and especially the client application were evaluated in a user study. The study was conducted with a server application that was created using the SeCuUI framework. This application prototype is elucidated in section 7.1. To give the user study a more realistic look two different fake-companies were created that have been used throughout the study. Their logos are shown in section 7.2. During the study the users had to complete five different tasks while the time it took them to complete those tasks was measured. Those tasks are closer described in section 7.3. The test setup and procedure for each participant is explained in section 7.4. For the study three different hypotheses were formulated. Those hypotheses can be found in section 7.5. After completing the five tasks the participants had to fill out an additional questionnaire. Its structure is shortly explained in section 7.6. Section 7.7 finally looks at the different results of the user study.

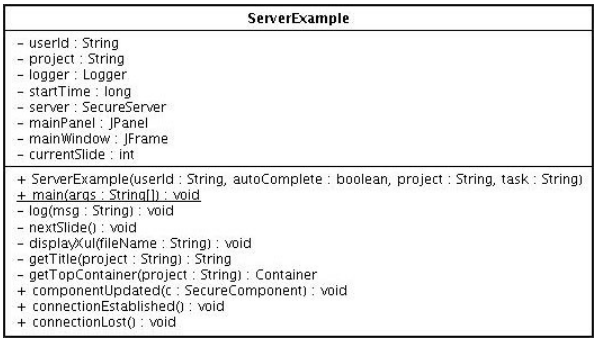

Figure 7.1: The ServerExample-class.

7.1 The Application Prototype

For the user study a prototype server application built upon the SeCuUI framework was created. The application was used to run all five different tasks with it. Mainly it served to measure the time it took each user to complete the different tasks. Besides this, the time of the connection process was measured, too. Each change to any of the components was also logged together with the time it occurred. An example of such a log-file has been attached to this document (see figure A.4).

The application itself is contained in a class named `ServerExample` (see figure 7.1). The `main`-method contained in this class first shows a dialog and prompts for a user number and a task. Besides the five tasks of the user study a test-task was created. This task was started first to show participants what the application looks like and to let them get a feeling how to use the application. Each participant was allowed to practice as long as he or she wanted to.

Depending on the user number and the task type an instance of the `ServerExample`-class was created. Showing either the “Pahn” or the “Kinomaxx”-logo (see section 7.2). The class created a logger-object first and logged which task type and user number was contained in this log-file. Also the status of the auto-complete feature and the project-type was logged. During the test itself, users had to fill in data into forms on multiple screens. These screens were handled as different “slides”. Each slide was represented by a XUL file and the different XUL files for each task have been numbered from zero upwards. Except for the “test”-task all “slideshows” started with a slide just displaying a button labeled “start”. With this button the participants started the measurement of time by themselves. Like this each participants measurements should be as identical as possible. More about the different task steps is explained in section 7.3.

After the program started, each time the users pressed the “next”-button at the end of the slides the `nextSlide`-method was called. This method increases a slide-counter and loads the

next slideshow into the main window using the `disaplyXul`-method. On the last slide the next-button was simply missing and the text “Vielen Dank” was displayed to fulfill the “closure”-principle [35]. One of the eight golden rules formulated by Ben Shneiderman.

Images of the different screens displayed during one task can be found at the end of the document (figures A.5 to A.12).

Figure 7.2: The company emblem of “Kinoxmaxx”.

Figure 7.3: The company emblem of “Deutsche Pahn”.

7.2 Deutsche Bahn and Kinomaxx

During the user study the users were asked to perform different task at a “public terminal”. Since the study was conducted at the premises of the department in a special room it was not so public at all. To give the participants a better reference to a real public situation two company logos were created. During the study tasks the users needed to buy something at the machine. The “Kinomaxx”-company – the fake logo is shown in figure 7.2 – should remind the user of a big German cinema chain originally called “Cinemaxx”. Some of the tasks referenced that company and the participants had to buy a cinema ticket out of three non-existing movie titles. In the other tasks the user needed to buy a card for special train ticket savings that is originally called “Bahncard” and is sold by the German railway company. For the study this company was called “Deutsche Pahn” and the product hence “Pahncard”. The alternative company logo is shown in figure 7.3.

7.3 The Different Tasks

During the study the users had to complete five different tasks. All of them required the user to buy something from a public vending machine. The machine was represented by an Apple Macbook on which a server application created for the study was running. For all of tasks the task procedure was nearly identical. Participants had to select the second out of three different products, notice the price for this product, enter their personal data – like first name, last name and address – and finally enter some credit card information. To distract people from the fact that they needed to do the same things over and over again the two fake companies explained in section 7.2 have been invented. This also should visualize that the SeCuUI client could potentially be used on any public terminal even among different companies.

The study was conducted having two independent variables with two values for each of them. The first variable was to activate or deactivate the auto-complete feature on the mobile phone. The second independent variable was the amount of data that was input using the mobile device. In one case all data was entered using the mobile device in the other case only the credit card number and

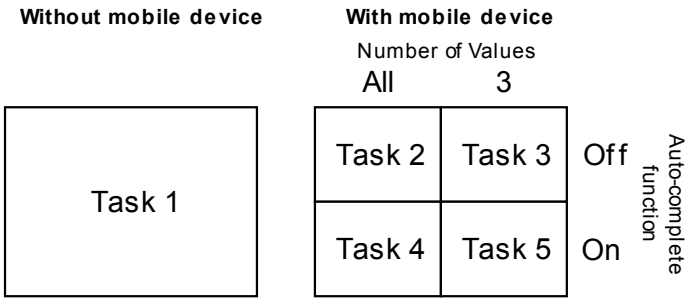

Figure 7.4: The five different tasks of the user study.

its expiry date was entered using the mobile device. An additional task used no phone connection at all. This task was called the reference task. This resulted in a total of five different tasks:

1. **No phone connection:** To measure the time how long it would take someone to enter all data without having any mobile device at all. The first task was performed by using only the keyboard and mouse of the test system.

2. **All values / no auto-completion:** After connecting the mobile device, the user had to enter every value using the mobile device. To do this she was not aided by the auto-complete values stored in the phone's database.

3. **Secure values only / no auto-completion:** In this task the participants were told to enter everything that is possible on the public terminal. Three of the fields the users had to fill out were blocked on the public terminal. The credit-card-number-field, and the expiry-date-fields (month and year). To fill in these three fields the users had to switch to their mobile device. Task description also instructed the participants to connect their mobile devices immediately when starting with the task even though they did not need to enter anything with the phone at first.

4. **All values / auto-completion:** The fourth task used the auto-completion function and so after connecting the mobile device users should enter all the data using this function. For each of the required fields the correct values were already present as an auto-complete value.

5. **Secure values only / auto-completion:** Task 5 again reduced the number of fields that had to be filled out to the same three ones as in task 3. This time the three fields should be filled in using the auto-complete-values. All other fields had to be filled out on the public terminal without having no auto-complete function there.

Figure 7.4 shows the five different tasks again in a diagram. One may notice that the reference task does not really fit into the task structure and so had no connection to the two independent variables. This made it hard to statistically evaluate this task against the other ones.

7.4 Test Setup and Procedure

As already mentioned the test was done at the "Lehr- und Forschungseinheit Medieninformatik". On two days 21 participants took part in the test. After their participation they were presented with a bar of chocolate. Each of them was separately tested under exact the same conditions. Figure 7.5 shows the test setup schematically. During the test only the participant and the conductor were present. The participants were first told, that they are going to participate in a user study concerning privacy and security on public terminals and that the current test setup should represent a public terminal. They should use it together with the mobile device as they would do in public. After that a document explaining the whole test was given to them. The document is attached

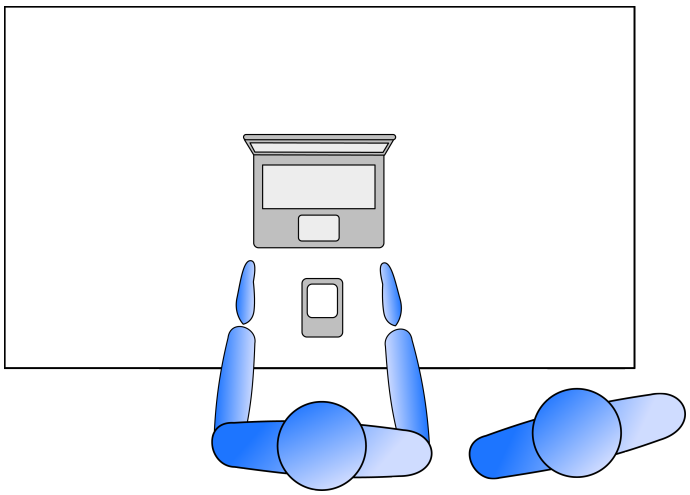

Figure 7.5: The test setup for this user study.

to this thesis (see figure A.13). After they had finished reading, they were shown how the system works and how to establish a connection between the two devices. After that the participants could practice entering data with the mobile device as long as they wanted to. When the participants finished this trial-phase they had to complete all five different task. None of them performed the tasks in the same order to avoid learning effects. The time the participants needed to connect the two devices and the overall-time it took them to complete the tasks was measured (see figure A.16). A document explaining what participants were to do in each task was handed to them just before each task in the order chosen for this participant. These task-descriptions are also attached to this thesis (see figures A.14 and A.15). After completing all tasks, the participants were asked to fill in a questionnaire providing information about themselves and how they felt using the system. See section 7.6 for more information. Figure 7.6 shows a student completing the questionnaire after the five tasks.

Figure 7.6: A student taking part in the user study.

7.5 Hypotheses

Subject of the user study itself was to prove different hypotheses. The hypotheses are taken into account when evaluating the results of the study in section 7.7. For the study the following three hypotheses have been formulated.

- **Hypothesis 1 (H1):** Using a mobile device in a more secure mode is slower than using an insecure public terminal.
- **Hypothesis 2 (H2):** Mobile input using an auto-complete function is faster than without.
- **Hypothesis 3 (H3):** Entering only some values with a mobile device is faster than entering all.

7.6 The Questionnaire

This section describes the structure of the questionnaire that had to be filled in by each participant. The results and different answers are looked at later in seciton 7.7. Participants filled in the questionnaire online, immediately after they took part in the user study and did their tasks. Like this they remembered best what they had just done and seen. The questionnaire had a sum of 46 questions whereas some of them were conditional questions that were optional in case of a certain answer beforehand. The questionnaire was composed of seven question categories. First some demographic data of the participants was collected. Things like gender, age or profession were prompted. The second part asked about their technical knowledge especially with vending machines and mobile devices. The third part of the questionnaire was dedicated to the possession and usage of mobile phones. In case a participant owned a mobile device he was asked about his usage and the function range of the device. The fourth block asked questions about the usage and experience with public terminals or vending machines and whether people preferred using machines over humans behind a counter. The test itself featured only one connection method but as mentioned above, SeCuUI is capable of a large number of such methods. Due to this fact part five of the questionnaire asked the participants if they could image to use other connection methods and which one they would prefer. Without prior knowledge of the participants different fields during the tasks shifted their state depending on the phones connection status. The amount for the product they were about to buy for example was only shown on the public terminal in case there was no mobile device connection. In part six of the questionnaire the users were asked whether they noticed this behavior and what they were thinking about it. The last part of the questionnaire featured an overall recapitulation of the test and its different tasks. It also contained the most important question: Would people use the system if it was really available in public. Finally at the end of the questionnaire additional comments could be made. Despite those comments the participants remarks during the study were also noted. Some of the them are being cited in section 7.7. A complete list can be found in figure A.29.

7.7 Results

This section describes the different results of the user study in-depth. Conclusions are drawn on an overall-basis but to get a better perspective on the different results this section is split in eight subsections referring to the seven categories of the questionnaire and section 7.7.5, which focuses especially on the results of the user study tasks. At first in section 7.7.1 the demographic profile of the participants is described. Section 7.7.2 shows how the participants rated their own technical experience. The next two sections describe the mobile device usage and the usage of vending machines (see sections 7.7.3 and 7.7.4). After that – in section 7.7.5 – the different tasks, their results, and a statistical analysis of those are presented. Section 7.7.6 shows the users' opinion to the connection method they used and to other proposed connection methods. In section 7.7.7 the results of the questions concerning the asterisk- and security-mode are evaluated. Finally the overall-ratings of the participants are stated in section 7.7.8 and the different hypotheses are verified. During the study some suggestions and possible improvements have been figured out. The most important of them are described in section 7.7.9.

7.7.1 Demographic Evaluation

Although 23 people completed the five different tasks only 21 of them have been evaluated lately. The other two participants did not complete the questionnaire and their data has been removed for this reason. The male/female ratio among the participants was nearly even with 11 male participants and 10 female participants. The average age of a participant was 27 with standard deviation of 9.0 whilst the youngest participant was 15 and the oldest 61 years old. 12 of the 21 participants were university students. The others had different professions. Looking at the highest education level of the participants. 11 participants had passed the "Abitur" whilst seven people already had a university degree. 1 participant had so far only finished primary school and two others had the german "Realschulabschluss". The data is also visually presented in figure 7.7.

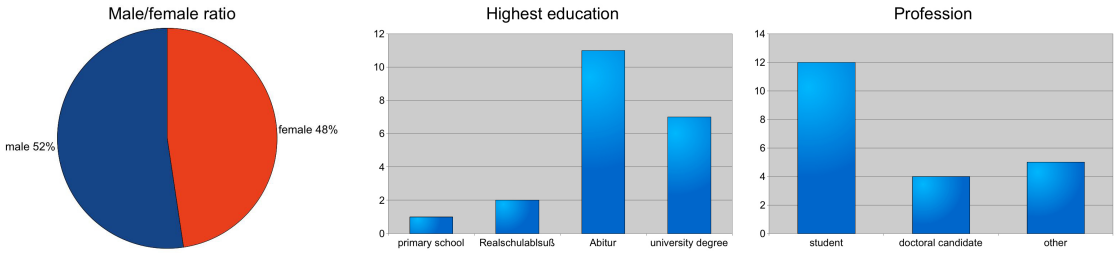

Figure 7.7: Demographic data of the participants.

7.7.2 Technical Abilities

Looking at the technical abilities of the participants each of them should rate their own technical expertise on a five point Likert-scale from 1-'very bad' to 5-very good'. The first question asked for the general or overall technical abilities. The average answer was 3.76. The next question in this category asked for the technical abilities concerning computers. Here the participants answered with 3.9 in average. Concerning their mobile device expertise the average value was 3.76 again. Since all values have reached nearly four points in average the participants attested themselves very good knowledge in technical things. The high number of graduates can be a reason for this as well as that people attending the study were quite young. Details on the answers in this category are shown in depth in figure 7.8.

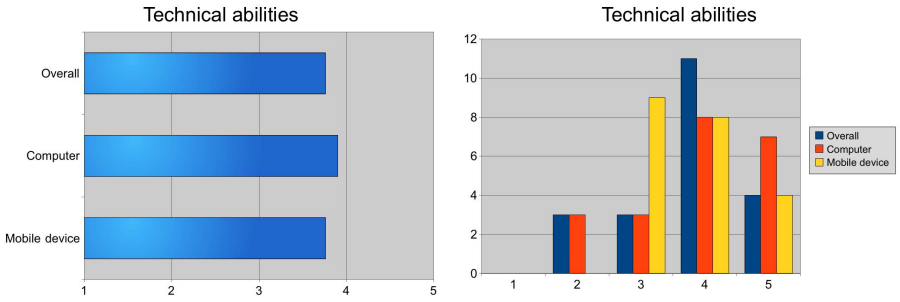

Figure 7.8: Self-assesment of the technical expertise.

Participants were also asked how worried they are about their security concerning different aspects of their lives. Again this was answered using a five point Likert-scale ranging from 1-'no worries at all' to 5-'I'm often thinking about that'. In average, participants were most worried about two different points that averaged to the same degree with 3.81. The first reason that made the participants worry is the security at vending or cash machines. Another topic people were concerned about is the misuse of their personal data for commercial purposes, spam emails or even

stalking. Immediately following those two topics was the security of online-banking applications. Here an average Likert-value of 3.71 occurred. Traditional theft of personal goods had an average value of 3.38. This shows that the participants worry more about loosing private data that could be misused instead of the theft of personal belongings. The compared averages and the exact answers are shown in figure 7.9.

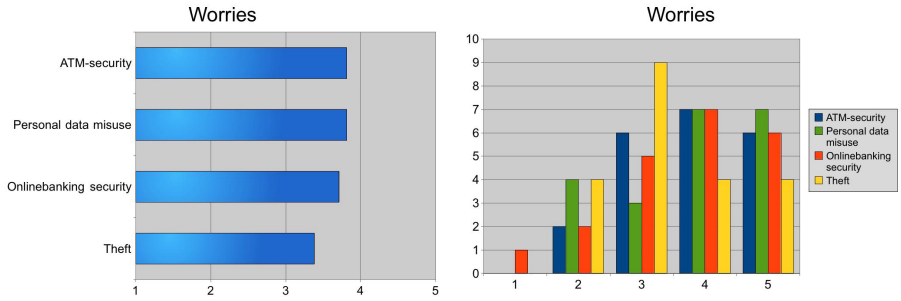

Figure 7.9: Self-assesment of the personal worries.

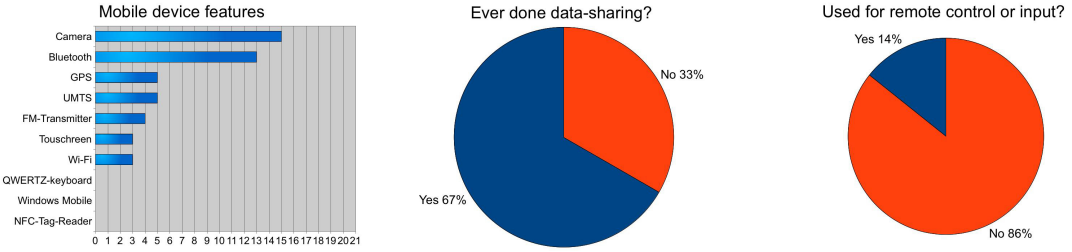

Figure 7.10: Features and usage experience of mobile devices.

7.7.3 Usage of Mobile Devices

All of the 21 participants that completed the study were owning a personal mobile device. To be able to use a mobile device at a public terminal it would not only be important to own such a device but also to carry the device with one all the time. When being asked how often they take their mobile device with them 18 of the 21 phone owners answered that they are carrying their device “(nearly) always”. The remaining three people take it with them at least “multiple times a week”. Other possible answers to this question were “never”, “once per month”, “once per week”. This shows that a huge number of people today own a mobile device and they normally carry it with them at least multiple times a week.

The questionnaire contained a list of different mobile device features and participants were asked to fill in which of these features were supported by their own device. Most often peoples’ phones had a camera (71 percent) immediately followed by bluetooth functionality (62 percent). Some people had WLAN, GPS, UMTS or a touchscreen. Nobody answered that he or she has a QWERTZ-keyboard, a windows mobile phone or an NFC reader. Figure 7.10 shows this in more detail. Depending on whether the people answered that their phones had bluetooth or a camera they were asked to additionally say how often they are using that feature. Possible answers were: “never”, “once a month”, “once a week”, “(nearly) always”. Out of the 15 people that did have a camera on their phone nobody said that he or she is using it nearly always. Four people use their camera once a week, 10 people once a month and one participant never uses the camera although the device is able to take pictures. One of the 13 bluetooth users has the bluetooth switched on all the time. Two others “once per week”, five people “once per month” and one person never uses the bluetooth functionality at all.

To see whether people had experience connecting their devices with other devices a general question asked: “Have you ever used your device to share data with others?”. 14 participants – that is two thirds – did this already while the last third of participants has never done this before. The most common action when sharing data on a mobile device is to send contact information from one phone to another, but using the phone to remotely control another device is something different. Due to this reason the questionnaire also asked for “remote control or input”. This had so far just be done by three of the participants. The other 18 participants did this for the first time when attending the study. Figure 7.10 contains diagrams with more details on the different results.

7.7.4 Usage of Vending Machines

SeCuUI is intended to be rolled out on various kinds of vending machines. The framework allows programmers to make nearly every application capable of SeCuUI. This is why the questionnaire also asked participants how they use vending machines today. The first question asked how often people use different kinds of machines. Again people could choose between “nearly never”, “once a month”, “once a week”, “multiple times a week”, “daily”. More than half of the participants use an ATM machine once a week (11 out of the 21 participants). Five people said they use them multiple times in a week and the remaining five participants only use them about once a month. Like this ATM machines are the most commonly used machines. Ticket machines for public transport are used more seldom. Six people barely never used them, eleven people once a month. Three people said they buy their public transport tickets once a week and one person even buys those tickets multiple times a week. Eventually this question was influenced by the fact that most of the participants of the study came from the proximity of Munich. In a city of this size far more people use public transport and hence have to buy their tickets at such machines. Vending machines for food or beverages are used nearly never by 9 people. Six people buy something out of those machines once a month, 4 people once a week and 2 people use them multiple times a week. Asking for other ticket vending machines used to sell cinema or event tickets 15 people nearly ever use those machines, whilst six people do so once a month.

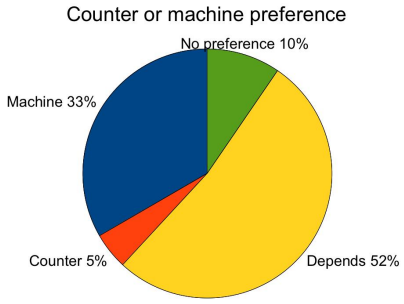

Figure 7.11: Counter or machine preference.

All those questions gave information on how often vending machines are used. But in most cases everything that can be bough or done using a machine can also be acquired in a more classical way asking a clerk behind a counter. One question of the questionnaire asked the participants whether they prefer using a machine or not. For this question one third of the participants (this means 7 people) agreed with the fact that they prefer machines over someone behind a counter. One person actually preferred someone behind a counter in every case. For 11 of the 21 participants using a vending machine or not depends on the situation and only two people had no certain preference when they had to choose. Diagram 7.11 shows this distribution. To get to know more about the motives that matter most to the participants when deciding what to use the seven people that chose “vending machine” as an answer were asked again. This time they could select multiple reasons for the fact they prefer machines. Six of them agreed with the statement “I’m getting faster

what i want". The statement "I already know what I need to do" was also chosen by 6 of the 7 vending machine users. Only one person did not like to have personal contact to someone working behind a counter an therefore uses vending machines. All of the seven vending machines users agreed that using vending machines is faster than asking someone behind a desk. As a last option the questionnaire offered the statement "I normally have no other possibility than using a vending machine" but nobody agreed with that. The one person that always preferred talking to a human was asked similar question customized to his prior answer. The only reason the single participant checked from the list was the third one denoting: "I think vending machines are impersonal".

Earlier in the questionnaire the participants had been already asked for their different worries they have, concerning security of public terminals. This time the question was asked again less subtle: "Have you every worried about your security at such machines?" 13 people agreed with that whilst eight denied. All of them who said they are worried where asked to express their worries. The different answers were very similar to each other. Nearly everyone mentioned that he or she was worried of the fact that someone could steal personal data that is entered to a machine. Some people even had ideas of distinctive methods how this could happen like using a miniature camera or copying the credit card data.

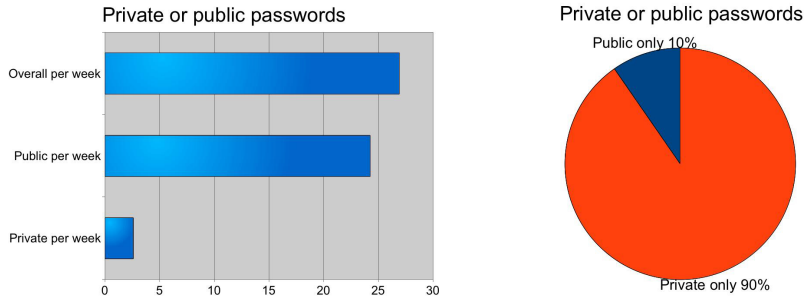

Figure 7.12: Private or public passwords.

The most important thing that needs to be protected in public is passwords of a user. Stealing a persons ATM card does not make any sense unless the thief does know the user's secret PIN, too. To get an approximate value of how many passwords people do use in everyday-life all participants where asked to enter how many passwords they enter each week. The first question asked for the number of passwords that are entered overall so in public and in private. Passwords that are used automatically without the user's knowledge should not be counted. Answeres to this question ranged from 1 to 100 and had an average value of 26.9 passwords each week. Since entering a passwords in private is not as dangerous as doing it in public – which is as well one of the key aspects of SeCuUI – a second question asked how many of those passwords are entered in in public. Here a minimum of 0 and maximum of 10 was registered whilst an average participant enters 2.6 passwords in public each week. This means that approximately 10 percent of the passwords are entered in public. See figure 7.12 for details.

Section 7.7.5 clearly shows that although the study showed good speed results for SeCuUI, it is still not as fast as a traditional PIN entry due to some overhead. To get to know what people expect at a vending machine they were asked to rate a list of non-functional requirements and decide on a Likert-scale from 1-'unimportant' to 5-very important' how important the different requirements appear to them (see figure 7.13). The thing that appears most unimportant to the participants is the "design" of such a machine. This non-functional requirement had an average score of 2.67. The most important things to the participants were "security" with 4.52, "simplicity" with 4.38 and "speed" with an average value of 4.24. For SeCuUI this means that security has top priority for customers using public terminals and so SeCuUI is heading in the right direction by making input more secure.

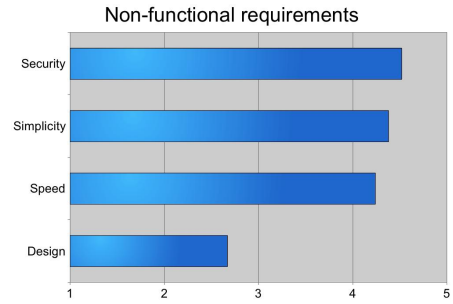

Figure 7.13: Non functional requirements.

time in seconds	Average connection	Average overall	Task 1 ratio
Task 1	0	76	
Task 2	28	238	3,3
Task 3	27	178	2,4
Task 4	35	129	1,8
Task 5	29	163	2,3

Average connection time	29,74

Figure 7.14: Average time taken for each task.

7.7.5 Task Results

So far the questions answered by the participants had no direct connection to the tasks they did just before filling out the questionnaire. During the user study the participants had to complete five different tasks in random order. For each of those tasks two times were measured automatically by the system. The overall time needed to complete the task and additionally for task two to five the time the user needed to successfully establish a secure connection. The measurement for this value started when the user clicked on the “Sicher verbinden”-button shown in figure A.7 and was stopped when the handshake process was completed successfully. A table showing the different connection and task times is included in the appendix (see figure A.16). Having a closer look at those values the average time for task 1 (the participants had to enter all values without a mobile device connection) is 76 seconds with a standard deviation of 29 seconds. Task 2 required them to use the mobile device for all of the values without having the auto-complete function enabled. This took users the longest time with an average 238 seconds (SD 76 seconds). Comparing this with the reference task participants needed 3.3 times more time to complete this task. Task 3 still did not involve the auto-complete function but it reduced the number of fields entered using the mobile device to three. This resulted in average time of 178 seconds (SD 52 seconds). That is still 2.43 times slower than our reference task 1. The next two tasks activated the auto-complete function an gave the users a list already containing the correct value for each field. In task 5 this feature had to be used for three fields only and the rest of the data had to be inserted using the public terminals keyboard. This task again took 163 seconds (SD 48 seconds) with a ratio of 2.32 times slower than task 1. The method that had the best results when using SeCuUI for terminal input was task 4. Still slower than task 1 the participants completed it in 129 seconds (SD 42.1) in average. This means that the reference task 1 was just 1.77 times faster than task 4 where participants could stay looking at their mobile phone while entering all the values using the auto-complete function. The different task times and ratios are again shown in figure 7.14. Figure 7.15 also shows a Box-Whisker diagram of the different task-execution times.

Having a look at the different connection times one can see that all connection times are

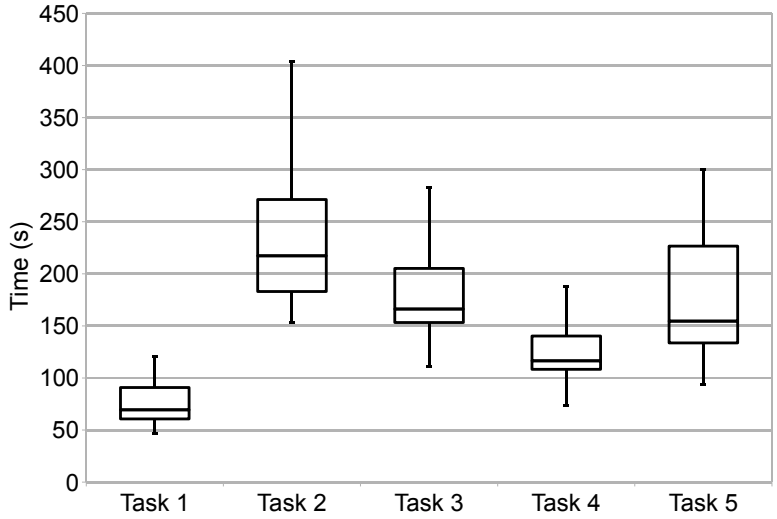

Figure 7.15: Average time needed to complete the different tasks.

roughly the same with an overall average of 29.7 seconds. This duration is relatively big and such is one reason why all tasks using the mobile phone took that much longer.

As mentioned earlier it is hard to perform a statistical comparison of the four different tasks using SeCuUI (2 to 5) and task 1 because task 1 has no common independent variable it shares with the other tasks. What can be done is to have a look at the two independent variables and their impact on tasks 2 to 5. When entering only some of the values using the auto-complete function (this means task 3 and 5) a statistical analysis shows, that that the time differences measured here were not statistically significant. In contrast entering all information using the auto-complete feature the time measured is highly significantly faster ($t(18) = 9.76$, $p<.001$) than doing all the input without an auto-complete function. In this case two participants were removed from the test because they were outliers. Another interesting aspect is that although task 4 is slower than task 1 participants had the impression that both tasks took the same time (see section 7.7.8).

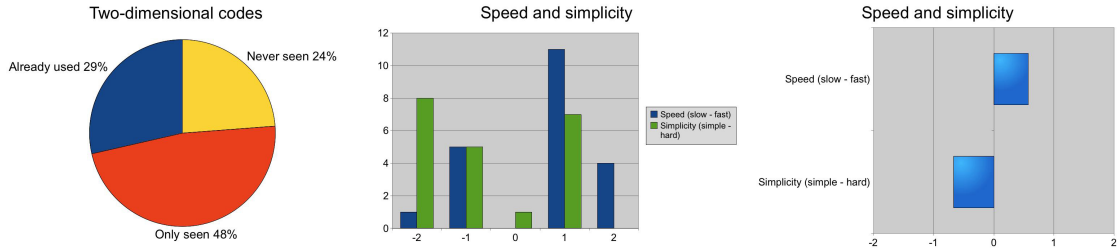

Figure 7.16: Knowledge of QR-Codes and non-functional analysis.

7.7.6 Different Connection Methods

During the study the participants had to connect a mobile phone to a public terminal. Four of the five tasks involved such a connection, which was always the same type of connection method: The QR-Code connection.

The first five questions in this section of the questionnaire covered this connection method. First participants should state whether they saw a two-dimensional barcode before. 76 percent (or 16 participants) did so. Those 16 were asked again in which context they saw such a barcode. The were no predefined answers to this question but most participants mentioned the internet-tickets

of the German railroad company. Additionally they appeared together with stamps on letters and on commercial billboards. The 16 people were also asked whether they had not only seen those codes but had also used them. Six people did so before, ten not. Now everyone was asked about the non-functional qualities of the connection-method. On a five-point scale ranging from 'slow' (-2) to 'fast' (+2) an average value of +0.57 was reached. Though people tended to say it was a 'little fast' this shows that speed could be optimized especially for the connection procedure. The other non-functional requirement was the simplicity. Again on a scale ranging from 'simple' (-2) to hard' (+2) results showed an average of -0.67, which also shows that the method is not perfectly simple. The diagrams in figure 7.16 visualize those results.

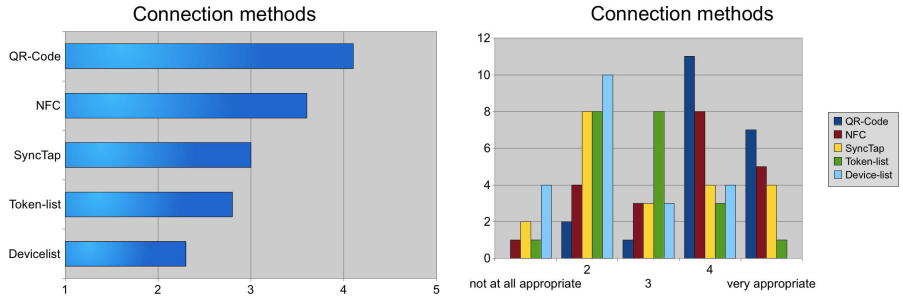

Figure 7.17: Appropriateness of different connection methods.

But as mentioned above the QR-Code connection is not the only possibility to establish a connection between the mobile device and the public terminal. It would be possible to add many other connection modules to SeCuUI. Because of this, the questionnaire asked the participants how appropriate they think would different connection methods be. For five different connection methods people should select the methods appropriateness ranging from 1-'not at all appropriate' to 5-'very appropriate'. The QR-Code connection was again included in this list and received a maximum average value of 4.1. Each method was described to the participants in one sentence. An NFC-method described as: "The device is brought to close proximity with the terminal and connects automatically." scored with 3.6 in average. The SyncTap connection described in section 2.4.1 got an average value of 3.0. The two last methods proposed a device list of surrounding bluetooth devices as it is done today for most bluetooth connections. One method proposed to display a user-image instead of the device's name. It received 2.8 in average. The plain device-list using text had 2.3 in average. The sum of answers is shown in figure 7.17.

7.7.7 Security and Asterisk-Mode

The next set of questions covered the fact that depending on the `asterisk` and `security`-values in the XUL document different interface components acted differently. During the experiment the participants were not told that. Because of this the first question here asked whether they noticed a change of the fields behavior during the different tasks or not. 86 percent or 18 people answered 'Yes'. The next question explained the fact that some fields changed their behavior to block input on the terminal side. Participants were asked how they felt about this. 11 people chose the answer "I think that's meaningful. Like this one immediately knows which fields are safety-ciritical". Eight people chose the second answer: "I would always like to decide for myself what values to enter where". The third possibility was chosen by 2 participants: "In general blocking security related fields is good, but during the test the fields were badly chosen." This shows that the participants do not really agree with each other for the one or the other methods. Any developer therefore should only block the fields when it is really necessary because with each normal field users can select for themselves where to enter the value.

During the test the amount that had to be paid was clearly visible on the terminal just in case

there was no mobile connection. As soon as it was established the amount was replaced by asterisks on the public terminal and shown instead on the mobile device. This behavior depends on the `astersisk`-value used in the user interface description. The next question drew the participants attention back to this fact and asked them how they felt about such a method: This time a majority of 15 people meant: “I think thats meaningful, [...]”. Five people disagreed selecting “I always like to see all values on the screen, [...]”. One person chose the last possibility: “In general I think that’s a good idea, but in this case it was not properly used”.

7.7.8 Participant Overall-Rating

The last part of the questionnaire summarized the user study and its tasks. Participants should say whether on the whole entering data with the mobile phone is helpful. 15 participants or 71 percent decided ‘Yes’ whilst six other thought No’. For the auto-completion feature everyone – all 21 participants – agreed that this feature is helpful for entering data with the mobile phone.

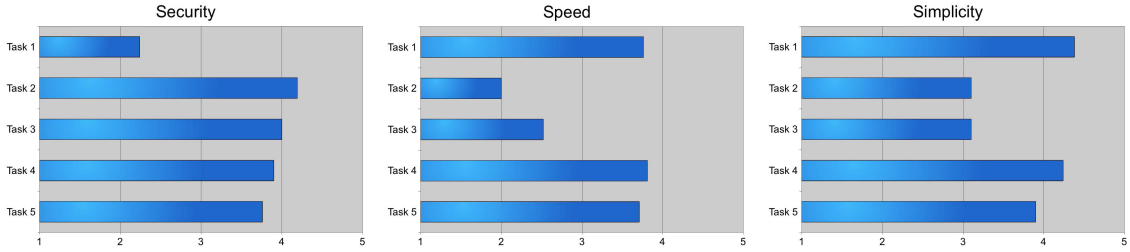

Figure 7.18: Ratings of the different tasks according to the non-functional requirements “security”, “speed” and “simplicity”.

Now each participant should rate for each of the tasks how well the following non-functional requirements, “security”, “speed” and “simplicity” were implemented. For each of the three requirements the five tasks were named as follows: “Input only at the public terminal”, “Input of all values using the mobile device (no auto-completion)”, “Input of security-related values using the mobile device (no auto-completion)”, “Input of all values using the mobile device (with auto-completion)”, “Input of security-related values using the mobile device (with auto-completion)”. For the non-functional requirement “security” participants should select on a Likert-scale from 1-‘not secure’ to 5-very secure’. Task 2 was rated the most secure with 4.19 in average followed by task 3 with 4.00, task 4 with 3.90, task 5 with 3.76 and with over 1.5 points less task 1 with 2.24 in average. This clearly shows participants are aware of the fact that entering data with a mobile phone is a lot more secure than using keyboards installed at a public terminal.

In case of the non-functional requirement “speed” the Likert-scale ranged from 1-‘very slow’ to 5-‘very fast’. Task 4 was rated best with 3.81 followed by task 1 with 3.76, task 5 with 3.71, task 3 having 2.52 and task 2 with 2.00 in average. This is very interesting because the auto-complete method using the mobile phone and the method with no-phone usage at all reached the same level. Looking at the time measured for the tasks (see figure 7.14) one can see that task 4 took in fact 1.8 times as long as task 1 in average but it seems the participants did not notice. This again shows that though connecting a mobile device first costs some time people seem to have a certain amount of tolerance for the loss of speed.

“Simplicity” was the last non-functional requirement requested. The possible answers ranging from 1-‘very complicated’ to 5-‘very easy’. In this case again task 1 and task 4 are very close together. Task 1 leading with 4.38 followed by task 4 with 4.24 in average. Task 5 got 3.9 points and task 2 and 3 averaged both with 3.1. The results of all three groups are shown in figure 7.18.

Finally participants should tell which one of the methods they just used during the different tasks they would use from now on in public if they were given the choice. In sum, 86 percent of the

participants selected one of the four different security-enhanced methods. The value is composed of 4.8 percent that would like to enter everything with their mobile device but not having an auto-complete feature, 38.1 percent that liked to input all of their values using the mobile phone but having auto-completion activated and finally 42.9 percent wanting to enter only security-relevant values using their device but having the possibility of auto-completion there. Nobody decided that he or she wants to enter only some of the values with an auto-completion value in the future. The remaining three participants or 14.3 percent of all of the participants chose to stick to the old way of not using any mobile device at public terminals.

Having a look at the different hypotheses formulated in section 7.5 one can finally say the following. Although SeCuUI is able to speed up the input process using a mobile device drastically by using the auto-complete function it is not able to beat the average time of the reference task. This confirms hypothesis H1 and so using the more secure method is still slower than using the more insecure public terminal only. For Hypothesis H2 one can say it has also be confirmed because entering information with an auto-complete feature is even significantly faster than entering information without this feature. Looking at the results of hypothesis one and two one might think this leads automatically to hypothesis three but when looking at a statistical analysis of the times needed to enter the values no significance is found. This might be du to the fact, that the connection time and attention shifts between the public terminal and the mobile device cost more time than the little faster input at the public terminal benefits.

7.7.9 Suggestions for Improvement

During and after the test different small mistakes and improvements have been noted. Comments made by the participants during the test have been noted, too. A complete list of those comments can be found in figure A.29. The most important things that should be considered in future studies are:

- **Building a more realistic public-terminal environment:** During the study only a notebook represented the public terminal. In fact the study was laid out to mimic two different public terminals of two different companies. It would have been better to build up two independent systems looking more like real vending machines and to use another input device on those "public terminals" that would be more common to today's machines. This could have been an on-screen-keyboard or an industrial aluminum-keyboard. With the normal-keyboard and mouse-input-method that most participants knew from everyday work at their companies the input times have probably been much faster especially for task 1 than they would have been on a touch-screen keyboard.

- **Standing instead of sitting:** Some participants said that when they normally use a public terminal they do not have the possibility to sit during their input. Conducting the study with all participants standing in front of the "public terminal" would have also been more realistic.

- **More values in auto-complete lists:** Whenever the auto-complete function was used it contained always only one value which always was the correct one. Participants said they would have expected a larger list of values. For this study this was done on purpose. A user who would use SeCuUI with her personal device would normally use the auto-complete function only for herself. This is why one can assume that on a normal device there would always be only one first name or one last name stored. In case of credit card informations or similar things it could be possible that someone owns multiple items of the same kind but despite this, one would never have more than three values per category.

- **Data-type should change keyboard settings:** Some of the fields users had to enter required only digits (zip-code or credit-card-number). On the small nine-digit phone keyboard en-

tering numbers is complicated as long as the phones keyboard is expecting letters to be entered. To help the participants each of them was shown how to switch to the "number entering mode" using the hash-key. In a future version it would be good to already set this mode for certain `dataTypes`.

- **Synchronizing focus and caret:** During the study changes made either on the mobile device or public terminal were immediately synchronized. But the position of the caret and the current input field focus where not. Like this it happened a few times that someone selected a field on the public terminal and then started to write with his mobile device. Due to the fact that the focus change was not synchronized he now wrote to the input field last selected on the mobile device. This normally was not intended by the participant.

8 Conclusion

This thesis presented the development process of an application suite called SeCuUI (Secure Custom User Interface). The suite consists of a framework for programmers to built server applications for public terminals. Applications built with this framework are automatically compatible to a client application developed as a second application of the suite. With this client application installed on the user's mobile device he or she is in charge to fill in and control every form-component of a public terminals.

To speed up the entry process and make it more comfortable to use the system the data a user enters is saved according to a specific topic or group this entry belongs to. On any other public terminal that requires an input of the same category the mobile phones automatically suggest prior made entries to the user.

8.1 Results

With this functionality SeCuUI represents a software that not only fulfilled the goals set up in section 1.2 it also demonstrated a balancing act between two different areas of research in this field. So far researches mostly thought of security on public terminals either in a microscopic way – inventing new mechanisms of password entry – or using a macroscopic approach – building systems to completely giving secure remote control over public terminals –. SeCuUI stroke a balance bringing some part of the remote user interface to the mobile device but in a special mobile device conform manner. By using an auto-completion feature SeCuUI was also able to address the issue of the big amount of additional time that more secure input mechanism usually take.

During the evaluation of a user study it was found out, that people today are already somehow devoted to the use of vending machines and public terminals. Despite security of their personal data especially at such machines is one of their biggest worries. Finally a sum of 86 percent of the participants of the user study chose to use one of the security enhanced methods they were shown from now on if possible. One last small disadvantage remains: Even the fastest security enhanced method used during the user study still is 1.8 times slower than not using any mobile device at all. Although this speed difference is important and should be taken into account for future research the participants themselves seemed not to notice it because they rated both methods equally fast when asked for their experience.

8.2 Future Work

This work presented an attempt to something one could also describe as "remotely used user interfaces". The results of this thesis showed that it is possible to enhance security in public space using such a method. Since not much research on such systems has been done so far is important to further inspect this topic in the future. Even SeCuUI itself could be improved by extending the module concept with new modules and performing a broader user study in a more realistic environment. The current user study with only 21 participants should be repeated with a larger number of participants focussing more on the difference between the basic input at a public terminal and only one chosen method of security enhanced mobile device usage. Also the setting of the user study should be more realistic, eventually even rolling out the system in a field study at a real public terminal.

In general one can say that although research for better and more secure interaction with public terminals is going on, for many years now the classic PIN entry as it was introduced with the first ATM machine in 1967 is still in use. This fact should always be a motivation for further research in this field.

A Appendix

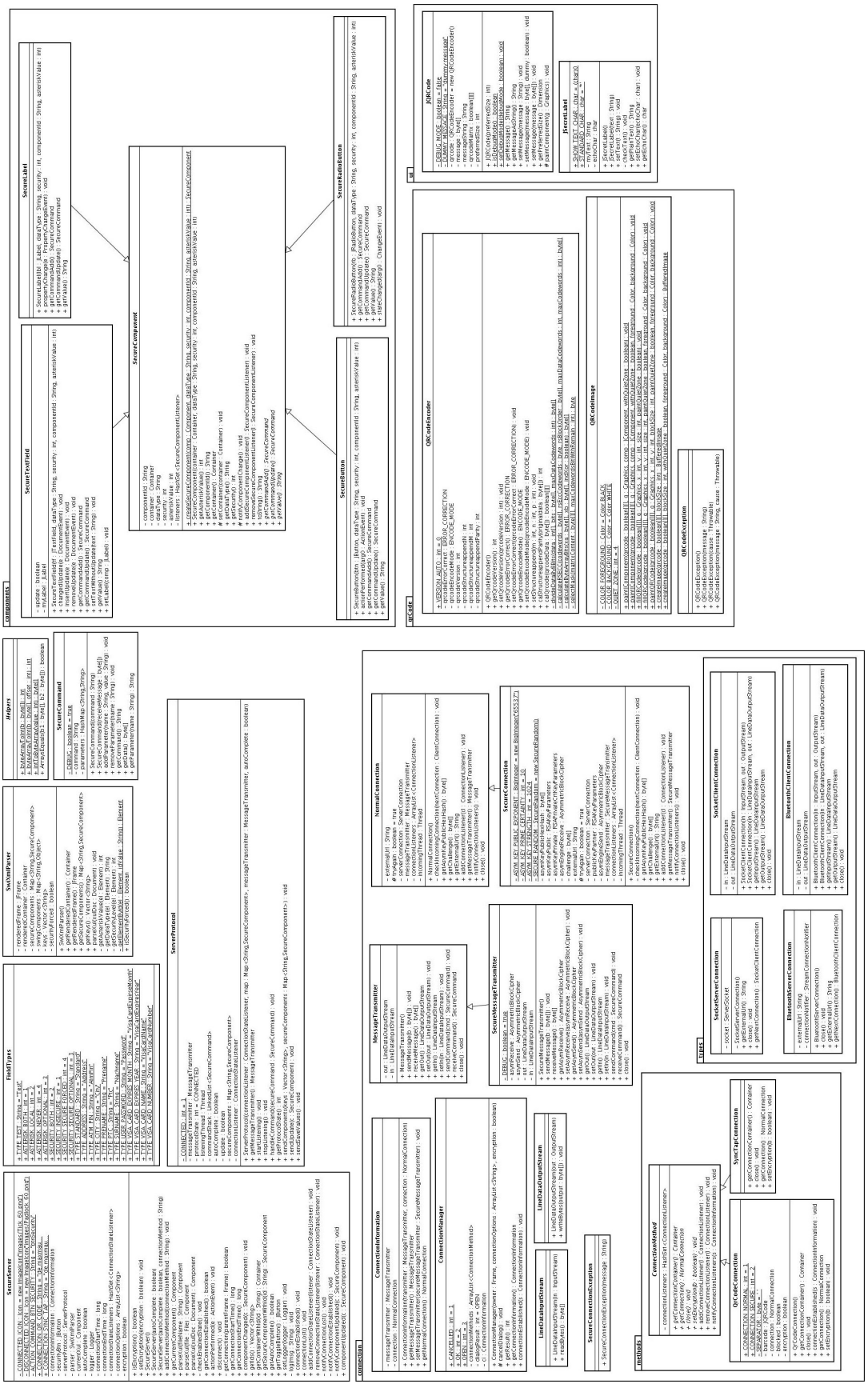

Figure A.1: All framework classes shown in a UML diagram.

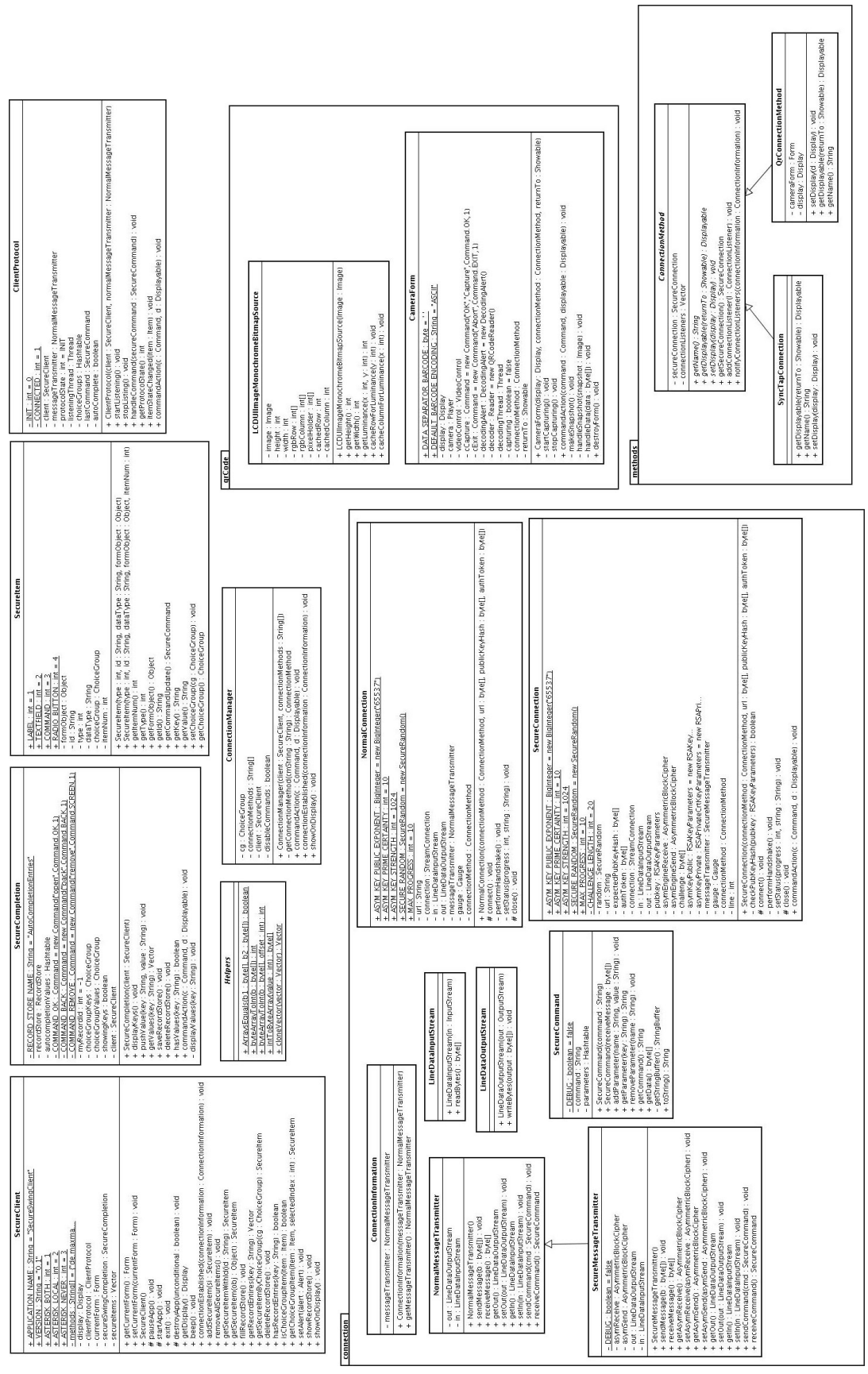

Figure A.2: All client classes shown in an UML diagram.

TYPE_STANDARD
TYPE_ATM_ACCOUNT
TYPE_ATM_PIN
TYPE_ADDRESS_CITY
TYPE_ADDRESS_COUNTRY
TYPE_ADDRESS_PTC
TYPE_ADDRESS_PROVINCE
TYPE_ADDRESS_STREET
TYPE_BODY_EYE_COLOR
TYPE_BODY_GENDER
TYPE_BODY_HAIR_COLOR
TYPE_BODY_HEIGHT
TYPE_BODY_WEIGHT
TYPE_CREDIT_CARD_CODE
TYPE_CREDIT_CARD_EXPIRES_MONTH
TYPE_CREDIT_CARD_EXPIRES_YEAR
TYPE_CREDIT_CARD_NAME
TYPE_CREDIT_CARD_NUMBER
TYPE_MASTER_CARD_CODE
TYPE_MASTER_CARD_EXPIRES_MONTH
TYPE_MASTER_CARD_EXPIRES_YEAR
TYPE_MASTER_CARD_NAME
TYPE_MASTER_CARD_NUMBER
TYPE_NAME_FIRST
TYPE_NAME_SECOND
TYPE_NAME_LAST
TYPE_USER_PASSWORD
TYPE_VISA_CARD_CODE
TYPE_VISA_CARD_EXPIRES_MONTH
TYPE_VISA_CARD_EXPIRES_YEAR
TYPE_VISA_CARD_NAME
TYPE_VISA_CARD_NUMBER

Figure A.3: The list of standard Type names that can be used with the `dataType`-attribute.

```
007;;0;;TASK;;3
007;;0;;AUTOCOMPLETE;;false
007;;0;;PROJECT;;kinomaxx
007;;0;;COMPLETE
007;;25558;;COMPLETE
007;;27309;;GET_CONNECTION
007;;47261;;CONNECTED
007;;51828;;CLIENT;;btn_next_1;;Weiter
007;;51982;;COMPLETE
007;;63000;;CLIENT;;btn_next_1;;Weiter
007;;63142;;COMPLETE
007;;67067;;CLIENT;;tf_prename;;
007;;67915;;CLIENT;;tf_prename;;M
007;;69246;;CLIENT;;tf_prename;;A
007;;70416;;CLIENT;;tf_prename;;
007;;77911;;SERVER;;tf_prename;;Ma
007;;78152;;SERVER;;tf_prename;;Max
007;;78296;;SERVER;;tf_prename;;Maxu
007;;79135;;SERVER;;tf_prename;;Maxu+
007;;80367;;SERVER;;tf_prename;;Maxu
007;;80599;;SERVER;;tf_prename;;Max
007;;81191;;SERVER;;tf_prename;;Maxi
007;;81327;;SERVER;;tf_prename;;Maxi
007;;82511;;SERVER;;tf_surname;;M
007;;82688;;SERVER;;tf_surname;;Ma
007;;82815;;SERVER;;tf_surname;;Mai
007;;83023;;SERVER;;tf_surname;;Maie
007;;83287;;SERVER;;tf_surname;;Maier
007;;84441;;SERVER;;tf_adress;;M
007;;84535;;SERVER;;tf_adress;;Ma
007;;84831;;SERVER;;tf_adress;;Mar
[...]
007;;140388;;CLIENT;;tf_visa_name;;Maxi
007;;141669;;CLIENT;;tf_visa_name;;Maxi
007;;142163;;CLIENT;;tf_visa_name;;Maxi m
007;;142553;;CLIENT;;tf_visa_name;;Maxi ma
007;;143051;;CLIENT;;tf_visa_name;;Maxi mag
007;;143730;;CLIENT;;tf_visa_name;;Maxi mah
007;;144426;;CLIENT;;tf_visa_name;;Maxi mai
007;;144565;;CLIENT;;tf_visa_name;;Maxi maid
007;;145309;;CLIENT;;tf_visa_name;;Maxi maie
007;;145496;;CLIENT;;tf_visa_name;;Maxi maiep
007;;145628;;CLIENT;;tf_visa_name;;Maxi maieq
007;;149679;;CLIENT;;tf_visa_month;;
007;;150112;;CLIENT;;tf_visa_month;;.
007;;151101;;CLIENT;;tf_visa_month;;.a
007;;151649;;CLIENT;;tf_visa_month;;.
007;;153968;;CLIENT;;tf_visa_month;;
007;;154225;;CLIENT;;tf_visa_month;;1
007;;157593;;CLIENT;;tf_visa_year;;
007;;158046;;CLIENT;;tf_visa_year;;0
007;;160229;;CLIENT;;btn_next_2;;Weiter
007;;160307;;CONNECTION_TIME;;19952
007;;160307;;COMPLETE
```

Figure A.4: An excerpt of a log-file created during one task of the user study.

Figure A.5: Screenshots of the server prototype during a user study session (1 of 8).

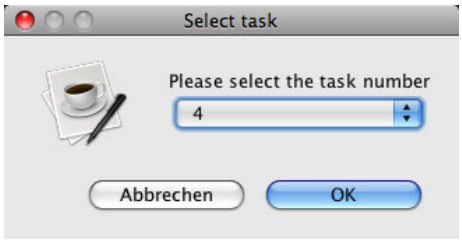

Figure A.6: Screenshots of the server prototype during a user study session (2 of 8).

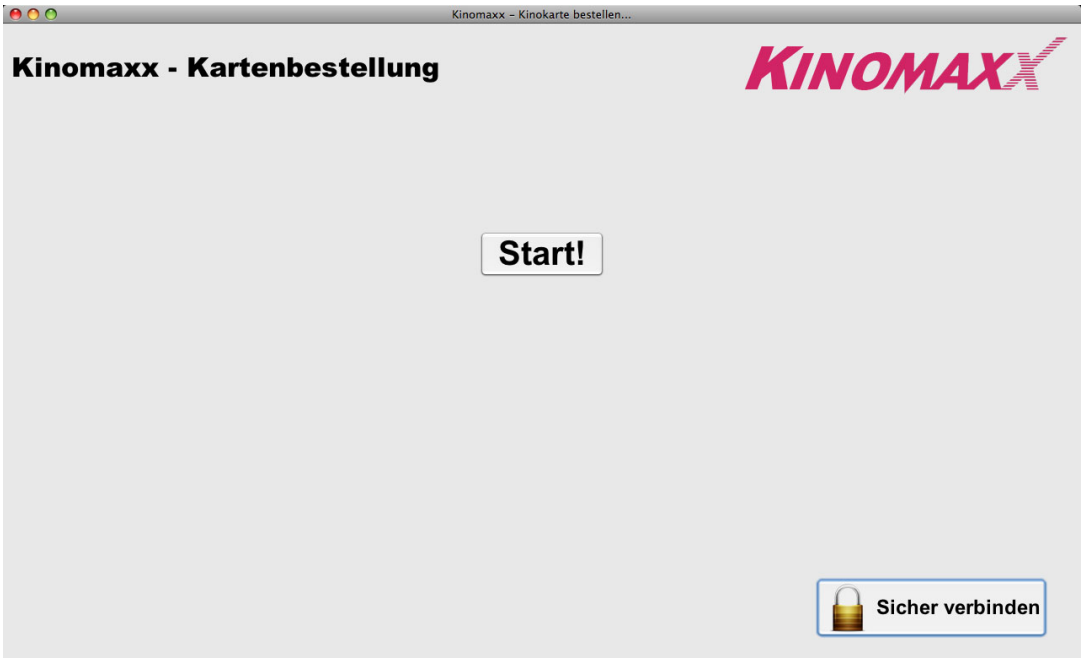

Figure A.7: Screenshots of the server prototype during a user study session (3 of 8).

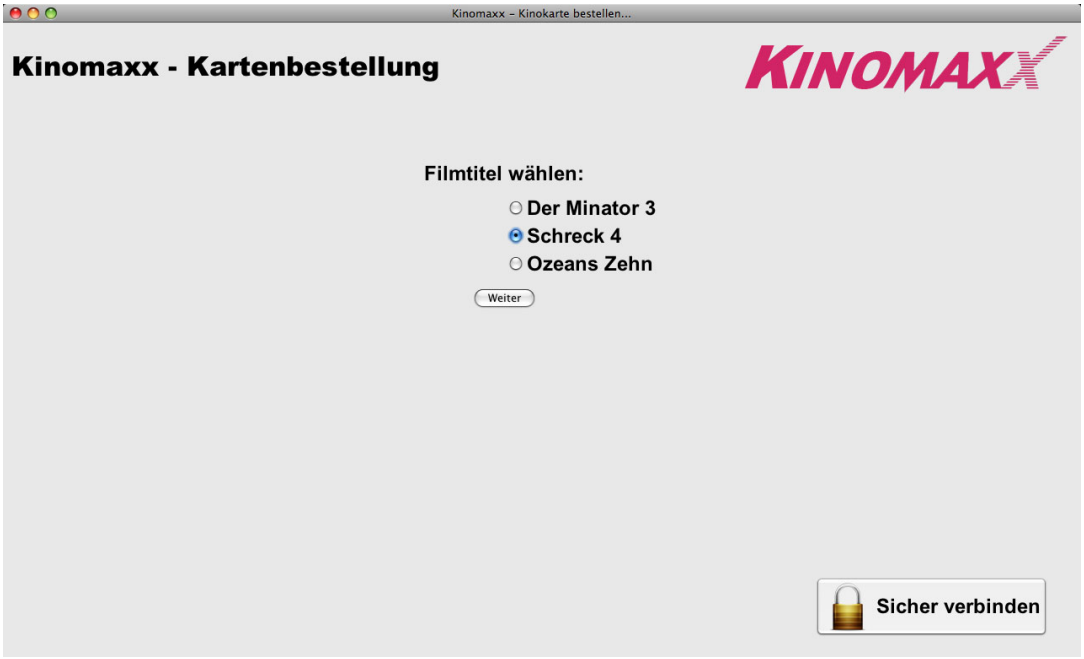

Figure A.8: Screenshots of the server prototype during a user study session (4 of 8).

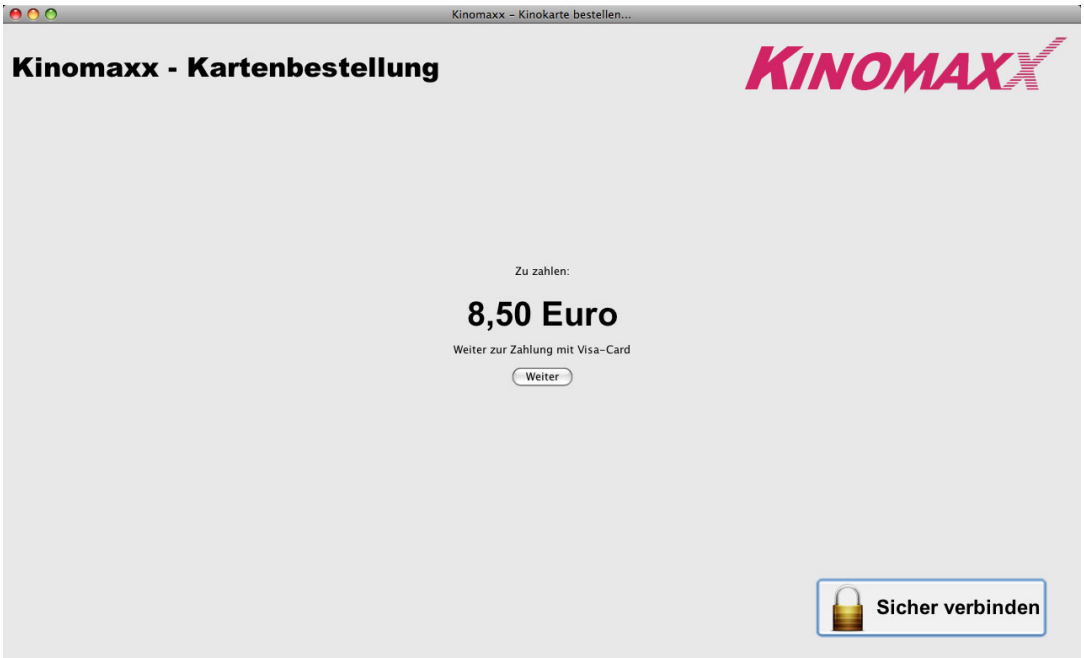

Figure A.9: Screenshots of the server prototype during a user study session (5 of 8).

Figure A.10: Screenshots of the server prototype during a user study session (6 of 8).

Figure A.11: Screenshots of the server prototype during a user study session (7 of 8).

Figure A.12: Screenshots of the server prototype during a user study session (8 of 8).

Benutzerstudie „SecureSwing“

Vielen Dank für Dein Interesse an der Benutzerstudie SecureSwing. Die Studie dauert ca. 20-30 Minuten. Während der Studie gilt es fünf verschiedene Aufgaben durchzuführen. Im Anschluss an diese Aufgaben muss noch ein kurzer Fragebogen beantwortet werden.

Wichtig: Währen der Studie wird lediglich die Usability des Prototypen getestet, nicht Du selbst!

Damit stets gleiche Voraussetzungen für alle Teilnehmer gelten bitten wir dich bei Eingaben die du später durchführst nicht deine eigenen Daten anzugeben.

Für diese Studie bist du:

Maxi Maier
Marktstrasse 1
34542 Essen

Für Zahlungen während der Studie verwende bitte deine Kreditkarte:

Teil a) Bitte mach dich als erstes mit dem mobilen Endgerät sowie mit der Software und dem Verbindungsaufbau vertraut.

Teil b) Führe die 5 Aufgaben aus.

Teil c) Bitte fülle den Fragenbogen zur Benutzerstudie aus.

Vielen Dank für deine Teilnahme!!!!

Figure A.13: The introduction sheet given to every participant.

Benutzerstudie „SecureSwing“

Aufgabe 1: Nur Eingabe am öffentlichen Display

Für deine nächste Zugfahrt möchtest du eine Pahncard 50 bestellen. Bitte benutze die Anwendung am öffentlichen Display um diese Pahncard zu bestellen. Verwende die persönlichen Informationen von der Übersichtsseite.

Für diese Aufgabe benötigst du kein mobiles Endgerät.

Aufgabe 2: Eingabe aller Werte am Handy (ohne Autocompletion)

Für deine nächste Zugfahrt möchtest du eine Pahncard 50 bestellen. Bitte benutze die Anwendung am öffentlichen Display um diese Pahncard zu bestellen. Verwende die persönlichen Informationen von der Übersichtsseite.

Klicke zuerst auf den „Start“-Button. Danach stelle eine sichere Verbindung mit deinem mobilen Endgerät her.

Nachdem du die Verbindung hergestellt hast verwende bitte **ausschließlich** das mobile Endgerät zur Eingabe der einzelnen Werte.

Aufgabe 3: Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion)

Du willst heute Abend ins Kino. Bitte benutze die Anwendung am öffentlichen Display um deine Kinokarte bei KinoMaxx zu bestellen. Du möchtest dir den Film „Schreck 4“ ansehen. Verwende die persönlichen Informationen von der Übersichtsseite.

Klicke zuerst auf den „Start“-Button. Danach stelle eine sichere Verbindung mit deinem mobilen Endgerät her.

Nachdem du die Verbindung hergestellt hast verwende bitte **nur wo nötig** das mobile Endgerät zur Eingabe der einzelnen Werte.

Figure A.14: Tasks 1 to 3 handed to the participants.

Benutzerstudie „SecureSwing“

Aufgabe 4: Eingabe aller Werte am Handy (mit Autocompletion)

Du willst heute Abend ins Kino. Bitte benutze die Anwendung am öffentlichen Display um deine Kinokarte bei KinoMaxx zu bestellen. Du möchtest dir den Film „ Schreck 4 “ ansehen. Verwende die persönlichen Informationen von der Übersichtsseite.

Klicke zuerst auf den „Start“-Button. Danach stelle eine sichere Verbindung mit deinem mobilen Endgerät her.

Nachdem du die Verbindung hergestellt hast verwende bitte **ausschließlich** das mobile Endgerät zur Eingabe der einzelnen Werte. Bei allen Werten, die du mit deinem mobilen Endgerät eingibst verwende die **Autocompletion-Funktion**.

Aufgabe 5: Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion)

Für deine nächste Zugfahrt möchtest du eine Pahncard 50 bestellen. Bitte benutze die Anwendung am öffentlichen Display um diese Pahncard zu bestellen. Verwende die persönlichen Informationen von der Übersichtsseite.

Klicke zuerst auf den „Start“-Button. Danach stelle eine sichere Verbindung mit deinem mobilen Endgerät her.

Nachdem du die Verbindung hergestellt hast verwende bitte **nur wo nötig** das mobile Endgerät zur Eingabe der einzelnen Werte. Bei allen Werten, die du mit deinem mobilen Endgerät eingibst verwende die **Autocompletion-Funktion**.

Figure A.15: Tasks 4 to 5 handed to the participants.

	Task														
Participant	1			2			3			4			5		
	C	O	1to1	C	O	2to1	C	O	3to1	C	O	4to1	C	O	5to1
1	0,00	65,17	1,00	23,99	210,29	3,23	20,97	225,48	3,46	24,17	109,09	1,67	25,47	157,94	2,42
2	0,00	108,35	1,00	32,87	218,04	2,01	38,32	267,67	2,47	26,54	133,79	1,23	29,35	143,46	1,32
3	0,00	46,44	1,00	27,37	159,84	3,44	24,56	162,28	3,49	26,51	99,51	2,14	28,96	261,66	5,63
4	0,00	60,74	1,00	22,71	160,17	2,64	22,30	110,68	1,82	59,59	125,51	2,07	21,78	121,21	2,00
5	0,00	60,32	1,00	35,37	279,66	4,64	26,75	205,14	3,40	50,33	140,24	2,32	29,43	144,75	2,40
6	0,00	54,68	1,00	26,96	153,71	2,81	34,24	166,84	3,05	25,87	94,65	1,73	24,10	110,51	2,02
7	0,00	66,98	1,00	24,30	232,13	3,47	19,95	160,31	2,39	20,43	73,92	1,10	23,56	93,71	1,40
8	0,00	80,75	1,00	50,43	338,09	4,19	32,95	167,84	2,08	30,23	177,59	2,20	29,04	188,48	2,33
9	0,00	95,76	1,00	22,62	185,06	1,93	25,20	165,54	1,73	27,19	118,54	1,24	30,23	162,49	1,70
10	0,00	50,23	1,00	23,41	220,98	4,40	28,52	183,38	3,65	26,43	108,28	2,16	46,39	155,00	3,09
11	0,00	53,92	1,00	25,11	216,60	4,02	25,22	118,90	2,21	27,32	114,33	2,12	41,21	193,51	3,59
12	0,00	65,78	1,00	22,62	195,41	2,97	24,43	153,21	2,33	24,30	90,13	1,37	44,09	167,00	2,54
13															
14	0,00	61,76	1,00	29,63	271,28	4,39	30,00	158,44	2,57	24,19	92,71	1,50	26,05	106,52	1,72
15															
16	0,00	115,96	1,00	26,08	314,73	2,71	33,11	286,77	2,47	31,25	184,30	1,59	26,12	176,07	1,52
17	0,00	77,76	1,00	25,07	183,05	2,35	28,57	189,19	2,43	25,53	113,91	1,46	23,75	116,62	1,50
18	0,00	90,85	1,00	36,25	465,88	5,13	32,89	279,18	3,07	37,33	234,99	2,59	35,86	257,14	2,83
19	0,00	117,18	1,00	26,86	239,26	2,04	22,97	128,13	1,09	25,27	121,77	1,04	23,97	154,16	1,32
20	0,00	85,95	1,00	25,54	373,08	4,34	23,70	171,14	1,99	171,14	230,16	2,68	26,79	133,61	1,55
21	0,00	78,91	1,00	26,16	172,00	2,18	27,24	148,83	1,89	25,28	113,36	1,44	26,74	246,08	3,12
22	0,00	121,05	1,00	24,60	228,36	1,89	31,10	252,82	2,09	26,19	120,14	0,99	22,17	189,98	1,57
23	0,00	71,89	1,00	26,44	165,27	2,30	22,40	114,15	1,59	26,44	165,27	2,30	20,67	135,91	1,89

Figure A.16: The time it took participants to connect and to fulfill the complete task (in seconds).

PDF-Export

Benutzerstudie "SecureSwing"

Demographische Informationen

* **01id: Bitte gib deine Teilnehmernummer an!**

Bitte schreibe Deine Antwort hier:

* **02gender: Bitte gib dein Geschlecht an**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ männlich
- ☐ weiblich

* **03age: Bitte gib Dein Alter an.**

Bitte schreibe Deine Antwort hier:

* **04Beruf: Welchen Beruf übst du zur Zeit aus?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ Student
- ☐ Sonstiges

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Student' war bei der Frage '04Beruf ']

* **04student: Welches Fach studierst du?**

Bitte schreibe Deine Antwort hier:

* **05Education: Welchen bisher höchsten Ausbildungsstatus hast du?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ Grundschulabschluss
- ☐ Hauptschulabschluss
- ☐ Realschulabschluss
- ☐ Abitur
- ☐ Hochschulabschluss
- ☐ Promotion

* **06Education2: Hast du eine abgeschlossene Berufsausblidung?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ Ja

Figure A.17: The survey the user-study participants had to fill out (page 1 of 12).

☐ Nein

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage '06Education2 ']
* **06_2: Welcher Beruf?**
Bitte schreibe Deine Antwort hier:

Eigene Einschätzung

personals: Wie schätzt du deine technischen Kenntnisse ein?
Bitte wähle die zutreffende Antwort aus:

	1 - sehr schlecht	2	3	4	5 - sehr gut
Technische Fähigkeiten allgemein	☐	☐	☐	☐	☐
Computerkenntnisse	☐	☐	☐	☐	☐
Umgang mit mobilen Endgeräten (z.B. Handys)	☐	☐	☐	☐	☐

* **worries: Bitte überlege zu jedem der Folgenden Themen, ob dir schon Gedanken über deine Sicherheit gemacht hast. (1 = keine Sorgen gemacht, 5 = denke ich häufig drüber nach)**
Bitte wähle die zutreffende Antwort aus:

Sicherheit an Geldautomaten	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Diebstahl von Eigentum (Geldbörse, Mobiltelefon)	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Mißbrauch persönlicher Daten (Spam, Werbung, Stalking)	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Diebstahl und Mißbrauch von Onlinebanking-Daten	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5

Nutzung von Mobilen Endgeräten

* **mobdev: Besitzt du ein mobiles Endgerät (Handy, PDA, usw.)?**
Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja
☐ Nein

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'mobdev ']

Figure A.18: The survey the user-study participants had to fill out (page 2 of 12).

* **takephone: Wie häufig hast du dein mobiles Endgerät dabei?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ nie
- ☐ 1 mal im Monat
- ☐ 1 mal pro Woche
- ☐ mehrmals pro Woche
- ☐ (fast) ständig

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'mobdev ']
details: Welche Funktionen unterstützt dein mobiles Endgerät?

Bitte wähle alle Punkte aus, die zutreffen:

- ☐ Bluetooth
- ☐ WLAN
- ☐ GPS
- ☐ NFC
- ☐ Kamera
- ☐ Radio
- ☐ UMTS/3G
- ☐ Touchscreen
- ☐ QWERTZ-Tastatur
- ☐ Windows Mobile

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Bluetooth' war bei der Frage 'details ']
* **bluet: Wie häufig benutzt du Bluetooth?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ nie
- ☐ 1 mal im Monat
- ☐ 1 mal pro Woche
- ☐ (fast) ständig

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Kamera' war bei der Frage 'details ']
* **camerausage: Wie häufig benutzt du deine Kamera?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ nie
- ☐ 1 mal im Monat
- ☐ 1 mal pro Woche

Figure A.19: The survey the user-study participants had to fill out (page 3 of 12).

☐ (fast) ständig

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'mobdev ']
* **connections: Hast du dein mobiles Endgerät schon mal mit anderen Geräten benutzt?**
Bitte wähle die zutreffende Antwort aus:

	ja	nein
Datenaustausch	☐	☐
Entfernte Steuerung oder Eingabe	☐	☐

Nutzung von Automaten

* **usage: Wie oft benutzt du folgende Automaten....**
Bitte wähle die zutreffende Antwort aus:

	fast nie	1 mal im Monat	1 mal pro Woche	mehrmals pro Woche	täglich
Geldautomaten (ATM)	☐	☐	☐	☐	☐
Fahrtkartenautomaten	☐	☐	☐	☐	☐
andere Ticketautomaten (Kino, Konzert)	☐	☐	☐	☐	☐
Verkaufsautomaten (Getränke, Snacks)	☐	☐	☐	☐	☐

* **counter: Wenn du die Wahl zwischen Automaten oder einer Bedienung am Schalter hast was ziehst du vor?**
Bitte wähle nur eine der folgenden Antworten aus:

☐ Automat
☐ Bedienung am Schalter
☐ hängt davon ab
☐ keine Präferenz

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Automat' war bei der Frage 'counter ']
machinewhy: Warum bevorzugst du Automaten?
Bitte wähle alle Punkte aus, die zutreffen:

☐ ich erhalte schneller was ich will
☐ ich weiß schon genau was ich machen muss
☐ ich möchte keinen persönlichen Kontakt mit Schalterpersonal
☐ meistens sind dort die Wartezeiten kürzer
☐ ich habe meist keine andere Möglichkeit als einen Automaten zu benutzen

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Bedienung am Schalter' war bei der Frage 'counter ']
perferpeople: Warum bevorzugst du die Bedienung am Schalter?

Figure A.20: The survey the user-study participants had to fill out (page 4 of 12).

Bitte wähle alle Punkte aus, die zutreffen:

- ☐ ich erhalte schneller was ich will
- ☐ die person am Schalter weiß schneller was ich will
- ☐ ich finde Automaten unpersönlich
- ☐ meistens sind dort die Wartezeiten kürzer
- ☐ ich habe meist keine andere Möglichkeit als an den Schalter zu gehen

---

* **security: Hast du dir schon einmal über die Sicherheit solcher Automaten Gedanken gemacht?**

Bitte wähle nur eine der folgenden Antworten aus:

- ☐ Ja
- ☐ Nein

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'security ']

security2: Was für Sicherheitsrisiken kannst du dir vorstellen?

Bitte schreibe Deine Antwort hier:

---

* **passwords1: Wie oft pro Woche verwendest du Passwörter oder PINs (privat oder in der Öffentlichkeit)?**

Bitte schreibe Deine Antwort hier:

---

* **passwords2: Wie oft pro Woche gibst du Passwörter in der Öffentlichkeit ein?**

Bitte schreibe Deine Antwort hier:

---

* **softfactors: Bitte betrachte die folgenden Werte bei der Interaktion mit öffentlichen Displays und bewerte wie wichtig dir die Eigenschaften jeweils sind. (1 = unwichtig, 5 = sehr wichtig)**

Bitte wähle die zutreffende Antwort aus:

Interaktionsgeschwindigkeit	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Sicherheit	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Einfachheit	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
Design	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5

Figure A.21: The survey the user-study participants had to fill out (page 5 of 12).

Connection methods

* **bluetooth: Hast du zuvor schon jemals zwei Bluetooth-Geräte miteinander gekoppelt?**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

* **qrcode: Hast du die zweidimensionalen Barcodes (QR-Codes) schon vorher gesehen?**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'qrcode ']

qrcode2: Wo hast du solche Barcodes schonmal gesehen?

Bitte schreibe Deine Antwort hier:

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Ja' war bei der Frage 'qrcode ']

* **qrcode3: Hast du die Codes auch schon mal benutzt?**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

* **04qrcode: Während der Studie hast du die Verbindung zwischen öffentlichem Display und mobilem Endgerät mit Hilfe eines solchen QR-Codes hergestellt? Bitte beurteile diese Verbindungsmethode.**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam	☐	☐	☐	☐	☐	schnell
einfach	☐	☐	☐	☐	☐	schwer

* **othermethods: Es wäre möglich andere Methoden zum herstellen einer Verbindung zwischen dem Terminal und dem Endgerät zu verwenden. Bitte gib jeweils an wie geeignet du eine solche Methode findest. (1 = überhaupt nicht geeignet, 5 = sehr geeignet)**

Bitte wähle die zutreffende Antwort aus:

QR-Codes: Zweidimensionale Codes werden abfotografiert. ☐ 1 ☐ 2 ☐ 3 ☐ 4

Figure A.22: The survey the user-study participants had to fill out (page 6 of 12).

Geräteliste: Das Terminal zeigt eine Liste der Geräte in der Nähe an, man wählt sein eignes Gerät dem Namen nach aus.	☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Tokenliste: Das Terminal zeigt eine Liste mit persönlichen Identifikationssysbolen (Bildern) der Benutzer wählt sein Symbol.	☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
SyncTap: Am Handy und am Terminal wird gleichzeitig und für die selbe Dauer eine bestimmte Taste gedrückt.	☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
NFC: Das mobile Endgerät wird sehr Nahe an einen bestimmten Ort des Terminals gehalten und verbindet sich danach automatisch.	☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Security and Asterisk

*** sec: Während den verschiedenen Aufgaben haben sich bestimmte Felder aus Sicherheitsgründen dynamisch an die Verbindung angepasst und eventuell keine Informationen mehr angezeigt oder keine Eingabe mehr zugelassen. Ist dir dieses Verhalten aufgefallen?**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

sec2: Gewisse sicherheitsrelevante Felder sperren ihre Eingabemöglichkeit auf dem öffentlichen Display sobald der Benutzer sein mobiles Endgerät mit dem öffentlichen Display verbunden hat. Wie findest du das?

Bitte wähle nur eine der folgenden Antworten aus:

☐ das finde ich sinnvoll, somit weiß man gleich welche Felder sicherheitskritisch sind

☐ ich möchte gerne immer selbst entscheiden wo ich die Werte eingebe

☐ prinzipiell finde ich eine Sperre der Felder gut, aber ich die im test gesperrten felder erschienen mir nicht sinnvoll

asterisk: Im Test, wurde der Betrag der zu bezahlen ist bei einer Handyverbindung nur am mobilen Endgerät dargestellt. In der Praxis könnte das Verfahren zum Beispiel noch benutzt werden, um an Geldautomaten den Kontostand nicht auf dem Monitor anzuzeigen. Was hältst du davon?

Figure A.23: The survey the user-study participants had to fill out (page 7 of 12).

Bitte wähle nur eine der folgenden Antworten aus:

☐ das finde ich sinnvoll, damit ist das erspähen von Informationen schwerer

☐ ich sehe lieber alle Werte auf einem Bildschirm, so muss ich immer hin und her schauen

☐ prinzipiell finde ich das Verstecken von Informationen gut, aber im Test wurde es nicht sinnvoll eingesetzt

Auswertung des Tests

*** mobileuse: Du hattest im Test die Möglichkeit Daten am Automaten mit Hilfe deines mobilen Endgeräts einzugeben. Fandest du dieses System hilfreich. (Bitte gib eine Begründung für deine Entscheidung an)**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

Bitte schreibe eine Kommentar zu Deiner Auswahl

*** autocompletionuse: In zwei Szenarien konntest du auf bereits im mobilen Endgerät hinterlegte Werte zurückgreifen (Autocompletion). Fandest du dieses Feature hilfreich oder nicht? (Bitte gib eine Begründung an.)**

Bitte wähle nur eine der folgenden Antworten aus:

☐ Ja

☐ Nein

Bitte schreibe eine Kommentar zu Deiner Auswahl

*** security: Bitte bewerten Sie die Sicherheit der getesteten Verfahren (1 = nicht sicher, 5 = sehr sicher)**

Bitte wähle die zutreffende Antwort aus:

Nur Eingabe am öffentlichen Display	☐1 ☐2 ☐3 ☐4 ☐5

Figure A.24: The survey the user-study participants had to fill out (page 8 of 12).

Eingabe aller Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe aller Werte am Handy (mit Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5

* **speed: Bitte bewerten Sie die Geschwindigkeit der getesteten Verfahren (1 = sehr langsam, 5 = sehr schnell)**

Bitte wähle die zutreffende Antwort aus:

Nur Eingabe am öffentlichen Display	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe aller Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe aller Werte am Handy (mit Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5

* **simplicity: Bitte bewerten Sie die Einfachheit der getesteten Verfahren (1 = sehr kompliziert, 5 = sehr einfach)**

Bitte wähle die zutreffende Antwort aus:

Nur Eingabe am öffentlichen Display	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe aller Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5
Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion)	☐1 ☐2 ☐3 ☐4 ☐5

Figure A.25: The survey the user-study participants had to fill out (page 9 of 12).

Eingabe aller Werte am Handy (mit Autocompletion)	1 2 3 4 5
Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion)	1 2 3 4 5

* **usage: Wenn Du die Möglichkeit hättest ab heute dein Handy an vielen öffentlichen Displays zu verwenden welche Technik würdest du einsetzen?**

Bitte wähle nur eine der folgenden Antworten aus:

- Nur Eingabe am öffentlichen Display
- Eingabe aller Werte am Handy (ohne Autocompletion)
- Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion)
- Eingabe aller Werte am Handy (mit Autocompletion)
- Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion)

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Nur Eingabe am öffentlichen Display ' war bei der Frage 'usage ']

* **best1: Du hast ausgewählt, dass die klassische Eingabemethode dir am liebsten ist, bitte vergleiche die klassische Eingabemethode mit den anderen Methoden. Die klassische Eingabemethode ist....**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam						schnell
ungewohnt						gewohnt
unsicher						sicher
schlecht bedienbar						bedienungsfreundlich
unpraktisch						praktisch

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Eingabe aller Werte am Handy (ohne Autocompletion) ' war bei der Frage 'usage ']

* **best2: Du hast ausgewählt, dass die Eingabe von allen Werten am Handy ohne Autocopmletion die am liebsten ist. Bitte vergleiche diese Methode mit der klassischen Eingabe ohne Handy. Die neuartige Methode ist....**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam						schnell
ungewohnt						gewohnt
unsicher						sicher
schlecht bedienbar						bedienungsfreundlich
unpraktisch						praktisch

Figure A.26: The survey the user-study participants had to fill out (page 10 of 12).

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Eingabe sicherheitsrelvanter Werte am Handy (ohne Autocompletion) ' war bei der Frage 'usage ']

* **best3: Du hast ausgewählt, dass die Eingabe von sicherheitsrelevanten Werte am Handy ohne Autocompletion dir am liebsten ist. Bitte vergleiche diese Methode mit der klassischen Eingabe ohne Handy. Die neuartige Methode ist....**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam	☐	☐	☐	☐	☐	schnell
ungewohnt	☐	☐	☐	☐	☐	gewohnt
unsicher	☐	☐	☐	☐	☐	sicher
schlecht bedienbar	☐	☐	☐	☐	☐	bedienungsfreundlich
unpraktisch	☐	☐	☐	☐	☐	praktisch

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Eingabe aller Werte am Handy (mit Autocompletion) ' war bei der Frage 'usage ']

* **best4: Du hast ausgewählt, dass die Eingabe von allen Werten am Handy mit Autocompletion dir am liebsten ist. Bitte vergleiche diese Methode mit der klassischen Eingabe ohne Handy. Die neuartige Methode ist....**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam	☐	☐	☐	☐	☐	schnell
ungewohnt	☐	☐	☐	☐	☐	gewohnt
unsicher	☐	☐	☐	☐	☐	sicher
schlecht bedienbar	☐	☐	☐	☐	☐	bedienungsfreundlich
unpraktisch	☐	☐	☐	☐	☐	praktisch

[Bitte beantworte diese Frage nur, falls Deine Antwort 'Eingabe sicherheitsrelvanter Werte am Handy (mit Autocompletion) ' war bei der Frage 'usage ']

* **best5: Du hast ausgewählt, dass die Eingabe von sicherheitsrelevanten Werten am Handy mit Autocompletion dir am liebsten ist. Bitte vergleiche diese Methode mit der klassischen Eingabe ohne Handy. Die neuartige Methode ist....**

Bitte wähle die zutreffende Antwort aus:

	-2	-1	0	+1	+2	
langsam	☐	☐	☐	☐	☐	schnell
ungewohnt	☐	☐	☐	☐	☐	gewohnt
unsicher	☐	☐	☐	☐	☐	sicher
schlecht bedienbar	☐	☐	☐	☐	☐	bedienungsfreundlich
unpraktisch	☐	☐	☐	☐	☐	praktisch

comments: Hast du abschließend noch Anmerkungen zum Test, zu den Prototypen oder allgemeine Hinweise?

Figure A.27: The survey the user-study participants had to fill out (page 11 of 12).

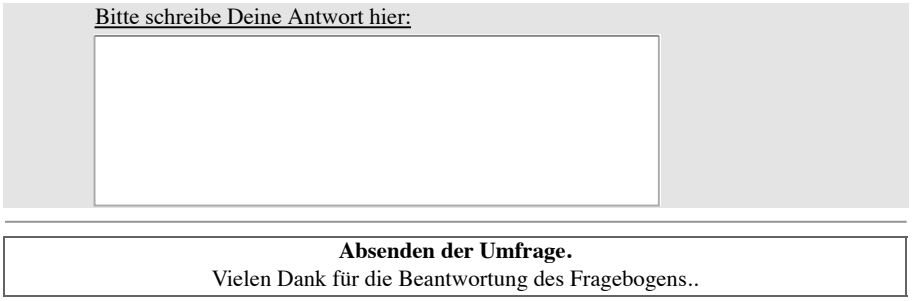

Figure A.28: The survey the user-study participants had to fill out (page 12 of 12).

Participant comments during the user-study

1. Zu viele Schritte
2. Blick auf den Bildschirm Eingabe am Handy
3. Radio Button auch gleich per Handy eingegeben
4. Ende des Formulars nicht ersichtlich
5. Normalerweise mehr Werte für auto-completion
6. Keine Summary am Ende des Verkaufsvorgangs
7. Feldeingaben einschränken auf Datentyp
8. OK-Button
9. Hintergrundgrafik auf dem Mobiltelefon irritiert
10. Audio Feedback bei Eingaben am Nokia Handy positiv
11. Gesicherte Felder nicht nur durch grau sondern auch durch ein Symbol kennzeichnen
12. Caret wird nicht mitsynchronisiert
13. Auto-completion bei der Eingabe
14. Ablagefläche am Automaten wäre sinnvoll
15. Berechtigung für Foto und Netzwerk störend
16. Backspace und Weiter auf dem Mobiltelefon verwechselt und roter Hörer (Beenden) liegen sehr dich beieinander
17. Dadurch dass kein Zwang zum Verbindungsaufbau besteht, will man sofort mit der Eingabe loslegen
18. Auto-ccompletion-Felder sind nicht immer zusammen mit dem Eingabefeld sichtbar
19. Auto-completion geht „Zack zack“
20. Betrag wird angezeigt
21. Probanden wurde die #-Taste für den Zahleneingabemodus erklärt
22. Häufig wurde für die Namenseingabe der Kreditkarte auch das Hand verwendet damit kein Diplaywechselt gemacht werden musste
23. Foto machen „gar nicht so einfach“
24. Kreditkartendaten häufig komplett ins Monatsfeld geschrieben
25. „Komisch wenn man am Terminal arbeitet und den Betrag woanders suchen muss“
26. UTF-8 Fehler keine Sonderzeichen und Umlaute
27. „Ein Display zuviel!“
28. „Kreditkartenummer wird nicht angezeigt! Das ist gut!“

Figure A.29: Individual comments given by participants during the user-study.

A APPENDIX

Contents of Enclosed CD

- Client source code
- Client documentation
- Framework source code
- Framework documentation
- User study application
- User study evaluation
- User study diagrams
- User study log-files
- Related work
- Diploma Thesis (PDF)
- Diploma Thesis (LaTeX-file including figures)

References

[1] S. Berger, R. Kjeldsen, C. Narayanaswami, C. Pinhanez, M. Podlaseck and M. Raghunath, Using Symbiotic Displays to View Sensitive Information in Public. In: PerCom: Third IEEE International Conference on Pervasive Computing and Communications, Koloa, Kauai, Hawaii, IEEE Computer Society, Washington, DC, USA, 139–148, 2005.

[2] BlueCove Team, BlueCove documentation. WWW page, accessed 12-June-2009. URL `http://www.bluecove.org`

[3] M. Claßen, Xparse-J XML Parser for Java. WWW page, accessed 12-June-2009. URL `http://www.webreference.com/xml/tools/`

[4] W. Claycomb and D. Shin, Secure Real World Interaction Using Mobile Devices. In: Proceedings of the Pervasive Mobile Interaction Devices Workshop, Dublin, Ireland, 2006.

[5] L. Coventry, A. D. Angeli and G. Johnson, Usability and biometric verification at the ATM interface. In: CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems, Ft. Lauderdale, Florida, USA, ACM, 153–160, 2003.

[6] A. De Luca and B. Frauendienst, A Privacy-Respectful Input Method For Public Terminals. In: Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges, Lund Sweden, ACM, New York, NY, USA, 455–458, 2008.

[7] A. De Luca, R. Weiss and H. Drewes, Evaluation of eye-gaze interaction methods for security enhanced PIN-entry. In: OZCHI '07: Proceedings of the 19th Australasian conference on Computer-Human Interaction, Adelaide, SA, Australia, ACM, New York, NY, USA, 199–202, 2007.

[8] A. De Luca, E. von Zezschwitz and H. Hußmann, VibraPass: secure authentication based on shared lies. In: CHI '09: Proceedings of the 27th international conference on Human factors in computing systems, Boston, MA, USA, ACM, New York, NY, USA, 913–916, 2009.

[9] Denso-Wave Inc., QR Code.com. WWW page, accessed 12-June-2009. URL `http://www.denso-wave.com/qrcode/index-e.html`

[10] T. Deyle and V. Roth, Accessible Authentication via Tactile PIN Entry. In: CG Topics, 3, 2006.

[11] H. Drewes, A. De Luca and A. Schmidt, Eye-gaze interaction for mobile phones. In: Mobility '07: Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology, Singapore, ACM, New York, NY, USA, 364–371, 2007.

[12] H. Drewes and A. Schmidt, Interacting with the Computer Using Gaze Gestures. In: Human-Computer Interaction, volume 4663 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 475–488, 2008.

[13] FierceWireless, Mobile Connections Reach 4 Billion Worldwide. WWW page, accessed 12-June-2009.
URL `http://www.fiercewireless.com/press-releases/mobile-connections-reach-4-billion-worldwide?utm_medium=nl&utm_source=internal&cmp-id=EMC-NL-FW&dest=FW`

[14] Financial Services Technology, Please enter your four-digit pin. WWW page, accessed 12-June-2009.
URL http://www.fsteurope.com/article/Issue-3/AML-AND-IT-Security/Please-enter-your-four-digit-pin/

[15] Google Inc., ZXing ("Zebra Crossing"). WWW page, accessed 12-June-2009.
URL http://code.google.com/p/zxing

[16] L. E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl and H.-W. Gellersen, Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts. In: Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia, USA, Springer, Berlin, Germany, 116–122, 2001.

[17] J. Hunter, JDOM. WWW page, accessed 12-June-2009.
URL http://www.jdom.org

[18] H. M. Hutchings and J. S. Pierce, Understanding the whethers, hows, and whys of divisible interfaces. In: AVI '06: Proceedings of the working conference on advanced visual interfaces, Venezia, Italy, ACM, New York, NY, USA, 274–277, 2006.

[19] M. Kumar, T. Garfinkel, D. Boneh and T. Winograd, Reducing shoulder-surfing by using gaze-based password entry. In: SOUPS '07: Proceedings of the 3rd symposium on Usable privacy and security, Pittsburgh, PA, USA, ACM, New York, NY, USA, 13–19, 2007.

[20] B. Malek, M. Orozco and A. El Saddik, Novel shoulder-surfing resistant haptic-based graphical password. In: Proceedings of the 6th EuropHaptics conference, Paris, France, 2006.

[21] Massachusetts Institute of Technology, Luther Simjian. WWW page, accessed 6-May-2009.
URL http://web.mit.edu/invent/iow/simjian.html

[22] B. Miligan, The man who invented the cash machine. WWW page, accessed 6-May-2009.
URL http://news.bbc.co.uk/2/hi/business/6230194.stm

[23] Mozilla Corporation, XML User Interface Language (XUL). WWW page, accessed 14-May-2009.
URL http://www.mozilla.org/projects/xul/

[24] Mozilla Corporation, XUL Tutorial - Adding Buttons. WWW page, accessed 12-June-2009.
URL https://developer.mozilla.org/en/XUL_Tutorial/Adding_Buttons

[25] B. A. Myers, Using handhelds and PCs together. In: Communications of the ACM, 44(11):34–41, 2001.

[26] Nokia, Nokia - Devices. WWW page, accessed 12-June-2009.
URL http://www.nokia.com/A4630648?category=n80#

[27] Nokia Inc., Technical specifications - Nokia N80 Internet Edition. WWW page, accessed 12-June-2009.
URL http://www.nokiausa.com/find-products/phones/nokia-n80-internet-edition/technical-specifications

[28] W. Paulus, SWIXML - Generate javax.swing at runtime based on XML descriptors. WWW page, accessed 12-June-2009.
URL http://www.swixml.org

[29] W. Paulus, swixml tags. WWW page, accessed 12-June-2009.
URL http://www.swixml.org/tagdocs/index.html

[30] J. Pierce and H. Mahaney, Opportunistic Annexing for Handheld Devices: Opportunities and Challenges. In: Proceedings of HCIC, Georgia Institute of Technology, Atlanta, Georgia, USA, 2004.

[31] J. Rekimoto, Y. Ayatsuka and M. Kohno, SyncTap: An Interaction Technique for Mobile Networking. In: Human-Computer Interaction with Mobile Devices and Services, Springer, Berlin, Germany, 104–115, 2003.

[32] V. Roth, K. Richter and R. Freidinger, A PIN-entry method resilient against shoulder surfing. In: CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, Washington, DC, USA, ACM, New York, NY, USA, 236–245, 2004.

[33] H. Sasamoto, N. Christin and E. Hayashi, Undercover: Authentication usable in front of prying eyes. In: CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on human factors in computing systems, Florence, Italy, ACM, New York, NY, USA, 183–192, 2008.

[34] R. Sharp, J. Scott and A. R. Beresford, Secure Mobile Computing Via Public Terminals. In: PERVASIVE 2006: Proceedings of the 4th international conference on pervasive computing, Dublin, Ireland, Springer, Berlin, Germany, 238–253, 2006.
URL http://www.springerlink.com/content/q8xp21281q777k14/

[35] B. Shneiderman and C. Plaisant, Designing the user interface. Pearson Education, 2005.

[36] Statistisches Bundesamt Deutschland, Einkommens- und Verbrauchsstichprobe (EVS). WWW page, accessed 6-May-2009.
URL http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Content/Statistiken/WirtschaftsrechnungenZeitbudgets/EinkommensVerbrauchsstichproben/Tabellen/Content75/AusstattungprivaterHaushalteInformationstechnik,templateId=renderPrint.psml

[37] Sun Microsystems, Inc., Java ME Technology. WWW page, accessed 12-June-2009.
URL http://java.sun.com/javame/technology/index.jsp

[38] Sun Microsystems, Inc., Overview (MID Profile). WWW page, accessed 12-June-2009.
URL http://java.sun.com/javame/reference/apis/jsr118/

[39] Sun Microsystems, Inc., RecordStore (MID Profile). WWW page, accessed 12-June-2009.
URL http://java.sun.com/javame/reference/apis/jsr118/javax/microedition/rms/RecordStore.html

[40] X. Suo, Y. Zhu and G. S. Owen, Graphical Passwords: A Survey. In: ACSAC: Proceedings of the 21st Annual Computer Security Applications Conference, Los Alamitos, CA, USA, IEEE Computer Society, Washington, DC, USA, 463–472, 2005.

[41] D. S. Tan, P. Keyani and M. Czerwinski, Spy-resistant keyboard: More secure password entry on public touch screen displays. In: OZCHI '05: Proceedings of the 17th Australia conference on Computer-Human Interaction, Canberra, Australia, Computer-Human Interaction Special Interest Group (CHISIG) of Australia, Narrabundah, Australia, 1–10, 2005.

[42] The Legion of the Bouncy Castle, bouncycastle.org. WWW page, accessed 12-June-2009.
URL http://www.bouncycastle.org

[43] twit88.com, .NET QRCode Library. WWW page, accessed 12-June-2009.
URL http://www.twit88.com/home/opensource/qrcode

[44] S. Wiedenbeck, J. Waters, L. Sobrado and J.-C. Birget, Design and evaluation of a shoulder-surfing resistant graphical password scheme. In: AVI '06: Proceedings of the working conference on advanced visual interfaces, Venezia, Italy, ACM, New York, NY, USA, 177–184, 2006.

[45] Wikipedia, The Free Encyclopedia, Java APIs for Bluetooth. Accessed 11-May-2009.
URL http://en.wikipedia.org/w/index.php?title=Java_APIs_for_Bluetooth&oldid=284221501

[46] Wikipedia, The Free Encyclopedia, Model-View-Controller. WWW Page, [accessed 26-January-2009].
URL http://en.wikipedia.org/w/index.php?title=Model-View-Controller&oldid=266636108

[47] Wikipedia, The Free Encyclopedia, QR Code. Accessed 10-May-2009.
URL http://en.wikipedia.org/w/index.php?title=QR_Code&oldid=287838386