Mario Lins, BSc

Design and implementation of an enterprise solution for asset inspections in the electrical power industry

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology

Graz, December 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

An electricity provider has to ensure the proper functioning of all components in the electrical field. Therefore, it is important to continuously perform testing and maintenance activities for all these components. There already exist several software solutions in order to support the responsible testers on performing these activities. However, there are still special circumstances where it is not possible to use such software solutions. As a consequence, the testers have to use the traditional error-prone pen and paper approach to perform their activities.

This master thesis is done in cooperation with the company OMICRON electronics GmbH. OMICRON already offers a software solution to support the tester in the field under common circumstances. The focus of this thesis is to design and realize an enterprise solution that combines the already existing software infrastructure with a solution to replace the error-prone pen and paper approach. This enterprise solution is called ADMO Inspect. The first part of this thesis is to analyse the current application landscape. Based on this analysis it is possible to define the requirements and in addition to plan the architecture of ADMO Inspect. The final decision regarding the architecture is to introduce an additional service layer and to realize a mobile app to support the tester in the field. The next part of this thesis is to evaluate possible technologies that can be used to realize these components. After a detailed comparison of the evaluation results the decision was made to use Windows Communication Foundation to realize the service layer and Xamarin to implement the mobile app. At this point this thesis focuses on several implementation details. First, it shows how to create a screen design for the mobile app. Then, the most important implementation details about the service layer and the mobile app are presented. The mobile app part comprises topics like code modularization, ”Inversion of Control”, ”Model-View-ViewModel”, the user interface and some details about data consistency and synchronization. In addition, this thesis discusses a developing practice called ”Continuous Integration” and also some details about the testing strategy of ADMO Inspect. The last part of this thesis is about the empirical evaluation of the app. This evaluation is done by performing a usability test with representative users.

Kurzfassung

Ein Energieversorger muss sicherstellen, dass alle Komponenten im elektrischen Feld richtig funktionieren. Dazu ist die Durchführung von kontinuierlichen Test- und Wartungsarbeiten für alle Komponenten essentiell. Es existieren bereits einige Software-Lösungen um die dafür zuständigen Tester bei diesen Arbeiten zu unterstützen. Allerdings gibt es immer noch gewisse Umstände, bei denen so eine Software-Lösung nicht eingesetzt werden kann. Das führt dazu, dass die Tester eine fehleranfällige Papier und Bleistift Methode verwenden müssen.

Diese Masterarbeit ist in Kooperation mit der Firma OMICRON electronics GmbH erstellt worden. OMICRON stellt bereits eine Software-Lösung zur Unterstützung von Testern, die unter normalen Umständen im Feld arbeiten, zur Verfügung. Der Fokus dieser Arbeit liegt auf der Planung und der Realisierung einer Unternehmenslösung, welche die bestehende Software-Infrastruktur mit einer Lösung, welche die fehleranfällige Papier und Bleistift Methode ersetzt, kombiniert. Diese Unternehmenslösung hat den Namen "ADMO Inspect".

Der erste Teil dieser Arbeit beschäftigt sich mit der Analyse der bestehenden Infrastruktur. Basierend auf dieser Analyse ist es möglich die Anforderungen zu definieren und zusätzlich die Architektur von ADMO Inspect zu planen. Die finale Entscheidung bezüglich der Architektur war es, eine zusätzliche Service-Schicht einzuführen und eine mobile App zu realisieren, um den Tester im Feld zu unterstützen. Der nächste Schritt dieser Arbeit beschäftigt sich mit der Evaluierung von möglichen Technologien, die für die Umsetzung der einzelnen Komponenten verwendet werden können. Aufgrund eines detaillierten Vergleichs der Evaluierungsresultate, wurde entschieden „Windows Communication Foundation“ für die Service-Schicht und Xamarin für die mobile App zu verwenden. Die darauffolgenden Kapitel beschreiben einige Implementierungsdetails. Zuerst wird gezeigt wie das Design der Benutzeroberfläche der mobilen App erstellt wird. Anschließend werden die wichtigsten Implementierungsdetails betreffend der Service-Schicht und der mobilen App präsentiert. Bezüglich der mobilen App werden Themen wie Code-Modularisierung, „Inversion of Control", „Model-View-ViewModel", die Benutzeroberfläche und Details bezüglich Datenkonsistenz und Synchronisation behandelt. Zusätzlich diskutiert diese Masterarbeit den Prozess „Continuous Integration" und die verwendete Teststrategie von ADMO Inspect. Der letzte Abschnitt dieser Arbeit beschreibt die empirische Evaluierung der App. Diese Evaluierung ist anhand eines Usability Tests mit repräsentativen Benutzern durchgeführt worden.

Contents

List of Figures

1 Introduction

On the 9th of November 1965, 5:16 p.m. in New York state and it's neighborhood. 800.000 people were trapped in subways and thousands could not leave office buildings, elevators or trains. 10.000 National Guardsmen and 5000 additional off-duty policemen were on alert to prevent looting.
All of this happened due to a great blackout [1]. A reason for this event was a human failure but also insufficient monitoring of the power supply system. This shows how crucial the infrastructure of an electricity provider is. That is why, nowadays an electricity provider has to continuously perform testing and maintenance activities for all the electrical components. Those testing and maintenance activities are very important to ensure that all the electrical components are working properly and therefore to prevent such problems like the blackout of 1965.

Meanwhile, there exist several software solutions to support performing and reporting maintenance and testing activities on electrical components. The main purpose of these solutions is to offer the tester a well defined, automatic and reliable process in order to guarantee the proper functioning of the assets. Although, these software solutions are continuously enhanced and improved, there are still circumstances where it is not possible to use such reliable processes in a satisfying way.

One example of such a special circumstance would be a maintenance task for an electrical tower. Most of the electrical towers are located in areas where it is not possible to get any kind of mobile connection. In addition, it is hardly possible to carry a laptop or similar up an electrical tower. Therefore, the testers have to use the traditional error-prone pen and paper approach to perform their activities. This approach leads to already stated problems,

[1] cf: history.com, 2015, rense.com, 2015.

where the reliability of the power supply system can be affected. Unfortunately, even nowadays there are some maintenance and testing activities, which face the same problem as the one that caused the blackout of 1965 - a high probability of human failure.

As a consequence, reporting of maintenance and testing results of those electrical components is often done via pen and paper. There are multiple error-prone steps by using this pen and paper method for reporting such important results. For example, hand writing cannot always be clearly deciphered at a later point of time. Another problem is the additional work, because when the tester is back in the office, the results have to be entered into a software system. Since the tester has to write down all the results twice, it is more probable to produce a mistake. In summary, it is not always possible for testers working under these special circumstances to use a reliable automated process, which reduces the probability of human failure, in order to perform their maintenance and testing activities.

This master thesis is done in cooperation with the company OMICRON electronics GmbH. One main business segment of OMICRON is to support electricity providers in performing those important testing and maintenance activities for electrical components. OMICRON already offers a software solution, which enables the tester to plan, manage and report those activities. In case of the mentioned special circumstances, the already existing solution, which is designed for office use, does not fully satisfy the requirements needed to replace the current error-prone pen and paper approach.

Therefore, this master thesis focuses on the design and implementation of an enterprise solution, that combines the already existing infrastructure with a solution to replace the pen and paper approach. It should minimize the probability of human failure and offer a reliable automatic process for reporting important test and maintenance results.

2 Application Landscape

OMICRON already offers a product to support electricity providers in managing their maintenance and testing activities. This master thesis describes the combination of this existing product with new components to form a solution that replaces the pen and paper approach. Therefore, the first step of this master thesis and thus also this chapter is to analyse this already offered product and its infrastructure.

It is very important for an electricity provider to continuously perform maintenance and testing activities on all kinds of electrical components to ensure their proper functioning. Planning, managing and reporting these activities is often difficult in practice. In order to support the customer in these tasks, OMICRON developed an innovative solution called ADMO. The following workflow illustrates the basic functionalities of ADMO and shows how the software can be used to support an electricity provider.

In most cases, an employee of the electricity provider, acts as the maintenance manager. When starting ADMO, the first task of the maintenance manager is to enter several kinds of information about the components of an electrical power system into the software. Before any of these electrical components, which are often called assets, can be created in ADMO, the maintenance manager needs to add the corresponding substations. Optionally it is possible to add voltage levels and feeders to a substation.

The following figure illustrates the location tree which can be used to add the location information and afterwards to find the containing assets.

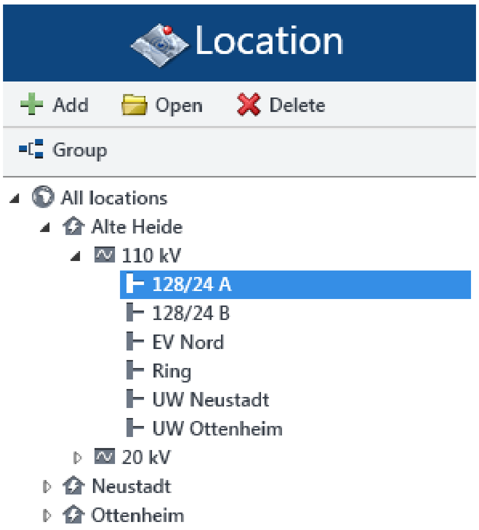


Figure 2.1: ADMO Location Tree. Source: Screenshot ADMO

After creating all necessary substations and optionally their voltage levels and feeders, the maintenance manager is able to create an asset in ADMO. There exist several asset kinds, which can be managed in ADMO - for instance: a circuit breaker, a protective relay or a current transformer.

The following illustration shows a section of the new asset dialog.

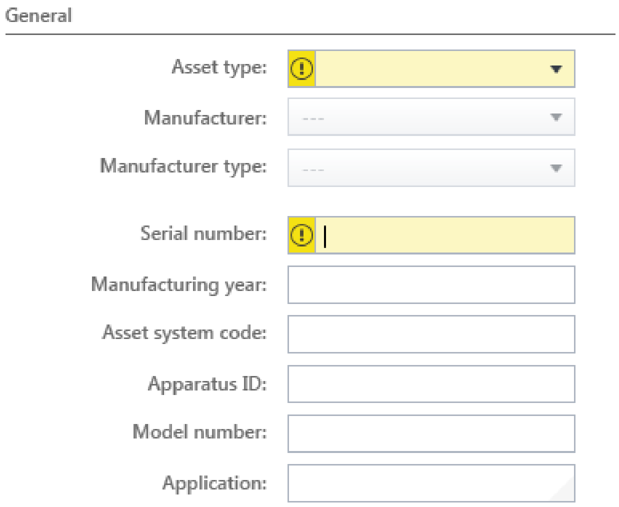
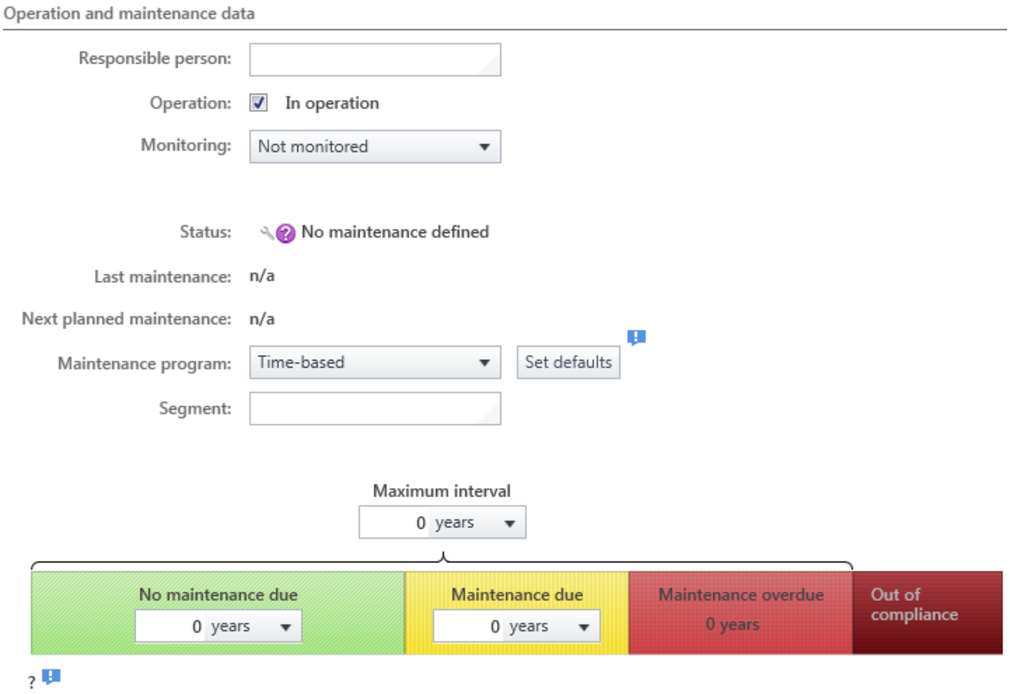


Figure 2.2: ADMO New Asset Dialog. Source: Screenshot ADMO

In this dialog, it is also possible to define a maintenance plan and to attach test templates. The maintenance plan is an important means of planning maintenance activities for an asset. In order to define the maintenance cycle of an asset, the maintenance manager is able to set different maintenance intervals. Another section of the new asset dialog allows the maintenance manager to attach test templates. These files are often necessary to perform specific maintenance activities.

The figure below shows the section of the dialog where it is possible to define the maintenance intervals of an asset:



Operation and maintenance data

Responsible person:

Operation: ☑ In operation

Monitoring: Not monitored

Status: No maintenance defined

Last maintenance: n/a

Next planned maintenance: n/a

Maintenance program: Time-based

Segment:

Maximum interval: 0 years

No maintenance due: 0 years

Maintenance due: 0 years

Maintenance overdue: 0 years

Out of compliance

Figure 2.3: Define maintenance intervals. Source: Screenshot ADMO

The planning of such activities needs to be done quite a long time before the activity itself is performed. The reason for this is, that often the whole feeder has to be disconnected from the network when performing such an activity. Thus, the maintenance manager has to precisely plan the next maintenance activity of an asset. In terms of ADMO, such a maintenance activity is called `Event`.
When creating an event related to an asset, the maintenance manager needs to have an overview of previous and already planned events, including their results. This kind of overview, which is realized by a structured timeline is also a part of ADMO. Thus, the maintenance manager is able to create a new event in consideration of the already completed ones. The timeline itself is structured by colors corresponding to the defined intervals, which can be set in the new asset dialog.

The following figure shows a maintenance timeline, that contains two events. One event for which maintenance was already performed, and assessed as ”passed” and a second event that is planned in the future.

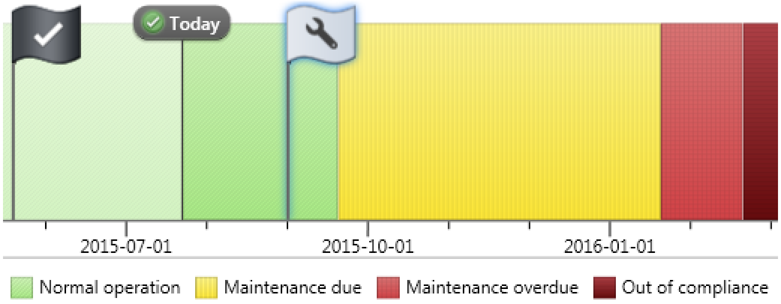


Figure 2.4: ADMO Time line. Source: Screenshot ADMO

At this point, the maintenance manager has completed the planning task for an asset and another person becomes involved - the tester. Testers are employees, who are performing such planned maintenance activities at assets. Once they are in the particular substation and start their maintenance activity, they make use of the asset specific test templates, which were created by the maintenance manager. When the test is completed, they have to enter the test results into ADMO. Afterwards, the maintenance manager can review the results and start planning further events based on these results.

That was one example for a workflow when using ADMO and gave a brief overview of some features. The next paragraph provides a more technical insight into the software.

ADMO uses a database to store information about assets. There are two different versions of ADMO. A standalone version and a client-server version. The standalone version stores all the information in a local database, while the client-server version uses a database server, which is typically located directly at the customer. Due to the problem of missing internet connectivity at some substations, it is not always possible to get a connection to the database server when using the client-server version. In that case ADMO offers a solution where it is possible to switch to a special offline mode. This means all the information, which is stored on the database server is replicated to a local database. In case of any offline changes of the data, it is of course also possible to synchronize those changes back to the database server, as soon as the connection to the server can be established again.

The following illustration shows an overview of the current client-server infrastructure:

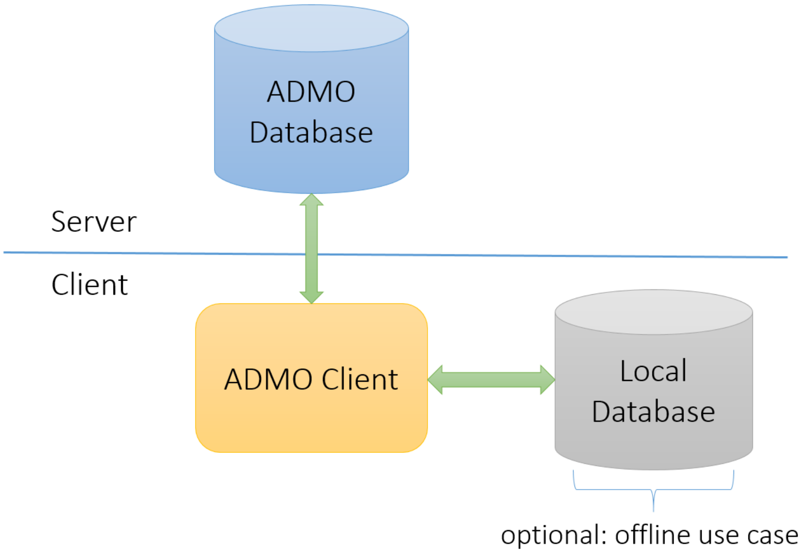


Figure 2.5: Client-server infrastructure. Source: Own representation

3 Requirements

This chapter is about the functional and non-functional requirements for the enterprise solution, which is called ADMO Inspect. Those requirements were defined by OMICRON.

3.1 Functional Requirements

As described in the introduction chapter, a tester has to face some problems when performing test activities at certain assets in the field. An example for an asset, where the tester has to deal with several problems, is an electrical tower. For instance, electrical towers are often located outside the range of any mobile connectivity. Another problem is that the testers cannot carry a laptop up the tower. Hence, the current testing process, which is mainly about dealing with checklists, relies on an error-prone pen and paper approach.

ADMO Inspect offers a solution to replace the current error-prone testing process. In order to deal with the mentioned problems the solution has to meet some requirements. One requirement of ADMO Inspect is to work without any mobile connectivity. Another requirement is to satisfy the mobile use case, especially in cases like climbing up a tower. Additionally, ADMO Inspect needs to offer a reliable automatic process to deal with checklists in order to replace the current pen and paper solution.

The next paragraph describes the functional requirements in the form of user stories. A user story is part of an agile software development process like SCRUM. It consists of a formal description of the requirement and the definition of acceptance criteria. The formal definition follows a pattern in the form of `"As a <user> I want to <requirement>"`. The acceptance criteria are used to define the results of a user story and are listed beneath each formal description [1][2].

As a tester I want to have access to important information about an asset.

- *Tester has access to information about an asset*
  - *General location information (i.e. address, postal code,...)*
  - *General asset data (i.e. serial number,...)*
  - *General maintenance information (i.e. maintenance intervals)*
- *Tester does not have access to all data available in ADMO (i.e. performed maintenance activities)*

As a tester I want to work with the same data as in ADMO.

- *The available data in ADMO Inspect is equivalent to the data in ADMO.*

As a tester I want to find an asset by selection of its location like substation, voltage level and feeder.

- *Selecting a substation shows all assets of this substation, its voltage levels and their feeders.*
- *Selecting a voltage level shows all assets of this voltage level and also its feeders.*
- *Selecting a feeder shows all assets of this feeder.*

As a tester I want to find an asset by scanning its QR-Code.

As a tester I want to use my checklists.

- *ADMO Inspect contains all checklist templates to create new checklists.*

As a tester I want to perform my test activity even when I am offline.

[1] cf: Henning Wolf, 2012, p. 128,
[2] cf: Roman Pichler, 2011, p. 40

- *Data about locations, assets and maintenance activities is available in ADMO Inspect even when being offline.*

As a tester I want to work with my test results in ADMO.

- *Completed checklists in ADMO Inspect are also available in ADMO.*

The vision of ADMO Inspect contains the following optional user stories:

As a tester I want to select the checklists required for test activities.

- *Beside general information (i.e. location information,...) it is also possible to select checklists, which are available in ADMO Inspect.*

As a tester I want to tag an asset with a QR-Code.

As a tester I want to tag an asset with GEO coordinates.

As a tester I want to use GEO coordinates to find an asset.

As a tester I want to be able to use fully modeled checklists.

As an electricity provider I want to define which users have access to which information.

- *A user, who is not authorized to access restricted information, cannot get access to those information.*
- *A user, who is authorized to access restricted information, is able to get access to this information.*

3.2 Non-functional Requirements

As described in the previous chapter ADMO directly communicates with the database. The database schema is defined by ADMO. Consequently, if ADMO changes this schema, all clients that are currently using the corresponding database, have to adopt these changes.
OMICRON wants the ADMO Inspect solution to be decoupled from ADMO release cycles. This means whenever a new version of ADMO is released, it is not necessary to update ADMO Inspect too. Thus, to achieve such independence, it is very important to detach the solution from the database.

OMICRON also wants this database abstraction to be available to new products in the future, independent of their platform. Additionally, this abstraction should ensure data consistency.

Another requirement for ADMO Inspect is to be compatible with the currently used database management system, which is Microsoft SQL-Server. Consequently, ADMO Inspect needs to be compatible with the existing Microsoft environment.

OMICRON also mentioned code reusability as an optional requirement. One reason for this is that ADMO already contains some code modules, that could be reused in ADMO Inspect.

4 Architecture

The previous chapter introduced the requirements for ADMO Inspect. Based on the definition of those requirements, this chapter defines the technical components of ADMO Inspect and gives an overview of the architecture.

Beginning with the already existing infrastructure, OMICRON wants to achieve independence between ADMO Inspect and ADMO like defined in the Requirements chapter. Independence can be gained by using an additional layer that is responsible for abstracting the database. The main advantage of using an additional layer is that when the database scheme gets changed, only this layer has to be adopted instead of all clients which are using the particular database. An alternative to an additional layer would be direct database access, which would obviously not comply with the requirements.

To make a decision about the concrete technology to realize this additional layer, it is necessary to have a look at further requirements. The main purpose of the additional layer is to offer an interface for accessing the database. This interface should be available independently of the customer's platform since future products may also use it. That means, the communication with this interface has to support multiple common communication technologies. Another requirement is to offer the electricity provider a way to restrict the access to the database.
In summary, this additional layer should meet the following requirements:

1. Encapsulate the database
2. Available to multiple products
3. Independent of the customer's platform
4. Support multiple communication technologies
5. Restriction possibilities for accessing the database

A common solution to achieve these requirements is the usage of a service. The book ”Programming WCF Services”[1] describes a service in the following way:

> ”A service is a unit of functionality exposed to the world.”

That means, a service can offer its users a set of functionality through a clearly defined interface. In the case of this thesis the service functionalities can be used to abstract the communication with the database. Additionally, the book ”Agile Database Techniques” lists some other relevant advantages of using a service [2]:

1. Service access can be platform independent.
2. Services can be reused.

In summary, these facts and abilities when using a service leads to the decision to create an additional service layer.

Nevertheless, ADMO Inspect also needs another component in order to communicate with the service layer and to interact with the tester. One requirement of ADMO Inspect regarding the tester, is that the tester should be able to work offline when no internet connection is available. Further requirements are to offer an intuitive way to work with checklists and to support the mobile use case. The solution, which is able to achieve those requirements, is to develop a mobile app running on a tablet. By using a tablet, the tester is able to work offline, since the data can be stored directly on the device. The app also offers an intuitive way to work with checklists. The mobile use case is fulfilled too, because it is much easier to carry a tablet around instead of a laptop. An alternative solution to the mobile app would be a website, with which it is also possible to achieve the given requirements. Nonetheless, based on the realization of the offline use case, it was decided to use the more established solution that is using an SQL-based database system like SQLite.

To sum up, ADMO Inspect consists of multiple components, in example the already existing database, the new service layer and an app in order to

[1] Löwy, 2010.

[2] cf: Scott, 2012, p. 341

support the tester during field testing. Hence, ADMO Inspect is based on a three-tier architecture, which is illustrated in the following picture:

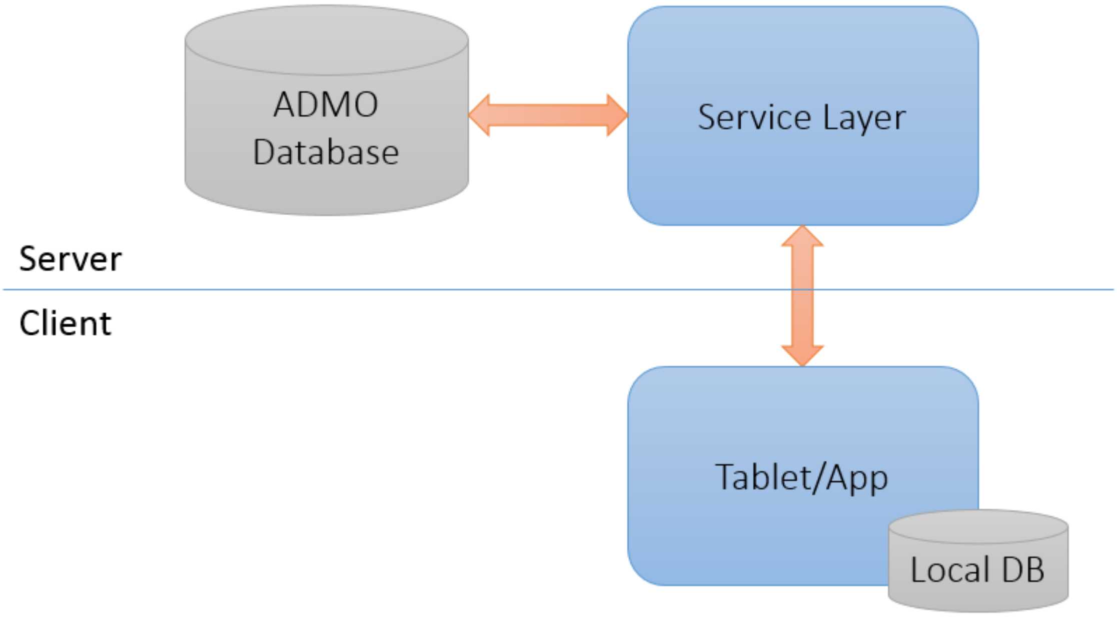


Figure 4.1: ADMO Inspect - Architecture. Source: Own representation

5 Technical Evaluation

The previous chapter defined the components of ADMO Inspect - the service layer and the mobile app. This chapter is about the evaluation of possible technologies for realizing these components.

5.1 Service Layer

This section describes the service layer which encapsulates the database access. In order to choose a proper technology for realizing the service layer, it is necessary to define its technical requirements. The following section gives an overview of these requirements, which were defined by OMICRON.

One important aspect is the compatibility with the Microsoft environment and also the currently used database management system, which is Microsoft SQL-Server. Since the database management system is running in a Microsoft environment, it is likely that the customer also wants the database access layer running in the same environment or even on the same machine. Therefore, the service layer needs to be compatible with Microsoft's operating system. As an optional requirement, OMICRON stated the code reusability. For instance, some code modules for accessing the database can be reused. Therefore, the database access layer needs to support the same programming language as ADMO, which is C# .

There is one popular solution developed by Microsoft, which fits all those requirements. The framework is called Windows Communication Foundation (WCF). Hence, the next section gives an overview on WCF and also a detailed evaluation report on this framework.

WCF Evaluation

WCF is a framework for developing services running in a Windows environment. It is part of the .NET framework and therefore, it can only be hosted on a Windows platform. The general purpose of a WCF service is to expose messages between a client and a service by using common transport protocols like HTTP, TCP ,... [1]

Architecture This first paragraph describes the architecture of WCF, which consists of several layers. The following figure gives an overview of these layers. Afterwards, a brief description of each layer is given. [2]

[1] cf: Löwy, 2010, pp. 1,3

[2] cf: Microsoft, 2015j

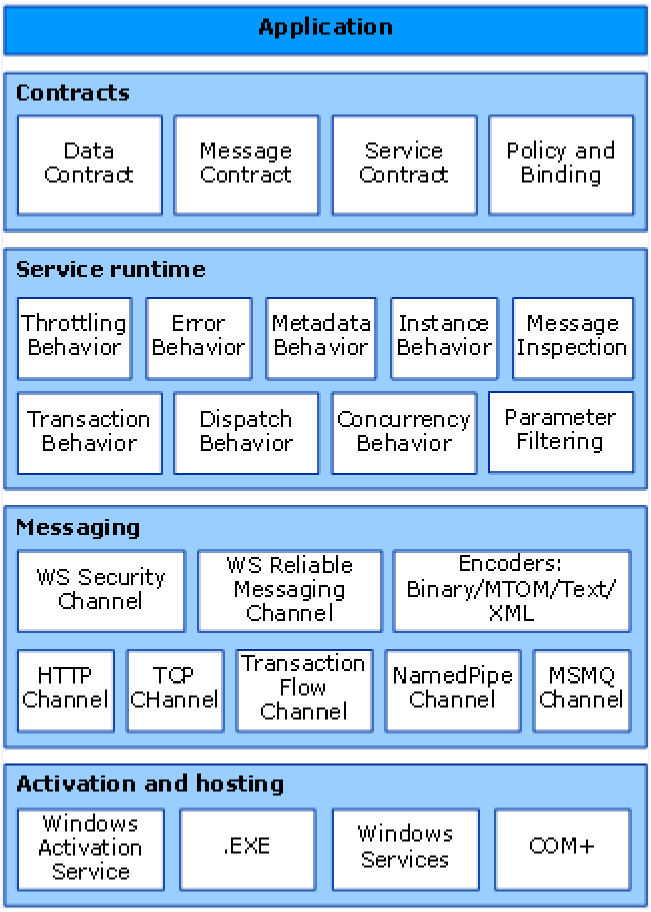


Figure 5.1: WCF Architecture. Source: Microsoft, 2015j

Contracts layer - In order to consume a service the client and the service need to define what type of data they are using. Therefore, WCF uses contracts like the data contract or the message contract. The difference between these contracts is that a data contract only defines the content of the message, while the message contract can define the whole message format. The following examples show a data contract and a message contract. The data contract defines the `Substation` class which can be transferred between the service and the client. The message contract defines the `SubstationRequestMessage` which also contains the header field `UserName`.

```
//This annotation creates a data contract
[DataContract]
public class Substation
{
  [DataMember]
  public string Id { get; set; }

  [DataMember]
  public string Name { get; set; }
}

//This annotation creates a message contract for
//requesting a substation
[MessageContract]
public class SubstationRequestMessage
{
  [MessageHeader()]
  public string UserName;

  [MessageBodyMember()]
  public string Id;
}
```

Another important contract is the service contract. The contract is defined via an interface class and lists all methods that can be called by a client. An example for such a service contract is shown below:

```
[ServiceContract]
public interface IService
{
  [OperationContract]
  IList<Substation> GetAllSubstations();
}
```

Service runtime This layer contains several modules regarding the behaviour of the service at runtime. One important module is the error behaviour, which describes how the propagation of an error to the client is handled. Restricted propagation is necessary, because exposing too many error details to the client, could enable an attacker to use that data to compromise the system.

Messaging The messaging channel contains several different channels, which are responsible for handling messages. There are two different types of channels - the protocol channel and the transport channel.
A protocol channel can be a `Web Service Security` (WS-Security) protocol[3], for instance. This protocol can be used, for example, for signing messages to guarantee the authenticity of the sender.
A transport channel can be HTTP, for instance, and it defines the way messages are delivered. Consequently, this channel is responsible for reading and writing messages.

Activation and hosting A service needs to be executed. This can be done either by implementing a self-hosted service or by using a hosted service. The different hosting scenarios are described in more detail in chapter ”Implementation - Service Layer”.

[3] Microsoft, 2015h.

Security Security is an important topic when developing WCF services. When talking about security in terms of a WCF service, there are several aspects like authentication, authorization and transfer security that should be considered. This chapter covers these three security aspects by giving a short description of each one of them and providing a brief evaluation of implementing these aspects when using WCF. The information in this paragraph and a more detailed description of the specific topics can be found on the official Microsoft website[4] and in the book ”Programming WCF Services” [5].

Authentication Authentication describes the process when the client or the service confirm their identity. When developing a service, a possible requirement is to restrict access to a specific function. Therefore, the client has to confirm his identity to execute this function. On the other hand, when the client transfers sensitive data to the service, it is important that the service confirms its identity to the client. A service authenticates itself by using a certificate. WCF offers several authentication possibilities regarding client authentication:

- No authentication - There is no authentication of clients.
- Windows authentication - The client uses his windows credentials for authentication
- User name and password - The client provides a user name and corresponding password
- Certificates - As an alternative to the user name and password approach, the client can also transmit a certificate
- Issue token - Service and client use a secure token service. The service offers the client a token, which can be verified by the service.
- NTLM - This approach uses a challenge response protocol (only available with HTTP)

Authorization Talking about authorization means defining of what a user is allowed to do. Therefore, the client has to be authenticated first. WCF

[4] Microsoft, 2015d.

[5] Löwy, 2010, p. 525

supports the following approaches regarding authorization:

- Role-based authorization - A common method where the user's role is used for granting access to a function. A role can be based on Windows groups, custom roles or ASP.NET roles.
- Identity-based authorization - The user credentials are used for authorization. A typical use case for this method is to issue token authentication.
- Resource-based authorization - A WCF service is a resource, which is secured by ACL (Windows Access Control Lists)

Transfer security The last security aspect is transfer security[6], which describes the process to secure messages between the service and the client. If messages are not secured when being transferred from the client to the service, the whole process of authentication and authorization can be compromised. There are some security related targets that needs to be achieved in regard to transfer security:

- Integrity - The messages should not be modified during transport.
- Confidentiality - The message can only be read by approved parties.
- Authentication - Only the trustworthy service is able to read the client's messages

The following approaches are offered by WCF to achieve these security aspects:

- None - The transfer of the messages is being not secured.
- Transport security - Transport security relies on a low level of transporting messages. The messages including the user authentication information is encrypted with common transport protocols like Transport Layer Security or Secure Socket Layer. These protocols provides integrity, confidentiality and authentication since the whole communication is encrypted. The following figure illustrates the transport security mode:

[6] Microsoft, 2015b.

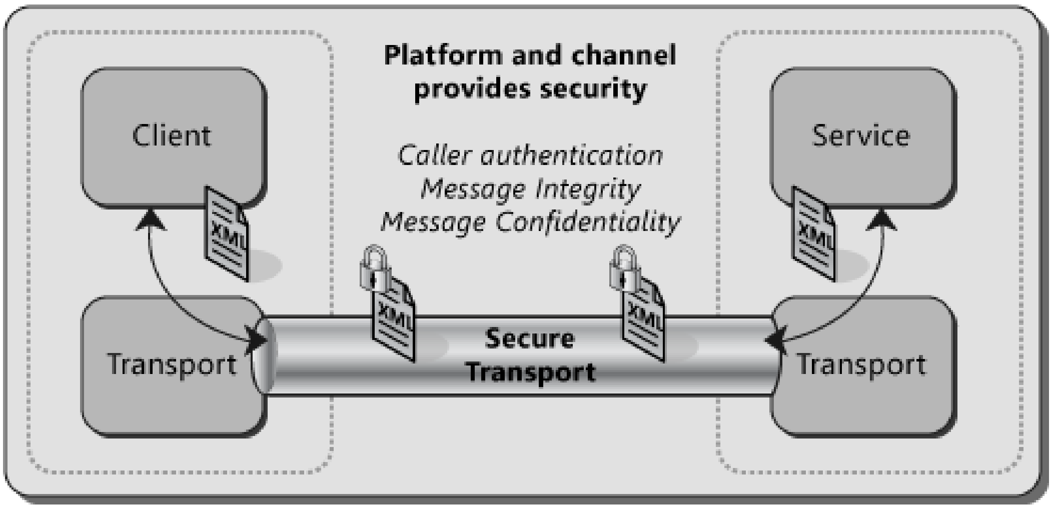


Figure 5.2: Transport Security. Source: Microsoft, 2015b

- Message security - Message security encrypts the messages, which also includes the user credentials, by using an approach defined by the WS-Security standard. The messages get encrypted at the client and decrypted at the service. Therefore, the transfer security does not rely on the transport layer. Since the messages get encrypted, this approach achieves the security requirements for the same reason as the transport security mode. The following figure illustrates message security:

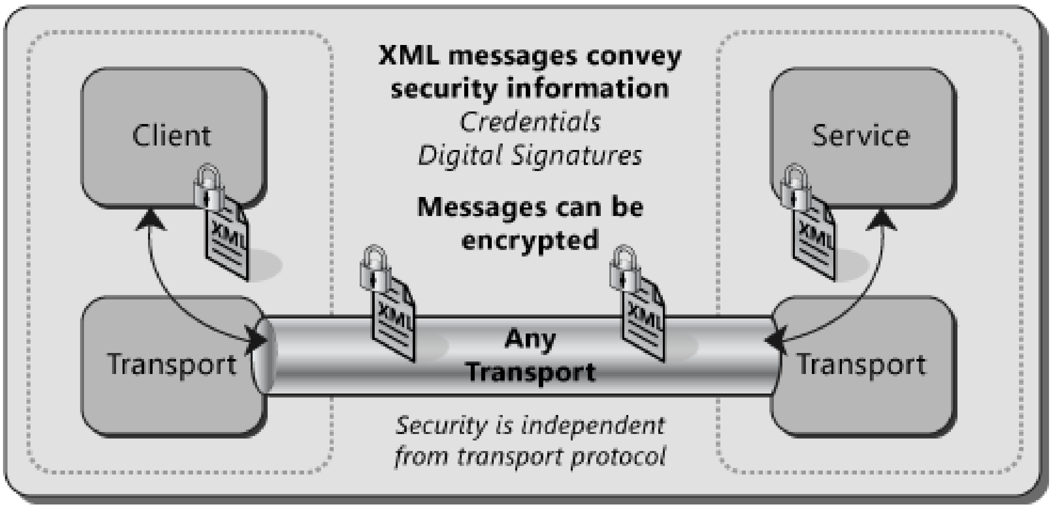

XML messages convey security information
Credentials
Digital Signatures
Messages can be encrypted
Client
Service
Transport
Any Transport
Transport
XML
XML
XML
XML
Security is independent from transport protocol

Figure 5.3: Message Security. Source: Microsoft, 2015b

- Both - Uses message and transport security. May be an overkill for some applications.
- Mixed - Uses transport security to achieve integrity, confidentiality and authentication regarding the messages and message security to secure the user's credentials.

Hosting scenarios A WCF service needs to be hosted within a Microsoft Windows environment in order to be usable. There are multiple possibilities to host such a service. The following list presents several ways of hosting WCF services [7]:

- Internet Information Service (IIS)
- Self-Hosting
- Windows Activation Service (WAS)
- Windows Server AppFabric

[7] cf: Löwy, 2010, p. 11

Internet Information Service This paragraph gives an overview on hosting a WCF service using IIS[8]. For the following explanation it is presumed that an IIS server is installed and that an IIS website is already set up for the service. An IIS hosted WCF service can be created via a template project offered by Visual Studio. When creating a project with this template, there are two important files regarding the hosting process. The first one is a `.svc` file, which contains the basic information about the `code behind file` and the service class. The following line of code shows an example of a `.svc` file, where the programming language, the debugging state and the service class are defined:

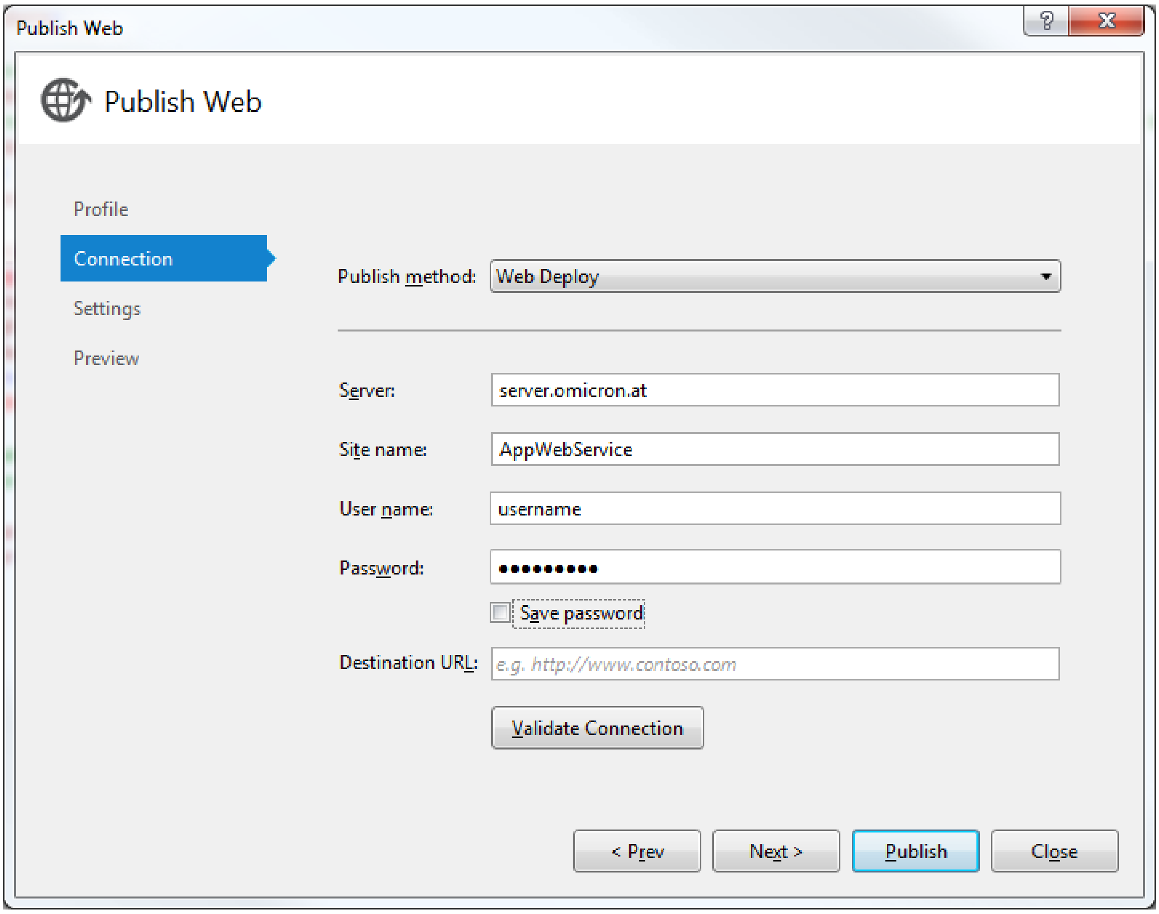
```
<%@ ServiceHost Language="C#" Debug="true"
 Service="Omicron.Service.AppWebService"%>
```

The second important file is a configuration file called `Web.config`. This file can be used to define bindings of a service, for example `HTTP` or `HTTPS`, or to define the version of the .NET framework. The following code extract shows an example where the program state is set to debug mode and the .NET framework version is defined:

```
<system.web>
    <compilation debug="true" targetFramework="4.5.1" />
</system.web>
```

After the configuration of these two files and the implementation of the service functionality is completed, the service is ready for IIS deployment. In order to do that, Visual Studio offers an easy-to-use deployment wizard, which allows the developer to define the deployment target. The following figure shows an example of the deployment wizard that deploys the service to `http://server.omicron.at/AppWebService.svc`:

[8] Microsoft, 2015a.



Publish method: Web Deploy

Server: server.omicron.at

Site name: AppWebService

User name: username

Password: ●●●●●●●●●

□ Save password

Destination URL: e.g. http://www.contoso.com

Figure 5.4: Publish WCF service. Source: Screenshot Microsoft Visual Studio 2013

Windows Process Activation Service (WAS) Another hosting approach for WCF services is using Windows Process Activation Service[9]. Hosting a service with IIS allows communication through HTTP or HTTPS, only. However, WAS also allows non-HTTP transport protocols like TCP. The main hosting configurations like the `.svc` file and the `Web.config` files are also used when hosting a service by WAS. The following example [10] shows how to implement a Windows service that hosts the service. The installation process of a Windows service is not described in the example but can be found on the official Microsoft website[11].

```
class Program : ServiceBase
{
    static void Main(string[] args)
    {
        ServiceBase.Run(new Program());
    }
    protected override void OnStart(string[] args)
    {
        ServiceHost serviceHost =
            new ServiceHost(typeof(HelloWorld));
        serviceHost.AddServiceEndpoint(typeof(IHelloWorld),
            new WSHttpBinding(), "");
        serviceHost.Description.Behaviors.Add(
            new ServiceMetadataBehavior {
              HttpGetEnabled = true });
        serviceHost.Open();
    }
}
```

Windows Server AppFabric Windows Server AppFabric is an extension for improving WAS, because WAS is not optimized for hosting WCF services like mentioned in the book ”Programming WCF Services” [12]. The main purpose of this extension is to offer more configuration possibilities regarding

[9] Microsoft, 2015e.

[10] cf: Microsoft, 2015c

[11] Microsoft, 2015c.

[12] Löwy, 2010, p. 20

the hosting process. Windows Server AppFabric also offers features like health monitoring and diagnostic functionality. A more detailed description of Windows Server AppFabric and of other hosting scenarios can also be found in the book ”Programming WCF Services”[13].

Self-Hosting One of the main differences when using a self-hosted service instead of an IIS or WAS hosted one, is that the developer has to create an instance of the service in the software code. Both the IIS and WAS approaches create this instance on their own, whenever a client makes a request. [14]

Using a self-hosted service means that the developer has to create the hosting process first. Such a hosting process can be implemented by developing a console application. The following code extract provides a short example of using a console application to host a WCF service. A more detailed example with explanation can be found on the Microsoft website[15].

```
private static readonly Uri ServiceAddress =
    new Uri("http://localhost:1234/HelloWorldService");

    static void Main(string[] args)
    {
        try
        {
            ServiceHost serviceHost =
                new ServiceHost(typeof(HelloWorldService),
                    ServiceAddress);

                        // Add endpoints to the service
            serviceHost.AddServiceEndpoint(
                typeof (IHelloWorldService),
                new WSHttpBinding(), "");

            // Add Metadata behaviour
            serviceHost.Description.Behaviors.Add(
```

[13] Löwy, 2010.
[14] cf: Microsoft, 2015f
[15] Microsoft, 2015f.

```
                new ServiceMetadataBehavior {
                    HttpGetEnabled = true });
            serviceHost.Open();
        }
        catch (Exception)
        {
            Console.WriteLine(
                "Error when starting service..");
        }
    }
```

5.2 Mobile App

This section provides an evaluation of several development technologies for developing the mobile app, which replaces the pen and paper approach. As part of this thesis, three popular app development technologies are evaluated and discussed in detail. Those technologies are native development, hybrid development and Xamarin. The evaluation comprises a brief overview of each technology and an investigation report about the feasibility of the evaluation criteria below. The main target operating system, which was defined by OMICRON, is Android.

5.2.1 Evaluation Criteria

The evaluation of the three app types is based on several criteria, which are necessary to achieve the requirements. One very important requirement is the possibility to work offline, since many substations do not have internet connectivity. Therefore, one evaluation criterion is to use the device's internal database. Since the app needs to communicate with the service layer, an additional criterion is the use of a REST service. In order to find an asset, the tester has the possibility to scan its QR-Code. Hence, the app needs to have access to the internal camera of the device.
An optional requirement defined by OMICRON is that the app can be used on different platforms. Thus, another evaluation criterion is cross-platform capability.

The list below gives an overview of the evaluation criteria:

1. Access device's internal database (SQLite)
2. Access camera
3. Using a REST service
4. Cross-platform capability (optional)

5.2.2 Native Development

Android is based on the Linux kernel and is developed by Google. It is an operating system primarily used for mobile devices like tablets and smartphones. Apps for the Android operating system are developed in the Java programming language. Nowadays, Android is the most popular operating system on mobile platforms [16]. Its market share is 81.1% [17].

Architecture

The following figure illustrates the Android software stack with its five layers:

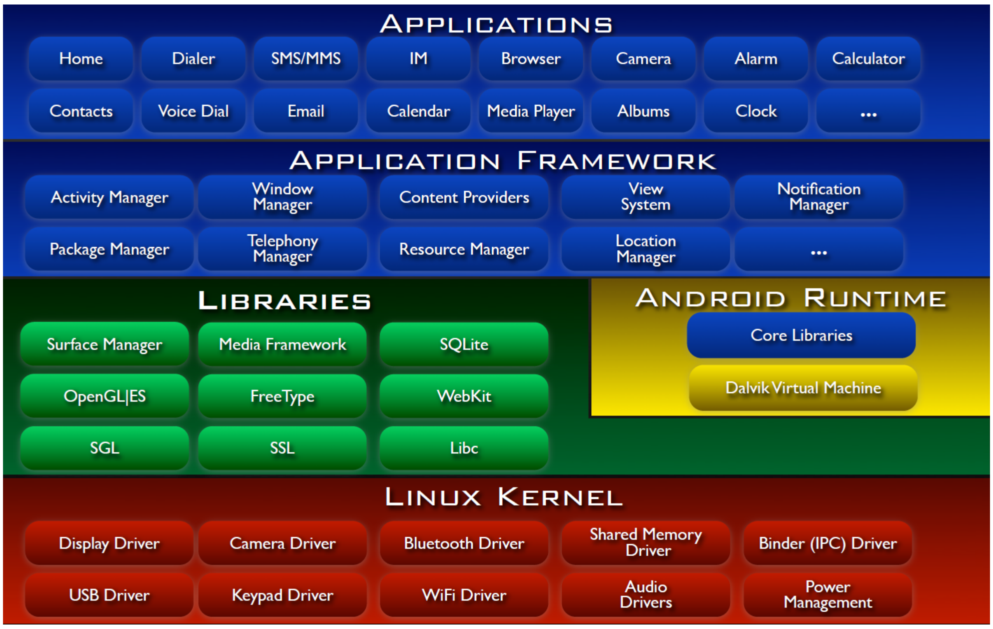


Figure 5.5: Android software stack. Source: Google, 2015a

[16] cf: Google, 2015b

[17] cf: Statista, 2015

At the bottom of the stack is the Linux kernel, which is responsible for low level operations like memory management, process management or drivers. Basically, the kernel is the abstraction between hardware and software.

The layer above is called ”Libraries”. Those libraries are written in C/C++ and can be used by developers of Android apps. One important library for this thesis is the SQLite module, for instance.

The layer to the right of the library layer is called ”Android Runtime”[18]. This layer transforms the byte code into native instructions.

The ”Application Framework” layer contains all the libraries that can directly be used with the App.

At the top of the stack are the applications itself. For instance, the already installed applications from Google and also the self-developed apps.

Components

In order to develop an Android app, there are several necessary components [19].

Activity:
One main component is called ”Activity”. An activity represents a single screen. The main purpose of an activity is to offer a user interface.

Services:
Other important components are services. A service is used to perform long running tasks and therefore runs in the background. For example, a service can be a task to synchronize the completed checklists of a substation.

Content providers:
The purpose of content providers is to handle app data. For instance, a

[18] Google, 2015e.

[19] cf: Google, 2015c

content provider is responsible for storing data in an SQLite database.

Broadcast receivers:
When using an app, it is possible to receive broadcasts from the system. For example, such broadcasts can be notifications concerning low energy or that the screen has been turned off.

Evaluation Criteria

Access internal database The first evaluation criterion is the possibility to access the internal database of the device. The software stack of Android already comprises a library for accessing the SQLite database. Therefore, when using native development it is possible to use this module for handling database access. The following examples are taken from the official Android website [20] and have been slightly modified. They provide an overview on working with the device's internal SQLite database.

First of all, the database scheme has to be determined. When using the method of native developing, this can be done by creating contract classes. The following example shows how to define the name and columns of a table:

```
public final class SubstationContract {

  public SubstationContract()  { }

  public static abstract class Substation implements
    BaseColumns {
      public static final String Table_Name =
        "substations";
      public static final String Column_Name_Id =
        "substationId";
      public static final String Column_Name_Name =
        "Graz";
  }
}
```

[20] cf: Google, 2015d

After defining the contract class, the ”create” and ”delete” statements can be determined as follows:

```
private static final String SQL_CREATE =
  "CREATE TABLE " + Substation.Table_Name + " (" +
  Substation.Column_Name_Id + " INTEGER PRIMARY KEY," +
  Substation.Column_Name_Name + " TEXT)";

private static final String SQL_DELETE =
  "DROP TABLE IF EXISTS " + Substation.Table_Name;
```

The most important task is to set up the communication with the database. To achieve this, it is recommended by Google to write a helper class for each database. Android offers a set of APIs in the `SQLiteOpenHelper` class. The following example shows such a helper class that implements the `SQLiteOpenHelper` class.

```
public class SubstationDbHelper extends SQLiteOpenHelper {
  public static final int Db_Version = 1;
  public static final String Db_Name = "Substation.db";

  public SubstationDbHelper(Context context) {
    super(context, Db_Name, null, Db_Version);
  }
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(SQL_CREATE);
  }
  .....
}
```

In the example, the function `onCreate()` is overwritten and executed when the database is created for the first time. The `SQLiteOpenHelper` class contains functions like `getWritableDatabase()`, which can be used to get an `SQLiteDatabase`- Object, or `getReadableDatabase()` to receive a read only database object.

Writing data to the database is now easily possible by instantiating the helper class and inserting new rows in the database.

```
SubstationDbHelper helper =
  new SubstationDbHelper(getContext());
SQLiteDatabase db = helper.getWritableDatabase();

//Map with values - column names are keys
ContentValues values = new ContentValues();
values.put(Substation.Column_Name_Id, id);
values.put(Substation.Column_Name_Name, name);

long newRowId = db.insert(
              Substation.Table_Name,
              values);
```

To read from the database, it is possible to use the `query()` method like in the following example:

```
SQLiteDatabase db = helper.getReadableDatabase();

String[] projection = {
  Substation.Column_Name_Id,
  Substation.Column_Name_Name };

String sortOrder = Substation.Column_Name_Name + " DESC";

Cursor cursor = db.query(
  Substation.Table_Name,   //Table name
  projection,              //Columns
  selection,               //Columns for WHERE clause
  selectionArgs,           //Values for WHERE clause
  null,                    //group rows?
  null,                    //filter row groups?
  sortOrder);              //sort order
```

Access camera The second evaluation criterion is about accessing the device's camera. Since it is sufficient to just snap a picture, the already existing camera app can be used instead of creating a new one. Android uses intents for calling external apps and getting their results. An intent is used to start an activity, which can also be part of another app.

The following example shows how to call the already existing camera app and save the result, in this case the picture.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  .....
  // first of all create an intent with existing camera app
  Intent intent = new Intent(
    MediaStore.ACTION_IMAGE_CAPTURE);

  //get the directory to save the picture
  Uri fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE);

  //set filename for picture
  intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);

  //start the activity in order to take a picture
  startActivityForResult(intent,
    CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
}
```

After the user finished taking a picture, the event `onActivityResult()` is triggered and receives the taken picture.

Calling a REST service Another important evaluation criterion is calling a REST service. This can be done with the help of the class `java.net.HttpURLConnection`. The following example shows how to call a REST address and get an input-stream. For this evaluation the results have not been parsed.

```
public void LoadSubstations()
{
  URL restUrl = new URL(
    "http://exampleservice.omicron.at/GetAllSubstations");

  HttpURLConnection connection =
    (HttpURLConnection)restUrl.openConnection();
  InputStream stream =
    new BufferedInputStream(connection.getInputStream());

  //here the stream should be parsed to business objects
  .....
}
```

5.2.3 Hybrid Development

This section describes hybrid app development and contains a detailed evaluation. First of all, a brief introduction provides an overview of the basics of hybrid apps [21]. Then, a detailed analysis of the realization of the evaluation criteria is given.

When talking about a hybrid app, the app is basically developed like websites. In other words, developing hybrid apps means using a combination of web technologies like HTML, CSS and JavaScript. The apps are called hybrid, because the so called "WebView" is not directly hosted in the browser, but within a native application. This aspect allows the app to access the device's internal hardware, which is not possible by using a common browser. Another very important difference between a website and a hybrid app is their distribution capability. A hybrid app can be distributed via the official app store, while a website has to be hosted on a web server. On the other side when comparing hybrid apps with native apps it is possible to share code across different platforms when using the hybrid approach.

As part of this thesis, a popular framework, called "Apache Cordova"[22] for developing hybrid apps is evaluated. The following section describes the framework and how it works [23].

Since Apache Cordova is a framework for hybrid app development it is a combination of native app development technologies and web technologies. Apache Cordova contains the following components:

1. Native application container
2. Core APIs
3. Additional tools

The native application container depends on the platform where the app is executed. Its purpose is to render the whole app.

[21] cf: Bristowe, 2015

[22] Cordova, 2015.

[23] cf: Wargo, 2014

The core libraries are part of the native application container, written in `Javascript` and have access to the native device capabilities. Corresponding to these libraries there are native implementations, which communicate directly with the hardware. Currently, the official Apache Cordova website lists the following APIs: Battery Status, Camera, Console, Contacts, Device, Accelerometer, Device Orientation, Dialogs, FileSystem, File Transfer, Geolocation, Globalization, InAppBrowser, Media, Media Capture, Network Information, Splashscreen, Vibration, StatusBar, Whitelist and Legacy Whitelist.

The additional tool set helps the developer with tasks like building native applications.

As stated on the Apache Cordova website[24], the framework supports the following platforms: Amazon Fire OS, Android, BlackBerry10, Firefox OS, iOS, Ubuntu, Windows Phone 8, Windows and Tizen.
When developing a Cordova application, the first step is to create a web application. After this task is completed, the web application has to be packed into the native container before it can be distributed to the users. The following illustration gives an overview on the process for building a Cordova app.

[24] Cordova, 2015.

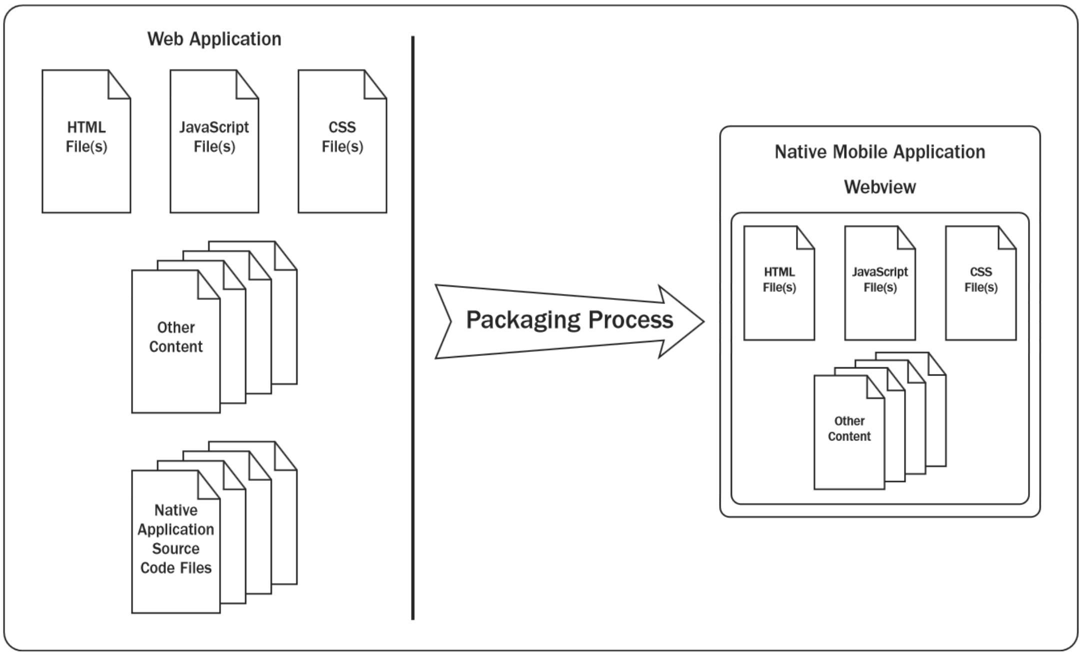


Figure 5.6: Building a Cordova app. Source: Wargo, 2014

On the left hand, the web application side, are the components of the web technologies like HTML, JavaScript and CSS files and the native application source code files to access the native device APIs. When starting the packaging process of the web application, the whole set of components gets packed and installed into a native app container.

Evaluation Criteria

Access internal database To get access to the internal SQLite database, a third party plugin has to be used. Among other plugins, the website http://ngcordova.com/docs/plugins/sqlite/ also contains the sqlite plugin reference[25]. This plugin can easily be installed by executing the following command and is compatible with Android, iOS and Windows Phone:

```
cordova plugin add
    https://github.com/litehelpers/Cordova-sqlite-storage.git
```

The following example illustrates how to access the database and execute a, SQL query.

```
module.controller("DatabaseController",
  function($scope, $cordovaSQLite) {

  var database = $cordovaSQLite.openDB(
      { name: "substation.db" });

  $scope.execute = function() {
    var query =
        "SELECT Name FROM Substations WHERE Id = 1";
    $cordovaSQLite.execute(database,
        query).then(function(result) {
      ... }, function(error) {
      console.error("Error");
    });
  };
});
```

[25] Brody, 2015.

Access camera The camera access evaluation criterion can be fulfilled too by using an additional plugin. Like the database plugin, this one also needs to be installed via the following command:

```
cordova plugin add org.apache.cordova.camera
```

In order to take a picture the plugin provides a `getPicture(...)` function. This is illustrated in the example below:

```
module.controller("CameraController",
  function($scope, $cordovaCamera) {
    document.addEventListener("deviceready", function() {
      var options = {
        quality: 40,
        sourceType: Camera.PictureSourceType.CAMERA,
        destinationType: Camera.DestinationType.DATA_URL,
        ....
      };

      $cordovaCamera.getPicture(options).then(
          function(image){
            var picture = document.getElementById("Photo");
            picture.src = "data:image/jpeg;base64," + image;
           }, function(error) {
          }
      );
    },false);
}
```

Call a REST service Since hybrid apps are mainly developed with `Javascript` it is easily possible to create a REST call using the `$.ajax()` method. The following example shows how to call a REST service and receive the result in JSON-format:

```
$.ajax({
  type: "GET",
  dataType: "json",
  url: "http://exampleservice.omicron.at/GetAllSubstations",
  success: successFunction,
  error: errorFunction
});
```

5.2.4 Xamarin

This section describes the development of mobile apps with Xamarin. First, an introduction on Xamarin [26] provides a basic overview on the product and how it works. Then, the evaluation criteria are analysed and tested.

Xamarin is a company that offers an intuitive new product for developing mobile apps. The key concept of Xamarin is the development of mobile apps for iOS, Android and Windows Phone by using one common programming language - C#.

Architecture To support the development for Android and iOS, Xamarin offers two products: ”Xamarin.iOS” and ”Xamarin.Android”. Both products are based on Mono.

Mono is an open source platform and based on the .NET framework. The correlated .NET implementation is based on the ECMA standards for C# and the Common Language Infrastructure. Mono comprises the following components:

- C# Compiler
- Mono Runtime
- Base Class Library
- Mono Class Library

A more detailed description of these components can be found on the official website.[27]

[26] cf: Xamarin, 2015c

[27] Mono, 2015.

Cross platform - Code sharing Xamarin offers two options to share code among different platforms (iOS, Android and Windows Phone): ”Shared Projects” and ”Portable Class library” [28] [29]. Basically, each platform target requires its own application project that references either as a shared project or a portable class library. For instance, when developing an app for Android and iOS, there is one specific project for each of these platforms and one shared project. The shared project can be used by both platform specific projects. The following illustration shows the architecture, which is used to share code among the three different mobile platforms.

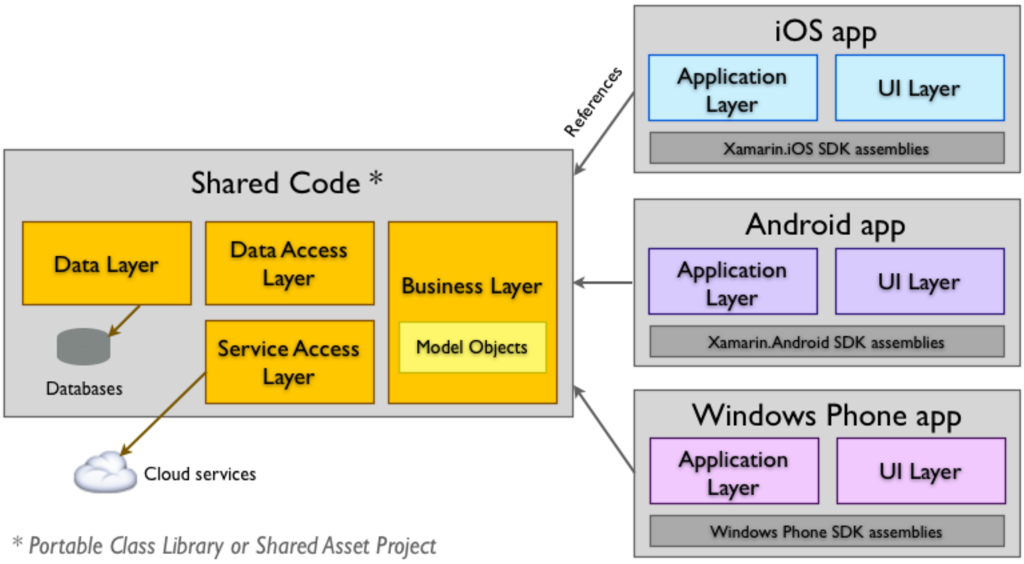


Figure 5.7: Architecture - Shared Code. Source: Xamarin, 2015b

[28] cf: Xamarin, 2015b

[29] cf: Hermes, 2015, p. 367

Shared Projects One way to share code across different platforms like iOS, Android and Windows Phone is to use the shared project approach. [30] The following illustration gives an overview on the architecture of shared projects. The most important aspect on this approach is that each platform-specific project includes the shared project. This means, when compiling the platform-specific project, the shared projects are directly compiled within it. This behavior is illustrated in the picture below. It shows that the shared project is also part of each of the individual platform application projects.

[30] cf: Hermes, 2015, p. 382

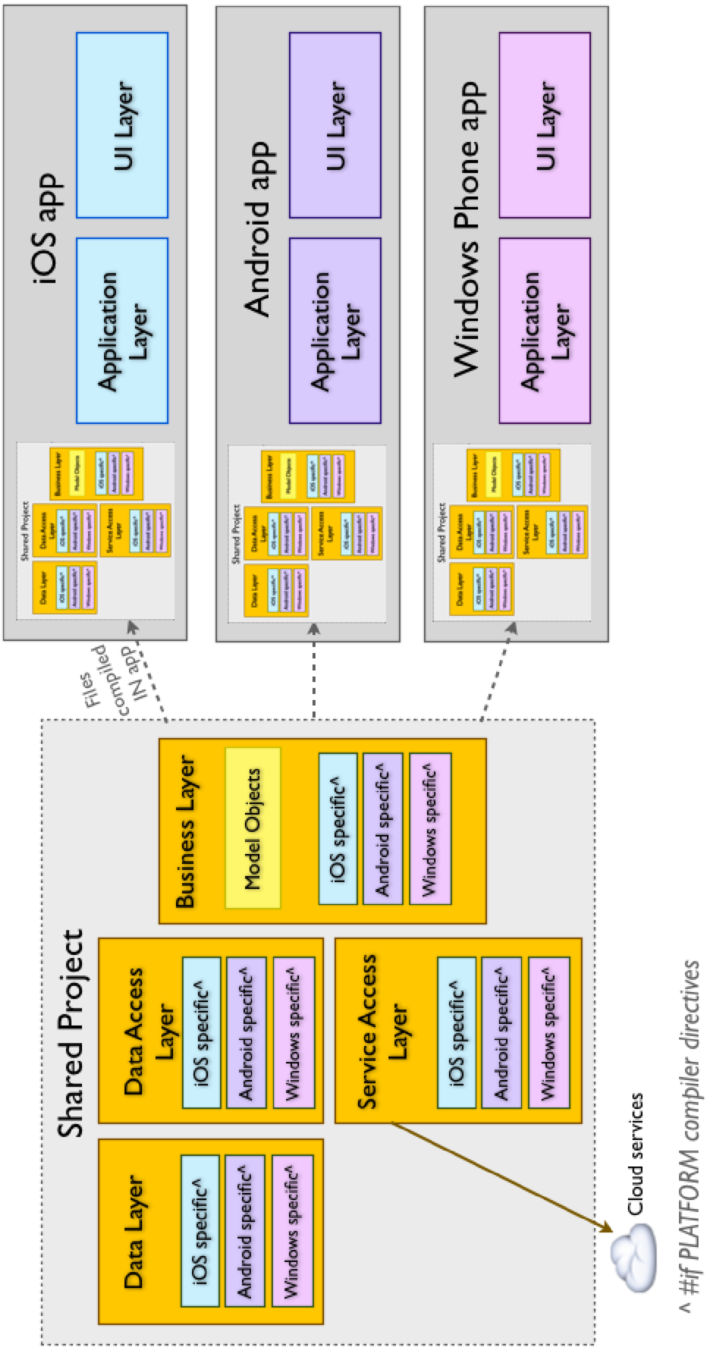


Figure 5.8: Shared Project. Source: Xamarin, 2015b

There are some benefits, but also disadvantages when using shared projects. One benefit is that the shared code can also contain platform-specific code by using compiler directives, for instance, `#if __ANDROID__`. A disadvantage is that a shared project does not produce an output assembly since it is compiled directly within the platform-specific application project.

Portable Class Library Another approach for sharing code across different platforms is the use of portable class libraries [31]. These libraries define the supported target platform, because each of the platforms contains different base class libraries. This means, the set of base class libraries for targeting Android and iOS does only contain libraries that are available on both platforms. It is possible to define a combination of the supported platforms.

The main difference between shared projects and portable class libraries is that portable class libraries produces their own DLLs when the projects are built. The following illustration gives an overview on the architecture for using portable class libraries among different platforms:

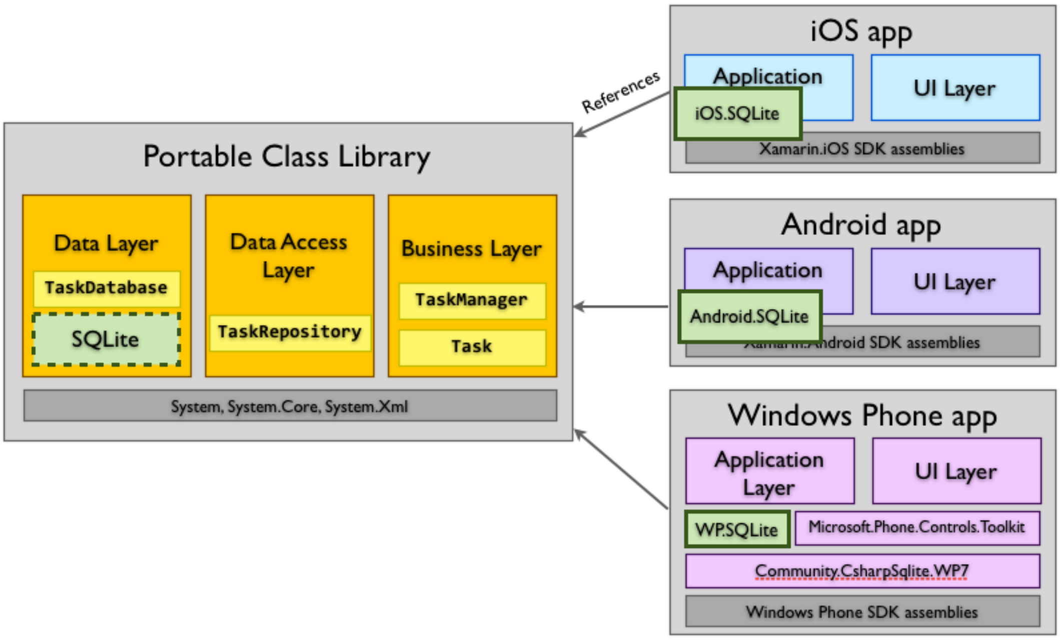


Figure 5.9: Portable class library. Source: Xamarin, 2015b

When the portable class library targets all three platforms, it can be referenced in the respective application project. There are also some advantages and disadvantages when using this approach. One main advantage is that

[31] cf: Hermes, 2015, p. 373

when the code in the portable class library is refactored, also the corresponding references are updated. A disadvantage is that, depending on the selected platforms, only the base class libraries are available.

Xamarin Forms Xamarin offers a toolkit called Xamarin Forms [32] [33], for code reusability across iOS, Android and Windows Phone. With this toolkit it is possible to share user interfaces across these platforms. A special key feature of this framework is that the rendering process of the user interface uses native controls for the corresponding platform. Hence, it is possible to share user interface code and still get a native looking app.

Xamarin Forms offers two possibilities for creating user interfaces. It is possible to define and structure the whole user interface either with source code or by using XAML (Extensible Application Markup Language). XAML was developed by Microsoft and can be used to define a user interface. The following illustration shows an example of Xamarin Forms running on iOS, Android and Windows Phone:

[32] cf: Xamarin, 2015a

[33] cf: Hermes, 2015, p. 368

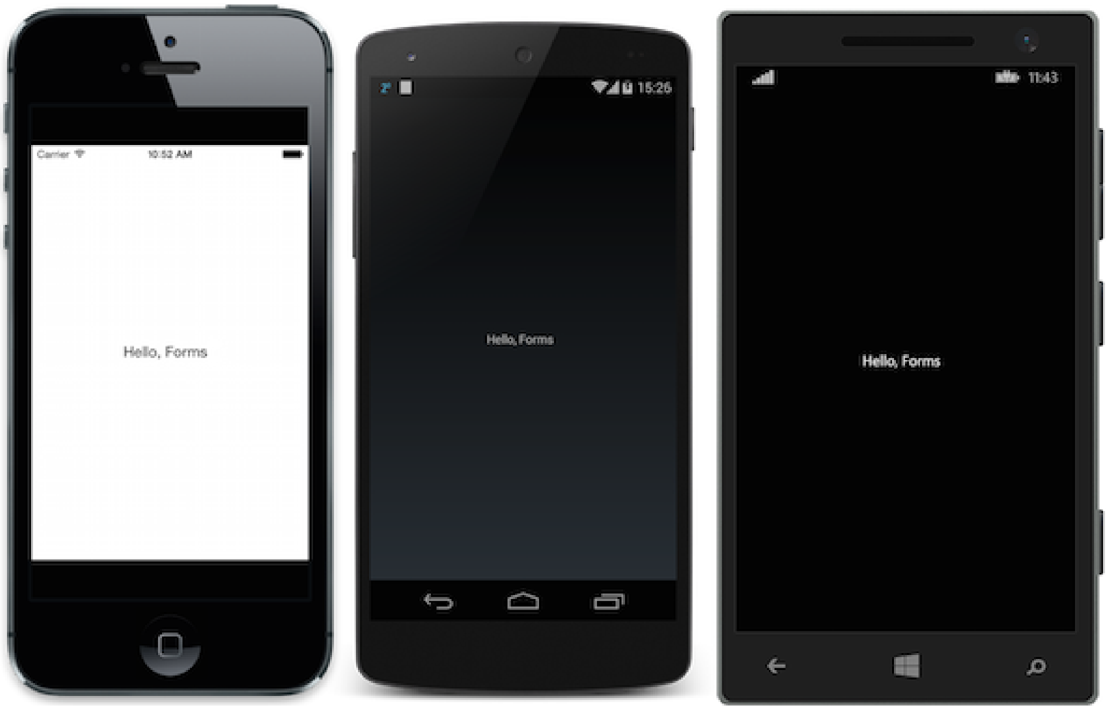


Figure 5.10: Xamarin Forms. Source: Xamarin, 2015a

Evaluation Criteria

Access internal database To get access to the internal sqlite database of the device, it is necessary to use an additional third party tool. Krueger Systems, Inc. developed this tool, which is available on https://github.com/praeclarum/sqlite-net.

The first step is to create the database. This can be done via a simple line of code:

```
var database = new SQLiteConnection(databasePath);
```

The parameter `databasePath` refers to the directory where the `SQLite` file is stored. This path is different on each platform and can be defined by the developer.

After creating the database, it is possible to insert items. The tool uses ORM (Object Relational Mapping), which enables the developer to directly store the object in the database instead of writing additional SQL statements. To illustrate this workflow, the following example shows an object that is stored in the database and afterwards read from the database. The first part explains how to define an object:

```
public class Substation {
  [PrimaryKey, AutoIncrement, Column("id")]
  public int Id { get; set; }

  [MaxLength(8)]
  public string Name { get; set; }
}
```

The third party tool also supports annotations regarding the database. In this example, an attribute is used for the primary key and to limit the length of the name.

The next step is to create an object, to create a table for the substation class and to store the object in the database:

```
var substation = new Substation();
```

```
substation.Name = "Boston";

var database = new SQLiteConnection(databasePath);

//create a table for the substation class
database.CreateTable<Substation>();

database.Insert(substation);
```

The next step is to get the stored substation from the database. There are multiple methods of how to get an entry from the database. For instance, it is possible to use sql queries for every database operation. The following example shows a method to get an object from the database, when the primary key is known:

```
var storedSubstation = database.Get<Substation>(1);
```

This call returns a substation object with the ID 1.

Access camera Xamarin also enables the developer to access the internal hardware of the device, such as the camera module. Since the camera access differs when using another platform, this section is more platform-specific. Due to the fact that the main target is Android, the following example shows Xamarin code for accessing the camera within an Android application. In this example, an already existing camera app is used, because it is not part of this thesis to write an own implementation for the camera.

```
private void TakeAPicture(object sender,
    EventArgs eventArgs) {
  Intent cameraApp =
    new Intent(MediaStore.ActionImageCapture);
  File picture =
    new File(directory, "picture.jpeg");
  intent.PutExtra(MediaStore.ExtraOutput,
    Uri.FromFile(picture));
  StartActivityForResult(intent,0);
}
```

Starting external activities is nearly the same as in native Android development. The result is also captured by an event called `OnActivityResult`.

Call a REST service Another important evaluation criterion is the capability to call REST services. Xamarin is able to use REST and also SOAP. In order to use a REST service, an asynchronous task is created like shown in the example below:

```
private async Task<JsonValue> GetAllSubstations() {

  sting restUrl =
    "http://exampleservice.omicron.at/GetAllSubstations";

  //define the request for the rest service
  HttpWebRequest request =
    (HttpWebRequest)HttpWebRequest.Create(new Uri(restUrl));
  request.Method = "GET";
  request.ContentType ="application/json";

  using(WebResponse response = await
    request.GetResponseAsync())
  {
    using(Stream stream = response.GetResponseStream())
    {
      return await Task.Run(() => JsonObject.Load(stream));
    }
  }
}
```

6 Comparison and Decision

This chapter presents a comparison between the different technologies evaluated in the previous chapter. It focuses on mobile technologies since there is no need for a comparison regarding the service layer.

6.1 Comparison

The previous chapter described the evaluation of native, hybrid and Xamarin app development based on the necessary requirements. These previously defined requirements are: access to the database, access to the camera and using a REST service. An additional optional requirement, namely cross-platform capability, is also mentioned.

All three technologies satisfy the defined requirements as already analysed in the previous chapter. Hence, the comparison is not based on the technical feasibility of the defined criteria, but on the details of the implementation itself. The following paragraphs describe similarities and differences when implementing the defined requirements in all three development approaches.

Access internal database The first evaluation criterion is the access to the internal database. One big advantage of the native development approach compared to the others is that the software stack of Android already contains the SQLite library for the communication with the database. By choosing the hybrid or the Xamarin approach the developer has to involve a third party library to meet this criterion.

Another major difference is the implementation itself. The library used with Xamarin offers the developer the possibility to use ORM (object-relational-mapping) out of the box.

As a reminder, ORM enables the developer to create tables based on the properties of a coded class. Hence, it is possible to read and write entire objects to and from the database. In contrast, standard SQL queries need to define the columns and their values within a query. One advantage of using object relational mapping is that whenever the object is being changed, only the related class needs to be adapted. When using SQL queries, all queries that depend on the object need to be updated.

The libraries used in native or hybrid development do not support ORM functionality out of the box.

Access camera The second evaluation criterion is access to the camera. Basically, all three technologies are able to access the camera. Regarding this criterion the native and the Xamarin approach are nearly identical. Both of them already support camera access out of the box and both are using ”intents” to call the existing camera app for taking pictures. In contrast, the hybrid approach requires an additional plugin to access the camera and to take pictures.

Consuming REST service The third evaluation criterion is the use of a REST service. All three technologies basically support this criterion. The native and the Xamarin approach are again nearly identical. Both are using a library which offers the functionality for accessing the HTTP traffic and for processing the corresponding stream. The hybrid approach is mainly written in javascript and therefore basic `ajax` calls are straight forward.

Cross-platform capability (optional) An optional criterion is cross-platform capability. Obviously, the native approach can not be used on different platforms. Hence, each of the supported platforms has to be implemented in its native coding language. In contrast the hybrid and the Xamarin technologies are suitable for cross-platform usage.

6.2 Decision

The first criterion - access internal database - can be realised by all three technologies, but only Xamarin supports ORM out of the box. This aspect is important for the final decision because using ORM enables better maintenance of the app and supports the developer when creating or modifying the database.

The hybrid approach needs an additional plugin for accessing the camera, therefore the preferred technology is either native or Xamarin, since both of them are nearly identical regarding camera access.

The third criterion is the use of a REST service. This can easily be realized by all three technologies and therefore, there is no preferred technology regarding this criterion.

Also included in the decision is the optional criterion, the cross-platform capability. Since OMICRON might want to run the app also on the iOS platform at a later point of time, it is desirable to choose a technology with which this is possible. Therefore, only the hybrid and the Xamarin technologies qualify.

In regard to the requirements, all three technologies could be used to implement the app. However, the decision was made to use the Xamarin technology. When comparing Xamarin to native development, there is no Android functionality within the scope of this thesis, that can only be used by its native implementation. In fact, Xamarin covers all necessary native functionality and is additionally platform-independent.
The hybrid and the Xamarin technology are nearly equal regarding their functionality. The decision to use Xamarin instead of the hybrid approach is based on prototyping experience. It showed that Xamarin offers a more comfortable programming environment, especially when considering that OMICRON developers are far more familiar with the .NET programming language. Therefore the proper choice is Xamarin.

7 Screen Design

The first task when implementing a mobile app is to create the screen design. The process, that leads to the final design of the app follows three different steps:

1. Flowchart
2. Scribble
3. Wireframe + Style guide

In the following paragraphs, these three steps are described in regard to creating the screen design for one of the features of the app: the creation of a new inspection event.

Flowchart First, a flowchart is drawn to visualize the workflow for creating a new inspection event. Basically, the flowchart defines what the user needs to do in order to create such an event [1]. The following list gives an overview of the necessary steps:

1. Select a template.
2. Fill out the checklist.
3. Enter the name of the checklist.
4. Need to create another checklist?
   a) Yes: Go back to 1).
   b) No: Continue.
5. Save the inspection event.

[1] cf: Jacobsen, 2014, p. 109

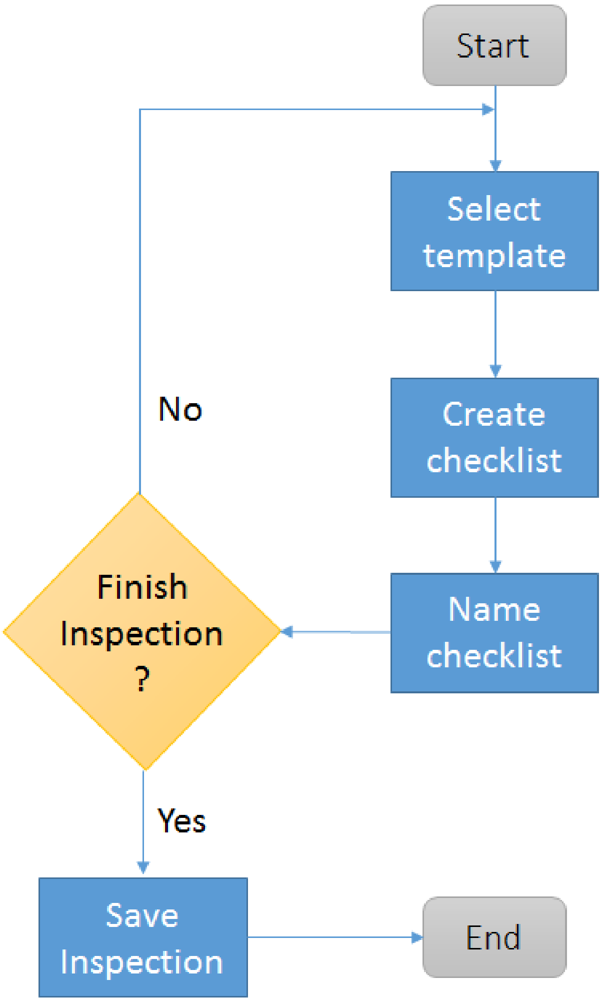


Figure 7.1: Flowchart. Source: Own representation

Scribble When the flowchart is done, the next step is to create a scribble. Basically, a scribble can be created with minor effort by using pen and paper. It is useful to get a better overview of a specific page of the app and its components.[2]

The following figure illustrates a scribble based on one step of the flowchart:

[2] cf: Jacobsen, 2014, p. 150

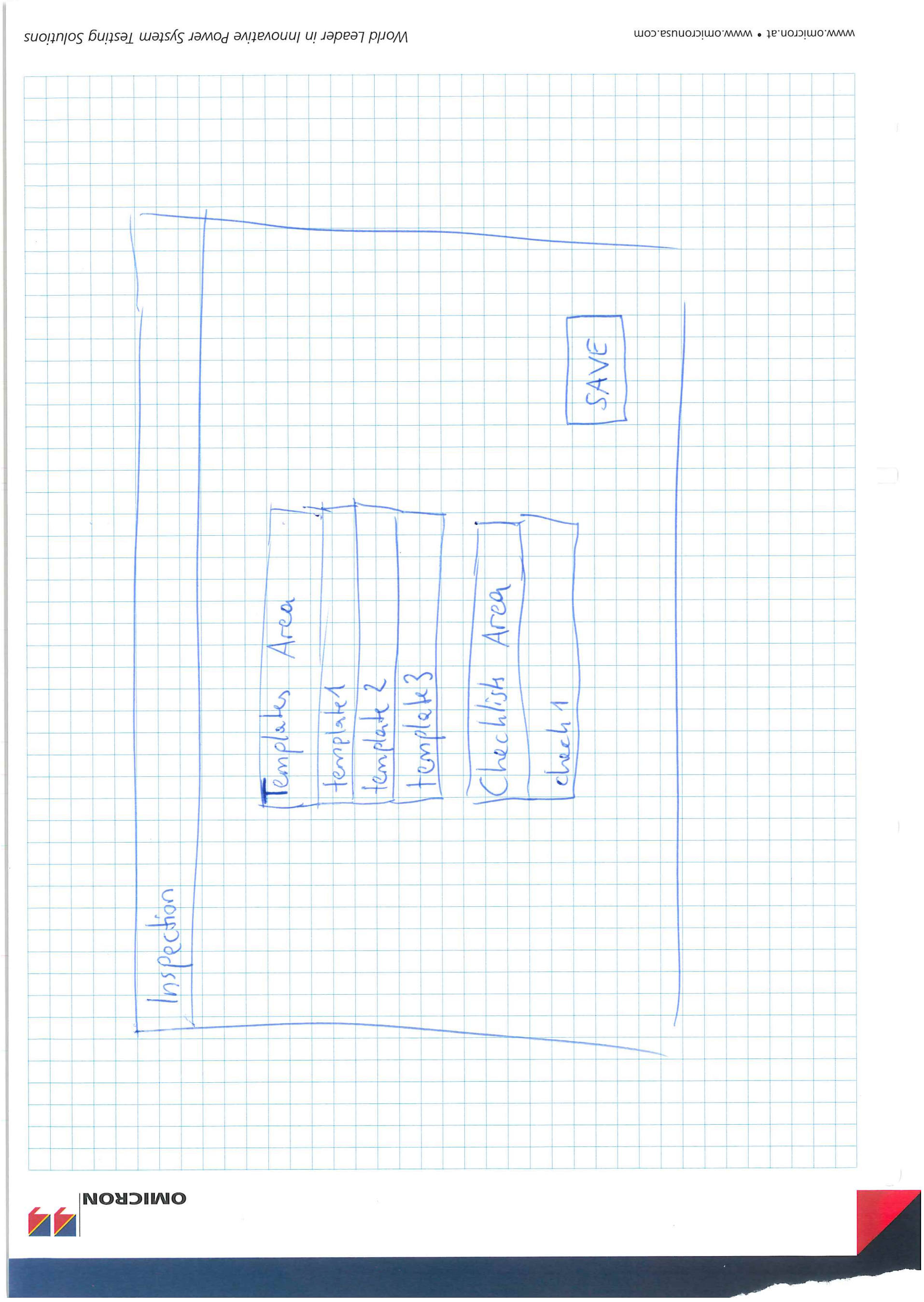
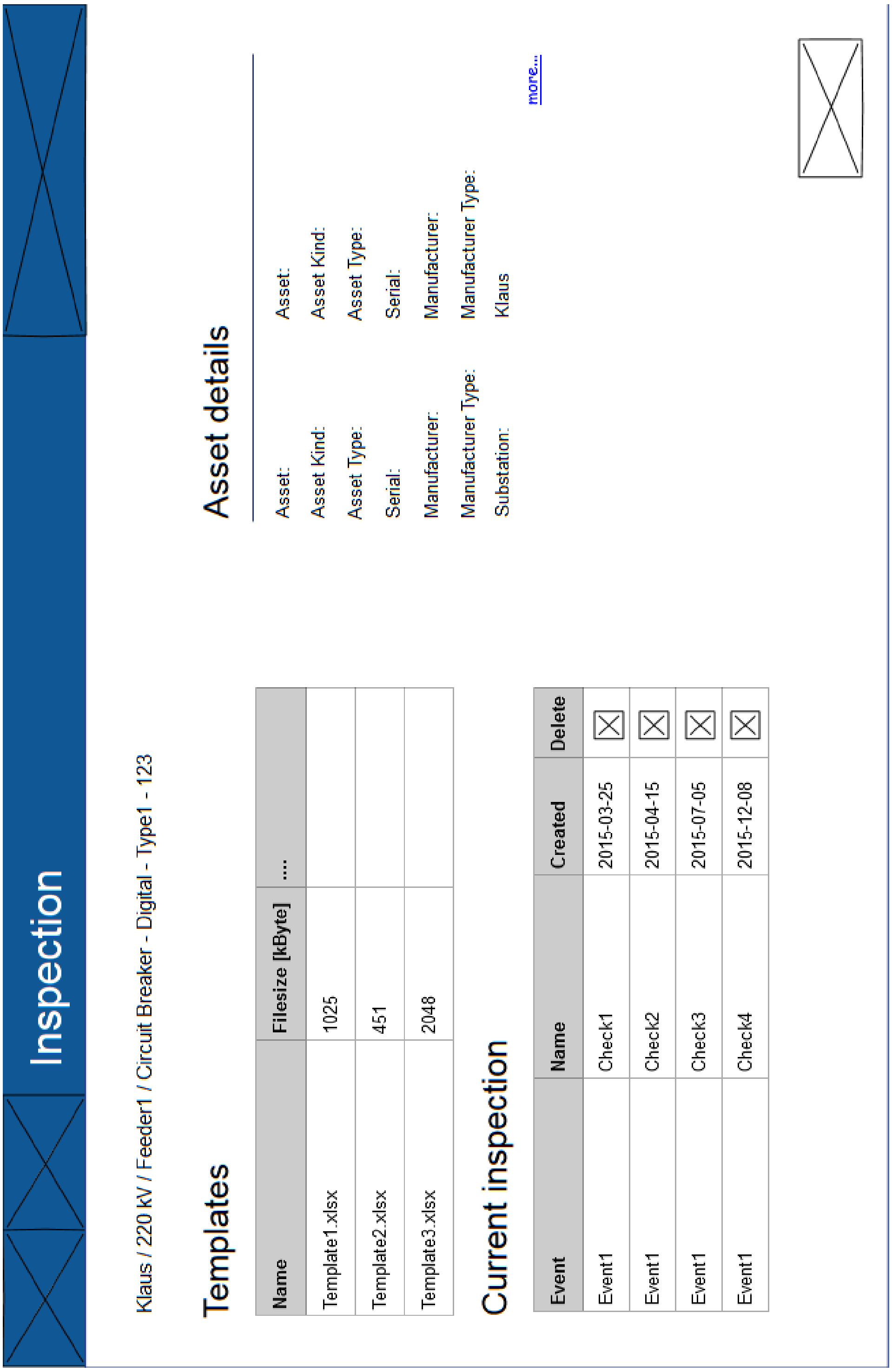

Inspection
Templates Area
template1
template 2
template3
Checklist Area
check 1
SAVE
World Leader in Innovative Power System Testing Solutions
www.omicron.at • www.omicronusa.com
OMICRON

Figure 7.2: Scribble. Source: Own representation

Wireframe At this point, the illustrated workflow can be verified by the customer. The next step is to create a more detailed illustration called wireframe. It visualizes the components of the elements of the app in a more precise way. However, the final position and design definitions are not part of a wireframe [3]. The following picture shows the wireframe corresponding to the requirements, the flowchart and the scribble:

[3] cf: Jacobsen, 2014, p. 151



Inspection

Klaus / 220 kV / Feeder1 / Circuit Breaker - Digital - Type1 - 123

Templates

Name	Filesize [kByte]	...
Template1.xlsx	1025	
Template2.xlsx	451	
Template3.xlsx	2048	

Current inspection

Event	Name	Created	Delete
Event1	Check1	2015-03-25	☒
Event1	Check2	2015-04-15	☒
Event1	Check3	2015-07-05	☒
Event1	Check4	2015-12-08	☒

Asset details

Asset:	Asset:
Asset Kind:	Asset Kind:
Asset Type:	Asset Type:
Serial:	Serial:
Manufacturer:	Manufacturer:
Manufacturer Type:	Manufacturer Type:
Substation:	Klaus

more...

Figure 7.3: Wireframe. Source: Own representation

Style guide The last step to get the final design of the app is to apply the style guide on the wireframe. The style guide describes the basic design of the user interface [4]. The style guide was defined by OMICRON in order to get a similar graphical user interface across all offered software applications. The reason for this is that the software applications should immediately be recognizable as OMICRON products. Therefore, the style guide provides recommendations on how to design buttons, screen layouts and many other design-related elements.

[4] cf: Jacobsen, 2014, p. 322

8 Implementation of Service Layer

One of the most interesting topics regarding the implementation of the service layer is code modularization. As already mentioned, there are several different hosting scenarios that can be used to offer a service interface. Choosing the appropriate code modularization is essential when code ought to be shared between the different hosting scenarios. This chapter gives an overview on the service modules and, in particular, on their dependencies.

The most important code modules are:

- ADMO Core
- Host services
- Service implementation

ADMO Core This module contains the reused code parts of the ADMO client such as the repositories that are responsible for communicating with the database. Basically, the whole data procurement process is implemented within this module. The common interface between the ADMO Core module and the service implementation is the domain object that is defined in the ADMO Core module. Every time the service implementation requests data from the database, the ADMO Core module collects that data and returns the corresponding domain objects. When new data is stored in the database, the service implementation transfers the new data via these domain objects to the ADMO Core module.

Service implementation Unlike the projects that are part of the host services module, the service implementation can be shared between different hosting scenarios. The main purpose of this module is to implement the service contract. This is usually done by only one single class. This class can reference other modularized implementation classes such as the location- or the asset-related implementation.

Host services As described in the Technical Evaluation chapter, a WCF service can be hosted by multiple different hosting approaches. The purpose of the host service module is to serve as a storage container for all necessary hosting specific projects. OMICRON already runs an IIS web server and therefore, the first task was to implement a WCF service that can be deployed to this server. Another reason for using IIS is that OMICRON wants to offer their customers a sample database hosted on Azure Cloud. The Web App hosting approach of Azure Cloud is the most inexpensive one and therefore, an IIS hosted service is needed. In respect to the multiple hosting approaches, the service implementation itself is decoupled from the hosting-dependent project. The advantage of this is that less work is involved in order to provide a different hosting option. In case of the implemented IIS hosting scenario the code-behind file of the `.svc` file is deleted and another shared implementation class is referenced.

The following figure illustrates the dependencies between these modules and shows the interfaces to the client and the database. It is apparent that the host service only needs to know the service implementation. The gap between the database and the service implementation is filled by the ADMO Core module. Therefore, whenever a client requests data, all modules are accessed. When calling the service, the host module gets involved. This module then runs the related implementation, which gathers the data by using the ADMO Core module. Another aspect that is visualized by the figure below is the already mentioned multi-tier architecture.

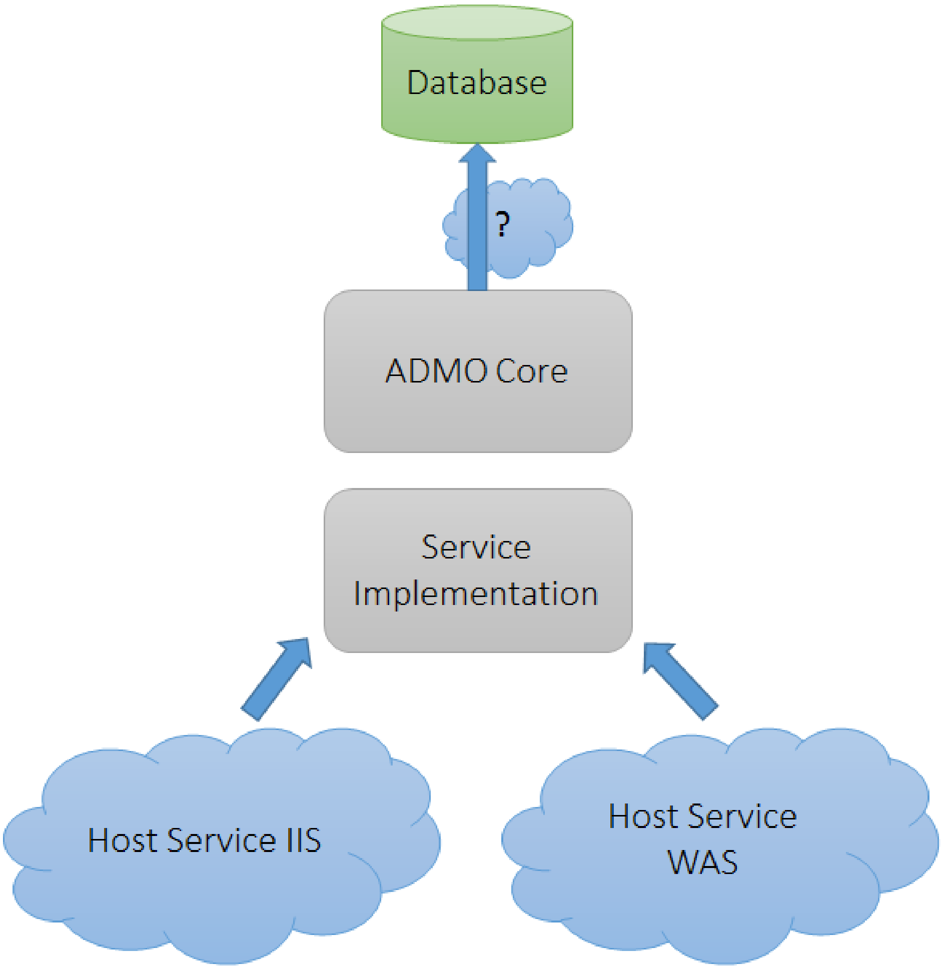


Figure 8.1: Service code modularization. Source: Own representation

9 Implementation of Mobile App

This chapter presents a few selected topics regarding the implementation of the mobile app. The first topic is about code modularization, especially about sharing code across different platforms. Then, a basic explanation and some implementation details about a pattern called Inversion of Control are given. Another pattern, which is also part of this chapter, is called Model-View-ViewModel . The last section of this chapter focuses on data persistence and the synchronization between app and service layer.

9.1 Code Modularization

Basically, the app can be grouped into two different modules. One module contains all platform-specific projects, whereas the other module contains platform-independent Xamarin.Forms projects. The latter one can be referenced by all platform-specific projects.

The interaction between these two modules can easily be explained. Each platform needs some kind of start-up project, which contains all code that is necessary to start the app. When using Android, this start-up project contains the `MainActivity`. These platform-dependent projects can reference the shared Xamarin.Forms projects that contain platform-independent code and the user interface implementation. Xamarin offers the possibility to write most of the code platform-independently. Therefore the start up project has to load a `Xamarin.Form` application. The following code extract shows an example where the `Xamarin.Form` application is loaded by an Android start up project:

```
[Activity]
public class MainActivity: FormsApplicationActivity
{
    // OnCreate of Android's main activity loads
    // Xamarin.Forms application
    protected override void OnCreate(Bundle bundle)
    {
        Xamarin.Forms.Forms.Init(this,bundle);
        LoadApplication(new AppController());
    }
}

// Xamarin.Forms AppController for starting the app
public class AppController : Xamarin.Forms.Application
{
    public AppController()
    {
        MainPage = new HomeScreen();
    }
}
```

However, some functionality, such as native hardware access, has to be implemented independently for each platform. Therefore, the Xamarin.Forms part defines a shared interface, which is implemented for each platform. An example for such a platform-specific implementation is starting the camera. The Xamarin.Forms module defines the `ICameraService` interface, which is implemented by an Android-specific class called `CameraService`:

```
public interface ICameraService
{
    void StartCamera(Action<byte[]> callback);
}

public class CameraService : ICameraService
{
    // returns the picture as byte[]
    public void StartCamera(Action<byte[]> callback)
    {
        Intent intent =
          new Intent(MediaStore.ActionImageCapture);
        Java.IO.File file =
          new Java.IO.File(tempDirectory, fileName);
        intent.PutExtra(mediaStore.ExtraOutput,
          Uri.FromFile(file));
        ((Activity)Forms.Context).
          StartActivityForResult(intent,1);
    }
}
```

In order to implement this interface correctly, Xamarin offers a class called `DependencyService`. This enables the developer to get the right implementation of the corresponding platform. Before using this service the implementation itself has to be registered on each platform. To do this the developer needs to add the following line to the `AssemblyInfo.cs` file:

```
[assembly: Dependency(typeof(CameraService))]
```

Then, the right instance can be resolved with the following syntax:

```
ICameraService cameraService =
  DependencyService.Get<ICameraService>();
```

9.2 Inversion of Control - IoC

Inversion of Control [1] is a pattern to achieve a loosely coupled software system and better testability. The basic principle of this pattern can be described by a famous Hollywood quotation: ”Do not call us, we will call you”. This means, when the Inversion of Control pattern is not used, the usual way to get dependencies is to instantiate their specific implementation. The following example shows two classes and how the SubstationManager class instantiates a Service class.

```
public class SubstationManager
{
  public SubstationManager()
  {
    Service service = new Service();
  }
}
```

By using this approach, the SubstationManager class depends directly on the Service class.

Inversion of Control uses an additional container, which manages the specific implementations of the registered interfaces. This means, the container is able to decide during runtime which implementation should be used. In order to realize this pattern, it is necessary to use interfaces for abstracting the implementations. One common realization of Inversion of Control is called Dependency Injection. It is part of this thesis and therefore, described in more detail.

Dependency Injection is used to inject the necessary dependencies into the particular constructor. An advantage of using such a pattern is better testability, for instance. When testing a class that gets all dependencies injected, it is easy to create a mock-up (simulation of a class) for each of these dependencies. On the other hand, when the dependency gets resolved by instantiating the specific implementation, it is not possible to create a mock-up for this instance. The following example shows how Inversion of Control decouples the SubstationManager class and the Service class by

[1] cf: ITWissen, 2015

using dependency injection. The IService parameter is the interface of the Service class.

```
public interface IService
{
  IList<Substation> GetAllSubstations();
}

public class SubstationManager
{
  public SubstationManager(IService service)
  {
    IList<Substation> substations =
      service.GetAllSubstations();
  }
}
```

The implementation shown above does not have any dependencies in regard to the specific implementation of the service class any more. Therefore, the container decides which implementation is resolved at runtime. The following paragraph gives a brief overview of a framework that is used to realize IoC in the implementation part of the app.

AutoFac AutoFac [2] is an IoC Framework for Xamarin. It offers the developer multiple ways of realizing the IoC pattern. When starting the app, all necessary modules have to be registered at the container.

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<Service>.As<IService>();
IContainer appContainer = containerBuilder.Build();
```

The example above illustrates how to register a Service-specific class as an interface called IService. Then, it is possible to resolve a Service implementation just by knowing the interface.

```
IService service = appContainer.Resolve<IService>();
```

[2] cf: Autofac, 2015

9.3 MVVM - ModelViewViewModel

This section is about the Model-View-ViewModel architectural pattern. The pattern itself consists of three components: the view, the model and the view model. The basic idea of this pattern is to separate the user interface (View) from the domain (Model) by using an additional layer, called the ViewModel. The following figure illustrates the relations between these components and their interactions. Afterwards, a more detailed description of this pattern is given [3].

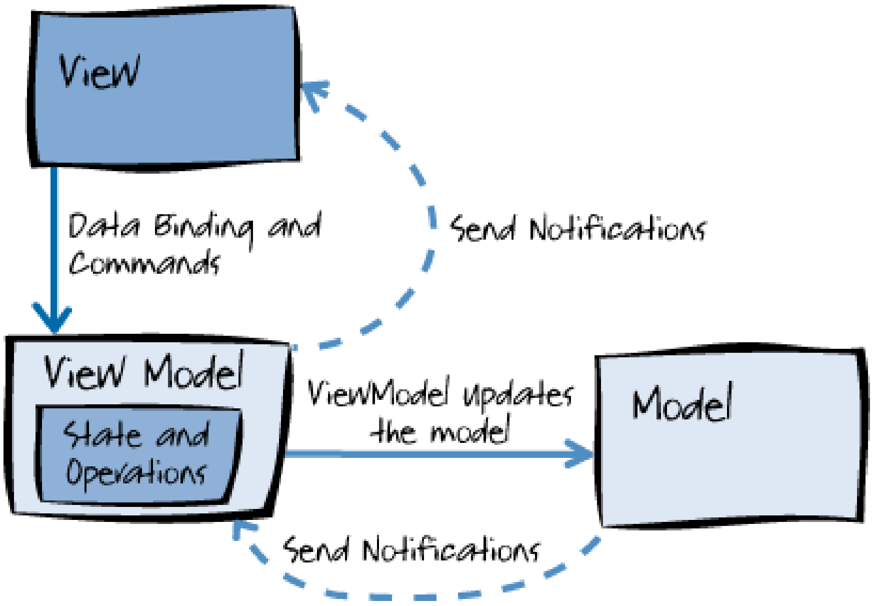


Figure 9.1: Model-View-ViewModel Pattern. Source: Microsoft, 2015g

Model The model represents the domain object. Basically, the model contains the information about an object. For instance, a model for the domain object `Substation` can contain properties like: Substation name, address, city, postal code,...

[3] cf: Microsoft, 2015g

View The view represents the user interface for interacting with the user and displaying all necessary information. Typically, each view is related to a specific view model. A more detailed description of how to create views is given in the paragraph "User Interface".

View model The view model is responsible for connecting all three components like shown in the figure above. One responsibility of the view model is to update the model every time when data is changed. In addition, the view model is responsible to bring the data of the model to the view. Therefore, the view defines view elements with data bindings. Such bindings are used to connect the content of the view element to a property that is defined in the view model. This property typically consists of data from the model.
Handling the behaviour of data bindings is similar to handling interactions with the user. The most popular example for this is the use of a button. The view is responsible for defining the look of the button. Additionally, the view defines the command that is executed whenever the button is pushed. The implementation of this command is done in the view model.
The figure above also shows the notification mechanisms. They are used to notify the relevant component in case of changes. Therefore, when a property of the view model is changed, the corresponding view element is updated accordingly. It is not necessary to render the whole view again.

9.4 User Interface

When using Xamarin, a `View` is called `Page` and represents a single page of the app. A page can comprise several view elements to display data or interact with the user. Xamarin.Forms supports two different methods of designing a page. One possibility is to use code only when the whole page and its components are defined in C# code. Another approach is XAML.

XAML (Extensible Application Markup Language) [4] is developed by Mi-

[4] cf: Microsoft, 2015i

crosoft and based on a markup language called XML. Its purpose is to visualize the application and therefore, to define the page's components. The following example shows how to define a label with XAML:

```
<Label Text="Substation Name:"
  VerticalOptions="Center"
  ... />
```

In contrast, the same example can also be implemented with the code-only approach:

```
Content = new Label {
    VerticalOptions = LayoutOptions.Center;
    Text = "Substation Name:";
}
```

For the implementation of the app, the decision was made to use XAML instead of code-only pages. One reason for this decision is the better readability of XAML files. Furthermore, ADMO also uses XAML for defining the user interface. Thus, already existing knowledge can be used.

ViewModel First There are two commonly methods to realize the navigation in an MVVM project: "View-First" and "ViewModel-First". The obvious difference between these two approaches is that either the view or the view model is called first. One main advantage of calling the view model first is that this makes it possible to directly pass parameters from one page to another. Let us assume that the user wants to navigate from page A to page B. When using the "View-First" approach there is no way to directly pass a parameter to another view. However, when the navigation is done via view models, it is of course possible to pass parameters. For instance, this can be done directly via the constructor. Additional examples can be found on the referenced website[5].

By default, Xamarin uses the "View-First" approach. In order to use the "ViewModel-First" method, the following components need to be implemented:

[5] András, 2015.

- Page factory
- Navigation service
- Registry

The first component is the page factory. This factory is responsible for storing information, about which view model is related to which page. This information is stored in a dictionary. When all pages and view models are registered, the page factory is able to deliver the page corresponding to the view model. The following code extract shows this resolving-functionality. The property called `viewModelInstanceMap`, is the dictionary containing the registered view models and views. The binding between the page and the view model is also done by using the resolving function.

```
public Page Resolve<TViewModel>(TViewModel viewModel)
{
  if(viewModelInstanceMap.ContainsKey(typeof(TViewModel)))
  {
    return viewModelInstanceMap[typeof(TViewModel)];
  }
  var pageType = viewModelMap[typeof(TViewModel)];
  Page page = Activator.CreateInstance(pageType) as Page;
  page.BindingContext = viewModel;
  return page;
}
```

When starting the app, one of the initialization tasks is to register the view models and views. The following code extract illustrates the initialization function of the page factory that registers the inspection view model and the related inspection page:

```
internal static IPageFactory InitializePageFactory()
{
  IPageFactory pageFactory = new PageFactory();
  pageFactory.Register<IInspectionViewModel,
    InsepctionPage>();
  ....
  return pageFactory;
}
```

The last step is to set up the navigation between pages. For that, an additional navigation service is used. The navigation is realized by using a navigation stack. To display a new page, the page has to be pushed to the top of the stack. When navigating backwards, the page is popped from the stack. The navigation service implements these push and pop functions and contains the reference to the page factory.

```
public async Task<IViewModel> PopAsync()
{
  //pops the page from the top of the navigation stack
  Page page = await navigationPage.PopAsync();

  //returns the view model of the page
  return page.BindingContext as IViewModel;
}

public async Task<TViewModel> PushAsync<TViewModel>(
  TViewModel viewModel)
{
  Page page = pageFactory.Resolve(viewModel);

  //Push the page to the top of the stack
  await navigationPage.PushAsync(page);
  return viewModel;
}
```

The example above shows the push and pop function of the navigation service. The developer only needs to know the view model of the new page. By passing the view model to the `PushAsync` method the corresponding page is resolved by the page factory and pushed to the top of the stack. The next example focuses on the navigation to the inspection page. The first step is to get an instance of the view model. For this, the navigation service contains the function `PrepareViewModel`, which resolves the specific implementation for the given interface. This function is necessary because only the navigation service contains the IoC-container. This means, the developer gets an instance of the given interface of the view model and can continue with loading the data to the view model. After the view model preparation is finished, it is pushed to the navigation stack.

```
IInspectionViewModel viewModel =
  navigationService.PrepareViewModel<IInspectionViewModel>();

viewModel.LoadDetails(selectedAssedId, currentLocationId);

navigationService.PushAsync(viewModel);
```

9.5 Data Consistency and Synchronization

Since the mobile app implementation needs to store a lot of data locally and to synchronize new inspection events with the service layer, this chapter describes the implementation details of data consistency and synchronization.

For storing data on the device, an SQLite database is used. Both, Android and iOS support this type of database and provide access to this module. The implementation of the platform specific database access is done via a shared interface and the `DependencyService` class. The Android specific implementation can be achieved with the following code extract:

```
public class DatabaseService : IDatabaseService
{
    public SQLiteConnection GetConnection()
    {
        string folder =
          Environment.GetExternalStoragePublicDirectory(
          Environment.DirectoryDownloads).Path;

        string databasePath =
           Path.Combine(folder,"ADMODatabase.db3");
        return new SQLiteConnection(databasePath);
    }
}
```

When using the `SQLiteConnection`, which is returned by the `DatabaseService` it is possible to query data from the database or to manipulate it. Since the SQLite.Net component supports ORM, it is possible to define the database scheme by objects as described in the Technical Evaluation chapter.

Synchronization Whenever a tester has completed an inspection event, it is possible to synchronize this inspection event, including all of its attachments, with the service layer. When the synchronization process is finished, the inspection event is available within the ADMO client. The synchronization process performs the following steps:

1. Delete tables on the local database.
2. Retrieve all data from the service layer.
3. Store these data in the local database.
4. Synchronize finished inspections.
5. Delete synchronized inspections.

First, the synchronization process deletes everything on the local database except the inspection events and their attachments. Then, the new data of the service layer is stored locally. The last step is to upload the completed inspection events, including their attachments to the service and to delete them afterwards. This process was chosen, because only new data gets stored in the productive database and therefore, no conflicts can appear.

10 Continuous Integration and Testing

One very important task when developing software is testing. This chapter presents a developing process called ”Continuous Integration” (CI) and also some details about the testing strategy of ADMO Inspect.

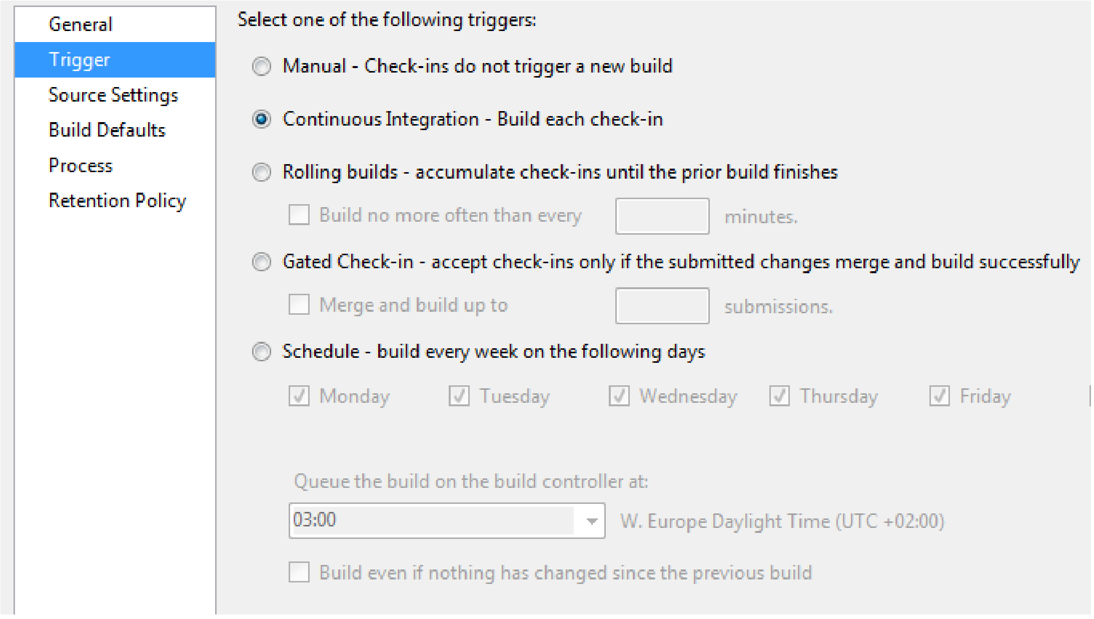
Continuous Integration Martin Fowler describes Continuous Integration in the following way:[1]

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. ...

When realizing CI, two components are necessary: an automated build infrastructure and a set of tests. Automated builds means that whenever a developer makes changes to the source code, the tests to check whether the changes affect the correct behaviour of the application are executed automatically. The following paragraph describes how such an infrastructure was used during the implementation of ADMO Inspect. Afterwards, a detailed overview of the testing strategy is given.

[1] Fowler, 2015.

Build infrastructure In order to support automated builds, OMICRON uses the (TFS) Team Foundation Server approach from Microsoft. TFS also provides a version control system, which manages the changes of the source code. Each software project can comprise several build definitions, with which it is possible to parameterize the build. A build parameter can be the trigger for the build, for instance. When using CI, TFS already offers a suitable build trigger as shown in the figure below:



Select one of the following triggers:

○ Manual - Check-ins do not trigger a new build

● Continuous Integration - Build each check-in

○ Rolling builds - accumulate check-ins until the prior build finishes

□ Build no more often than every minutes.

○ Gated Check-in - accept check-ins only if the submitted changes merge and build successfully

□ Merge and build up to submissions.

○ Schedule - build every week on the following days

☑ Monday ☑ Tuesday ☑ Wednesday ☑ Thursday ☑ Friday

Queue the build on the build controller at:

03:00 W. Europe Daylight Time (UTC +02:00)

□ Build even if nothing has changed since the previous build

Figure 10.1: TFS-Build configuration. Source: Screenshot Microsoft Visual Studio 2013

For a better understanding of the build process at OMICRON, the following example is given. A developer adds a new functionality to a software that is managed by TFS. To check if the new functionality is working properly, the developer also writes some tests for this functionality. After everything is implemented, the changes are checked in to the version control system. Whenever a check-in is done the trigger of the build definition, which is configured as ”Continuous Integration” build, is executed. This means, the whole software is being built and the tests that are defined in the build definition are executed. The main purpose of the last step is to check whether a new functionality leads to any unwanted behaviour of any other component of the software.

Testing strategy Testing is necessary to ensure good code quality. The testing strategy that was chosen for the mobile app is based on a suggestion mentioned in the book ”Agile Testing” [2]. In this book, the authors use a test automation pyramid, which was introduced by Mike Cohn[3]. For the scope of this thesis, a slightly modified version of the pyramid was used. The used testing pyramid is shown in the following figure:

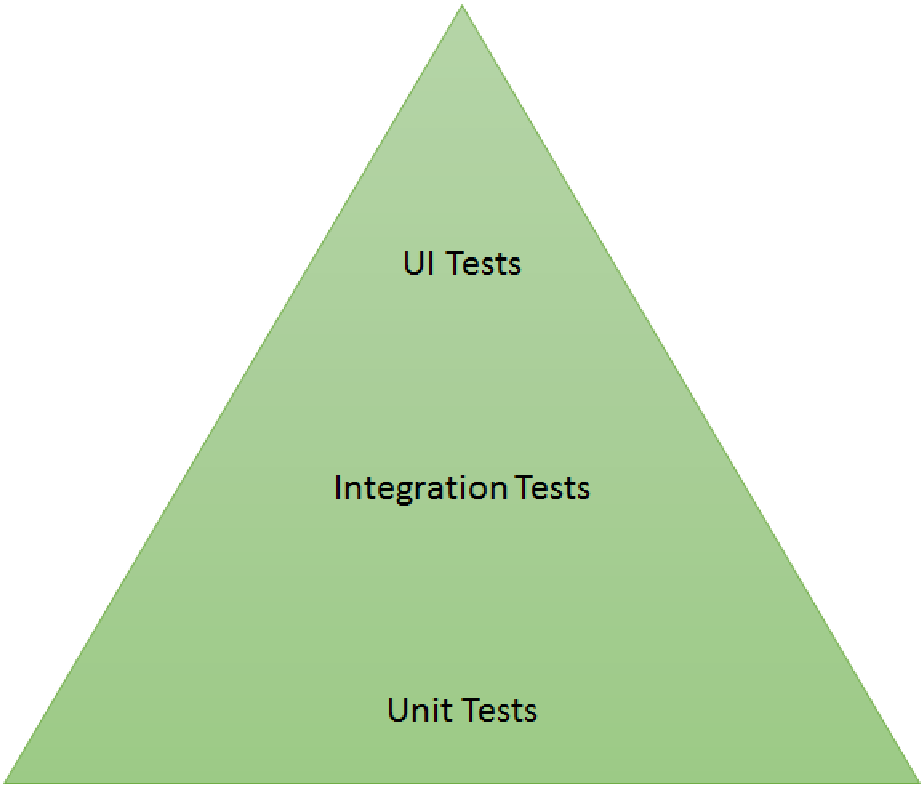


Figure 10.2: Testing pyramid. Source: In dependence on Lisa Crispin, 2009, p. 277

The difference between the used version and the version described in the book is the middle tier. While the authors of the book use a more abstract

[2] cf: Lisa Crispin, 2009, p. 276

[3] Cohn, 2015.

category called ”Acceptance Tests” as the middle-tier layer, this thesis focuses only on a part of this abstraction called ”Integration Tests”.

The main purpose of this pyramid is to visualize the proportion between the amounts of tests for each test category. In respect to this pyramid, most of the tests are unit test, and the least are UI tests. All three test categories that are shown in the pyramid are explained in more detail in the following paragraphs.

Unit test A unit test is the simplest of all three test types. Its purpose is to test a single unit and nothing more. The main advantage of unit tests compared to the other test types is that these tests can be written very quickly and are usually much less complex than the others. Consequently, it is desirable to write as much tests as possible for this testing level. The following code extract shows a unit test of ADMO Inspect:

```
[TestMethod]
public void LoadNewInspection_ShouldSetLastSelectedAsset()
{
    // Arrange
    InspectionViewModel viewModel = new InspectionViewModel(
        navigationServiceMock, fileServiceMock, ....);

    // Act
    viewModel.LoadNewInspection("123", "12354");

    // Assert
    Assert.AreEqual(lastSelectedAsset.GetAssetAsString(),
        viewModel.CurrentAsset);
}
```

The test given in the example above checks if the property `CurrentAsset` is set correctly when the method `LoadNewInspection` is called. An important aspect about this test is that all parameters of the `InspectionViewModel` constructors are mock-ups. This means, the test does not really depend on these parameters and therefore, the real implementation does not affect the test.

Integration test There are some parts of an application that cannot be tested by a unit test. For instance, the database access. In this case, the developer needs to write an integration test that combines several necessary modules and checks their interactions. Another difference between a unit test and an integration test is that an integration test can only be run on a specified platform, whereas a unit test does not rely on any platform details. The following code extract gives an example of an integration test for the Android platform:

```
[Test]
public void GetSubstations_ShouldReturnSubstationList()
{
    // Arrange
    ILocationRepository repository =
        appContainer.Resolve<ILocationRepository>();
    databaseConnection.InsertAll(substations);

    // Act
    IList<SubstationListItem> result =
        repository.GetAllSubstationListItems().ToList();

    // Assert
    Assert.IsNotNull(result);
    Assert.AreEqual(substations.Count, result.Count);
}
```

The test given above validates the `ILocationRepository` implementation and that all substations in the database are returned. For this, the real database is prepared with data. Then, the repository method is called and the test checks if the quantity of substations stored in the database is equal to the quantity returned by this method.

UI test The third test type that is used in ADMO Inspect is the user interface test (UI test). This test simulates a user interaction in the application and verifies the behaviour of the application. The infrastructure of a UI test is a little bit different than the infrastructures of unit and integration tests. A virtual machine was set up for the UI tests to enable easy maintenance and because it is much easier to reset a virtual machine than a physical computer. Every time the UI tests are executed, an Android emulator is started. This Android emulator is used to run the app and to simulate a few user interactions. The following line shows how to start the Android emulator by using C# code:

```
string command = @"c:\android-sdk\tools\emulator.exe
                      -avd nexus9 -no-boot-anim -noaudio";
ProcessStartInfo startInfo =
    new ProcessStartInfo("cmd", "/c " + command);
startInfo.RedirectStandardOutput = true;
startInfo.UseShellExecute = false;
startInfo.CreateNoWindow = hideWindow;
Process process = new Process { StartInfo = startInfo };
process.Start();
```

After the emulator is finished, the boot process is complete and the `TestMethod` is executed. The following code extract gives an example of a UI Test that simulates a tap on the `Inspection` button:

```
[TestMethod]
public void InspectionButton_ShouldStartInspectionScreen()
{
    // Arrange
    Func<AppQuery, AppQuery> InspectionButton =
        c => c.Marked("InspectionButton");

    // Act
    app.Tap(InspectionButton);

    // Assert
    AppResult title = app.Query(element =>
        element.Property("Text", "Inspection"));
    Assert.IsNotNull(title);

}
```

The button can be found by the test because the property `StyleId` is assigned to it in the Xaml file:

```
<Image
    HorizontalOptions="Start"
    StyleId="InspectionButton"
    HeightRequest="250"
    WidthRequest="250"
    Source="inspectionButton.png">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Command="{Binding StartInspectionWorkflowCommand}" />
    </Image.GestureRecognizers>
</Image>
```

11 Usability

The last part of this thesis is about the empirical evaluation of the app. The developer of an application can only assess the behaviour of the application from a technical point of view. Therefore, it is very important to test how representative users experience the application. To evaluate the user experiences it is very common to perform a usability test. This kind of test helps to discover undetected problems and to gain a better understanding of the user expectations.

11.1 Definitional Basics

First of all, some definitional basics are given. The following terms are defined in the EN ISO 9271-210[1] standard.

Usability ”extend to which a system product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”

User ”person who interacts with the product”

Goal ”intended outcome”

[1] Standardization, 2015.

Effectiveness ”accuracy and completeness with which users achieve specified goals”

Efficiency ”resources expended in relation to the accuracy and completeness with which users achieve goals”

Satisfaction ”freedom from discomfort and positive attitudes towards the use of the product”

Context of use ”users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used”

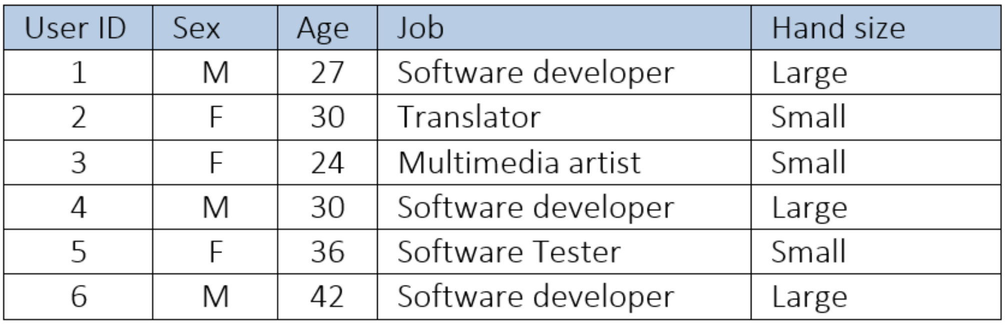
11.2 Usability Test Setup

This section gives an overview of the test setup that was used to perform the usability test. In order to talk about the test process itself, a detailed description of the test components involved is given. The selection of these components and slight modifications were done in cooperation with Dr. Jörn Walsdorf, usability engineer at OMICRON. This selection and the modifications were necessary to obtain comparable cross-project results. The full test can be found in the appendix.

Test candidates The following table gives an overview on the participants that were identified for the test. Jakob Nielsen, a usability evangelist at Nielsen Norma Group, showed that it is sufficient to test 5 candidates. When testing more than 5 people, no other results than the already detected problems will be obtained [2]. The usability test of the app was performed with an additional candidate to have a backup person if technical problems occur. The following table lists a few details about the test candidates. The

[2] cf: Nielsen, 2000

selection of these candidates was based on their previous knowledge of ADMO. Five of them were familiar with ADMO, only one of them was not.



User ID	Sex	Age	Job	Hand size
1	M	27	Software developer	Large
2	F	30	Translator	Small
3	F	24	Multimedia artist	Small
4	M	30	Software developer	Large
5	F	36	Software Tester	Small
6	M	42	Software developer	Large

Table 11.1: Test candidates

Test scenarios The usability test serves to find problems related to four different real world scenarios. The core functionality of the app is to create an inspection event for an asset. When creating an inspection event, it is possible to attach a picture, a file from the file system, and a checklist based on a predefined template. It is also possible to delete one of the attached files. Based on these core functionalities, the test candidates had to perform tasks in regard to four different test scenarios. As an example, the following section describes one of the test scenarios, which is about taking a picture and adding it to a new inspection event. The other scenarios can be found in the appendix.

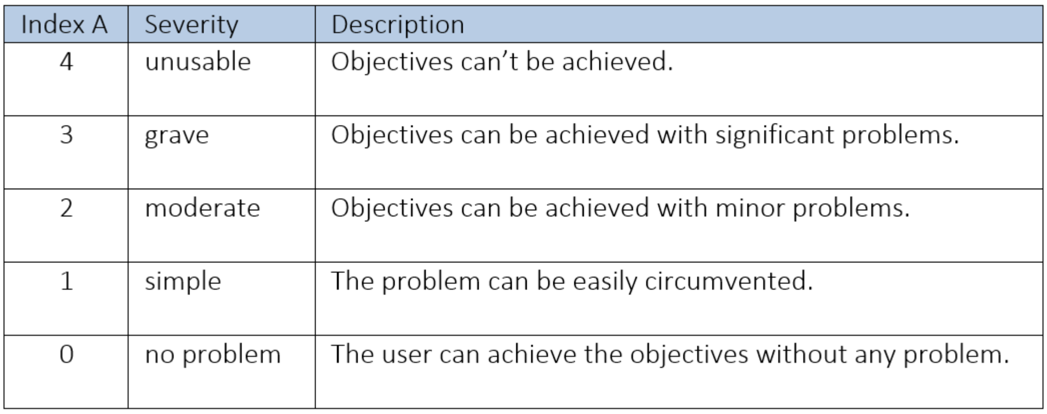
Introduction:
Please imagine the following situation: You are a tester of electrical components working for an electricity provider. You are currently performing an inspection of a voltage transformer which is located in the substation called Paris. You are using the new ADMO Inspect app to do the inspection.

Task introduction:
You know that the voltage transformer is located in the substation called

Paris and its voltage level is 110 kV. The corresponding feeder is called "D" and the serial number of the asset is h8h85g48. During your inspection you notice that something at the voltage transformer is wrong. There are some rusty parts which may affect the proper working of the asset. Of course, you should document those noticeable problems.

Task:
Find the corresponding voltage transformer and start a new inspection. To document the rusty parts during your inspection you take a picture of them. Name the picture "Problem1", finish the inspection and return to the start screen.

Test observations While the test candidates are completing the different test tasks, it is important to document any observations. An efficient way to do this is the usage of a prepared documentation sheet, which lists all necessary steps to complete a task. Such a subtask is, for instance, pressing a specific button. These subtasks have different levels of severity that can be ticked off on the list. The table below gives an overview of the used severity. It is based on Rubin J., 2008, p. 261, but was slightly modified as agreed with Dr. Jörn Walsdorf.

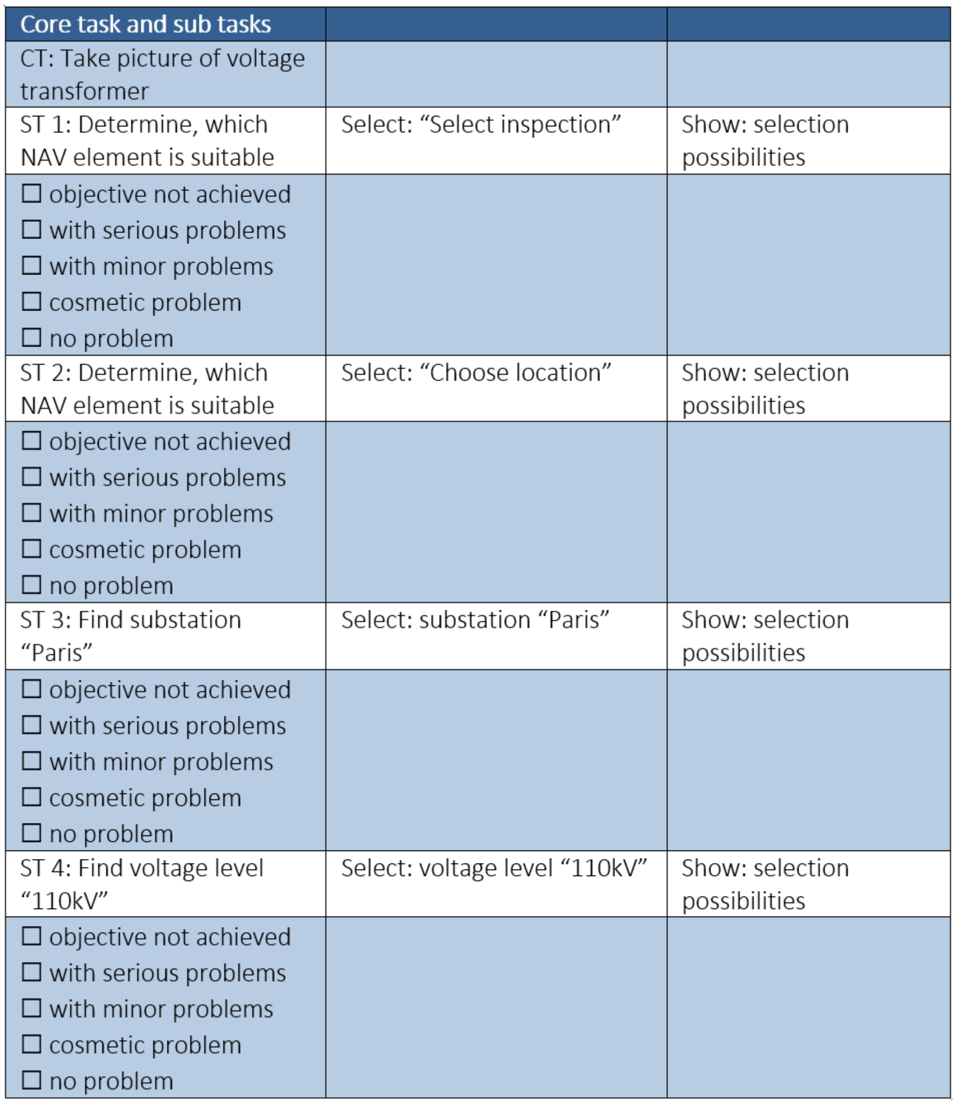


Index A	Severity	Description
4	unusable	Objectives can't be achieved.
3	grave	Objectives can be achieved with significant problems.
2	moderate	Objectives can be achieved with minor problems.
1	simple	The problem can be easily circumvented.
0	no problem	The user can achieve the objectives without any problem.

Table 11.2: Severities. Source: In dependence on Rubin J., 2008, p. 261

The documentation sheet for this test is based on the table above, but the wording was slightly modified as agreed with Dr. Jörn Walsdorf.

The following table illustrates such a documentation sheet listing the core task and a couple of subtasks:

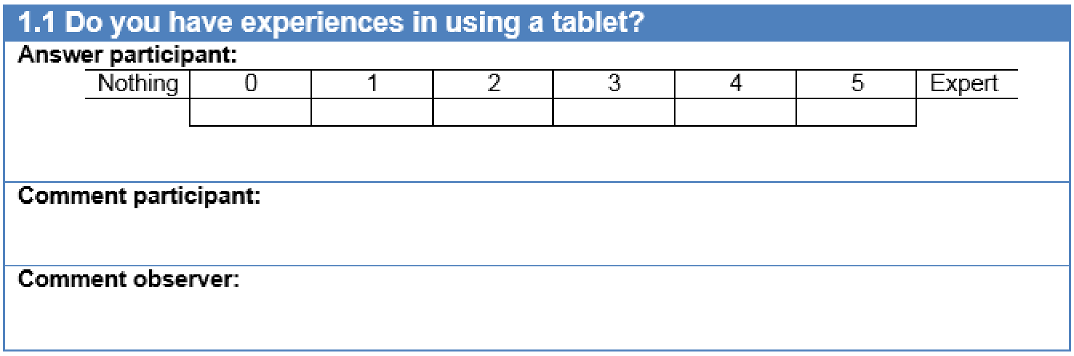


Core task and sub tasks		
CT: Take picture of voltage transformer		
ST 1: Determine, which NAV element is suitable	Select: "Select inspection"	Show: selection possibilities
☐ objective not achieved ☐ with serious problems ☐ with minor problems ☐ cosmetic problem ☐ no problem		
ST 2: Determine, which NAV element is suitable	Select: "Choose location"	Show: selection possibilities
☐ objective not achieved ☐ with serious problems ☐ with minor problems ☐ cosmetic problem ☐ no problem		
ST 3: Find substation "Paris"	Select: substation "Paris"	Show: selection possibilities
☐ objective not achieved ☐ with serious problems ☐ with minor problems ☐ cosmetic problem ☐ no problem		
ST 4: Find voltage level "110kV"	Select: voltage level "110kV"	Show: selection possibilities
☐ objective not achieved ☐ with serious problems ☐ with minor problems ☐ cosmetic problem ☐ no problem		

Table 11.3: Test sheet. Source: Modified as agreed with Dr. Jörn Walsdorf

Questionnaire The questionnaire basically depends on a mix of the IsoMetrics questionnaire [3] and some suggestions mentioned in the book "The user is always right"[4]. One thing that was modified in regard to the IsoMetrics questionnaire is the scale. IsoMetrics uses a scale from 1 to 5. Since OMICRON wants to get a tendency of the test candidate an odd-numbered scale from 0 to 5 is used.

The main purpose of the questionnaire is to get additional feedback regarding usability from the test candidate. At the beginning of the usability test, the candidate is asked questions like:



1.1 Do you have experiences in using a tablet?

Answer participant:

Nothing	0	1	2	3	4	5	Expert

Comment participant:

Comment observer:

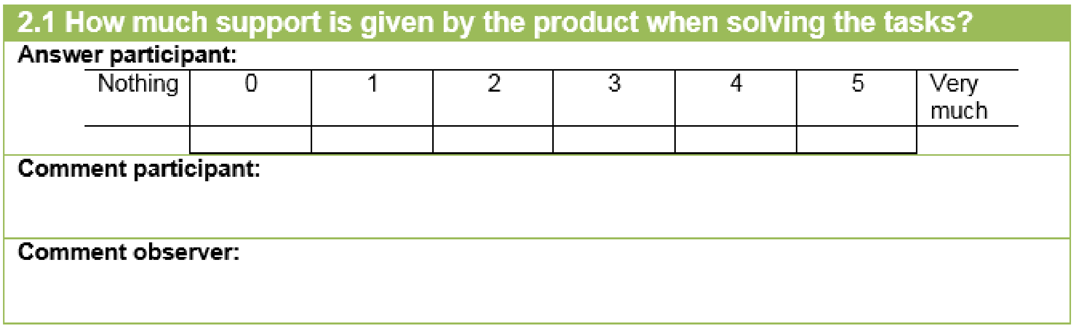
Figure 11.1: Questionnaire before the test. Source: In dependence on IsoMetrics, 2015 and Steve Mulder, 2007

This kind of questions are used to get a better understanding of the experiences the test candidate has already gained.

[3] cf: IsoMetrics, 2015

[4] Steve Mulder, 2007, p. 64.

There are also some additional questions that are asked after the test, such as:



2.1 How much support is given by the product when solving the tasks?

Answer participant:

Nothing	0	1	2	3	4	5	Very much

Comment participant:

Comment observer:

Figure 11.2: Questionnaire after the test. Source: In dependence on IsoMetrics, 2015 and Steve Mulder, 2007

The full questionnaire can be found in the appendix.

11.3 Test Process

This section describes the test process in more detail. It gives an overview on the field of application of the components mentioned above.

First, the test candidate is introduced to the test process and some facts about the test itself. This includes a very short explanation of ADMO and of the scope of application of the ADMO Inspect app. The candidate is also told that it is a thinking aload test.

When doing a thinking aload test it is very important that the candidate always says what he/she is currently thinking. Its not only about commenting out each click, but also about telling possible expectation thoughts. [5]

Another important point is that only the product is tested, but not the skills of the test candidate. This should reduce the excitement of the candidate and soften the not natural test situation. The last step of the introduction part is to ask the specific questions of the questionnaire.
Afterwards, the test is started by talking about the first test scenario. While the candidate is performing the tasks in regard to the given test scenario, all relevant observations are being documented with the help of the documentation sheet. After all test scenarios are completed, the final questions of the questionnaire are asked.

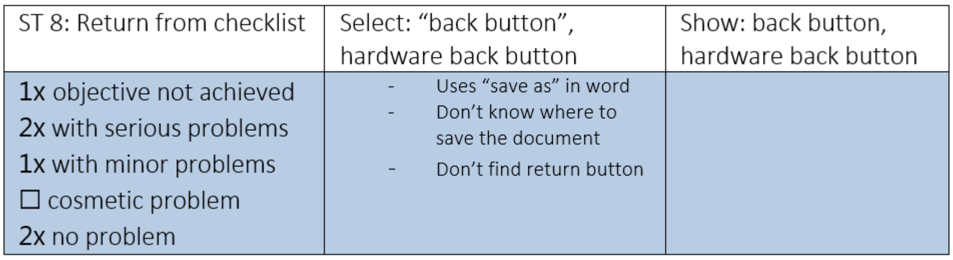
This process is used for each test candidate. After the last test candidate has completed the test, it is possible to start with the evaluation.

[5] cf: Nielsen, 2015

11.4 Evaluation Result

The last step of the usability test is to merge and evaluate the results. The main purpose of this evaluation is to assess the level of criticality of all observed problems. The problem with the highest criticality should be solved as fast as possible.

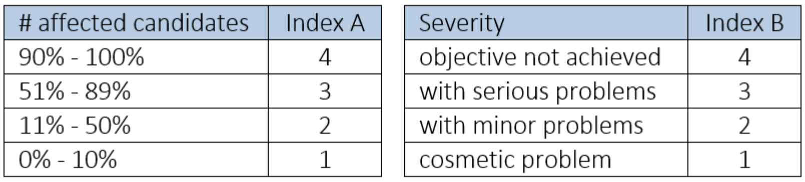
First, the subtasks are summarized and the ticked severities are counted like in the example below:



ST 8: Return from checklist	Select: "back button", hardware back button	Show: back button, hardware back button
1x objective not achieved 2x with serious problems 1x with minor problems ☐ cosmetic problem 2x no problem	- Uses "save as" in word - Don't know where to save the document - Don't find return button	

Table 11.4: Subtasks summary and severities

The example above shows the counted severities and in addition, all comments of all test candidates. The next step is to calculate the level of criticality. This can be done by using the following tables, which was modified by Dr. Jörn Waldorf in dependence on Rubin [6].



# affected candidates	Index A
90% - 100%	4
51% - 89%	3
11% - 50%	2
0% - 10%	1

Severity	Index B
objective not achieved	4
with serious problems	3
with minor problems	2
cosmetic problem	1

Table 11.5: Frequency of occurrence and overview of severities. Source: Rubin J., 2008, p. 262

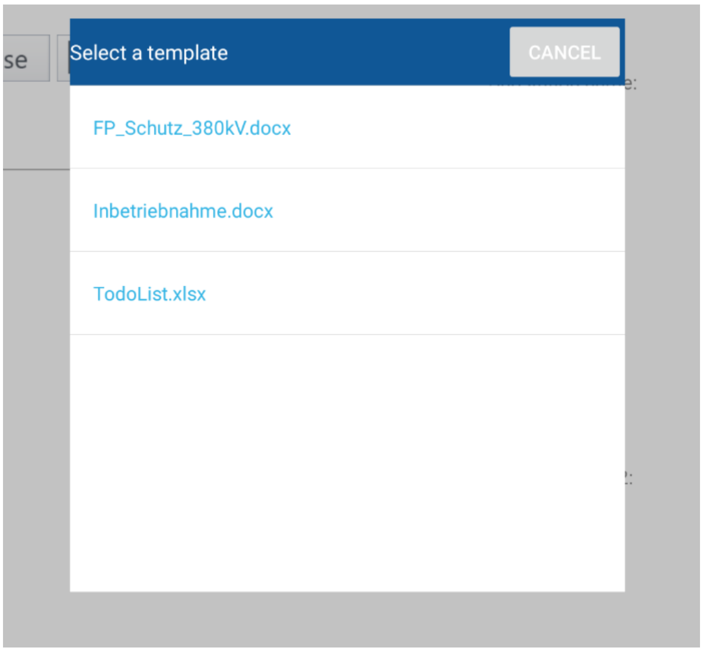
[6] cf. Rubin J., 2008, p. 262

The level of criticality is calculated by adding up the value of Index A and Index B. An example is shown in the subtask summary above (Figure 11.4). There are two users ”with serious problems”. Two users out of six are approximately 33%. Therefore, 33% of the candidates are affected by this problem. When looking up 33% in the first table, Index A is 2.
Then, the level of severity is looked up in the second table. The severity level ”with serious problems” is rated as 3. Hence, Index B is 3.
The level of criticality is calculated by adding up Index A and Index B. Therefore, the level of criticality of this subtask is 5.

Important to mention about this method is that the severity level ”no problem” does not affect the level of criticality and therefore, is not part of the calculation. Each subtask can have several levels of severity. It is necessary to calculate the level of criticality for each of these severities levels. The highest level of criticality of all severity levels is the final result for the specific subtask. The full evaluation can also be found in the appendix.

11.5 Final Result

This section lists the final results including the observation comments. The problem that occured most was how to return to the ADMO Inspect app after creating a new checklist. The first step to create a checklist is to select a template like in the figure below:



Select a template

CANCEL

FP_Schutz_380kV.docx

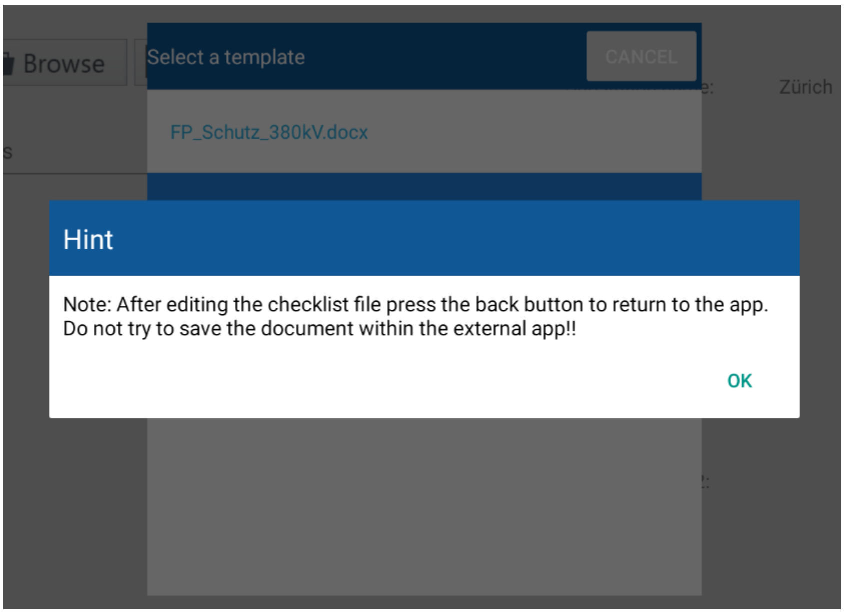
Inbetriebnahme.docx

TodoList.xlsx

Figure 11.3: Template dialog. Source: Screenshot ADMO Inspect

As soon as the user selects a template, a third party app is started in order to edit the checklist. After editing the template, the user needs to go back to the app where it is possible to save the changes and create a new checklist. After testing three people, it became clear that this behaviour is not acceptable and needs to be changed. Therefore, an additional hint dialog was added that provides a brief introduction on how to return to the app for the test candidates. As soon as this dialog was introduced, two of the three remaining test candidates did not have problems any more.

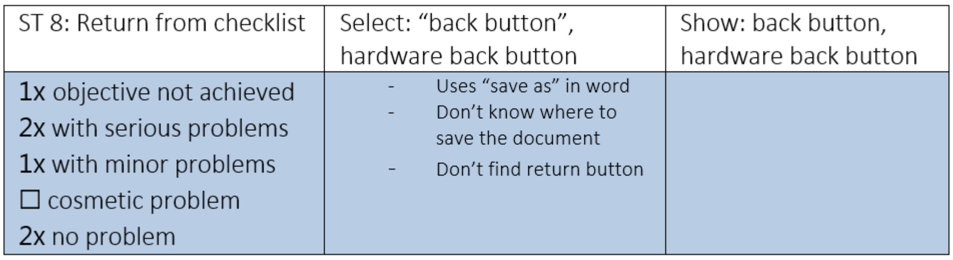
The following figure shows the additional dialog that appears after selecting a template:



Note: After editing the checklist file press the back button to return to the app. Do not try to save the document within the external app!!

Figure 11.4: Hint dialog. Source: Screenshot ADMO Inspect

The level of criticality of this problem is 6, and the result is shown in the figure below:



ST 8: Return from checklist	Select: “back button”, hardware back button	Show: back button, hardware back button
1x objective not achieved 2x with serious problems 1x with minor problems ☐ cosmetic problem 2x no problem	- Uses “save as” in word - Don’t know where to save the document - Don’t find return button	

Table 11.6: Problem evaluation

The second problem that could be observed was to display all assets of a substation. One reason for this problem was that the test candidates could not find the Select button next to the respective substation. The following figure illustrates the location tree:

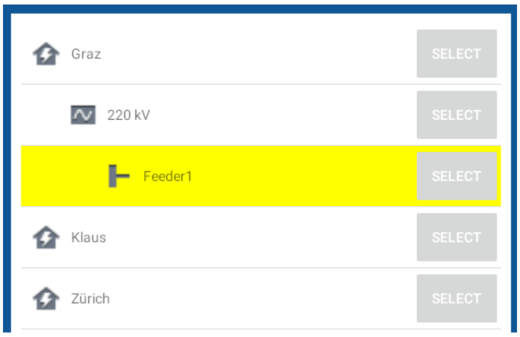
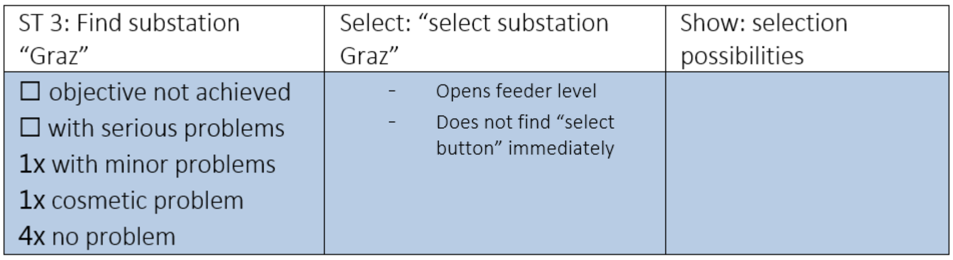


Figure 11.5: Location tree. Source: Screenshot ADMO Inspect

The calculated level of criticality is 4. The following figure illustrates the observation results of this subtask:



ST 3: Find substation "Graz"	Select: "select substation Graz"	Show: selection possibilities
☐ objective not achieved ☐ with serious problems 1x with minor problems 1x cosmetic problem 4x no problem	- Opens feeder level - Does not find "select button" immediately	

Table 11.7: Problem evaluation

The third problem is the least critical one and is basically a cosmetic problem concerning the location button that is illustrated in the figure below:

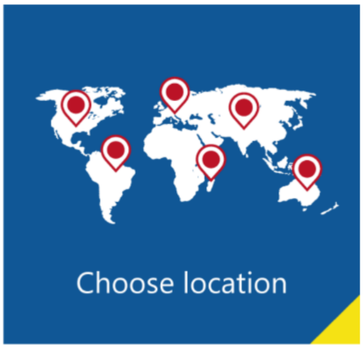
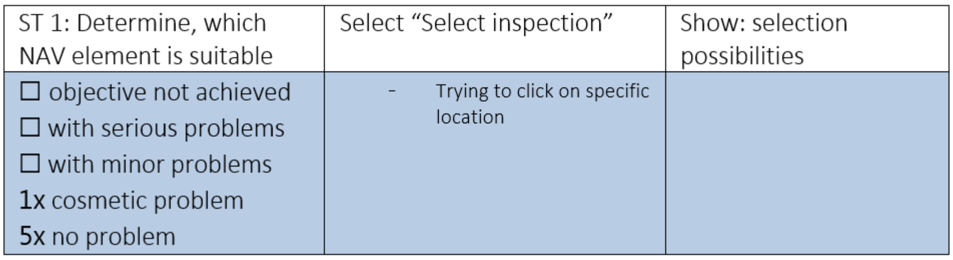


Figure 11.6: Location button. Source: Screenshot ADMO Inspect

The location button shows a map of the world that includes different location markers. One out of six test candidates thought that it is necessary to activate the button by pressing the specific marker. The calculated level of criticality is 3. The results are shown in the following figure:



ST 1: Determine, which NAV element is suitable	Select “Select inspection”	Show: selection possibilities
☐ objective not achieved ☐ with serious problems ☐ with minor problems 1x cosmetic problem 5x no problem	- Trying to click on specific location	

Table 11.8: Problem evaluation

12 Summary and Outlook

This chapter gives a summary of each individual chapter of this thesis, including their results. Finally, an outlook on further development plans for ADMO Inspect is given.

12.1 Summary

Electricity providers have to ensure the proper functioning of the electrical system in order to prevent blackouts. Hence, it is very important to regularly perform testing and maintenance activities on all electrical components. These activities are supported by modern software solutions, which provides a reliable process to minimize possible failures. Nevertheless, there are still problems that occur under special circumstances when the current software solutions do not fully satisfy the needs of these special test scenarios. One of the most relevant problems is that many testers are using an error-prone pen and paper approach to document their test results. This kind of documentation is mainly done by using individual checklists for the respective activity. When the testers have finished a test and completed the checklists, they need to go back to their offices and enter all results into the existing software system. A more detailed overview about the actual situation and a description of the associated problems is given in the Introduction chapter.

One of the already mentioned software solutions that supports the tester in the electrical field is developed by OMICRON and is called ADMO. However, there are still some situations where this solution does not fully satisfy the requirements to replace the error-prone pen and paper approach. To provide an innovative solution that solves the problems associated to

these special situations, it is necessary to analyse the already existing infrastructure of ADMO.
Chapter three focuses on this analysis and gives an overview on ADMO. It is shown that the infrastructure comprises the ADMO client and a database server. At the end of this chapter, the reader will have a better overview of the existing application, its environment and the involved components.

Then, it is necessary to analyse the requirements that are needed to develop an innovative solution called ADMO Inspect. There are two different kinds of requirements. On the one hand there are the functional requirements that are mainly relevant from the tester's point of view and on the other hand there are the non-functional requirements that are mainly relevant from the technical point of view, especially in regard to the existing infrastructure. Therefore, chapter four is divided according to these two kinds of requirements. The most important functional requirements for ADMO Inspect can be described by three roughly defined scenarios: First of all, it is necessary to provide an easy-to-use solution that supports working with checklists. Additionally, it should be possible to work offline and to gain access to the already existing asset data of ADMO. The non-functional requirements are about combining the already existing infrastructure with ADMO Inspect. One very important aspect in regard to this combination is that the new solution should not be dependent on the release cycles of ADMO. This means, when a new version of ADMO is released, the customers do not have to update ADMO Inspect too. Thus, to achieve such an independence, it is important to abstract the database.

Based on these requirements, the fifth chapter defines the technical components of ADMO Inspect. One component of the new architecture is a mobile app for tablets. The decision to create a mobile app is based on a detailed analysis of the requirements. Furthermore, it is necessary to combine the mobile app with the already existing system, and to decouple it from the database and the ADMO release cycles. Consequentially, the component introduced next is an additional service layer. In summary, the architecture of ADMO Inspect comprises the three tiers - the existing database, an additional service layer, and a mobile app.

Since there are many possibilities to implement these components, the next step is to evaluate possible technologies. The technology used for realizing the service layer is called `Windows Communication Foundation (WCF)`. The decision to use WCF is based on the requirements given by OMICRON, such as the compatibility with Microsoft Windows and the possibility to reuse existing C# code.
The second part of this chapter focuses on a detailed evaluation of the most common technologies to realize a mobile app. The considered technologies are native, hybrid and Xamarin development. Each section contains an introduction on the respective technology and a detailed evaluation based on predefined criteria. The purpose of this chapter is to gain better understanding of the possibilities that these technologies offer and to create the basis for deciding which one should be used.

The seventh chapter discusses the evaluation results of the reviewed technologies. The first section contains a comparison between the three technologies that is based on the satisfaction of the evaluation criteria. The second part of this chapter presents reasons for the decision which technology will be used. Basically, all technologies can be used to satisfy the main requirements. However, the decision was made to use Xamarin because it is cross-platform compatible and supports the .NET programming language that is far more familiar to OMICRON developers.

After analysing the requirements and taking the decision on which kind of technology should be used, the next step is to plan the screen design. Chapter eight gives a detailed description of the process that is used to create the screen design. This process follows three steps: creating a flowchart, creating a scribble, and finally creating the wireframe and applying the OMICRON style guide to it. At the end of the chapter the reader has an overview on how the screen design was created.

Then, it is time to start with the implementation. Chapter nine, which covers the service layer, describes the code modularization that was used for the implementation. The most important aspect of the modularization used is that it supports the service implementation for different hosting scenarios.

At the end of this chapter, the reader has a detailed overview on the structure of the service layer implementation.

The section about the implementation in chapter ten, explains how the mobile app is realized and describes the most relevant implementation aspects. The first part of the chapter is about code modularization, especially about sharing code between different platforms. Basically, the app can be divided into two code categories: platform-specific and platform-independent. Then, some important implementation strategies are discussed. This discussion focuses on two patterns called Inversion of Control and Model-View-ViewModel. The following section describes some implementation details concerning the user interface, for instance, the ViewModel-First principle. The last section of this chapter deals with data consistency and the synchronization process with the existing infrastructure.

After the part concerning implementation, chapter eleven explains a development method called ”Continuous Integration” (CI) and some details about the testing strategy of ADMO Inspect. The first part gives a brief description of CI and its components. For realizing the CI process it is necessary to use an automated build infrastructure, which is described in a subchapter. The last part of this chapter shows the testing strategy of the mobile app that is based on a common test automation pyramid. The pyramid defines three types of tests: at the bottom are unit tests, followed by integration tests, and at the top of the pyramid are UI tests. A detailed description of these test types is also given in this chapter.

The last chapter of this thesis describes the evaluation of the app. Such an evaluation can be carried out via usability tests and is necessary to find out how representative users experience the app. The first section describes the components involved in such tests: test candidates, test scenarios, test observations, and a questionnaire. Furthermore a detailed overview on the test process and the use of these components is given. In order to evaluate the test results it is necessary to calculate the level of criticality of the detected problems. This calculation is also described in this chapter. At the end of the chapter, the test results including the calculated level of criticality are presented.

12.2 Outlook

This is the last topic of this thesis and gives an overview of the vision for ADMO Inspect.

The first step will be to present ADMO Inspect to potential customers in order to receive more feedback. This step is necessary to create a feature list according to the customer's requirements.

Additionally, the test results of the usability test will be addressed. The most critical problem was the interaction with another app on the tablet when creating a checklist. A solution that is already being discussed involves the full integration of the checklists into the app and therefore, also into ADMO.

Within the scope of this thesis, only the Android operating system was covered. At the beginning, however, a few first steps towards the iOS operating system have already been made. Hence, a completely adapted iOS version of the app will also be realized.

Regarding the synchronization mechanism, it is already planned to support partial replication. This is a very important feature, because of the memory restrictions on tablets. A possible solution for this problem would be to synchronize preselected templates only.

This thesis provides a good basis for further expansions and improvements. The implemented workflow for checklists is a first example of how to support testers in the electrical field. In regard to mobile computing, there is an enormous potential for further innovative solutions that support the tester in the field and the electricity provider to ensure the proper functioning of the system.

Appendices

4

Test scenarios

Scenario 1

Introduction:

Please imagine the following situation: You are a tester of electrical components working for an electricity provider. Your boss asked you to perform an inspection of a circuit breaker, which is located in a substation called Graz. For performing this test activity you are using the new ADMO Inspection App.

Task introduction:

You know that the circuit breaker is located in the substation called Graz and its serial number is 12345. To ensure a proper working of this circuit breaker you were asked to perform an inspection whereat you have to fill out a checklist based on a template called "Inbetriebnahme.docx".

Task:

Find the corresponding circuit breaker and start a new inspection. Attach a new inspection checklist, which is based on the template called "Inbetriebnahme.docx". Name the new checklist file "Checklist1". Afterwards save the inspection and return to the start screen.

Scenario 2

Introduction:

Please imagine the following situation: You are a tester of electrical components working for an electricity provider. You are currently performing an inspection of a voltage transformer which is located in the substation called Paris. You are using the new ADMO Inspection App to do the inspection.

Task introduction:

You know that the voltage transformer is located in the substation called Paris and its voltage level is 110 kV. The corresponding feeder is called "D" and the serial number of the asset is h8h85g48. During your inspection you notice that something at the voltage transformer is wrong. There are some rusty parts which may affect the proper working of the asset. Of course, you should document those noticeable problems.

Task:

Find the corresponding voltage transformer and start a new inspection. To document the rusty parts during your inspection you take a picture of them. Name the picture "Problem1", finish the inspection and return to the start screen.

Scenario 3

Introduction:

Please imagine the following situation: You are a tester of electrical components working for an electricity provider. It is a sunny day and you are on the way to a circuit breaker located in the substation called Los Angeles. Yesterday, you have already created an inspection for this circuit breaker and now you want to append an additional file to this inspection.

Task introduction:

You know that the circuit breaker is located in the substation called Los Angeles and its voltage level is 20V. You also know the corresponding feeder which is called “Bc”. The serial number of this circuit breaker is h8h85g56. You already created a word file which contains some notes about the inspection. Now, you have to append this word file to the already created inspection.

Task:

Find the corresponding circuit breaker and have a look at the already created inspection. Edit the existing inspection and attach the word file to it. The file itself can be found at the following location: “sdcard/Download” and its name is “AdditionalNotes.docx”. Save the inspection and return to the start screen.

Scenario 4

Introduction:

Please imagine the following situation: You are a tester of electrical components working for an electricity provider. Yesterday, you performed an inspection of a station DC supply component located in the substation called Cleveland. Today you have reviewed this inspection and noticed that you have done a mistake.

Task introduction:

You know the station DC supply is located in the substation called Cleveland and its voltage level is 110kV. Its serial number is h8h85g44. Your mistake was that you attached a picture of your lunch to the inspection. Of course you do not want that this picture is attached to the inspection. Now, you have to delete this picture.

Task:

Find the corresponding station DC supply and also find the related finished inspections. Edit the already created inspection and find the picture of your lunch. Delete this picture and save the inspection. Return to the start screen.

User_ID: 4 Date: 03.11.15
13 30

Evaluation questions before the test

1. Domain Experiences

1.1 Do you have experiences in using a tablet?

Answer participant:

Nothing	0	1	2	3	4	5	Expert
			~~X~~		X		

Comment participant: bericht! kein Tablet

Comment observer:

1.2 Which operating system of tablets are you using? (iOS, Android, Windows,..)

Answer participant: Android
– kein iOS, kein Windows

Comment participant:

Comment observer:

1.3 Do you have experiences in using ADMO?

Answer participant:

Nothing	0	1	2	3	4	5	Expert
					X		~~X~~

Comment participant:

Comment observer:

1.4 Do you have experiences in using Microsoft Word?

Answer participant:

Nothing	0	1	2	3	4	5	Expert
					X		

Comment participant:

Comment observer:

User_ID:________ Date:_________

Evaluation questions after the test!!!

2. Goals and behaviors

2.1 How much support is given by the product when solving the tasks?

Answer participant:

Nothing	0	1	2	3	4	5	Very much
					X		

Comment participant: 2 x edit drücken

Comment observer:

2.2 Which feature characteristic did you like best and why?

Answer participant: Hat das getan was ich erwartet habe

Comment participant:

Comment observer:

U 03.11.15
13.50

3. Attitudes and motivators

3.1 How would you describe the product to an OMICRON employee?

Answer participant: Zugriff auf die wichtigsten Daten im Feld. Alle Kernfunktionen, die man im Feld braucht

Comment participant:

Comment observer:

4. User Experience

4.1 How much do you agree, that the tool is easy to use?

Answer participant:

I'm totally disagreeing	0	1	2	3	4	5	I'm totally agreeing
				X			

Comment participant: 1. 2 x edit
2. Template dialog

Comment observer:

4.2 How much do you agree, that you understood every function of the product?

Answer participant:

I'm totally disagreeing	0	1	2	3	4	5	I'm totally agreeing
						X	

Comment participant:

Comment observer:

4.3 Do you think it is possible to use this product even if ADMO is not known?
Answer participant: braucht man Vorwissen Stelle "Create direkt unter Subsolution"
Comment participant:
Comment observer:

5. Opportunities

6.1 Where do you think is room for improvement?
Answer participant: edit - edit Template dialog "Create "RA" from template"
Comment participant:
Comment observer:

7. Demographic questions

7.1 Age?
Answer participant: 26

7.2 Sex?		
Answer participant:	☒ F	□ M

7.4 Education?
Answer participant: Eng.
Comment participant:
Comment observer:

1

User_ID: 4 Datum: 02.11.15
13 30

Core task and sub tasks		
CT: Create new inspection checklist for CB-12345 at Graz		
ST 1: Determine, which NAV element is suitable	Select: „Select inspection“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 2: : Determine, which NAV element is suitable	Select: „Choose location“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 3: Find substation “Graz”	Select: „select substation Graz“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☒with minor problems ☐cosmetically problem ☒ no problem	→ öffnet auch	Fenster Eisen
ST 4: Find asset “CB-12345”	Select: „select new inspection“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 5: Find button „Template-Dialog“	Select: „Template-Dialog“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 6: Find template „Inbetriebnahme.docx“	Select: template „ Inbetriebnahme.docx“	Show: selection possibilities
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem	Zweite Suche nicht angezeigt [illegible] – Beim zweiten ist es klar.	
ST 7: Confirm hint dialog	Select: OK button	Show: hint dialog with OK button
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		

ST 8: Return from checklist	Select: „back button“, hardware back button	Show: Back button, hardware back button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 9: Name checklist	Enter name: „Checklist1“	Show: Text field
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑no problem		
ST 10: Save inspection	Select: „save inspection“ button	Show: save button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem		
ST 11: Return to start screen	Select: „back“ button or hardware „back“ button	Show: “back” button and “hardware back” button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑no problem	Hardware Button	

1. Erst anlegen, dann öffnen

User_ID: 4 Datum: 03.11.15
13^{30}

2

Core task and sub tasks		
CT: Take picture of voltage transformer		
ST 1: Determine, which NAV element is suitable	Select: „Select inspection“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem	Wozu brauche ich die Inspektion über die letzte Inspektion	
ST 2: : Determine, which NAV element is suitable	Select: „Choose location“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 3: Find substation „Paris“	Select: substation „Paris“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem		
ST 4: Find voltage level „110kV“	Select: voltage level „110kV“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem		
ST 5: Find feeder „D“	Select: feeder „D“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 6: Find asset VT-h8h85g48	Select: new inspection	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 5: Find button „Camera“	Select: „Camera“ button	Show: selection possibilities

☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☐ no problem	↩ als "back" gemacht	
ST 6: Return from taking picture	Select: „ok button“, hardware back button	Show: OK button, hardware back button
☐objective not achieved ☐with serious problems ☒with minor problems ☐cosmetically problem ☒ no problem		
ST 7: Name picture	Enter name: „Problem1“	Show: text field
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 8: Save inspection	Select: „save inspection“ button	Show: save button
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		
ST 10: Return start screen	Select: „back“ button or hardware „back“ button	Show: “back” button and “hardware back button”
☐objective not achieved ☐with serious problems ☐with minor problems ☐cosmetically problem ☒ no problem		

User_ID: 4 Datum: 05.11.15
13 30

3

Core task and sub tasks		
CT: Attach file to inspection		
ST 1: Determine, which NAV element is suitable	Select: „Select inspection“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 2: : Determine, which NAV element is suitable	Select: „Choose location“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 3: Find substation „Los Angeles“	Select: substation „Los Angeles“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 4: Find voltage level „20V“	Select: voltage level „20V“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 5: Find feeder „Bc“	Select: select feeder „Bc“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 6: Find asset “CB-h8h85g56”	Select: **edit** inspection	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem	sortieren nach Namen? mindestens Alphabet!!	

User_ID:________ Datum:_________

ST 7: Find existing inspection	Select: „Edit“ button	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem	wenn nur eine Inspection dann direkt "show" anstatt "edit"	
ST 8: Find “File-Browser” button	Select: „File-Browser“ button	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem	"File Browser" umbenennen!	
ST 9: Find word file at „sdcard/Documents“	Select: file	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem		
ST 10: Save inspection	Select: „save inspection“ button	Show: save button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem	Feedback speichern Feedback	
ST 11: Return start screen	Select: „back“ button or hardware „back“ button	Show: “back” button and “hardware back button”
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒no problem		

User_ID: 4 Datum: 05.11.15
13 50

4

Core task and sub tasks		
CT: Delete picture of existing inspection		
ST 1: Determine, which NAV element is suitable	Select: „Select inspection“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 2: : Determine, which NAV element is suitable	Select: „Choose location“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 3: Find substation „Cleveland“	Select: substation „Cleveland“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 4: Find voltage level „110kV“	Select: voltage level „110kV“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem	Autoscroll damit alle Fenster angezeigt werden	
ST 6: Find asset “station DC supply - h8h85g44”	Select: edit inspection	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☒ no problem		
ST 7: Find existing inspection	Select: „Edit“ button	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems		

□cosmetically problem ☑ no problem		
ST 8: Find lunch picture	Select: „problem2.png“	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem □ no problem		
ST 9:Return to inspection screen	Select: back button, hardware back button	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem		
ST 10:Delete picture “Problem2.png”	Select: delete button	Show: selection possibilities
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem	Thumbnail	
ST 11: Confirm deletion	Select: OK button	Show: dialog with OK button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem		
ST 12: Save inspection	Select: „save inspection“ button	Show: save button
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem		
ST 13: Return start screen	Select: „back“ button or hardware „back“ button	Show: “back” button and “hardware back button”
□objective not achieved □with serious problems □with minor problems □cosmetically problem ☑ no problem		

Linie highlighted von Bild

Bibliography

András (2015). *AutoView – the Missing Link for a Good ViewModel First Approach*. URL: http://dotneteers.net/blogs/vbandi/archive/2014/03/25/autoview-the-missing-link-for-a-good-viewmodel-first-approach.aspx (cit. on p. 77).

Autofac (2015). *Autofac*. URL: http://autofac.org/ (cit. on p. 74).

Bristowe, John (2015). *What is a Hybrid Mobile App?* URL: http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/ (cit. on p. 38).

Brody, Chris (2015). *Cordova/PhoneGap sqlite storage adapter*. URL: https://github.com/litehelpers/Cordova-sqlite-storage (cit. on p. 41).

Cohn, Mike (2015). *The Forgotten Layer of the Test Automation Pyramid*. URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid (cit. on p. 85).

Cordova, Apache (2015). *Apache Cordova*. URL: http://cordova.apache.org/ (cit. on pp. 38, 39).

Fowler, Martin (2015). *Continuous Integration*. URL: http://www.martinfowler.com/articles/continuousIntegration.html (cit. on p. 83).

Google (2015a). *Android Anatomy and Physiology*. URL: http://androidteam.googlecode.com/files/Anatomy-Physiology-of-an-Android.pdf (cit. on p. 31).

Google (2015b). *Android, the world's most popular mobile platform*. URL: http://developer.android.com/about/android.html (cit. on p. 31).

Google (2015c). *Application Fundamentals*. URL: https://developer.android.com/guide/components/fundamentals.html (cit. on p. 32).

Google (2015d). *Application Fundamentals*. URL: https://developer.android.com/guide/components/fundamentals.html (cit. on p. 33).

Google (2015e). *ART and Dalvik*. URL: https://source.android.com/devices/tech/dalvik/ (cit. on p. 32).

Henning Wolf Rini van Solingen, Eelco Rustenburg (2012). *Die Kraft von Scrum*. First. Pearson (cit. on p. 10).

Hermes, Dan (2015). *Xamarin Mobile Application Development*. First. Apress (cit. on pp. 45, 46, 49, 50).

history.com (2015). *The Great Northeast Blackout*. URL: http://www.history.com/this-day-in-history/the-great-northeast-blackout (cit. on p. 1).

IsoMetrics (2015). *Über den Fragebogen 'IsoMetrics'*. URL: http://www.isometrics.uni-osnabrueck.de/qn.htm (cit. on pp. 95, 96).

ITWissen (2015). *IoC (inversion of control)*. URL: http://www.itwissen.info/definition/lexikon/IoC-inversion-of-control-Umkehrung-des-Kontrollflusses.html (cit. on p. 73).

Jacobsen, Jens (2014). *Website-Konzeption, Erfolgreiche Websites planen, umsetzen und betreiben*. Seventh. Dpunkt (cit. on pp. 60, 62, 64, 66).

Lisa Crispin, Janet Gregory (2009). *Agile Testing - A practical guide for testers and agile teams*. First. Addison-Wesley (cit. on p. 85).

Löwy, Juval (2010). *Programming WCF Services*. Third. O'Reilly (cit. on pp. 14, 17, 21, 24, 27, 28).

Microsoft (2015a). *Authentication and Authorization in WCF Services*. URL: https://msdn.microsoft.com/en-us/library/ff405740.aspx (cit. on p. 25).

Microsoft (2015b). *Chapter 7: Message and Transport Security*. URL: https://msdn.microsoft.com/en-us/library/ff648863.aspx (cit. on pp. 22–24).

Microsoft (2015c). *How to: Host a WCF Service in a Managed Windows Service*. URL: https://msdn.microsoft.com/en-us/library/ms733069.aspx (cit. on p. 27).

Microsoft (2015d). *How to: Host a WCF Service in IIS*. URL: https://msdn.microsoft.com/de-de/library/ms733766(v=vs.110).aspx (cit. on p. 21).

Microsoft (2015e). *How to: Host a WCF Service in WAS*. URL: https://msdn.microsoft.com/en-us/library/ms733109(v=vs.110).aspx (cit. on p. 27).

Microsoft (2015f). *Self Hosting Windows Communication Foundation Services*. URL: https://msdn.microsoft.com/en-us/library/ee939340.aspx (cit. on p. 28).

Microsoft (2015g). *The MVVM Pattern*. URL: https://msdn.microsoft.com/en-us/library/hh848246.aspx (cit. on p. 75).

Microsoft (2015h). *Understanding WS-Security*. URL: https://msdn.microsoft.com/en-us/library/ms977327.aspx (cit. on p. 20).

Microsoft (2015i). *Was ist XAML?* URL: https://msdn.microsoft.com/de-de/library/cc295302.aspx (cit. on p. 76).

Microsoft (2015j). *Windows Communication Foundation Architecture*. URL: https://msdn.microsoft.com/en-us/library/ms733128(v=vs.110).aspx (cit. on pp. 17, 18).

Mono (2015). *About Mono*. URL: http://www.mono-project.com/docs/about-mono/ (cit. on p. 44).

Nielsen, Jakob (2000). *Why You Only Need to Test with 5 Users*. URL: http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (cit. on p. 91).

Nielsen, Jakob (2015). *Thinking Aloud: The #1 Usability Tool*. URL: http://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/ (cit. on p. 97).

rense.com (2015). *The Great Northeast Blackout of 1965*. URL: http://www.rense.com/general40/95.htm (cit. on p. 1).

Roman Pichler, Stefan Roock (2011). *Agile Entwicklungspraktiken mit Scrum*. First. Dpunkt (cit. on p. 10).

Rubin J., Chisnell (2008). *Handbook of Usability Testing. How to Plan, Design, and Conduct Effective Tests*. First. Wiley (cit. on pp. 93, 98).

Scott, Ambler (2012). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. First. Wiley (cit. on p. 14).

Standardization, International Organization for (2015). *Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems*. URL: https://www.iso.org/obp/ui/#iso:std:52075:en (cit. on p. 90).

Statista (2015). *Prognose zu den Marktanteilen der Betriebssysteme am Absatz vom Smartphones weltweit in den Jahren 2015 und 2019*. URL: http://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/ (cit. on p. 31).

Steve Mulder, Ziv Yaar (2007). *The User Is Always Right, A practical guide to creating and using personas for the web*. First. New Riders (cit. on pp. 95, 96).

Wargo, John M. (2014). *Apache Cordova API Cookbook*. First. Addison-Wesley Professional (cit. on pp. 38, 40).

Xamarin (2015a). *An Introduction to Xamarin.Forms*. URL: https://developer.xamarin.com/guides/cross-platform/xamarin-forms/getting-started/introduction-to-xamarin-forms/ (cit. on pp. 50, 51).

Xamarin (2015b). *Sharing Code Options*. URL: https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/ (cit. on pp. 45, 47, 49).

Xamarin (2015c). *Xamarin*. URL: http://xamarin.com/ (cit. on p. 44).