ulm university universität ulm

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Bachelor thesis
in Media Informatics Course

Conception and Implementation of a Location-based Augmented Reality Kernel

Targeting the Windows Phone 8.1 Operating System

presented by

Emre Inanc

June 2015

Referee	Prof. Dr. Manfred Reichert
Supervisor	Rüdiger Pryss
Matriculation number	750771
Presented	June 22, 2015

Abstract

The availability of sophisticated mobile applications on many platforms constitutes a challenging task. In order to cover the most relevant mobile operating systems and make the best use of their underlying features, the native development on the target platform still offers the most diverse possibilities. Aside from the most widely spread mobile operating systems - namely Android and iOS - the Windows Phone platform offers a unique design language and many developer tools and technologies for building Windows Store apps. Making use of the capabilities of modern smartphones enables the development and use of desktop-like applications. The built-in sensors, cameras and powerful processing units of such a device offer a versatile platform to build against. As a result, many mobile applications and technologies have emerged. However, information on profound insight into the development of such an application is hard to find. In this work, the development of AREA on the Windows Phone 8.1 platform is presented. AREA is a location-based mobile Augmented Reality engine and already available on Android and iOS. By porting the engine to yet another mobile platform, more third-party mobile business applications can integrate AREA and make use of its efficient and modular design. This work also points out the differences in implementation between the Windows Phone version and its counterparts on Android and iOS. Insights into the architecture and some references to the mathematical basis are also provided.

Statutory Declaration

I declare on oath that I completed this work on my own and that I used no other than the declared accessories. Information which has been taken from other sources (including electronic media sources) has been noted as such.

Ulm, June 22, 2015 Emre Inanc

Contents

List of Figures

Zu einem guten Ende gehört auch ein guter Beginn.

Konfuzius, (551 - 479 v. Chr.)

1 Introduction

The use of sophisticated mobile applications has become an essential part of today's world. Since the breakthrough of smartphones, the need for accessing information systems from mobile devices has not only reached private individuals, but also corporate environments[1] and even institutions that are regulated by public law[2]. However, even if the number of mobile applications is growing [USA15], it remains a challenging task to design and develop them[3]. This is attributed both to hardware-specific limitations of the devices that run the applications and to the characteristics of the underlying operating system. While the former might limit the developer with the availability of physical resources (e.g. screen size or battery capacity), the latter requires him to have profound knowledge of the platform basics (e.g. app model architecture and programming concepts). In view of these circumstances, making an application available to a larger audience across different devices has never been as easy as today. The open source mobile operating system Android has been pushed by Google to a market share of 78%, giving both manufacturers and developers the opportunity to address hundreds of millions of devices with a single application. Its open source nature might have contributed to this outcome, since the remaining two major operating systems iOS (18.3%) and Windows Phone (2.7%) are proprietary [USA15]. All handsets that ship with one of these major operating systems, have more or less the same devices and sensors that are available to the developer. A GPS sensor can locate the device up to a precision of a few meters, motion sensors can track the attitude of the device in space and high-resolution cameras enable to augment the reality with additional information. Business applications can benefit from these built-in sensors, but it is up to the developer to address them correctly and make use of their provided data. In order to deliver the same user experience across all mobile operating systems, several functional and non-functional requirements have to be met by using the operating system specific concepts and application programming interfaces (APIs). This paper deals with the design and development of a location-based Augmented Reality (AR) kernel on the Windows Phone 8.1 operating system. It is based on the AREA kernel that is already available on iOS and An-

[1] Delta Air Lines flight attendants are using Windows Phone handsets to interact with customers [Blo13].
[2] The German Bundestag provides an app that runs on several platforms [Bun13].
[3] Take, for example, the *TrackYourTinnitus* project on Mobile Crowd Sensing [PRH+15] [PRLS15] or large scale data collection with mobile devices [SPR15] [SSP+13]

droid [GPSR13]. AREA is a generic application that can be integrated into real-world mobile business applications.

1.1. Statement of the problem

In this work, the AREA kernel shall be ported to Windows Phone 8.1, using the Windows Runtime XAML Framework as the presentation technology and C# as the programming language.[4] The integrated development environment (IDE) that has been used for developing the application is *Microsoft Visual Studio Community 2013*. AREA augments the real image that is previewed from the camera of a smartphone with points of interest (POIs) that surround the user. It does so by taking the position, the attitude and the current angle of view of the device into account. As the core engine has already been developed for iOS and Android, the main purpose of this work is to give an insight into the engineering process of the Windows Phone version. Even if the basic concepts have been discussed [GPSR13], there are notable differences in implementing these. In order to meet basic requirements of the AREA engine, such as coping with limited resources and the integration into other applications, task and platform mappings have to be investigated and the engine needs to be adapted to the platform characteristics. Making use of the native APIs of the Windows Runtime is essential in order to address the sensors and the camera correctly.

1.2. Content structure

In the following, the structure of this paper is explained. In section 2, the main features and basic concepts of AREA are described. Section 3 gives an insight into related work, i.e. the engineering of AREA on iOS and Android. The requirements of the AREA engine are defined in section 4, followed by the architectural approach on Windows Phone 8.1 that is discussed in section 5. This section also includes a class structure diagram and an architecture diagram with the utilized libraries. In section 6, the implementation is outlined, including a discussion of a few aspects of implementation. The app is presented in section 7. After discussing whether the requirements are met in section 8, this work is summarized in section 9. This includes a short outlook on future work. The appendix section includes some formulas on the mathematical basis of AREA and detailed class diagrams.

[4] The Windows Phone Silverlight 8.1 app model has a limited feature set, as it is mainly preserved for backwards compatibility with Windows Phone 8 apps. DirectX is tailored for game development and WinJS enables app development with HTML5 and JavaScript. Since C# is similar to the Java programming language, it seems to be the most appropriate to choose the Windows XAML/C# app model (cf. Figure 1.1 and Figure 1.2)

Figure 1.1.: The Windows Developer Platform in 8.1 [WS14]

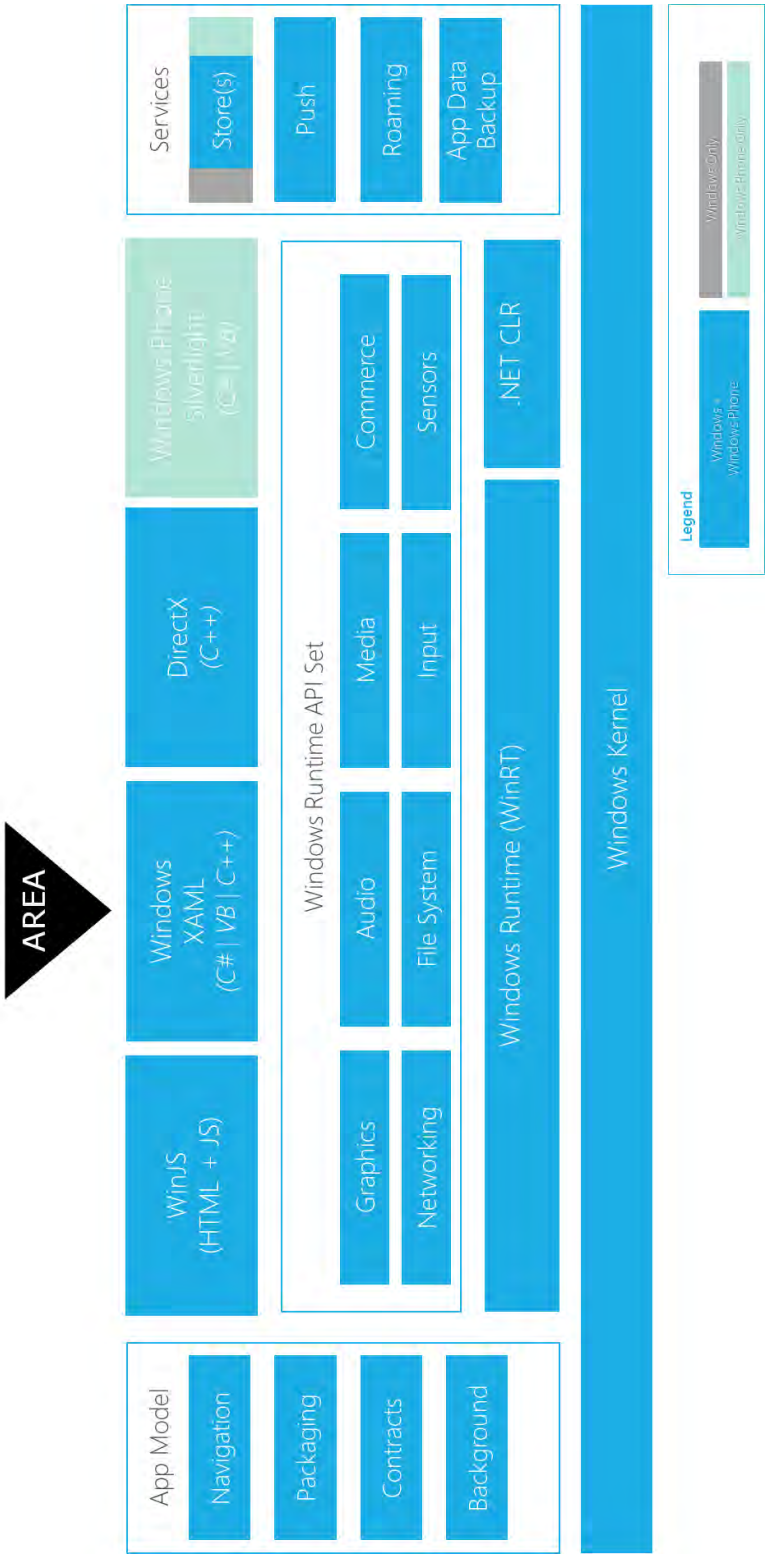
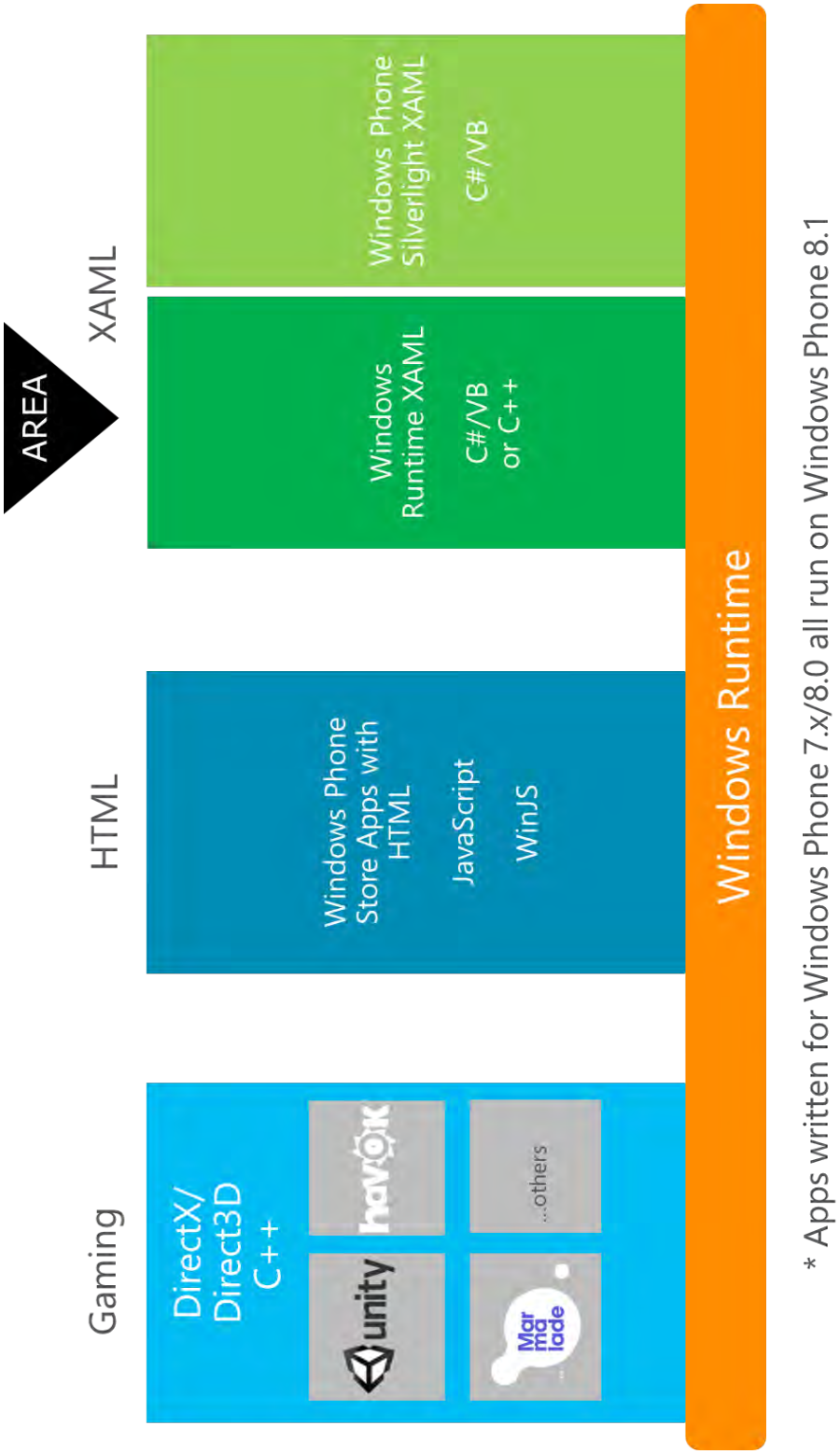


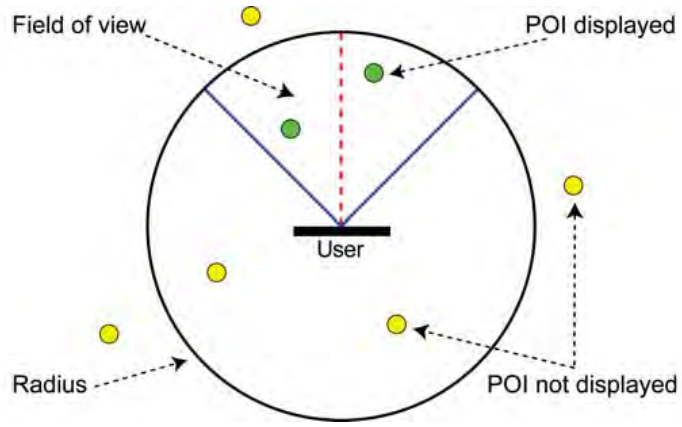
Figure 1.2.: Native App Development on Windows Phone 8.1 [WS14]



2 AREA

AREA is an abbreviation for *Augmented Reality Engine Application*[1]. The AREA engine is making use of the built-in sensors and devices of a smartphone. By determining the location and the attitude of the device, AREA can find surrounding points of interest (POIs) and show them on top of the camera preview, thus augmenting the real footage with additional information. Only POIs that are located in the same direction where the back of the smartphone is pointing at are shown to the user. This is because the camera that is utilized by AREA is the rear camera. Note that this visible field of view is limited by the camera's specifications (i.e. image sensor size and focal length). Its size must be calculated in order for AREA to work correctly [GPSR13].

Figure 2.1.: Schematic illustration of surrounding and visible POIs [GPSR13]



2.1. Data input

There are two parties that are involved in providing the data input whereby the position of the points of interest are calculated and eventually shown on the smartphone's display: the user and the provider of the application that has built in the AREA engine. As the user moves the smartphone, he changes its GPS position and its attitude in space. Unless the user is moving inside some means of transportation, it is very unlikely that the GPS data changes rapidly. In contrast, the sensor data is highly variable, because even slight movement of the device will

[1] More information can be found at `http://www.area-project.info`

affect it. This already implies that the querying of the sensors and the calculations need to be efficient and maintain the app's stability in order to guarantee a smooth user experience and reduce battery life consumption. On the other hand, the provider is responsible for supplying the engine with an XML file that contains the points of interest. This data is rather constant, as the provider usually updates the set of POIs periodically. With this in mind, there are three data categories to be collected and processed. Only under the premise that the surrounding POIs are within a given radius, the following data must be taken into account in order to place the correct POIs in the field of view [GPSR13]:

1. The altitudes of the device and POIs,
2. The bearing between them relative to the north pole, and
3. The attitude of the device based on three axes of the accelerometer.

Note that the radius can be determined by the user in order to limit the amount of POIs to be displayed on the screen. Any POIs that are not located within the given radius are neglected and therefore not included in the calculations above. The user will not see these POIs, even if they would be inside the field of view.

2.2. Feature set

Apart from the basic functionality of showing POIs on the display, AREA enhances the user experience with some additional features. The user can switch to a map view where he can see his current position and the surrounding POIs. If the full map experience is not needed, the radar feature will suffice. It includes all POIs relative to the position and viewing direction of the device. By tapping on a POI, the user can see additional information that is associated with it. Just as the iOS and Android counterparts, the Windows Phone version of AREA shall provide the possibility to integrate the engine into other applications. In order to facilitate such a procedure, third party apps have access to public interfaces and benefit from a high modularity. In particular, POIs can be extended by further properties without affecting the overall functionality of the engine [GPSR13]. The POIs are parsed from an XML-file that the third party app provides, but can also be added or removed at runtime.

2.3. Sensors

The Android version of AREA handles two sensors only, namely the magnetic and gravity sensor. It is possible to calculate the horizontal heading, the vertical heading and the pitch of the device on the basis of magnetic and gravity sensor data. While this approach is implemented in the Android version, the Windows Phone counterpart benefits from the combined sensors of the Windows Runtime API. Making use of the native API set of the underlying operating system is part of the sophisticated approach of AREA. Therefore, the horizontal heading and pitch are not calculated manually, but queried from the Compass and Inclinometer sensors of the Windows Runtime. Note that these are not standalone sensors that are based on physical units inside of the smartphone. Instead, they are virtual sensors that combine the data of other physical sensors. This combined sensor data is also known as *Sensor Fusion* [MSD15g]. Only the vertical heading needs to be calculated. Because there is no gravity sensor that provides

the data for calculating the vertical heading on the Windows Runtime platform, this is done by isolating the gravity force values from the accelerometer sensor data (just like in iOS [GPSR13]). In order to avoid inaccurate sensor data, the magnetometer's directional accuracy enumeration is queried and the user is prompted with instructions on how to calibrate the sensors. Before using any sensor data for calculations, normalizing and smoothing functions are applied.

2.4. Formal foundation

The formal foundation and mathematical basis of AREA have been discussed in [GPSR13] and [GSP+14] in great detail. Nevertheless, giving a short insight into the mechanics of the AREA engine can help to understand the functional principles of Augmented Reality applications[2]. In order for AREA to determine the surrounding POIs, the distance between them and the smartphone needs to be calculated. This is achieved by using the *Haversine-formula*, which eliminates the ill-conditioning of the *Great-circle distance* that is based on the spherical law of cosines [Gey13]. The Android version is making use of the native `Location.distanceTo()` method[3], whereas on Windows Phone the Haversine formula needs to be implemented manually. After obtaining the distance of the POI, there remain two other pieces of information that need to be considered when determining whether it should be placed on the screen or not. The horizontal heading holds a value from 0° to 360° and indicates the bearing between the user and the POI relative to the north pole. It is calculated based on the Great-circle distance. This leaves the vertical heading, which can be calculated by determining the adjacent elevation angle of the hypotenuse between the user and POI [GPSR13]. Should the horizontal heading of the POI contain a value between the left and right and the vertical heading a value between the top and bottom azimuth (i.e. the boundary of the field of view), the POI can be placed on top of the camera view. As mentioned in chapter 1, the size of the field of view needs to be calculated. As it depends on the image sensor size and focal length of the built-in camera, the respective values need to be researched and used for the calculation of the field of view angles. For this work, the angles are calculated and hard-coded based on the Nokia Lumia 735's camera specifications, getting a horizontal field of view of 75° and a vertical field of view 47°. The sizes for other devices can be calculated and saved inside of an enumeration for future work.

2.5. AREA in the field

The AREA engine is already being used in third party applications, such as the *LiveGuide* apps for *Ditzingen* [Dit15], *Bühlerzell* [Bü15] and *Muswiese*. All of these apps make use of the AREA engine in order to let users find points of interest. Demos of these and other apps with AREA can be viewed at `http://area-project.info/`.

[2] All formulas are included in the appendix A.

[3] Note that the Android method is using the *Inverse Formula* of Vincenty [Gre15] [Vin75]. It is less accurate, but a little faster than the Haversine formula.

3
Related work

In this chapter, a brief look at related work is taken. There are some papers that deal with location-based mobile Augmented Reality, but none of them seem to give a profound insight into developing a generic engine that runs on several mobile operating systems. Furthermore, there are a few open-source and proprietary SDKs, frameworks and libraries, as well as some commercial apps that implement different solutions for Augmented Reality applications.

3.1. Related papers

Some works focus on the design of location-based Augmented Reality applications. For instance, [FSBA06] covers the conceptual design of *Spatial Information Appliances*, which denotes a way of interacting with the physical environment. A pointer-based interaction model enables the user to point on real world objects and show related information on a mobile device. However, this work gives neither some insight into the engineering process, nor does it address the integration into existing mobile applications. [CFA+10] gives a great overview on Augmented Reality technologies, including some insight into research findings and existing applications. The work of [PT10] comes close to the issues and topics of this work, as it focuses on location-based Augmented Reality on mobile phones. The presented approach augments the camera view with 3D virtual objects, which are retrieved from a database. This requires the correct application of mathematical formulas and the efficient use of built-in sensors, such as the accelerometer and magnetometer. The result is a sophisticated application written in C++ for a Nokia N97 mobile handset. Even though the problem of limited resources on mobile phones is mentioned, this work does not provide an in-depth look at the engineering process. A rather interesting implementation of location-based Augmented Reality applications for mobile devices is presented by [RS03]. The authors discuss an indoor tracking system that is able to track the user within an environment, which is equipped with fiducial markers. The fairly large set of hardware that is attached to the body of the user (including a notebook and a helmet with sensors and a display), hosts the engine that is based on the ARToolkit libary [Lam15]. Furthermore, two applications that are built upon the engine are introduced. Finally, [LKKS09] introduces a web-based Augmented Reality application that can browse web media and overlay it on top of POIs. The mentioned works focus rather on the conceptual and architectural design than on the actual implementation. However, if any of the approaches are implemented, they target already outdated technologies and operating systems (Symbian or the CE kernel-based Windows Mobile) and don't provide any information on how the engineering task has been carried out.

3.2. Frameworks and Libraries

There is a rather large choice of Augmented Reality frameworks and libraries to build applications on. Among the open-source libraries, ARToolKit seems to be the most prominent one. It allows the tracking of optical markers to determine the users or real-world objects' position. This library was also used in the work of [RS03]. It is worth mentioning that there are many spin-offs and similar libraries, which is a common phenomenon in open-source projects. By taking a look at proprietary solutions, one comes across many popular frameworks. With the Wikitude SDK, developers can build Augmented Reality apps based on the advanced Wikitude library [Wik15]. The software supports many platforms and third-party frameworks, as it is even available to cross-platform development with Cordova [Cor15] or Xamarin [Xam15]. Junaio provides yet another powerful API for creating mobile Augmented Reality applications [Jun15]. However, there are only very few framworks for the Windows Phone platform. One that is quickly found is the *Geo AR Toolkit* (GART) for Windows Phone and Windows 8, designed by the Microsoft employee Jared Bienz [Bie15]. It was originally designed for Windows Phone 7.5 and needs to be manually ported to Windows Phone 8.1 in order to work properly. GART shows some similarity to AREA by including a heading indicator, a map and camera view. Well known AR apps on Windows Pone are Here City Lens [B.V15] or Yelp [Yel15], both showing public POIs, such as restaurants, drugstores or bars.

3.3. AREA-related work

This work is mainly based on the existing work on AREA, elaborated by [GPSR13], [GSP+14] and [SPSR15]. Since the AREA engine has already been developed for iOS and Android, this work focuses on the porting of the engine to the Windows Phone platform. In the process of doing so, the engineering process is outlined and special emphasis is placed on the various differences between the Windows Phone version and the iOS and Android counterparts. See chap 6 for more information on this.
[GPSR13] gives profound insight into the engineering process of AREA. At first, the requirements of the AREA engine are thoroughly elaborated and the engineering process is described in general. The formal foundation is also part of this work, the respective formulas that are used throughout all implementations of AREA can be viewed in the appendix A. This also applies to the common architectural design of the engine. The unique characteristic of giving insight into the actual engineering and implementation on the respective platform is part of all AREA-related work. Thus, the *Model View Controller* pattern provides the common basis. [GPSR13] also includes a survey in order to evaluate AREA. It covers general questions about the use of the participants' smartphones and their knowledge about Augmented Reality. Furthermore, the usability and quality aspects of the individual features of AREA are surveyed. The outcome shows that AREA is perceived as positive and intuitive. The architecture of an application on top of AREA is briefly discussed, followed by some details on the engineering process and its evaluation in regard to some self-penned topics. Finally, the integration of AREA into *LiveGuide Ditzingen* app is described. [GSP+14] contains a more general discussion on AREA and points out challenges, examples and lessons learned. The differences between the iOS and Android implementations are also discussed. Finally, [SPSR15] does that in a similar fashion.

4

Requirements

The requirements of AREA on Windows Phone are identical to the ones that were originally elaborated in previous works [GPSR13]. Several functional and non-functional requirements have to be met in order to deliver the full experience of AREA. The validation of these requirements is discussed in chapter 8. The functional requirement of the utmost significance is the provision of POIs on top of the camera view. It is closely linked to the requirement of determining the visibility of the POIs, which is dependent on the smartphone's current visible field of view. If the engine has decided to place a POI on the camera view, further real-time updates of the POIs are necessary. This functionality delivers the very basic user experience of AREA, while requiring a major part of the engineering effort. In order for this to work, another fundamental requirement is the sophisticated use of the built-in sensors and the application of mathematical formulas (see appendix A). A map view offers another useful UI perspective, which is detached from the augmented camera view. Providing an adjustable radius, within which the relevant POIs are to be included in the basic calculations, is an additional means of making the AREA engine more useful to the user. In this regard, a radar on top of the camera view and flyouts for showing related information of POIs when touching them add some extra usability. The AREA engine shall support a portrait and a landscape mode for showing POIs on the display when the smartphone is in oblique position. As for non-functional requirements, the AREA engine aims for efficient and accurate calculations as well as for overall stability. In addition to that, the efficiency of screen drawing shall not be neglected. Maintenance support shall complete the non-functional requirements. Implementation aspects, such as the consistent and extendable specification of POIs plus the easy integration into other applications, are also to be ensured. Tab. 4.1 summarizes all requirements.

Table 4.1.: The requirements of AREA (adjusted to Windows Phone) [GPSR13]

Requirement	Type
Provide POIs on camera view	functional
Provide POIs on map view	functional
Enable POIs on camera view only if they are inside the visible field of the user's view	functional
Provide that POIs on camera view and on map view react to touch events	functional
Read sensors to determine position of the smartphone (i.e. accelerometer, magnetometer, inclinometer, compass and GPS sensor)	functional
Update data and POI in real-time while all possible movements	functional
Provide adjustable radius for the distance of viewable POIs	functional
Provide a radar in camera view showing POIs in the environment and inside the radius	functional
Provide additional information according to touch events	functional
Support portrait mode and landscape mode	functional
Provide efficiency of calculations	non-functional
Provide accuracy of calculations	non-functional
Provide efficiency of screen drawing	non-functional
Provide stability	non-functional
Support maintainability	non-functional
Provide consistent specification of POIs	implementation
Provide that POIS can be easily extended	implementation
Provide easy integration into other applications (modularity)	implementation

5 Architecture

In this section, the architecture of AREA is presented and described. The engine is composed of four main modules, based on the *Model View Controller* (MVC) architectural pattern. This modular design is essential to the core of AREA, as it enables both exchangeability and extensibility [GSP+14] of the main components. The key advantage of this architectural design is not only the possibility of exchanging and extending the modules within the AREA engine itself, but also across several developer platforms and programming languages. This is a fundamental trait of the AREA engine, that has proven its absolute advantage in this work, when porting the app to the Windows Phone platform. The following sections will explain this in more detail.

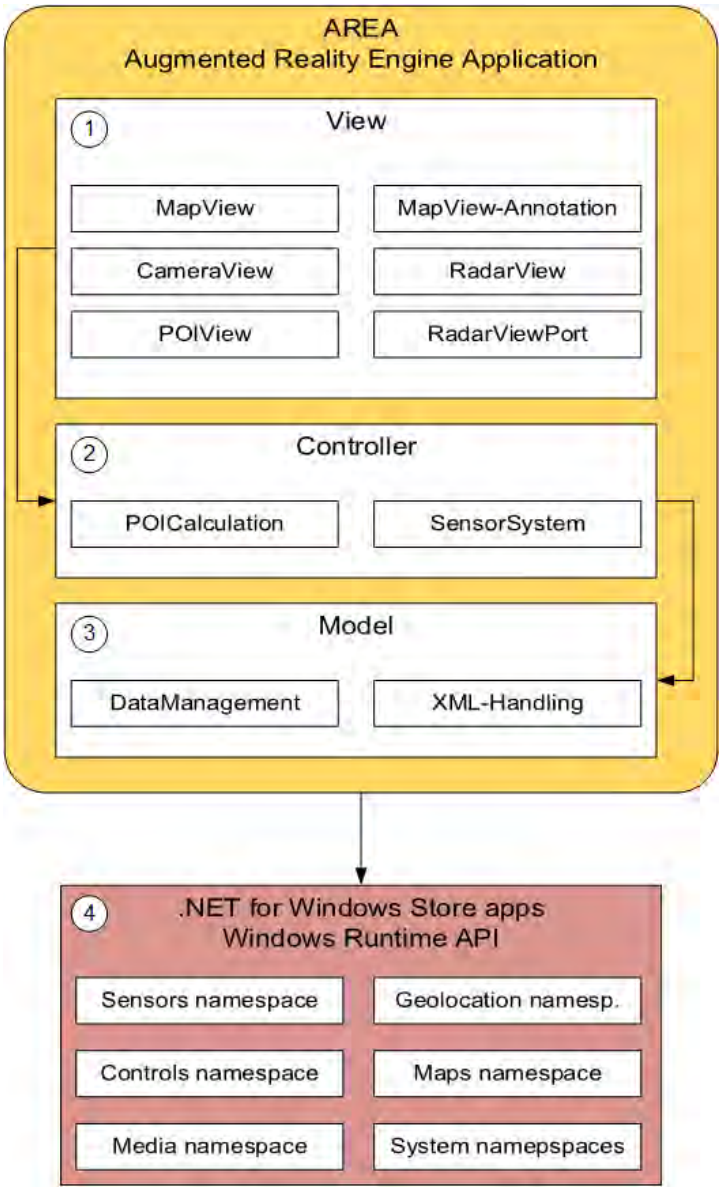
5.1. The core of AREA

The core of AREA comprises the main modules that are part of the AREA engine. It is part of all implementations of AREA so far, no matter what platform they're engineered for. This includes the Windows Phone version that has been developed within the context of this work. In order to comply with the MVC pattern, upper tiers can only access the functionality of lower tiers by using the respective interfaces. Processing any information and viewing it to the user is only possible when there is access to the relevant data. This is ensured by the *Model*, which handles the management of data by providing a base class for POIs, a store where they can be saved, an XML-schema and the respective XML-parser for loading POIs. The *Controller* manipulates the model's state. With the help of the queried sensor data and the formulas, which were both mentioned in chapter 2, the visibility and position of the POIs can be calculated and saved. In turn, the Controller is notified about any changes of the Model, the smartphone's position or its attitude. In accordance with the passed data, the *View* can then be updated. This includes the POIs visibility state and position on the screen or other UI elements, such as the radar view and the radar view port. The modular structure of AREA allows a third party developer to add intra-core extensions or customize the main modules at his leisure. Fig. 5.1 gives an overview of the AREA architecture on Windows Phone.

5.2. Windows SDK and Assemblies

The Windows Software Development Kit (SDK) for Windows 8.1 comes with the *.NET for Windows Store apps* Framework, which is a subset of managed types that can be used to create Windows Store apps [MSD15d]. The *Windows Runtime API* complements this subset with

Figure 5.1.: Multi-tier architecture of AREA [GPSR13]



additional types and is part of the Windows Phone 8.1 framework. Both frameworks provide the set of assemblies that are needed for app development on the Windows 8.1 platform. This module is also included in Fig. 5.1. The AREA engine is accessing the types that reside in the respective namespaces to make use of the native members, such as classes and interfaces.

5.3. Platform-specific extensions

The MVC pattern allows some platform-specific extensions, since the main modules are not affected by additional assemblies. In this work, a maintenance assembly has been added. It contains a simple background uploader task that is able to upload log files to an ASP.NET web application. This is a useful solution for accessing any log files that are created with the Diagnostics API of Windows [MSD15b]. When the app crashes because of an unhandled error, the entire log data of a logging session is stored in the application's local folder. Upon the next execution of the app, the *LogUploaderTask* is triggered every 15 minutes and uploads all log files that have been previously saved just before crashing. This functionality has been tested with a simple ASP.NET web application running on Internet Information Services Express (IIS Express) that comes with Visual Studio 2013 for testing purposes. Note that a real production web server would require some adaptions to the uploader task, such as the usage of user credentials and a custom URI that points to the ASP.NET application.

5.4. Platform-specific exchanges

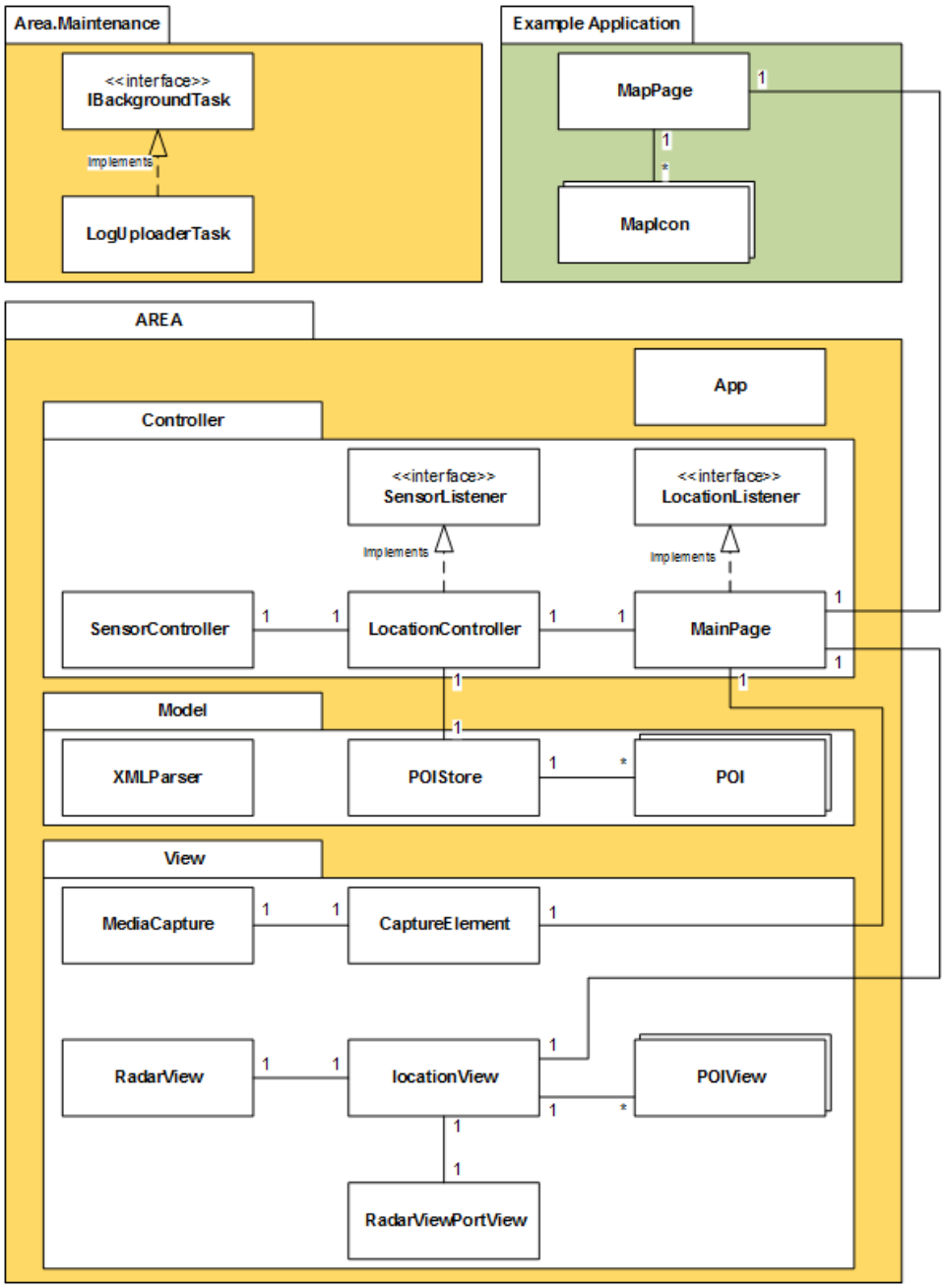
The characteristics of a given platform require replacements or changes to code that had originally been developed for another platform. When porting the AREA engine to Windows Phone, there certainly had to be made some significant adaptions. This is due to the mere fact that iOS or Android libraries are not available on Windows Phone. These exchanges require different approaches of implementation, in order to reproduce the same functionality and the almost identical look and feel of AREA. Chapter 6 gets in to more detail in that regard.

5.5. Class structure

The class structure of the Windows Phone version of AREA is very similar to its siblings on iOS and Android. The app page model allows easy integration into existing applications, as a simple page navigation call is sufficient for AREA to initialize the required resources and components. As a consequence of this page navigation, the `MainPage` is set as the content of the `Frame` object, which is created in the code-behind file of the `App` class. Note that the `App` class also handles the saving process of log files upon crashing and the registering of the `LogUploaderTask` (refer to section 5.3). Since it is the entry point of a Windows Store app, it should be customized for the existing application that integrates the AREA engine. The `SensorController` class is responsible for reading the various sensors of the smartphone. After being instructed to start collecting sensor data by the `LocationController`, it notifies the same whenever the GPS-position or the readings of the inclinometer, compass, accelerometer or magnetometer change. In order to receive any of the mentioned events, the `LocationController` implements the `SensorListener` interface. Using the sensor data along with the `POIs` in the `POIStore`, it can then perform the necessary calculations for determining the visibility and position of the `POIs` to be displayed. As the `MainPage` implements the `LocationListener` interface, it gets notified about any changes to the `POIs` and the smartphone's attitude and position. With the passed data the `MainPage` can update the UI by placing the respective `POIViews` on the `locationView` and by updating

the `RadarView` and the `RadarViewPortView`. The `CaptureElement` control is provided with a `MediaCapture` element by a helper class and provides functionality for previewing the camera's video stream. In contrast to the iOS counterpart, the `CaptureElement` does not have an overlay property, which can hold custom sub-views [GPSR13]. Instead, a simple `locationView` Grid control contains the respective POIs. All XAML controls are added to the `ContentPanel`, the XAML root element of the `MainPage`. The model includes the `POIStore`, which contains the `POIs` that were parsed from an XML file with the help of the `XMLParser`. The `MapPage` contains a `MapControl` and `MapIcons` to represent the surrounding POIs. Similar to the Android intent practice, the page navigation to the `MainPage` can be performed from here. The AREA engine sets up everything else autonomously from there on. Fig. 5.2 shows the associated class structure. Detailed class diagrams are included in the appendix B.

Figure 5.2.: Class structure of AREA and an application on top [GPSR13]



6 Aspects of implementation

The major part of this work is discussed in this chapter. It's been a considerable amount of work to make the preparations of porting the AREA engine to the Windows Phone 8.1 operating system. It began with exploring the platform basics of the Windows Universal app model, which is currently undergoing a change with the emerging Windows 10 platform. Microsoft has been striving for a truly converged developer platform, wherein targeting a single operating system with a single app constitutes the ultimate goal of universality. By providing about 90% API convergence, Windows 8.1 has been a major step forward into the right direction. Eventually, this goal is achieved with the new *Universal Windows Platform* (UWP) for Windows 10. However, this work started with Windows 8.1 as the prevailing platform for Windows apps. Therefore, the implementation focuses on a universal Windows app for Windows Phone 8.1. The notion of porting the app to Windows 10 could provide the basis of future work, which is discussed in chapter 9.

6.1. Platform mappings

The AREA engine is available on two other platforms. When porting the app to Windows Phone, one has to analyze the already available code and understand the characteristics of the underlying platform. This section will give a brief overview on key similarities and differences between Windows Phone, Android and iOS.
By taking a look at the runtime technologies of the mobile operating systems, the affinity between Android and Windows Phone can be easily detected. Android applications are executed in the *Dalvik Virtual Machine* (VM), a managed runtime environment that is set up for each application. The Dalvik core class library ensures a developer experience that is similar to Java Standard Edition (SE). However, the latter is using a stack-based *Java Virtual Machine* (JVM), whereas the Dalvik VM is register-based and optimized for running on mobile devices [Inc15c]. More Java platform technologies, such as a subset of Java libraries and the programming language Java, complement the Android Runtime. The successor to the Dalvik VM is the *Android Runtime* (ART, not to be confused with the common terminology when referring to the Android Runtime in general), which introduced some major improvements, including Ahead-of-Time (AOT) compilation and an advanced garbage collection (GC) [Inc15b]. Windows Phone 8.1 is making use of the *Common Language Runtime* (CLR) of the .NET Framwork. Just like the Dalvik VM, it uses Just-in-Time (JIT) compilation and runs a Windows Phone application in a *Sandbox* environment [MSD15a]. Android and Windows Runtime APIs provide access to packages and namespaces that include graphics, media, storage, networking and sensor libraries. Apple's iOS is using the Objective-C runtime to execute applications,

without the need of any intermediate technology such as Java or .NET. Just like on Android and Windows Phone, iOS apps run in a restricted and secured Sandbox, which is not specified in greater detail. The iOS operating system is presented as a layered architecture, with Cocoa Touch being at the very top. Most objects in Cocoa Touch are subclasses of the `NSObject` class [Inc15a] and offer access to functionality that is similar to Android and Windows Runtime APIs. Frameworks organize the iOS API just like packages in Android or namespaces in Windows. Some runtime functions can be accessed in order to use C and replicate compiler-based functionality [Inc15a]. Tab. 6.1 gives a more detailed overview on the different platforms.

Table 6.1.: Platform overview and developer tools of iOS, Android and Windows Phone (8.1) [MSD15c]

	iOS	**Android**	**Windows Phone**
Runtime	Objective-C Runtime	Android Runtime	Windows Runtime
	Cocoa Touch Media Core Services Core OS	Android Libraries	WinRT Libraries Language Projections
	iOS Frameworks and System Libraries	Java Libraries	.NET Libraries
		Dalvik VM or Android runtime (ART)	.NET CLR
Kernel	XNU (Darwin)	Linux Kernel	Windows NT Kernel
Development OS	Mac Windows	Windows Mac Linux	Windows Mac
IDE	Xcode Visual Studio (2015)	Android Studio Eclipse IntelliJ NetBeans Visual Studio (2015)	Visual Studio
SDK	iOS SDK	Android SDK	Windows SDK for 8.1
Language	Objective-C Swift	Java	C# Visual Basic (VB) C++ HTML+JavaScript

The user interface mappings in Tab. 6.2 might suggest that iOS closely resembles Windows Phone. That applies to the identifiers of the UI components (e.g. Page and Control), but when it comes to implementation, Android shares many similarities with Windows Store apps. For the sake of brevity, the main focus is laid on the comparison between Android and Windows Phone from here on. To provide some level of completeness, iOS is included in some tables and might sometimes be mentioned. Before dwelling on some code to point out implementation differences, some UI basics should be discussed.

The unit of display is called *Activity* on Android and *Page* on Windows Phone. In regard of the paradigm, Activities and Pages are almost identical [MSD15c]: An app is a collection of Activity/Page objects, each designed to perform a function (sending a message or taking a photo). Only one Activity/Page is visible at a time, with the exception that Activities can be invisible or used for non-interactional purposes. An Activity/Page is a collection of layout controls and widgets (e.g. buttons, images or text boxes). Both Activities and Pages are added to some sort of stack. Activity objects are added to a back stack, which consists of cohesive task units. A task is a collection of activities that are related to each other by the job they are meant to perform [Inc15d]. Pages are added to a navigation stack that is not available across the entire system, but within each app. The layout of an Activity is defined in .xml files, the runtime code is in a .java file. Pages have their layouts defined in .xaml files, with the code being in .cpp/.cs/.vb files (.cs in the case of AREA). Event handling can be used by developers in similar fashion, though Windows Store apps make use of .NET delegates. Note that in Windows Store apps, control references are generated automatically and can be accessed directly in runtime code files, whereas in Android a widget needs to be explicitly located by using the `findViewById` method of the Activity class. The use of composite UI is essential to both platforms. Android UI consists of compound components and fragments, while Windows Store apps use child objects of control-based classes.

Table 6.2.: User interface mappings of iOS, Android and Windows Phone (8.1) (cf. [MSD15c])

	iOS	**Android**	**Windows Phone**
Unit of Display	Page	Activity	Page
Widget Base	Control	View	Control
Layout Base	Collections Containers	ViewGroup	Panel
Event Handling	Delegates	Observer interface	Delegates
UI Components	Child objects of UIControl	Compound Components and Fragments	Child objects of Control

6.2. AREA related task mappings

In order to address the correct APIs and namespaces, taking a look at the task mappings is of great advantage. In this section, the use of the APIs on Android and Windows Phone are pointed out by means of the actual implementation. The implementation differences are organized by the MVC components, starting with small differences and going over to significant variations.

6.2.1. Model

The Syntax of the Java and C# programming languages are very similar. Many Model classes in AREA on Windows Phone are not substantially different from their Android counterparts.

However, one distinct feature of C# is implemented throughout the entire app: Properties. In Java, accessor and mutator methods need to be implemented explicitly. In the following Java example, the private field variable `kDeviceWidth` shall be publicly accessible, but only the class that holds it shall be able to change its value:

Listing 6.1: Java Accessor and mutator methods

```
private static double kDeviceWidth;

//getter
public static double getkDeviceWidth() {
                return kDeviceWidth;
}

//setter
private static void setkDeviceWidth(double kDeviceWidth) {
                AREAConstants.kDeviceWidth = kDeviceWidth;
}
```

In C#, this can be implemented in a much more elegant way, using *Auto Properties*. The private field doesn't have to be declared explicitly, because the compiler handles that. By using properties, classes with many fields look much less cluttered and get along with few lines of code:

Listing 6.2: C# Field Property

```
public static double KDeviceWidth { get; private set; }
```

Properties are also very useful when using *Singleton* classes. Singleton classes make sure that only one object of that class is instantiated and globally accessible. Note that this time the private field is declared explicitly in order to handle the lazy instantiation. The setter is omitted:

Listing 6.3: C# Properties in Singleton classes

```
private static AREAStore instance;

//The Instance property that handles the lazy instantiation.
public static AREAStore Instance
{
    get
    {
        if (instance == null)
            instance = new AREAStore();
        return instance;
    }
}
```

Other classes just need to reference the `AREAStore.Instance` in order to call the `get` method of the property.

6.2.2. View

In Android, custom UI components are implemented by extending the `View` class. It is the base class for widgets and can be used to define new interactive UI components. If a widget shall contain children views, the custom class needs to derive from the `ViewGroup` class. The correct mapped class in Windows Phone would be `UIEelement`. Although it serves as a base class for most Windows Runtime UI objects, custom controls are not derived from it [MSD15f]. Instead, the `UserControl` offers functionality for defining new controls that encapsulate existing controls and provide its own logic.
An important compound UI component in AREA is the `AREAPointOfInterestView`. An object of this class represents a single POI on the screen. Comparing the constructors between the Android and Windows Phone version of the class already reveals some of the few, but defining differences:

Listing 6.4: Java The constructor in `AREAPointOfInterestView`

```
1  public class AREAPointOfInterestView extends View {
2          ...
3          public AREAPointOfInterestView(Context context,
               AREAPointOfInterest poi) {
4                super(context);
5                this.context = context;
6                this.poi = poi;
7                init();
8          }
9          ...
10 }
```

In Android, every `View`-based class needs to claim a `Context` in its constructor in order to have access to the applications current resources and classes. This is often needed when manipulating other views, for example when updating a `TextView` in consequence of an `onClick` event in a Button. In such cases, the respective event handler of the Button needs to use the `findViewById()` method, which only works if the Button was instantiated with the right context (i.e. the Activity that holds the TextView). As mentioned above, the references to other XAML controls are created automatically in Windows Phone. The constructor in Windows Phone is therefore implemented like this:

Listing 6.5: C# The constructor in `AREAPointOfInterestView`

```
1  public class AREAPointOfInterestView : StackPanel
2  {
3          ...
4          public AREAPointOfInterestView(AREAPointOfInterest poi) : base()
5          {
6                  Poi = poi;
7                  init();
8          }
9          ...
10 }
```

Note that in the implementation for Windows Phone, the class extends `StackPanel`. It can also be used as a base class for derived custom classes, just like a custom `UserControl`. The `StackPanel` comes in very handy, because it can arrange its child elements into a single horizontal or vertical line. This is needed for the appearance of a POI, which consists of a circle and a text block right underneath it.
It becomes apparent that the absence of a context-like object passed around in UI elements is of low importance for Windows Phone, when examining how it handles scaling to pixel density compared to Android. With the automatic scaling of Windows Runtime apps, the developer doesn't have to deal with manual pixel density calculations. In Android, the display metrics need to be queried from the context in order to get the density dots per inch (DPI). This value can then be used to perform density-pixel-to-pixel or pixel-to-density-pixel calculations to ensure that the UI proportions look about the same on every Android device. Windows Phone 8.1 apps use a scaling system based on the screen properties of the device (e.g. screen size, resolution and dpi) and apply a three-stage scaling factor whenever the UI elements would be too small for user interaction. By building the UI with XAML and paying attention to some dos and don'ts, the automatic scaling takes place and the developer is exempted from further actions [MSD15e].
When creating custom view classes in Android, the use of a `Canvas` object is needed. For example, in the `onDraw` method of the earlier mentioned `AREAPointOfInterestView` class, the `Canvas` is responsible for drawing the elements that are contained in a POI view. The drawing calls are performed by using previously defined `Paint` objects, such as circles and rectangles. The following code is showing the required implementation steps for including a circle in the POI view:

Listing 6.6: Java Sub-components of the `AREAPointOfInterestView` class

```
1  public class AREAPointOfInterestView extends View {
2          ...
3          private void init() {
4                  ...
5                  circleFill = new Paint();
6                  circleFill.setFlags(Paint.ANTI_ALIAS_FLAG);
7                  circleFill.setColor(Color.argb(179, 249, 196, 49));
8
9                  circleStroke = new Paint();
10                 circleStroke.setStyle(Paint.Style.STROKE);
11                 circleStroke.setStrokeWidth(strokeWidth);
12                 circleStroke.setFlags(Paint.ANTI_ALIAS_FLAG);
13                 circleStroke.setColor(Color.argb(255, 25, 25, 25));
14                 ...
15         }
16         ...
17
18         @Override
19         protected void onDraw(Canvas canvas) {
20                 super.onDraw(canvas);
21                 canvas.drawCircle(width / 2.0f, circleSize / 2.0f +
                       dpToPixel(2),
22                                 circleSize / 2.0f, circleFill);
```

```
23                  canvas.drawCircle(width / 2.0f, circleSize / 2.0f +
                        dpToPixel(2),
24                                  circleSize / 2.0f, circleStroke);
25                  ...
26          }
27  }
```

The custom `StackPanel`-based implementation of the `AREAPointOfInterestView` class in Windows Phone doesn't have an `onDraw` method or anything alike. Instead, XAML handles the drawing process automatically when a `Control` or `Shape` is added to the `StackPanel`. In this case, a simple `Ellipse` is sufficient:

Listing 6.7: C# Sub-components of the `AREAPointOfInterestView` class

```
1   public class AREAPointOfInterestView : StackPanel
2   {
3           ...
4           private void init()
5           {
6                   circle = new Ellipse();
7           circle.Height = circleSize;
8           circle.Width = circleSize;
9           circle.StrokeThickness = strokeWidth;
10          circle.Fill = new SolidColorBrush(Color.FromArgb(179, 249, 196,
                49));
11          circle.Stroke = new SolidColorBrush(Color.FromArgb(255, 25, 25,
                25));
12
13                  this.Children.Add(circle);
14          }
15          ...
16  }
```

6.2.3. Controller

Initializing the sensors in Windows Phone is a rather quick task. In the `AREASensorController`, the sensor classes that reside in the Sensors namespace and were discussed in chapter 2, can be referenced in order to call the `GetDefault()` method. There is no need for a `SensorManager` to access the sensors, which is yet again obtained through a context object in Android. The `Geolocator` is an exception, as it needs to be created and gets its `DesiredAccuracyInMeters` and `ReportInterval` properties assigned in the process of doing so. As to the other sensors, the requested report interval of 16 milliseconds is compared to the minimal supported interval. The code sets the requested interval if the minimum supported one is not greater than it. When debugging the code on the Lumia 735, it became apparent that apart from the accelerometer, all other sensors support a minimum report interval of exactly 16. This approach is also suggested by the documentation of the Windows Runtime API.

Listing 6.8: C# Initializing the sensors

```
...
private readonly Nullable<uint> desiredAccuracyInMetersValue = 5;
private const uint reportIntervalValue = 500;

...
public AREASensorController()
{
        ...
        // Create sensors.
        accelerometer = Accelerometer.GetDefault();
        compass = Compass.GetDefault();
        inclinometer = Inclinometer.GetDefault();
        magnetometer = Magnetometer.GetDefault();

        geoLocator = new Geolocator
        {
           // Create Geolocator with tracking accuracy and interval.
           DesiredAccuracyInMeters = desiredAccuracyInMetersValue,
           ReportInterval = reportIntervalValue
        };

        // Establish the report interval for each sensor.
    if (accelerometer != null)
    {
        uint minReportInterval = accelerometer.MinimumReportInterval;
        uint reportInterval = minReportInterval > 16 ? minReportInterval :
            16;
        accelerometer.ReportInterval = reportInterval;
    }

    if (compass != null)
    {
        uint minReportInterval = compass.MinimumReportInterval;
        uint reportInterval = minReportInterval > 16 ? minReportInterval :
            16;
        compass.ReportInterval = reportInterval;
    }

    if (inclinometer != null)
    {
        uint minReportInterval = inclinometer.MinimumReportInterval;
        uint reportInterval = minReportInterval > 16 ? minReportInterval :
            16;
        inclinometer.ReportInterval = reportInterval;
    }

    if (magnetometer != null)
    {
        uint minReportInterval = magnetometer.MinimumReportInterval;
```

```
47          uint reportInterval = minReportInterval > 16 ? minReportInterval :
                16;
48          magnetometer.ReportInterval = reportInterval;
49      }
50  }
```

When starting the sensors, the delegates - in this case `TypedEventHandlers` - are attached to the `ReadingChanged` and `PositionChanged` events, in order to get any updates from the sensors. When obtaining the position for the first time, the asynchronous programming in Windows Store apps is used. By calling `await geoLocator.GetGeopositionAsync()` in order to get the current position, the responsiveness of the App is maintained. The asynchronous method `startSensoring()` returns a `Task<bool>`, which handles the asynchronous operation of obtaining the position in the background, without blocking the UI thread.

Listing 6.9: C# Start sensoring

```
1  public async Task<bool> startSensoring()
2  {
3      channel.LogMemberName();
4
5      // Assign event handlers for the reading, position and status changed
           events.
6      accelerometer.ReadingChanged += new TypedEventHandler<Accelerometer,
           AccelerometerReadingChangedEventArgs>(accelerometer_ReadingChanged)
           ;
7      compass.ReadingChanged += new TypedEventHandler<Compass,
           CompassReadingChangedEventArgs>(compass_ReadingChanged);
8      inclinometer.ReadingChanged += new TypedEventHandler<Inclinometer,
           InclinometerReadingChangedEventArgs>(inclinometer_ReadingChanged);
9      magnetometer.ReadingChanged += new TypedEventHandler<Magnetometer,
           MagnetometerReadingChangedEventArgs>(magnetometer_ReadingChanged);
10     geoLocator.PositionChanged += new TypedEventHandler<Geolocator,
           PositionChangedEventArgs>(geoLocator_PositionChanged);
11     geoLocator.StatusChanged += new TypedEventHandler<Geolocator,
           StatusChangedEventArgs>(geoLocator_StatusChanged);
12
13     // Get current position.
14     currentPosition = await geoLocator.GetGeopositionAsync();
15
16     notifyPositionChanged();
17
18     channel.LogMessage("Sensoring started ...");
19
20     return true;
21 }
```

Data Binding is very frequently used in XAML programming. It enables a simple way of displaying the underlying data of a XAML control, while the properties get altered by the logic of the application. When the bound data changes, the update of the XAML control is handled automatically. This functionality can further be customized by setting the mode of binding, which determines whether the respective target property is updated once (upon creation), whenever

the source changes (one way) or when either target and source should be updated when either changes. In this case, using the slider will change the `Radius` property of the location controller (lc) and any update to the `Radius` property will propagate to the slider.
When setting up the `MainPage`, instances of the `CameraCapture` helper class and the `CaptureElement` XAML control are created. The latter renders the camera stream , while the former helps initializing camera resources and start showing the preview stream.

Listing 6.10: C# Bindings and camera preparations

```
public MainPage()
{

        ...
        // Set up radius slider. Bind Radius to it.
    radiusSlider.Maximum = (int)AREAConstants.KMaxDistance - AREAConstants
        .KMinDistance;
    Binding binding = new Binding
    {
        Source = lc,
        Path = new PropertyPath("Radius"),
        Mode = BindingMode.TwoWay
    };
    radiusSlider.SetBinding(Slider.ValueProperty, binding);

    ...
    // Get an instance of CameraCapture helper class and set up the
        capturePreview.
    cameraCapture = new CameraCapture();
    capturePreview = new CaptureElement();

    ...
}
```

The following code shows how the camera is initialized when navigating to the camera view (i.e. the `MainPage`). The release of the camera resources when navigating from the page is important, otherwise other applications won't be able to access the camera. Note how the camera view registers itself for future updates from the location controller by calling `lc.registerListener(this)`. The interface methods are implemented in order to get notified about the calculations of the controller. The UI, in particular the POIs, can then be updated. Starting and stopping the reading of the sensors is also handled here.

Listing 6.11: C# Camera initialization and release

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    // Hide navigation and status bar
    //ApplicationView.GetForCurrentView().SuppressSystemOverlays = true;
    await StatusBar.GetForCurrentView().HideAsync();

    // Initialize camera resources and show preview.
    cameraCapture = new CameraCapture();
    capturePreview.Source = await cameraCapture.InitCaptureResources();
```

```
10      await cameraCapture.StartPreview();
11
12      lc.registerListener(this);
13      lc.startUpdating();
14  }
15
16  protected override async void OnNavigatedFrom(NavigationEventArgs e)
17  {
18      // Release camera resources.
19      if (cameraCapture != null)
20      {
21          await cameraCapture.StopPreview();
22          capturePreview.Source = null;
23          cameraCapture.Dispose();
24          cameraCapture = null;
25      }
26
27      lc.stopUpdating();
28      lc.unregisterListener(this);
29  }
```

Implementation in regard of the appliance of mathematical calculations (e.g. Haversine-formula or other trigonometrical calculations) can be viewed in [GPSR13] in more detail. There is a neglectable difference between the code that performs the calculations on iOS, Android and Windows Phone. Therefore, it has been omitted in this chapter. For the sake of completeness, the utilized formulas can be viewed in the appendix A.

In summary it can be said that the implementation approach is very similar to Android, as the paradigms are about the same. However, when getting into the details, many differences can be detected. The Windows Phone platform handles various things automatically and liberates the developer from legacy-like tasks that can be found in Android. On top of that, the API seems more streamlined and easier to use, resulting in easy-to-understand and shorter code. On the other hand, it feels like Android gives more freedom over some implementation details. After all, Windows is a proprietary platform and provides no insight into the actual implementation of classes and their functions. Altogether, the potential of the Windows platform is vast and is about to reach its climax with Windows 10 right around the corner.

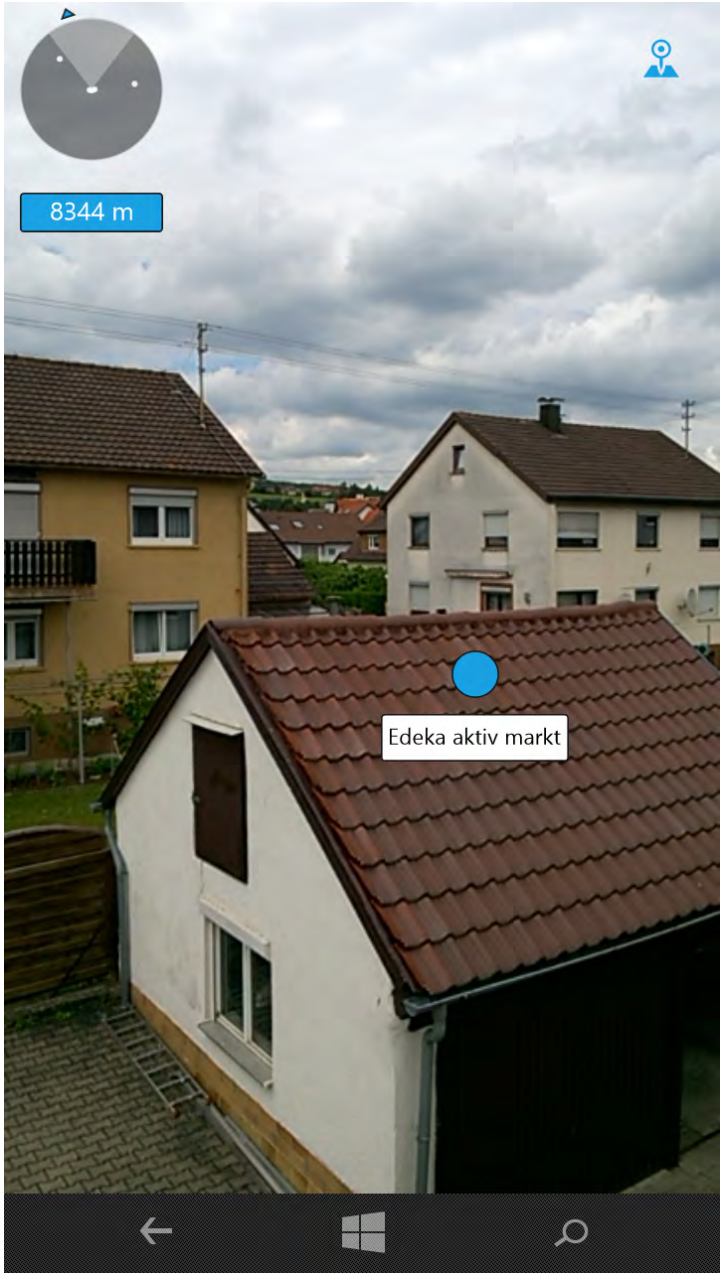
7

Introduction to the application

This chapter gives a short introduction to the actual UI of the AREA engine, running on a Nokia Lumia 735 with Windows Phone 8.1 Update.

When starting AREA, the camera view is opened in order to show surrounding POIs. In Figure 7.1, AREA shows a single POI with a circle and a text label underneath it. The radar view in the top left corner indicates that there are more POIs within the current radius of 8344 meters. A little triangle-shaped indicator moves around the radar view and points northwards. Navigating to the map view is done by tapping the map icon on the top right corner.

Figure 7.1.: AREA's camera view with a single POI

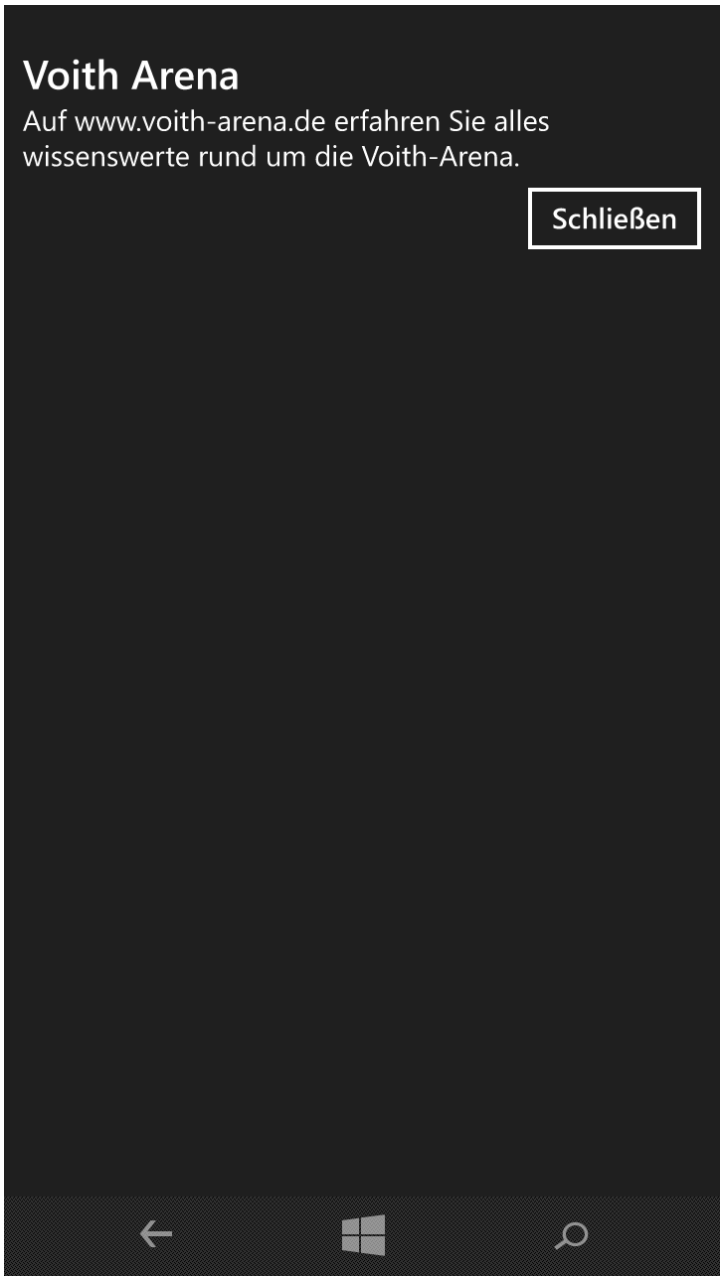


When turning the device to left side, three other POIs show up on the screen. The radar indicates that one is rather far away ("Voith Arena"), while the other two must be very close ("Vergölst Partnerbetrieb" and "Autohaus Penka"). To be exact, "Vergölst Partnerbetrieb" is even closer, because it overlays the other POI.

Figure 7.2.: AREA's camera view with two POIs

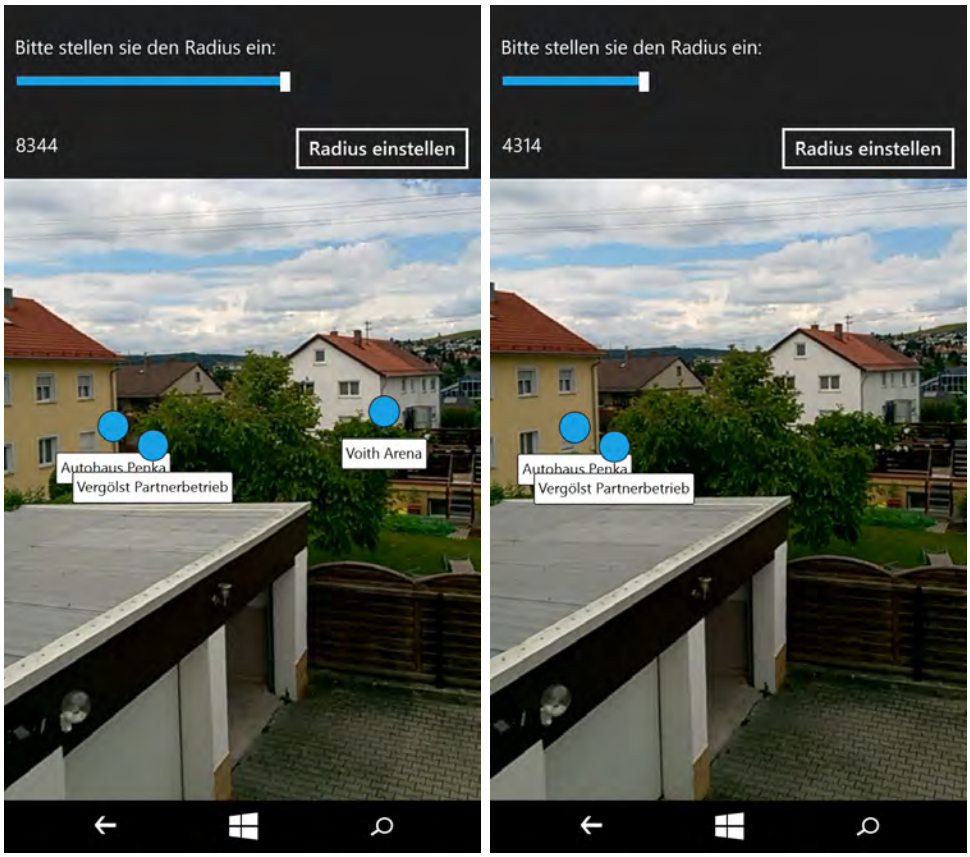
Tapping on a POI shows a full-screen flyout with related information. This flyout can be customized with other XAML controls, for example with images, map controls or other text blocks.

Figure 7.3.: Information about a POI shown in a customizable flyout



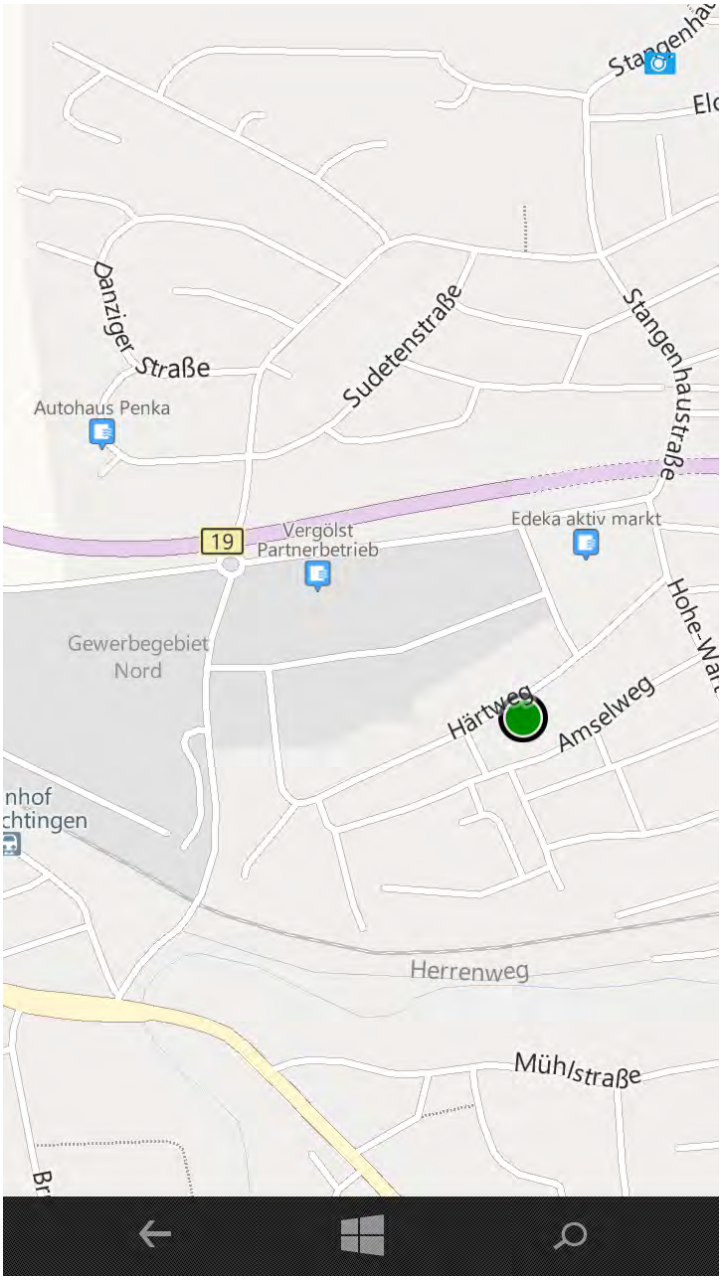
By tapping on the distance view right below the radar, the user can change the radius setting. Any change to the radius immediately affects the visible POIs on the screen. When going down with the radius, the "Voith Arena" disappears right away. Setting the radius higher makes it pop up again. The update of the radius is immediately applied, the button "Radius einstellen" only closes the flyout.

Figure 7.4.: Setting the radius in AREA



The map view can be reached by tapping the map icon (Figure 7.1). A green circle indicates the user's position, while standard map icons mark the surrounding POIs. The set of POIs is passed as an argument of the navigation call to the map view. Tapping the back button on the bottom or the camera icon in the top right corner will get the user back to the camera view.

Figure 7.5.: AREA's map view



8

Reconciliation of requirements

An important task of software engineering is the reconciliation of requirements. In chapter 4, the requirements of AREA have been introduced. This chapter is meant to check whether the requirements are met in general. Pointing out what really works as intended is as important as focusing on remaining issues and challenges. Before going into detail about the latter, the basic functionality of AREA is addressed. The current version of AREA on Windows Phone can show POIs on a camera and map view. POIs are only visible to the user, when they are located in the same direction and within the boundaries of the location view. Tapping on a POI will trigger an event, which currently shows a flyout with additional information. However, code can be added in order to adapt the functionality of the event handlers to the current needs. The engine can read the sensors of the device and update both data and UI in accordance with the provided values. Providing some UI elements to help the user making the most out of the AREA engine, such as a radar view and an adjustable radius rounds up the basic experience. However, there is one feature that is not working as desired in the current iteration of AREA on Windows Phone. While the portrait mode is working correctly, several issues arised when implementing the landscape mode of the engine. A solution to this bug is being investigated, but it is unlikely that an update can be delivered before the end of this work. The next few lines describe the issues in regard of that matter.
The `CaptureElement` that renders the camera's video stream is a XAML control. When rotating the device towards landscape orientation, the `CurrentOrientation` jumps to landscape at some point. This causes all XAML controls to rotate and rearrange, including the camera view. The latter is not rotated correctly and renders the camera stream with an incorrect orientation. There are many solutions to this, such as locking the display orientation to landscape. However, these solutions require the code for drawing the POIs to be adapted. This is because the reference axes of some controls change upon rotating and the sensor data needs to be remapped. The next iteration of the engine shall include this functionality.
Other non-functional requirements, such as efficiency and accuracy, can be further improved for devices with lower processing power. Running the AREA engine on a high-end device, such as the Nokia Lumia 930, makes a significant difference when compared to the main device used for this work (Lumia 735). The code can easily be maintained with future updates and the modular design allows additional extensions to the POIs and other components.

9 Summary

The development of AREA on Windows Phone has been a challenging task. Despite the availability of some major groundwork [GPSR13] [GSP+14], acquiring the knowledge about the Windows Phone 8.1 platform was essential to the successful porting of the app. As the work progressed in its early stage, this turned out to be no easy task. Microsoft is migrating the API documentation for Windows apps to the new Windows 10 platform. As a consequence, some MSDN articles on Windows 8.1 get removed or updated in favor of Windows 10. However, after conceiving the platform bit by bit, the vast potential of Windows Phone emerges. Microsoft's platform is open to every developer and with Windows 10 it's possible to target multiple devices with a single app. The availability of several technologies to write a Windows app enables the use of already familiar programming languages. The C# + XAML app model proved to be very appropriate for AREA. In combination with the Windows Runtime API, developing for Windows Phone turned out to be a very great experience. Many features of the platform, such as automatic scaling to pixel density, auto-references to XAML controls and an overall well-structured and intuitive API facilitate the work of the developer.
In future work, some improvements to the engine can be applied. By porting the app to Windows 10 and adjusting the code, the XAML performance can be further increased. The platform capabilities of Windows 10 also allow the app to target devices with multiple screen sizes. Additionally, the app could be localized in order to make it available in several languages. Improvements and new features for AREA, such as handling clusters of POIs in the UI [Mü14], are already discussed. Due to the modular design of AREA, these new capabilities can certainly be integrated into the Windows Phone version.
With this work, the engineering process of AREA has been elaborated on all major mobile operating systems. The location view approach ensures efficiency and accuracy when adding POIs to the screen and manipulating their position. Integrating AREA in many mobile business applications is a desirable goal in the longer term. Many cases of application can make use of the AREA engine. In particular, mobile process management and support applications in the medical field (e.g. supporting medical ward rounds with MEDo [PLRH12] [PMLR15]) can benefit from Augmented Reality, just as much as data collection scenarios with high workload [SSP+14] [SSPR15].

Appendices

A
Formulas

Haversine-formula

This formula is used to calculate the distance between the user and a given POI. The result D is given in kilometers.

$$\theta = 2 \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_A \cos\phi_B \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) D = \theta * 6371km \tag{A.1}$$

D: Distance between user and POI
A: position of the user
B: position of POI
ϕ: latitude
λ: longitude
$\Delta\lambda = \lambda_B - \lambda_A$
$\Delta\phi = \phi_B - \phi_A$

Bearing

The bearing between the user and a given POI relative to the north pole is calculated with the following formula. The result needs to be transformed with $(\theta + 360°)\ mod\ 360°$ in order to map it to the interval $0° \ldots 360°$.

$$\theta = \arctan 2(\sin(\Delta\lambda)\cos\phi_B, \cos\phi_A \sin\phi_B - \sin\phi_A \cos\phi_B \cos(\Delta\lambda)) \tag{A.2}$$

Elevation Angle

The altitude difference between user and POI is calculated using this formula. Note that $\sigma = 1$ if the POI is located higher than the user, otherwise $\sigma = -1$. The result is an angle between $-90°$ and $+90°$.

$$\theta = \sigma * \frac{180}{\pi} \arctan\left(\frac{\Delta h}{d}\right), \sigma \in \{-1, 1\} \tag{A.3}$$

---

d: Distance between user and POI $\quad$ Δh: altitude difference between user and POI

Field of View

The vertical and horizontal dimensions of the Field of View are calculated with the following formula.

$$\alpha = 2 \arctan\left(\frac{B}{2f}\right) \tag{A.4}$$

α: Vertical or horizontal Field of View angle
B: image sensor size of the smartphone's camera
f: focal length of the smartphone's camera

B

Class Diagrams

Figure B.1.: Class diagram with class relations, collapsed view. Classes below Model are auto-generated

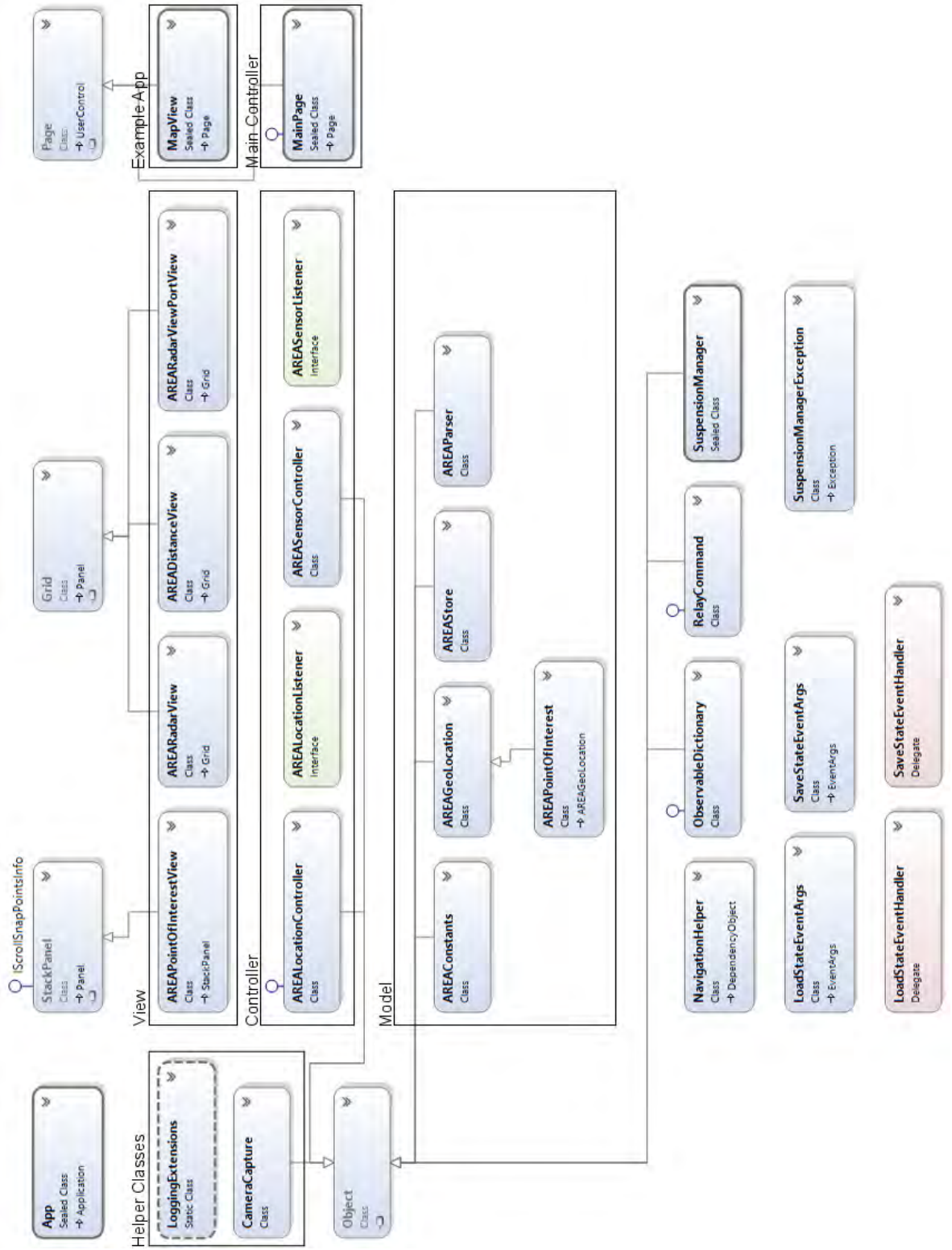


Figure B.2.: Class diagram, view expanded

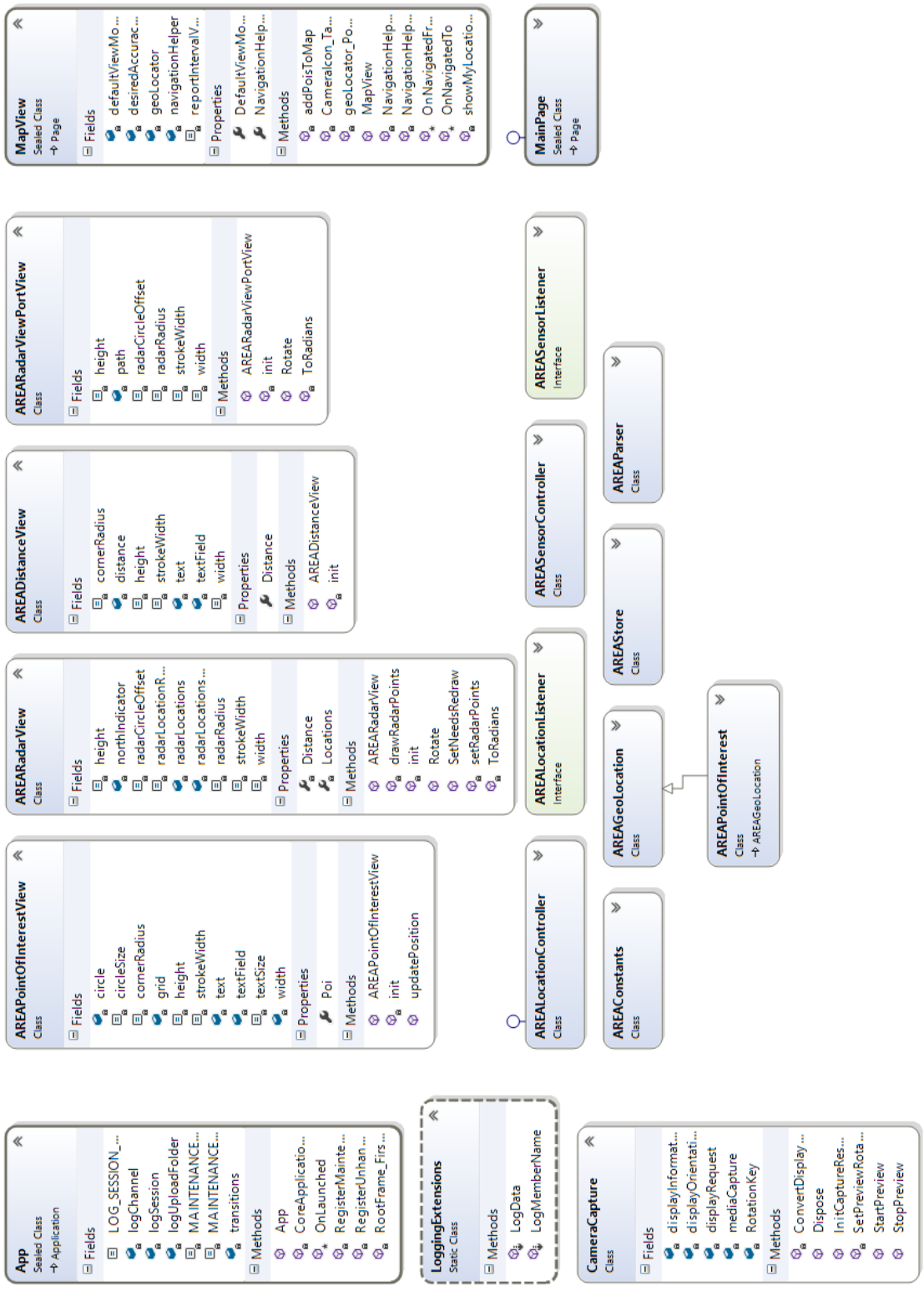


Figure B.3.: Class diagram, controller expanded

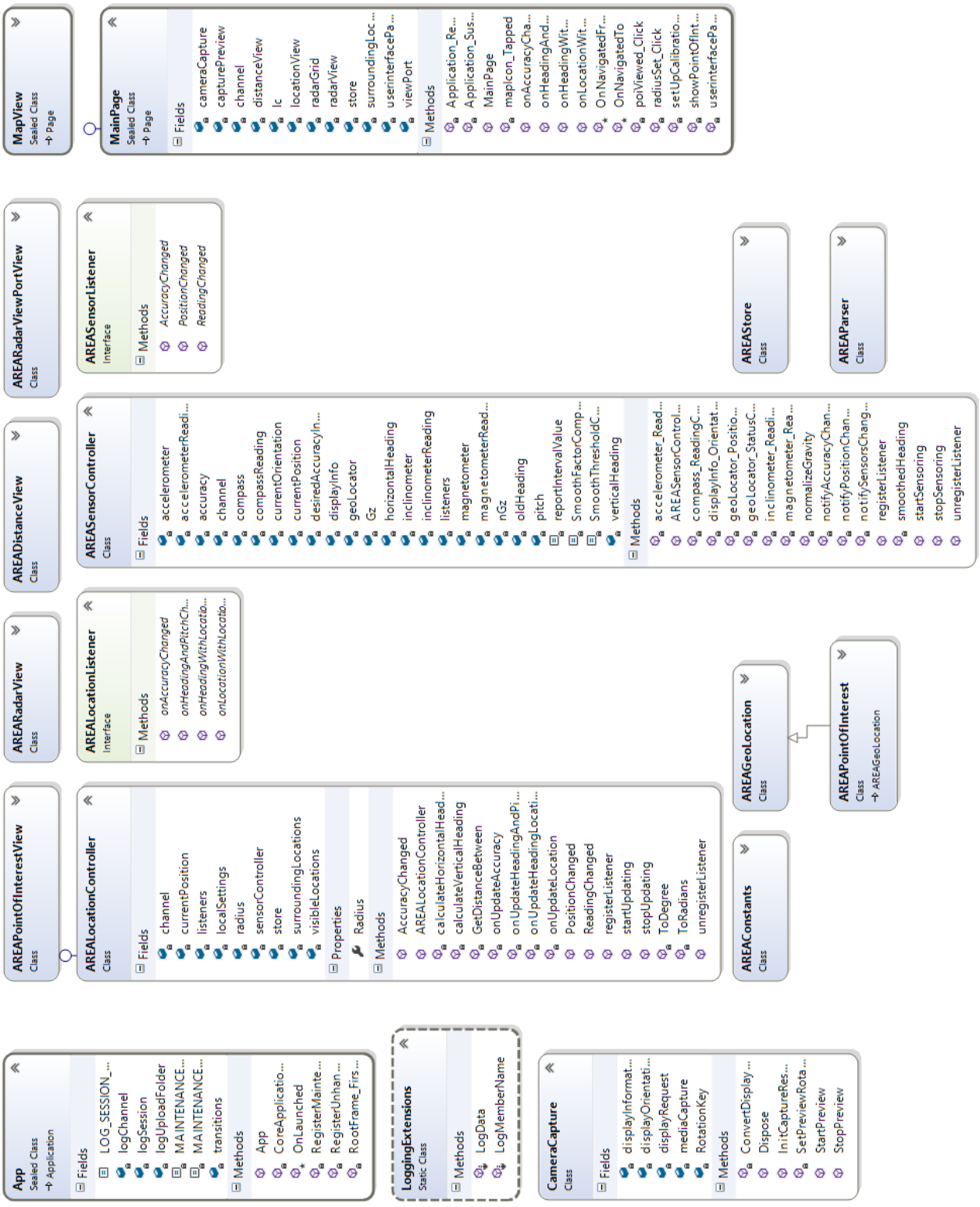
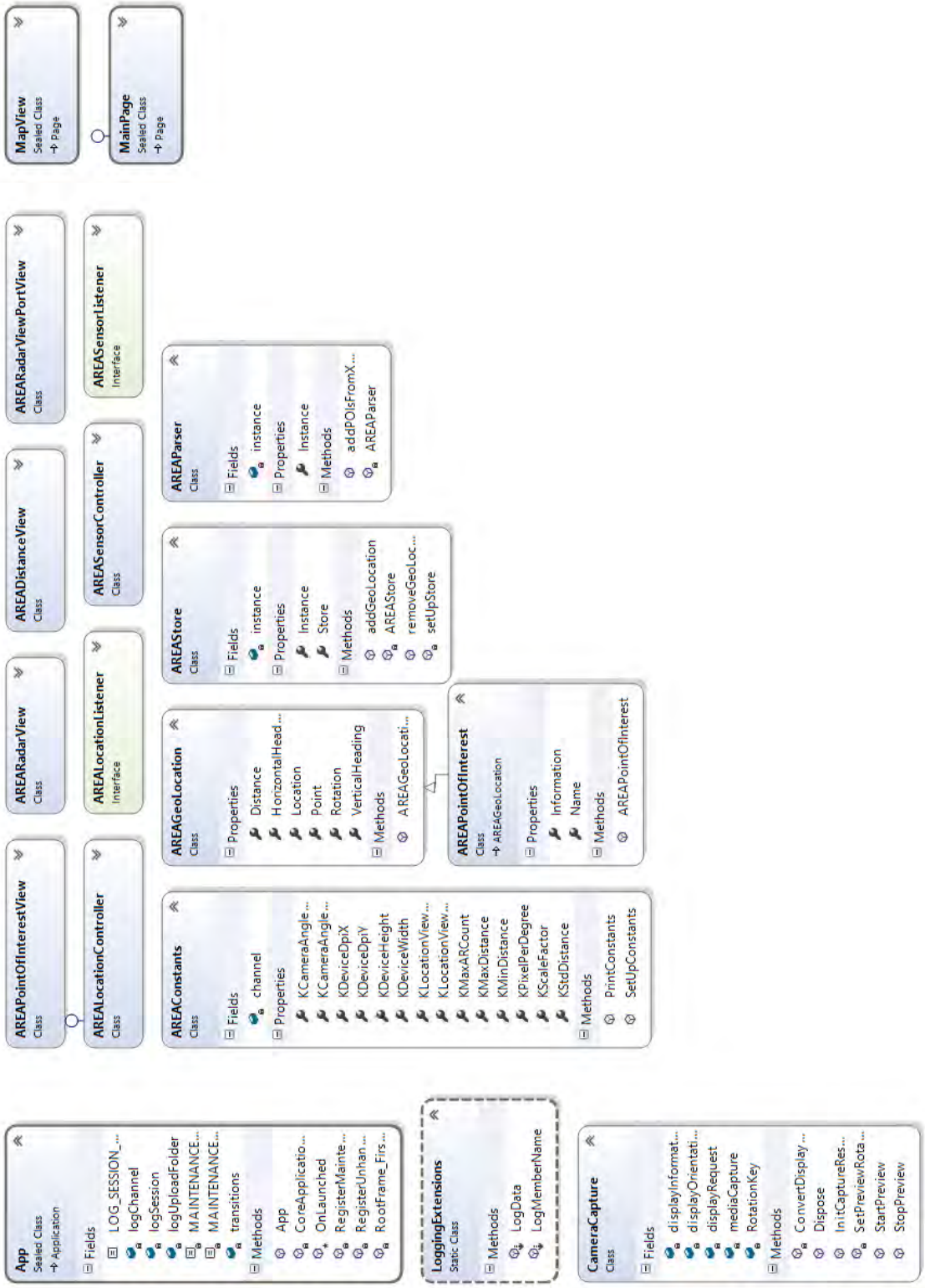


Figure B.4.: Class diagram, model expanded



Bibliography

[Bü15] Gemeinde Bühlerzell. Liveguide bühlerzell. http://www.buehlerzell.de/index.php?id=182, 2015. Accessed: 19.06.2015.

[Bie15] Jared Bienz. Gart. http://gart.codeplex.com/, 2015. Accessed: 12.06.2015.

[Blo13] The Official Microsoft Blog. Delta air lines soars with more than 19,000 new windows phone 8 and microsoft dynamics for retail devices. http://bit.ly/X2xd0t, 2013. URL shortened. Accessed: 18.06.2015.

[Bun13] Deutscher Bundestag. Bundestag-app für smartphones und tablets. https://www.bundestag.de/apps, 2013. Accessed: 18.06.2015.

[B.V15] HERE Europe B.V. Here city lens. Search in the Store on https://www.windowsphone.com/, 2015. Accessed: 12.06.2015.

[CFA+10] Julie Carmigniani, Borko Furht, Marco Anisetti, Paolo Ceravolo, Ernesto Damiani, and Misa Ivkovic. Augmented reality technologies, systems and applications. In *Multimedia Tools and Applications*, pages 341–377. Springer Science+Business Media LLC, December 2010.

[Cor15] Apache Cordova. Apache cordova. https://cordova.apache.org/, 2015. Accessed: 19.06.2015.

[Dit15] Stadt Ditzingen. App & liveguide. http://www.ditzingen.de/index.php?id=651, 2015. Accessed: 19.06.2015.

[FSBA06] Peter Fröhlich, Rainer Simon, Lynne Baillie, and Hermann Anegg. Comparing conceptual designs for mobile access to geo-spatial information. In *Proceedings of the 8th conference on Human-computer interaction with mobile devices and services*, pages 109–112, 2006.

[Gey13] Michael Geyer. Geometric analysis of an observer on a spherical earth and an aircraft or satellite. Technical report, U.S. Department of Transportation - Research and Innovative Technology Administration - John A. Volpe National Transportation Systems Center, 2013. DOT-VNTSC-FAA-13-08.

[GPSR13] Philip Geiger, Rüdiger Pryss, Marc Schickler, and Manfred Reichert. Engineering an advanced location-based augmented reality engine for smart mobile devices. Technical Report UIB-2013-09, University of Ulm, Ulm, October 2013.

[Gre15] GrepCode. android.location.location. http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/5.1.0_r1/android/location/Location.java#Location.computeDistanceAndBearing%28double%2Cdouble%2Cdouble%2Cdouble%2Cfloat[]%29, 2015. Accessed: 19.06.2015.

[GSP+14] Philip Geiger, Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert. Location-based mobile augmented reality applications: Challenges, examples, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 383–394,

April 2014.

[Inc15a] Apple Inc. Objective-c runtime programming guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtInteracting.html#//apple_ref/doc/uid/TP40008048-CH103-SW5, 2015. Accessed: 15.06.2015.

[Inc15b] Google Inc. Art and dalvik. https://source.android.com/devices/tech/dalvik/, 2015. Accessed: 15.06.2015.

[Inc15c] Google Inc. Glossary. https://developer.android.com/guide/appendix/glossary.html, 2015. Accessed: 15.06.2015.

[Inc15d] Google Inc. Tasks and back stack. http://developer.android.com/guide/components/tasks-and-back-stack.html, 2015. Accessed: 15.06.2015.

[Jun15] Junaio. Junaio. http://www.junaio.com/, 2015. Accessed: 12.06.2015.

[Lam15] Philip Lamb. Artoolkit. http://www.hitl.washington.edu/artoolkit/, 2015. Accessed: 19.06.2015.

[LKKS09] Ryong Lee, Daisuke Kitayama, Yong-Jin Kwon, and Kazutoshi Sumiya. Interoperable augmented web browsing for exploring virtual media in real space. In *Proceedings of the 2nd International Workshop on Location and the Web*. ACM New York, 2009.

[Mü14] Julia Müller. Konzeption und prototypische implementierung eines verfahrens zur poi clusterbehandlung innerhalb einer augmented reality anwendung. Bachelor thesis, Universität Ulm, 2014.

[MSD15a] MSDN. Common language runtime (clr). https://msdn.microsoft.com/de-de/library/8bs2ecf4%28v=vs.110%29.aspx, 2015. Accessed: 15.06.2015.

[MSD15b] MSDN. Diagnostics. https://msdn.microsoft.com/en-us/library/windows/desktop/ee663269%28v=vs.85%29.aspx, 2015. Accessed: 19.06.2015.

[MSD15c] MSDN. Move from android to winrt. https://msdn.microsoft.com/en-us/library/windows/apps/jj945421.aspx, 2015. Accessed: 15.06.2015.

[MSD15d] MSDN. .net for windows store apps overview. https://msdn.microsoft.com/library/windows/apps/xaml/br230302.aspx/, 2015. Accessed: 19.06.2015.

[MSD15e] MSDN. Richtlinien zum skalieren auf die pixeldichte. https://msdn.microsoft.com/de-de/library/windows/apps/hh465362.aspx, 2015. English version not available anymore. Accessed: 17.06.2015.

[MSD15f] MSDN. Uielement class. https://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.uielement.aspx, 2015. Accessed: 17.06.2015.

[MSD15g] MSDN. Windows.devices.sensors namespace. https://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.sensors.aspx, 2015. Accessed: 19.06.2015.

[PLRH12] Rüdiger Pryss, David Langer, Manfred Reichert, and Alena Hallerbach. Mobile task management for medical ward rounds - the medo approach. In *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, number 132 in LNBIP, pages 43–54. Springer, September 2012.

[PMLR15] Rüdiger Pryss, Nicolas Mundbrod, David Langer, and Manfred Reichert. Supporting medical ward rounds through mobile task and process management. *Information Systems and e-Business Management*, 13(1):107–146, February 2015.

[PRH+15] Rüdiger Pryss, Manfred Reichert, Jochen Herrmann, Berthold Langguth, and Winfried Schlee. Mobile crowd sensing in clinical and psychological trials ? a case study. In *28th IEEE Int'l Symposium on Computer-Based Medical Systems*. IEEE Computer Society Press, June 2015.

[PRLS15] Rüdiger Pryss, Manfred Reichert, Berthold Langguth, and Winfried Schlee. Mobile crowd sensing services for tinnitus assessment, therapy and research. In *IEEE 4th International Conference on Mobile Services (MS 2015)*. IEEE Computer Society Press, June 2015.

[PT10] Rémi Paucher and Matthew Turk. Location-based augmented reality on mobile phones. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 9–16. IEEE Computer Society Conference on, June 2010.

[RS03] Gerhard Reitmayr and Dieter Schmalstieg. Location based applications for mobile augmented reality. In *Proceedings of the Fourth Australasian user interface conference on User interfaces*, pages 65–73. Australian Computer Society, Inc., 2003.

[SPR15] Johannes Schobel, Rüdiger Pryss, and Manfred Reichert. Using smart mobile devices for collecting structured data in clinical trials: Results from a large-scale case study. In *28th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2015)*. IEEE Computer Society Press, June 2015.

[SPSR15] Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert. An engine enabling location-based mobile augmented reality applications. In *Web Information Systems and Technologies - 10th International Conference, WEBIST 2014, Barcelona, Spain, April 3-5, 2014, Revised Selected Papers*, LNBIP. Springer, 2015.

[SSP+13] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Hans Nienhaus, and Manfred Reichert. Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. In *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, pages 509–518, May 2013.

[SSP+14] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Fabian Maier, and Manfred Reichert. Towards process-driven mobile data collection applications: Requirements, challenges, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 371–382, April 2014.

[SSPR15] Johannes Schobel, Marc Schickler, Rüdiger Pryss, and Manfred Reichert. Process-driven data collection with smart mobile devices. In *Web Information Systems and Technologies - 10th International Conference, WEBIST 2014, Barcelona, Spain, Revised Selected Papers*, LNBIP. Springer, 2015.

[USA15] IDC Corporate USA. Smartphone os market share, q1 2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015. Accessed: 18.06.2015.

[Vin75] Thaddeus Vincenty. Survey review. Technical Report 176, Directorate of Overseas Surveys of the Ministry of Overseas Development, 1975. Vol. XXIII.

[Wik15] Wikitude. Wikitude. http://www.wikitude.com, 2015. Accessed: 12.06.2015.

[WS14] Andy Wigley and Matthias Shapiro. Building apps for windows phone 8.1 jump start. http://mslcc-admin.mscareerconference.com/Uploads/Downloadables/WP81JSDay1_6c299b3a951544a0a78dc081911f1a1d_99802e7554794e1c84ee52bb62ec0735.zip, 2014. Microsoft Virtual Academy.

[Xam15] Xamarin. Xamarin. http://xamarin.com/, 2015. Accessed: 19.06.2015.

[Yel15] Yelp. Yelp. http://www.yelp.com/, 2015. Accessed: 12.06.2015.