Abstract

In recent years, functional programming has emerged from the academic world and has steadily gained popularity in the software industry. Still, it has not yet fully arrived in all application fields where it could bring its advantages to bear. This thesis examines the suitability of functional programming for the development of mobile applications using F# and Xamarin and compares it to the established method of object-oriented programming with C#. It therefore first shows how apps are typically built today within the Xamarin ecosystem. It then introduces functional programming in general and F# in particular, and it examines how app development with Xamarin could benefit from it. A case study is then carried out in which an app for Android and iOS is implemented twice: once object-oriented with C#, once functional with F#. The evaluation of this case study shows that functional programming with F# can be a viable alternative to the established object-oriented approach with C#.

Keywords: *Functional Programming, OOP, F#, C#, Xamarin, iOS, Android*

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

API	Application Programming Interface
BCL	Base Class Library
CDs	Cognotive Dimensions of Notations
DSL	Domain-specific Language
FP	Functional Programming
FSI	F# Interactive
GUI	Graphical User Interface
HCI	Human Computer Interaction
IDE	Integrated Development Environment
IL	Intermediate Language
MVU	Model-View-Update
MVVM	Model-View-View Model
JVM	Java Virtual Machine
OOP	Object-oriented Programming
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

1 Introduction

1.1 Motivation

Industry developers have long spurned functional programming as a niche product that can only be used in either academics or a few rather exotic problem domains. The origin of this perception may have been that functional programming languages often ”look” and work fundamentally different than more popular ones, e.g., from the C family.

That is not entirely dismissable. The first functional programming language, LISP, has been introduced 60 years ago by John McCarthy (McCarthy, 1959). Based on the groundbreaking work of Alonzo Church on the Lambda calculus (A. Church and Rosser, 1936; A. Church, 1941), McCarthy created a whole new category of programming languages (Turner, 2012). Those languages would become widespread and successful first and foremost in the scientific community in the decades to come.

However, since the early 2000s, functional programming has become more and more popular in the software industry as well. Programming languages that call themselves ”functional-first,” such as Scala and F# have been introduced. And established ”object-oriented” programming languages such as Java and C# adopt functional ideas with accelerating speed.

Furthermore, it can regularly be observed that experienced developers with a background from the object-oriented world, who come into contact with functional programming, soon become almost enthusiastic: ”Once you learn the benefits of functional programming, you find that it improves all the code you write. When I learned functional programming a few years ago, it re-energized my enthusiasm for programming. I saw new, exciting ways to approach old problems. The rigor of functional programming complemented the design and testing benefits of *test-driven development*, giving me greater confidence in my work.” (Wampler, 2011, p. vii)

At the same time, many companies still struggle to apply functional concepts and methods in concrete software projects. Although it can be assumed that doing so could provide them a competitive edge, as the example of Jet.com shows. For tactical reasons, Jet.com decided to adopt functional programming with the explicit goal of becoming an attractive target for the most talented developers on the market (Han, 2015). The company was eventually sold to Walmart for $3.3 billion only 2.5 years after it had been founded[1].

So functional programming is an established and mature programming paradigm. A variety of programming languages support it. Many developers who try it quickly ”get hooked.” Also, from an economic point of view of a company, it is possible to build a conclusive argument for its introduction.

[1] https://techcrunch.com/2016/08/08/confirmed-walmart-buys-jet-com-for-3b-in-cash/ (retrieved August 8, 2019)

But still, in practice, it does not always find application in all the fields it would presumably provide benefits over current mainstream approaches.

That is the starting point of this thesis, which will explore the suitability of functional programming in one of the most popular areas of commercial software development of the last decade – mobile application development. It chooses the Xamarin platform which not only features object-oriented programming with the popular language C# but also supports an often overlooked but powerful functional programming language: F#.

Hence this research question arises: How suited is functional programming with F# compared to object-oriented programming with C# for the development of mobile applications with Xamarin?

The author hypothesizes that the ecosystem around F# has reached a state where it enables Xamarin developers to build comprehensive mobile applications. This is made possible by leveraging modern functional programming concepts, a rich set of libraries, and support by major IDEs. Functional programming, therefore, is a viable alternative to the established object-oriented approach with C#.

It is out of the question that mobile applications built on the Xamarin platform can at least technically be written in F#. But how far does it go? This thesis will try to find out if it is possible to not only build such software with the ”functional-first programming language” F# but to leverage FP in a more holistic way. A way that not only benefits from a couple of functional features in the language itself. But a way that harnesses competitive advantages of FP by implementing concepts and utilizing libraries and frameworks deeply rooted in the F# ecosystem. A way that produces results that can compete with those built through the traditional imperative and object-oriented approach.

Secondary questions, which will also be covered: Why is FP still so uncommon in the software industry? Could C# developers apply FP in their language as well? And what needs to be considered when C# and F# are being mixed together?

1.2 Related work

Little research has been done so far with regards to the application of functional programming in the mobile application space. There do exist plenty of papers on both functional programming and mobile app development, but with almost no intersection.

The most remarkable paper in this regard has been published by Petersen, Gorges, Dunsmuir, Ansermino, and Dumont, 2013, who have shown how to develop a cross-platform framework for embedded and mobile applications written in Scheme for the medical field.

Kiss, 2014 compared functional and object-oriented programming approaches for the development of graphical user interfaces (GUIs) in an exhaustive study, using Java and Scala primarily.

Czaplicki, 2012 introduced a new functional programming language called Elm. Elm is based on the ideas of Functional Reactive Programming and provided important preliminary work for derived concepts used to build web and mobile applications in various programming environments.

1.3 Structure

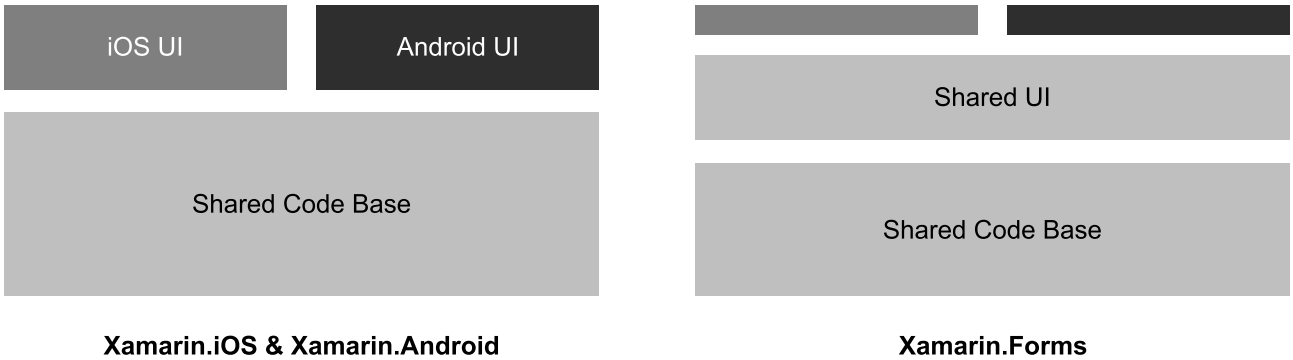
This thesis will first introduce mobile app development with Xamarin and show how apps are typically built today within the Xamarin ecosystem. It will then introduce functional programming in general, examine the paradigm's current state of adoption in the industry, and then take a look at FP in .NET, with emphasis on F# and a short excursion to C#. The next step is to show how functional programming could be implemented in the context of Xamarin. Three different ways are identified, which all allow to leverage FP to a different degree. Based on these theoretical fundamentals, a case study can now be carried out. The requirements and research methodology are discussed, and finally, the results are presented and evaluated.

2 Mobile app development with Xamarin

2.1 Overview

Xamarin is a mobile app development framework targeting iOS and Android primarily. It was founded in 2011 (de Icaza, 2011) and sold to Microsoft in 2016, after managing to acquire more than 15.000 customers in 120 countries in its previous five years of existence, including 20% of the Fortune 500 companies and 1.3 million unique developers (Guthrie, 2016).

Figure 1: The Xamarin Products



Source: Own illustration.

Its key feature is to enable developers to share code written in any .NET language across multiple platforms (Petzold, 2016, p. 6). This is not only true for code running on Apple's mobile operating systems and Android, but also on other platforms like Windows, macOS, or Linux. This enables code

sharing between mobile clients and the backend, which can be especially useful for apps that, for example, must work offline.

Its direct competition can be categorized as two-fold. Its two core products, Xamarin.iOS and Xamarin.Android compete with native implementation approaches like writing apps in Swift or Java, its third product Xamarin.Forms is in a fierce competition with React Native and Flutter (Kuitunen, 2019, p. 9; Fayzullaev et al., 2018, p. 9).

Xamarin.Forms provides a comprehensive UI toolkit on top of both platforms. It enables developers to share not only code for business logic and infrastructure tasks, but also the user interface. Which is especially attractive for line of business applications and also for any developer who wants to prototype and test their ideas fast while still delivering a native user experience.

2.2 Technical fundamentals

Xamarin apps are built on top of Mono, ”an open source implementation of Microsoft’s .NET Framework based on the ECMA standards for C# and the Common Language Runtime” (Mono Project, 2019). This means developers can leverage a wide range of functionalities provided by the .NET BCL, use third-party libraries, and share application code through different platforms.

Next to so called ”Shared Projects”, and Portable Class Libraries, which are deprecated, code sharing is achieved through .NET Standard, ”a formal specification of .NET APIs that are intended to be available on all .NET implementation” (Microsoft, 2019).

Xamarin apps are first compiled, like any other .NET application, into Intermediate Language (IL) code (Gough and Gough, 2001). Which means that any language that can be compiled to IL can, in theory, be used to build Xamarin apps. Commonly used right now are C#, F#, and VB.NET – with C# being the lead.

Depending on the target platform, different approaches are then taken based on the IL compilation result. On iOS, a technique called ahead of time compilation is applied to compile the IL code to native ARM assembly code. On Android, the IL code is being compiled just in time (JIT compilation) when the app starts (Microsoft, 2017). Eventually, Xamarin apps can be seen as full native apps, providing access to platform-specific APIs, including native UI elements.

Many also claim that there is no noticeable difference from a user’s perspective in terms of performance. While this may be true for most cases, Willocx, Vossaert, and Naessens, 2015 have shown that there is a small performance penalty involved compared to purely native apps. This penalty increases when Xamarin.Forms is used (AltexSoft, 2017).

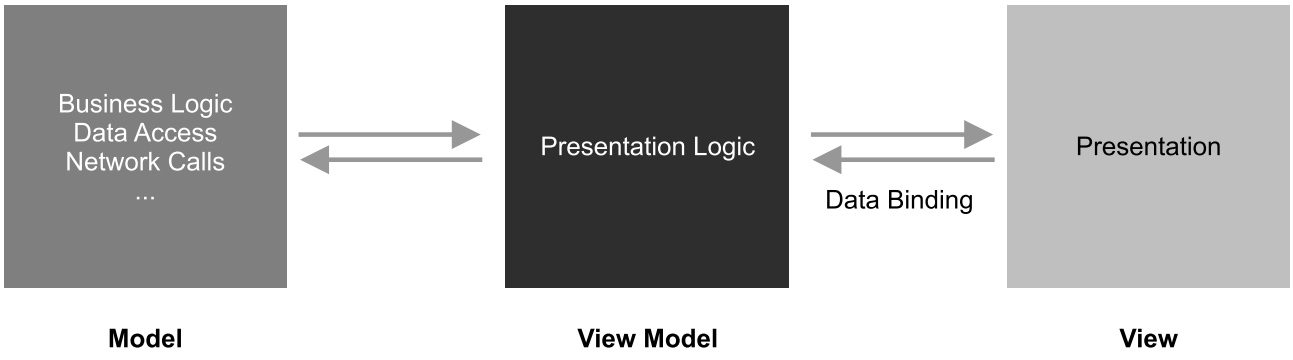
2.3 Architectural styles and patterns

Apps are as of today on nearly all platforms primarily developed in an object-oriented style. This only seems natural as ”the benefits of OOP include encapsulation mechanisms and intuitive ways to model complex domains in software. OOP is a natural fit for GUIs, which probably drove the mainstream adoption of OOP in the 1980s, when GUIs also went mainstream. Once prevalent, OOP also proved broadly applicable.” (Wampler and Clark, 2010).

Closely attached to OOP are five principles that shall enable developers to build understandable, flexible, and maintainable software: ”The SOLID principles are a set of basic principles for designing OO programs. The name itself is an acronym, with each of the five principles named after one of the letters: Single responsibility, Open/ closed, Liskov substitution, Interface segregation, and Dependency inversion. The principles act as a set of guidelines to help you implement code that is easy to maintain and extend over time.” (Warburton, 2016, p. 7).

Apart from those fundamental principles, patterns often arise around specific tasks. At some point, they are then (formally) described and given a name. The original design pattern that many developers apply when building native iOS and Android apps is Model-View-Controller (MVC). It was presumably first described by Krasner, Pope, et al., 1988 and has been the starting point for experiments and the discovery of many derived architectural styles since. A lot of different patterns have been proposed for building mobile applications in particular, e.g., VIPER (Gilbert and Stoll, 2014), Clean Swift (Law, 2015), or MVI (Dorfmann, 2016).

Figure 2: The MVVM Architectural Pattern



Source: Own illustration.

However, the dominating pattern used for building Xamarin apps is Model-View-View Model (MVVM). First mentioned by Gossman, 2005 in the context of Windows desktop development, it made its way to mobile app development later on. Xamarin made the transition especially easy for developers used to technologies like WPF, where MVVM was already successfully applied, by supporting it natively with Xamarin.Forms. But also for Xamarin.iOS and Xamarin.Android developers can chose from a

wide range of MVVM frameworks and libraries like MvvmCross[2], ReactiveUI[3], or Prism[4].

2.4 Ecosystem and tooling

Xamarin apps can be developed on Windows, macOS, and Linux, using IDEs such as Visual Studio, Visual Studio for Mac, or JetBrains Rider. In order to build applications for the Apple platform, tools are required, which are only available on macOS.

Both open-source developers and commercial vendors supply a vibrant ecosystem of third-party libraries aside from the official Xamarin products. Libraries that do not exist on the Xamarin platform itself can be integrated into Xamarin apps through so-called bindings. This is supported for iOS[5] and Android[6] as well.

3 Functional programming

3.1 Overview

Finding a good and commonly accepted universal definition of functional programming is not as easy as it seems to be. Alexander, 2017, p. 39, for example, is referring next to his version to five other variants. Almost every publication on the matter delivers a slightly different interpretation of what FP might be – or not be. However, Hutton, 2002 can help us out here with his take:

”Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language [sic!] are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.” He also notes that ”it is often possible to write functional-style programs in an imperative language, and vice versa. It is then a matter of opinion whether a particular language can be described as functional or not.”

Which language counts as functional and which not is, in fact, part of an ongoing debate among the functional programming community. Some even see languages such as Java and C# in that category, which is fair enough as both got a lot of functional elements such as lambda expressions added to them over the last years. More uncontroversial examples include LISP, Haskell, ML, Scheme, and of course also some more recent candidates such as Elm, Clojure, PureScript, Scala, and F#.

If it is so hard to find a standard definition of the programming paradigm, it is maybe a good idea to

2 https://www.mvvmcross.com/ (retrieved August 8, 2019)
3 https://reactiveui.net/ (retrieved August 8, 2019)
4 https://prismlibrary.github.io/docs/ (retrieved August 8, 2019)
5 https://docs.microsoft.com/en-us/xamarin/ios/platform/binding-objective-c/ (retrieved August 8, 2019)
6 https://docs.microsoft.com/en-us/xamarin/android/platform/native-libraries (retrieved August 8, 2019)

take a look at some core fundamentals that are commonly attributed to the functional programming paradigm in order to get a better understanding of what it defines. The more of those fundamentals a language supports or even favors, the more likely it becomes that developers accept it as a functional programming language. Abraham, 2018, p. 3 concentrates on three main aspects: immutability, expressions, and functions as values.

Immutability

One of the elementary operations of many programming paradigms is to change data. Especially in OOP, it is pervasive to create an object and manipulate its state over the time of its existence. This is fine for many cases but can quickly lead to problems. For example, as soon as concurrency comes into play. When an object is changed from multiple threads, the outcome quickly becomes unpredictable, erroneous, and hard to debug (Bishop, Dilger, et al., 1996). Instead of fighting the symptoms, the idea of immutability offers a solution at the core of the problem. If data cannot be changed but only newly created, the described effects are eliminated. F# records, for example, are immutable by default. This also means one can rely on them always being entirely constructed, as partial construction is impossible.

Listing 1: FP fundamentals: Immutability

```
type User { Name: string }
let user = { Name = "Leon" }
user.Name <- "Amelie" // -> Compiler error!
```

This would not compile, as the name property cannot be changed until explicitly marked as mutable. Also, it is not possible to create a user without providing their name.

Expressions

As the toolset of the world of imperative and object-oriented programming, statements are the primary instruments of choice to manipulate data and define a program's control flow: "The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition." (Microsoft, 2015).

However, as immutable values are preferred over variables in functional programming, some alternative is needed and found in the usage of expressions. Gauld, n.d. notes: "Functional programming is all about expressions. In fact another way to describe FP might be to term it expression oriented programming since in FP everything reduces to an expression".

But what is an expression? In a nutshell: "An expression is a construct in code that produces a value. … An expression can be trivially replaced by a function call." (Microsoft, 2018a).

Listing 2: FP fundamentals: Expressions

```
let add x y = x + y
let z = add 1 2
z = 3 // -> true
z = z // -> true
z = add 1 2 // -> true
add 1 2 = 3 // -> true
```

When passed `x=1` and `y=2`, the `add` function returns the value `3`, which is then bound to `z`. `z` can now be compared to other values or be replaced by another function call.

Functions as values

As Wlaschin, 2018, p. 151 notes, ”in most modern languages functions are first-class objects, but using functions (or lambdas) occasionally doesn't mean that you are 'doing' functional programming. The key thing about the functional programming paradigm is that functions are used *everywhere*, for *everything*.” In order to enable this kind of programming, functions must be able to be treated as data, stored as values, passed as arguments to other functions and returned as the results of expressions (Harrop, 2008, p. 35).

The key concept here is that of the higher-order function: ”Functions that input or output other functions or take functions as parameters” (Wlaschin, 2018, p. 153).

Listing 3: FP fundamentals: Higher-order functions

```
type User = { FirstName: string; LastName: string }

let firstName user = user.FirstName
let fullName user = sprintf "%s %s" user.FirstName user.LastName
let sayHello formatter user = sprintf "Hello, %s!" (formatter user)

let user = { FirstName = "Jane"; LastName = "Doe" }

user |> sayHello firstName // -> Hello, Jane!
user |> sayHello fullName // -> Hello, Jane Doe!
```

In this example, the `sayHello` function returns (the expression of) a greeting. How that greeting is formulated depends on the formatter function that is being passed to it next to the user.

Pure functions

A fourth aspect that is worth to be mentioned is the concept of purity applied to functions. A pure function is a function that does not have any side-effects. This means, that whenever and no matter how often it is called given the same arguments, it will always return the same result (Meijer, 2008). Because it neither relies on external state, nor does it change it.

Listing 4: FP fundamentals: Pure functions

```
let formatDate1 (date:DateTime) = date.ToString("yy.MM.dd") // Pure
let formatDate2 = DateTime.Now.ToString("yy.MM.dd") // Impure!
```

The second function relies on the `DateTime.Now` property, which will always (uncontrollably) change when it is called. This makes it impure.

3.2 Industry adoption

Despite all the prophecies of doom, functional programming can be considered an elementary part of today's IT landscape. Already more than 20 years ago Wadler presented "an angry half-dozen" examples of functional programming use cases in the real world (Wadler, 1997, p. 25). For example, Erlang, which is today seen as one of the fundamental building blocks of applications serving billions of users every day (Reed, 2014). However, despite those examples of successful application, in comparison to other paradigms and especially OOP, functional programming is still not a mainstream paradigm.

Table 1: Programming Language Adoption and Perception

Language	Platform[7]	Type[8]	Used[9]	Loved[10]	Dreaded[11]	Wanted[12]
C#	.NET	OOP	31.9%	67.0%	33.0%	7.0%
F#	.NET	FP	n/a	61.7%	38.3%	3.3%
Java	JVM	OOP	39.2%	53.4%	46.6%	8.3%
Kotlin	JVM	OOP	6.6%	72.6%	27.4%	11.1%
Clojure	JVM	FP	1.5%	68.3%	31.7%	2.2%
Scala	JVM	FP	4.2%	58.3%	41.7%	4.3%
Objective-C	Apple	OOP	5.2%	n/a	68.7%	n/a
Swift	Apple	OOP	6.8%	69.2%	30.8%	5.8%

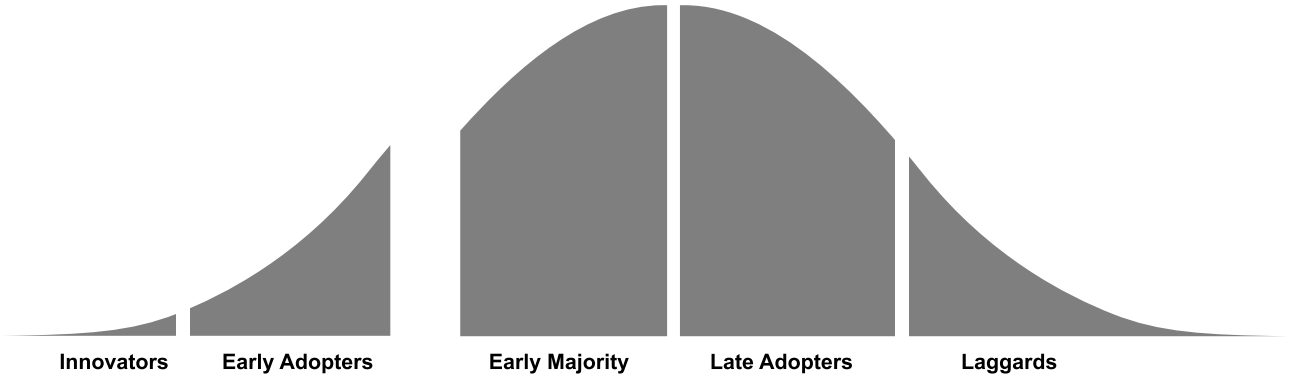
Source: Own illustration based on Stack Overflow, 2019

7 Apple: macOS, iOS, iPadOS, watchOS.
8 Most of the languages listed here are in fact multi-paradigm languages which offer both functional and object-oriented capabilities. The type in this context refers to how those languages are mainly used.
9 By professional developers.
10 Developers want to continue to work with these languages.
11 Developers do not express interest in continuing to work with these languages.
12 Developers do not yet use these languages, but want to learn them.

If one thing gets clear from the numbers presented in table 1, then that traditional and well-established languages like Java and C# still dominate their respective platforms. The exception here is Swift. Considering its young age (it was published in 2014), this seems surprising at first. However, looking at how discontent programmers are with Objective-C, not so much anymore. Not even the immaturity of the tooling around the language like its IDE Xcode (Rebouças et al., 2016) could stop the migration.

So why are most of the young contenders stuck with little recognition in their niches, while Swift is taking off so quickly? Sink, 2015 predicted Swift's success precisely while writing about why F# is not making ground against C#. He based his explanation on the theory of "The Chasm" by Geoffrey A. Moore, first published in the book "Crossing the Chasm" (Moore, 1991).

Figure 3: The Revised Technology Adaption Life Cycle



Source: Own illustration based on Moore, 1991, p. 13.

The chasm describes the gap of adoption between the group of early adopters and the majority of users. While early adopters may accept some problems on the way (like immature tooling), the majority expects a definite productivity improvement without disrupting the way they work (evolution instead of revolution). Sink concludes that it needs severe problems to tackle and massive discontent with existing solutions in order to accept new approaches by the majority, which for those approaches then means "to cross the chasm."

For developers building applications in the Apple ecosystem, the primary hard problem seems to be Objective-C. In other words, "the pain" is so significant that developers were almost desperately waiting for an alternative language and are now happily joining the movement of the Swift language.

How about the JVM and .NET? Java did evolve relatively slowly over the last years compared to C#, which early on got features like Generics (Kennedy and Syme, 2001), LINQ (Meijer, Beckman, and Bierman, 2006), and async/await (Hejlsberg, 2010). That may have made some room for improvement brought by new languages like Clojure and Scala, and especially Kotlin. Also, as seen in table 1, all three Java contenders are more appreciated than Java is itself within the JVM community. Interestingly, the opposite seems to be the case for .NET: F# programmers love their language, but so do C# programmers. Which may explain the low adoption rate of F# compared to C#.

What could be done to make functional programming attractive for a broader audience? Wadler, 1998, p. 23 lists several factors that a functional language must support in order to attract people to adopt it on a larger scale: ”To be widely used, a language should support interlanguage working, possess extensive libraries, be highly portable, have a stable and easy to install implementation, come with debuggers and profilers, be accompanied by training courses, and have a good track record on previous projects.” The first five of those factors, which are mainly technical, are provided by almost all of the functional languages.

Training, as the sixth factor, is indeed a problem not to be underestimated. As Hinsen, 2009, p. 86 notes, ”functional programming is very different from traditional programming … and thus requires a lot of learning and unlearning.” However, as some of the functional languages and architectures become more and more widespread, a lot of learning materials are being published, making it easier today to dig into the matter than ever before. Also, new teaching methods are tested which try to teach FP to developers used to other programming paradigms, especially OOP (Petricek, 2012).

A good track record, the seventh factor, however, is probably the hardest part and the missing piece. Wadler, 1998, p. 26 calls it the need for a ”killer app”: ”Experience shows that users will be drawn to a language if it lets them conveniently do something that otherwise is difficult to achieve. Like other new technologies, functional languages must seek their killer app.”

Wampler and Clark, 2010 make a case for FP to be the key to solve hard problems occurring in concurrent computing as with its concepts of immutability and functions free of side effects. That, for example, may have helped Scala to gain traction as Akka, ”a toolkit for building highly concurrent, distributed, and resilient message-driven applications” is written in it (Lightbend, 2019).

On an even larger scale, functional programming can benefit from the latest trends in cloud computing towards serverless archictectures (Lynn, Rosati, Lejeune, and Emeakaroha, 2017). As Leitner, Wittern, Spillner, and Hummer, 2019 note, ”building Serverless and FaaS applications requires a different mental model that emphasizes ’plugging together’ small microservices. … Adopting this differentmental model may be different, but experience with functional programming and the immutable infrastructure paradigm helps.”

In the field of mobile app development, Kotlin is since 2017 officially supported by Google for its Android platform (Cleron, 2017), which may have lead to a significant boost in recognition and eventually, acceptance. Kotlin may, as of today, primarily be used in a more traditional imperative way. However, it also supports functional programming, which could sooner or later become attractive for many Android developers. Especially since functional programming with Swift starts to spread on the competing Apple platform, e.g., through the new development tool SwiftUI[13].

[13] https://developer.apple.com/xcode/swiftui/ (retrieved August 8, 2019)

3.3 F# – A functional-first .NET language

3.3.1 Overview

F# is a strongly-typed so-called general purpose programming language running on the .NET platform. Unless other popular .NET languages such as C# and VB.NET, it favors a functional-first programming style (F# Software Foundation, n.d.).

It has been developed in response to the success of Java ”through Project 7, the initial effort to bring multiple languages to .NET, leading to the initiation of .NET Generics in 1998 and F# in 2002. F# was one of several responses by advocates of strongly-typed functional programming to the ’object-oriented tidal wave’ of the mid-1990s …” (Syme, 2018).

Published first in 2005, it is today maintained by the F# Software Foundation, Microsoft, and individual contributors. It is licensed under the MIT License[14].

During its ongoing development, the language has been heavily inspired by OCaml, Haskell, Python, and C# (Syme, 2010). The latter may surprise at first, but in order to stay compatible with code written in other languages of the .NET platform, F# supports next to functional also object-oriented and imperative programming models (Microsoft, 2016a).

3.3.2 Language features

It would go beyond the scope of this thesis to describe all the language features of F# in depth. This chapter instead focuses on those parts that provide its unique character and set it apart from C#.

Algebraic data types

The probably most fundamental building-block of FP with F# are algebraic data types, among others especially records and discriminated unions.

A record is the product of all of its (named) values. It can be extended by members if needed.

Listing 5: F# Record

```
type User =
    { FirstName: string
      LastName: string }
```

Whereas a discriminated union represents the sum of different cases. It can be seen as an enumeration which can optionally take a payload for each case. This opens a variety of possibilities, and it makes it especially easy to make invalid states unrepresentable.

[14] https://github.com/dotnet/fsharp/blob/master/License.txt (retrieved August 8, 2019)

Listing 6: F# Discriminated union

```
type Session =
    | Anonymous
    | Authenticated of user:User
```

It also allows the definition of so-called single case discriminated unions which can be used to efficiently work with custom types where otherwise simple types would be necessary and prone to errors.

Listing 7: F# Single case discriminated unions

```
type UserId = UserId of int
type SessionId = SessionId of int

let signIn (userId:UserId) (sessionId:SessionId) =
    ()
```

The `signIn` function could also take two integers as parameters. However, by using single case discriminated unions, the compiler can ensure the correct values are passed. When only using integers, the `UserId` could easily be mixed up with the `SessionId` and vice-versa. This is now impossible.

Type inference

As already mentioned, F# is a statically-typed language. The F# compiler is for many cases able to deduce the correct types on its own, based on the usage of a construct.

Listing 8: F# Type inference

```
let sayHello name = // string -> string
    sprintf "Hello, %s!" name
```

Furthermore, it supports a feature called automatic generalization. If the type of a parameter cannot be deduced by its usage inside of its function, the parameter becomes generic.

Listing 9: F# Automatic generalization

```
let getLast set = // seq<'a> -> 'a
    set |> Seq.last

[ 1; 2 ] |> getLast // int
[ 1.0; 2.0 ] |> getLast // float
```

Immutability

Immutability is another fundamental building-block. Accordingly, most types in F# are immutable by default unless explicitly marked otherwise.

Listing 10: F# Immutability by default

```
let firstName = "Jane" // Can never be changed
let mutable lastName = "Doe" // Can be changed
```

However, obviously, at some point, changes of information need to be reflected. F# supports the ”copy and update record expression,” which enables a developer to do exactly what it says in its name.

Listing 11: F# Copy and update record expression

```
let jane = { FirstName = "Jane"; LastName = "Doe" }
printfn "%s %s" jane.FirstName jane.LastName // "Jane Doe"

let janet = { jane with FirstName = "Janet" } // Copy & Update
printfn "%s %s" janet.FirstName janet.LastName // Janet Doe
```

Structural equality

Another thing that is helpful in many situations is to be able to compare two objects with each other by looking at their shape – or structure – instead of their memory address. While this is easy in any language for value types such as two integers, F# also enables this for most more complex types such as records out of the box.

Listing 12: F# Structural equality

```
let user1 = { FirstName = "John"; LastName = "Doe" }
let user2 = { FirstName = "John"; LastName = "Doe" }

user1 = user2 // true
```

Pattern matching

Pattern matching is a powerful technique that works with a wide range of data types, and especially well with discriminated unions: ”Patterns are rules for transforming input data. They are used throughout the F# language to compare data with a logical structure or structures, decompose data into constituent parts, or extract information from data in various ways.” (Microsoft, 2016b).

Listing 13: F# Pattern matching

```
let greeting session =
    match session with
    | Anonymous -> "Hello!"
    | Authenticated user -> sprintf "Hello, %s!" user.FirstName

greeting Anonymous // Hello!
greeting (Authenticated(jane)) // Hello, Jane!
```

One additional benefit is that pattern matching in F# is always exhaustive. That means that as soon as a case is left out, the compiler will respond with a warning. This way not only impossible or redundant cases are being caught, but also missing ones.

Optional types

There have been many decisions with a significant impact on the history of software development. The probably most expensive one was the introduction of null references (Hoare, 2009). Working with null would be extraordinarily cumbersome in a language that heavily uses expressions over statements. To be able to represent missing data, F# uses optional types instead. They allow a callee to explicitly express the absence of information and the caller to handle the situation accordingly.

Listing 14: F# Optional types

```
let tryGetAuthenticatedUser session = // Session -> User option
    match session with
    | Authenticated user -> Some(user)
    | Anonymous -> None

let user = tryGetAuthenticatedUser Anonymous
match user with // Hey, anonymous!
| Some user -> sprintf "Hey, %s!" user.FirstName
| None -> sprintf "Hey, anonymous!"
```

Pipe operator

The pipe operator, or more precisely the pipe-forward operator, is heavily used in F#. It helps ”piping together” function calls, by passing the result of one function onto the next. As every function returns a value, this enables an elegant way of expressing data flowing through the program.

Listing 15: F# Piping

```
[ 1..9 ]
|> List.filter (fun x -> x > 5)
|> List.filter (fun x -> x % 2 = 0)
|> List.map (fun x -> x * 2)
|> List.sum // -> 28
```

As Syme, 2018 notes, ”the use of the pipeline symbol is particularly important in F# because type-inference is propagated left-to-right and name resolution occurs based on information available earlier in the program.” Although it is attributed to F# where it was introduced in 2003, it dates back to 1994 when it was originated by Tobias Nipkow (Syme, 2011).

Pure functions

Like other programming languages, F# supports side-effect-free pure functions. However, unlike, for

example, Haskell with its I/O system[15], the F# compiler does not provide any means to enforce purity (Seemann, 2018). Which, unfortunately, leaves it to the developer to ensure that a function does not unwillingly produce side-effects. On the other hand, this may be helpful in a phase of transitioning existing programming knowledge from the imperative to the functional world, as less restrictiveness at this point lowers the entry-barrier.

3.3.3 Ecosystem and tooling

Supported by a wide range of different IDEs[16] and editors[17], developers can build and run applications written in F# today not only on Windows but also on macOS and Linux. The tooling is not as sophisticated, for example, as for C#, but it supports standard features such as syntax highlighting, code completion, or renaming of symbols, to name a few. Also, F# code bases can leverage advanced IDE features such as graphical interfaces for automated test execution.

F# can also be used as a scripting language through F# Interactive (Bandt, 2017). The REPL[18] allows developers to try things out quickly. So to test a couple of lines of code, this way, it is not necessary to set up and compile a full project.

As part of .NET, F# applications can make use of the full capacity of the .NET BCL. Also, the F# Core Library provides a set of functions, collection classes, control constructs for asynchronous programming, and more utilities to support a functional-first development experience.

Besides those core features, it is not only possible to consume C# code from within the same solution, but also to reference any .NET package as provided mainly through NuGet, the package manager for .NET[19]. For the latter, it does not matter in which language a specific package has been written. So F# code could even use a library built with VB.NET.

Unfortunately, most of the BCL features and even most of the packages in the .NET world do not feature functional thinking. This inspired the F# community over the years to build their own functional solutions for a wide range of fields. Some of them, like the build tool FAKE[20] and the alternative dependency manager Paket[21], are today even used by a broader audience beyond the ”inner F# ecosystem.” Whereas others, like type providers[22] or the F# to JavaScript compiler Fable[23], are more or less unique to the F# ecosystem.

15 https://www.haskell.org/tutorial/io.html (retrieved August 8, 2019)
16 Examples: Visual Studio, Visual Studio for Mac, JetBrains Rider.
17 Examples: Visual Studio Code, Vim, Emacs, Sublime.
18 Read–eval–print loop.
19 https://www.nuget.org/ (retrieved August 8, 2019)
20 https://fake.build/ (retrieved August 8, 2019)
21 https://fsprojects.github.io/Paket/ (retrieved August 8, 2019)
22 https://fsharp.github.io/FSharp.Data/ (retrieved August 8, 2019)
23 https://fable.io/ (retrieved August 8, 2019)

Fable is widely seen as the biggest success of the F# ecosystem so far. Some even consider F# to be a language with two main compilation targets: .NET and JavaScript.

F# can also be used in one of the latest industry trends, server-less computing. It is, for example, natively supported by Azure Functions and AWS Lamda.

3.4 Functional programming in C#

While F# is a functional language that can be used in an imperative and object-oriented style, C# is often claimed to work the other way around: As an imperative and object-oriented language, that can also be used functionally. Both F# and C# had some influence on each other over the years. While the work on generics (Kennedy and Syme, 2001) lead to the creation of F#, things like ”var” (C# 3), ”async/await” (C# 5), tuples, pattern matching (C# 7), and non-null pointers (C# 8) have been heavily influenced by F# (Syme, 2018, p. 33). C# developers are also used to other functional techniques, especially LINQ with its heavy usage of lambda expressions (Bierman, Meijer, and Torgersen, 2007; Meijer, 2011, p. 6), and extension methods (Biczó, Pócza, and Porkoláb, 2009).

C# supports various functional aspects so that it may be worth to take a glance at the previously introduced core fundamentals of functional programming from its perspective.

Immutability

The .NET BCL offers a set of immutable collections (Brais, 2017). Beyond that, most things are mutable by default in C#, except for simple value types and strings. While it is possible to design, for example, classes to have read-only members only, a built-in copy and update mechanism is missing.

Listing 16: C# Immutability

```
public class User
{
    public string FirstName { get; }
    public string LastName { get; }

    public User(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public User WithFirstName(string firstName) => new User(firstName, LastName);
    public User WithLastName(string lastName) => new User(FirstName, lastName);
}

var user = new User("Jane", "Doe"); // Jane Doe
user = user.WithFirstName("Janet"); // Janet Doe
```

The sample shows how to implement copy and update manually. However, this will only last until

a type is used that is not under control by the developer. Because for third-party types immutability cannot be ensured. But there are, in fact, alternative approaches. Sturm, 2011, p. 173, for example, proposes a technique he calls automatic cloning. It is also possible to serialize an object and to deserialize it to retrieve a copy and avoid modifying the original.

Expressions

Compared to F#, where everything is an expression, the case for C# is fundamentally different. Usually, a lot of statements dominate the picture, leading to an imperative style applied by most developers. However, it is possible to use expressions over statements for a lot of cases in C#, too.

Listing 17: C# Expressions

```
int Add(int x, int y) => x + y;
int z = Add(1, 2);

bool equals1 = z == 3; // -> true
bool equals2 = z == z; // -> true
bool equals3 = z == Add(1, 2); // -> true
bool equals4 = Add(1, 2) == z; // -> true
```

Functions as values

The sample presented in listing 4 can be re-written in C# as follows:

Listing 18: C# Higher-order functions

```
string FirstName(User u) => u.FirstName;
string FullName(User u) => $"{u.FirstName} {u.LastName}";
string SayHello(Func<User, string> formatter, User u)
    => $"Hello, {formatter(u)}!";

var user = new User("Jane", "Doe");
SayHello(FirstName, user); // Hello, Jane!
SayHello(FullName, user); // Hello, Jane Doe!
```

Which shows that functions are first-class citizens in C# as well. It is possible to pass along static functions, class members, or even local functions.

Pure functions

In terms of purity, C# offers the same (dis)comfort as F#. Pure functions can be used, but there is no way to actually enforce purity on them.

Summary

As shown, C# supports some important functional concepts. In direct comparison to F#, however, it suffers, from some limitations at type inference, the absence of data types like discriminated unions,

and especially much boilerplate code that has to be written in order to achieve immutability and to enable structural equality comparisons. Buonanno, 2018, p. 216 even suggests C# developers to consider to define their types in F# rather than C# in order to get rid of those limitations.

4 Functional programming in F# for Xamarin

4.1 Substituting one language by another

As outlined before, Xamarin apps can, in theory, be written in any language that compiles to IL and is therefore supported by the .NET platform. F# is one of those languages, and it is possible to use it as a replacement of the de-facto standard language for Xamarin, C#. This is shown, for example, by Petzold, 2015 and Shackles, 2017.

While a developer can benefit from the strengths of F#, some obvious obstacles might let feel using the language in this context counter-intuitive especially to functional programmers.

First of all, Xamarin works on top of highly object-oriented APIs of iOS and Android, and Xamarin.Forms is designed in an object-oriented way, too. In order to implement, for example, native view controllers (iOS), activities (Android), or pages (Xamarin.Forms), it is necessary to use OOP concepts such as inheritance. Due to the nature of object-oriented APIs, much state-mutation is necessary in order to make things work. When configuring the properties of an UILabel control on iOS, for example, this is done through setting its properties and therefore mutating the label's state.

Listing 19: A standard iOS view controller written in F#

```
[<Register("ViewController")>]
type ViewController(handle : IntPtr) =
    inherit UIViewController(handle)

override x.ViewDidLoad() =
        base.ViewDidLoad()
        let label = new UILabel(x.View.Frame)
        label.BackgroundColor <- UIColor.Yellow // Mutation
        label.Text <- "Hello World" // Mutation
        x.View.Add label
        ()
```

Those obstacles become even more evident when the app is built with an object-oriented architecture style like MVVM. In order to make bindings between view and view model possible, properties of the view model are being updated, and their state is being changed frequently.

All in all, it can be stated that F# can be used to replace C#, but the benefits most likely will not outweigh the disadvantages.

4.2 Taking advantage of both worlds

Instead of using a functional-first language for mainly object-oriented tasks, an alternative approach is to use both paradigms and languages side-by-side. Everything related to UI, including views and view models, can still be written in C#, leveraging its natural object-oriented capabilities in conjunction with the object-oriented frameworks the mobile app environment provides. However, everything ”below,” in particular the model part of MVVM, can be written in F# in a functional style.

Bandt, 2018, for example, presents an implementation of a local Redux store[24] written in F#, which sits at the core of a C# Xamarin application. F# is favored for this specific task over C# because of its built-in features regarding immutability, structural equality, and its discriminated union type. Features that are not available through C# or that would come with additional costs attached. However, the sole consumer of the store's features is the C# mobile app, which is built ”around” it. That is possible because, as lined out before, everything in .NET compiles eventually to IL code. So, F# and C# projects can be used side-by-side within a single ”hybrid” solution.

The F# compiler has some characteristics that make it even more convenient to use F# code from within C#. As Petricek and Skeet, 2009, p. 255 mention, ”classes or records with members, … appear as standard C# classes and can be used without any trouble.”

Listing 20: Defining a Record type in F#

```
type Article =
    { Title: string
      Author: string }
```

The F# Article record can be constructed in C# as any other C# class.

Listing 21: Constructing an F# Record type in C#

```
var article = new Article("Hello World", "Jane Doe");
```

It is worth noting that the characteristics of an F# record are preserved even when used from C#. All of its properties need to be provided during construction; afterwards, they are immutable. On top of that, structural equality is provided, as well.

One of the most powerful features of F# that C# is still lacking is the support of discriminated unions. However, discriminated unions ”are nothing but a bunch of classes generated by the F# compiler” (Núñez and Fahad, 2016, p. 104).

[24] https://redux.js.org/ (retrieved August 8, 2019)

Listing 22: Defining a Discriminated Union type in F#

```
type ArticleType =
    | Editorial
    | Column of columnist: string
    | Essay
```

For the payload case, the compiler creates a factory method that helps to create the object in C#.

Listing 23: Constructing an F# Discriminated Union type in C#

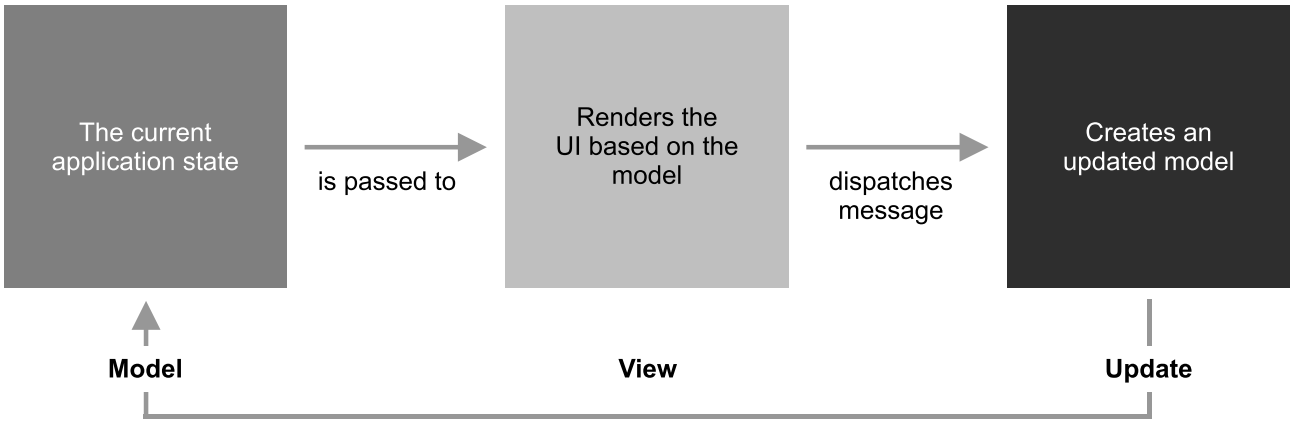
```
var column = ArticleType.NewColumn("Jane Doe");
```

After all, it is not only technically feasible to use F# and C# side-by-side, but this approach also offers an opportunity to introduce functional programming with F# for many software development teams in the first place. Existing object-oriented knowledge, concepts, and code artifacts can continue to be used while solving specific problems with functional programming at the same time.

4.3 Leveraging a new architectural pattern

Czaplicki, 2012 presented Elm, a new programming language that focuses on building purely functional graphical user interfaces. The language has evolved since then and managed to build an active community of users who primarily build web applications with it. While the community of Elm developers was growing, and more and more applications were developed with it, a specific pattern has been discovered. What today is widely called The Elm Architecture, or MVU for Model-View-Update outside of the Elm ecosystem, ”seems to emerge naturally in Elm. Rather than someone ’inventing’ it, early Elm programmers kept discovering the same basic patterns in their code.” (Czaplicki, 2018).

Figure 4: The MVU Architectural Pattern



Source: Own illustration.

MVU has since been part of a movement towards architectures supporting unidirectional dataflows for user interfaces (Staltz, 2015).

Unlike other architectural patterns like MVVM, ”the Model” in MVU does not stand for an unspecified set of services and utilities, but for a very specific data structure that contains the (whole) state of the application. This data structure is immutable.

For rendering the view, the model is passed to a view function that returns the UI based on that exact model. This function is pure, which makes it possible to unit-test it — something hard to achieve or even impossible in most alternative solutions and especially for XAML UIs.

Of course, users want to interact with the interface, so it is not static and needs to react to input. That is done through commands, which eventually dispatch messages which are then being processed by an update function.

Update functions are pure, too. They take in the current model and a message and return a modified (updated) copy of the model. The update is being performed based on the message. Whenever a message is being dispatched, and therefore a new model is being created by the update function, the view is being re-rendered.

All of this ensures that data flows only in one direction through the whole application. This makes it very easy to reason about the program, it makes its components testable, and it enables features like time-travel debugging (James, 2014). As Loder, 2018, p. 121 notes, this is made possible through much wiring that is automatically being done in the background: ”This makes it a little bit difficult at first to understand what is going on, but once the concept is clear we see that it reduces code in our application significantly.”

Introducing Fabulous

Thanks to Fable, JavaScript has been an attractive compilation target for F# developers for many years. Consequently, in 2016, Eugene Tolmachev announced the first version of what he called Elmish: an implementation of MVU for Fable (Tolmachev, 2016).

In 2018, Don Syme was working together with the Xamarin team in his role as a researcher at Microsoft Research, trying to find out if there was a way to make app development with Xamarin as compelling to F# developers as it was to build web applications (Syme and Bennett, 2018).

The existing solutions did not convince Syme: XAML seemed complicated and even unnecessary to him, and all the MVVM approaches were based on mutable data. Inspired by Fable and Elmish, Syme focused his research on a solution that could deliver a developer experience that was comparably easy and functional-first. He eventually came up with a library called Fabulous[25], an open-source project

[25] https://fsprojects.github.io/Fabulous/ (retrieved August 8, 2019)

which is not affiliated to Microsoft but developed by its community of volunteers. Fabulous allows building mobile applications in a functional way on top of Xamarin.Forms by offering two different flavors: ”Full Elmish” and ”Half Elmish.”

Full Elmish

When choosing Full Elmish, the entire Xamarin application can be written in F# following the original idea of The Elm Architecture:

Listing 24: A Fabulous sample program

```
type Model =  { Count : int }
type Msg = | Increment | Decrement

let init () = { Count = 0; }, Cmd.none

let update msg model =
    match msg with
    | Increment -> { model with Count = model.Count + 1 }, Cmd.none
    | Decrement -> { model with Count = model.Count - 1 }, Cmd.none

let view (model: Model) dispatch =
    View.ContentPage(content =
        View.StackLayout(children = [
            View.Label(text = sprintf "Current Count: %d" model.Count)
            View.Button(
                text = "Increment",
                command = (fun () -> dispatch Increment))
            View.Button(
                text = "Decrement",
                command = (fun () -> dispatch Decrement))]))
```

The sample shows an (almost) complete Fabulous program, including all essential parts: model, messages, and the three functions to initialize and update the model and to render the view.

The model, in this case, contains just a simple count property which is initialized to 0. There are precisely two operations that can be done with this app: depending on the intent, which is expressed by a message, a new model is being created through the update function, with the count property either increased by one or decreased by one.

The view function takes in the model and a dispatch function. This allows creating a hierarchy of view elements depending on the model. In this simple case, the model’s count value is rendered as a label. By using the dispatch function, updates can be triggered when the user interacts with the app through pressing either the increment or the decrement button.

Fabulous supports almost all Xamarin.Forms elements, which can be used in its terse DSL to express the view in a very similar hierarchical way as it would be done traditionally through XAML. The key difference is that there are no bindings that react to changes inside of the view.

However, the view is being re-evaluated as soon as the model changes. To provide adequate rendering performance, the evaluation of views is handled by Fabulous in a way that allows a developer to specify custom differential updates for specific scenarios. So not the whole (complex) view is being re-rendered all the time but only those parts that need to reflect a change (Syme and Bennett, 2018).

As defining UI in code quickly becomes a tedious process when every change would require a complete compilation and deployment cycle, Fabulous offers a mechanism it calls ”Live Update”[26]. Live Update will send changes made to the code to the device or simulator/emulator, where the code is then evaluated, and the app is being refreshed immediately.

What developers get for free with Fabulous is the ability to unit-test most of the critical parts of an app, as those parts are implemented as pure functions that do not have any side-effects. The view, for example, can be tested by passing a mocked model to it. This is a unique advantage of the architecture and the Fabulous library, something hardly possible with XAML.

Some operations need to involve side-effects, like loading data from a database, or making network requests. Those operations are implemented through commands. Both the `init` and the `update` function return a tuple containing the model and a command. If there is something else returned than `Cmd.none`, that command is being executed by Fabulous. The command itself is implemented as a function that can take in any parameter and almost always returns a new message which will then be passed to the update function again. This way, the – unidirectional – message loop stays intact.

Half Elmish

Developers with a background in XAML and C# may be reluctant to choose the Full Elmish approach, especially because of the view part. Half Elmish[27] offers a compromise: Views can still be created (or re-used from existing solutions) in XAML, while model and update function are being written in F#, next to a view function that wires up the bindings (Bennett, 2018). This way, existing knowledge and code can be transferred, and the transition from XAML + C# to an application entirely written in F# is being made more convenient.

5 Comparison setup

5.1 Scope and complexity

While Xamarin offers a variety of different paths to implement mobile apps, comparing all of them would go way beyond the scope of this thesis. Therefore Xamarin.Forms is chosen. It is not only prevalent amongst Xamarin developers, but it is also the foundation of the Fabulous library.

[26] https://fsprojects.github.io/Fabulous/Fabulous.XamarinForms/tools.html (retrieved August 8, 2019)
[27] https://fsprojects.github.io/Fabulous/Fabulous.StaticView/ (retrieved August 8, 2019)

A case study will be conducted by building a small sample application, IUBH TOR. That app will be implemented twice: in C# with an object-oriented MVVM architecture, and in F# with a functional MVU architecture. Both implementations will address the same problem to provide feature-parity results in terms of the functional requirements introduced later on.

It is not possible to cover every conceivable technical scenario. Instead, the focus is on several features that are at the core of most modern mobile applications: managing a persistent user session; exchanging data with a backend; transforming data and persisting it permanently (e.g., in a local database); displaying data in a list; displaying data on a detail dialog; navigation between dialogs; periodic, scheduled, background work; delivering messages to the user through the platform's notification system. Design (UI) and usability (UX) are not in focus during the case study.

5.2 Functional requirements

In order to build an application that covers all of the technical features listed in the previous section, a practical case related to the IUBH has been chosen.

Students enrolled to distance learning courses at IUBH have access to a website within the university's ”CARE portal” that allows them to get an overview of the current state of booked modules and the grades they received – their transcript of records. However, at the time of writing, there is no notification system in place that would let them know about a new grade or any other change to their transcript of records, which leads for thousands of students to the tedious task of continuously logging in to the portal and navigating to that website, often multiple times a day.

The sample app built for this thesis, IUBH TOR, is addressing this problem. Its detailed specification, including visual mockups of its user interface, can be found in Appendix A.

5.3 Methodology

The comparison of programming languages of the same type alone would be a challenging task to tackle. Comparing different languages that propagate different programming paradigms is even more challenging. Especially when there is little agreement on what those paradigms precisely characterize (as seen previously in regards to functional programming).

Also, it is questionable what should be compared: The size of the final executables? Memory footprint? The number of assemblies, namespaces, or types? Those quantitative measurements, which provide a rather narrow technical glimpse towards the result, do not seem to be exceptionally insightful in the context of this thesis.

Software Quality Models

Another angle is taken by sophisticated software quality models, which exist in various forms like Boehm (Boehm, Brown, and Lipow, 1976), McCall (McCall, Richards, and Walters, 1977), FURPS (Grady, 1994), or ISO/IEC 25010:2011 (International Organization for Standardization, 2011), just to name a few. All of those are trying to define characteristics important to software quality.

Deissenboeck, Juergens, Lochmann, and Wagner, 2009 propose to classify such quality models by purpose: ”Although definition, assessment and prediction of quality are different purposes, they are obviously not independent of each other: It is hard to assess quality without knowing what it actually constitutes and equally hard to predict quality without knowing how to assess it.” They also note that the ideal model, which would cover the definition, assessment, and prediction, does not exist (yet): ”Existing quality models do not necessarily cover all aspects equally well. The ISO 9126, for example, defines quality but gives no hints for assessing it …”.

In particular, ISO/IEC 25010:2011 seems to be a reasonable choice, for example, to define quality criteria for software to be purchased (e.g., Chua and Dyson, 2004). But it does not seem to be a good choice to evaluate two competing programming paradigms in the context of the case study of this thesis. Especially under the aspect that requirements, implementation, and evaluation must be carried out by the same person - the author of this thesis. Instead, it seems more appropriate to examine the development approach itself from different angles.

Cognitive Dimensions of Notations

A method that meets this requirement is provided by the ”Cognitive Dimensions of Notations” (CDs) framework. It does not enforce an exclusive look at the product or the formal process, but instead takes the perspective of human-computer interaction (HCI) and therefore allows a holistic look at the two programming approaches at hand:

”’Cognitive dimensions’ are features of computer languages considered purely as information structures or notations. They therefore apply to many types of language—interactive or programming, high or low level, procedural or declarative, special purpose or general purpose. They are ’cognitive’ dimensions because they control how (or whether) the preferred cognitive strategy for design-like tasks can be adopted …” (T. R. Green, 1989)

Initially introduced by T. RG Green in 1989 with the first five dimensions, T. R. G. Green and Petre, 1996 proposed an extended set of 14 dimensions, which was detailed further by Blackwell and Green, 2003. In their systematic literature review on the matter, Hadhrawi, Blackwell, and Church, 2017 found 1.638 unique articles citing the CDs framework.

In the context of programming languages, methodologies, and tools the CDs framework has, for

example, been used to evaluate parallel programming features of programming languages (Sadowski and Kurniawan, 2011), to evaluate the usability of a class library (Clarke and Becker, 2003), and to compare object-oriented and functional programming for GUI programming (Kiss, 2014).

What seems to make the CDs framework particularly suitable for the task of this thesis is that its dimensions are not strict guidelines but discussion tools. Non-specialists can use those tools in HCI because they avoid ”death by detail.” The framework further works task-specific, concentrating on the processes and activities rather than the final product (Blackwell and Green, 2003).

Nine of the ”official” 14 dimensions are chosen to evaluate the two IUBH TOR implementations: abstraction, visibility, hidden dependencies, hard mental operations, closeness of mapping, error-proneness, diffuseness, secondary notation, and progressive evaluation (see the following sections for a detailed explanation of each of those dimensions).

In addition, as the 10th dimension, ”accessibility” is being newly introduced for the sake of the particular comparison subject to this thesis. It covers aspects of ”getting started” with the development approaches and takes a look at the maturity of tooling and ecosystem. Topics that are essential for deciding on whether an investment in functional programming for mobile app development may be justified or not.

6 Results

6.1 Overview

Both implementations of IUBH TOR have been completed, each supporting the full set of features as specified in Appendix A. The source code is available on GitHub[28].

In both the C# and the F# implementation for iOS and Android a student would be able to authenticate against the CARE system, see their full transcript of records including all details of all courses, and get automatically notified through system notifications as soon as new information is being detected. If it was a real-world project, either version of the app could be shipped to actual students through Apple’s AppStore for iOS and Google Play for Android. Screenshots of the app can be seen in Appendix B.

During the time of the implementation, some ”environment variables” changed, as it happens in real-world projects regularly, too. For example, the structure of the transcript of records HTML document, which is used as the data source for the app’s module list, has been changed without notice by the CARE developers. Through the high coverage of automated tests for both app versions, it was possible to identify the conflicting changes through failing tests and to fix it eventually. Also, the authors of the Fabulous library, which was used for the F# version, made the unexpected decision to

[28] https://github.com/aspnetde/IUBH.TOR (retrieved August 8, 2019)

restructure the library[29] which made some subsequent changes inevitable in order to keep the app technically up to date. All of those necessary updates could be accomplished in a relatively short time.

While successfully fulfilling the functional requirements, all of the previously defined technical scenarios could have been covered, too. The following table provides an overview of those scenarios and the particular class/module which covers the implementation.

Table 2: IUBH TOR: Technical Scenarios

Scenario	C# Implementation	F# Implementation
Managing a persistent user session	CredentialStorage	Authentication
Exchanging data with a backend	CredentialValidator, CoursePageHtmlDownloader	Authentication, CoursePageHtmlDownloader
Transforming data	CoursePageHtmlParser	CoursePageHtmlParser
Persisting data permanently	CourseUpdater	App (global model)
Displaying data in a list	CourseListPage	CourseListPage
Displaying data on a detail dialog	CourseDetailPage	CourseDetailPage
Navigation between dialogs	CourseListPage → CourseDetailPage	CourseListPage → CourseDetailPage
Periodic background work + Notifications	AppDelegate.PerformFetch(), DroidBackgroundSyncJob	AppDelegate.PerformFetch(), DroidBackgroundSyncJob

Source: Own illustration

When building a more complex application in C#, it is usually recommended to consider using one of the common MVVM frameworks. For IUBH TOR, however, it seemed unnecessarily complicated and unreasonable, as only three different dialogs were required. Therefore view models have been implemented manually, and no framework has been used.

As shown in the screenshots of both applications in Appendix B, the results look almost identical. Both apps use the Material Design visual system provided by Xamarin.Forms and both apps could make use of the same UI structure – just expressed technically differently as described in the following sections.

6.2 Evaluation dimensions

6.2.1 Abstraction

For Jackson, 2006 ”abstractions are the essence of software development.” He defines it as conceptual structures that are either discovered in the problem domain (e.g., a typeface family), invented by

[29] https://github.com/fsprojects/Fabulous/pull/481 (retrieved August 8, 2019)

the designer (e.g., a drawing layer), or something in between (e.g., a spreadsheet). He further states that good software uses robust and flexible abstractions that provide a clear model for the user and clean interfaces for developers.

Questions

- Does the notation add any new layer of abstraction on top of existing abstractions?
- How easy or difficult are the abstractions provided by the notation to work with?

Evaluation

In the domain of cross-platform mobile app development, abstractions are ubiquitous. The most crucial goal of tools like Xamarin is to minimize development efforts by enabling developers to share functionality across different platforms.

Especially when it comes to code-sharing for building user interfaces, the notation offered to developers can only consist of the lowest common denominator. That naturally leads to new concepts that live on top of the native target platforms, e.g., a Xamarin.Forms entry element[30] that gets rendered as an UITextField[31] on iOS and an EditText element on Android[32].

Xamarin.Forms itself has already introduced a considerable complex set of techniques, like custom renderers and the visual system. In addition, its layout engine contains some hand-made implementations that are relatively hard to maintain, as Jason Smith, one of the original authors, stated: ”I regret that I have to maintain a layout system. It’s a very difficult piece of code to keep running. There are all sorts of edge-cases you have to cover. And there are... I don’t think we made the wrong call. I think we made the far harder call.” (Smith, 2014).

Those abstractions can clearly define an entry barrier. But it does not end at that point. Even when a developer did learn all those concepts, there is a high chance that they will sooner or later need to implement some details of their application ”bare-metal” on one or multiple native platforms. For example, when existing native components should be integrated, or some platform-specific customizations need to be made. Therefore they need to understand what is happening under the level of abstraction they are usually dealing with, too.

When working with Fabulous, also, much complex wiring is happening in the background in order to implement the MVU architecture on top of Xamarin.Forms. For example, the uni-directional dataflow is not directly visible – functions may seem to be called ”magically” when messages get dispatched. In order to understand what is happening and how all those functions relate, at least a basic understanding of MVU is needed.

30 https://docs.microsoft.com/de-de/dotnet/api/xamarin.forms.entry (retrieved August 8, 2019)
31 https://developer.apple.com/documentation/uikit/uitextfield (retrieved August 8, 2019)
32 https://developer.android.com/reference/android/widget/EditText (retrieved August 8, 2019)

But that alone will not be sufficient, as Fabulous does closely map Xamarin.Forms's techniques to create user interfaces. Therefore it brings its own view DSL, which, from a developer's perspective, needs to be understood and learned as well. Furthermore, that view DSL must be maintained by the Fabulous authors. Which means that for every Xamarin.Forms update they need to match the additions and changes made to Xamarin.Forms on the Fabulous side.

Besides, with Fabulous.SimpleElements[33] there does at least one additional library exist that tries to simplify that view DSL. It claims to provide especially easy API discoverability by enabling the developer to "dot through" the code and see what attributes can be used. The downside: When using this library, the UI code would be at least four levels of abstraction off from the original platform code (Platform → Xamarin.Forms → Fabulous → Fabulous.SimpleElements).

During the implementation of both versions of IUBH TOR, however, both building the UI with XAML and with the Fabulous DSL turned out to work generally unproblematically.

6.2.2 Visibility

Solving complex problems in a simple way is one of the most challenging tasks in software development. In order to improve long-term maintainability, an application's architecture should be structured in a way that, among other qualities, allows a developer to join the dots quickly. Therefore it is essential to have a strategy in place that allows structuring system components in a way that makes them easily accessible.

Questions

- Does the notation allow to get a quick overview of the structure of the application?
- Are there essential details hard to understand due to encapsulation enforced by the notation?

Evaluation

The core project of the IUBH TOR F# implementation does not contain a single subfolder but does consist only of a dozen files. That is relatively common in the F# space. Organizing F# files in nested folder structures instead would make it harder to organize the solution. Because the only thing that matters to the compiler is the eventual list of files and their (full) path.

Also, for pages, view, model, update, and init functions are always put together in one single file and can therefore easily be looked at at a glance. That may only become a problem when such a page file grows very large over time and contains multiple different functions, types, etcetera. But that did not happen during the the development of IUBH TOR.

Services are organized into F# modules, and each module lives in its own file, which makes it easy to

[33] https://github.com/Zaid-Ajaj/fabulous-simple-elements (retrieved August 8, 2019)

discover the different services. There is no noticeable part of the F# solution that would ”obfuscate” details.

In contrast, the IUBH TOR C# project consists of many different files that needed to be organized. Lilienthal, 2017, p. 117 suggests to ”slice” an application not from a technical point of view, but rather a domain-centric one. The reasoning behind this is the idea of getting coherent modules that are easier to reason about. For example, for a specific topic (”bounded context”), everything related to it is placed in a particular location where it can easily be discovered.

IUBH TOR C# therefore contains in its core project only global infrastructure on its root level, e.g., for data access. Everything else is placed into packages that are called modules (”Authentication,” ”Courses,” and ”Shared”). Which makes it relatively easy to reason about those modules and their inner working.

On a detail level, view (XAML) and view model are separated in different files and written in different languages. That makes it relatively hard to get an overview, as functionality can live either on the page, in the view model, or even in the XAML part of a view.

6.2.3 Hidden dependencies

In order to get an understanding of the code of a software system, it is not only necessary to identify and classify its components. Learning how those components depend on each other is a crucial part of being prepared for applying significant changes (or sometimes any changes at all). What may sound obvious, often is a complicated issue. Hidden dependencies are not only a source of unexpected problems and newly introduced bugs during maintenance, but also the cause of unforeseen delays and costs. Notations with the ability to avoid those hidden dependencies are therefore having an advantage, whereas the lack of those abilities can be seen as a threat to a project’s long-term success.

Questions

- Are hidden dependencies quickly introduced or even encouraged by the notation?
- Does the notation contain mechanisms to avoid hidden dependencies?

Evaluation

Overall, dependencies are clearly modeled in the F# version of IUBH TOR. However, some functions depend on other non-local functions, e.g. `CourseLoader.tryLoadCoursesFromCARE` which calls functions from 4 other modules: `Authentication`, `CoursePageHtmlDownloader`, `CoursePageHtmlParser`, and `CourseUpdater`.

That is a trade-off that was deliberately accepted in order to make the `tryLoadCoursesFromCARE` function itself more usable. If it was not calling those four functions on its own, its callee would need to know

and provide its implementation details. Moreover, even worse, if the function would be used multiple times across the application, any callee would have to know the implementation.

However, that is a design decision that depends highly on the context. In general, following a ports and adapters architectural style in FP with F# is possible and in many cases useful and encouraged (Seemann, 2016).

One classic example of a hidden dependency in object-oriented systems is the fragile base class problem (Mihailov and Sekerinski, 1998). There are two occurrences of that problem contained in the C# version of IUBH TOR: `ContentPageBase<TViewModel>`, and `ViewModelBase`.

Both provide fundamental functionality that is being used by different implementations (pages and view models). If one would change that functionality, the compiler would not be able to offer protection from producing unexpected side effects.

Those effects, therefore, would only occur on runtime. Unfortunately, it is not much that can be done about that besides massive duplication of code, which leads to even worse problems as those that could come along with the fragile base class problem.

6.2.4 Hard mental operations

In order to understand the structure of a software system, its components, and their relationship, it is crucial to gain a thorough overview. What mainly means to understand how data flows and is processed within the system. That becomes relevant at the latest when something went wrong, and the problem must be identified. The notation of a system can make this noticeable easy or rather difficult.

Questions

- Does the notation make it harder or easier to understand the life cycle of the program?
- Can errors, if occurred, easily be traced to their sources?

Evaluation

One benefit of the MVU architecture is that due to the unidirectional data-flow, which was described earlier, it is always clear how data moves through the different parts of the application. Whenever a problem occurs, it can be reproduced by applying the same state to the program and then triggering the command that caused it. This can be seen as a major advantage of MVU over different architectures, and especially MVVM.

More generally, the by the F# compiler enforced order of execution on both a macro and a micro level makes it easier to reason about a whole F# project as well as a single F# file. Anything that should

be used (functions, types, etcetera) must have been declared before. A good example is the project file of the IUBH TOR core project.

Listing 25: Extract of an F# project file

```
<ItemGroup>
    <Compile Include="Constants.fs" />
    <Compile Include="Domain.fs" />
    ...
    <Compile Include="CourseLoader.fs" />
    <Compile Include="LoginPage.fs" />
    ...
    <Compile Include="App.fs" />
</ItemGroup>
```

The order of the file entries in IUBH TOR.fsproj defines the order in which the F# compiler processes those files. That makes, for example, the ”root node” always come last – in this case, that is the App.fs file, which is the app's starting point. It also helps to avoid cyclic references: while LoginPage.fs could reference Domain.fs, it is not possible vice-versa. What makes it quite easy to reason about an F# program in general and a Fabulous program in particular.

Concerning the C# implementation of IUBH TOR it can be stated that, as Syme and Bennett, 2018 note, ”XAML is not simple”. In order to render the course list page, for example, much knowledge is needed to configure the necessary attributes in the root node alone. There is also some knowledge necessary about converters, platform-specific attributes, and bindings. All of this takes place in an additional layer written in another language than the rest of the application.

During the development of the C# version of IUBH TOR, a bug in the Xamarin.Forms list view component was found. When applying the material design visual style, selecting cells from a list was not possible anymore. A workaround could be found. However, some additional C# code became necessary. In order to fully understand the course list page, reading and understanding not only of the XAML code but also of the C# code for page and view model became therefore necessary. Also, subsequent defects can be introduced in all of those parts now, which makes them potentially hard to trace down.

6.2.5 Closeness of mapping

When building mobile applications, one particularly important part of the job is to create graphical user interfaces, which are often very sophisticated. Usually, those interfaces are built as hierarchical structures that consist of multiple nested layers of information. A button, for example, might consist of a rectangle on layer 0, and a label, containing some text, on layer 1 on top of it.

Therefore it is helpful when the notation allows modeling elements in a hierarchical way as close to

the final structures as possible, instead of forcing the developer to construct them linearly.

Good examples for those notations in practice might be the Extensible Markup Language (XML) or the JavaScript Object Notation (JSON). Both are widely used in scenarios where hierarchical data needs to be expressed. XML, in particular, has been a research subject for quite some time (e.g., Luyten, Abrams, Vanderdonckt, and Limbourg, 2004) and has become the defacto standard for building user interfaces of all kinds. It is also heavily used in different dialects in the mobile app space on all major platforms (e.g., Storyboards on iOS, AXML on Android, or XAML on Windows).

Questions

- Does the notation allow to describe the UI in a way that is close to its hierarchical nature?
- Are there characteristics of the notation that appear to be unnatural when building the UI?

Evaluation

Fabulous introduces a custom DSL to build views. The DSL closely maps existing Xamarin.Forms elements and their attributes. That makes it quite easy for developers who are used to the XAML notation to get started. Developers who do not have a background in Xamarin.Forms development can start with the official documentation[34], as names of elements and attributes are taken over 1:1.

Thanks to the nature of F# as a language where (almost) everything is an expression, the view DSL is quite expressive. It allows building the user interface in a natural hierarchical way, as does XAML.

As an additional benefit over XAML, the view can be split up into multiple local or even global functions. That not only reduces deep nestings and repetition, but it also allows to group parts of the view into reusable components. For example, to render a row on the course detail page, the following row function is being used.

Listing 26: Fabulous: Rendering a row on the course detail page

```
let private row title value margin =
    View.StackLayout(
        orientation = StackOrientation.Vertical,
        isVisible = not (String.IsNullOrWhiteSpace value),
        children = [
            View.Label(
                text = title,
                margin = margin,
                fontSize = Constants.UI.FontSize.Small,
                textColor = Color.SlateGray)
            View.Label(text = value)])
```

With regards to C#, user interfaces for Xamarin.Forms can be built both programmatically in C# and in a declarative way in XAML. XAML allows representing the actual view hierarchy almost 1:1 with

[34] https://docs.microsoft.com/en-us/xamarin/xamarin-forms/ (retrieved August 8, 2019)

some minor exceptions. For example, the notation for hiding or showing a frame on the course list, depending on the current state of that list, could be seen as over-complex and even unnatural:

Listing 27: XAML: Toggling a frame through a binding

```
<Frame.IsVisible>
    <Binding Path="State" Converter="{StaticResource courseStateConverter}">
        <Binding.ConverterParameter>
            <domain:CourseListState>Loading</domain:CourseListState>
        </Binding.ConverterParameter>
    </Binding>
</Frame.IsVisible>
```

If the same view would be built programmatically in C#, the result would usually contain much procedural code with many temporary variables. However, there do exist ideas and first implementations inspired by MVU which try to enable developers to write user interfaces in C# in a declarative and hierarchical way[35].

6.2.6 Error-proneness

A world in which software does not contain any error at all is something everyone would dream of. However, reality often is telling a different story. While not every program does require ”rocket science” to be written, even small flaws in line of business applications can lead to enormous costs. That is true especially in the mobile app space, where it is usually not possible to ship fixes to a user as often and as fast as in server applications. The complex deployment and review process, enforced primarily by Apple, does not allow it.

Questions

- How likely are (runtime) errors caused by mistakes expressed in the notation?
- Does the notation provide any obstacles that make it hard to test programs automatically?

Evaluation

In general, it seems worth considering to choose a development approach over another, which is less error-prone. Ray, Posnett, Filkov, and Devanbu, 2014, for example, found out that ”There is a small but significant relationship between language class and defects. Functional languages have a smaller relationship to defects than either procedural or scripting languages.”

While developing the F# version of IUBH TOR, almost all parts of the application could be covered by unit tests – including its views. That made the usage of test-driven development (Beck, 2003) quite attractive from the beginning.

[35] https://ryandavis.io/declarative-code-based-xamarin-forms-ui/ (retrieved August 8, 2019)

However, the weak spot of Fabulous in that regard is the testability of commands. At the time of writing, there does not exist a way to evaluate the result of a command that got triggered through the update function.

The compromise has been to introduce a naming convention (`Cmd*`) for those message types that would trigger a command. This way, unit tests could be written, which make sure that a command is really triggered. When the update function then receives such a `Cmd*` message, their actual command function is returned. This function itself can then be tested in isolation of the MVU lifecycle.

One example is the logout functionality of IUBH TOR:

Listing 28: Fabulous: Unit tests covering the logout functionality

```
type ``When the logout is being started``() =
    [<Fact>]
    let ``The Logout Command is being dispatched``() =
        let _, cmd, _ = CourseListPage.update CourseListPage.Msg.StartLogout
        ↪  (initialModel())
        cmd |> dispatchesMessage CourseListPage.Msg.CmdLogout |> should be True

type ``When the Logout Command is being executed``() =
    [<Fact>]
    let ``A LogoutSucceeded message is being returned when the credentials could be
    ↪  removed``() =
        let remove () = Ok()
        let result = CourseListPage.tryLogOut remove
        result |> should equal CourseListPage.Msg.LogoutSucceeded

    [<Fact>]
    let ``A LogoutFailed message is being returned when the credentials could not be
    ↪  removed``() =
        let errorMessage = randomString()
        let remove () = Error errorMessage
        let result = CourseListPage.tryLogOut remove
        result |> should equal (CourseListPage.Msg.LogoutFailed(errorMessage))
```

When a user presses the logout button, a `CmdLogout` message is being dispatched. That message is then processed by the update function, which starts the actual logout procedure by returning a command that initiates the execution of the `tryLogOut` function. The only part that cannot be covered by a test is the initiation of the logout process itself:

Listing 29: Fabulous: Extract of the global update function

```
let update (msg: Msg) (model: Model) =
    match msg with
    ...
    | CmdLogout ->
        model, (Cmd.ofMsg (tryLogOut
        ↪  Authentication.tryRemoveCredentialsFromSecureStorage)), NoOp
```

That last line must be taken care of manually, which seems to be an acceptable trade-off.

In general, the F# version of IUBH TOR benefited from the capabilities of F# as a language, its type system, and its compiler. Primarily features like results, options, and built-in immutability for most types turned out to be very helpful in order to prevent errors.

Unfortunately, some of this comfort has been given up on by the authors of Fabulous in regards to building views. The current DSL works with a lot of optional arguments that expect objects as values. This lack of type-safety makes it, for example, possible to pass a boolean where a float is expected. Which would lead to runtime errors.

In comparison, the C# implementation of IUBH TOR suffered from some shortcomings that are natural to how Xamarin.Forms works. For example, XAML views are static and cannot be unit-tested. Finding errors and making sure they do not reappear becomes a tedious task under some circumstances, especially when a view grows over time.

What turned out to be helpful was to enable the XAML compiler which directly compiles XAML into IL. That does not only reduce startup time and the file size of the final assembly, but it also performs some compile-time checkups to ensure the XAML markup is valid (Microsoft, 2018b).

Another part that is hardly testable and can cause unexpected side-effects during runtime is the inversion of control container[36]. A developer must make sure that all necessary dependencies have been properly registered in the correct order. Otherwise, exceptions could be thrown because of unresolvable types, which can lead to crashes of the application.

6.2.7 Diffuseness

The less code needs to be written to solve a particular task, the less code needs to be maintained and therefore debugged, read, and understood in the future (Atwood, 2007). This is even more important as developers usually only spend roughly 10% of their time writing code, but 20% understanding a problem and astonishing 70% reading code (Lilienthal, 2017, p. 69).

When a notation is complex or bloated, understanding and reading code becomes a tedious task. Therefore notations that allow concisely expressing things are seen as beneficial long-term, even when they seem hard to learn in the beginning.

Questions

- Does the notation allow to express things on a micro-level concisely and efficiently?
- On a macro level, does the notation cause the developer to write much boilerplate code?

[36] https://martinfowler.com/articles/injection.html (retrieved August 8, 2019)

Evaluation

F# code is considered to be more concise than C# code (Liu, 2013, p. xiii), which is a characteristic of many functional programming languages (Nanz and Furia, 2015). For example, Odersky, Spoon, and Venners, 2008, p. 49 state, that ”a … conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java.”

This could have a significant impact on the economics of a software development project. Not only can less code be written in less time, it will with high probability also contain fewer bugs (Ray et al., 2014).

Table 3: IUBH TOR: Lines of Code

Language	Number of files[37]	Blank lines	Comment lines	Code lines
C#	88	1.047	452	4.855
XAML	4	10	0	355
F#	28	457	223	2.121

Source: Own illustration

As shown in table 3, the F# implementation of IUBH TOR uses 28 files and 2.121 lines of code, whereas the C# implementation consists of 92 different files containing 5.210 lines of code (C# + XAML combined). In other words: the C# code base is almost 2,5x as large as the F# equivalent.

While F# code, in general, is much more terse than C# code, building IUBH TOR with the Fabulous library and its view DSL clearly shows the advantages in particular. For example, when taking a look at the course detail page which does not contain advanced logic but only renders the information of a course, the F# implementation consists of only a single file and 45 lines of code. Whereas the C# implementation uses three files containing 169 lines of code in sum, not included the usage of the shared `HideEmptyDataConverter` class.

Aside from the UI implementation, both alternatives suffer from some boilerplate code that needs to be written. For F# and Fabulous, implementing the nested navigation stack (list → detail) required some additional message types and mapping operations inside the app’s update function. That was necessary in order to preserve the single message loop the MVU architecture is based on.

For C#, achieving a scalable architecture requires some boilerplate code to be written around enabling

[37] Counted by using cloc. Only files within /src/cs for C# and within /src/fs/ for F# were evaluated. Resource.designer.cs was deleted before the test was run for both the C# and the F# Android project, as it gets automatically generated. Also, TinyIoC.cs contained in the C# core project was deleted as it was copied to the project as supposed by the library’s author, but not written for the app itself.

the inversion of control container to work. For example, every module needs to take care of registering its dependencies to the global registry.

6.2.8 Secondary notation

Software libraries and frameworks are tools addressing particular technical ecosystems, enabling their users to solve their own (business) problems on those platforms. However, the capabilities of those tools can always only go so far as their authors have been planning for it. In practice, developers often have to deal with problems that go far beyond the scope the tool authors had initially in mind for their products. It is therefore essential that those tools can be extended for such custom use cases. That can be difficult if, for example, specific APIs of such a tool are not publicly available and therefore not usable for customization (Zibran, Eishita, and Roy, 2011).

Questions

- Does the notation allow to be extended for individual use cases not officially covered?
- Is extending the notation officially encouraged, or have ”hacks” to be applied?

Evaluation

While basic examples of Fabulous often only include simple scenarios, even building the F# version of IUBH TOR with just three different dialogs required some extensions of the default instruments.

For example, the app’s model should contain session state information, so it could be determined whether the login page or the course list page should be shown after the app starts. However, changes to the session state are triggered by the login page (when logging in) and the course list page (when logging out), not through the app module itself. In order to be able to retain one single message loop for the whole application, those two pages must be able to ”hook into” that loop. That could be realized by introducing a second type of messages: external messages.

Listing 30: Fabulous: External messages

```
type Msg =
    | CourseListPageMsg of CourseListPage.Msg

let update (msg: Msg) (model: Model) =
    match msg with
    | CourseListPageMsg msg ->
        let m, c, em = CourseListPage.update msg model.CourseListPage
        let model = { model with CourseListPage = m; }
        let model, ec =
            match em with
            | CourseListPage.ExternalMsg.LogoutSucceeded ->
                (fst (init false)), Cmd.ofMsgOption (notifyAboutLogout())

    model, Cmd.batch [ Cmd.map CourseListPageMsg c; ec ]
```

This listing shows an extract of the app's update function which handles the course list page's external message, which signals that the user just logged out successfully. Also, while the parent knows about its child (the course list page), the child, on the contrary, is not aware of its parent (the app module). The contract between both is the ExternalMsg type provided by the course list page, the child.

Extensions of the existing instruments are possible, even if not officially encouraged. What is encouraged, however, is to build a wrapper to enable first-class support for existing libraries and Xamarin.Forms controls[38].

On top of that, all existing customization options provided by Xamarin.Forms can be used. Both implementations of IUBH TOR use, for example, the Material Design visual system for Android and iOS[39], a set of readily prepared custom renderers that support the styling of UI elements using the Material Design system[40] created by Google.

The concept of custom renderers is not limited to official implementation but can be used wherever necessary for one's account. Both iOS implementations of IUBH TOR, for example, contain a custom renderer for list view cells. It makes sure a cell that has been selected by the user does not stay in its visual selected state forever. Which, unfortunately, is the default behavior.

All in all, it can be stated that both Xamarin.Forms itself and Fabulous on top can be customized and extended in all sorts of ways.

6.2.9 Progressive evaluation

Receiving feedback on applied changes, e.g., while developing the user interface of a cross-platform mobile application, is an elementary part of the development cycle. In the simplest case, it is, e.g., necessary to check changes made to the positioning of an element on the screen for multiple target platforms. Moreover, while the result can be satisfying on platform A, it may be different or even wrong on platform B.

Also, it may be helpful to sketch out an early version of an algorithm in a way that does set as few constraints as possible to the developer. So they can try things out without the heavy-lifting of compiling and running a full application on a target platform.

Questions

- Does the notation support building UIs in an incremental way?
- Does the notation allow to ”sketch things out” also for non-UI related code?

---

38 https://fsprojects.github.io/Fabulous/Fabulous.XamarinForms/views-extending.html (retrieved August 8, 2019)

39 https://devblogs.microsoft.com/xamarin/beautiful-material-design-android-ios/ (retrieved August 8, 2019)

40 https://material.io/ (retrieved August 8, 2019)

Evaluation

One of the most tedious routines of working with Xamarin until recently has been to start the build and deploy cycle to see code changes ”in action.” Depending on the app’s size and the deployment target, this would take between a couple of seconds and several minutes. However, those times are gone now in general as a couple of live update tools have been published in the recent past.

Fabulous ships with ”Live Update” which allows reloading the application after changes were made to the F# code. During the development of IUBH TOR that worked for both iOS and Android and both simulator/emulator and real physical devices. However, only changes made to the view function were reflected reliably. Changes made to the model and other parts of the codebase sometimes required to start a new debug cycle. Also restoring the state did not work well, so some hacks needed to be applied (like temporarily commenting out code in order to get the right page loaded).

But working on the UI was only one part of the development process that was being done incrementally in small iterations. Other parts included the development of algorithms, learning how to apply different libraries (e.g., F# Data), and getting to know concepts like railway-oriented programming[41]. All of this could be done comfortably by using the previously mentioned F# Interactive console and F# Scripts.

For C#, the usage of ”HotReload”[42] has proven to be more reliable than Fabulous’ ”Live Update”. The open-source tool reacts to changes made to XAML views and reloads them ”in place,” so not the whole application is being reloaded. However, proven unreliable has the XAML preview in Visual Studio for Mac. This tool tries to render a view depending on the XAML markup, but it fails to do so as soon as custom data-bindings are being applied.

6.2.10 Accessibility

Whether or not someone new to a notation can master it, depends on a wide variety of different aspects. For example, without a certain level of motivation, stamina, and self-discipline, it will be hard to get proficient in any new programming language or technology ecosystem. On the other hand, however, the success of a new notation in terms of broad acceptance and adaption by their target audience can, to a certain degree, be influenced by the creators of the notation itself. Besides some fundamental qualities like the suitability to solve the addressed problem as communicated, the notation must be ”accessible” by users unfamiliar with it.

Questions

- How comprehensive are existing learning materials?
- How mature are tooling and the ecosystem?

41 https://fsharpforfunandprofit.com/rop/ (retrieved August 8, 2019)
42 https://github.com/AndreiMisiukevich/HotReload (retrieved August 8, 2019)

Evaluation

Trying to get started with both F# and Fabulous at the same time will most likely be overwhelming for developers lacking basic knowledge of both the language and the library. On top of that, some basic knowledge of Xamarin.Forms and mobile application development in general is helpful. All in all, this appears to be a steep learning curve.

On the other hand, getting started with F# and Xamarin.Forms is supported by a myriad of books (e.g., Abraham, Wlaschin), blog posts[43], recorded presentations at conferences and user group meetings, video lectures, and last but not least thousands of answered questions about F#[44] and Xamarin.Forms[45] on Stack Overflow.

At the same time, the official documentation for both F#[46] and Xamarin.Forms[47] is quite comprehensive and of good quality. The official documentation of Fabulous is maintained by its community of volunteers. However, at the time of writing those Fabulous docs are not as complete. But for the development of IUBH TOR, it was nonetheless possible to get started quickly.

Working with Fabulous is straight-forward for all IDEs that support Xamarin, which is Visual Studio, Visual Studio for Mac, and JetBrains Rider. The code can be edited in Visual Studio Code, too, but running and debugging a Xamarin app from there is currently not supported. JetBrains Rider turned out to be a reasonable choice. Code analysis, unit test execution, setting and evaluating breakpoints while debugging the app on iOS or Android, running Fabulous Live Update from the integrated terminal – all of that worked reliably.

However, where F# falls short compared to C# is the set of refactoring operations offered by all IDEs. The renaming of symbols is the only operation that works reliably. Compared to dozens of refactoring options offered especially by JetBrains Rider, this is a disadvantage that is only compensated through the powerful language features of F# itself.

Besides, the F# ecosystem is rather small compared to C#. But the work on the F# version of IUBH TOR showed its maturity. Libraries like F# Data or FsUnit[48] could be easily integrated and used. In addition, F# can leverage large parts of the C# ecosystem as well. For example, widespread .NET libraries like xunit[49] could be used.

Developing the C# version of IUBH TOR worked as expected for an environment that could be considered to be one of the largest software development ecosystems worldwide. Tooling, learning ma-

43 A true treasure box is https://fsharpforfunandprofit.com/ (retrieved August 8, 2019)
44 https://stackoverflow.com/questions/tagged/f%23 (retrieved August 8, 2019)
45 https://stackoverflow.com/questions/tagged/xamarin.forms (retrieved August 8, 2019)
46 https://docs.microsoft.com/en-us/dotnet/fsharp/ (retrieved August 8, 2019)
47 https://docs.microsoft.com/en-us/xamarin/xamarin-forms/ (retrieved August 8, 2019)
48 https://fsprojects.github.io/FsUnit/ (retrieved August 8, 2019)
49 https://xunit.net/ (retrieved August 8, 2019)

terials, documentation – all of this receives much more investment by vendors like Microsoft and JetBrains, as well as the development community. That seems justified as the target audience is orders of magnitudes more significant than in the functional programming camp. E.g. there are right now more than 1.3 million questions asked[50] about C# on Stack Overflow.

7 Conclusion and future work

Using the Cognitive Dimensions of Notations framework allowed the thorough evaluation of IUBH TOR, the app developed for this thesis. By evaluating ten dimensions, valuable insights could be gained. From the perspective of the F# version, those are:

- Xamarin.Forms provides a sophisticated abstraction over iOS and Android, which Fabulous leverages to implement its idea of functional programming for app development.
- IUBH TOR consists of relatively few files, and all parts of its pages are contained within a single file per page. Which increases the visibility of relevant parts of the application.
- Dependencies are generally clearly modeled. However, at some points, compromises were made, and hidden dependencies were deliberately taken into account.
- The strict ordering enforced by the F# compiler makes it generally easy to understand an F# program, and the uni-directional data flow of MVU helped to understand and debug IUBH TOR.
- Views could be built in a natural and concise way. Fabulous closely maps Xamarin.Forms' XAML views and adds some additional benefits, like splitting up re-usable UI parts as functions.
- Unit tests could be written easily for every part of IUBH TOR, except for commands. Furthermore, F# features such as results, options, and built-in immutability helped to prevent different types of defects. However, Fabulous gives up type-safety partly for building views, which can lead to runtime errors.
- The resulting code base is much more concise than the C# version, which contains 2,5x as much code and 3x as many files.
- The basic MVU structure needed to be adapted in order to make a nested navigation stack feasible. Further customizations through, e.g., custom renderers, were applied, too.
- Fabulous' ”Live Update” helped to build the app incrementally. Unfortunately, it only worked reliably for the views, not for other parts. F# Interactive, however, provided a comprehensive way to ”sketch out” things like algorithms or try new libraries.
- Comprehensive learning materials for all technologies involved in the case study were available. The tooling for F# is stable but in terms of refactoring support not as sophisticated as the tooling for C#. Disregarding the rather small ecosystem of F#, mature libraries for all kinds of tasks could be found and applied.

[50] https://stackoverflow.com/questions/tagged/c%23 (retrieved August 8, 2019)

As so often, decisions on which technologies to adopt for a new app project must be formulated under a state of uncertainty, given the complex nature of such a call. However, looking at the results of the evaluation, it can be stated that functional programming with F# – on the Xamarin platform – can be a viable alternative to the established object-oriented approach with C#.

F# as a language and FP as a pardigm have proven to keep their promises as advertised. F# provides first-class FP capabilities while offering seamless integration in the OOP world of iOS, Android, and Xamarin. Tooling in terms of IDEs might in comparison to C# not be as sophisticated. However, stable and comfortable editing solution were found. Last but not least, the ecosystem in terms of available libraries provided different options for all kinds of tasks.

Therefore, when Xamarin.Forms is chosen as a platform, the implementation of Fabulous can be recommended.

For cases where Xamarin.Forms is not an option in the first place, writing views and view models in F# might not be the best option. Instead, it can be considered to write the UI in C# and the application's core in F#. Fortunately, .NET and therefore the Xamarin build system allows mixing C# and F# projects within one solution.

This thesis has shown that F# and FP are suitable for mobile app development in general. However, due to the limited scope, it has only been possible to cover the essential parts of the subject. This may set the foundation for further research.

For example, the higher the level of abstraction, the more critical it is that the library one mainly depends on performs proper resource management, especially on mobile systems. Examining performance characteristics and memory usage of applications built with Fabulous, in particular, could provide valuable insights.

A long-term study accompanying a longer running complex app project realized with Fabulous could also provide essential findings. In particular, evaluating how a large app can be built with MVU and how well that architecture scales for such a case would be interesting to see. Something that could not be covered during the implementation of IUBH TOR because of its small size.

Furthermore, to better understand what might make functional programming more attractive for mobile developers in general (or what might tend to hold them back), user studies could be conducted.

All in all, this subject offers much potential for further research that can contribute to freeing functional programming from its academic niche and to further establish it as a viable alternative to the mostly object-oriented mainstream techniques, which currently dominate the software development industry.

References

Abraham, I. (2018). *Get Programming with F#*. Birmingham: Manning Publications.

Alexander, A. (2017). *Functional Programming, Simplified - (Scala Edition)*. CreateSpace Independent Publishing Platform.

AltexSoft. (2017). Performance Comparison: Xamarin.Forms, Xamarin.iOS, Xamarin.Android vs Android and iOS Native Applications. Retrieved August 8, 2019, from https://www.altexsoft.com/blog/engineering/performance-comparison-xamarin-forms-xamarin-ios-xamarin-android-vs-android-and-ios-native-applications/

Atwood, J. (2007). The Best Code is No Code At All. Retrieved August 8, 2019, from https://blog.codinghorror.com/the-best-code-is-no-code-at-all/

Bandt, T. (2017). Powerful Scripting With F#. Retrieved August 8, 2019, from https://thomasbandt.com/powerful-scripting-with-fsharp

Bandt, T. (2018). Building An F# Redux Store For C# Xamarin Apps. Retrieved August 8, 2019, from https://thomasbandt.com/fsharp-redux-store-for-xamarin-apps

Beck, K. (2003). *Test Driven Development: By Example*. Addison-Wesley Professional.

Bennett, J. (2018). Building mobile apps in F# using Xamarin.Forms and Elmish. Retrieved August 8, 2019, from https://www.jimbobbennett.io/building-mobile-apps-in-f-using-xamarin-forms-and-elmish/

Biczó, M., Pócza, K., & Porkoláb, Z. (2009). Runtime access control in C# 3.0 using extension methods. In *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* (Vol. 30, pp. 41–60). Retrieved August 8, 2019, from http://ac.inf.elte.hu/Vol_030_2009/041.pdf

Bierman, G. M., Meijer, E., & Torgersen, M. (2007). Lost in translation: Formalizing proposed extensions to C#. In *OOPSLA* (Vol. 7, pp. 479–498). doi:10.1145/1297027.1297063

Bishop, M., Dilger, M. et al. (1996). Checking for race conditions in file accesses. *Computing systems*, *2*(2), 131–152. Retrieved August 8, 2019, from http://static.usenix.org/publications/compsystems/1996/spr_bishop.pdf

Blackwell, A., & Green, T. (2003). Notational Systems—The Cognitive Dimensions of Notations Framework. doi:10.1016/B978-155860808-5/50005-8

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* (pp. 592–605). doi:10.1109/WAINA.2009.190

Brais, H. (2017). Immutable Collections. Retrieved August 8, 2019, from https://msdn.microsoft.com/en-us/magazine/mt795189.aspx

Buonanno, E. (2018). *Functional Programming in C#*. Manning, Shelter Island.

Chua, B. B., & Dyson, L. E. (2004). Applying the ISO 9126 model to the evaluation of an e-learning system. In *Proc. of ASCILITE* (pp. 5–8). doi:10.1515/itc-2015-0008

Church, A. (1941). *THE CALCULI OF LAMBDA-CONVERSION*. Princeton University Press.

Church, A., & Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society*, *39*(3), 472–482. doi:10.2307/2268572

Clarke, S., & Becker, C. (2003). Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*. Retrieved August 8, 2019, from https://pdfs.semanticscholar.org/1193/4bc6a4cb76700c38d86c7106b09fe2147d3d.pdf

Cleron, M. (2017). Android Announces Support for Kotlin. Retrieved August 8, 2019, from https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html

Czaplicki, E. (2012). Elm: Concurrent FRP for Functional GUIs. *Senior thesis, Harvard University*. Retrieved August 8, 2019, from https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf

Czaplicki, E. (2018). The Elm Architecture. Retrieved August 8, 2019, from https://guide.elm-lang.org/architecture/

de Icaza, M. (2011). Announcing Xamarin. Retrieved August 8, 2019, from https://tirania.org/blog/archive/2011/May-16.html

Deissenboeck, F., Juergens, E., Lochmann, K., & Wagner, S. (2009). Software quality models: Purposes, usage scenarios and requirements. In *2009 ICSE Workshop on Software Quality* (pp. 9–14). doi:10.1109/WOSQ.2009.5071551

Dorfmann, H. (2016). Model-View-Intent on Android. Retrieved August 8, 2019, from http://hannesdorfmann.com/android/model-view-intent

F# Software Foundation. (n.d.). About F#. Retrieved August 8, 2019, from https://fsharp.org/about/index.html

Fayzullaev, J. et al. (2018). Native-like Cross-Platform Mobile Development: Multi-OS Engine & Kotlin Native vs Flutter. Retrieved August 8, 2019, from https://www.theseus.fi/bitstream/handle/10024/148975/thesis_Jakhongir_Fayzullaev.pdf

Gauld, A. (n.d.). Learning to Program. Retrieved August 8, 2019, from http://alan-g.me.uk/tutor/tutfctnl.htm

Gilbert, J., & Stoll, C. (2014). Architecting iOS Apps with VIPER. Retrieved August 8, 2019, from https://www.objc.io/issues/13-architecture/viper/

Gossman, J. (2005). Introduction to Model/View/ViewModel pattern for building WPF apps. Retrieved August 8, 2019, from https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/

Gough, J. J., & Gough, K. J. (2001). *Compiling for the .Net Common Language Runtime*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Grady, R. B. (1994). Successfully applying software metrics. *Computer*, *27*(9), 18–25. doi:10.1109/2.312034

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, *7*(2), 131–174. doi:10.1006/jvlc.1996.0009

Green, T. R. (1989). Cognitive dimensions of notations. *People and Computers V*, 443–460. Retrieved August 8, 2019, from https://pdfs.semanticscholar.org/4ca2/4a6a487c3fa92d60a17b760cc3515708896a.pdf

Guthrie, S. (2016). Microsoft to acquire Xamarin and empower more developers to build apps on any device. Retrieved August 8, 2019, from https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/

Hadhrawi, M., Blackwell, A. F., & Church, L. (2017). A Systematic Literature Review of Cognitive Dimensions. In *PPIG* (p. 3). Retrieved August 8, 2019, from http://www.ppig.org/sites/ppig.org/files/2017-PPIG-28th-hadhrawi.pdf

Han, M.-F. (2015). On How Jet Chose F#. Retrieved August 8, 2019, from https://medium.com/mftech/on-how-jet-chose-f-df6fc2eb0588

Harrop, J. (2008). *F# for Scientists*. Wiley-Interscience.

Hejlsberg, A. (2010). Introducing Async – Simplifying Asynchronous Programming. Microsoft. Retrieved August 8, 2019, from https://channel9.msdn.com/Blogs/Charles/Anders-Hejlsberg-Introducing-Async

Hinsen, K. (2009). The promises of functional programming. *Computing in Science & Engineering*, *11*(4), 86. doi:10.1109/MCSE.2009.129

Hoare, T. (2009). Null references: The billion dollar mistake. Presentation. London.

Hutton, G. (2002). Frequently Asked Questions for comp.lang.functional. Retrieved August 8, 2019, from http://www.cs.nott.ac.uk/~pszgmh/faq.html

International Organization for Standardization. (2011). ISO/IEC 25010:2011. Retrieved August 8, 2019, from https://www.iso.org/standard/35733.html

Jackson, D. (2006). Software abstractions. Retrieved August 8, 2019, from http://sdm.mit.edu/conf06/Presentations/jackson_sdm06.pdf

James, M. (2014). Time Travel made Easy – Introducing Elm Reactor. Retrieved August 8, 2019, from https://elm-lang.org/blog/time-travel-made-easy

Kennedy, A., & Syme, D. (2001). Design and implementation of generics for the .NET Common language runtime. In *ACM SigPlan Notices* (Vol. 36, pp. 1–12). doi:10.1145/378795.378797

Kiss, E. (2014). Comparison of Object-Oriented and Functional Programming for GUI Development. Retrieved August 8, 2019, from https://github.com/eugenkiss/7guis/raw/master/thesis.pdf

Krasner, G. E., Pope, S. T. et al. (1988). A Description of the Model-View-Controller UserInterface Paradigm in the Smalltalk-80 System. *Journal of object oriented programming*, *1*(3), 26–49. Retrieved August 8, 2019, from https://www.researchgate.net/profile/Stephen_Pope/publication/248825145_A_cookbook_for_using_the_model_-_view_controller_user_interface_paradigm_in_Smalltalk_-_80/links/5436c5f30cf2643ab9888926/A-cookbook-for-using-the-model-view-controller-user-interface-paradigm-in-Smalltalk-80.pdf

Kuitunen, M. (2019). *Cross-Platform Mobile Application Development with React Native* (B.S. thesis). Retrieved August 8, 2019, from https://trepo.tuni.fi/bitstream/handle/123456789/27139/Kuitunen.pdf?sequence=4&isAllowed=y

Law, R. (2015). Clean Swift iOS Architecture for Fixing Massive View Controller. Retrieved August 8, 2019, from https://clean-swift.com/clean-swift-ios-architecture/

Leitner, P., Wittern, E., Spillner, J., & Hummer, W. (2019). A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software*, *149*, 340–359. doi:10.7287/peerj.preprints.27005v1

Lightbend, I. (2019). Akka: Build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka. Retrieved August 8, 2019, from https://akka.io/

Lilienthal, C. (2017). *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. dpunkt. verlag.

Liu, T. (2013). *F# for C# Developers*. Microsoft Press.

Loder, W. (2018). *Web Applications with Elm: Functional Programming for the Web*. Apress.

Luyten, K., Abrams, M., Vanderdonckt, J., & Limbourg, Q. (2004). Developing user interfaces with XML: Advances on user interface description languages. In *Sattelite workshop of Advanced Visual Interfaces 2004*. Retrieved August 8, 2019, from https://dial.uclouvain.be/downloader/downloader.php?pid=boreal:168618&datastream=PDF_01

Lynn, T., Rosati, P., Lejeune, A., & Emeakaroha, V. (2017). A preliminary review of enterprise server-less cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (pp. 162–169). doi:10.1109/CloudCom.2017.15

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality. volume i. concepts and definitions of software quality*. GENERAL ELECTRIC CO SUNNYVALE CA. Retrieved August 8, 2019, from https://apps.dtic.mil/dtic/tr/fulltext/u2/a049014.pdf

McCarthy, J. (1959). Recursive functions of symbolic expressions and their computation by machine. doi:10.1145/367177.367199

Meijer, E. (2008). Erik Meijer: Functional Programming. Retrieved August 8, 2019, from https://channel9.msdn.com/Shows/Going+Deep/Erik-Meijer-Functional-Programming

Meijer, E. (2011). The world according to LINQ. *Queue*, *9*(8), 60. doi:10.1145/2001269.2001285

Meijer, E., Beckman, B., & Bierman, G. (2006). LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (pp. 706–706). doi:10.1145/1142473.1142552

Microsoft. (2015). Statements (C# Programming Guide). Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements

Microsoft. (2016a). F# Language Reference. Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/index

Microsoft. (2016b). Pattern Matching. Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching

Microsoft. (2017). What is Xamarin? Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/introduction-to-mobile-development

Microsoft. (2018a). Introduction to Functional Programming in F#. Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/dotnet/fsharp/introduction-to-functional-programming/

Microsoft. (2018b). XAML Compilation in Xamarin.Forms. Retrieved August 8, 2019, from https://docs.microsoft.com/de-de/xamarin/xamarin-forms/xaml/xamlc

Microsoft. (2019). .NET Standard. Retrieved August 8, 2019, from https://docs.microsoft.com/en-us/dotnet/standard/net-standard

Mihailov, L., & Sekerinski, E. (1998). The Fragile Base Class Problem and Its Impact on Component Systems, 353–358. doi:10.1007/3-540-69687-3_72

Mono Project. (2019). Cross platform, open source .NET framework. Retrieved August 8, 2019, from https://www.mono-project.com/

Moore, G. A. (1991). *Crossing the Chasm*. HarperBusiness.

Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 778–788). doi:10.1109/ICSE.2015.90

Núñez, A. G.-C., & Fahad, S. (2016). *Mastering F#. A comprehensive and in-depth guide to writing functional programs using F#*. Packt Publishing.

Odersky, M., Spoon, L., & Venners, B. (2008). *Programming in Scala*. Artima Press.

Petersen, C. L., Gorges, M., Dunsmuir, D., Ansermino, M., & Dumont, G. A. (2013). Experience report: Functional programming of mHealth applications. In *ACM SIGPLAN Notices* (Vol. 48, pp. 357–362). doi:10.1145/2544174.2500615

Petricek, T. (2012). Teaching Functional Programming to Professional .NET Developers. Retrieved August 8, 2019, from https://pdfs.semanticscholar.org/e3ae/0867e724b9af2227f13ac519d4f369d96103.pdf

Petricek, T., & Skeet, J. (2009). *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co.

Petzold, C. (2015). Writing Xamarin.Forms Apps in F#. Retrieved August 8, 2019, from http://www.charlespetzold.com/blog/2015/10/Writing-Xamarin-Forms-Apps-in-FSharp.html

Petzold, C. (2016). *Creating Mobile Apps with Xamarin.Forms*. Microsoft Press.

Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 155–165). doi:10.1145/2635868.2635922

Rebouças, M., Pinto, G., Ebert, F., Torres, W., Serebrenik, A., & Castor, F. (2016). An empirical study on the usage of the swift programming language. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (Vol. 1, pp. 634–638). doi:10.1109/SANER.2016.66

Reed, R. (2014). That's Billion with a B: Scaling to the next level at WhatsApp. Retrieved August 8, 2019, from http://www.erlang-factory.com/static/upload/media/1394350183453526efsf2014whatsappscaling.pdf

Sadowski, C., & Kurniawan, S. (2011). Heuristic evaluation of programming language features: Two parallel programming case studies. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools* (pp. 9–14). doi:10.1145/2089155.2089160

Seemann, M. (2016). Functional architecture is Ports and Adapters. Retrieved August 8, 2019, from https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/

Seemann, M. (2018). Functional architecture: A definition. Retrieved August 8, 2019, from https://blog.ploeh.dk/2018/11/19/functional-architecture-a-definition/

Shackles, G. (2017). You, Too, Can Build Xamarin Apps with F#. Retrieved August 8, 2019, from https://visualstudiomagazine.com/articles/2017/01/01/build-xamarin-apps.aspx

Sink, E. (2015). Why your F# evangelism isn't working. Retrieved August 8, 2019, from https://ericsink.com/entries/fsharp_chasm.html

Smith, J. (2014). Gone Mobile 28: Behind the Scenes of Xamarin.Forms with Jason Smith/Interviewers: Greg Shackles, Jonathan Dick. Retrieved August 8, 2019, from https://player.fm/series/gone-mobile-podcast-52734/gone-mobile-28-behind-the-scenes-of-xamarinforms-with-jason-smith

Stack Overflow. (2019). Annual Developer Survey. Retrieved August 8, 2019, from https://insights.stackoverflow.com/survey/2019

Staltz, A. (2015). Unidirectional User Interface Architectures. Retrieved August 8, 2019, from https://staltz.com/unidirectional-user-interface-architectures.html

Sturm, O. (2011). *Functional programming in C#: Classic programming techniques for modern projects*. John Wiley & Sons.

Syme, D. (2010). Don Syme: Geek of the Week/Interviewer: Richard Morris. Retrieved August 8, 2019, from https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/don-syme-geek-of-the-week/

Syme, D. (2011). Archeological Semiotics: The Birth of the Pipeline Symbol, 1994. Retrieved August 8, 2019, from https://blogs.msdn.microsoft.com/dsyme/2011/05/17/archeological-semiotics-the-birth-of-the-pipeline-symbol-1994/

Syme, D. (2018). The Early History of F#. In *HOPL IV*. Advance online publication. Retrieved August 8, 2019, from https://fsharp.org/history/hopl-draft-1.pdf

Syme, D., & Bennett, J. (2018). Making Mobile App Programming Simple With F#. Published: Presentation given at Functional Londoners Meetup Group. Retrieved August 8, 2019, from https://www.youtube.com/watch?v=bEO7bl79uAM

Tolmachev, E. (2016). Cross-platform UIs with F# and Fable. Retrieved August 8, 2019, from https://thoughtspam.wordpress.com/2016/09/05/cross-platform-uis-with-f-and-fable/

Turner, D. A. (2012). Some history of functional programming languages. In *International Symposium on Trends in Functional Programming* (pp. 1–20). doi:10.1007/978-3-642-40447-4_1

Wadler, P. (1997). Functional programming: An angry half-dozen. In *International Workshop on Database Programming Languages* (pp. 25–34). doi:10.1007/3-540-64823-2_2

Wadler, P. (1998). Why no one uses functional languages. *Sigplan notices*, *33*(8), 23–27. doi:10.1145/286385.286387

Wampler, D. (2011). *Functional Programming for Java Developers: Tools for Better Concurrency, Abstraction, and Agility*. O'Reilly Media.

Wampler, D., & Clark, T. (2010). Guest editors' introduction: Multiparadigm programming. *IEEE Software*, *27*(5), 20–24. doi:10.1109/MS.2010.119

Warburton, R. (2016). *Object-Oriented vs. Functional Programming*. O'Reilly Media.

Willocx, M., Vossaert, J., & Naessens, V. (2015). A Quantitative Assessment of Performance in Mobile App Development Tools. In *2015 IEEE International Conference on Mobile Services* (pp. 454–461). doi:10.1109/MobServ.2015.68

Wlaschin, S. (2018). *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. O'Reilly UK Ltd.

Zibran, M. F., Eishita, F. Z., & Roy, C. K. (2011). Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering* (pp. 151–155). doi:10.1109/WCRE.2011.26

Appendix A: IUBH TOR Specification

Problem and solution

Students enrolled to distance learning courses at IUBH have access to a website within the university's "CARE portal" that allows them to get an overview of the current state of booked modules and the grades they got for them. However, at the time of writing, there is no notification system in place that would let them know about a new grade, which leads for thousands of students to the tedious task of continually logging in to the "CARE portal" and navigating to that website, often multiple times a day.

IUBH TOR is addressing this problem by providing fast and instant access to all information regarding students courses. Like their current status, grades, and other data provided by the CARE system. It will furthermore regularly check for updates and notify users about changes made to their transcripts of records, so they do not have to check manually.

Functional requirements

App Startup

As a student, I want to get presented the login dialog when I am currently not signed in (yet), so I can provide my credentials and sign in.

As a student, I want to get presented the course list dialog when I am currently signed in, so I can start using the app directly.

Dialog: Login

As a student, I want to be able to sign in to the app so I can use its features.

- The dialog provides a text field for my user name, a password field, and a submit button (M01).
- The button is only enabled when both the user name and the password are being provided.
- User name and password count as provided when both contain at least one character.
- When the authentication succeeds, I am being forwarded to the course list.
- When the authentication fails, an error message is shown (M02).

Dialog: Course List

As a student, I want to be able to sign out directly from the course list dialog, so I can make sure the app stops pulling data from CARE or sign in with another account.

As a student, I expect all my data to be deleted from the storage of the app after I signed out, so I can be sure there is no personal data left.

As a student, I expect data to be fetched from CARE immediately after I signed in to the app so I can see the latest data.

As a student, I want to be able to fetch new data from CARE through „pull to refresh” at any given time so I can manually check for updates.

As a student, I expect a loading indicator and some information text to appear while data is being fetched from CARE so I can see and understand what is happening (M03).

As a student, I expect any error that occurred while loading data to result in a visible error message whenever I started the process personally, or it got started automatically after signing in, so I can make a decision based on that error message (M04).

As a newly enrolled student, I expect some information being shown that there is no data available to display yet instead of an empty screen when there‘s no course to display for my account, so I don‘t confuse that state with an erroneous behavior of the app (M05).

As a student, I want to see all the courses I had or will have an exam for in a list (M06), including course name, exam date, exam status, and grade (if passed).

As a student, I want to see when the data from CARE has been updated for the last time, so I can decide on whether I should try to refresh myself or not.

As a student, I want to navigate to a course’s detail dialog by selecting its entry in the list of all courses, so I can see all the detailed information that is not shown on the list.

Dialog: Course Detail

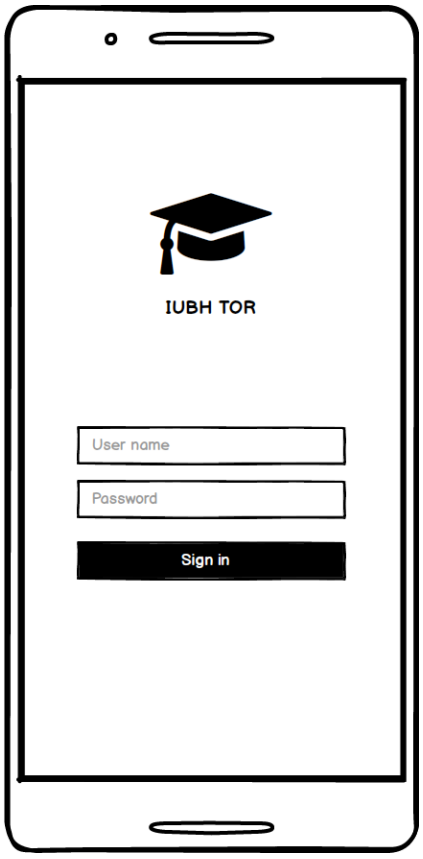
As a student, I want to see detailed information about a course on its detail page so I can see all the things that is not shown on the list (M07).

Background Synchronization and Notification

As a student, I want the app to check regularly for updates to my transcript of records and to notify me when something has changed, so I can stay up to date.

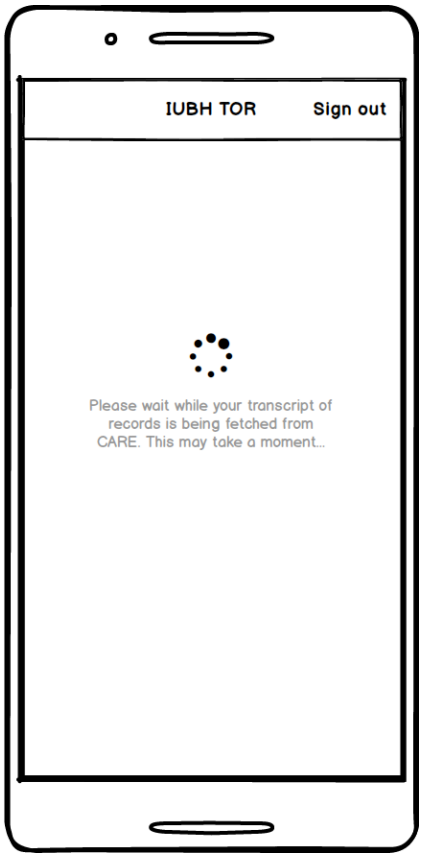
As a student, I want a system notification to be shown when something changed at my transcript of records so I can get notified about that change (M08).

Mockups

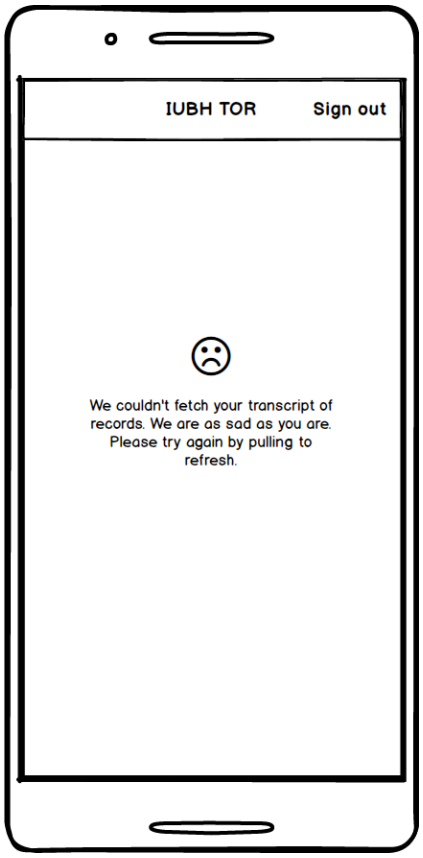


M01

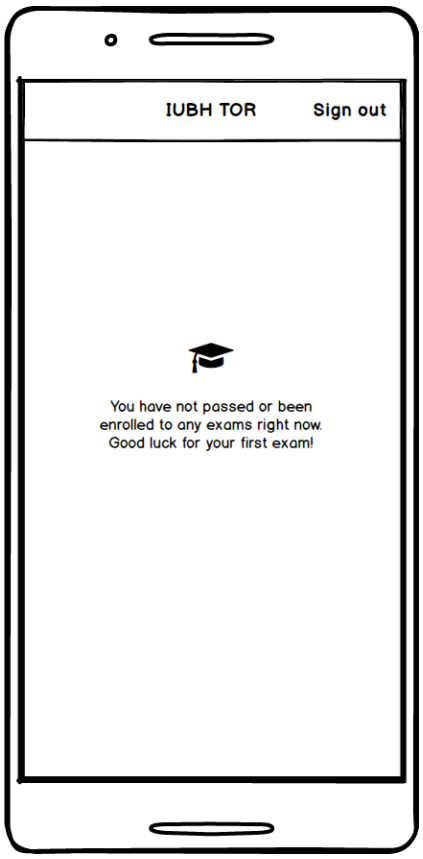
M02



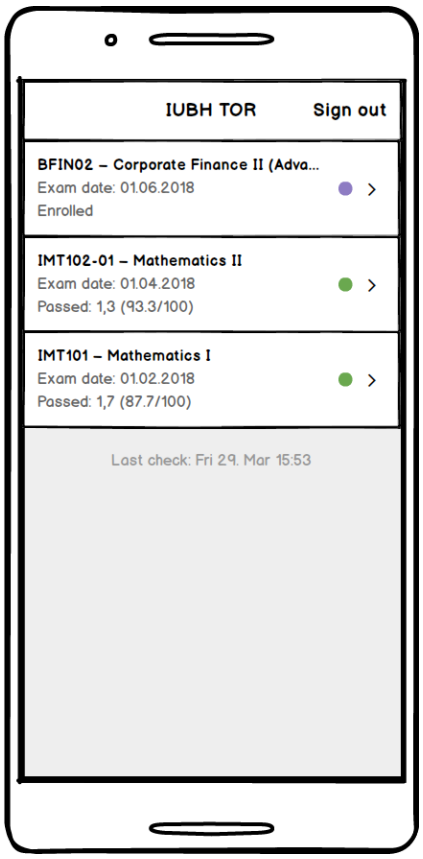
M03



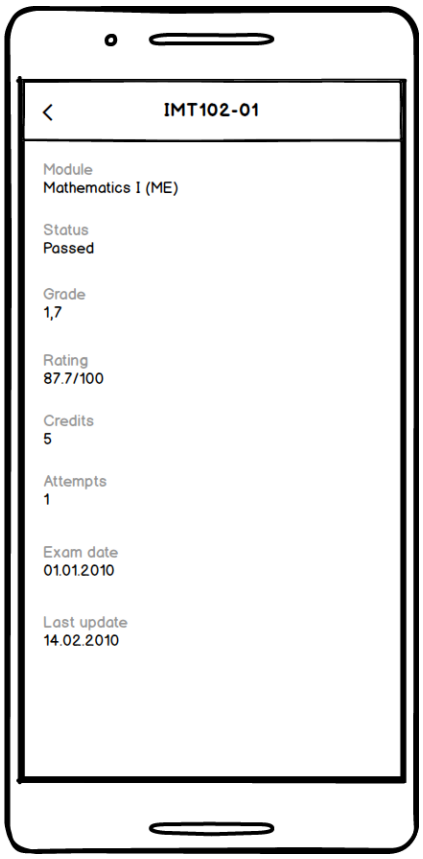
M04



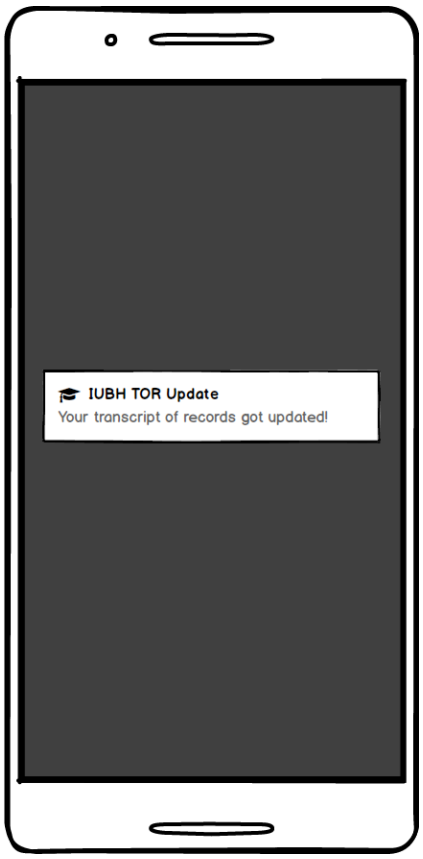
M05



M06



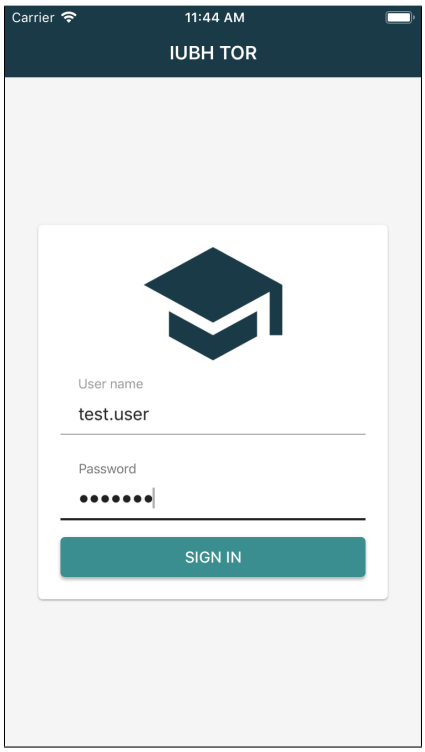
M07



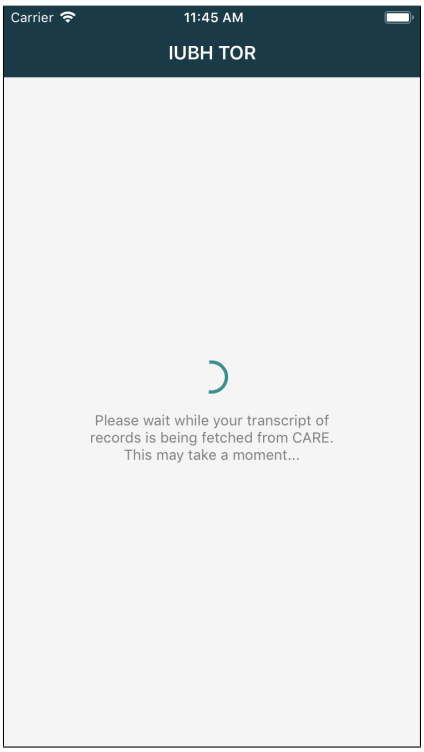
M08

Appendix B: IUBH TOR Screenshots

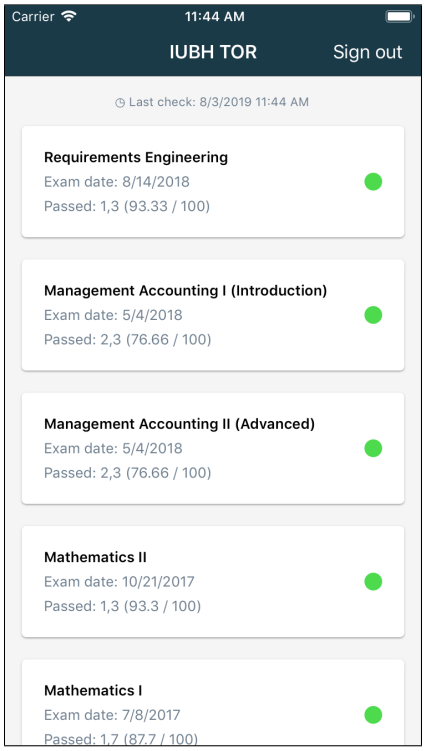
F# / iOS 12.4



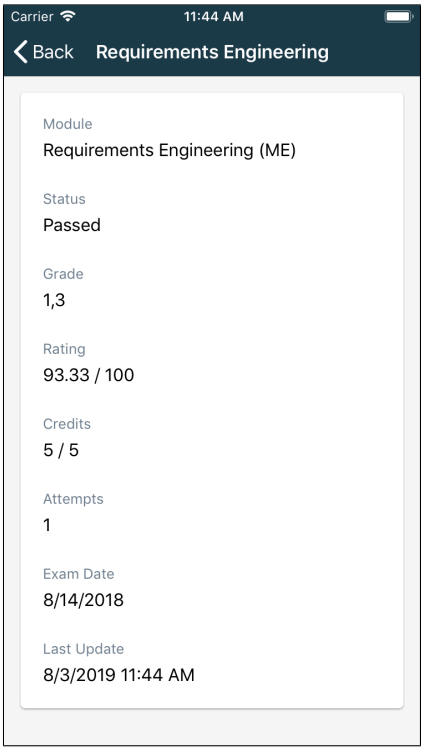
Login



Loading State

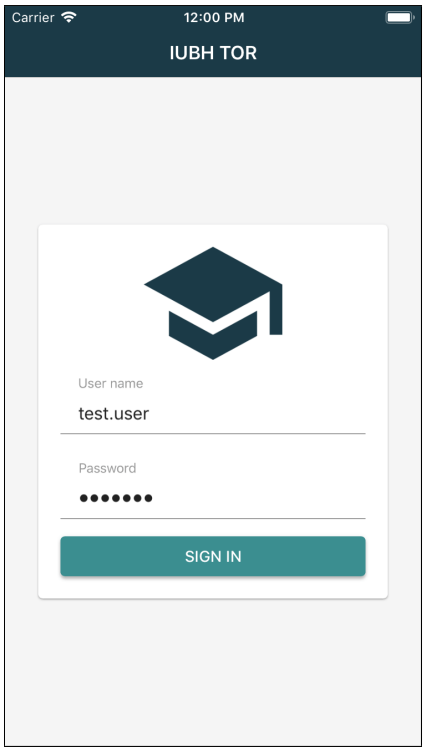


Course List

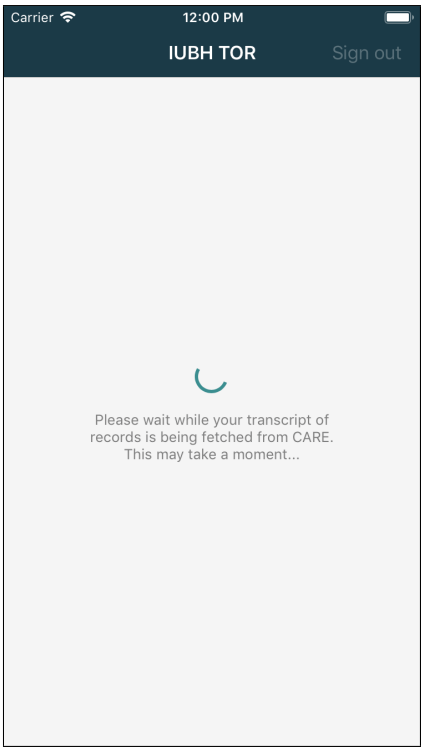


Course Detail

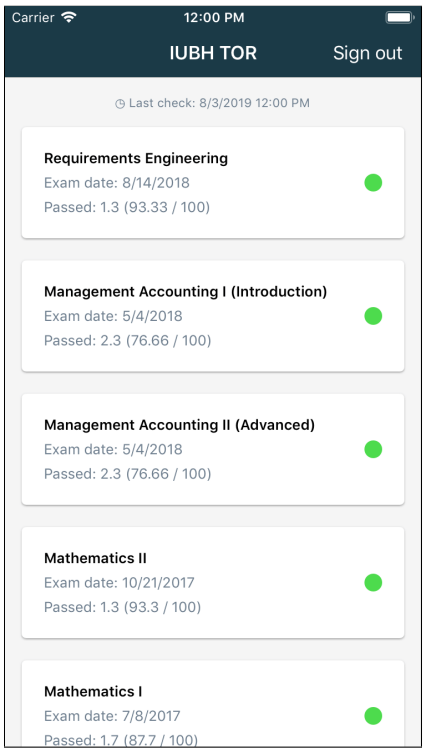
C# / iOS 12.4



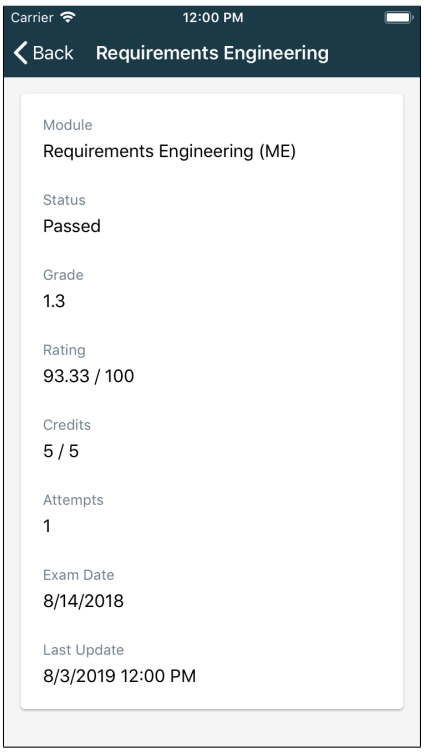
Login



Loading State

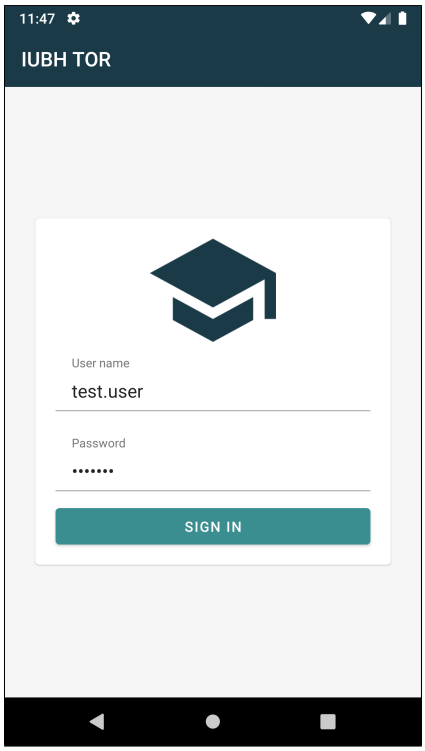


Course List

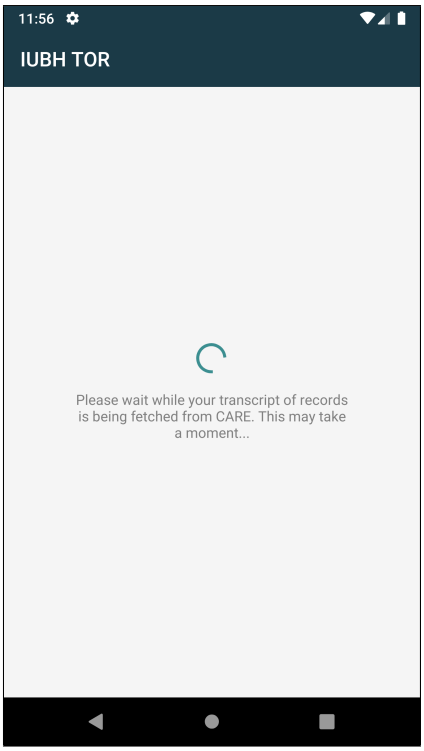


Course Detail

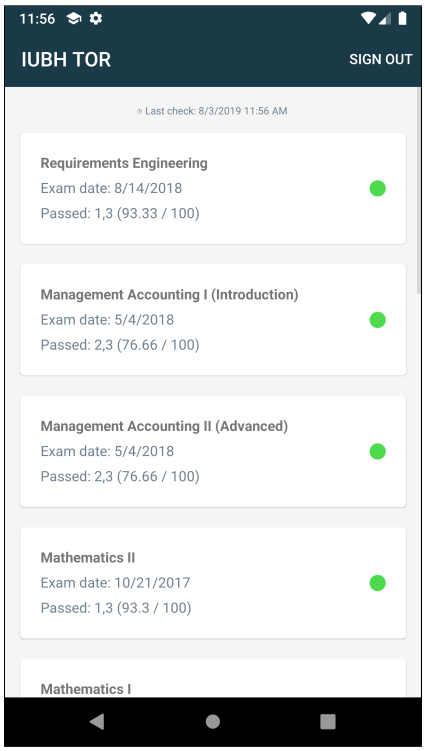
F# / Android 9



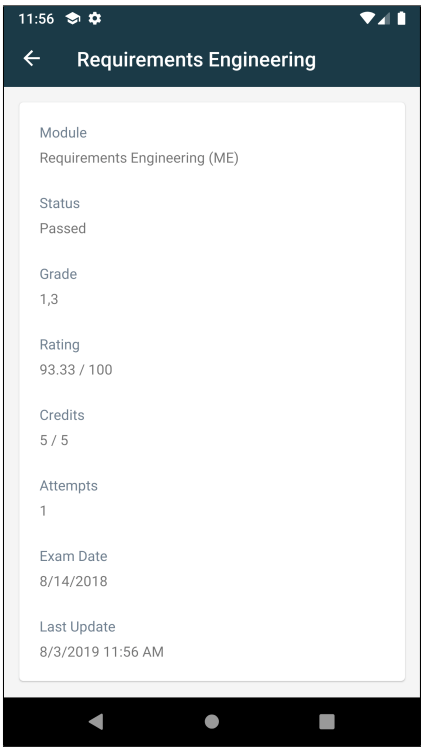
Login



Loading State

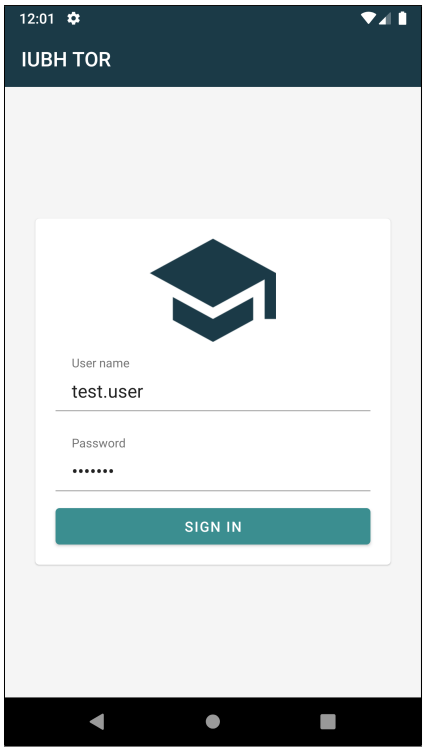


Course List

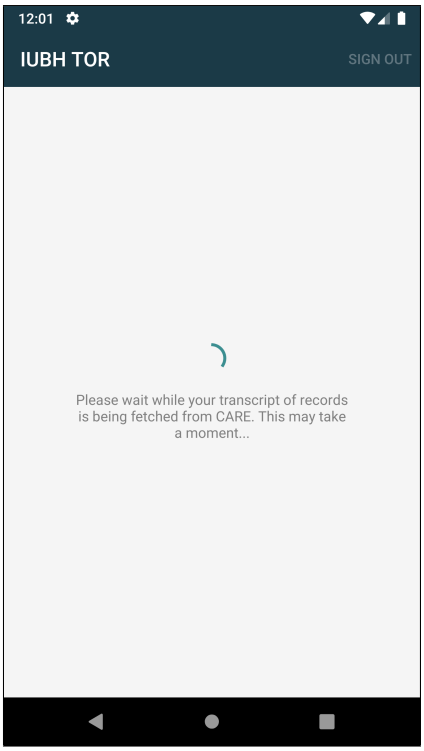


Course Detail

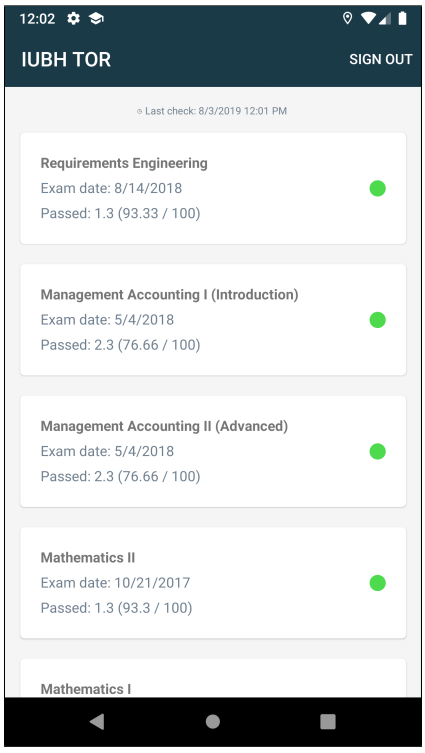
C# / Android 9



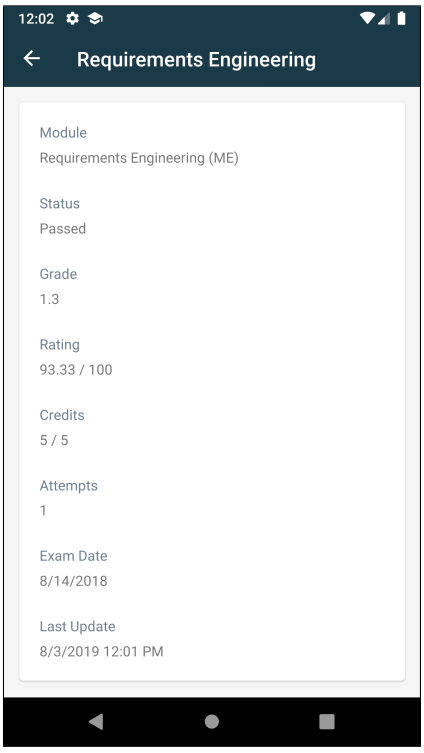
Login



Loading State



Course List



Course Detail