Development of a remote control application for a set-top box on a Windows 8 tablet PC

Patrick Persson
LIU-IDA/LITH-EX-G--13/043—SE

2013-11-14
Examensarbete

Development of a remote control application for a set-top box on a Windows 8 tablet PC

Utveckling av fjärrkontrollsapplikation för set-top box till Windows 8-surfplatta

av

Patrick Persson
LIU-IDA/LITH-EX-G--13/043—SE
2013-11-14

Handledare: Kristofer Liew, AppCentric
Examinator: Mikael Asplund, Linköpings Universitet
Abstract

This report describes the development of a Windows Store application which runs on a Windows 8 tablet PC. The application was developed for a company that targets television based services and have the requirements of displaying a television guide based on data from a device connected to the television. Both the application and the device communicates with each other via an intermediary web based service.

Established design patterns for object-oriented programming languages were used along with a method called test-driven development in order to accomplish this. Furthermore a comparison between mocking frameworks will be presented for choosing a suitable framework when developing Windows Store applications. The communication model between application and the device is also discussed and evaluated for whether or not it is a suitable method of communication.
# Table of Contents

1. Introduction ................................................................................................................. 4  
   1.1. Background ............................................................................................................ 4  
   1.2. Problem .................................................................................................................. 4  
   1.3. Constraints ............................................................................................................ 5  
   1.4. Goals ..................................................................................................................... 5  
   1.5. Measureable requirements ...................................................................................... 6  
   1.6. Approach ............................................................................................................... 6  

2. Technologies .................................................................................................................. 8  
   2.1. Electronic Program Guide ..................................................................................... 8  
   2.2. Windows Store ...................................................................................................... 9  
   2.3. Test-driven development ...................................................................................... 11  
   2.4. SOLID .................................................................................................................. 13  

3. Using test-driven development with WinRT .................................................................. 15  
   3.1. Why use test-driven development ........................................................................ 15  
   3.2. Test framework ..................................................................................................... 15  
   3.3. Choosing a mocking framework ........................................................................... 15  

4. Viasat Remote Control ................................................................................................. 20  
   4.1. Overview ............................................................................................................... 20  
   4.2. Communication model .......................................................................................... 20  
   4.3. Model View ViewModel ......................................................................................... 21  
   4.4. Provider versus Service ....................................................................................... 22  
   4.5. Responsibilities for creating concrete dependencies .......................................... 24  
   4.6. Design choices ..................................................................................................... 25  
   4.7. Sharing data between pages ................................................................................ 27  
   4.8. Sample data ......................................................................................................... 28  
   4.9. Dialogs, pages and controls ................................................................................ 29  
   4.10. Service Delivery Platform ................................................................................... 31  
   4.11. Screenshots ........................................................................................................ 31
1. Introduction

This report details the thesis work in which an application was developed for a Windows 8 tablet PC in order to control a set-top box. The thesis work was done between January 2013 and August 2013 as part of a bachelors degree in computer science.

1.1. Background

Today, many households use a set-top box (STB) to receive television broadcasts and related information which is then displayed on their television sets. The service for downloading information about what programs are broadcasted during the day is called an Electronic Program Guide (EPG) and is offered by various companies that sell television related services. The STB can keep track of a specific customer and the channels which are available to them, and download information specific to that customer. The STB is traditionally controlled using an infrared remote control and the information downloaded by the STB is displayed on the connected television set, in effect limiting the user to be in the vicinity of the television set when browsing the information offered by the STB.

In today’s society hand-held devices such as mobile phones and tablet PC’s are used to control and query information about other devices such as television sets. Windows 8 is an operating system which is offered for both desktop computers, tablet PC’s and mobile phones as an operating system, and as such the user can choose the platform most suited for them to use. Viasat is a company that offers television related services in the Nordic and Baltic countries, broadcasting EPG-data and television broadcasts via satellite. NDS is a company partner of Viasat which develops the web service able to communicate with the STB sold by Viasat. The current web service offers services for downloading EPG-data as well as API’s for controlling the STB. AppCentric is the company behind the development of the application for Viasat during this thesis work.

At the moment Viasat is missing an application running on a Windows 8 tablet PC to control their STB. An application on a tablet PC could improve the usability of controlling the STB. Some use cases could be to query what movies or series are showing during the day. Another example is to control the STB to record a program that the user would like to watch at a later point.

1.2. Problem

The problem in this thesis work is to create an application that communicates with a web service in a reliable way, such that errors that might happen during communication are dealt with without the application crashing or behaving incorrect. Incorrect behaviour can also occur in other parts of the application, for instance when storing or accessing information from a storage location, such as a file system. This should also be addressed, since the application will be restoring and saving state when the application is suspended by the operating system.

Another problem is to decrease the number of test runs required to make sure the application functions correctly. A test run might involve starting the application on a device and clicking a
button which downloads EPG-data and see if the correct information is sent as a request, and that the response data is displayed on the screen.

In order to accomplish this a software development process called test-driven development will be used to automate many of the tasks usually performed when running an application, and verifying the behavior of classes with respect to other classes.

1.3. Constraints

The programming language used for developing the application will be C#. This is the main programming language used by AppCentric.

Even though a Windows Store app is in full-screen mode, different resolutions are available depending on size of the screen and specific features in the operating system. Because of this fact views should be designed for different resolutions in mind, but with full-screen resolution as the default. There’s a feature in Windows 8 to put two applications side-by-side, so called snapped mode, which the application needs to handle as well.

The STB used in this thesis work is one specific from Viasat, and in order to communicate with it the NDS API’s will be used. Currently the API’s that work against the STB is a virtual machine called softbox with generic data hosted by NDS.

The primary language of the user interface of the application will be in Swedish, not including downloaded content.

1.4. Goals

The primary goal of this thesis work is to create an application which will be able to communicate with a Viasat STB, as well as describing the parts of the development process that others developing Windows Store apps might find relevant.

Another goal is to use test-driven development when developing the application, and follow standards set by users of test-driven development as well as for Windows Store app development, in terms of design patterns, class relationships and hierarchies. A part of this goal is finding a suitable test and mocking framework available for Windows Store app development, if any exist.

The application should also meets the demands requested by many users today. That means a responsive and good looking user interface that highlights most common tasks and delivers information on demand.

The goals are listed in priority order, with highest priority first.

1. The application should be able to pair with a STB, meaning the application will be able to detect and send instructions to a specific STB.
2. The application should be able to display EPG-data for multiple channels at once, and also be able to display EPG-data for the currently viewed channel on the STB.
3. The application should be able to schedule a program to the STB for recording.
4. The application should be able to delete recordings, both scheduled and finished.

1.5. Measureable requirements

A part from the goals above, the finished application should be able to accomplish the following requirements. These can not be automated and are validated by running the application:

- A page in the application should be displayed within 2 seconds of requesting to see that page.
- The user of the application should be able to scroll when viewing EPG-data to see at least 24 hours of data from the current time of day.
- When a user changes page, the application should be able to load and display the other page in at least 2 seconds; aborting or suspending downloads from the currently viewed page in order to meet the demands.
- The user should be able to scroll a larger timespan when viewing EPG-data, for instance scroll 24 hours ahead to make viewing the next day of EPG-data without delay.
- The user interface should not freeze when the user performs a task.
- The view for displaying EPG-data for multiple channels should resemble the one used by Viasat by the STB on a television.

1.6. Approach

The approach is the following, divided into the following phases:

- Studying of API's for downloading of data,
- Conceptual design of the views in the application,
- Choosing a test framework
  - By doing an online search for available test frameworks
- Choosing a mocking framework
  - By doing an online search for available mocking frameworks
- Implementing tests for services, models and viewmodels
- Implementing the application
  - Using test-driven development with the tools from the previous bullets
  - Creating low-level classes
  - Creating views and services using the low-level classes
- Testing, validating and evaluating measurable requirements

The study phase means reading the specifications of the API from NDS, to determine the capabilities of controlling the STB and downloading EPG-data. This is necessary in order to determine what classes should be available in the application. A class will be implemented for each application.
service at a later stage that facades these services for use in the application. At the start of the project, the services need to address the following functionality:

- downloading EPG,
- controlling the STB, and
- downloading movie and series information.

The use cases will highlight what a potential user will do when using the application, and what functionality each page needs to cater. The use cases will create a concept of a workflow in the application, basic actions and navigation a user needs to do to perform a task. This will in effect direct what pages should exist along with their design. Each page can be developed independent of each other, so they will be created in order of their priority, as specified in the goals.

The measurements for displaying a page will be accomplished using timers in code. This can be accomplished using a test library which is notified when a page is about to be navigated to and later when it is shown, using events in C#. It will also be measured manually using a stopwatch. By measuring both, the data will be more accurate of the actual time spent, and the perceived time taken.
2. Technologies

This thesis project uses several technologies that the reader might not be familiar with. The ones that are important for understanding the later chapters are discussed here, with an explanation of how they are relevant to the development process.

An Electronic Program Guide is used to display what shows are available to watch given either channels or some on demand access to content a user might have. The concept requires some sort of content server that a client can download information from, and an interface to which clients can connect and download the data.

Windows Store is the name of applications running in full-screen on Windows 8 devices. Applications developed for Windows Store can be made available via the Microsoft Store application which is installed on all Windows 8 devices. In order to publish an application in the Windows Store it needs to go through the certification process offered by Microsoft.

The development process for a Windows Store application is free to choose by the developer. One such process is test-driven development, which is used in this thesis project. When developing using test-driven development, class hierarchies are set up to be dependent on each other instead of having one class perform many tasks. By separating classes and making them dependent on each other, the behaviour between the classes can be tested. A behaviour can be described as the way a class interacts with another class, which methods it calls for instance and the state it will be in given some actions. This behaviour can be verified with the use of another project with the use of tests. The structure of setting up the hierarchies in this project is well described by SOLID, which is a set of principles for creating classes that works well when developing with test-driven development.

2.1. Electronic Program Guide

With today’s television services offering many channels for a user to watch, the value of having an electronic program guide increases to filter or find the shows that the user is interested in. As described by European Telecommunications Standards Institute

*The value of an EPG to a user is to be informed of the most interesting programmes that fit his viewing criteria. Now the user can see if a programme of his choice is available within the next few days and on what channel. (Institute European Telecommunications Standards, 2003)*

The document also describes that other services can be integrated with the television broadcast, for instance to be able to record a show with a VCR given a program code. That was from 2003, today digital boxes connected to television sets are often equipped with harddrives (Viasat) to record shows and can be integrated with the electronic program guide in a more convenient way since the box handles all information. Television sets are also getting equipped with harddrives to record live TV, and can integrate its own electronic program guide in the television.
2.2. Windows Store

Microsoft is the company behind Windows 8 and Windows Store apps. Microsoft describes Windows Store apps as the following:

"A Windows Store app is a new type of app that is sold in the Windows Store and runs on Windows 8 devices. They install easily and uninstall cleanly. They run in a single window that fills the entire screen by default." (Microsoft, 2013)

Windows 8 was released on October 26 2012 (Newman, 2012), and even though it’s a relatively new operating system it has many tools for developing applications which are free for download. Windows Store apps is the name of applications running on Windows 8 devices (Microsoft, 2012) (Microsoft, 2013). The Windows Runtime, or WinRT for short, is a "native API built into the operating system" (Microsoft, 2013) for Windows 8 devices that enables developers to create applications using either JavaScript, C#, Visual Basic, or C++ (Microsoft, 2013).

A Windows Store app is different in respect to a Windows desktop application. A Windows Store app is full-screen and all chrome is removed (such as borders and the common buttons for minimize, maximise and close) that are commonly used in desktop applications found in other operating systems at the time of writing, such as Windows 7 or Mac OS X. It is also sandboxed, which means it cannot access files on the computer and in effect resources of another application without the explicit consent of the user (Microsoft, 2012).

2.2.1. XAML

When developing a Windows Store app, XAML can be used to create the graphical user interface of the application, including view related resources such as fonts, colors and behaviors of controls.

XAML, or Extensible Application Markup Language is an XML-based markup language developed by Microsoft. (Microsoft, 2013) XAML for Windows Store apps is a subset of what is available in XAML for Windows Presentation Foundation. XAML can be used to separate a design of an application from the data access layer. A data access layer is often used when developing applications where data is saved or loaded, for instance from persistent storage such as a database, or a cloud-based storage.

Since the design is separate from the other parts, designers don’t have to be familiar with a particular programming language when creating the look of an application. On the other hand, every programmer now needs to learn how XAML works in order to create the look of an application. Further more, many of the markup features available in WPF is not available when developing a Windows Store app, and some controls differ and/or are missing as well, which means there’s a small gap if a WPF designer wants to transition into developing a Windows Store app.

2.2.2. Data Binding

When developing an application with XAML the need to display data from the data access layer in the view is solved with the use of data binding, which is a way to tie data from the data access layer to a control in the view.
A piece of data could be a collection object, an XML file, a web service, a database table, a custom object, or even a WPF element such as a Button. Therefore, data binding is about tying together arbitrary .NET objects. (Nathan, 2011)

Data binding is the process that establishes a connection between the application UI and business logic. (Microsoft, 2013)

Data binding is used extensively when using XAML, since every DependencyProperty or property of an object can be bound to another object. Take this XAML code for example.

```xml
public class MainViewModel {
    public string PropertyValue { get; }
}

<Page x:Class="AppCentric.RemoteControl.MainPage"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    <Page.DataContext>
        <vm:MainViewModel/>
    </Page.DataContext>
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
          DataContext="{Binding PropertyValue}"/>
</Page>
```

The Text property of the TextBlock in this example is bound to PropertyValue which links to the property in MainViewModel, which is the DataContext of the control. The DataContext is in some respect inherited between controls with a parent child relationship.

When encountering a Binding without an explicit source object, WPF traverses up the logical tree until it finds a non-null DataContext. (Nathan, 2011)

This means the TextBlock control will search up its visual tree, via Grid and then Page, in order to find a non–null DataContext. So in the example above the DataContext which is defined on Page will be inherited to both Grid and TextBlock.

2.2.3. Model View ViewModel

Since data binding is worked into XAML to such an extent, a popular architectural pattern has evolved that works well with XAML and data binding. It’s called Model View ViewModel (MVVM) and is explained by Josh Smith in detail. There’s a few important aspects to note. First, The ViewModel, never the View, performs all modifications made to the model data. Second, The view classes have no idea that the model classes exist, while the ViewModel and model are unaware of the view. In fact, the model is completely oblivious to the fact that the ViewModel and view exist. (Smith, 2009) This abstraction between layers is used to a great extent in this thesis project.

In effect, every view and control in the view should bind to a ViewModel class, that works as a mediator for fetching and updating data from the model. Since this works so well when developing Windows Store apps, this architectural pattern will be used when developing the application.
2.3. Test-driven development

Test-driven development is a development strategy where certain units of code can be verified to act in a certain way against other units of code.

2.3.1. Overview

When working with test-driven development a test module is created that tests features of the application. The test module is composed of several small tests that test each individual unit of source code. This is also referred to as unit testing, that is each individual unit of an application is tested. When working with test-driven development test cases are created at the start, and in effect should fail before development starts since no logic has been implemented.

Another thing to note is that future readers of the code can read the tests to understand the behavior a class should have with respect to its dependencies. It also describes how a class can be used, and what state, if necessary, dependencies should have in order for the class to work as expected.

It can also help reduce total development time (Müller, 2013) since the behavior of a component is tested with respect to its dependencies at an early stage. If a dependency changes or if a refactor in the class is made, all tests that depends on that change will be validated again and the programmer will immediately see if a test fails.

2.3.2. Arrange, Act, Assert

Arrange, Act, Assert is a pattern when setting up a test. The pattern is used to make tests easier to understand for programmers other than the author since it separates what is being tested from the setup and verification steps (Grigg, 2012). An example might describe the pattern best.

```csharp
[Test]
public void CallReturnsValue() {
    // Arrange
    var input = "some value";
    Model model = new Model();
    // Act
    var result = model.Process(input);
    // Assert
    Assert.That(result, Is.Not.Null);
}
```

The Arrange part is where all dependencies and values used by the test can be initialized. The Act part should use the method on the unit to be tested. Lastly in the Assert part all behavioral verifications and expected results can be validated.

2.3.3. Abstraction of dependencies

When working with tests, one of the key concepts is that all dependencies on auxiliary services should be avoided. This includes the file system, resources and web requests among others. The reason is that we are testing a behavior of a class, and not how the dependency behaves. If a class uses a file from a file system and that file does not exist, the test might fail because the test might expect the file to be properly loaded and read.
This means that when a new class is created and it depends on an auxiliary service, that service is abstracted away as an interface and is passed to the class via a constructor or via a property. This is called dependency injection or property injection, respectively depending on how the dependency is set. This pattern is similar to the behavioral design pattern Strategy pattern.

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (Freeman & Freeman, 2004)

Because we abstract away services using the strategy pattern and by using interfaces, tests can easily use a custom class that inherits from the interface and returns a fixed value for instance. This is so common in TDD that there’s a way to create these custom classes using proxy classes, called a Mock.

2.3.4. Mocking
To mock, or to be mocked is defined as

1. To treat with ridicule or contempt; deride.
2. To imitate; counterfeit
3. An imitation or a counterfeit. (TheFreeDictionary, 2013)

In other words we are passing a counterfeit dependency to the tested class that can return a fixed value or throw an exception, depending on what the test should assert. When mocking a dependency the behavior of the dependencies are consistent between tests, and hence we can validate that the result should be consistent as well.

2.3.4.1. Mock vs Stub
According to Martin Fowler, there’s a difference between a mock and a stub. In short words, a mock is used for behavior verification, and a stub is used for state verification (Fowler, 2007). A mock is used when we want to test a behavior of a class with respect to another class, for instance test the behavior of a method being called on a dependency. A stub is instead used if the behavior is not interesting in the test, for instance when returning a fixed value. So a dependency can return a stub as a return value, which in turn has a fixed state.

The difference might be best described by the following pseudo code.

```java
public class SomeTest {
    Stub state = new Stub() { Value = "" };
    Mock behaviour = new Mock();
    behaviour.CallMe.Returns(state);
    unit = new Unit(behaviour);

    unit.UpdateState();

    behaviour.VerifyThatMethodWasCalled("CallMe");
    Assert.That(state.Value, Is.EqualTo("state"));
}
```
The behaviour mock is used to verify that its method `CallMe` was in fact called. The stub is not interested in verifying any such calls, but is instead only used as a return value from the mocks, and to verify that the properties of the stub variable is updated given a certain state.

2.3.5. Testing with Model View ViewModel
A page or view class uses a view model class which in turn can have dependencies on a service class. A view model class also uses a model class with the actual state of the object. As described earlier, all behavior of a view is delegated to the view model, so there’s no need to write tests for a view. Instead the tests will focus on the service, view model and model classes.

2.4. SOLID

2.4.1. Overview
SOLID is an abbreviation of five principles when developing using an object oriented programming language. These are The Single Responsibility Principle, The Open Closed Principle, The Liskov Substitution Principle, The Interface Segregation Principle, and The Dependency Inversion Principle (Martin & Martin, 2007). Two of these are more important in respect to this thesis work.

The concepts introduced in this chapter will be used when designing the application.

2.4.2. Single Responsibility Principle

*A class should have one, and only one, reason to change.* (Martin & Martin, 2007)

This can also be described as a class should have one responsibility or concern. It is often appealing when programming to create a class that knows just about everything in an application, and can do as much. The problem with such a class is it will be difficult to maintain and may limit the extensibility of an application, since changes to the class may affect classes that depend on it.

As an example, a class that parses some arbitrary data should with this in mind not be responsible of where the data comes from, but should rather depend on a stream or a service that fetches the data for it. This means that the parser class can parse data from a local or remote resource without changing the internal implementation, simply by passing a different dependency to it.

2.4.3. Dependency Inversion Principle

*Depend on abstractions, not on concretions.* (Martin & Martin, 2007)

This can be compared to the dependency injection concept introduced earlier in this report, and that dependencies should be interfaces, not concrete classes. This goes hand in hand with the single responsibility principle as well. Each class should have one responsibility, and depend on the rest.

2.4.4. Coupling and Cohesion

*Coupling and cohesion* is described in a report by Timothy A. Budd.

- *Cohesion is the degree to which the tasks assigned to a component seem to form a meaningful unit. Want to maximize cohesion.*
Coupling is the degree to which the ability to fulfill a certain responsibility depends upon the actions of another component. Want to minimize coupling. (Budd, 1996)

The concept of grouping tasks related to each other is somewhat related to the dependency injection concept. If the output of a dependency is the input to the component the cohesion is high as the unit of the two components are grouped together.

Furthermore, by depending on interfaces the coupling is decreased since the dependency is not a concrete class but rather a contract a class should fulfill in order to be used.

With this in mind the five principles goes well with the concepts of cohesion and coupling, since a system or application created using these principles will create a system with loose coupling and high cohesion.
3. Using test-driven development with WinRT

Since the project was developed using test-driven development, a test framework and a mocking framework were required. The company had previously not developed applications targeting WinRT, so these two frameworks had to be evaluated before usage.

3.1. Why use test-driven development

In this thesis project there was a need to validate that the services returned a correct value based on some input. A large part of testing was of course the service classes that targeted the SDP web service regarding download of content.

In order to test a web service with test-driven development responses are downloaded for each request that the application could send. The response data was used to test that the class parsing a response created instances of the data objects detailed in the response data. The classes were also tested for error behavior, mocking an incorrect response to verify that the exception was handled gracefully in the application.

3.2. Test framework

The test framework deals with finding methods in a project used for testing, as well as verifying the assertions made in those methods. In C# and other .NET languages classes and methods intended for use in testing are decorated with attributes, which the test framework can find using reflection.

3.2.1. Choosing a test framework

There were a few test frameworks available when searching. One popular test framework was NUnit, which had been used previously in other projects at the company that ordered the application. Since a familiarity was already established, this was chosen as the test framework to use in this project as well. NUnit provides classes for asserting values during a test using a more natural language compared to the built in assert functions provided by Microsoft and .NET. For instance, NUnit lets you type

```csharp
Assert.That(result, Is.Not.Null);
```

Compared to the assert used by the regular assert class.

```csharp
Assert.IsNotNull(result);
```

Both provide the same functionality, but NUnit made it easy to read the test code and see what was being verified.

3.3. Choosing a mocking framework

A mocking framework has another concern than that of the test framework. The mocking framework generates dependencies with some behavior that a unit under test can use. The mocked dependency can be used to verify that the unit under test behaves in a correct fashion, by counting method calls and keeping track of state during testing.
This is why dependency injection works very well with test-driven development, since mocked dependencies which are created based on interfaces can be passed to classes that need to be tested.

3.3.1. Requirements
There were a few requirements the mocking framework needed to accomplish.

- Count method calls for a particular method
- Return different values between invokes of a method.
  This was important since calls to a method in the code were expected to change between calls.
- Assert that a specific method was called during a test.
  In order to properly verify usage of services
- Throw exceptions after invoke of a method.
  This is important when validating error handling, for example a `FileNotFoundException` or `IOException` when using classes handling IO operations.
- Return values based on input.
- Have access to variables passed to a method for assertion.
- Trigger fake events

These were set up based on the requirements of the application and the structure of how it was built.

3.3.2. Candidates
The mock frameworks found during search were Rhino Mocks, Moq, MoqRT, NSubstitute and Microsoft Fakes. Out of the five, Microsoft Fakes is only available when using a paid version of Microsoft Visual Studio Ultimate 2012 or a more expensive version All the other frameworks are free. Because the customer was not interested in using Microsoft Fakes, it was omitted from testing.

The similarity in names of Moq and MoqRT is because the MoqRT framework is a rewrite of the Moq mocking framework intended for use with WinRT.

3.3.3. Difficulties with mocking frameworks and Windows store apps
The common mocking frameworks were not available to use in Windows Store app projects due to the limitation of being able to generate classes dynamically. This feature is not available in the Windows Runtime framework, as opposed to the standard .NET framework. Only one of the frameworks was able to be used in Windows Store app projects, and that was MoqRT. Unfortunately there were other limitations with MoqRT.

MoqRT had some limitations which made it unable to use though. It could not be used to mock classes that contained events, since an exception was thrown if they did. It was also not possible to create a mock of an object containing properties with a return type inheriting from ValueType, such as integer values or enum values, since a random value was returned from these instead of the intended mocked value.
The problem as described earlier with dynamically generated classes are only present when using a project type in Visual Studio that targets the WinRT development platform. The solution to the problem is to have two solution files, each sharing the same source files, but targeting different target platforms, Windows Store apps and regular .NET.

There are some limitations to this; the solution files requires manual maintenance, and there’s differences between the core libraries of WinRT and .NET. The first problem describes that of adding source files to one of the solutions, which means it needs to be added to the other solution as well manually, since there’s no link between the two. The other problem is that of missing classes in the core library from Microsoft between the two platforms. This was solved somewhat by tricking the test project that it could use WinRT components, but this only worked to a certain degree.

In places where WinRT specific code was not available in the .NET test project, preprocessor directives were used to omit the offending code using #if blocks. Since the implementation of some classes could not be included in the test solution, some features could as a result not be tested. This had no large effect on the overall testing though, since classes using these missing core features were moved to a separate component that could be mocked when used by other parts of the application due to dependency injection.

Even though setting up the test solution this way is somewhat time consuming and needs to be maintained manually, there’s a nice payoff in that the fallacies of MoqRT can be ignored. This meant other mock libraries could be used as well.

3.3.4. Evaluation of mocking frameworks

MoqRT was the only framework which failed to meet all the demands set up by the requirements. The two requirements that failed are displayed in this table.

<table>
<thead>
<tr>
<th>Events can not be mocked</th>
<th>MoqRT</th>
<th>Moq</th>
<th>RhinoMocks</th>
<th>NSubstitute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties with return type of ValueType returns random values</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All of the other mocking frameworks were able to handle the requirements set up earlier. Therefore all frameworks were suitable to use in the project. In order to guide the decision a short comparison between the frameworks were made, and is available in the table below.

<table>
<thead>
<tr>
<th>Lazy evaluation of methods in arrange</th>
<th>MoqRT</th>
<th>Moq</th>
<th>RhinoMocks</th>
<th>NSubstitute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>No distinction between Mock and Stub</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
With Rhino Mocks it was possible to use the Arrange Act Assert pattern, but an older pattern was used in some examples where a record and replay order was used instead. This stems from the fact that Rhino Mocks is an older mocking framework that was developed before the use of the Arrange Act Assert pattern and as such had another way of setting up tests at its start. Newer examples were available though where claims are set up using an Expect method in the arrange phase which are verified in the end, which in some sense follows the pattern.

### 3.3.5. Decision

First of, Moq and NSubstitute does not differentiate between Mock and Stub. NSubstitute doesn’t even use the concept of Mock, it just creates the type the user wants. This fact made it very easy to start using both of them since the difference between mock and stub from the beginning was not really clear. In a sense, both frameworks abstracts behavior and state verification to one concept, which is easier to understand.

All of the frameworks can be used when setting up tests using the Arrange, Act, Assert pattern. Moq uses a method `Verify` which specifies the method that should be called, and takes an argument that specifies the number of times the method should be called. In Rhino mocks the newer examples described the method of setting up constraints in the arrange phase with `Expect`, and those constraints were later verified using a method called `VerifyAllExpectations` in the assert phase. The method `VerifyAllExpectations` removed the need to call verify for each method used. There was another way of verifying one method at a time, and that was to use `AssertWasCalled` instead. With NSubstitute, a method called `Received` is instead used, with the same functionality as `Verify`. All in all the mocking frameworks had all the same features available.

When using Moq, a generic Mock object is instantiated with the mocked interface. The actual mocked object passed as dependencies to other classes is reached via the `Object` property.

```csharp
Mock<IModel> mock = new Mock<IModel>();
IModel model = mock.Object;
```

This is in contrast to Rhino Mocks and NSubstitute where a singleton class is used to create a mock instance, and the returned value is of the type expected.

```csharp
IModel model = MockRepository.GenerateMock<IModel>(); // Rhino Mocks
IModel model = Substitute.For<IModel>(); // NSubstitute
```

Once you get used to it though, they are equally simple to use, and it comes down to more of a personal preference than anything else in the end.

There was one problem when it came to serializing with reflection though. When using Moq the actual mocked object did not have any additional properties other than those that were mocked. This was in contrast to the ones created by Rhino Mocks were the mocked object had dynamically created properties. When using reflection on the additional properties, each of them had their own
dynamically created properties, and so on. Needless to say this created an endless loop and in effect a stackoverflow exception when a test was made for a class that serialized a value of one of those instances. The solution was to use a custom serializer that ignored properties with a special signature, properties exclusive to Rhino Mocks.

All in all the frameworks were equally easy to use, and there was no real benefit of using one framework over the other since they were all able to perform the same tasks in the end.

At the end of comparing mocking frameworks a choice was made to use Rhino Mocks. Even though the other frameworks were easier to get started with and understand, further investigations were needed before abolishing Rhino Mocks. Rhino Mocks have been around for a longer time and as such the level of complexity might stem from backwards compatibility, but as such could also offer more possibilities.
4. Viasat Remote Control

4.1. Overview

This chapter will describe how the application developed during the thesis project works, and the decisions which were made that led up to the finished product. The chapter will begin with describing the web services available to get information and how this information is shared with the STB. With the knowledge of the services, an architectural design of the application is discussed, with the benefits it contains when working with test-driven development. With the use of the architectural design and list of services, a use case and a basic design of the application can be made illustrating the basic functionality the finished product will contain.

The Viasat Remote Control application is a Windows Store app, and as such there is only one window. This raises the issue of sharing data between pages, a page being somewhat similar to what a window is in previous operating systems such as Windows 7 or Mac OSX. The later parts of this chapter will discuss the use of sample data, used for designing and demo purposes, and will end with a short display of how the application looked when completed.

4.2. Communication model

The application and the set-top box will communicate via a third party web service, with the help of the Service Delivery Platform API which describes the available methods for downloading information and controlling the set-top box. The web service is accessed via Internet, which means that the application requires an Internet connection on the device in order to use the functionality.

The advantage of having the web service as an Internet accessed web service is that the device running the application does not need to be on the same network as the set-top box, which means the user can use the functionality of the service anywhere an Internet connection is available.

Another way would be to communicate with the set-top box directly. Unfortunately, the set-top box used by Viasat for this thesis project does not run a server which means it cannot receive requests. Instead, it communicates with the same web service as the application, and listens for requests by the web service for recording programs as well as other requests available via the API.
4.3. Model View ViewModel

The classes in the project followed the class architecture of Model View ViewModel, which is used to separate the view from the responsibilities of the model, which is that of fetching information, for instance downloading data from the EPG service.

The model in turn has the responsibility of using services in order to load appropriate data for each view. With the use of services which were injected via dependency injection to the models, the model could load cached data, download new data if needed, and save data to cache when the download was completed.

The ViewModel layer is responsible of forwarding the information the model supplies, and the information the view needs. Therefore, if the model is missing some information which is needed by the view, the ViewModel layer can create this property. An example of such a property that occurred in this project is that of displaying images. The model had access to an URL of an image, but had no particular reason to load the image data. This was instead handled in the ViewModel, which loaded the image data and could hand that over to the view when requested.

The view model also has view specific methods for controls in the view, alleviated via so called commands. Each control that should do something when interacted with, for instance a button, can bind to a command which is then executed when an appropriate action occurs, for instance when a button is clicked.

Several view model classes were also developed that acted as an intermediate layer between the presentation and the data layer, but was not tied to a particular view. These view model classes were used by the view when displaying content in a collection in a view. Among others were the ChannelViewModel, ContentViewModel, and ScheduledInstanceViewModel. Each of these view model classes were used when presenting collections of data in the view, such as a list of channels, or a list of programs in a channel.
The models in the project never had a direct responsibility of downloading data or sending data to the SDP service, or storing data to the device. Whenever a need of that kind was required, a service class was implemented.

4.4. Provider versus Service

In this project a distinction was made between the concept of a provider and a service.

A provider class acts as a service with lazy instantiation such that when a certain implementation of a class or interface can not be determined at the time of injecting a dependency a provider is passed instead. At a later stage when the information needed to complete is available, the concrete instance of a service is created.

An example in this project was that of creating a token holding authentication information about a logged in user. An authentication token was then used by a client which made the actual requests to the web service. When the application was launched, the token was not created, and as such the creation of a client instance needed to be delayed. This was accomplished with the use of a concrete class of the IClientProvider interface.

The client provider class had one method, Create.

```java
public IClient Create(IAuthToken session) {
    return new SdpClient(session, navigation);
}
```

The method receives the authentication data and creates a concrete class of the IClient interface. Usages of the provider could then call this method whenever the user was logged in and an IAuthToken was created. This pattern works very much like the factory pattern in that the provider creates a concrete class implementation, but the consumer of the provider class only knows the interface of the returned instance, not the concrete one.

A service class uses the façade pattern against some other part in the code base, grouping similar operations into one component. Many of the service classes in this project abstracted operations towards the SDP web service, such as the RecordingService for scheduling and querying recordings, and the ProgramGuideService used to download EPG-data.

There were also such services as the FileService class that hid the true implementation of storing files on the harddrive on the users device. By using service classes like these, new service classes can be implemented that for instance targets another EPG service or stores data to a database, but the consumer classes use them in the same way. With this separation of concerns testability is improved as well, since a service accessing the file system can be mocked and thus no actual file access is required when testing.
In some cases a service would need access to other services as well. One such example was the StateFileStorageService service that was responsible for saving and restoring state for a specific type, available below.

```csharp
public sealed class StateFileStorageService : IStateStorageService
{
    private const string FileName = "cache/state.xml";
    private readonly IFileService storage;
    private INodeNavigator state;

    public StateFileStorageService(IFileService storage)
    {
        this.storage = storage;
    }

    public Task SaveStateAsync(Type page, INodeNavigator navigator)
    {
        // Task.Run will potentially perform on a different thread.
        // The caller can await the Task to create a continuation point
        // which will execute when the task is completed.
        return Task.Run(async () =>
        {
            if (state == null)
            {
                state = new SettingsNavigator("state");
            }
            state.Add(navigator.Root, string.Format("/state/{0}", page.FullName));
            string content = new XmlSettingsSerializer().Serialize(state);
            // save state to storage
            await storage.WriteTextAsync(FileName, content);
        });
    }

    public Task<INodeNavigator> LoadStateAsync(Type page)
    {
        return Task.Run(async () =>
        {
            if (state == null)
            {
                try
                {
                    // read state from storage
                    string content = await storage.ReadTextAsync(FileName);
                    state = content != null
                        ? new XmlSettingsSerializer().Deserialize(content)
                        : new SettingsNavigator("state");
                } catch (Exception)
                {
                    // create default empty state
                    state = new SettingsNavigator("state");
                }
            }
            // only return state related to page parameter
            return state.Find(string.Format("/state/{0}" , page.FullName));
        });
    }
}
```
The base interface exposes two methods, `LoadStateAsync` and `SaveStateAsync`, one for saving state and one for loading state based on a specific `Type`, but makes no reference as to where the state is stored, be that a web service or file storage. Each of the methods are asynchronous since the average call to these methods are expected to exceed 50ms, this of course depends on where the state is stored and the amount of data. The `StateStorageService` class saved state to the harddrive of a users device and used a dependency of type `IFileService` to access the storage, but never handled IO operations by itself. By separating the responsibilities of the classes the `StateStorageService` class can be shared between projects targeting different platforms, simply by passing a different dependency to the constructor. In this case a `FileStorage` class exclusive to the WinRT platform can be used to save files on a device running a Windows Store application. These files were used by the project to save and load state between different executions of the applications.

4.5. Responsibilities for creating concrete dependencies

Given that the classes designed in this thesis project all relies on dependency injection, one might wonder where the dependencies are created in the first place. There were several possibilities available when deciding on an implementation;

- Service providers
- XML configuration files
- Creating the dependencies when they are needed

A service provider would work like the factory design pattern in that it creates a specific class but the consumer is unaware of its actual implementation. The service provider could in that case take into consideration whether the application was in a demo state, running live at a consumer, or in a design state. To create instances of the required classes, it would use reflection with the aid of attributes and/or configuration files that lists what types are available to create in certain conditions. Of course, if a provider is to be used, who then instantiates a concrete service provider class?

Considering the options, the development of this project took the latter route of creating each dependency when they were needed, due to time restrictions and simplicity, weighing in the downside of not being configurable. The dependencies were needed when a view created its data context, the data context being its view model. That means that each view class had the responsibility of creating its own view model, along with any needed dependencies. The view models in the project each have a constructor accepting a corresponding model.

Here is an excerpt from the `RecordingsPage.xaml.cs` file.

```csharp
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    SharedNavigationContext context = (SharedNavigationContext)e.Parameter;
    IProgramService service = context.ProgramService;

    IFileService roamingFileService = new FileService(true);
    INavigationService navigation = new SdpAuthorizationNavigationService();
}```
In this example the DataContext property is set to an instance of the `RecordingViewModel`, which requires a parameter of type `IRecordingModel`. In order to create an instance of the model, other dependencies need to be created. One dependency that isn’t created is the `IProgramService` which is instead fetched from the navigation parameters.

### 4.6. Design choices

When developing an application it needs a purpose or a task it can perform. Designing the application is of course dependent on what the application needs to accomplish, in this case dependent on the use cases.

#### 4.6.1. Use cases

Since the application was not extensive in functionality, the use cases were reduced to the following:

- Viewing the EPG for the current day
- Scheduling a movie for recording
- Searching for a specific movie or episode
- Searching for a type of movie

The use cases were discussed with the customer and a workflow for how the user should work with the application was designed to cater to these.

#### 4.6.2. Application workflow

In order to decide how a user will work with an application, it’s useful to create pictures of how an application could look like before implementing any views. Using the sketches and use cases the concept of workflows can be designed, a workflow being the actions a user needs to make in order to complete a task within the application. For this application, several pictures were designed for each potential view.

- Overview and EPG guide
These were the pages the user would use the most. The overview page lists the users favorite series and movies, and could in the future be used to present recommendations to the user. The EPG guide page displays the current programs available to the user, and also offers the user a chance to see what will be showed at a later period of time.

- Content specific pages

- Management and settings
Last were the management pages, used to display information to the user about the current login status, as well as information about the status of the set-top box; for instance what shows have been recorded earlier. These pages would also give the user an opportunity to display recorded shows and control other aspects of the set-top box, such as what program is displayed or the volume of the television.

The sketches went through some versions before the ones presented here, and not all functionality were represented or available in the final application. With the sketches available, the initial design was made to include several different pages with different responsibilities. A landing page, a program guide page, content specific pages, a device information page and recordings overview and a settings page.

At a later stage the settings page was not needed since no global application settings were produced. The content specific pages were also not included since the program guide page displayed the needed information about a show, along with tasks related to them. The application did not include information about the harddrive capacity in the schedule editing view page since the SDP service did not provide this information.

4.7. Sharing data between pages

When navigating between pages sometimes pages should share data between them. In WinRT, there’s no relationship between pages that makes one aware of the other on an instance basis. Instead, a navigate method is used on a frame instance available from the current window instance. The navigate method takes two parameters, the type of page to navigate to, and a parameter to pass to that page.

Each page has access to the parameters that were passed to it in its OnNavigatedTo method. In the project, when the main page navigates to another page the main view model also sends along its services, the program service in this case. Sharing the program service means cached data can be shared instead of needing to download or load it from a cache in the second page.

Sharing data by passing it via the navigate method was chosen over using for instance a singleton class, or saving data in the App class, since it minimizes cohesion between the App class and each page class. There’s also the concept of only sharing with a particular other class, instead of sharing
instances with the entire application. Changes made to a global instance may be difficult to track, but a shared instance between two objects is not. A problem can arise if the programmer forgets to attach parameters when navigating. This was solved by using a navigate command class that encapsulates all navigation in the application.

4.8. Sample data

When designing the user interface for the application, working with sample data in the visual designer makes creating the design of pages and controls easier, faster and less error-prone. The simple explanation is that the visual designer can display data when designing, and the application does not need to be run in order to see if the design pans out with sizes of text and other things that a design includes.

Since the view fetches its data from the view model a view model needs to be created at design time for this to work. This is possible using XAML, by setting the data context of the page to an instance of a view model.

```xml
<PageDataContext>
  <vm:ViewModel/>
</Page>
```

The problem with this approach is that no dependencies can be passed to the view model. However, since the view model can be responsible for creating the sample data if no arguments is passed to it this is a valid solution.

The view model is always created using this approach, even at runtime when no default view model for sample data is needed, which means that two view model instances are created and one is immediately garbage. So instead of setting a view model using the `Page.DataContext` directive, it can instead be set using the `d:Page.DataContext`. The difference here being the prefix `d:` which is parsed when using the visual designer, but ignored by the XAML parser when launching the application due to the `mc:Ignorable="d"` directive in the page xaml.

Further more, a special binding of type DesignInstance can be used when specifying the datacontext. The binding has three parameters, Type, IsDesignTimeCreatable, and CreateList. By using this binding, the view model can be bound as an attribute instead of adding a new element. It also highlights the use of it being a design template, and not the view model used at runtime.

Because a view model of the same type as the one used at runtime is bound when designing, all properties and commands can be picked up by intellisense, as well as displaying demonstrational data in the designer.

```xml
<P x:Class="AppCentric.RemoteControl.RecordingsPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```
There is of course a flaw to this design. The view model now has two responsibilities, relaying the state of the model as well as creating sample data. This breaks the separation of concern principle for the class. The way to mitigate this is to either create a subclass of the view model, or extract the parts which are relayed to the view and create a parent class of which both the sample data view model class inherits from as well as the class to be used at runtime. Another approach is to create a design-specific model in the parameterless constructor of the view model. This would not break the separation, but would instead create a tight bond between a view model and a specific sample data model.

Due to simplicity and the size of this project a decision was made to not create another class, but instead have the view model use a provider class for sample data which it used to fill collections of data items normally loaded from the model. Considering the alternatives, refactoring the view models to instead use a design-specific model would probably create a clearer distinction between the two, and would avoid the view model being polluted with code handling design-time data.

4.9. Dialogs, pages and controls

It was decided to have 4 pages the user could navigate between as a start. This could change later depending on requirements from the client. The pages in the end had one additional page for searching;

- Overview page
- Program guide page
- Recordings overview page
- Device information and remote controller
- Search page

The remote controller page ended up being an information about the users’ devices since the Service Delivery Platform API descriptions for controlling the STB was not correct and no support for the API was available at the time of writing this thesis.

The search page was implemented to only search cached content, and as such if the user had not logged in and downloaded any information no search results were displayed.

Due to time limitations no settings page was implemented. All settings in the application is therefore non-changeable. Settings that are saved are the EPG-data and authentication token for use if the user opens the application before the expiration date. At the moment there’s no way to erase these though.
4.9.1. TimelineTray
When developing a graphical user interface most graphical components can be put together using existing controls. One of the requirements for the projects was to create a view which resembled the customers previous implementations of an EPG-view.

![Figure 4 Sketch of customers previous look of EPG-view](image)

Since a control was not available via the standard control library to accomplish this, a custom control had to be created. In WinRT, a custom control is set up using a design similar to that of Model View ViewModel. A class is created for the control but there’s no code related to drawing on a screen. Instead all view related items are defined using XAML, and is attached to the class using a style and a controltemplate.

Even though the control has a few properties which are set and bound to properties in the viewmodel class, in effect it’s controlled from the view consuming the control. The visual elements are templated meaning these are also not controlled by the custom control, but instead defined using XAML in the resources of the consuming page. The layout of the controls are defined using the control template, and in the specific case in this project, on each occasion where a “channel item” is inserted, a ChannelTemplate item template can be used.

This goes well with the principles set up earlier; TimelineTray shouldn’t be aware of the type of data it displays, and as such this design is minimizing cohesion between the two.

4.9.2. VisualStateManager
The VisualStateManager plays a key role when developing Windows Store apps. It defines the states that the user interface can transition between. As an example the main page in the application has one state for when the user is logged in, and one for when the user is not. Depending on certain events in the view, for instance clicking on a button and logging in, the view can transition between the states with the help of VisualStateManager.

In the application the VisualStateManager was used to setup a transition state from the main page when navigating, since loading a page could take some time. The navigation process loaded the second page in the background when the user requested it, displayed a loading state, and navigated when the page was complete, in effect switching between the pages without delay. This avoided the freezed state the page would otherwise have if the navigation had been displayed and loaded on the user interface thread.
4.10. Service Delivery Platform

Three assemblies were created to separate concerns. Classes in the Service Delivery Platform assembly were data classes that handled data from the web service. The idea of putting these in a separate assembly was to be able to use them in other projects as well, outside from Windows Store development.

A part from the classes described earlier the services used a class called `SdpClient` which was the main channel towards the Service Delivery Platform web service. This class was responsible for creating a request object and, with the help of a parser for JSON, attaching instances of objects to each other. An example is when a schedule was downloaded and an existing channel was used. Instead of parsing the value of channel and using the parsed one, the existing one was used.

```csharp
1. public IResponse<IScheduledInstance> RequestChannelSchedule(IChannel channel)
2. {
3.     IDictionary<string, string> query = getQueryParameters();
4.     string url = createUrl(string.Format("platform/channels/{0}/schedule", channel.ID));
5.     SdpResponse<IScheduledInstance, ScheduledInstance> response =
6.         new SdpResponse<IScheduledInstance, ScheduledInstance>(navigation, url, query, "/schedule/scheduleInstance");
7.     response.ItemCreated += (o, e) =>
8.         { e.Item.Channel = channel; }
9.     return response;
10. }
```

In this example the channel is set when the response instance creates a new instance of the `ScheduledInstance` class. On row 9 an event is set up to trigger when an item is created, and when that happens the client can use existing instances of classes and attach them to each other. This means that the framework in the end did not have to create unnecessary instances of objects when parsing response data from the web service.

Most other classes in this assembly were data classes holding information about a specific data type.

4.11. Screenshots

There were two pages of particular interest when the application was complete. The main overview page and the program guide page which displayed the EPG for all channels.
The overview page displayed series and movies during the day. The idea was that the user would be able to look for a series instead of a particular episode or a movie when browsing.

The program guide page displayed information for all channels the user could access through the STB along with program information. The description of the show was downloaded on demand, when an item was selected in the timeline tray, as to avoid unnecessary downloads.
5. Results

This chapter will go over the results of the method and the finished application.

When the application was completed there were some issues that had not been dealt with. The API provided by NDS had some limitations making it impossible to accomplish some of the goals listed at the start. This chapter discusses those limitations and afterwards reflects on the goals and requirements set up at the start of the report.

5.1. Missing features

One of the goals of this application was to control the STB via the application, specifically scheduling a program for recording but also playing a recorded instance on the television. This was not completed, since the Service Developer Platform API specification was not up to date and as such did not work. Unfortunately, the support was not able to help, so this requirement did not get finished in the application and the feature is hence unavailable. Since the methods and planning of classes and visuals were created though, adding this functionality at a later stage would not prove difficult.

The only problem with the API though was the actual control aspect. Downloading information, EPG-data and device capabilities, were available, and as such the application today can display a program guide tailored for a specific user.

In order to download EPG-data a request was made to the SDP service. Unfortunately, the request to download all programs in one batch did not work so an alternative method was used. Instead, a list of channels was required to be downloaded and a request was made for each channel to display the programs available on that channel. This proved somewhat time-consuming. In the test data offered by the service 60 channels were available, which in turn made downloading the entire EPG-list to have at least 61 HTTP requests. Since each channel had data for about a day ahead, the data was split into several requests to only download the information that was to be displayed. In the application this time period ranged for three hours from the current hour of the system time on the device, which was adjustable by the user. In effect, each channel downloaded data in packages of 10 programs for each request. If each channel have 30 shows a day the number of requests for a whole day would amount to 180 requests, which is a lot of overhead for downloading about 800 KB of data. This was mitigated somewhat in the application since each request for downloading a channels programs was doubled each time a request was made, so the early programs were downloaded fast, and later programs were downloaded slower, but with less overhead. Processing the data was set up to display every time a list of programs available to a channel became available, so the user would get incremental updates for each channel that downloaded information. At least this did not freeze the application and the user could browse the channels programs as they were downloaded instead of waiting, since a full download could take about 4 minutes. With cached data, the deserialization and displaying of the GUI took about 5 seconds.
As the channel and program data was downloaded, additional information about specific applications would be downloaded when the user selects a program from the timeline tray in the program guide page. The additional information displayed in the application was the description of the program and a poster image.

The application saved EPG-data and the authentication token in the roaming file storage. There is a limitation of the size of the storage in order to sync via the built-in cloud storage available for Windows 8 devices but this limit was not addressed during this thesis work. The reasoning of all data being saved in the roaming storage is to have the user logged in at all their devices. The EPG-data was saved there as well, but this could most likely be put in the local file storage instead because of the amount of data downloaded. The EPG-data could easily be downloaded from the EPG-service instead of the cloud storage without affecting the user experience, other than the time aspect mentioned earlier.

5.2. Test-driven development

Working with test-driven development in projects targeting Windows Store apps was, based on the result of the application, a success. When the application was run on the target device for the first time all features which had tests set up were found to have no issues, and the user was able to login and download information without errors. Based on previous experience in other projects, the first run of an application seldom perform as flawless as that.

Some classes are not shared between the .NET 4.5 version and .NET for Windows Store apps. In particular, accessing file storage and the network classes for accomplishing HTTP requests differ between the two platforms. In effect classes that dealt with these components were excluded from testing since there was no suitable mocking framework for WinRT development. This was reduced to two classes in total, since the separation of concern for accessing file storage was put in one class, and was injected to those classes that needed it, and a network access class respectively.

All view classes were also omitted from testing, since all code related to views that should be tested is rerouted to the view model classes, which were included in the tests.

Since the project in this thesis work was set up using .NET, the Model View ViewModel architecture, dependency injection and the principles of SOLID, it was easy to share the implementation of classes between different platforms, simply by encapsulating exclusive parts to separate classes as explained earlier.

5.2.1. Test cases

This project worked with test-driven development, and therefore with test cases, or unit tests. When the application was completed, there were 41 unit tests in the test project. Out of these 27 were making sure that the core library for communicating with the NDS service was downloading and parsing information from the web service. The tests were also set up to verify that the client created the correct instances of content classes based on the response. Based on the tests, the communication
was verified to behave in a reliable way. In other words based on certain input or errors the user interface should have the correct information available to present data to the user.

As an example, the method for downloading channel information was tested to return a success state when data was returned.

```csharp
[Test]
public async void RequestChannelScheduleReturnsSuccess()
{
    // Mock
    var navigation = MockRepository.GenerateStub<INavigationService>();
    var channel = MockRepository.GenerateStub<IChannel>();
    var loader = new WebResponseLoader(
        new Response_200_PlatformChannelSchedule_OK_JSON());
    const string channelId = "20";

    navigation.Stub(m => m.RequestAsync(string.Format(
        "https://api.sdp.nds.com/v1.1/platform/channels/20/schedule" +
        "?format=json&oauth_token={0}&count=10",
        accesstoken)))
        .Return(Task.FromResult<WebResponse>(loader));

    channel.Stub(c => c.ID).Return(channelId);

    // Act
    SdpClient client = new SdpClient(token, navigation);
    IScheduledInstance actual = client.RequestChannelSchedule(channel);
    bool success = await actual.Next();

    // Assert
    Assert.That(actual.Success, Is.True);
    Assert.That(success, Is.True);
}
```

The test above is set up to load a fake response using a mocked navigation service. All responses from the web service needed by the application was downloaded beforehand in order to use them in tests like the one above.

Tests were also added which made sure that errors that could happen during communication, such as the client being disconnected or getting incorrect data back were handled correct. Errors that occurred in the service classes were caught by the model classes, and the view model classes could either ignore the error or report the error to the user via the views. Some errors which were easy to verify later in the application was that of trying to log in with an incorrect user, or sending an incorrect request to the web service. Sending incorrect requests should not happen unless the service API is changed, which should not happen since a version number is sent with each request, but either way it was handled.

The other unit tests in the same project had a similar structure, which is that of creating a fake response and verifying the result. Error handling were also tested, as an example the code for handling an authentication token to storage handled the error of a non-existing file.

```csharp
[Test]
public void StorageCantLoadUnsavedToken()
{
    // Act
    StorageClient client = new StorageClient();
    UnsavedToken actual = client.LoadUnsavedToken();
    bool isNull = null;

    // Assert
    Assert.That(actual, Is.Null);
    Assert.That(isNull, Is.False);
}
```
This test made sure that the service for loading an authentication token returned a null token if the file where the application saved a token to file was missing. Other classes dealing with auxiliary systems such as storage locations were set up in the same way to handle errors that could occur. In the finished application, errors that were set up with tests could occur without the application behaving incorrect or crashing.

Test cases makes sure that the code behaves correct with respect to its dependencies, in turn reducing logic errors when running the application. The result of this statement was verified in this project, since the application could download EPG-data and related information from the service, as well as handle exceptions that could occur at the time of running the application. It was also verified to work when functionality was added or refactoring took place, since old tests could be run to make sure nothing was broken.

5.3. Goals

The goals that were set up in the beginning were not all completed. Some were impossible to accomplish due to third party tools and other goals were not completed due to not having enough time.

1. The application should be able to pair with a STB, meaning the application will be able to detect and send instructions to a specific STB. **Completed**

2. It should be able to display EPG-data for multiple channels at once (**Completed**), and also be able to display EPG-data for the currently viewed channel on the STB. **Not completed**

3. The application should be able to schedule a program to the STB for recording. Scheduled recordings should also be able to be deleted. **Not completed**

The first goal was met, the user can login to the Service Delivery Platform service and get information about what channels and programs are available through their device. The requirement for controlling the STB was not met, since the methods available via the web service for accomplishing this were faulty.

```csharp
{  
    // Mock  
    var fileservice = MockRepository.GenerateMock<IFileService>();  
    var serializeservice = MockRepository.GenerateMock<ISettingsSerializer>();  

    fileservice  
        .Stub(m => m.ReadAllTextAsync(Arg<string>.Is.Anything))  
        .Throw(new FileNotFoundException());  

    // Act  
    var storage = new AuthTokenStorage(fileservice, serializeservice);  
    IAuthToken actual;  
    bool loaded = storage.TryLoadToken(out actual);  

    // Assert  
    Assert.That(loaded, Is.False);  
    Assert.That(actual, Is.Null);  
}
The visual components of the application were met to some degree. The most important aspect of the application, the program guide with the use of TimelineTray was completed. Certain colors and icons were not included as they were not the focus of this thesis work. Even though they were not completed, the application is setup to make these changes easy to adjust at a later stage by changing the XAML files, so no coding is required.

### 5.4. Measurable requirements

The requirements set up on the application were all met, save one. In contrast to the goals the requirements were more based on the user interface, and as such were easier to affect during the development.

- A page in the application should be displayed within 2 seconds of requesting to see that page. **Completed**
- The user of the application should be able to scroll when viewing EPG-data to see at least 24 hours of data from the current time of day. **Completed**
- When a user changes page, the application should be able to load and display the other page in at least 2 seconds; aborting or suspending downloads from the currently viewed page in order to meet the demands. **Completed**
- The user should be able to configure a timespan for scrolling in the view for EPG-data, to for instance scroll 24 hours ahead to make viewing the next day of EPG-data easier. **Not completed**
- The user interface should not freeze when the user performs a task. **Completed**
- The view for displaying EPG-data for multiple channels should resemble the one used by Viasat on the STB. **Completed**

The only requirement not met was the ability to specify a timespan to scroll through the data. Since downloading of data was limited to about 24 hours ahead by the web service, there was no need to scroll further than that in the user interface. This made the requirement not that important and as such it was not implemented.

Since most of the actions performed by the view models were handled asynchronous, the navigation was not notably affected and worked very fast. When a page requested information it was always scheduled to run after the visual components of the page was displayed. This created a delay in the information to be displayed, but the user could work with the application and the user interface was not frozen during the load period.

The user interface was tested using a specific run of the application. A small breakdown for this run can be seen below. The run consisted of a few steps to normalize the time:

1. Start application (Main page)
2. Log in and start to load data
3. Open app bar and navigate to program guide page
4. Let the page load completely
5. Navigate back
6. Open app bar and navigate to recordings page
7. Navigate back
8. Open app bar and navigate to device information

With this setup, all pages were visited during load and when the application was idle. The run was performed about 20 times and the values measured were very similar. The run was performed on a Windows 8 tablet PC, Intel Atom 1.8GHz with 2GB RAM which represents a Windows 8 device that can be used as of writing this report.

<table>
<thead>
<tr>
<th>Loading page</th>
<th>3,6479s</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceInformationPage</td>
<td>0,60s</td>
</tr>
<tr>
<td>ProgramGuidePage</td>
<td>6,88s</td>
</tr>
<tr>
<td>RecordingsPage</td>
<td>0,23s</td>
</tr>
<tr>
<td>Switching page</td>
<td>0,29s</td>
</tr>
<tr>
<td>DeviceInformationPage</td>
<td>0,15s</td>
</tr>
<tr>
<td>ProgramGuidePage</td>
<td>0,45s</td>
</tr>
<tr>
<td>RecordingsPage</td>
<td>0,10s</td>
</tr>
</tbody>
</table>

The table displays two measured values for forward navigation. During development, the target tablet PC for measurements was very slow when navigating and displaying a page. The navigation was rather fast, as can be seen, but creating and displaying the timeline tray was cumbersome and the page was not displayed until it was fully loaded. The first requirement in the list of requirements correspond to the “Switching page” column, where the longest display time is 0.5s which is reasonable below 2s.

This was not mitigated by aborting threads or downloads, since measurements were done even when those were not running. Instead, a navigation state was introduced in the main page which showed a progress bar when the user navigated away, as to display that the application was working and therefore the user interface would not freeze.

This was most likely due to the timelinetray rendering about 200 controls at the time of load. There was no time to investigate how to accomplish how one would display a page and then load the controls so the page would not stall the application.

The time measured was done in two ways. The first was in code, where a timer started when the user navigated from a page and stopped when a page was loaded, and the second was a manual stopwatch. The timings of the stopwatch were very similar to those above, and as such are not displayed.

Back navigation was very fast, less than 1 second and therefore those measurements are not shown.
6. Reflections

The reflections made in this chapter discusses some of the issues discovered when developing the application.

6.1. Communication model

As previously noted the communication between application and set-top box is made possible with the use of a web-service available through the Internet. The download speed of data from the service was not that great though, and in the end presenting the first few hours of EPG-data to the user could take up to several minutes. This result can be based on several things which can be fixed in future changes.

One reason is that the service had not been used that extensively earlier. Only a few other applications had been created to interact with it. So although the service was not optimized yet, this can be mitigated with a faster network for instance.

Unfortunately the methods used by the applications targeting the web service were not optimized either. It was not possible to make a request to download all program information from all channels and sort them by date for instance, which would decrease the number of requests made to the service from about 50 to 1. The reason was not because the method was not available, but because an error occurred when trying to use it. This was a common reason for issues with the application, an incorrect API to the service, and little to no support for fixing it.

If the application is to be further developed, it would be wise to change the communication model, since it was made clear that it took too long to use given the current implementation.

6.2. Test-driven development

During the development of this project some advantages were clear as to why one should use test-driven development;

1. Adding requirements or functionality to classes does not interfere with old requirements
2. The code gets documented in the form of tests instead of code comments. If code changes, the tests needs to change in order to compile and run, while a comment might get obsolete
3. Refactoring code can be immediately verified to not break other parts of code

One thing that wasn’t done in this project was a clear design or a class diagram to know how classes should interact with each other. As to the first item mentioned above; once the class design is clear, adding functionality is very easy, and older tests remain to make sure that new code does not break the old code. Even though a overall idea of how the project was supposed to work, more work could have been used at the early design stage.
During the development of this application, a design choice was made as to moving functionality between two classes. This in effect lead to moving specific tests, and the tests would have to be rewritten with respect to dependencies of the other class. Although time consuming, during the change it was easy to verify that other parts of the code still worked, even though a large code refactoring took place. This would most likely proved to be much more difficult if test-driven development had not been used, since verifying in that case would have required running the application and test each part of the application that depended on the changed units of code.

Regarding the second item above; a test describes how a class can be used, which is what comments are used to as well. But rather than reading a comment on what a method should do, the test describes what a method does. During this project tests were named to reflect this as well, such as StorageReturnsSavedState for instance, a test which makes sure that the StateStorageService class returns state if it is available from the file storage.

Regarding the choice and evaluation of Rhino Mocks. At the end of the development phase the added level of complexity provided by Rhino Mocks when it comes to state versus behavior verification wasn’t needed, at least not in this project. If a decision was to be made again, Rhino Mocks would probably not be chosen, but that may well depend on the type of project developed.

There’s somewhat of a balance between using test-driven development and not. It takes much longer to implement tests for each feature of a class as to not doing it, but future development time is reduced since verification of the code and that it works with respect to its dependencies can be made every time a change is made.

6.2.1. Test-driven development for Windows Store apps
With regards to using test-driven development when developing a Windows Store app; even though an extra project needs to be set up, and some code can’t be tested, the reasons for using test-driven development still outweighs the reasons not to. With the added layer of testability, one can argue that you also get a platform independent application, since the developer needs to abstract away code that only works on one platform.

6.3. Code hierarchy
When developing a Windows Store app the architectural pattern Model View ViewModel is very much recommended, both from the code that is set up when a new project is created, and in documentation and examples offered by Microsoft.

When working with test-driven development the class hierarchy is guided towards creating small units that depend on each other. These two, Model View ViewModel, and test-driven development, together guides a programmer to use the principles set up by SOLID. Whether or not it is needed it will probably depend on the type of project developed. It may very much be a balance between getting a product out fast, and using the product for a long time. If a product should be developed fast, such as a prototype, setting up tests might prove too time-consuming.
These practices can be recommended since other developers reading the code can read the tests to understand how everything works, and maintain the code with the use of the tests, to make sure the logic works even after major changes.

6.4. Lessons learned

Working with API's and frameworks not previously used takes a lot of time. The development was delayed several times due to controls not available for Windows Store apps, issues with Visual Studio in regards to running the application, and creating tests with no prior knowledge.

Windows Store development issued a lot of problems as well. As an example, sometimes when starting the application during the development phase a dialog would show up in Visual Studio stating the application could not be run, with a url that did not link to anything. With no particular reason to why, the solution was to change an ID in a text field in a dialog in Visual Studio for the project. This seemed almost random, and even though it didn’t require more than a few minutes of development time, it was still very annoying.

Another drawback was the lack of controls and possibilities when working with XAML for Windows Store applications. In WPF one is able to attach events to the view model class using so called triggers in XAML. These are not available in XAML for Windows Store applications, but instead Microsoft uses solely the VisualStateManager class to trigger updates to the view model and also for visual transitions. This took some time to get used to and learn.

Using the async await was troublesome when testing commands. Commands in general were not interested of when the asynchronous operation was complete, just the fact that they were completed at some point. This meant that a test that try to verify the behavior of a view model command would have to either wait until certain conditions were made or ignore the test altogether. The problem with waiting for a certain condition is the fact that it may never trigger, leaving the test in a deadlock state. This was mitigated for the most part by waiting a max amount of time, which is not ideal, but solves the issue.

6.4.1. Writing tests

As well as frameworks, the other time-consuming part was constructing tests without prior knowledge. One of the difficulties in particular was when a mock could not be used. When using MoqRT this was more noticable because it obviously had constraints, and when using Rhino Mocks basically everything could be mocked, from method calls to raising events.

Here I will list some short paragraphs of things learned while working with test-driven development in this thesis project.

1. Test one feature at the time.

   This means that each tested method can have several tests, and each test verifies different things. This makes it easy to spot what fails in a method, and what might fail if a refactor took place.
2. Separation of concerns.
   When working with TDD and in effect dependency injection I noticed that it’s very easy to separate
   the responsibilities of each class from each other. Based on experience, since each class basically has
   one responsibility and depends on services for everything else it is easier to maintain in the future, and
   easier to read for other programmers.

3. Avoid setup methods of tests
   NUnit used a special attribute for a method that could be invoked before each test ran, in which
   common used variables could be stored. This pattern was used for a long time during the development
   but when tests were later revised it was discovered that they were not easy to follow, since setup code
   were separated between different methods. As a programmer it is easy to think that duplicate code
   should be avoided, but when it comes to setting up tests each method should be considered its own
   unit, with all initialization code in it to improve readability. This also makes it easier to move tests if
   code is moved between units, since a test is contained in one place.

6.5. Solution
The problem in this thesis work was to create an application which communicated with a web
service, and via that service control a set-top box which displayed data related to television based
services. With the use of test-driven development, communication with the web service was set
up with the use of tests to specify what the application should do based on certain conditions that
could occur when communicating with the service. Given that, the communication worked very
well. The only problem was the lack of speed and the ability to change the implementation of the
web service to operate faster. The other problem was that of controlling the set-top box, which did
not work due to the fact that the web service did not function properly.

Another problem during this thesis work was that of creating a responsive user interface. The large
control developed during this thesis project that displayed information about what channels were
available and what programs were displayed on these did not perform as rapidly as was expected.
Although this was mitigated somewhat by not displaying the page containing the control until the
control was loaded, in retrospect the control should have been able to control its load in another way,
by loading in the background while displaying a progress bar for instance.

All in all the application was a good start for the company that ordered it to continue work if a
decision was made to continue its development.
7. Bibliography


8. Appendix A

The Service Delivery Platform (SDP) is the name of the service developed by NDS in order to communicate with the STB used by Viasat. If not otherwise specified the content in this chapter is based on the API specification for the SDP v1.1 (NDS, 2012).

8.1. NDS

NDS is the company that provides the SDP API.

8.2. Security

In order to use the service an API key and an API secret have to be provided to the service. These are exclusive for each application and are obtained by registering an application in the SDP developer portal.

Furthermore each user of the application needs a username and a password to login with. The service can use these users credentials to provide personal information based on the users STB and filter information accordingly.

8.3. OAuth 2.0

In order to access the API, the user needs to sign on to the service. The sign on process is done using OAuth 2.0 which is described in request for comment article #6749.

> The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. (Internet Engineering Task Force (IETF), 2012)

In order to download any data from the service the user needs to be logged in. The SDP sign on process issues an authorization token that is valid for about 24 hours. This means the user needs to login every 24 hours unless the application is used before the expiry date, in which case a request for a new authorization token can be issued by the application using the old one as validation, omitting the need to enter credentials again.

8.4. REST

REST, or Representational State Transfer, is a protocol for transmitting object data over an HTTP channel.

Each request to SDP is performed using HTTP and the response is returned using REST. The format of the response can be either XML or JSON by using a control argument when requesting data. Both are formats of representing object data using human readable text as opposed to binary data.

An example of an XML response:
8.5. Functionality

Using the API, the user can browse available programs, detect what is being viewed and schedule programs to be recorded. The application developed in the thesis work will focus on displaying EPG-data and being able to schedule shows for recording.

8.6. Downloading EPG-data

The user can request content information along with meta-data such as if a show is in a series or if it is a movie. The user can also browse the channels available in the STB along with the shows displayed during the day.

8.7. Controlling the STB

Apart from scheduling shows for recording, the user can via the API also control other aspects of the STB. As an example, the user can power on or off the STB, control the volume or play some recorded content on the device.

8.8. Key components

The API is extensive and lists several resource types that can be requested from the service. A few key components can be important to be aware though which is related to displaying EPG-data.

8.9. Content

Content represents metadata about any TV content that is made available on a platform (movie, tv show, series...).

A content response consists of a title and a synopsis among other fields. The synopsis is another word for the description of a show. The response also contains information for logotypes and references to people related in creating the content, and has some additional meta-data as well. This is an abstract description of a show in some respect.

8.10. Channels

This resource represents a specific channel.

A channel response consists of an id in order to request related content, along with a name and a number so the user can identify it. It also contains a description and some additional meta-data.
8.11. ScheduledInstance

A ScheduledInstance is the actual broadcast of a content on a channel. In other words it can be viewed as a connection between a content instance and a channel instance, along with the time broadcasted and formats for the displayed content such as display aspect ratio and audio. It also contains a duration field as this might differ from the content duration. When displaying content on a channel with commercials for instance the duration will be longer than that of the actual content duration.