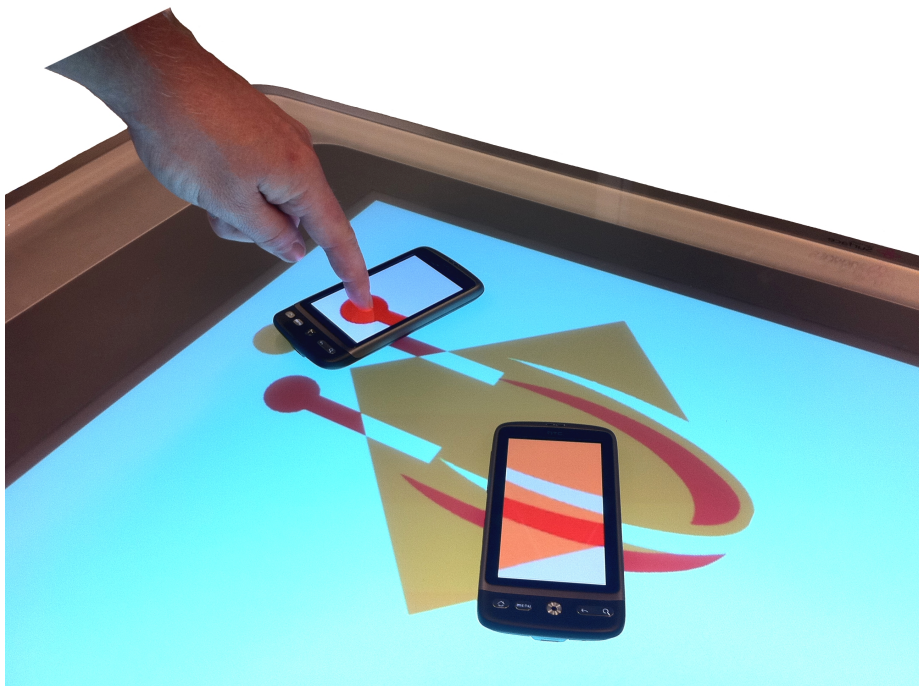

Non-anonymous user interaction on tabletop displays



Master thesis

by

Thomas Berglund
(090773-berglund)

Michael Thomassen
(040685-mtho)

Supervisor:

Professor Jakob E. Bardram

1 August 2011

IT University of Copenhagen, Denmark

IT University of Copenhagen
Rued Langgaards Vej 7, DK-2300 København S, Denmark
Phone +45 72185000, Fax +45 72185001
itu@itu.dk
www.itu.dk

Abstract

Contemporary tabletop computers lack the ability to natively link an action to the identity of its originator in a way that allows applications to discriminate between simultaneous users.

This thesis proposes a solution that allows developers to build tabletop applications that depend on non-anonymous user interaction.

In comparison with prior research projects, this work integrates with the tabletop without requiring special gestures, is secure, and uses standard hardware.

The results are that it is possible to design and implement a framework for non-anonymous user interaction that is secure and uses standard hardware. The framework features a user experience design, a set of *identified events*, and a set of UI controls. A security analysis establishes that the framework also must feature a customisable authentication mechanism.

The evaluation shows that the framework can be used for building non-trivial applications, and the main results of a usability test are that users quickly learn and accept the features of the framework, and users' comments have led to five design guidelines that can help leverage the quality of an application when using the framework.

The thesis concludes that the *NAI* framework solves the problem of having non-anonymous user interaction on tabletop displays, and thereby allows personalised user experience in future tabletop applications.

Acknowledgements

First and foremost, we would like to thank Professor Jakob Bardram, IT University of Copenhagen, Denmark (ITU), for guiding us through the art of writing a master thesis.

Next we thank Mads Frost, Ph.D student, ITU, for good advice on how to design and conduct a usability evaluation, and all the people volunteering for participating in the evaluation.

Finally, we thank the people associated with pITlab (The Pervasive Interaction Technology Lab) at ITU. In particular, for the many supporting and helpful comments and Sebastian Büttrich for letting us borrow the hardware.

Contents

| | |
|--|------------|
| Abstract | i |
| Acknowledgements | iii |
| 1 Introduction | 1 |
| 1.1 Context and motivation | 1 |
| 1.2 Problem statement | 3 |
| 1.3 Research methods | 3 |
| 1.4 Results | 4 |
| 1.5 Thesis overview | 5 |
| 2 Related work | 7 |
| 3 The <i>NAI</i> framework | 13 |
| 3.1 General user experience design | 13 |
| 3.2 Usage | 14 |
| 3.3 Setup | 25 |
| 4 Security analysis | 33 |
| 4.1 Area 1: Non-anonymous user interaction | 33 |
| 4.2 Area 2: Tabletop – smartphone communication | 35 |
| 4.3 Area 3: Tag recognition | 35 |
| 4.4 Area 4: Pairing tag with smartphone | 36 |
| 4.5 Summary | 37 |
| 5 Implementation | 39 |
| 5.1 States | 39 |
| 5.2 Communication | 40 |
| 5.3 Tabletop | 42 |
| 5.4 Smartphone | 51 |
| 6 Using the <i>NAI</i> framework for application building | 57 |
| 6.1 Restaurant of the future | 57 |
| 7 Evaluation | 63 |
| 7.1 Parameters | 63 |
| 7.2 Method | 64 |
| 7.3 Test sessions setup | 66 |
| 7.4 Participants | 66 |
| 7.5 Results | 67 |

| | |
|--|-----------|
| 8 Discussion | 71 |
| 8.1 User experience | 71 |
| 8.2 Framework implementation | 73 |
| 8.3 Security | 74 |
| 9 Conclusion | 75 |
| 9.1 Future work | 76 |
| References | 79 |
| A Framework source code | 81 |
| B Evaluation material | 83 |
| B.1 Test scenarios | 83 |
| B.2 Semi-structured interview - prepared questions | 84 |
| B.3 ITU newsletter advertisement | 85 |
| C Responsibilities | 87 |
| C.1 Thesis | 87 |
| C.2 Code | 87 |

Introduction

1.1 Context and motivation

Today, people use many types of computers, not just a desktop or laptop computer. They differ in size, capabilities, and the way you interact with them. For example, a smartphone is a small hand-held computer powered device, where the display typically doubles as input as well as an output device. This also applies to the increasingly popular tablet computers, that have a larger screen than a smartphone, but are not used for telephony. At the same time the laptop still relies on a keyboard, a mouse pointer, and a conventional display.

In the future, the multitude of devices of an ordinary user will possibly also cover tabletop computers (hereafter just *tabletop*). That is, computer connected tables equipped with a display and an input system that allow direct manipulation upon the display using finger or hand gestures.

While the smartphone, tablet, and the laptop in essence are *personal* devices, tabletops are being designed for collaborative multi-user settings. In this sense, the display can be seen as a *shared*, or *public*, space and there might not be a natural orientation of the screen as the users gather



Figure 1.1: A Microsoft Surface supporting guests at a Sheraton Hotel.

around the table (see figure 1.1). It is features like these that make this type of device distinctly different from the personal devices. The case studies for the *Microsoft Surface*¹ (hereafter MSS) reflect public settings like tourist guide planning at hotels (as depicted in figure 1.1) and social activities at bars like Hard Rock Café and iBar in the Rio All-Suite Hotel & Casino.

In general, contemporary tabletops do not have the ability to discriminate the touch of one user from the touch of another. This thesis is about adding support for non-anonymous user interaction to tabletop displays. It will allow developers to build applications for a tabletop, like the MSS, that have access to the identity of the user carrying out an action.

Presently, tabletop computers are not a commodity. Not until Jeff Han in 2005 [6] showed how a tabletop can be build with relative cheap means, were such tabletops widely deployed in research departments [13]. The MSS, first released in 2007, is one of the first serious attempts at creating a complete commercial product targeted end users. Because the price tag is very high, and the set of available applications for it are limited such a device is not something people stumble upon everyday. Besides the hardware, Microsoft has developed a complete Software Development Kit (SDK) including a Surface API that extends Windows Presentation Foundation (WPF), so that developers have easy access to the features.

The work presented in this thesis is targeted for use on a MSS, and it is therefore natural that the solution is implemented as a framework that builds on top of the Surface API. Because a framework must be extendable and flexible enough to be applied in a large variety of settings, it is designed to be secure. Since the identity of the user is essential to the framework, it includes a customisable user authentication mechanism that supports varying user authentication requirements.

Research in discrimination of the users actions on a tabletop is basically motivated by two sets of problems. The first set of problems is related to *access control*. For example, a basic malice and fraud scenario where person A attempts at deleting or manipulating objects that person B owns, or while person B is away, person A tries to send messages in the name of person B to a third party [15]. Ringel et al. [14] envisions a meeting scenario, where access to data, like digital documents, is controlled by specific gestures in combination with the knowledge of who is manipulating the object. Schöning et al. [21] imagines a scenario where a single large interactive surface acts as a shared control unit for dispatching emergency units after a flooding. Although using a shared display, critical commands can only be executed by the person with the right authorisation.

The second set of problems is related to *dynamically customising the user interface*. IdLenses [20] uses the user identity to create a personalised clipboard for copying content independent of other users actions, or automate repeated actions like entering credentials to a website. The iD-widgets by Ryall et al. [16] enhance the functionality of standardised GUI

¹<http://www.microsoft.com/surface/en/us/default.aspx> (as of 29 June 2011)

elements with the notion of who is using it. In a multi-user setting, this allows for reuse of a widget by having personalised views or customised functionalities. In addition, it reduces clutter and saves screen real estate.

1.2 Problem statement

Contemporary tabletop computers lack the ability to link an action to the identity of its originator. This challenge tabletop applications where *access control* and a *dynamically customised user interface* is part of the user experience.

We ask, is it feasible to build a system that allows for *non-anonymous user interaction on tabletop displays*?

In particular we ask:

- Can the system be used for building UI controls?
- Can the system be implemented as a framework for others to use?
- Can the framework be made secure?

In addition to these question we ask if the system can run on standard hardware, such as the *Microsoft Surface* and an Android powered smartphone?

1.3 Research methods

The questions asked in the section above, will be answered using a number of methods. There will be

- an investigation of non-anonymous user interaction in relation to tabletop displays and surfaces,
- a user experience design,
- an analysis of the security threats,
- an implementation as a framework,
- a proof-of-concept by showing how to use the framework for programming a non-trivial application, and
- a usability test of the framework.

In order to keep the project manageable, the framework will only support single touch events. Multi-touch events are available both on the Android powered devices, as well as on the *Microsoft Surface*, so it might be possible to extend the framework to support multi-touch events in the future.

In addition to this, the framework will only feature a simple automatic tabletop discovery process, leaving more advanced and robust methods to be implemented in later iterations.

1.4 Results

The main contribution of this thesis is the *NAI* framework, that allows for non-anonymous user interaction with tabletop displays using standard hardware like the MSS and Android powered smartphones. The smartphone acts as a two-way mediator between the user and the MSS when it is placed on the tabletop surface. On the one hand, the area of the MSS that is covered by the phone is projected on to the display of the phone, and on the other hand, the users' touch events on the display of the phone are forwarded to the tabletop.

The source code has been made public available and appendix A describes where to find it.

The investigation into prior work shows that the work presented here is unprecedented. Specifically, the user experience design integrates with the tabletop and does not require special gestures, it is secure, and it uses standard hardware. Three features that in combination distances this from related work.

The security analysis establishes that a secure system is indeed possible, but there is a separation of concern. The framework handles security threats by using a secure data connection and requires a pairing procedure for pairing a secure connection to a physical phone. A developer using the framework must implement a sufficient authentication mechanism in accordance with the requirements of the application, and the end user must make sure no other persons use the phone while it is connected to the tabletop.

The framework features some UI controls, three sets of *identified events*, and a customisable authentication mechanism. For example, the `IdentifiedSurfaceButton` extends the functionality of a normal `SurfaceButton` by being able to react to the identity of the user touching, and the `IdentifiedViewPort` can hide and reveal information from smartphones hovering over the control based on the identity of the smartphone users. Developers using the framework can use *identified touch events*, *identified hover events*, and *identified person events* to build their own controls and functionality.

The framework is used for building a non-trivial application for the restaurant of the future. The application takes advantage of the *identified events* by integrating the framework UI controls `IdentifiedViewPort`, `IdentifiedSurfaceButton`, and `PersonalizedView` with UI controls of the Surface API, and thereby allowing automated ordering and payment.

Besides the simple fact that the framework does operate as intended, the evaluation establishes that users quickly learn to use features of the framework. The most significant evaluation result is a list of five design guidelines. Based on the comments of users, this list can help leverage the quality of an application. For example, it is better to lessen the number of times the phone is moved, by moving an item to the phone, instead of moving the phone to the item. Another example is, that an application must feature visual clues to where to use the phone, especially when

using the `PersonalizedView` control. Otherwise users might overlook the functionality.

1.5 Thesis overview

The remainder of this thesis has been structured as follows:

Related work (chapter 2) investigates prior work in relation to non-anonymous user interaction on tabletop displays and surfaces.

The NAI framework (chapter 3) describes the user experience design as well as how to use the framework for building applications.

The *Security analysis* (chapter 4) describes the security threats. This have affected the *Implementation* (chapter 5) that describes in details the important aspects of how the system has been built.

Using the NAI framework for application building (chapter 6) documents that the framework can be used for building a non-trivial application, and the *Evaluation* (chapter 7) documents a usability test of the framework.

The thesis ends with a *Discussion* (chapter 8) of the results, and a *Conclusion* (chapter 9) that summarises the results.

Related work

Research in co-located user interaction is not a new thing. Even before large surface displays were widely adopted by the research groups investigation into multi user settings were conducted. For example Steward et al. [22] shared a drawing application on a single display, and users were represented by a mouse pointer. Bier et al. [1] also allowed users to register to a mouse pointer and perform personalised interaction. The Pebbles project [11] used a number of PDAs as input devices to a shared display. Each PDA representing its user had the *RemoteCommander* application running that allowed turn-based control of the shared PC, but essentially it was a remote control. The *PebblesDraw* was a shared drawing application that investigated the challenge of having different PDAs operate on a shared experience. The weight of this kind of research has been put on the investigation of the interaction itself. That is, an investigation into *how* a user interface could be designed when moving beyond the single user environment of PC into a collaborative setting. More recent research adds to this area as Ryall et al. [16, 17] try to overcome

user interface challenges such as access control, preference optimisation, reach, and surface clutter

when operating with tabletops and multi user surfaces with the concept of *IdWidgets*.

Although these studies are all important, this thesis has another focus. We provide a technical solution as a framework that allows for non-anonymous user interaction on tabletop displays. Therefore the remainder of this chapter will discuss previous work on the *technical aspects* of how non-anonymous user interaction for wall-sized surfaces and tabletop has been solved hitherto.

The *DiamondTouch MultiTouch* [4] by Dietz et al. is an important contribution. Being a touch sensing surface only (it is not a display in itself) it is required that the displayed image is projected upon it. The *DiamondTouch* consists of a horizontal and a vertical aligned array of antennas with electrically conductive material. To protect them from direct touch and the environment, these are covered with an insulating

layer, that in return allows liquids to be spilled and even fire to be lit during operation.

When a user touches the *DiamondTouch* a capacitive coupled circuit is completed. The transmitter sends a unique signal through each and every antenna, and when a user touches a subset of antennas, the signals travels through the user and back to the receiver via a capacitive connector in the chair. The receiver interprets the signals, and transform them into a personal touch coordinate. This research is further extended by Dietz et al. [3] to include physical objects like buttons, levers, and even a resistive touch screen that allows identified single-touch operations on a LCD display.

Apart from the problems raised by the fact that the image is projected from above, and hence is easily obstructed, the real challenge lies in that this solution “only” allows for a *discrimination* between the users seated around the table, i.e it can distinguish *user 1* from *user 2*, but it cannot uniquely authenticate a person. In essence you are where you sit, and nothing more. Other measures must be done in order to pair the seat to an identity. Because the identifying signals travel through the body the identification mechanism is easily challenged. If *user 1* touches *user 2* while touching the surface, the touch is detected as being performed by both, and if *user 1* lifts himself from the chair while touching *user 2*, the touch will be interpreted as originating from *user 2* only. In a scenario where an amount of security is needed, this is much too insecure, while gaming scenarios might fit perfectly.

Instead of embedding the identification into the touch technology itself, other projects have taken another approach by utilising computer vision. That seems to integrate nicely with surface displays that registers touch events by processing the images captured by video cameras because the hardware, and possibly software, is already present.

HandsDown [19] is a research project by Dominik Schmidt et al. that can identify the user touching the tabletop by recognising the hand-contour. For example placing a stretched out hand on a digital item on the tabletop associates it with the user. Likewise the user can un-tag the digital item again by replacing the hand.

The simplicity of the user interaction is at the same time both an advantage and a drawback. The advantage is the simplicity itself by not requiring any extra devices. The drawback of such an approach is that identified interaction with the tabletop must adapt to the notion of placing your hand with spread fingers in order to let the surface detect the identity of the user. It may result in an unnatural or limited interaction when extending its usage beyond the exemplifying scenarios of the article. Extending the usage scenarios to hospitals or other places where actual authentication is a requirement, using only a hand contour is simply not viable. The hand contour can be reproduced without much effort, and other authentication factors are needed as well, but that will ruin the simplicity. Imagine the user had to enter a PIN code every time he placed

his hand on the surface.

Other research projects are centered around specialised hardware or objects that must be worn or used, in order to detect a personalised touch. This gives a more freely interaction than for example *HandsDown*.

The *The IR Ring* [15] by Volker Roth et al. is a good example of this other approach. Inspired by the Java Ring, the IR ring should be worn when interacting with a multi-touch display. Each ring transmits a pseudo random bit sequence via IR that is recognised by the capturing system of the tabletop. Because the system knows the bit sequence it allows the otherwise anonymous touch events to be associated with an unique user. Although the IR signal was not encrypted, the authors argue that it could be encrypted for increased security, but they are in general aware of the trade-off between strict security policies, computational resources and usability.

In a setting where you need two-factor authentication, or even more strict authentication mechanisms, the otherwise simple interaction with the tabletop becomes cumbersome. For example if the system requires you to enter a PIN code every time the IR ring is used for a personalised touch event, that will quickly become a major obstacle when using it.

A similar project to the IR ring is *IdWristbands* by T. Meyer and D. Schmidt [9]. It uses Arduino based LilyPads controller boards attached to a wristband. The board controls two LEDs sending IR light in order to identify the user interacting with a MSS. This project was aiming for differentiating the users in contrast to the IR ring, that ultimately wants to be able to authenticate its users.

Taking the IR light even further the *pPen* [12] research project uses a pressure sensitive pen identified by an IR light emitter at the tip, and a RF module that sends data to the system. A user is authenticated to the system (or really the pen) by having the user re-write a password that was entered previously. The computer compares the pressure variance of the handwriting to the database and authenticate if a match can be made. Undoubtedly an elegant solution, it relies on probability when comparing the pressure variance to the database. A 100% match is not likely, therefore the solution needs to set a threshold that is less in order to find a match. False positives may increase as the user base increases. While this may be sufficient in some settings, it can be insufficient in others. Our solution relies on encrypted wireless connection with a completely customisable authentication mechanism, without requiring specialised hardware. In addition to this, the *pPen* rely on pressure events of the tip. Our solution have, besides the *touch* event also a *hover* event, and hence provide a larger set of interaction possibilities for the UI designer.

Fiduciary-tagged gloves by Marquardt et al. [8], are gloves decorated with special tags supplied as a part of the API to the MSS. Tags are put on every finger, palm, top of hand, and on the side. This allows for a highly differentiated set of touch possibilities, while operating with a clear distinction between the users (or gloves as it is). This does call for novel interaction design, but it is not useful in a setting with even a small degree of security, as there is no way a touch can be authenticated by itself. Even though a glove may be owned by a person, the tags can easily be replicated.

Mobile phones, and especially smartphones, of today are capable of much more than making a phone call. These capabilities have been exploited in research projects that pair up a phone and a surface computer in a way that allows for identified and potentially authenticated interaction. Furthermore, targeting hardware like a mobile phone has the advantage, that users are not burdened with extra hardware. Users of today already own a mobile phone, and it is just a matter of extending its functionality by installing software to allow the phone to be used in new settings.

In the *PhoneTouch* [18] project, a smartphone is used to identify the touch of a user. By tapping the corner of a phone on to a tabletop right next to the touch of a user, the computer associates the identity provided by the phone to the touch event of the user. The phone itself detects the tap using the built-in accelerometer, and sends a timestamp to the tabletop via a wireless connection. The optical system of the computer detects the tap and records the timestamp and pairs the tapping event with the event sent from the phone.

The interaction possibilities of *PhoneTouch* is restricted by the fact that you need to tap with your phone whenever an identified action is wanted. Smartphones of today have enough CPU power to be able to supply a strong authentication and a secure connection to a computer, but in this setting the authentication will be questioned by the uncertainty of using timestamps and associating taps to touch events by probability. As the number of simultaneous users increases, the uncertainty will increase as well.

Schöning et al. [21] have designed a system that allows for spontaneously authentication and interaction with a multi-touch surface wall. It is designed to allow smaller regions of a large surface to be associated with an authenticated user, hence adding support for multi-touch application in a critical setting like an emergency control room. The system requires a user to tap and hold a mobile phone on to the surface. The mobile phone detects the tap either via the built-in accelerometer or using the microphone. Then it initialises the authentication with the surface computer by connecting via Bluetooth while lighting up the flash. The surface computer pairs the flash signal to the Bluetooth signal, and if both signals are received within an acceptable time frame and no ambiguity from other signals are disturbing, that area or that control the flash light is pointing to is associated with the authenticated person. Any conven-

tional touch in that area will be controlled by the authenticated person. Detection of the flashes requires an additional camera that operates in the visual spectrum of light to be set up behind the surface wall.

Although this solution allows a more freely interaction than for example *PhoneTouch*, the interaction is not completely fluent. The interaction is obstructed whenever a control requires an authentication before it can be activated.

To summarise, prior work differs internally in a number of ways. Some projects requires specialised hardware to be able to detect the identity of a user [12, 4, 3, 9, 15]. The *PhoneTouch* [18] project utilises a standard smartphone, but interaction is restricted to tapping with the corner of the phone on the tabletop. *HandsDown* [19] requires a special hand gesture to operate, but does use other hardware than the tabletop itself. Schöning et al. [21] uses a standard mobile phone equipped with a flash, with the potential of having a strong user authentication, but authentication must be performed every time a user wants to use a control, somewhat obstructing a fluent user experience.

In comparison to prior work, our work have been designed so that the user experience integrates with the tabletop and does not require special gestures. In particular, it is implemented as a framework, and is secure. In addition to this, an important detail is that the framework uses standard hardware.

The *NAI* framework

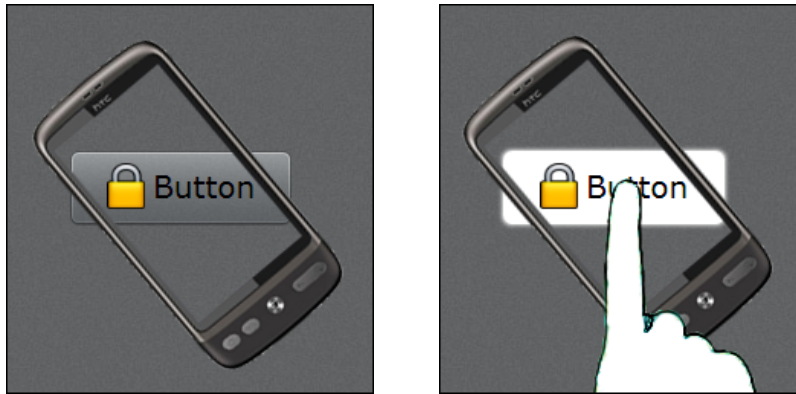
A MSS tabletop cannot natively link an action to the identity of its originator. The *NAI* framework provides a solution to this problem, which tries to be as close to the user experience of a tabletop display as possible. The idea is not to require the users to learn new interaction schemes or use specialised hardware. Section 3.1 describes the general user experience design which is the foundation of the *NAI* framework. Section 3.2 walks through the developer API provided by the *NAI* framework and for each component in the API, explains the user interaction design as well as how to use the component as a developer. Section 3.3 describes the required setup procedure for the smartphone before identified interaction is possible.

The framework features described in the following sections are also illustrated in a video found here:

<http://www.youtube.com/watch?v=H9n6vRNqNR8>

3.1 General user experience design

The idea behind the *NAI* framework is to use a smartphone to mediate a user's interactions with the tabletop. The mediation is two-way, meaning that the smartphone does both present output as well as accepting input. Output is presented on the smartphone display when it is placed on the tabletop. The phone displays the segment of the tabletop screen which the phone covers. This gives a glass like effect where you can “see through” the phone, see figure 3.1(a). If the phone is moved over some input accepting control on the tabletop, like a button, the user may click the button on the smartphone display. That click event is sent to the tabletop computer, where it is treated as an *identified event*, because the tabletop assumes that it is the smartphone user who clicked the button, see figure 3.1(b). Besides clicking the screen, the user can interact by moving the smartphone. Moving the smartphone closely resembles moving a pointing device like a mouse.



(a) The smartphone screen behaves like a piece of glass when placed on the tabletop.

(b) The smartphone's touch screen is used for identified inputs, like clicking a button.

Figure 3.1: The smartphone acts as a two-way mediator between the tabletop and the user.

3.2 Usage

The *NAI* framework encapsulates the complexity of the identified interaction design introduced above in a developer API with a set of components. The API can be used to create new tabletop applications that rely on and utilise user identification. Most of the components in the API are related to the tabletop, where applications are written in WPF (Windows Presentation Foundation), which is a mixture of XAML and C# code. The Surface API is built on top of WPF, and the *NAI* framework is built upon the Surface API. On the smartphone the programming is done on top of the Android software stack. A diagram overview of the developer API of the *NAI* framework is depicted in figure 3.2. Components below the dashed line represent the code that makes the framework operate, and components above the dashed line constitute the developer API. The following list summarises the available components in the developers API:

| Name | Type | Platform |
|------------------------------|-----------------|----------|
| IdentifiedInteractionArea | Items control | MSS |
| IdentifiedSurfaceButton | Control | MSS |
| IdentifiedViewport | Control | MSS |
| PersonalizedView | Content control | MSS |
| <i>Identified events</i> | Event set | MSS |
| <i>Custom authentication</i> | Mechanism | Both |

The first five components are described further below. *Custom authentication* is related to the setup procedure and is hence described in section 3.3.

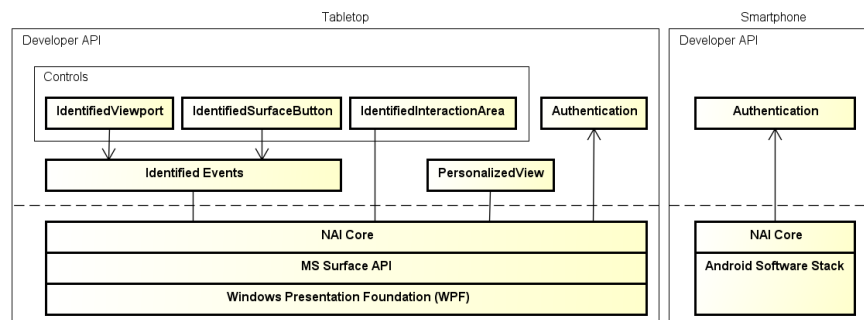


Figure 3.2: Developers view of the *NAI* framework.

3.2.1 IdentifiedInteractionArea

The key to the *NAI* framework is the `IdentifiedInteractionArea`. It is the required, logical container that make identified controls work. Identified controls are any custom controls or UI element that make use of the *identified events* provided by the *NAI* framework.

3.2.1.1 User experience design

The control in itself is not visible to the user, because it is basically an empty container. Figure 3.3 shows an example of an application where the right side of the tabletop display contains an `IdentifiedInteractionArea` with some identified controls inside. The coloured borders are added to illustrate that the `IdentifiedInteractionArea` only covers the right side of the tabletop.

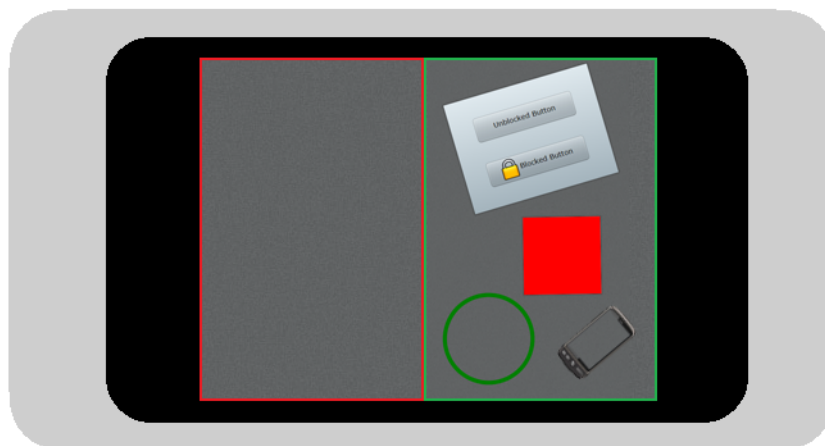


Figure 3.3: An `IdentifiedInteractionArea` filling the right side of the tabletop display.

3.2.1.2 Developer API

The `IdentifiedInteractionArea` component is an *Items control*, with the same basic property as the WPF class `ItemsControl`: It presents a collections of items. The control is special, because it is an ancestor control, which enables identified interaction on the elements placed as descendants of the control.

The `IdentifiedInteractionArea` control must be placed as an outer element in the element tree and must only be declared once. Even though it is recommended to let the `IdentifiedInteractionArea` control fill the entire tabletop display area, it is not required. In the example application shown in figure 3.3, the `IdentifiedInteractionArea` control only fills the right side of the surface display. The XAML code for that application is:

```
<Grid>
  ...
  <Border Grid.Column="0" BorderBrush="Red"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
  </Border>

  <Border Grid.Column="1" BorderBrush="Green"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
    <id:IdentifiedInteractionArea>
      ...
      <!-- Identified Controls -->
      ...
    </id:IdentifiedInteractionArea>
  </Border>
</Grid>
```

3.2.2 IdentifiedSurfaceButton

The `IdentifiedSurfaceButton` is an extended version of the `SurfaceButton` from the Surface API, which again is an extended version of the `Button` from WPF. Hence the name becomes `IdentifiedSurfaceButton`. The control is suitable in scenarios where the user must choose or confirm something important, that requires an *identified event*.

3.2.2.1 User experience design

Since the button extends the `SurfaceButton` from the Surface API, the look and feel are not different. The button is depicted in figure 3.4.

3.2.2.2 Developer API

The XAML code for an `IdentifiedSurfaceButton` looks like this in its most simple form:

```
<id:IdentifiedSurfaceButton IdentifiedClick="OnIdentifiedClick">
  Click me!
</id:IdentifiedSurfaceButton>
```

The `IdentifiedSurfaceButton` have been designed so the usage resembles that of a `SurfaceButton`. A `SurfaceButton` raises its own `Click` event when pressed, and an event handler is added by using the



Figure 3.4: The IdentifiedSurfaceButton control with a smartphone on top.

Click property. The IdentifiedSurfaceButton raises its own IdentifiedClick event when pressed via the screen of a connected smartphone, and an event handler is added by using the IdentifiedClick property. The code below shows the event handler registered above.

```
...
private void OnIdentifiedClick(object sender,
    RoutedIdentifiedEventArgs e)
{
    ...
}
...
```

The RoutedIdentifiedEventArgs type contains the inherit event arguments, but also information about the identity of the user. This allows the developer to use the *identified event* and customise the behaviour accordingly. More information on events and event handling can be found in section 3.2.6.

As mentioned above the IdentifiedSurfaceButton is a specialisation of the SurfaceButton and the inherit Click event still remains active. A developer can therefore distinguish between un-identified and identified click events.

The un-identified event can be blocked by setting the property BlockClickEvent to True. It prevents both the Click event from being raised and the animation from being activated when a user presses the button. The following example shows a button with where un-identified events will be blocked.

```
<id:IdentifiedSurfaceButton
    IdentifiedClick="OnIdentifiedClick"
    BlockClickEvent="True">
    Click me!
</id:IdentifiedSurfaceButton>
```

3.2.3 IdentifiedViewPort

The IdentifiedView control is used to hide information from specific users.

3.2.3.1 User experience design

The idea is, that a solid shape covers an area of the tabletop display and only certain users can, by placing their smartphone on top of it, see what is behind the solid shape. The control is illustrated in figure 3.5. A customisable filter function delegate determines which users are allowed to see through the shape, and the layout of the shape is specified by the developer using the control.

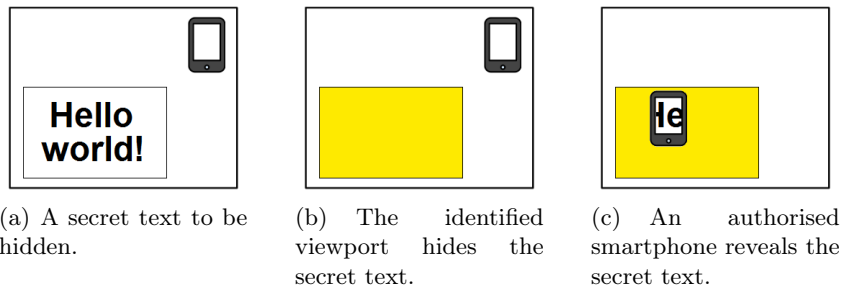


Figure 3.5: The IdentifiedViewport control.

3.2.3.2 Developer API

In the example shown in figure 3.5 the yellow IdentifiedViewport hides the text. The XAML code below shows how to create such a viewport:

```
<id:IdentifiedViewport x:Name="MyIdentifiedViewport" Fill="Yellow">
  <id:IdentifiedViewport.Base>
    <RectangleGeometry Rect="0,0,700,500" />
  </id:IdentifiedViewport.Base>
</id:IdentifiedViewport>
```

The Base property defines the shape of the covering area, and the Fill property the colour. The colour is an instance of a Brush and more advanced colouring possibilities exists by adding custom brushes in the code-behind file to the Fill property. In the example above the covering shape is a rectangle, but it can be any shape as long as it is a specialisation of the Geometry type from the WPF framework.

By default, any connected smartphone can see through the IdentifiedViewport, but a filtering delegate function can be added for customised behaviour.

Example Adding a customised filter delegate to the IdentifiedViewport in the code behind file:

```
...
MyIdentifiedViewport.FilterDelegate = new
  IdentifiedViewportFilterDelegate(UserFilter);
...
private bool UserFilter(RoutedIdentifiedHoverEventArgs e){
  if (e.ClientId.Credentials.UserId.Equals("berglund@itu.dk"))
    return true;
  return false;
}
...
```

This authorises only users with the user id *berglund@itu.dk* to see through the viewport, and thereby see the hidden information. The filtering function can be set at any time during runtime.

Applications built for a tabletop display typically allow the visual elements to move, rotate, and/or scale based on users interactions. Moving an `IdentifiedViewport` will invalidate its layout. For example when considering figure 3.5(c), moving the yellow rectangle while not moving the phone, will not close the viewport. Whenever an instance of `IdentifiedViewport` is moved, the method `Moved` must be called. That will update the layout and place the port holes correctly.

3.2.4 PersonalizedView

The image displayed on the smartphone is considered as being private to the owner of the phone. This gives the opportunity to display information targeted a specific user. That functionality is called a `PersonalizedView` within the *NAI* framework.

3.2.4.1 User experience design

A `PersonalizedView` can be any visual element, like a text, a picture or even a small dialog with buttons, and it can for example be added to a user's phone display during the handling of an *identified event*. Figure 3.6 shows an example, where users get a small personal colour palette to choose the colour of an ellipse. The two smartphones have different colour palettes.

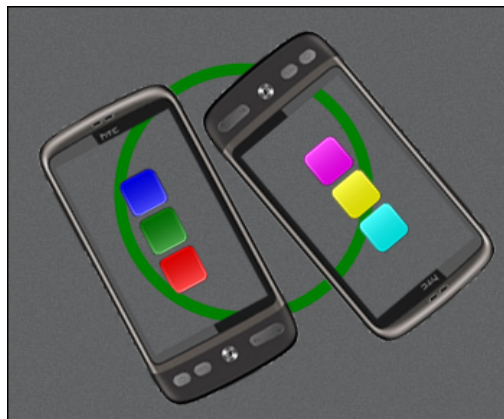


Figure 3.6: A `PersonalizedView` can be used to give a differentiated user experience. Here is an example where two users get different colour palettes.

A `PersonalizedView` is not projected directly onto the screen of the phone, instead it is displayed underneath the phone directly on the tabletop display. If the phone is moved, the `PersonalizedView` automatically follows along.

3.2.4.2 Developer API

A *PersonalizedView* is a *Content control* type, with the same basic property as the WPF class *ContentControl*: It holds a single piece of content. That piece of content must be a specialisation of the WPF class *UIElement*.

In the example application in figure 3.6 the *PersonalizedViews* are added and removed based on whether the phones hover over the ellipse. The *identified hover events*, introduced in section 3.2.6, are used to achieve this. Different users get different *PersonalizedViews*. The C# event handling code which adds and removes the *PersonalizedView* colour palettes looks like this:

```
private void Ellipse_HoverOver(object sender,
    RoutedIdentifiedHoverEventArgs e)
{
    if (e.ClientId.Credentials.UserId.Equals("berglund@itu.dk"))
    {
        e.ClientId.PersonalizedView.Add(
            new ColourPickerCMY(MyEllipse));
    }
    else
    {
        e.ClientId.PersonalizedView.Add(
            new ColourPickerRGB(MyEllipse));
    }
}

private void Ellipse_HoverOut(object sender,
    RoutedIdentifiedHoverEventArgs e)
{
    e.ClientId.PersonalizedView.Remove();
}
```

The *RoutedIdentifiedHoverEventArgs* instance provides access to the identity of the originator of the event (*e.ClientId*) containing an interface to add and remove a *PersonalizedView*. In the example, user *berglund@itu.dk* gets a different *PersonalizedView* than everybody else. More information about the events, event handling and the identity of the user follows in the following sections.

3.2.5 User identity

Within the *NAI* framework, a user's identity is represented via an instance of the class *ClientIdentity*. As can be seen from the diagram in figure 3.7, it contains references to an instance of *TagData* and *ClientCredentials* as well as an interface to add and remove a *PersonalizedView*.

In order for the *NAI* framework to operate it is required that a small tag is stuck onto the back of the smartphone. The instance of *TagData* represents this visual marker that is used to track the movements of the phone. Section 3.3, Setup, and chapter 5, Implementation give more in-depth information on the usage of the tag.

An instance of *ClientCredentials* represents the information the framework knows about a specific user. Different applications have different requirements. The developer has complete control of the type of

information needed, but the framework requires `ClientCredentials` to contain at least a user id. How the developer obtains the control is explained in section 3.3.2.

The identity of the user is bound to the events that the user causes. That is, an instance of `ClientIdentity` is sent along to the event handler when ever the users trigger an *identified event*. Developers can then use this information to personalise the user experience.

3.2.6 Identified events

The *NAI* framework raises events related to identified interaction, and these can be used in many ways. The events may be used to create sophisticated controls, like the `IdentifiedViewport` or may be used alone in more simple identified interaction scenarios.

The set of *identified events* in the *NAI* framework falls into three classes. First, a set of *identified touch events* which is related to finger based input on the smartphone. Secondly, a set of *identified hover events* which is related to what the smartphone screen is displaying. Finally, a set called *identified person events*, which is related to the presence of users (represented by their smartphones) at the tabletop. All event types of the framework are placed in one single class `IdentifiedEvents`.

A detailed overview of the three sets of *identified events* in the *NAI* framework is depicted in figure 3.7. Like in WPF, all *identified events* related directly to user input are raised in pairs. One tunneling and one bubbling event. WPF convention prefix names of tunneling events with `Preview-`. In the diagram (figure 3.7) as well as in the following sections, the listing and explanation of the tunneling events have been omitted, as they are identical to bubbling events. Tunneling events are used in some of the code examples.

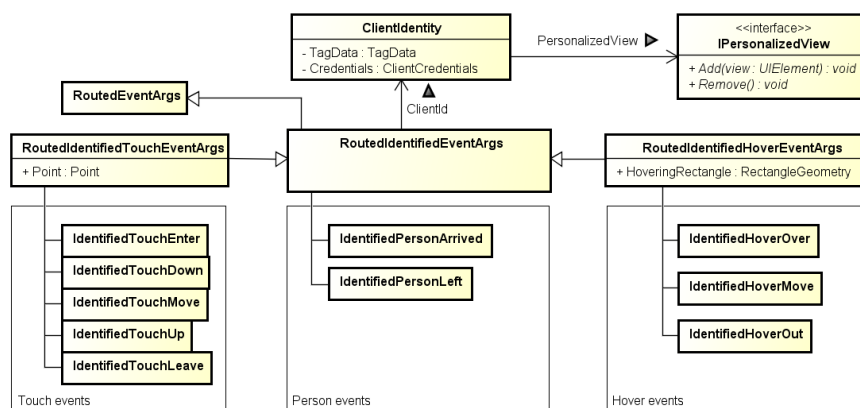


Figure 3.7: Overview of the *identified events*

3.2.6.1 Identified touch events

The *identified touch events* are normal touch events augmented with the identity of its originator. As for normal touch events, an identified touch raises a series of *identified touch events* in a specific order¹. The events are:

IdentifiedTouchEnter Raised once on every element the user starts touching.

IdentifiedTouchDown Raised once on the *first* element the user starts touching.

IdentifiedTouchMove Raised every time the user moves the finger on elements the user already is touching.

IdentifiedTouchUp Raised once on the *last* element the user touched after the touch has ended.

IdentifiedTouchLeave Raised once on every element the user no longer touches.

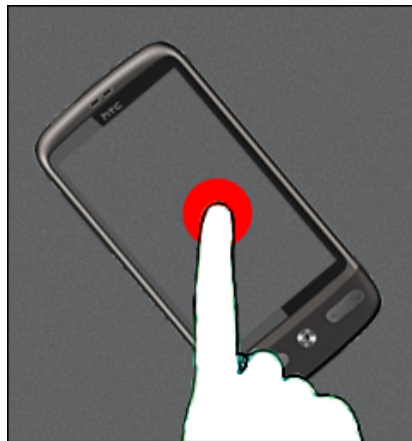


Figure 3.8: Example application showing the usage of the *identified touch events*.

Example Figure 3.8 shows a simple application that uses *identified touch events* to position, re-position, and remove coloured balls. When a finger touches the smartphone display, a ball coloured with the user's favorite colour appears underneath his finger. If the finger is moved, the ball follows along, and if the finger is lifted the ball disappears. An example of how this can be implemented is listed below. The XAML code that defines the basic layout looks like this:

```
...
<Border Name="myBorder" Background="Transparent">
  <Canvas Name="MyCanvas"/>
</Border>
...
```

¹<http://msdn.microsoft.com/en-us/library/ms754010.aspx> (as of 11 July 2011)

The event handler of three *identified touch event* are added to the `Border` in the constructor of the code behind file:

```
...
MyBorder.AddHandler(IdentifiedEvents.PreviewIdentifiedTouchDownEvent,
    new IdentifiedEvents.RoutedIdentifiedTouchEventHandler(
        OnIdentifiedTouchDown));

MyBorder.AddHandler(IdentifiedEvents.PreviewIdentifiedTouchMoveEvent,
    new IdentifiedEvents.RoutedIdentifiedTouchEventHandler(
        OnIdentifiedTouchMove));

MyBorder.AddHandler(IdentifiedEvents.PreviewIdentifiedTouchUpEvent,
    new IdentifiedEvents.RoutedIdentifiedTouchEventHandler(
        OnIdentifiedTouchUp));
...
```

The implementation of the three event handlers for the coloured ball application look like this:

```
...
private void OnIdentifiedTouchDown(object sender,
    RoutedIdentifiedTouchEventArgs e)
{
    Brush favoriteColour = GetFavoriteColour(e.ClientId);
    Ellipse elip = CreateEllipse(favoriteColour);
    SetCenterPosition(elip, e.Point);
    MyCanvas.Children.Add(elip);
    ellipsesInUse[e.ClientId] = elip;
}

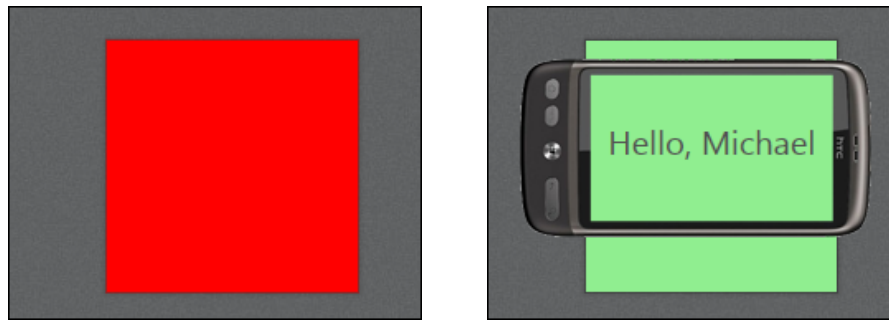
private void OnIdentifiedTouchMove(object sender,
    RoutedIdentifiedTouchEventArgs e)
{
    SetCenterPosition(ellipsesInUse[e.ClientId], e.Point);
}

private void OnIdentifiedTouchUp(object sender,
    RoutedIdentifiedTouchEventArgs e)
{
    MyCanvas.Children.Remove(ellipsesInUse[e.ClientId]);
    ellipsesInUse.Remove(e.ClientId);
}
...
```

The `OnIdentifiedTouchDown` handler for the `PreviewIdentifiedTouchDown` event creates an ellipse with the user's favorite colour. The identity of the user, the `ClientIdentity` instance is supplied by the `RoutedIdentifiedTouchEventArgs` and is used to get the correct favorite colour via the `GetFavoriteColour` method. The ball is positioned on the canvas based on the `Point` value also supplied by the `RoutedIdentifiedTouchEventArgs`. For future event handling, the ellipse is stored in a dictionary indexed by the instance of `ClientIdentity`.

3.2.6.2 Identified hover events

Moving a smartphone across the tabletop display is similar to moving a classic computer mouse pointer across a computer monitor. As for the mouse pointer, the *NAI* framework uses a set of hover events to indicate what the smartphone is hovering over. This set of events is not derived from or related to any existing events on either of the platforms involved. The *identified hover events* in the *NAI* framework are identity augmented



(a) The rectangle is red when no smartphone hovers over it.

(b) The rectangle turns green and shows a greetings message when a smartphone hovers over it.

Figure 3.9: Example application showing the usage of the *identified hover events*.

versions of the normal set of hover events which are applicable for hover enabled input devices like a computer mouse. The computer mouse uses hover events a bit different, because it by design only hovers over a single XY coordinate at any given time. In comparison, the smartphone hovers over the area covered by its display. This affects when the *identified hover events* are raised. The *NAI* framework defines a smartphone as being hovering over an element if any point of the smartphone display hovers over the element. The set of *identified hover events* are:

IdentifiedHoverOver Raised once on elements the smartphone starts to hover over.

IdentifiedHoverMove Raised every time the smartphone moves on elements the smartphone is already hovering over.

IdentifiedHoverOut Raised once on elements the smartphone is no longer hovering over.

Example Figure 3.9 shows a simple application that takes advantage of the *identified hover events*. A red rectangle changes colour, and shows a personalised greeting when a connected smartphone completely hovers over the rectangle. The following code shows one way of implementing this behaviour. The event handler is setup directly in the XAML code, just as you can with events specific for WPF or Surface API:

```

...
<Border Background="Red" Name="MyBorder"
    ide:IdentifiedEvents.IdentifiedHoverOver="OnIdentifiedHover"
    ide:IdentifiedEvents.IdentifiedHoverMove="OnIdentifiedHover"
    ide:IdentifiedEvents.IdentifiedHoverOut="OnIdentifiedHover">
    <TextBlock Name="MyTextBlock" Visibility="Hidden"/>
</Border>
...

```

The logic for the event handler is declared in the code behind file:


```

...
private void OnIdentifiedHover(object sender,
    RoutedIdentifiedHoverEventArgs e)
{
    if ((e.RoutedEvent == IdentifiedEvents.IdentifiedHoverOverEvent ||
        e.RoutedEvent == IdentifiedEvents.IdentifiedHoverMoveEvent) &&
        IsRectangleInsideBorder(e.HoveringRectangle))
    {
        MyBorder.Background = Brushes.Green;
        MyTextBlock.Text = "Hello, " + e.ClientId.Credentials.UserId;
        MyTextBlock.Visibility = Visibility.Visible;
    }
    else
    {
        MyBorder.Background = Brushes.Red;
        MyTextBlock.Visibility = Visibility.Hidden;
    }
}
...

```

The `RoutedIdentifiedHoverEventArgs` instance passed along when handling *identified hover events* includes a rectangle, called `HoveringRectangle`, that is sized and rotated to exactly mark the smartphone display. The method `IsRectangleInsideBorder` (not implemented) can use the rectangle together with the `VisualTreeHelper` from the WPF framework to determine whether or not the rectangle is completely hovering over the `Border`. As shown in figure 3.9(b), the smartphone display must be placed completely within the `Border` before the colour changes and the `UserId` becomes visible.

3.2.6.3 Identified person events

The fact that users connect their smartphones to the tabletop in order to do identified interaction, allows the tabletop to keep track of who is connected, and hence who is present around or near the table. Two event types are raised when users arrive at or leave the tabletop:

IdentifiedPersonArrived Raised once when the user has successfully completed the pairing of the smartphone tag with the tabletop (step 3 in section 3.3.1).

IdentifiedPersonLeft Raised once when the user's smartphone disconnects from the tabletop.

These events are not directly related to user input, so they are only raised once in a bubbling version. The events are always raised on the `IdentifiedInteractionArea` control. There is no example of how these events are used, as adding and declaring an event handler resembles the examples from above. The base class `RoutedIdentifiedEventArgs` is sent via the event handler which only provides access to the instance of `ClientIdentity`.

3.3 Setup

The smartphone runs a special application, which is part of the *NAI* framework. The application must be installed before identified interac-

tion with a *NAI* framework application on the tabletop is possible. The current implementation of the *NAI* framework only includes an application for Android powered phones, version 2.1 or newer. Furthermore, the phone is required to have a MSS tag stuck on the back in order to let the tabletop computer detect the presence of the phone on the tabletop surface as well as its position and rotation. The tag must preferably be placed in the middle of the back, and oriented towards the top of the phone. An ideal position is depicted in figure 3.10.



Figure 3.10: A MSS tag stuck on the back of an HTC Desire.

3.3.1 Smartphone application startup procedure

When the smartphone application is started, it tries to automatically discover a tabletop computer on the same network and establish a secure (SSL) connection to it. The tabletop must have a SSL certificate installed in the local key store in order to successfully complete the SSL handshake.

The current implementation of the *NAI* framework only provides a minimum implementation of automatic tabletop discovery and SSL connection setup. It is not an objective of this thesis to provide a nice customisable developer interface to these areas of the framework.

Once the connection has been setup, there is a mandatory two or three step setup procedure the user must go through to prepare the phone to be used for identified interaction on the tabletop. The steps are:

1. User authentication
2. Smartphone and tag pairing
3. Smartphone calibration [Only the first time]

Each step is explained below. When the steps have been completed, the smartphone begins displaying the tabletop segment it covers and it accepts touch inputs.

3.3.1.1 Step 1: User authentication

The first thing the user sees is an authentication dialog on the smartphone. The *NAI* framework allows users to use any smartphone as long as they are able to provide the necessary credentials to be successfully authenticated.

The authentication mechanism in the *NAI* framework is customisable, meaning that the type and amount of credentials required to be authenticated may differ. It is part of the developer API of the framework that the authentication mechanism can be customised and scaled to different application contexts and security requirements. Section 3.3.2 describes how a custom authentication mechanism can be implemented and plugged into the framework.

The framework provides a minimum default authentication mechanism implementation where the user simply enters a user id, see figure 3.11.



Figure 3.11: User authentication. The user provides the necessary credentials.

3.3.1.2 Step 2: Smartphone and tag pairing

When the user is successfully authenticated, it is time to pair the smartphone with the tag. At this point the tabletop has established a secure connection for communication and has recognised the tag on the back of the smartphone, but for security reasons, it may not assume an association between a connection and a tag (see chapter 4 for more).

The pairing procedure is depicted in figure 3.12. The tabletop displays a PIN code on both sides of the phone, but it is only revealed if the user press and holds the button next to the PIN code area. When the user has entered the correct PIN code on the smartphone, the pairing is completed.

3.3.1.3 Step 3: Smartphone calibration

The last step in the setup procedure is calibration of the smartphone. The purpose is to tell the tabletop how big the smartphone is to allow the right screen segment to be transmitted and displayed on the smartphone. The



Figure 3.12: Pairing procedure. The user enters the PIN code, shown on the tabletop, on the smartphone.

calibration step is only necessary the first time the smartphone is used on the tabletop, because calibrations are saved by the tabletop.

The calibration step is depicted in figure 3.13. A special calibration control is displayed on the tabletop where the user has to place the phone in the center and adjust the four dashed lines to mark the size of the smartphone display. The calibration is completed when the user taps on the smartphone display and the calibration is saved.

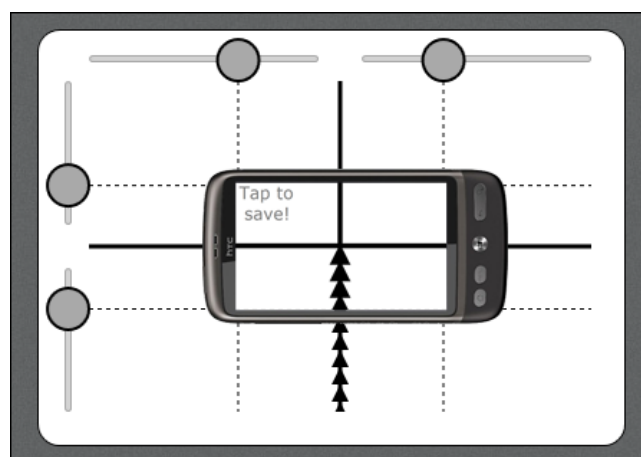


Figure 3.13: Smartphone calibration. The user places the smartphone in the center and adjust the dashed lines to mark the size of the smartphone display.

3.3.2 Custom authentication mechanism

In order to support deployment into most settings the *NAI* framework have a flexible authentication mechanism. The framework comes with a very basic authentication mechanism, but it can be customised to almost any needs, as the developer can have complete control over the UI on the Android client, and the credentials used for authentication, as well as the implementing how the server side accepts these credentials. Note, that authentication is a requirement. At least some minimal authentication mechanism must be executed to verify the identity of the user.

3.3.2.1 Tabletop

The tabletop framework API provides a simple pluggable interface that a developer must use to create his own authentication mechanism. The interface to implement is called `IAuthenticationHandler` and is illustrated in figure 3.14. The interface uses any specialisation of the `ClientCredentials` class. It is up to the framework developer to decide which credentials his application requires from the user.

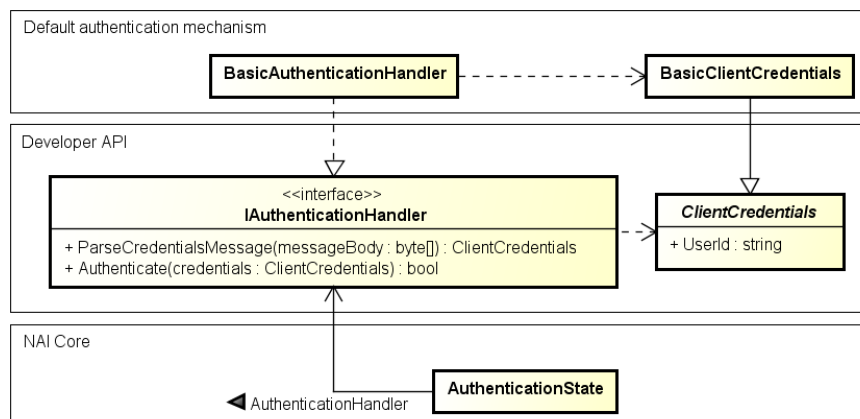


Figure 3.14: Class diagram of the customisable authentication mechanism on the tabletop.

The `IAuthenticationHandler` interface requires the developer to implement two methods:

ParseCredentialsMessage Given a byte array (sent from the smartphone) containing the user credentials, parse and return an instance of the self-defined specialisation of the `ClientCredentials` class.

Authenticate Given an instance of the self-defined specialisation of the `ClientCredentials` class, decide whether the smartphone user is authenticated.

The following code shows the implementation of `IAuthenticationHandler` that the framework uses:

```

public class BasicAuthenticationHandler : IAuthenticationHandler
{
    public ClientCredentials ParseCredentialsMessage(
        byte[] messageBody)
    {
        string userId = Encoding.UTF8.GetString(messageBody);
        return new BasicClientCredentials(userId);
    }

    public bool Authenticate(ClientCredentials credentials)
    {
        // Accept everyone!
        return true;
    }
}

```

The `AuthenticationState` class (see figure 3.14) has a static reference to an `IAuthenticationHandler` instance that is used every time a smartphone tries to authenticate its user. The framework developer has access to replace this `IAuthenticationHandler` instance via the runtime settings of the framework in class `RuntimeSettings` in the `NAI.Properties` namespace. For example:

```

...
NAI.Properties.RuntimeSettings.AuthenticationHandler = new
    MyCustomAuthHandler();
...

```

When a smartphone user is authenticated, the `ClientCredentials` object instance created by the authentication handler is available to the developer when handling *identified events*. The `ClientIdentity` instance provided by the `RoutedIdentifiedEventArgs` has the reference to the specialisation of the `ClientCredentials` object (see section 3.2.6).

3.3.2.2 Smartphone

The smartphone side of the authentication is handled quite differently than the tabletop side. Besides sending the customisable credentials in a byte array to the tabletop, the developer must also be able to customise the user interface, thus allowing the user to submit any kind of credentials. The class diagram in figure 3.15 shows the core classes and the implementation of the default authentication mechanism in the *NAI* framework.

`BaseActivity` is the basic class controlling the application, and it requires a specialisation of the abstract class `AuthenticationHandler` to handle the authentication. By default the application uses `BasicAuthenticationActivityHandler`. This class requests a launch of the `BasicAuthenticationActivity`, a simple activity that allows the user to enter a username. Both of these can be replaced by the developer.

The abstract `AuthenticationHandler` class requires the developer to implement four abstract methods:

requestCredentials When the `BaseActivity` is launched and a secure connection to the tabletop has been established, the user must be prompted for credentials. The most versatile solution is to use an activity, but a simple dialog may also be used. To launch an activity

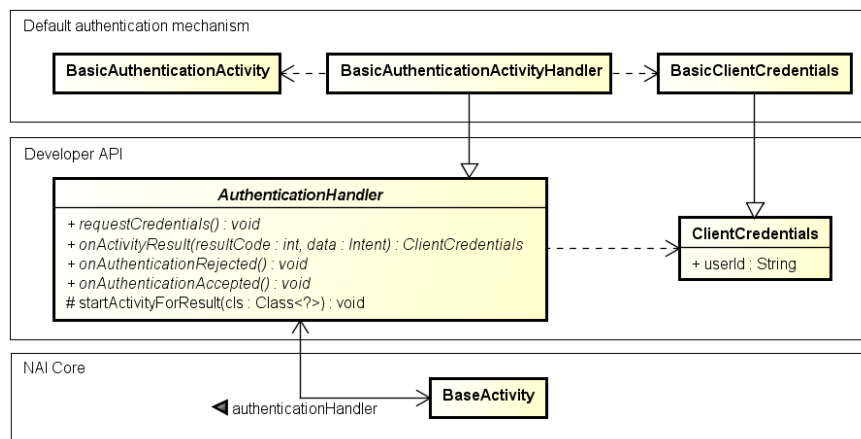


Figure 3.15: Class diagram of the customisable authentication mechanism on the smartphone.

to prompt the user for credentials, the protected method `startActivityForResult` must be used. In the default authentication implementation, the `BasicAuthenticationActivityHandler` launches the `BasicAuthenticationActivity` that allows the user to enter his or her username (a screenshot is provided in figure 3.11).

onActivityResult If `requestCredentials` launched an activity, results from that activity are returned to the handler via `onActivityResult`. An instance of the `ClientCredentials` must be returned to the `BaseActivity`. Notice, that the developer must implement a specialisation of the `ClientCredentials` class. The default implementation `BasicClientCredentials` requires only a user id. `BaseActivity` sends the credentials to the tabletop².

onAuthenticationRejected If the tabletop rejects the credentials, the handler is notified via this callback method, and the developer should take the appropriate actions. In the case of the `BasicAuthenticationActivityHandler` the `BasicAuthenticationActivity` is relaunched.

onAuthenticationAccepted When the tabletop accepts the credentials, this method is called, allowing the developer to use that information. For example, for logging, or to store information for easier authentication the next time.

The implementation of `BasicAuthenticationActivityHandler` is shown below, and is used as the default authentication mechanism in the *NAI* framework:

²Communication to the server goes via a SSL connection and is therefore encrypted. There is no immediate need for further encryption of the credentials, but it is of course optional.

```

public class BasicAuthenticationActivityHandler extends
    AuthenticationHandler
{
    public BasicAuthenticationActivityHandler(BaseActivity context)
    {
        super(context);
    }

    public void requestCredentials()
    {
        startActivityForResult(BasicAuthenticationActivity.class);
    }

    public ClientCredentials onActivityResult(int resultCode, Intent
        data)
    {
        if (resultCode == Activity.RESULT_OK)
        {
            String userId = data.getExtras().getString(
                BasicAuthenticationActivity.INTENT_EXTRA_USER_ID);
            if (userId != null && userId.length() > 0)
            {
                return new BasicClientCredentials(userId);
            }
        }
        // Fall back - close the application.
        context.finish();
        return null;
    }

    public void onAuthenticationAccepted() {}

    public void onAuthenticationRejected()
    {
        Toast.makeText(context, "Authentication rejected", Toast.
            LENGTH_SHORT).show();
        requestCredentials();
    }
}

```

A developer can easily plug in a customised authentication mechanism. Assuming that the entry point of the Android application is the `Main` class, the authentication handler is set in the constructor, as shown below.

```

public class Main extends BaseActivity
{
    public Main()
    {
        setAuthenticationHandler(
            new BasicAuthenticationActivityHandler(this));
    }
}

```

Any new activities must be declared in the Android manifest as normal.

Security analysis

Designing a framework for others to safely build applications upon, require a careful analysis of potential security threats. Especially in the case of the *NAI* framework, where identity and authenticity of users are important. This security analysis explains the security threats to the framework and specify the measures that have been taken in the design of the framework.

A system design overview, figure 4.1, states four areas (marked with numbers) where the system is vulnerable to different attacks. The following sections treat each of these four areas with an explanation of the possible threat(s) as well as providing a solution for mitigation or complete elimination:

4.1 Area 1: Non-anonymous user interaction

The basic idea behind the *NAI* framework is to provide a way to link an action to the identity of its originator. This predominant feature is called *non-anonymous user interaction*. All of the security related requirements of the *NAI* framework discussed in this security analysis originate from this feature.

Non-anonymous user interaction is the same as *identified* user interaction. The word *identified* is the adjective version of the noun *identification*, which some may confuse with *authentication*. Even though the

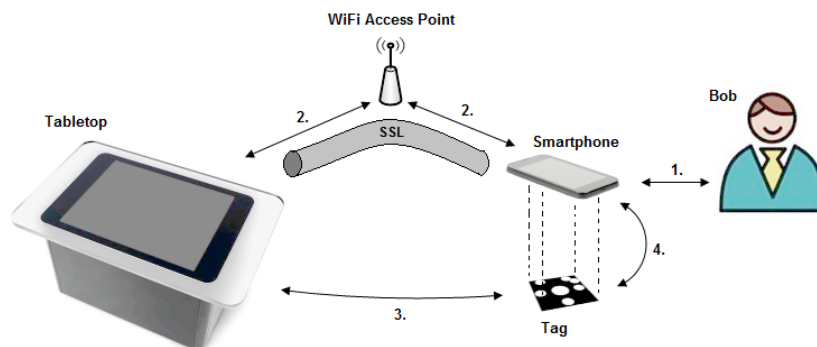


Figure 4.1: A security threats overview of the *NAI* framework.

two words are closely related, they are fundamentally different. Stephen Downes [5] states the difference very clearly with his definition of the two words, quoting:

Identification is the act of claiming an identity, where an identity is a set of one or more signs signifying a distinct entity.

Authentication is the act of verifying that identity, where verification consists in establishing, to the satisfaction of the verifier, that the sign signifies the entity.

If these definitions are translated to the context of the *NAI* framework, with Bob as an example, then Bob is a distinct entity. The signs to signify him is the smartphone he uses and the credentials he provides during the initial user authentication. Every time Bob uses his smartphone to create an *identified event* in the *NAI* framework, his identity is associated with the event. The problem is, that when such an event occurs, the smartphone alone signifies Bob as a distinct entity. Another person could impersonate Bob, by using Bob's smartphone to create an *identified event*, because the framework then would associate Bob's identity with the event. This is because the individual *identified events* are not authenticated. The argument for not doing this narrows down to a discussion about the tradeoff between security and usability. It would be annoying and obstructive for the users interacting with the tabletop if every *identified event* required a password to be entered, but it would on the other hand be very secure. So the fact that events in the *NAI* framework are only *identified*, and not *authenticated*, introduces a security threat regarding impersonation. It is up to the application developers using the *NAI* framework to decide the severity of this threat in their application context.

The *NAI* framework authenticates users only one time during the startup procedure of the smartphone application. In here lies the assumption, that once the tabletop has established that Bob is the user of his smartphone, then Bob is the only one using his smartphone for interaction with the tabletop application throughout the application session. This is equivalent to other computer systems, where users are required to log in on startup, and log out when they are finished.

Authentication is, as stated above: “...*verification to the satisfaction of the verifier...*”, which basically means that how authentication is done depends on the application context. Possible applications for the *NAI* framework include the emergency room control unit for dispatching emergency units after a flooding [21], and an order and payment system for a restaurant, where the guests order and pay their meal without involving a waiter (see chapter 6). The level of authentication required in those two applications are different, because of the consequences of the actions carried out by users. So the real *verifier* in the authentication process must be the application running on top of the *NAI* framework. At the application level, it should be possible to specify a satisfactory level of user authentication. The *NAI* framework does not *use* the iden-

tity of users – it only *provides* the identity. So the framework does not care *how* Bob is authenticated – just that he *is* authenticated.

4.2 Area 2: Tabletop – smartphone communication

Communication between smartphones and the tabletop computer is done over a Wi-Fi connection as illustrated by the links (nr. 2) in the system overview figure. The messages being transmitted back and fourth contain sensitive information, like for instance a touch event being transmitted from a smartphone to the tabletop or an image stream containing secret information being sent from the tabletop to a smartphone. If the communication is unencrypted, an eavesdropper may intercept secret information being transmitted and possibly execute a man-in-the-middle attack, for instance by altering a touch event message sent from a smartphone to the tabletop. This threat is challenging the underlying security assumptions about identified interaction. The way to eliminate the threat is to ensure that the communication channel provides both confidentiality and integrity. Using a Secure Socket Layer (SSL) connection provides a secure tunnel for the tabletop and smartphone to communicate through, which ensures confidentiality and integrity, because communication is encrypted end-to-end. This is illustrated by the tunnel connection between the tabletop and smartphone in the system design figure above. An SSL connection requires the server, in this case the tabletop computer, to be authenticated using a X509 certificate.

4.3 Area 3: Tag recognition

The tags supported by the Microsoft Surface platform allow movement tracking of each smartphone on the tabletop surface by sticking a tag on the back of each phone. The tags are optically recognised by the infra red cameras underneath the tabletop surface. In some sense this may be categorised as a (primitive) communication channel (nr. 3 in the figure). While Microsoft has designed the *identity tag* series to potentially allow every smartphone in the world to have its own unique tag, there is a problem in using them for identification. The tags are supposed to be unique, but they are not encrypted or secret, and may therefore be duplicated by an adversary. If for example Bob had his tag from his smartphone duplicated by an adversary, it could be used to trick the tabletop computer to believe that Bob's smartphone was at a different position on the tabletop surface.

The solution to mitigate this threat is to only allow the tabletop computer to recognise one of each unique tag at a time. Furthermore the tabletop computer may not associate any sensitive information with a given tag unless the tag has been, and still are, paired with a smartphone. This solution still allows an adversary to duplicate Bob's tag and trick the tabletop computer, but only if Bob's smartphone is connected

and not laying on the tabletop surface. The adversary would still require Bob's smartphone to do any serious damage like impersonation, so this attack is merely to be considered as a mild version of denial-of-service. Another aspect of this threat is that both Bob's smartphone, and hence also Bob, and the adversary must be at or close to the tabletop computer at the same time when the attack is carried out. This attack type is therefore to be considered impracticable.

4.4 Area 4: Pairing tag with smartphone

Once a smartphone has a tag stuck on its back, it unifies the two communication channels mentioned in the previous two sections. The challenge here is, how to securely tell the tabletop computer which tag belongs to which smartphone. This act is called *pairing* and is illustrated by the connection nr. 4 in the figure above. A pairing can only be done if both communication channels have been established. That is, the smartphone (with tag) is positioned on the tabletop surface and the SSL handshake has been completed successfully. The tabletop computer performs a pairing by a simple challenge-response scheme. The challenge is a unique random generated PIN code which is associated with a recognised tag and displayed on the tabletop display next to the tag. The response comes via the SSL communication channel from the smartphone, where the user enters the PIN code.

This pairing procedure is potentially vulnerable in three areas:

1. If the randomly generated PIN code is not unique, for example if two persons are pairing their smartphones with their tags at the same time and tabletop computer generates the same PIN code for both of them. This could result in the two smartphones being paired with the other person's tag. This vulnerability is effectively avoided if the tabletop computer does not allow the same PIN code to be used in multiple simultaneous pairings.
2. An adversary may deliberately misuse another person's PIN code. If for example an adversary uses the PIN code associated with Bob's tag to pair with his own smartphone, Bob would then not be able to complete his pairing and instead he would control what the segment of the tabletop display is sent to the adversary's smartphone. This vulnerability is difficult to avoid, but it is also not very severe. The adversary would not be able to impersonate or disclose any secret information from Bob. The attack is merely a kind of denial-of-service.
3. As mentioned before, tags can be duplicated, so an adversary may choose to duplicate the tag Bob's has stuck on his smartphone. If the adversary pairs his smartphone with the tabletop before Bob arrives, Bob would be unable to pair his smartphone. This attack is closely related to nr. 2 and is also categorised as denial-of-service.

4.5 Summary

There is a separation of concern in regards to the security aspects surrounding the *NAI* framework. Both the framework implementation, the framework developer and the end user are involved in securing the *NAI* framework. Most important is the framework implementation, because it provides a secure environment for the framework developer to build applications for end users. Using the CIA properties of computer security [2] as a basis, the framework implementation features:

Confidentiality Using SSL to secure the communication between the smartphone and tabletop effectively ensures that no information is disclosed to unauthorised individuals.

Integrity Same as for confidentiality.

Availability There are some minor threats regarding denial-of-service possibilities surrounding the mandatory tags on the back of the smartphones. However, the attacks can only be carried out if the attacker and the victim are at the tabletop computer at the same time which make the attacks impracticable.

The framework developer is responsible for the authentication of end users. This is because the framework cannot decide the level of authentication required in the developer's application context. Authentication is scalable and should hence be customisable by the framework developer.

The interaction design in the *NAI* framework requires end users to use smartphones to do *identified user interaction*. And furthermore, the end users are assumed not to hand over their smartphones while interacting with the tabletop. This makes the end users vulnerable to *social engineering*, which is a non-technical threat. Some examples of social engineering attacks include users who are persuaded to leave or forget their smartphones at the tabletop, or users who are persuaded to disclose information that was otherwise only meant to be shown to them. It is difficult to quantify the severity of this threat, because it depends on the application context. However, the chance of successfully executing a social engineering attack is to be considered small, because it is like persuading a person to disclose the PIN code of his cash card, or persuade him to forget the cash card at the ATM.

Implementation

This chapter provides details on the *NAI* framework implementation on the two platforms. The overview of the framework in figure 5.1 shows the *NAI* Core which is the main focus here. The implementation on the two platforms share some architectural similarities which are described first in section 5.1 and 5.2, before implementation details for the two platforms are described in the rest of the chapter.

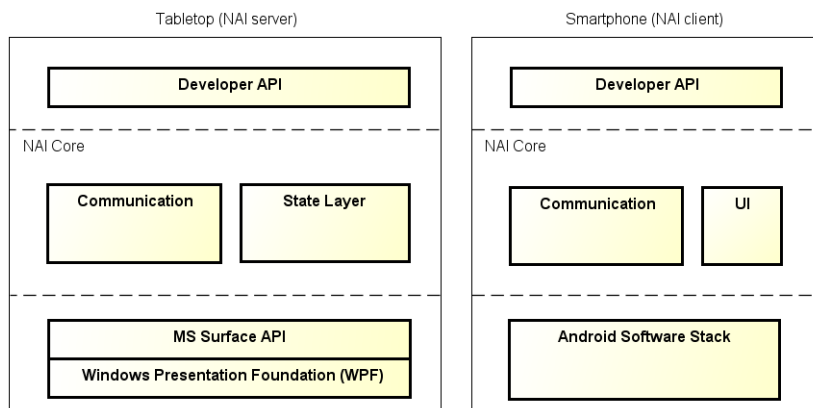


Figure 5.1: An overview of the *NAI* framework implementation.

5.1 States

The time period where a smartphone (*NAI* client) maintains a secure socket connection for communication with the tabletop (*NAI* server) is defined as a *client session*. During a session, the *NAI* client is required to pass through different states, which are *authentication*, *pairing* and possibly *calibration* before reaching the *streaming* state where the user can use the smartphone to perform identified interaction with the tabletop. In other words, a client session is a simple state machine. The state diagram in figure 5.2 depicts the states, with the two normal paths marked with solid arrows. The dashed arrows cover scenarios where something goes wrong, like a sudden lost socket connection.

To get to the streaming state, the *NAI* client must at least pass

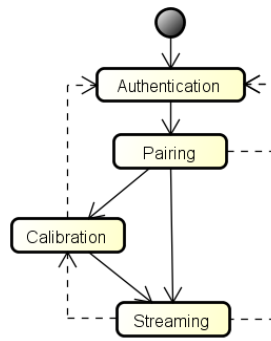


Figure 5.2: States in a *client session*

through the authentication and pairing state. The calibration state is skipped if the smartphone has already been calibrated previously. However, the first time the smartphone is used with the tabletop, calibration is mandatory.

The *NAI* server keeps track of the state of each client session. The implementation of each state on the tabletop is discussed further below in section 5.3. The *NAI* client does not have the same strict state machine design, but is instead ready to react to any input coming from either the user or the tabletop. The implementation details of the *NAI* client is discussed further below in section 5.4.

5.2 Communication

The *NAI* client and *NAI* server use a custom communication protocol, which is described here. They communicate over a normal TCP socket connection. A TCP connection is not secure, because it does not provide confidentiality and integrity, and it is at the bottom line made for sending and receiving bytes, which is rather primitive. Both of these inconveniences are dealt with in the following.

5.2.1 Establishing and securing the TCP connection

The *NAI* client uses the UDP protocol to multicast a simple message in order to automatically discover any *NAI* server residing on the same wireless network. When the *NAI* server intercepts a discovery message it responds via UDP with its IP address to the *NAI* client which can then establish a normal TCP socket connection. The socket connection is wrapped in a secure SSL tunnel. The *NAI* server requires a certificate installed in the local key store to perform the SSL handshake and to be authenticated to the *NAI* client. The SSL protocol supports client authentication, but the *NAI* framework does not require it because of its design. Authenticating a smartphone as device is not important, but authenticating its user is important. The framework provides its own customisable user authentication mechanism, see section 3.3.2.

Once the connection is made secure it is time to deal with the primitive nature of the TCP socket capabilities. The standardised OSI (Open

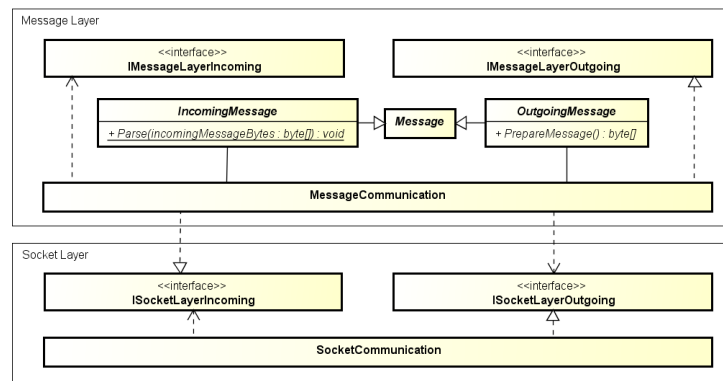


Figure 5.3: The communication layers, as implemented in the *NAI* server.

Systems Interconnect) model for communication and protocol design¹ is the basis of the TCP/IP model² used in most modern network setups including the internet. By using the principles of the model, two communication layers have been designed and put on top of the primitive socket. The two layers exist on both platforms, but have some small differences. The implementation of the layers on the *NAI* server is illustrated in figure 5.3. The implementation on the *NAI* client is illustrated and described further in section 5.4.2.

5.2.2 Socket layer

The secure socket is wrapped by the socket layer at the bottom of the layered architecture. The socket layer maintains two threads, one sending and one receiving. For the sending thread an interface is provided to send simple arrays of bytes. For the receiving thread an interface for handling incoming byte array messages is required. Before the socket layer sends a byte array through the socket, it prepends a four byte header to the array which is a 32 bit integer value indicating the length of the byte array. When the byte array is received at the other end, the header is read first, in order to know how many bytes to read from the socket stream.

5.2.3 Message layer

The message layer provides a further abstraction of the socket layer, where the byte arrays of the socket layer are translated to high level *NAI* framework messages. The messages are implemented in classes with a distinction between `IncomingMessage` and `OutgoingMessage`. A message may contain an arbitrary amount of data. The message layer provides an interface to send an `OutgoingMessage` to the other party, and it requires another interface to pass any `IncomingMessage` to. This design requires that an `OutgoingMessage` on one platform must have a `IncomingMessage` counterpart on the other platform.

¹http://en.wikipedia.org/wiki/OSI_model (as of 30 June 2011)

²<http://tools.ietf.org/html/rfc1122> (as of 30 June 2011)

When an `OutgoingMessage` is prepared for the socket layer, it is transformed to a byte array with a one byte header prefix indicating the type of message. The receiving part uses the header to correctly parse the incoming byte array to the correct specialisation of `IncomingMessage`.

5.3 Tabletop

Before the *NAI* server implementation is explained, there is first a section with a small introduction to specific areas of the Microsoft Surface Platform which are important for the *NAI* server implementation.

5.3.1 Integration with MS Surface API

The Microsoft Surface (MSS) is a complete platform from hardware to software designed by Microsoft. Version 1 of the platform, which the *NAI* framework is built upon, is from 2007 and features a 30" inch 1024*768 pixels rear projected display enclosed in a big box. Inside the box rests the projector for the display as well as 5 cameras for input detection.

The SDK for MSS includes two APIs for application development, one for high-end graphics demanding applications based on Microsoft XNA, and one for more normal Windows-like applications based on Windows Presentation Foundation (WPF). The *NAI* framework uses the WPF based API.

5.3.1.1 Contacts

The MSS platform handles up to 52 touch inputs at a time, which may come from various different input sources. Any kind of physical interaction on the tabletop screen is called a `Contact`. The underlying vision system is based on infrared cameras and is able to detect any sized objects placed on the screen, as long as they reflect infra red light. Recognised objects are analysed by the tabletop computer and the type of `Contact` is derived. There are three different `Contact` types:

Finger Any human finger.

Tag The MSS recognises certain tags, which can be used to identify and distinguish specific objects. Tags play an essential role in the design of the *NAI* framework and are discussed further below.

Blob Any object not recognised as a finger or tag.

A `Contact` instance also provides geometrical information such as size, orientation and position of the recognised object.

The `Contact` type is used when raising events whenever something interacts with the tabletop screen. Typical interaction events are when an object is being placed, moved or lifted from the screen.

Conceptually, an event triggered by input from a finger is called a *touch* event. Since a smartphone is only able to detect touches from fingers, the *identified touch events* in the *NAI* framework are called "Touch" and not "Contact".

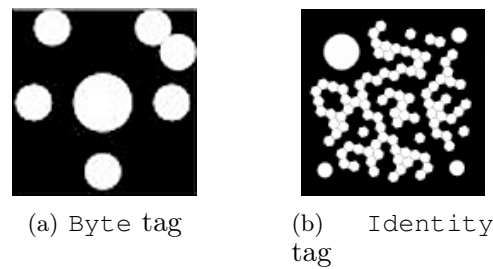


Figure 5.4: Example of the two tag types that Microsoft Surface detects.

5.3.1.2 Tagged objects

The MSS vision system is able to detect two kinds of tags, `Byte` and `Identity` tags. The difference between the two types are the number of distinct tags available. For `Byte` tags the total number of unique tags is 256 (hence the name `Byte`), and for `Identity` tags the number is 2^{128} , represented as two 64 bit numbers. The two numbers are meant to represent a series identifier and a value identifier. Both tag types are designed to allow the tabletop computer to determine the orientation of the tag when placed on the surface. Examples of the two tag types are depicted in figure 5.4.

Internally, a tag is simply a `Contact` and may be dealt with by the normal `Contact` events raised when the tag is detected. However, a more sophisticated way of working with tagged objects in a MSS application is to use the `TagVisualization` control. It is a special MSS platform control which uses the primitive `Contact` events raised by tags in a clever way. The control presents a view underneath a tag at a fixed distance and angle when it is recognized by the tabletop computer. Furthermore, the `TagVisualization` control has the special property, that the presented view automatically follows and rotates according to the movement of the tag it visualises. The view is initially empty, because it is meant to be populated with visible content by the application developer. In order to use the `TagVisualization` control, a container is required where the individual `TagVisualization` instances can move around. This required container is the `TagVisualizer` control.

5.3.2 The *NAI* server architecture

The state machine in figure 5.2 is the foundation of the architecture of the state layer in the *NAI* server. It is implemented as the acknowledged state design pattern³. An overview of the architecture with the state pattern is depicted in figure 5.5. The state machine controller for a single *NAI* client is the class `ClientSession`, which has access to the communication interface of the Message layer and thereby provides a way for the different states to send messages to the *NAI* client. `ClientSession` also receives all incoming messages sent from the *NAI* client to the *NAI*

³http://en.wikipedia.org/wiki/State_pattern (as of 30 June 2011)

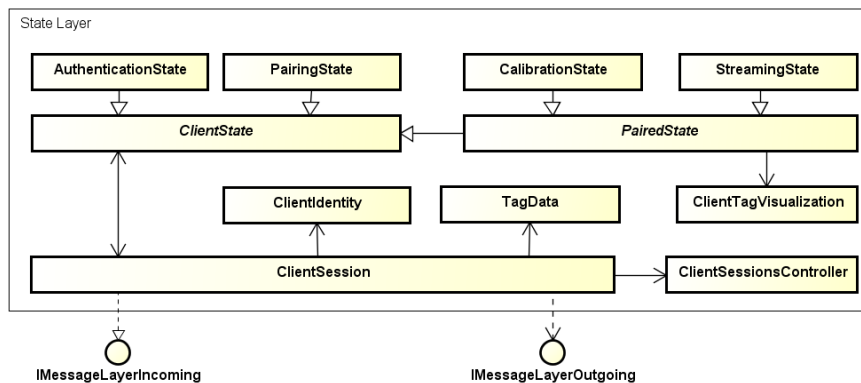


Figure 5.5: Class overview of the state layer in the *NAI* server

server, but these are just passed on to the current state of the client session, because the different messages have different meaning depending on the state of the client session and hence they should be handled locally at state level. Besides communication, the `ClientSession` also keeps a reference to information about the client for the different states to use. This information is kept in the two classes `ClientIdentity` and `TagData`. All instances of `ClientSession` are registered in the global singleton class `ClientSessionsController`.

The two paired states, `CalibrationState` and `StreamingState` has access to `ClientTagVisualization` which is a view-class required for the visualisation presented underneath the smartphone positioned on the tabletop surface. The class is a specialisation of the Surface API class `TagVisualization`, which is created when a tag is recognised. The look and behavior of `ClientTagVisualization` changes according to the current state of the client session.

The following sections describe some behavior and implementation details of the four different states in the state machine:

5.3.3 Authentication state

The purpose of the authentication state is to authenticate the smartphone user to the application and the *NAI* framework. As explained in the security analysis, the framework does not care *how* the authentication takes place, just that it *is* taking place. The authentication mechanism is hence pluggable and defined by the framework developer as described in section 3.3.2.

The authentication takes place after the secure SSL socket connection has been setup, so the framework provides a secure environment without security threats to the authentication. The *NAI* server has little responsibility in the authentication procedure other than sending and receiving messages to and from the *NAI* client. When some credentials are received, the *NAI* server validates them by calling the `Authenticate` method of the pluggable `AuthenticationHandler` (see figure 3.14). If the authentication is successful, the *NAI* server moves to the pairing state.

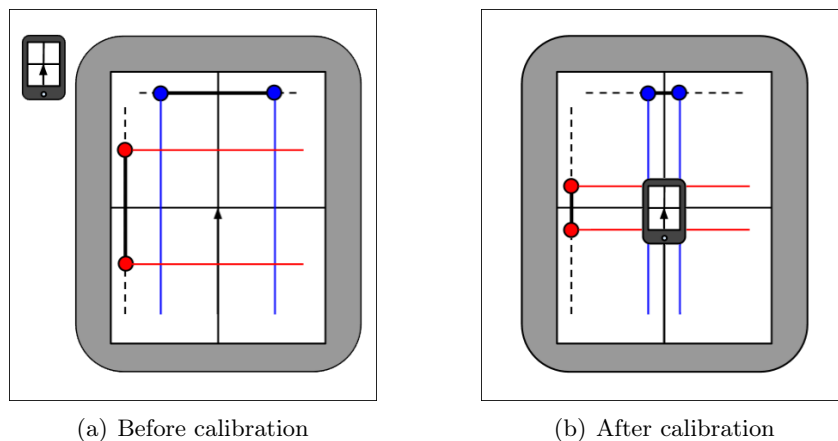


Figure 5.6: How to calibrate the smartphone

5.3.4 Pairing state

The purpose of the pairing state is to associate a smartphone with its tag. Because tags are easily duplicated, the *NAI* server cannot assume any relationship between a specific tag and a smartphone and hence pairing is mandatory for every client session.

The mechanism used is described in section 4.4. When a tag is recognised and the PIN code has been generated it is presented via the `Client-TagVisualization` class on both sides of the tag (see figure 3.12). This required user involvement is meant to increase security.

When a valid PIN code is received from a smartphone, the `Client-TagVisualization` instance representing the tag is put in the `Client-Session` instance where the PIN code has been received. This completes the pairing.

5.3.5 Calibration state

Initially, the *NAI* server does not know the screen size of the phone nor does it know where the screen is positioned on the phone. These are two key requirements in order for the *NAI* server to stream the right segment of the tabletop display to be shown on the smartphone's display. The act of providing these parameters to the *NAI* server is called *calibration*.

When calibration starts, the view class `ClientTagVisualization` for the tag changes its content to a calibration control the smartphone user can manipulate with normal finger touch input on the tabletop surface. The calibration control is identical to the sketch in figure 5.6. The only thing the tabletop for sure is able to detect is the tag stuck on the back of the smartphone, so it is used as an origin for the calibration. It works as follows:

1. The starting point is figure 5.6(a), where the smartphone displays a static image containing a center aligned cross with an arrow pointing at the center from one direction. The tabletop displays the

calibration control which also contains a black cross and an arrow. On the top and left side are two special range sliders with lines forming a rectangle. The black cross will always be in the center of the rectangle, no matter how the range sliders are adjusted.

2. The smartphone owner positions his smartphone on the tabletop surface such that the two crosses align. The arrows also need to point from the same direction on both devices. With this step completed, the tabletop computer now knows the center position of the smartphone screen relative to the tag position and orientation. The arrow ensures that the smartphone and tabletop agree on what is up and down on the smartphone.
3. Next step for the user is to adjust the red and blue range sliders so they align with the edge of the smartphone screen. This alignment tells the tabletop exactly how big the smartphone screen is.
4. With the sliders adjusted, the calibration is now complete as illustrated in figure 5.6(b). The calibration is accepted when the smartphone owner taps his smartphone screen and accepts the calibration. By putting the calibration acceptance on the smartphone minimises the risk of someone accidentally interfering with the calibration.

The five calibration parameters recorded from a calibration are saved to a file, because the smartphone user then only have to calibrate during his first session. The data is stored in a simple XML file with the tag value as a key, because the calibration is bound to the device, and the device is bound to its tag. Although the security analysis (chapter 4) states that it is not safe to associate any information with a tag, the calibration data is considered as low value data, because the worst thing that can happen is an adversary changing the calibration. The user can always initiate a new calibration.

5.3.6 Streaming state

Once authentication, pairing and possibly calibration are completed, the client session moves to the streaming state, where the *NAI* server can start sending the picture stream of the segment of the tabletop display covered by the smartphone. When the picture streaming has started, the *NAI* server is ready to raise *identified events* caused by moving the smartphone or by touching its display.

The view associated to the tag on the smartphone, the `ClientTagVisualization`, changes to a transparent rectangle which is sized and positioned based on the calibration to exactly mark the display of the smartphone. The purpose of the rectangle is threefold (covered in details below):

1. Mark the area to be screen captured for the picture stream to the smartphone.

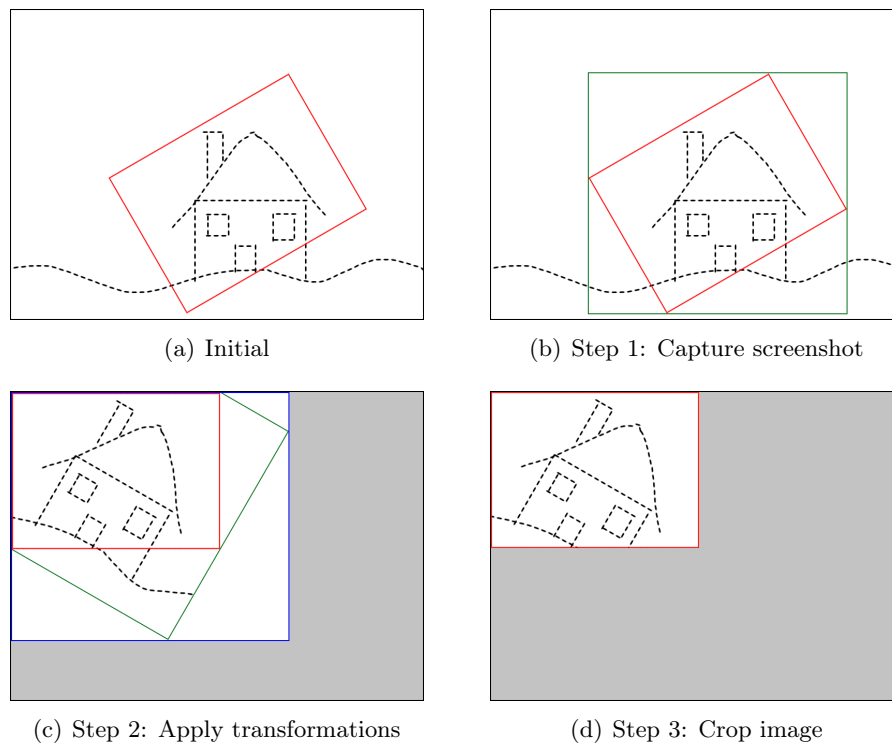


Figure 5.7: Steps in screenshot algorithm

2. Mark the area for hit testing for raising *identified hover events*.
3. Container for `PersonalizedView`.

5.3.6.1 Streaming pictures to smartphone

The transparent rectangle in the view class continuously marks the area to be screen captured and send to the smartphone. The Windows operating system and the .NET framework only provide a mechanism for capturing a rectangular screenshot without rotation, so an algorithm is needed to capture a non-rotated bounding box screenshot, rotate and crop it to perfectly fit area marked by the transparent rectangle and hence the area underneath the smartphone's display.

The algorithm has three steps, which are easiest explained by a concrete example of how one single screenshot is captured and manipulated. The example is illustrated in figure 5.7 and explained below:

Initial The smartphone is positioned on the tabletop surface and the smartphone display covers the rectangle marked with the red square (figure 5.7(a)). So the red square is marking the otherwise invisible transparent rectangle in the `ClientTagVisualization` class. Given the smartphone's tag and the its calibration, the tabletop computer is able to tell the exact location and rotation of the red square.

1. Screenshot The non-rotated screenshot mechanism available in Windows requires an initial larger screenshot to be captured. The captured screenshot is the bounding box marked with the green square in figure 5.7(b), which is calculated using the WPF class `VisualTreeHelper`. This helper class provides different methods for transforming coordinates of local coordinate systems of visual elements to coordinate systems of visual elements elsewhere in the visual tree using matrix transformations from linear algebra. In this case, the transformation needed is from the `ClientTagVisualization` (the red square) to the root of the visual tree which is the `SurfaceWindow` enclosing the application. Fullscreen windows are used by default by MSS applications and hence their coordinate systems are the same as used in the native screenshot mechanism in Windows OS.

2. Transformation When the screenshot marked by the green square has been captured, it is rotated to become parallel with the the screen. The `VisualTreeHelper` class provides the matrix transformation needed to rotate the captured screenshot the appropriate angle. WPF allows multiple transformation matrices to be concatenated into one matrix by multiplying them. For the screenshot this means that three transformations are applied in one step. First, a translation transformation to put the center of the screenshot in the origin of the coordinate system and hence make it the pivot point for the rotation. The second transformation is the rotation. The third transformation is a translation, which moves the origin to the top left corner of the red square and thereby prepares the screenshot for final cropping. After applying the three transformations the screenshot is as marked by the blue square in figure 5.7(c).

3. Cropping The last step is to crop the image to cut away the excess screenshot parts between the red and blue square. The final picture is in depicted in figure 5.7(d) and is ready to be send. The picture is JPEG encoded before it is sent to the smartphone.

The streaming process is computation intensive, and hence it runs in its own thread. This is all encapsulated in the class `StreamingProcessor` which again is controlled by the state class `StreamingState`. To get the glass-like effect of the smartphone screen, the pictures streamed to the smartphone must be produced at a high rate, about 15 frames per second (FPS). In comparison, television operates with a frame rate around 25 FPS. A captured JPEG encoded screenshot for a 3.7 inch smartphone display is about 12 kB. If three smartphones are connected simultaneously to the tabletop computer running at 25 FPS, the network bandwidth requirement is 7,03 Mbit/s which is low compared to the 54 Mbit/s (theoretical) capacity of the IEEE 802.11g protocol the MSS tabletop supports. The default frame rate in the *NAI* framework is 15 FPS, but it can be changed in the config file on the tabletop. However, increasing the streaming frame rate also increases the CPU load on the smartphone, which drains the smartphone battery faster.

5.3.6.2 Raising events

The second purpose of the streaming state is to raise *identified events* when they are applicable. The set of events are listed in the API specification (section 3.2.6). By convention in WPF, input related events are raised in pairs with a *tunneling* event followed by a *bubbling* event. The *NAI* framework follows this convention and has two versions of each the *identified touch* and *hover events*. The *identified person events* are only raised in a bubbling version.

The three classes of identified events are raised in different ways:

Touch events are triggered by the smartphone user when the display of the smartphone is touched. The smartphone sends special touch event messages to the *NAI* server with the relative coordinate of the touch event on the smartphone. The `VisualTreeHelper` class in WPF helps to first transform the relative local coordinate to the global coordinate system of the tabletop computer display, and second to perform hit testing using the global coordinate. The first visual element to get hit by the hit testing gets the appropriate *identified touch event* pair raised.

Hover events are triggered when the smartphone is moved. The tabletop continuously tracks the movement of all tags positioned on its display, and every time a tag moves, a hit test is performed on the area covered by the transparent rectangle in the view to raise the appropriate *identified hovering events* on the visual elements found in the hit test.

Person events are raised when a smartphone is successfully paired, or a smartphone drops its connection to the tabletop. The events are bubbling and raised on the mandatory outer control `IdentifiedInteractionArea`.

5.3.6.3 PersonalizedView

The `ClientIdentity` instance provided by the `RoutedIdentifiedEventArgs` class which is passed along when an *identified event* is raised, provides an interface to add or remove a `PersonalizedView`. A `PersonalizedView` can be any specialisation of the WPF class `UIElement` and is added as a child element of the transparent rectangle in the `ClientTagVisualization` class in the element tree. The `PersonalizedView` is then displayed on the tabletop display exactly where the screenshots are captured and streamed to the smartphone and hence the smartphone gets a personal view.

5.3.7 Identified controls development

The *identified events* can be used to create new controls which use identified input. They can also be used to extend existing controls in the

MSS API or WPF, which for example is the case with the `IdentifiedSurfaceButton` control in the *NAI* framework. When naming new *identified controls*, it is recommended to use the same naming convention as used for naming the four current *NAI* framework specific controls. That is, put “Identified” as a prefix of the name.

There are some considerations and pitfalls to be aware of when extending existing controls or creating new ones that use *identified events*:

5.3.7.1 Handling normal touch events

The *identified touch event* set does not replace the native `Contact` event set of the MSS API. So if an existing control that uses normal touch input, is extended, it is important to consider what should happen with the normal touch events.

Example The `IdentifiedSurfaceButton` (ISB) control extends the functionality of the `SurfaceButton` control. So the developer must choose whether or not normal touch clicks on the button are accepted. The ISB has a bool property `BlockClickEvent` that, if set to `True`, ensures that normal touch clicks are not allowed. The way it is implemented is to catch the tunnel `PreviewContactDown` and mark it as handled before it reaches the ISB. The event handler to catch the tunnel event is in the mandatory `IdentifiedInteractionArea`, because it is always the case that the `IdentifiedInteractionArea` precedes all other elements, so it is safe to catch the event in that class. The following XAML code snippet shows an example of an ISB that does not accept clicks by normal touch:

```
...
<id:IdentifiedInteractionArea>
    ...
    <id:IdentifiedSurfaceButton BlockClickEvent="True">
        Click Me!
    </id:IdentifiedSurfaceButton>
    ...
</id:IdentifiedInteractionArea>
...
```

The `IdentifiedInteractionArea` has a public method that can be used to register elements where `PreviewContactDown` events should be blocked. The ISB invokes this method in its constructor:

```
public class IdentifiedSurfaceButton : SurfaceButton
{
    public IdentifiedSurfaceButton()
    {
        if (BlockClickEvent)
            IdentifiedInteractionArea.KillEventsForIdentifiedUIElement(
                this);
        ...
    }
    ...
}
```

5.3.7.2 Moving objects

Controls like the `ScatterView` and `LibraryBar` in the MSS API are examples of container controls for moving objects. This type of controls is a problem for *identified hover events*, because those events are only raised when the smartphone is moved. So it is possible to move an element, e.g. a `ScatterViewItem` into the smartphone display area without having *identified hover events* being raised on the elements contained in the `ScatterViewItem`.

Example This problem has been addressed in the `IdentifiedViewport` control in the *NAI* framework, where it is a key property that information can be hidden from certain users.

The `IdentifiedViewport` control uses the WPF control `GeometryCombineMode` to create a custom shape that covers some area, but at the same time contains holes where smartphones are allowed to see through. This is done with the `GeometryCombineMode` property set to “Exclude” and thereby having the set of rectangles underneath the smartphone displays hovering over the control excluded from the base geometry shape. If one of the smartphones are moved, the control takes care of updating the exclude-set via the *identified hover events*. If the entire `IdentifiedViewport` control is moved, for example when being contained inside a `ScatterViewItem`, it requires a forced update by invoking the method `Moved` from the outside. This method does, among other things, invoke the public `UIHelper` class to get a list of current smartphones hovering over the control which are used to analyse which *identified hover events* to raise. The `UIHelper` class is useful when developing new controls that use *identified hover events*.

5.4 Smartphone

The *NAI* framework client is implemented on the Android platform. A high level view of the core components can be seen in figure 5.8. The figure has been cleaned for details, showing only the most important classes and packages. Starting from below the most important classes from the Android API that are needed for the application to function are described in section 5.4.1. The `UI` package contains components related to the user interface. That is all the specialisations of the `Activity` class. The authentication package contains classes related to user authentication, which have been covered in section 3.3.2. Finally, the communication component covers all the communication, and its implementation is described in section 5.4.2.

5.4.1 Basic Android building blocks

An Android application can be made of a number of basic components, that when combined allows an application to operate as intended. The components are represented as classes and they are very different in nature, and usage. To use them, you need to extend the class and implement

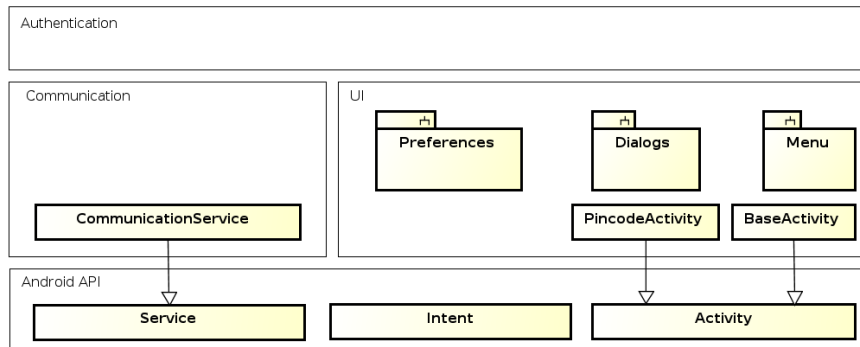


Figure 5.8: Overview of central components for implementation for the Android client. The diagram has been simplified in order to avoid clutter.

the application specific logic⁴. A list of the ones used in the framework can be seen below.

Activity An `Activity`⁵ represents a single screen with a user interface, and even simple applications typically consists of multiple activities. A gallery may have one activity showing a list of pictures, but showing a single picture in a detail view could start another activity. While the user experience will be coherent, the application itself will be firing separate activities during operation. In figure 5.8, the most important classes that extends the `Activity` can be seen. `BaseActivity` being the most important as it is the entry point of the *NAI* client.

When an `Activity` starts, it is placed on a stack. Android only allows one `Activity` to be running at a time, pausing or killing the rest. When an activity ends, the next in the stack is activated. This forces the application to handle this volatile environment by saving state information when paused. In the *NAI* framework, the *NAI* client establishes a connection to a *NAI* server when started, and that connection must be kept alive, if other activities is placed on top of the stack. For example, when the phone rings. Since an `Activity` will be paused, and hence the connection is lost, the `Service` component must be used to control the connection.

Service Unlike an `Activity` a `Service`⁶ does not provide a user interface. A `Service` is used for long-running operation like playing music while doing something else, or keeping a network connection alive. Activities that need access to a `Service` must bind to it, and start it, if it isn't started yet. As can be seen from figure 5.8, the `CommunicationService` extends the `Service` class in order to

⁴For a quick overview of basic components in Android look at <http://developer.android.com/guide/topics/fundamentals.html> (as of 2 July 2011)

⁵Activity: <http://developer.android.com/guide/topics/fundamentals/activities.html> (as of 2. July 2011)

⁶Service: <http://developer.android.com/guide/topics/fundamentals/services.html> (as of 2 July 2011)

keep the connection to the *NAI* server alive.

There is no guarantee that Android will not kill the service. If resources are low, and the OS needs the resources, the service can be killed as well. Contrary to an `Activity`, Android will try to keep services running as long as possible.

Intent Because an `Activity` is only running when placed at the top of the view stack, and the operating system is in charge of what should be active, there can't be a direct reference to other activities. In stead, communication between basic Android components like activities and services are wrapped in an `Intent` class. This defines what should be started, and it contains the resulting data if an activity must supply a response to the calling activity. Figure 5.8 includes the `Intent` class in the diagram but the association lines have been omitted to avoid cluttering up the diagram.

An `Activity` does not represent the *view* in a model-view-controller pattern by itself. The Android environment allows developers to define the visual components and textual content in separate XML-documents. In general, it is considered best practice to separate the concerns, but Android allow exceptions. Views can be created and manipulated dynamically directly within the activity itself. The *NAI* client have separated the view into xml-files, leaving the activities to handling events and transitions between activities. Activities can be found in the UI package because an activity controls and defines the user interface.

5.4.2 Communication layers

The *NAI* client implements the shared communication model as described in section 5.2, with its distinct separation of socket layer and the message layer. But the client implementation has some differences. First, it is obvious that what is incoming messages on the server is outgoing messages on the client, and vice versa, but more importantly, the client must be able to support interactions with the user. The user can take the initiative to setup a connection in two different ways, and the user must be prompted to act when the automatic lookup service fails to find the *NAI* server, or the connection is lost.

Figure 5.9 shows the client version of the communication layers. Because of the extended responsibility of handling connection events and user issued commands as well as handling messages, a more general naming has been chosen.

The socket layer is centered around the `CommunicationHandler` class. It uses the `UdpComm` class to automatically find the *NAI* server, and the `TcpComm` class to setup and handle the TCP connection to the server. As described in section 5.2 the *NAI* client initially tries to find the *NAI* server by broadcasting a special message via UDP. If not an identical UDP response is received from the server within a short time, the attempt is failed, and the failure event is being sent upward through the

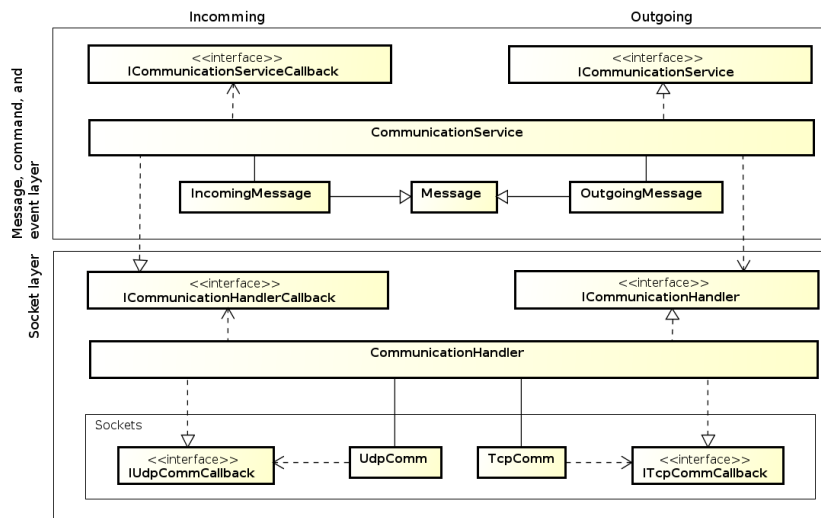


Figure 5.9: Class overview of the communication components of the Android client. It is an augmentation of the model seen in figure 5.3

layers. Eventually, the user will be prompted to take action. If a response from the server is received, then a TCP connection can be established via the `TcpComm` class, and the success will equally be reported up through the layers. It is also possible to connect to the *NAI* server directly without the automatic lookup, but it must be issued by the user. Hence a `connectDirect(IP, port)` command is forwarded down the layers until the `CommunicationHandler`.

5.4.3 Solutions to accommodate the environment

In general, it is a very different programming experience to program for the Android environment, than for example for a conventional desktop environment. In the sections above it is mentioned that an `Activity` can be paused and killed at any time. When comparing to standard laptop or desktop computers of today, CPU, RAM, screen size, and energy are all limited resources on a mobile device. The operating system environment tries to save power, and computations quickly rise to a level where it becomes a problem if the developer does not handle it. The following shows two examples from the implementation of the *NAI* framework client where issues like these come into play.

5.4.3.1 Wake lock

By default, an Android device tries to avoid using too much power. It does that by shutting down the processor and turning off hardware as soon as the device is unused⁷. This means that if you leave the device unused for a period of time the screen will turn off, and the application put on pause. Even if an application is running.

⁷The time-out for this is completely configurable by the user, but the user typically wants to have the devices going into sleeping mode as soon as the device is left alone

In the case of the *NAI* framework, it is obvious that the screen has to stay on, and remain lit while connected to the *NAI* server. In order for the device to do that, you need the permission of the user. The following code shows the `AndroidManifest` making the permissions explicit. When the application is installed the user is prompted to accept that permission request:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
...
    <uses-permission android:name="android.permission.WAKE_LOCK" />
</manifest>
```

After requiring the permission, the following code shows how to activate it:

```
public class BaseActivity extends Activity implements
    ICommunicationServiceCallback, OnTouchListener {
    ...
    public void onCreate(Bundle savedInstanceState) {
        ...
        PowerManager pm = (PowerManager) getSystemService(Context.
            POWER_SERVICE);
        wakeLock = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK |
            PowerManager.ACQUIRE_CAUSES_WAKEUP, "NAIClient");
        wakeLock.setReferenceCounted(false);
        ...
    }
    ...
    private void acquireWakeLock(){
        if (wakeLock != null)
            wakeLock.acquire();
    }

    private void releaseWakeLock(){
        if (wakeLock != null)
            wakeLock.release();
    }
    ...
}
```

The type of wake locks are optional, but we need the screen to be lit all the time during operation.

5.4.3.2 Asynchronous calls in and out of UI threads

Every `Activity` run in its own thread. The thread is locked in the sense that other threads are not allowed to manipulate any fields or object in the so called UI thread. The body of method calls coming from other threads should be wrapped in an instance of the type `Runnable`, and queued on the UI thread with the special method `runOnUiThread(Runnable)` that `Activity` class implements. This forces the developer to make methods calls asynchronous when calling methods on the UI thread. Otherwise, the UI would not be running fluently.

Although the *NAI* client application does not have many UI elements, there is a lot of updates of the single `ImageView` element attached to `BaseActivity`, and the user must be able to invoke the options menu at any time. Picture frames comes at approximately 15 frames per second, and they must be processed and inflated on to the `ImageView` for the user to see. While incoming method calls must be asynchronous, also outgoing

calls from the UI thread must be asynchronous, to allow the UI to be updated as fast as possible. That is why the `CommunicationService` has a list of tasks of the type `Runnable`s, so that incoming messages from the communication can be delivered to the UI thread without blocking the socket connection as well as `BaseActivity` can deliver messages to the server asynchronously via the task pool of `CommunicationService` without blocking the UI thread.

5.4.4 Android versions and device types

At present, Android has been released in a number of versions⁸. Every release has a slight change in the API's adding new features, support for different hardware types, or performance improvements for every release. Up to this point older applications are supported by newer Android releases.

Currently the 3.1 Honeycomb is the latest release from Google, but it is only targeted tablet devices, and 2.3.X is called Gingerbread and is the latest release targeted smartphones. Only the most recent phones have this version installed.

Application development must target a specific API version and for the *NAI* framework, version 2.1 Eclair has been chosen in order to be able to support more than 95%⁹ of already existing Android devices, while still utilising the benefits of the improvements over the 1.X versions, like performance, and a better API.

⁸<http://www.android.com>

⁹According to <http://developer.android.com/resources/dashboard/platform-versions.html> (as of 2 July 2011). Data was collected during a 14-day period ending on June 1, 2011.

Using the *NAI* framework for application building

This chapter provides an example of a fully implemented non-trivial application illustrating the use of the *NAI* framework. Code examples are provided where specific framework components have been used. The application is also used as part of the evaluation of the interaction design and framework features in chapter 7.

6.1 Restaurant of the future

The scenario is a restaurant, which has replaced all dining tables with interactive tabletops. The tabletops are running a special ordering and payment application which has been built using the *NAI* framework, so the customers can order and pay their food and beverages directly at the table using their smartphones. For simplicity reasons the restaurant visit is divided into three states: ordering, dining and checkout.

6.1.1 Ordering

This is the initial state, where the guests arrive. The restaurant application uses the basic authentication mechanism of the *NAI* framework where users only have to enter their names. The restaurant application presents a menu (figure 6.1) and an order summary (figure 6.2). When the restaurant guests have completed pairing during the setup procedure, their names automatically appear in the order summary. This functionality is made possible by the `IdentifiedPersonArrived` event. The menu contains a set of dishes and beverages with special order buttons (+ and -), which may only be clicked via a smartphone. The XAML code for the buttons must hence block normal click events and have an `IdentifiedClick` handler. The XAML code for the add item button (+) is:

```
<id:IdentifiedSurfaceButton BlockClickEvent="True"  
    IdentifiedClick="AddItem">+  
</id:IdentifiedSurfaceButton>
```

When finished ordering, the guests move their smartphones to the center of the tabletop screen to confirm their orders. The confirmation



Figure 6.1: The menu in the restaurant application. The order buttons on the right are only clickable via a smartphone.



Figure 6.2: The order summary in the restaurant application. Automatically updated based on different *identified events*.



Figure 6.3: The confirm order `PersonalizedView` in the restaurant application. A personal dialog only visible on the smartphone display.

happens through a small personal confirm dialog appearing on the smartphone display when it is near the center of the screen (figure 6.3). The confirm dialog is a `PersonalizedView` and appears when an `IdentifiedHoverOver` event is raised on the red ellipse in the center. The C# event handling code which adds and removes the `PersonalizedView` looks like this:

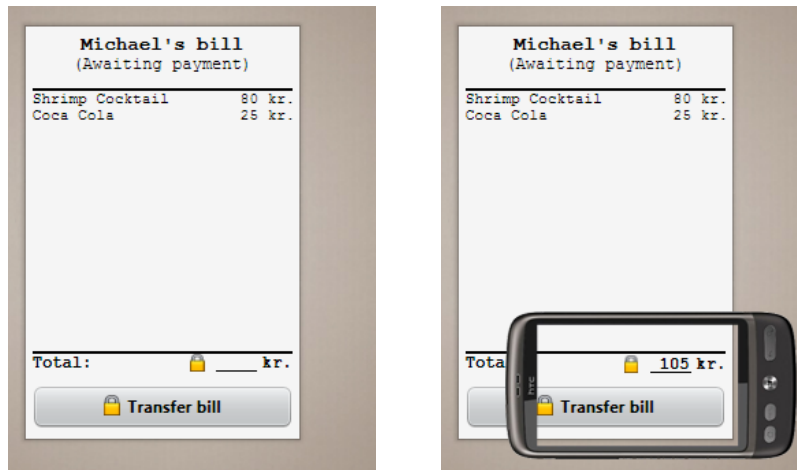
```
private void Ellipse_HoverOver(object sender,
    RoutedIdentifiedHoverEventArgs e)
{
    e.ClientId.PersonalizedView.Add(new ConfirmOrderPersonalView());
}

private void Ellipse_HoverOut(object sender,
    RoutedIdentifiedHoverEventArgs e)
{
    e.ClientId.PersonalizedView.Remove();
}
```

The `RoutedIdentifiedHoverEventArgs` instance provides access to the identity of the originator of the event where the `IPersonalizedView` interface is used to add a `PersonalizedView`. If the smartphone is moved away from the center, the `PersonalizedView` is removed again by the `IdentifiedHoverOut` event handler.

6.1.2 Dining

When finished ordering the guests wait for their food and beverages to arrive. At this state the tabletop is passive and waits for the guests to finish eating. The guests may proceed to checkout by confirming with their smartphone in the middle of the tabletop screen the same way as for ordering.



(a) The total amount is by default not visible.

(b) The bill owner's smartphone may reveal the total amount.

Figure 6.4: The personal bill in the restaurant application.

6.1.3 Checkout

At this state the guests are presented with a personal bill containing the items ordered initially (figure 6.4(a)). Only the owner of the bill is able to see the total amount of the bill at the bottom, by placing the smartphone on top of the area (figure 6.4(b)). To create this functionality, the `IdentifiedViewport` control is extended:

```
public class PrivateIdentifiedViewport : IdentifiedViewport
{
    ...

    public ClientIdentity ClientId
    {
        get { return (ClientIdentity)GetValue(ClientIdProperty); }
        set { SetValue(ClientIdProperty, value); }
    }

    public PrivateIdentifiedViewport()
    {
        base.FilterDelegate = new IdentifiedViewportFilterDelegate(
            Filter);
    }

    private bool Filter(RoutedIdentifiedEventArgs e)
    {
        if (ClientId == null) return false;
        return e.ClientId.Equals(ClientId);
    }
}
```

This new class has an `ClientIdentity` dependency property which is a reference to the owner of the bill. The property is used in the `Filter` function that is set up to only allow the bill owner to see through. In the XAML code for the bill, which is a `ScatterViewItem` from the Surface API, the `PrivateIdentifiedViewport` is set up to be a small white rectangle, just big enough to cover the total amount. Data binding is

used to set the owner of the bill. A simplified version of the XAML code for the bill with the `PrivateIdentifiedViewport` looks like this:

```
...
<s:ScatterViewItem ScatterManipulationDelta="onSMD">
    ...
    <id:PrivateIdentifiedViewport x:Name="TotalAmountVP"
        ClientId="{Binding Path=ClientId}" Fill="White">
        <id:PrivateIdentifiedViewport.Base>
            <RectangleGeometry Rect="5,0,30,13" />
        </id:PrivateIdentifiedViewport.Base>
    </id:PrivateIdentifiedViewport>
    ...
</s:ScatterViewItem>
...
```

The `ScatterViewItem` has a handler for the `ScatterManipulationDelta` event. The event is raised every time the bill is moved, scaled or rotated. The handler must ensure that when the bill is moved, the `PrivateIdentifiedViewport` over the total amount is notified, so it can hide the total amount if the bill moves outside the display area of the bill owner's smartphone. The handler calls the `Moved` method on the `PrivateIdentifiedViewport`, which updates the layout of the viewport:

```
...
private void onSMD(object sender, ScatterManipulationDeltaEventArgs e)
{
    TotalAmountVP.Moved();
}
...
```

A guest may transfer another guest's bill by clicking the `Transfer` button with his smartphone at the bottom of the bill (figure 6.4). The button is an `IdentifiedSurfaceButton` that has a small picture of a pad lock to hint that this button requires an *identified click*. The XAML code for the button is:

```
...
<id:IdentifiedSurfaceButton x:Name="BtnTransfer" BlockClickEvent="True"
    IdentifiedClick="Transfer_Click">
    <StackPanel Orientation="Horizontal">
        <Image Source="/Restaurant;component/Resources/SecureLock.png"
            Margin="0 0 5 0" Height="15" />
        <TextBlock VerticalAlignment="Center">Transfer bill</TextBlock>
    </StackPanel>
</id:IdentifiedSurfaceButton>
...
```

The `IdentifiedClick` event handler in the `C#` code-behind file has access to the owner of the bill as well as the data model of the restaurant, where the new payer is retrieved from, based on his identity (`e.ClientId`):

```
...
private void Transfer_Click(object sender, RoutedIdentifiedEventArgs e)
{
    Person oldPayer = this.Owner;
    Person newPayer = Restaurant.GetPerson(e.ClientId);
    Restaurant.PayForPerson(oldPayer, newPayer);
}
...
```

The bill is paid by placing the smartphone on the center of the tabletop screen and confirming the total amount in the personal dialog on the smartphone display (figure 6.5). The dialog is a `PersonalizedView` similar to the confirm dialogs in the ordering and dining state. The only difference in this dialog is that the total amount is displayed above the confirm button.

When all bills have been paid, the restaurant visit, and hence the application ends.

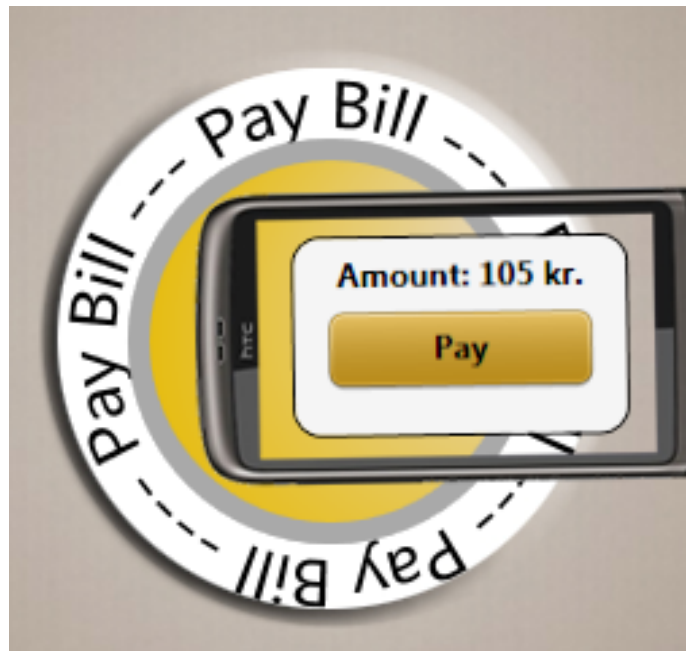


Figure 6.5: The confirm payment `PersonalizedView` in the restaurant application. The dialog is personal and shows the amount to be paid.

Evaluation

The way a smartphone is used for non-anonymous interaction in the *NAI* framework is unprecedented, so the question of the level of usability in this new way of interacting with a tabletop is interesting and important to investigate. It has been one of the goals with the interaction design that the required involvement of the smartphone does not introduce special gestures, but instead resembles normal tabletop interaction to allow end users a seamless and quick adaptation. If end users accept this new way of interacting, it helps in convincing tabletop application developers to adopt and use the *NAI* framework.

The only way to run an evaluation with end users is to build an application using the *NAI* framework. This makes it difficult to get qualified feedback from users on the underlying features of the *NAI* framework without including the application context, but it is nevertheless the purpose of this evaluation.

The evaluation involves two applications built using the *NAI* framework. The first is a technical warmup application only containing the main UI features of the *NAI* framework, like the `IdentifiedSurfaceButton`, `IdentifiedViewport` and `PersonalizedView`. The second application is the restaurant introduced in chapter 6, because it is built around a familiar scenario, a restaurant visit, and it utilises all the components in the *NAI* framework.

7.1 Parameters

Three main aspects are covered in the evaluation: *technical*, *adaptation*, and *experience*.

The *technical* aspect operates with trivial questions like *does the framework operate as intended?* Exposing the *NAI* framework to end users may uncover shortcomings or limitations of the framework design which has not been thought about previously.

More interestingly is it to see how the users *adapt* to the framework. How quickly do they figure out the new interaction design? Do they find it difficult to use?

The *experience* is about how the users interact when they have adopted the design. Do they figure how to use the new UI features provided

by the *NAI* framework, the `IdentifiedSurfaceButton`, `IdentifiedViewport` and `PersonalizedView`?

7.2 Method

To get qualified data to answer the evaluation parameters, test sessions involving participants have been conducted. The idea of having user identification in applications on a tabletop computer is more interesting if there is some distinction between simultaneous users like in the restaurant application, so the participants are coupled in pairs.

At a more scientific level, the purpose of the test sessions is to *generate* and *gather* data to answer the evaluation parameters. The data is the test participants' opinions and experiences.

7.2.1 Generating data

To generate data, the participants go through two test scenarios, one *exploratory* and one *guided*.

Exploratory The participants are in turn shown the technical warm-up application containing some standard MSS features as well as the features of the *NAI* framework which the test participants may interact with freely. If the participants are not used to interacting with a tabletop, this application also serves well as a playground. The participants do this in turn because they are not supposed to give each other ideas on how to using the smartphone for interaction with the tabletop. The application as such, has no other purpose than showing the different features from within one screen, allowing the user to explore the interaction possibilities. The participants are given only a brief introduction to the framework so they know what to use the smartphone for. This part of the test gives a unaffected view of how users initially perceive the interaction design and the *NAI* framework features. The data generated here is the most valuable, because it gives an idea of whether users are interacting as expected by the *NAI* framework design. After a couple of minutes the exploratory parts ends. Figure 7.1 shows an image of the warm-up scenario from one of the test sessions.

Guided The test persons are taken through the restaurant application that implements the features of the *NAI* framework. The scenario puts the two participants into a *host* and a *guest* role. The restaurant visit involves ordering, eating (only simulated) and payment. The participants are guided by a narrator introducing the three steps in the restaurant visit as well as telling them exactly what to do and in which order. The guide scenario in the test session is to ensure completeness, so after the restaurant visit, the participants have experienced some mandatory features which give them the foundation needed for providing a truthful view of the interaction experience and the *NAI* framework features. Figure 7.2



Figure 7.1: A test participant interacting with the warm-up application.

shows an image of the guided scenario from one of the test sessions.

The detailed scenario descriptions that have been used in the evaluation sessions can be found in appendix B.1.

7.2.2 Gathering data

To conclude a test session, a semi-structured interview is conducted with each of the test participants. An interview is preferred over a survey, because it allows the participants to talk more or less freely about their experiences. The semi-structured interview includes some questions which focus on how easy it was to adapt to the interaction design, and how the user experienced using it. If the participant has other relevant comments or has done something other than expected during the test scenarios, there is a chance to dig further into this. The questions can be found in appendix B.2.

One shortcoming of conducting interviews is that they suffer from problems of recall [7]. The participants are reporting on experiences they remember, and that is one step away from reality. If the interview is conducted during the test scenarios, for example when sitting in the restaurant, the answers would probably be different. To overcome this shortcoming, researchers suggest to use observations in combination with interviews. Observations help to understand the relationship between what the participants say and what they do: “*look at behavior, listen to perception*” (Miller and Crabtree [10]). In the evaluation of the *NAI* framework, observations are gathered by recording the test sessions with a video camera.



Figure 7.2: Two test participants ordering food in the restaurant application. The narrator sits in the background.

7.3 Test sessions setup

The MSS tabletop is set up in a cosy and comfortable meeting room at ITU without any chance of interruptions during the test sessions. Two HTC Desire smartphones running Android 2.2 are used. The required authentication of the smartphone user, and the pairing and calibration of the smartphone are carried out before the phones are handed over to the participants.

The area around the tabletop surface is recorded on video during all test sessions to afterwards be able to look closer at how the users interact. Also still images are captured during the sessions. The interviews conducted in the end of the sessions are all voice recorded for later analysis. The setup for the test sessions is depicted in figure 7.3.

7.4 Participants

Recruiting the right participants for a human–computer interaction (HCI) test is a delicate task. Choosing a set of participants which closely resemble the characteristics of the final end users of the product is crucial for the validity and usability of the evaluation results. For the *NAI* framework, the potential end users are anybody, or at least anybody with a smartphone. The restaurant application introduced in chapter 6 is a good example of a setting where a wide variety of users will encounter the *NAI* framework. So the amount of experience with computers and technology for the end users is expected to be varying, and it should preferably be reflected by the choice of participants in the test sessions.

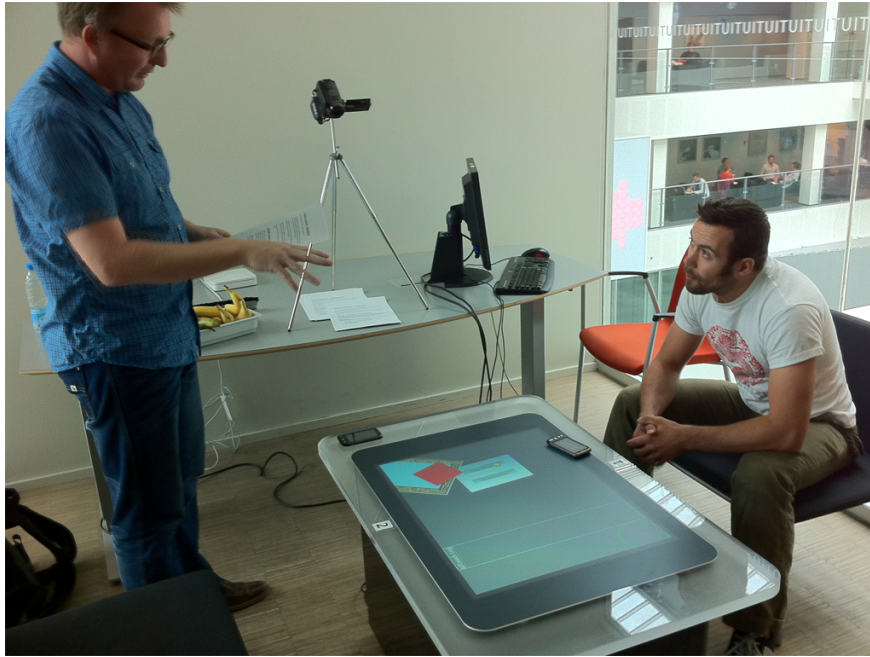


Figure 7.3: The setup for the test sessions.

As stated earlier, tabletops are not a commodity, and hence not something people stumble upon everyday. So in order to avoid making the test sessions an education and training in tabletop use, only people with interest and experience in technology have been recruited to participate.

Researchers in HCI argue differently for the number of participants needed in a user interaction evaluation. Virzi [23] has shown that as few as 4-5 participants can be enough to uncover 80% of the usability problems in an application.

In this evaluation 17 persons have participated in total. They are divided into 7 pairs and 3 individuals. A dummy plays the role of the guest in the restaurant scenario for the 3 individuals. In total 10 test sessions. 4 of the 17 participants have been recruited by advertising in the weekly edition of the IT University newsletter (see appendix B.3) and the rest are first or second hand associates of the pITLab at ITU.

7.5 Results

The ten test sessions have generated a lot of data. The video recordings and interviews have been analysed to report on tendencies and common opinions. The results are divided in the three areas of interest:

7.5.1 Technical

In general, the *NAI* framework worked as intended. On the positive side, the framework supported the different usage patterns observed when the smartphones were moved around during the sessions. The smartphones were either slid directly over the surface or lifted from place to place.

When lifted, the smartphone temporarily loses its image stream, because the tabletop cannot detect the tag, but the socket connection between the tabletop and smartphone is persisted, so the image streaming is resumed almost instantly when the phone again is placed on the tabletop surface.

On the negative side there was a problem with the back of the smartphones which is related to the infrared vision system of the tabletop. The back of an HTC Desire is almost non-reflective, but it has a small shiny metal frame around the camera which is detected by the tabletop and sometimes results in a `Blob Contact` event. Some test participants visually experienced these unwanted events, because they led to the phone could be used to move objects around on the tabletop screen.

7.5.2 Adaptation

When asked whether the system was difficult to use, more than 90% of the participants agreed that there is a small learning curve. One participant said “*When you got used to it, it was pretty clear what to do*”. Words like “*simple*”, “*intuitive*” and “*straight forward*” were used to describe the initial experience.

There were also some more critical comments. First, several participants could foresee problems in elderly people using the system. Second, about one third of the participants could see potential security and privacy problems in using the smartphone for personalised interaction, for example when putting the phone on the table, you do not have the same control over it, as when it is in your pocket.

7.5.3 Experience

The UI features of the two applications were not all equally successful. For the `IdentifiedSurfaceButton`, in 60% of the test sessions, the first person to order food in the restaurant tried to click the order buttons with normal finger touch, even though it was visually hinted with a padlock (as depicted in figure 6.1) that the smartphone was required. On the other hand for the `IdentifiedViewport`, 86% instantly figured how to use the smartphone to reveal the total amount on the bill in the restaurant. The `PersonalizedView` got a generally bad reception. Many participants commented that it was difficult to know when they would emerge.

The interviews have provided a lot of comments about the UI design in relation to the test applications. For example, “*The dish title was separated from the order button, so it was difficult to choose the right button*” and “*... the middle had no indication that you had to put your phone there to confirm*”. Even though these two comments are related directly to the restaurant application, it is possible to extract the fundamental usability problems from them. This strategy has been used for many of the comments and have resulted in a set of general design guidelines which are useful when developing applications with the *NAI* framework:

List of choices When having a list of choices (like a restaurant menu), it should be very easy to see what item you are choosing with the

smartphone. The small displacement caused by the height difference between the surface display and the smartphone screen may make it difficult to distinguish the individual list items from each other if not designed correctly.

Visual clues For the user it is difficult to know when a `Personalized-View` emerges. Some visual clue indicating where to put the phone to get the view is important. The same goes for the `Identified-Viewport` control.

Screen real estate Balance the number of objects on the tabletop with the interaction area of the tabletop surface. There needs to be enough room for the phone(s) to operate, as well as for other interaction.

Lessen the movements Lessen the number of times the phone should be moved. Users care about their phone, and do not like to slide it along the surface, and get tired of lifting the phone from place to place. For example, a solution could be to allow users to order items from a restaurant menu via normal finger touch on the tabletop, but submit the order items by using the phone. Another scenario could be to place the phone at a certain spot, and then drag items to the phone via normal touch on the tabletop.

Unintended touch When people move their phone across the tabletop screen, they may unintentionally touch the surface display underneath with their fingers. A thin smartphone increases the chance, because it is more difficult to grab. The problem may be minimised by UI design, if not all visual elements are made moveable and touchable by normal touch input.

Discussion

In contrast to a personal device, devices designed for shared user experiences call for a different way of implementing applications. A tabletop cannot natively link an action to the identity of its originator. This renders it impossible to implement multi-user applications for tabletops that rely on *access control* or *dynamically customises the user interface*.

This thesis has changed that, by presenting a framework that allow developers to build applications for tabletop displays that can utilise non-anonymous user interaction, and by using standard hardware like an Android powered smartphone.

The framework has been implemented on the basis of a user experience design and a security analysis, and in order to qualify the work presented, the framework has been used for building a non-trivial application and a usability test has been conducted.

The following discusses the work presented in this thesis. The first thing that is relevant to discuss is the *user experience* of non-anonymous user interaction on tabletop displays using smartphones. Secondly, it is also relevant to discuss how the *technical solution* performed, and finally, it is relevant to discuss whether the *security* aspects were fulfilled.

8.1 User experience

As described earlier in this thesis, the Android powered smartphone functions as a two-way mediator between the tabletop display and the user. The smartphone screen functions both as output and input. The output is the screen showing a segment of the tabletop display, and the input is the user's actions by touching and moving the phone.

Using standard hardware like a smartphone has the benefit, that users may already own such a device and know how to operate it. A modern smartphone is a highly *personal* device and the evaluation established that the users found it was natural to use it for representing themselves.

Prior work has mostly tried to implement non-anonymous user interaction by using customised hardware, but Schöning et al. [21] and *PhoneTouch* [18] uses standard mobile phones. They do so by restricting the interaction to a tap with the phone, or an obstructing authentication every time a visual element has to be associated with an identity. The

HandsDown project avoids using other hardware than the tabletop itself, but instead requires that the user has a hand with spread out fingers fixed on the tabletop during interaction. We have chosen an approach that integrates with the gestures normally used on a tabletop display. A user must only make sure that the smartphone is placed over the object that needs to be touched.

During the evaluation several test participants tried to perform multi-touch gestures like pinch-and-zoom on the display of the smartphone, with the expectation that the object below would scale accordingly. This shows two things. First, it shows that even though multi-touch was excluded from the work for simplification reasons, it is clearly relevant for future work. Secondly, and more important, the fact that people intuitively expected and performed gestures known from touch enabled devices, indicate that we have designed a user experience that integrates with gestures normally used on tabletops.

The tabletop does not have hover events natively. When moving the phone around on the tabletop, it resembles the usage of a conventional mouse pointer, except from the fact that the phone hovers over an area, whereas the mouse pointer points to a single XY-coordinate. This is contrasted by prior work. As mentioned above, prior work tends to restrict the interaction possibilities to special gestures. Our solution enhances the set of interactions.

The evaluation participants were deliberately chosen among people with a strong knowledge of IT. The goal was to be able to skip the education and training in using the smartphone and tabletop, in order to keep focus on the features of the framework as such. But this raises the question, can the *NAI* framework features be targeted people without a strong knowledge of IT? Almost all, more than 90% of the test participants agreed that it was intuitive to use with a small learning curve. Not all had tried a tabletop before. Based on that, we find it safe to assume that non-IT people can use the framework features, for example at a restaurant, but it is of course required that they know how to use a smartphone, and have an idea of what a tabletop is.

The *NAI* framework features a number of UI controls. They were incorporated into the test applications, and used for *access control* and *dynamically customising the user interface*. The evaluation showed that the UI controls were not equally successful. 60% of the test participants instantly figured out to use the `IdentifiedSurfaceButton`, and 86% the `IdentifiedViewport`. The `PersonalizedView` gave the test participants problems, as it was not clear that hovering over a certain field would show a personalised view. These results point at a fundamental problem of evaluating a framework like the *NAI* framework. The test participants react to the user experience design in the context of the test applications, while the goal of the evaluation was to quantify the usability of the framework in isolation from the context. Nevertheless, the many comments pointing at the limitations of *how* the framework was incorporated into the design of the test applications have been summarised in a set of five design guidelines. The guidelines are listed in the evalu-

ation results, but one example could be, that test participants found it obstructive if the phone was to be moved all the time. It is therefore recommended that designers of future applications lessen the movements, by allowing users to move objects to the phone, rather than moving the phone to the objects.

8.2 Framework implementation

We have used all the components in the developer API to build the non-trivial restaurant application and the evaluation has shown that the components work and operate as intended.

The evaluation pointed at one weakness in the current implementation of the framework. If the back of a smartphone is able to reflect IR light, the tabletop may recognise the phone as a `Blob` and raise `Blob Contact` events. Several test participants experienced this phenomenon and were confused, because the smartphone some times accidentally moved elements around. This inconvenience is however a matter of implementation, and can be fixed if `Blob Contact` events raised in the area underneath the phone are suppressed.

To get the glass effect on the smartphone display, there is a strong requirement for a reliable and fast connection between the smartphones and the tabletop. This has been the primary reason for choosing the primitive socket connection in the transport layer with communication messages being converted to byte arrays. If we had chosen an application layer protocol like HTTP or had wrapped the communication messages in an XML based format, there would have been some extra payload to each message. In the current implementation we can stream 60 frames per second (FPS) to an HTC Desire smartphone. This is four times as much as the required 15 FPS to get the glass effect on the smartphone display. The framework does therefore not have any problems in supporting multiple simultaneous smartphones.

Once the connection between the smartphone and tabletop has been established, it is persisted throughout the application session. It means that if the phone rings or if the user lifts his phone from the tabletop, the underlying `Service` running on the Android smartphones keeps the connection alive. During the evaluation it was discovered that some users tended to lift the phone when it should be moved around. However the connection persistence meant that the image streaming was easily resumed when the phone was put bag on the tabletop surface.

We have used the framework to build a non-trivial application for a restaurant scenario. The application gives users the possibility to order and pay for food and beverages using their smartphones. The implementation of the restaurant application has shown that using the *NAI* framework resembles how traditional MSS applications are written based on WPF. The framework only has a small footprint in the amount of code lines needed to implement non-anonymous user interaction.

8.3 Security

The separation of concern in regards to security which has been described in the security analysis (chapter 4) is also the outset for this discussion.

The *NAI* framework uses the SSL protocol to secure the communication between smartphones and the tabletop. SSL effectively eliminates threats like eavesdropping and impersonation from message forgery, because of the confidentiality and integrity properties provided by the protocol. Compared to related work, the *HandsDown* [19] and *DiamondTouch* [4] both suffer from being vulnerable to impersonation attacks. In *DiamondTouch* impersonation can occur if users touch each other while interacting with the tabletop. In *HandsDown*, the identity of users are determined based on the contours of the users' hands, which may be duplicated by an adversary.

The required tag stuck on the back of the smartphone may easily be duplicated. So the *NAI* framework does not assume any relationships between particular tags and smartphones, but instead it requires every smartphone to be paired with its tag at startup. Furthermore the tabletop only allows one of each distinct tag to be recognised at a time.

The use of SSL and the strict tag management in the *NAI* framework provide a secure environment for developers to build applications with non-anonymous user interaction. The developers must however provide their own user authentication mechanism that satisfies the security requirements of their applications. The *NAI* framework has a pluggable interface where the user authentication mechanism can be customised. We have provided a basic authentication mechanism as an inspiration and to show that the pluggable interface works. Our mechanism only requires the user to enter a user id which is not very secure. But it can for example be extended to a two-factor approach like the classic username-password scheme.

The required user authentication on startup has not been part of our evaluation of the *NAI* framework, since the procedure is somewhat trivial and it resembles how users authenticate to other systems, for example on the internet. We did however get feedback on some security aspects of the *NAI* framework during the evaluation sessions. One user said that he would feel uncomfortable in removing his phone from his pocket and place it on the table. In regards to the *social engineering* threat the *NAI* framework suffer from, this is a good comment. If users are persuaded to leave their phones at the tabletop, the security of the *NAI* framework may be compromised. But when a user says that he prefers having his phone in his pocket, it means that he is not easily persuaded to leave it behind. We think this expression of awareness and protection of the phone come from the fact that it is a phone. The phone is a *personal* device as described earlier, and it is becoming an ever larger part of our everyday life. We use it for many activities all the time. This usage pattern fits well with the assumption in the *NAI* framework that users do not leave a paired phone behind at the tabletop and it provides further proof of the *NAI* framework being secure.

Conclusion

This thesis addressed the problem of linking an action to the identity of its originator in a way that allows applications to discriminate between simultaneous users. The motivation originated from related work, which stated problems of having *access control* [15, 14, 21] and *dynamic customisation of the user interface* [20, 16] on tabletop displays without non-anonymous user interaction. Prior work has solved these problems by using special hardware or by introducing special gestures.

Our solution to the problem was to create a new user experience design, where an Android powered smartphone is used as a two-way mediator between the user and a MSS tabletop. We implemented the user experience design as a framework called the *NAI* framework¹. It features a set of *identified events* and some UI controls for developers to use for application building. In a security analysis we showed that the framework is secure and that it is necessary to delegate the responsibility of user authentication to the framework developer. Hence we have implemented a customisable user authentication mechanism in the framework.

The *NAI* framework builds upon the MSS API and has proven successful for implementing a non-trivial application, an ordering and payment system for a restaurant. The application was used during a user evaluation of the usability of both the novel interaction design with the smartphone as well as the UI features provided by the *NAI* framework. The evaluation showed that the framework operates as intended. The users found the required use of the smartphone intuitive and easy to learn, but was generally not happy about moving the phone around too much. They also pointed at some shortcomings in the UI design of the restaurant application. These user comments have been turned into a valuable set of design guidelines recommended for developers using the *NAI* framework to follow.

We feel comfortable to say that the *NAI* framework solves the problem of having non-anonymous user interaction on tabletop displays, and thereby allows tabletop applications to feature *access control* and *dynamic customisation of the user interface*.

¹Appendix A describes where to find the source code.

9.1 Future work

During the development of the *NAI* framework, some ideas for future work has emerged. The ideas span both improvements of existing functionality as well as completely new features which can make the framework even more powerful.

The framework does not support identified multi-touch on the smartphone. It was decided from the beginning not to include this functionality for simplicity reasons. Both smartphones and the MSS tabletop support it, so there are no technical limitations in implementing this functionality.

The evaluation showed that the current implementation of the framework is dependent on the smartphones having a non-reflective back. Even the small metallic ring around the camera is enough to cause unwanted Contact events. This limitation can be eliminated by suppressing the Contact events raised in the area underneath the phone. The tabletop can determine the size of the smartphone by adding an extra step to the calibration.

The current customisable authentication mechanism in the framework is limited to only be used during the startup of the smartphone application. Applications may very well require stronger authentication functionality where some individual events, like a click on a button, requires the user to be authenticated again.

The current pairing mechanism using PIN codes is somewhat cumbersome and requires the user's attention. The pairing could however be performed automatically, for example by using a pairing mechanism where a unique colour sequence is displayed by the tabletop underneath the phone and detected by the smartphone camera. The colour sequence is then converted to numeric code by the smartphone and sent to the tabletop for verification.

References

- [1] Eric A. Bier and Steven Freeman. MMM: a user interface architecture for shared editors on a single screen. In *Proceedings of the 4th annual ACM symposium on User interface software and technology, UIST '91*, pages 79–86, New York, NY, USA, 1991. ACM.
- [2] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.
- [3] P. Dietz, B. Harsham, C. Forlines, D. Leigh, W. Yerazunis, S. Shipman, B. Schmidt-Nielsen, and K. Ryall. DT controls: adding identity to physical interfaces. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 245–252. ACM, 2005.
- [4] P. Dietz and D. Leigh. DiamondTouch: a multi-user touch technology. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 219–226. ACM, 2001.
- [5] Stephen Downes. Authentication and identification. *International Journal of Instructional Technology and Distance Learning*, 2(10):3–18, October 2005.
- [6] J.Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118. ACM, 2005.
- [7] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research Methods in Human-Computer Interaction*. Wiley Publishing, 2010.
- [8] Nicolai Marquardt, Johannes Kiemer, and Saul Greenberg. What caused that touch?: expressive interaction with a surface through fiduciary-tagged gloves. In *ACM International Conference on Interactive Tabletops and Surfaces, ITS '10*, pages 139–142, New York, NY, USA, 2010. ACM.
- [9] T. Meyer and Dominik Schmidt. IdWristbands: IR-based user identification on multi-touch surfaces. In *ACM International Conference on Interactive Tabletops and Surfaces*, pages 277–278. ACM, 2010.
- [10] W.L. Miller and B.F. Crabtree. *Clinical Research: A Multimethod Typology and Qualitative Roadmap*. Sage Publications, 1999.

-
- [11] B.A. Myers, H. Stiel, and R. Gargiulo. Collaboration using multiple pdas connected to a pc. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 285–294. ACM, 1998.
- [12] Yongqiang Qin, Chun Yu, Hao Jiang, Chenjun Wu, and Yuanchun Shi. pPen: enabling authenticated pen and touch interaction on tabletop surfaces. In *ACM International Conference on Interactive Tabletops and Surfaces, ITS '10*, pages 283–284, New York, NY, USA, 2010. ACM.
- [13] Aaron Quigley. *From GUI to UUI: Interfaces for Ubiquitous Computing*, chapter 6, pages 237–284. Ubiquitous Computing Fundamentals. CRC Press, 2010.
- [14] M. Ringel, K. Ryall, C. Shen, C. Forlines, and F. Vernier. Release, relocate, reorient, resize: Fluid techniques for document sharing on multi-user interactive tables. Technical report, Mitsubishi Electric Research Laboratories, April 2004.
- [15] V. Roth, P. Schmidt, and B. Gldenring. The IR ring: Authenticating users' touches on a multi-touch display. In *Proceedings of the 23rd annual ACM symposium on User interface software and technology*, pages 259–262. ACM, 2010.
- [16] K. Ryall, A. Esenther, K. Everitt, C. Forlines, Morris, M. R., C. Shen, S. Shipman, and F. Vernier. iDwidgets: Parameterizing Widgets by User Identity. In *Proceedings of IFIP INTERACT'05: Human-Computer Interaction*, Short Papers: Tools, pages 1124–1128, 2005.
- [17] K. Ryall, A. Esenther, C. Forlines, C. Shen, S. Shipman, MR Morris, K. Everitt, and FD Vernier. Identity-differentiating widgets for multiuser interactive surfaces. *IEEE Computer Graphics and Applications*, 26(5):56–64, 2006.
- [18] Dominik Schmidt, Fadi Chehimi, Enrico Rukzio, and Hans Gellersen. PhoneTouch: a technique for direct phone interaction on surfaces. In Ken Perlin, Mary Czerwinski, and Rob Miller, editors, *UIST*, pages 13–16. ACM, 2010.
- [19] Dominik Schmidt, Ming Ki Chong, and Hans Gellersen. HandsDown: hand-contour-based user identification for interactive surfaces. In Ebba Hvannberg, Marta Kristin Lrusdttir, Ann Blandford, and Jan Gulliksen, editors, *NordiCHI*, pages 432–441. ACM, 2010.
- [20] Dominik Schmidt, Ming Ki Chong, and Hans Gellersen. IdLenses: Dynamic personal areas on shared surfaces. In *ACM International Conference on Interactive Tabletops and Surfaces*, pages 131–134. ACM, 2010.
- [21] Johannes Schning, Michael Rohs, and Antonio Krger. Using mobile phones to spontaneously authenticate and interact with Multi-Touch surfaces. In *AVI 2008: Workshop on designing multi-touch*

interaction techniques for coupled private and public displays, May 2008.

- [22] J. Stewart, B.B. Bederson, and A. Druin. Single display groupware: a model for co-present collaboration. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pages 286–293. ACM, 1999.
- [23] Robert A. Virzi. Refining the test phase of usability evaluation: how many subjects is enough? *Hum. Factors*, 34:457–468, August 1992.

Framework source code

The source code for the *NAI* framework has been released under an open source license and is available at:

<http://code.google.com/p/nai-framework/>

The code base includes the two applications, Warm-up and Restaurant, which have been used in the evaluation of the framework.

The code is accompanied by some wiki articles on how to get started with using the framework, and a video presenting the features of the framework. The table of contents of the wiki pages are:

- What is the *NAI* framework?
- Install and run the demo applications
- Create your own applications
 - Customising the framework to your needs
- Improving the *NAI* framework

Evaluation material

B.1 Test scenarios

B.1.1 Scenario 1

The first scenario is unstructured. After a short introduction (see below), the test persons can freely interact with a demo application that puts forward all the features of the *NAI* framework. There are no guidance by the supervisors.

B.1.1.1 Introduction read aloud

This evaluation is a part of a project that address the problem of having a tabletop computer distinguish between its users and their actions. The problems can be solved by using the screen of a smartphone.

For example like this [place the button control below the phone and press the button via the phone. Point to the users name in the log].

This is the first of two applications that you are going to test. This is a technical application, showing a set of interaction features, the second application will be related to a real life scenario.

You can try it out yourself during the next couple of minutes. Please go ahead, and say aloud what come to your mind.

B.1.2 Scenario 2

The second scenario is structured. The following states the test read aloud, including the instructions for the test persons for what to do. This scenario is simulating a real life scenario: Ordering, eating, and paying for food or beverages at a restaurant.

B.1.2.1 Instructions

You are now sitting as guests in a restaurant. You are here because you want something to eat. Person 1 (point him out) has invited a person out for dinner. The restaurant has incorporated a new interactive dining experience where tables are interactive surfaces, and orders are placed and paid via a smartphone. A typical restaurant visit, like this one, involves

three states which you are now going through. They are ordering, dining, and checkout. I will tell you what to do. Let us get started...

Ordering state This is the Ordering state. The Summary shows you what you and others have ordered. The state end placing your orders via the center field (point to it).

1. Host: Choose from the menu a dish and one beverage for yourself
2. Guest: Choose the same dish as your host, and a beverage of your own choice.
3. Host: Wait for your guest to choose a dish and something to drink.
4. Host: Cancel your dish, and choose another dish from the menu.
5. Both: Place your order.

Dining state This is the Dining state. After a short while you will get your food, and when you have finished eating it, please proceed to checkout using the center button (point to it, give the participants a little something to eat).

1. Both: Eat up and proceed to check out

Checkout state This is the Checkout state. Each person gets a bill based on the items from the menu that he or she ordered. For privacy reasons, the actual cost has been hidden. The scenario ends when all persons have payed their bills using the center field (point to it)

1. Guest: Check the bill for errors. Is the summation correct?
2. Guest: Announce the correctness and total amount of your bill to your host
3. Host: Transfer the bill of your guest to yourself.
4. Host: Check the bill for errors. Is the summation correct?
5. Host: Pay your bill

B.2 Semi-structured interview - prepared questions

The prepared set of questions are split into two sections. First, questions on adaptation. Secondly, questions related to the users experience.

B.2.1 Adaptation

Users evaluation of the difficulty in learning the interaction design. Learning curve.

- Did you find it difficult to use?
- Was it difficult in the beginning?
- Was it easier as the session progressed?
- Do you think it could be a challenge for other users to use applications like the one you just tried?
- Why not?/Why do you think that? (depending on the previous answer).

B.2.2 Experience

Users evaluation of the framework features and two general questions in the end.

- Do you remember the situation where you ordered something to eat?
- Do you remember if that made you think?
- If **YES**: What?
- If **NO**: Do you have any remarks on that?

- Do you remember the situation where you checked the bill for summation errors? Do you remember if that made you think?
- if **YES**: What?
- if **NO**: Do you have any remarks on that?

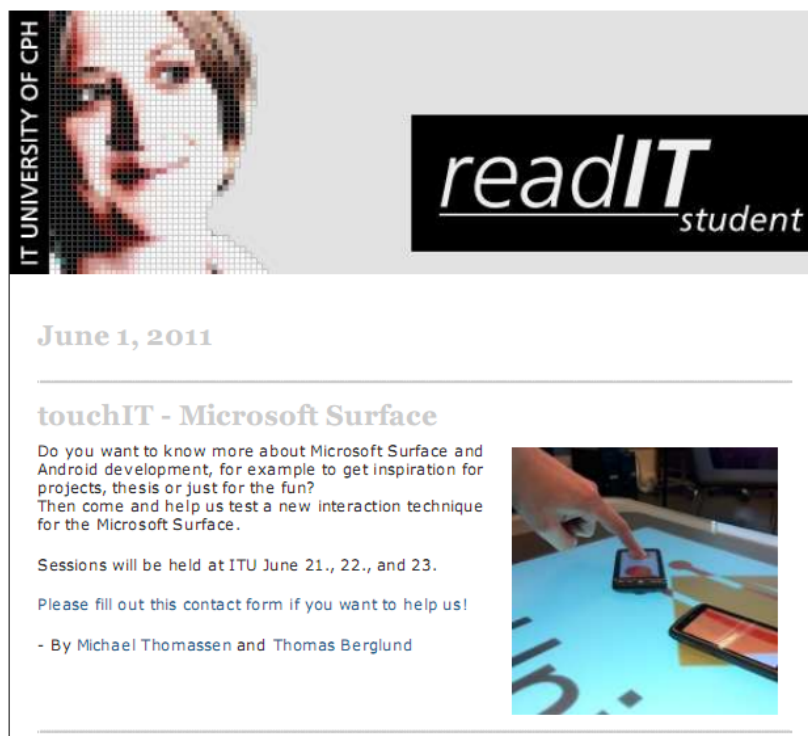
- Do you remember the payment situation, where the total amount was confirmed on the phone?
- Do you remember if that made you think?
- If **YES**: What?
- if **NO**: Do you have any remarks on that?

- Did you at any time find the required usage of the smartphone obstructive?
- If **YES**: When? Why?
- If **NO**: Why is that, you think?


- Do you think it is a problem to use the mobile phone for personalized interaction?
- Why?/Why not?

B.3 ITU newsletter advertisement

The advertisement from the weekly newsletter “readIT” in figure B.1.



IT UNIVERSITY OF CPH



readIT
student

June 1, 2011

touchIT - Microsoft Surface

Do you want to know more about Microsoft Surface and Android development, for example to get inspiration for projects, thesis or just for the fun? Then come and help us test a new interaction technique for the Microsoft Surface.

Sessions will be held at ITU June 21., 22., and 23.

Please fill out this [contact form](#) if you want to help us!

- By Michael Thomassen and Thomas Berglund




Figure B.1: The advertisement for evaluation participants in the ITU weekly newsletter 1 June 2011

Responsibilities

C.1 Thesis

| Chapter/section | Responsible |
|--|-------------------------------------|
| 1 Introduction | Thomas Berglund |
| 2 Related work | Thomas Berglund |
| 3 The <i>NAI</i> framework | Thomas Berglund & Michael Thomassen |
| 4 Security analysis | Michael Thomassen |
| 5 Implementation | |
| 5.1 States | Michael Thomassen |
| 5.2 Communication | Michael Thomassen |
| 5.3 Tabletop | Michael Thomassen |
| 5.4 Smartphone | Thomas Berglund |
| 6 Using the <i>NAI</i> framework for ... | Michael Thomassen |
| 7 Evaluation | Michael Thomassen |
| 8 Discussion | Thomas Berglund & Michael Thomassen |
| 9 Conclusion | Michael Thomassen |

C.2 Code

C.2.1 Research and design

| Task | Responsible |
|-----------------------------------|-------------------------------------|
| Architecture | Thomas Berglund & Michael Thomassen |
| Pilot project: Controlling events | Thomas Berglund |
| Pilot project: Data Binding | Thomas Berglund |

C.2.2 Smartphone (Android)

| Weight | Responsible |
|--------|-------------------|
| 10% | Michael Thomassen |
| 90% | Thomas Berglund |

C.2.3 Tabletop (Microsoft Surface)

| Weight | Responsible |
|---------------|--------------------|
| 90% | Michael Thomassen |
| 10% | Thomas Berglund |

C.2.4 Evaluation

| Task | Responsible |
|------------------------|--------------------|
| Warm-up application | Thomas Berglund |
| Restaurant application | Michael Thomassen |