



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MATTI MÄÄTTÄ
REACTIVE PROGRAMMING IN IOS APPLICATION DEVELOPMENT

Master of Science thesis

Examiner: University Lecturer Terhi
Kilamo

Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 1st March 2017

ABSTRACT

MATTI MÄÄTTÄ: Reactive Programming in iOS Application Development
Tampere University of Technology
Master of Science thesis, 58 pages
September 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: University Lecturer Terhi Kilamo
Keywords: Reactive programming, Reactive extensions, Rx, iOS, Swift

Today's mobile applications are highly interactive, which means the applications must react to various events originating from the applications or their outside environment. The use of conventional sequential programming approaches, such as callbacks, make these applications difficult to implement, because having to manually orchestrate asynchronous tasks leads to programs that are complex, error-prone and hard to maintain. Reactive programming paradigm is proposed to be a more suitable approach to implement these applications.

This master's thesis examines how a reactive extension library is used to implement an iOS application utilising the reactive programming paradigm. The research is done by implementing an iOS application utilising the paradigm, and by evaluating the implementation to describe the benefits and problems of reactive programming.

The benefits of reactive programming are reduced program complexity when compared to the use of callbacks, the ability to express application logic declaratively by chaining observable sequences with operators, and data-binding enabled by reactive extensions. The problems are strong reference cycles and compiler type inference errors in the Swift programming language caused by a few common patterns, and maintainability concerns related to the complexity of the reactive extension model and its implementation differences in other programming languages.

TIIVISTELMÄ

MATTI MÄÄTTÄ: Reaktiivinen ohjelmointi iOS-sovelluskehityksessä
Tampereen teknillinen yliopisto
Diplomityö, 58 sivua
Syyskuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastaja: Yliopistolehtori Terki Kilamo
Avainsanat: Reaktiivinen ohjelmointi, Reactive extensions, Rx, iOS, Swift

Nykymobiilisovellukset ovat hyvin interaktiivisia, jonka johdosta sovellusten täytyy reagoida moniin sovellusten sisältä tai ulkomaailmasta lähtöisinä oleviin tapahtumiin. Tavanomaisten sekventiaalisten ohjelmointikäytäntöjen, kuten takaisinkutsujen, käyttö tekee edellä mainittujen sovellusten toteuttamisesta vaikeaa, koska käsin tehtävä asynkronisten tehtävien orkestrointi johtaa monimutkaisiin, virhealttiin ja hankalasti ylläpidettäviin ohjelmiin. Reaktiivisen ohjelmointiparadigman on ehdotettu olevan sopivampi tapa toteuttaa edellä mainittuja sovelluksia.

Tässä diplomityössä tutkitaan, kuinka reaktiivista laajennoskirjastoa käytetään reaktiivista ohjelmointiparadigmaa hyödyntävän iOS-sovelluksen toteuttamiseen. Tutkimus suoritetaan toteuttamalla kyseistä paradigmaa hyödyntävä iOS-sovellus, ja kuvaamalla käytettyjen reaktiivisten ohjelmointitekniikoiden hyödyt ja ongelmat toteutusta arvioimalla.

Reaktiivisen ohjelmoinnin hyödyt ovat vähentynyt ohjelman monimutkaisuus verraten takaisinkutsujen käyttöön, sovelluslogiikan kuvaaminen deklaratiiivisesti ketjutamalla tarkkailtavia sekvenssejä operaattoreilla ja tiedon sidonta käyttäen reaktiivisia laajennoksia. Ongelmat ovat vahvat viitesykliä ja kääntäjän tyyppitysten päättelyn virheet Swift-ohjelmointikielessä aiheutuen muutamasta yleisestä toimintamallista toteutuksessa, sekä ylläpidolliset huolet liittyen reaktiivisen laajennosmallin monimutkaisuuteen ja sen toteutuseroihin muissa ohjelmointikielissä.

PREFACE

I would like to thank my examiner Terhi Kilamo for providing valuable feedback during the writing process of this thesis, and my employer Vincit Oy and my project leader Anssi Kuutti for making this masters thesis possible.

Tampere, 14.9.2017

Matti Määttä

TABLE OF CONTENTS

1. Introduction	1
2. Background	3
2.1 Reactive Programming	3
2.2 Software Design Patterns	5
2.2.1 Client-Server Model	5
2.2.2 Model-View-ViewModel	6
3. iOS Application Development	8
3.1 Swift Programming Language	8
3.1.1 Closures	8
3.1.2 Extensions	9
3.1.3 Automatic Reference Counting	10
3.1.4 Type Inference	11
3.2 Software Development Kit	12
4. Reactive Programming in Swift	14
4.1 Reactive Extensions	14
4.2 Observable	14
4.3 Observer	16
4.4 Operators	17
4.5 Subject	18
4.5.1 BehaviorSubject	18
4.5.2 Variable	18
5. Application Design	20
5.1 Software Overview	20
5.1.1 REST API	21
5.1.2 Socket.IO API	22
5.2 Requirements	22
5.2.1 Registration and Login	22

5.2.2	Listing Nearby Car Wash Sites	24
5.2.3	Purchasing Washes	25
6.	Implementation	28
6.1	Model-View-ViewModel and Rx	28
6.1.1	Data and Event Binding	29
6.1.2	Navigation	31
6.2	Asynchronous Services	34
6.2.1	HTTP Client	34
6.2.2	Socket.IO Client	36
6.2.3	Location Service	39
6.3	Forms and Input Validation	40
6.4	Listing Nearby Car Wash Sites	44
6.5	Car Washer Operation	45
7.	Evaluation	48
7.1	Reactive Programming Benefits	48
7.1.1	Asynchronous Programming	48
7.1.2	Data-Binding	50
7.2	Problems	52
7.2.1	Programming Language	52
7.2.2	Maintainability	54
8.	Conclusions	55
	Bibliography	57

LIST OF ABBREVIATIONS AND SYMBOLS

GUI	Graphical User Interface
Rx	Reactive extensions
MVVM	Model-View-ViewModel
API	Application Programming Interface
REST	Representational State Transfer
SDK	Software Development Kit
ARC	Automatic Reference Counting
VPN	Virtual Private Network
ALPR	Automatic License Plate Recognition
JSON	JavaScript Object Notation
JWT	JSON Web Token
XHR	XMLHttpRequest

1. INTRODUCTION

Developing today's mobile applications requires reacting to all sorts of events originating from the applications or their outside environment. Such events include touch gesture events for reacting to user interactions and network request events for exchanging data with background services. The most interactive part of an application is usually the Graphical User Interface (GUI) layer, which needs to coordinate between multiple events originating from user interactions and internal application state changes [1, p. 52:1].

The use of conventional sequential programming approaches, such as *callbacks*, make these applications difficult to implement, because the programmer has to manually orchestrate asynchronous tasks, leading to programs that are complex, error-prone and hard to maintain [2, p. 1]. Bainomugisha et al. [1, p. 52:2] propose reactive programming paradigm as a more suitable approach to implement these applications. Reactive programming provides abstractions to express programs declaratively as reactions to external events and having the language manage the flow of time and data computation dependencies.

Reactive programming can be done in iOS application development using libraries offering abstractions for reactive programming concepts. One popular model is reactive extensions (Rx), which was first introduced by Meijer [3] for the .NET 4 platform. The model is centered around the concept of observable data sources, which can be transformed using various combinators and operators. As the generic computational model of Rx has increased in popularity, the model has been implemented for many other languages, such as *Java*, *JavaScript* and more recently *Swift* [4].

The research problem of this thesis is to examine how a reactive extension library is used to implement reactive iOS applications. The research is performed by implementing an iOS application using a reactive extension library called RxSwift [5] that implements the Observable model for the Swift programming language, and Model-View-ViewModel (MVVM) architecture pattern [6, p. 461] to model the GUI layer of the application. The application was implemented as part of a customer project at

Vincit Oy, and is called the *Superoperator* app for the iPhone family of iOS devices, featuring car wash services and account management.

The implementation objective is to show how both internal and external application events and GUI layer interactions can be transformed to observable sequences in order to express the rest of the application logic reactively. The implementation results are then evaluated subjectively and compared to conventional callback-based asynchronous models to understand how the use of reactive programming benefits iOS application development. Problems discovered during the implementation related to the used reactive extension library and the Swift programming language are also described as part of the evaluation.

The second chapter presents the theoretical definition of the reactive programming paradigm, and software design patterns utilised in the application implementation. The third chapter introduces the key parts of iOS Software Development Kit (SDK), including the Swift programming language to describe the used tools in iOS application development. The relevant parts of the Swift programming language are described in detail as background to code examples presented in the thesis and to give reasoning behind problems discovered during the application implementation related to the programming language itself. The fourth chapter introduces the RxSwift reactive extension library and describes the Observable model it implements. The fifth chapter describes the design of the Superoperator app, the used background services, and the app's requirements. The sixth chapter describes how the MVVM architecture pattern is used in the application and how the application requirements are fulfilled by implementing application logic reactively using Observables. The seventh chapter evaluates the implementation and shows the key benefits and problems related to the proposed reactive programming approach described in the implementation chapter. The last chapter summarises the results presented in the implementation and evaluation chapters, and proposes further research opportunities.

2. BACKGROUND

This chapter explains the key concepts of the reactive programming paradigm, and the software design patterns utilised in the application implementation. The definition of the paradigm gives the background necessary to understand the behaviour and propagation of change in reactive programs. The design patterns include the client-server model, which describes how the application is separated to client and server components and the Model-View-ViewModel, which is used as an architectural pattern in the GUI layer.

2.1 Reactive Programming

Reactive programming paradigm is oriented around asynchronous data streams and propagation of change. An asynchronous data stream can be described as a continuous stream of time-varying values. Reactive programming facilitates declarative development of event-driven applications by allowing developers to express application logic in terms of what to do instead of how to do it [1, p. 52:3].

Propagation of change can be explained with the following program:

```
a = 1
b = 2
c = a + b
```

In imperative sequential programming the variable `c` would always have value 3 which is the sum of initial values of variables `a` and `b`, even when value of `a` or `b` changes after evaluation of `c`. However, in reactive programming the value of `c` would be always kept up to date because `c` would get recomputed every time `a` or `b` is changed. In reactive programming terminology variable `c` can be said to be dependant of variables `a` and `b` through the addition operator, which is illustrated in the following dependency graph shown in Figure 2.1. [1, p. 52:3]

Evaluation model describes how changes are propagated in a reactive program across

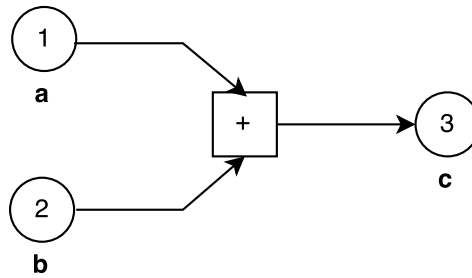


Figure 2.1 Expression dependencies in a reactive program. A line's arrow indicates the direction of data flow.

dependencies, as shown in the dependency graph in Figure 2.1. From a programmer's perspective, changes are propagated transparently but on programming language or a reactive extension library level a decision needs to be made on who initiates the propagation of change. In this section the two evaluation models found in reactive programming literature are described: push and pull [1, p. 52:6]. Push- and pull-based evaluation models are shown in Figure 2.2.

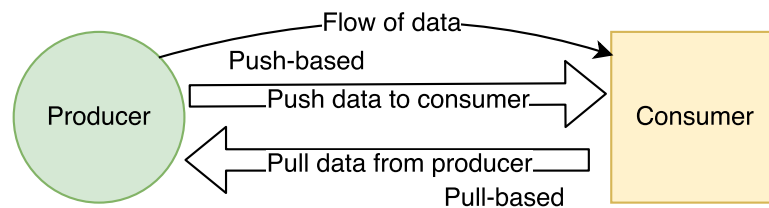


Figure 2.2 Push- and pull-based evaluation models.

In the *pull-based* evaluation model, the computation (consumer) that requires a value needs to request it from the producer as shown in Figure 2.2. The propagation is driven by demand for new data by the computation and is called *demand-driven* propagation [1, p. 52:6]. The pull model can *lazily* pull new values when it needs them. One major criticism of pull-based models is that it may result in significant latency between an event occurrence and reaction [1, p. 52:6].

In the *push-based* evaluation model the producer pushes data to its dependant computations (consumers) when new data is available. The propagation is driven by availability of new data and is called *data-driven* propagation [1, p. 52:6]. The push-based model is the most commonly used implementation according to Bainumugisha et al. [1, p. 52:6]. Implementations of the push-based model need to handle wasteful recomputations efficiently since recomputations happens each time new data is available.

While the pull-based evaluation model has advantages in applications where sampling is done on time-varying values, the push-based model is more appropriate for

applications that require instantaneous reactions [1, p. 52:7].

2.2 Software Design Patterns

Software design patterns are general solutions to commonly occurring problems in software development. In this section two general patterns used in the application implementation are presented: client-server model and Model-View-ViewModel. The client-server model describes how the application utilises background services to send and receive data, and the MVVM pattern describes the architecture of the application's GUI layer.

2.2.1 Client-Server Model

The Superoperator app exchanges data with a background service over the internet to implement car wash and account management features. The most frequently used architecture style to implement such network-based applications is the *client-server* style [7, p. 45]. The client-server architecture can be described with the following figure:

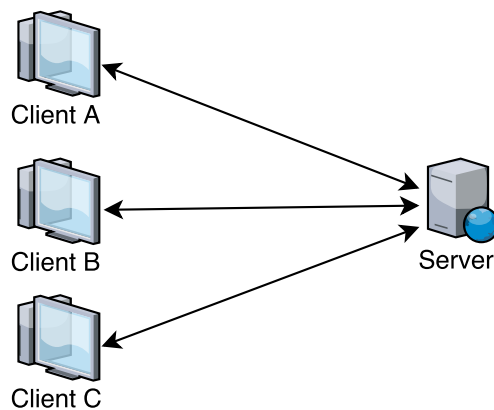


Figure 2.3 Client-server model.

Figure 2.3 shows three *clients* connected to a *server*. The server component provides a set of services and listens for incoming requests over a *connector*. The client initiates a request and sends it to the server. The server can either reject or perform the request and return with a response back to the client.

Separation of concerns is the principle behind the client-server constraints, where proper separation simplifies the server component in order to improve scalability [7, p. 45]. By moving all the user interface functionality into the client component, the server component can evolve independently provided the interface does not change.

Backwards compatible interfaces between the client and server components are required in the Superoperator app in order to distribute server updates without breaking existing client functionality. Respectively, user interface features can be updated or changed without having to update the server when user interface functionality is built into the client.

The problem with the client-server separation in native mobile applications is that neither Apple App Store or Google Play marketplaces enforce critical updates of distributed apps [8]. This may leave some users affected by security vulnerabilities or bugs even though an update fixing the issue is distributed through the marketplace. The client-server interface also has to stay backwards compatible when updating the server component, otherwise old client versions that are still being used may break.

2.2.2 Model-View-ViewModel

Model-View-ViewModel architecture [6, p. 461] focuses on the separation of view's logic and presentation. Instead of defining a view's look and behaviour (the code which updates the view's state) in the same class, a separate *view model* class implements the view's behaviour. The view model then exposes data and operations to the view or views who consume it. This results in view's code consisting mostly of *data-binding* glue code and presentation logic. The following figure shows the relationship between the view, view model and model:

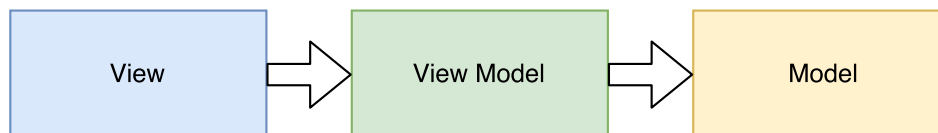


Figure 2.4 Relationships between the View, ViewModel and Model.

The *View* (as shown in Figure 2.4) contains presentation logic and glue code to consume view model's data and operations, such as touch events. As per definition, the view's logic and behaviour should be placed entirely in the view model, however in practice this is often an unrealistic goal [6, p. 462]. For example tasks that require direct interaction with the view would require complex logic to be implemented in the view model, decreasing maintainability.

The *Model* usually contains data and business logic that is in some form presented in the view [6, p. 465]. A model can for instance represent data returned by a Representational State Transfer (REST) [7, p. 76] API call or be some internal data component of the application. Models are usually retrieved from a domain service [6, p. 465] instead of constructed directly in the view model.

The *ViewModel* maintains both the state of the view and handles the view's behaviour [6, p. 465]. The view model exposes data from the model to the view either directly or by mapping it to a view-consumable format. For example, view model could describe the visibility state of a component as a boolean data type that is computed from some model property, such as a value indicating whether the model data is loaded from the server.

A benefit of the MVVM architecture is that a view model can be bound to multiple types of views that consume the same interface, for example to provide a different layout for portrait and landscape device orientations. This allows applications to have different presentation logic for different screens without duplicating view's behaviour. View models can also be *unit tested* separately from view presentation logic [6, p. 465].

3. IOS APPLICATION DEVELOPMENT

iOS is the Apple Inc.'s operating system running on iPhone and iPad mobile devices. The operating system provides the technologies needed to develop native applications using the iOS Software Development Kit (SDK) [9]. The SDK contains the tools necessary to develop, run, test and install native iOS applications. Native applications can be programmed using either Objective-C [10] or Swift [11] programming languages. In this chapter the Swift programming language is introduced along with the key SDK components used in the application implementation. The chapter also goes into detail of a few Swift programming language features to give background necessary to understand code examples presented in the later chapters and to understand the problems related to the Swift programming language described in the evaluation chapter.

3.1 Swift Programming Language

Swift is Apple Inc.'s latest¹ programming language for developing native macOS, iOS, watchOS and tvOS applications, building upon the best features of Objective-C and C [11]. Swift is a procedural and object-oriented language that adopts new modern language features such as optional types, tuples, type safety and generics. The used reactive extension library relies on use of *closures*, *extensions*, Automatic Reference Counting (ARC) and *type inference*. These language features are described in detail to understand code examples shown in the later chapters and to explain issues when these features are used in particular usage patterns.

3.1.1 Closures

Closures provide functionality to capture and store references to any constants and variables from the environment in which the closure is defined. Closures in Swift are similar to *blocks* used in Objective-C, or more commonly *lambdas* in other programming languages [11]. For example, Listing 1 shows the simplest form of a closure that can capture values.

¹At the time of writing.

```
1  func makeIncrementer(startingAt start: Int) -> () -> Int {
2      var total = start
3      return {
4          total += 1
5          return total
6      }
7  }
```

Listing 1 Function containing a nested function.

Listing 1 shows a function `makeIncrementer(startingAt:)` which contains a nested function. The nested function captures two values: `start` and `total` from its lexical context. The value `total` is initialised to value of `start`. The nested function is then returned as a closure that increments `total` by 1 each time it's called.

The return type of `makeIncrementer(startingAt:)` is `() -> Int` that is a function taking no parameters and returning an integer value. The following example shows the closure in action:

```
1  let increment = makeIncrementer(startingAt: 2)
2  increment()
3  // returns a value of 3
4  increment()
5  // returns a value of 4
```

The example declares a constant `increment` referring to a function that returns increasing values starting from the value 2. Capturing references to the variables `start` and `total` shown in Listing 1 allows the value of `total` to be incremented by subsequent `increment` calls.

3.1.2 Extensions

Extensions enable adding new functionality, such as methods, to existing types. Extensible types in Swift include the *class*, *structure*, *enumeration*, and *protocol* types [11]. Swift also allows extending types for which the source code is not available.

Extensions can add both *computed* instance and type properties to existing types [11]. The following example shows how the `String` type is extended:

```
1  extension String {
2      var greeting: String {
```

```
3     return "Hi, " + self
4   }
5 }
6
7 "Matt".greeting
8 // returns "Hi, Matt"
```

The example shows the native `String` type being extended with a computed instance property `greeting`. Computed properties are read-only and are expressed with a code block returning a value. In the example the computed property prepends the string `"Hi, "` to the value of `self`.

3.1.3 Automatic Reference Counting

Automatic reference counting (ARC) is a type of *garbage collection* approach used in the Swift programming language [11]. A *reference counted* garbage collector keeps count of references to each object and reclaims the memory when the reference count falls to zero [12, p. 73]. By contrast, a *tracing* garbage collector relies on a set of *roots* to keep track of references to objects.

In ARC, the compiler inserts memory management instructions, freeing the programmer from explicitly calling *retain* and *release* [13]. The following example shows automatic reference counting in action:

```
1 class Greeter {
2   init() {
3     print("Hello!")
4   }
5   deinit {
6     print("Goodbye!")
7   }
8 }
9
10 var reference1: Greeter?
11 var reference2: Greeter?
12
13 reference1 = Greeter() // prints Hello!
14 reference2 = reference1
```

```
15
16 reference1 = nil
17 reference2 = nil // prints Goodbye!
```

The class `Greeter` has both an initialiser and deinitialiser which print a message when the class is being initialised and deinitialised. On line 13, an instance of `Greeter` is initialised and set to variable `reference1`, printing the message *Hello!*. Then, a new variable named `reference2` is set to the value of `reference1`, so both variables reference the same object. Now, `reference1` can be set to `nil` on line 16 without causing the object to be deallocated because `reference2` is still keeping the object's reference count at one. Only when `reference2` is set to `nil` on line 17, the reference count falls to zero and the message *Goodbye!* is printed from the deinitialiser.

It is possible to write code where two class instances reference each other, causing the objects' reference counts to never fall to zero. This is known as a *strong reference cycle* [11]. Swift can resolve strong reference cycles by using *weak* and *unowned* references.

A *weak* reference does not cause the referenced object to be retained [13], preventing the reference from becoming part of a strong reference cycle. Because a weak reference does not retain references to an object, it is possible for the object to become deallocated while the variable is still referencing it. Therefore, Swift represents weak references as *optional types* because ARC sets the reference to `nil` when the referred object is deallocated [11].

An *unowned* reference behaves like a weak reference, but does not set the reference to `nil` when the referenced object is deallocated [11]. This means that unowned references are represented as non-optional types, but a *runtime error* is raised if a deallocated object is accessed through an unowned reference.

3.1.4 Type Inference

Type inference allows types of expressions to be deduced during compile time. Swift uses type inference extensively, allowing the programmer to omit type annotations when the type of expression can be deduced from its context [11]. For example, Swift allows omitting the type annotation of expression `let x: Int = 10` by writing `let x = 10`, where the compiler infers the type `Int` for `x`.

Type inference also works for complex expressions, for example:

```

1 let basket = ["eggs": 6]
2 type(of: basket) // returns Dictionary<String, Int>.Type

```

where the type of `basket` is inferred from the dictionary initialiser. Type information is passed up towards the root of the expression tree [11], for example in case of the above example, the type of `x` is inferred by first determining the type of `10` and passing that information up to the variable `x`.

Type information can also flow towards the opposite direction, specifying the type [11]. For example the expression `let pi = 3.14` would infer `Double` for the type of `pi`, however the explicit type annotation `let y: Float = 3.14` would infer `Float` for the type of `y`.

3.2 Software Development Kit

The iOS Software Development Kit consist of the tools and Application Programming Interfaces (APIs) required to develop, test and deploy applications for the iOS operating system [9]. Figure 3.1 shows how the APIs are divided in four layers. The lower layers contain fundamental services, such as the CFNetwork framework, providing network protocol abstractions. The higher-level layers build on the lower level interfaces and provide more higher level abstractions for the developers.

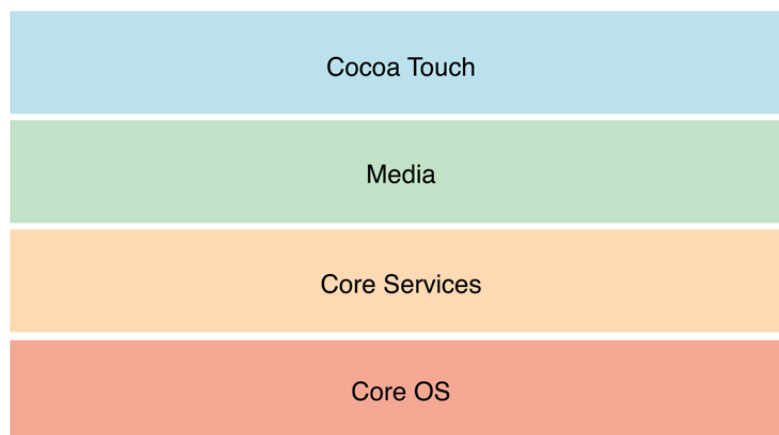


Figure 3.1 Layers of iOS. [9]

The *Cocoa Touch* layer (as shown in Figure 3.1) contains the highest-level interfaces, such as the UIKit framework that is used to develop graphical and event-driven applications [9]. The layer also contains other high-level features such as *auto layout*, *multitasking*, TextKit and *storyboards*.

The UIKit framework contains the necessary interfaces required to develop iOS applications [14]. The framework provides the *window* and *view* architecture to

implement graphical user interfaces, the event-handling infrastructure to respond to user input events such as touch gestures, and the *app* model needed for the main run loop and system interaction. To implement graphical user interfaces, UIKit provides a *view controller* abstraction that is implemented by the `UIView` and `UIViewController` classes.

The `UIView` class provides interfaces for managing its contents on a rectangular area on the screen [14]. Instances of the class handle rendering and user interaction in the view's area that is known as *frame*. Views can also embed other views, forming parent-child relationships. An example of the parent-child relationship is an `UIButton` view that embeds an `UILabel` view for rendering text inside the button's frame.

The `UIViewController` class manages a set of views that form a portion of the application's user interface [14]. The view controller is responsible for updating content of its views, including loading and disposing of said views. The view controller also responds to user interactions within its views and handles the layout and size of the *root view*, where the view controller's child views are embedded.

Event handling in UIKit is implemented using `UIResponder` objects [14]. Many key components of UIKit are responder objects, such as the `UIView` and the `UIViewController`. When an event such as touch event happens, UIKit forwards it to the application's responder objects for handling. Responder objects can also form a chain, for example an `UIView` in an `UIViewController`. A responder object has a choice to handle the event, or pass it forward to a *parent* responder object.

4. REACTIVE PROGRAMMING IN SWIFT

Reactive programming can be done in Swift using third party libraries offering reactive programming abstractions. RxSwift library [5] is an implementation of reactive extensions known as Rx [15] that enables easy composition of asynchronous event and data sequences with operators. Alternative reactive libraries exist, such as ReactiveSwift [16], however RxSwift is chosen for the application implementation because the generic computation model of Rx is also implemented for many other languages, such as *JavaScript*, *Java*, *Python* and *C#* [4]. Understanding the generic computation model allows the same reactive programming patterns to be used in projects implemented in other programming languages.

4.1 Reactive Extensions

Reactive extensions (Rx) is a library for composing asynchronous event and data streams using observable sequences for the .NET platform [15]. Due to the gained popularity, the library's generic computation model has been implemented for multiple different platforms, such as RxSwift for iOS and macOS environments [4].

Reactive extensions extend the *Observer pattern* [17, p. 293] to provide an abstraction for asynchronous event and data sequences known as the *Observable* along with operators for combining and transforming those sequences in a declarative manner [4].

4.2 Observable

The Observable model in Rx provides an abstraction for asynchronous streams of events and data that can be transformed with simple and composable operators [4]. The model is explained by comparing the Observable model to the synchronous *Iterator* [17, p. 257] pattern. Table 4.1 shows how Observables provide access to asynchronous sequence of multiple items.

Table 4.1 shows single and multiple item access patterns in synchronous and asynchronous execution models. In the synchronous model a single item can be accessed

	Single item	Multiple items
Synchronous	<code>var data: T</code>	<code>var data: Iterable<T></code>
Asynchronous	<code>var data: Future<T></code>	<code>var data: Observable<T></code>

Table 4.1 Observables allow easy access to asynchronous sequence of multiple items. [4]

as is, and multiple items can be accessed using the *Iterator pattern*. In the asynchronous model a single item can be accessed using a *Future* [18], and the *Observable pattern* provides the missing abstraction to access asynchronous sequences of multiple items.

Futures add non-trivial complexity when they are nested or used to compose conditional asynchronous execution flows. Observables on the other hand are intended for composing asynchronous sequences of data [4]. The following table shows how Observables implement the same typical conditions as the Iterator pattern:

Event	Iterator (pull)	Observable (push)
Retrieve data	<code>next() -> T?</code>	<code>onNext(T)</code>
Discover error	<code>throw Error</code>	<code>onError(Error)</code>
Complete	<code>next() == nil</code>	<code>onCompleted()</code>

Table 4.2 Synchronous Iterator (pull) and asynchronous Observable (push) sequence. [4]

Table 4.2 shows typical events related to accessing finite streams of data using both Iterator and Observable patterns. The Observable pattern extends the Observer pattern by adding two missing semantics that exist in the Iterator pattern [4]. First is the `onCompleted` method that allows the producer to signal the consumer that there is no more data available. Second is the `onError` method that allows the producer to signal the consumer that an error has occurred.

Observable in RxSwift uses the *push* evaluation model, where the producer pushes new values to consumers when new values are available. By contract, in the Iterator pattern new values are *pulled* from the producer, blocking the calling thread until new values are available. To compare both patterns further, the following Table 4.3 shows how similar high-order functions can be used in both Iterator and Observable patterns.

Observables can be divided in two categories based on *when* they start emitting values [4]. A *hot* Observable starts emitting values as soon as it is created, so when an observer subscribes to a hot Observable it may start observing the sequence somewhere in the middle. A *cold* Observable on the other hand starts emitting values once an observer subscribes to it, so the observer sees the entire sequence emitted by the Observable.

Iterator	Observable
<pre> getStuffSync() .suffix(from: 10) .prefix(5) .map { "transformed: \(\$0)" } .forEach { print("next: \(\$0)") } </pre>	<pre> getStuffAsync() .skip(10) .take(5) .map { "transformed: \(\$0)" } .subscribe(onNext: { print("next: \(\$0)") }) </pre>

Table 4.3 Similar high-order functions can be used in both Iterator and Observable patterns.

4.3 Observer

In RxSwift an observer *subscribes* to an Observable [4]. The observer reacts to events or items emitted by the Observable. An observer can be *unsubscribed* in RxSwift using a Disposable that is returned when the observer subscribes to an Observable. In the following example, an ordinary synchronous method call is shown which is later compared to an asynchronous example using Observables:

```

1 do {
2   let result = try fetchResult()
3   // do something with result
4 } catch let error {
5   // handle error
6 }

```

The return value of `fetchResult()` is assigned to constant `result`. Since the method can throw an error, the expression is contained in a `do - catch` block. Error handling is done in the `catch` block and the result can be accessed in the `do` block. The same program is implemented using Observable in the following way:

```

1 let observable = fetchResult()
2 let disposable = observable.subscribe { event in
3   switch event {
4     case .next(let result):
5       // do something with result
6     case .error(let error):

```

```
7     // handle error
8     case .completed:
9         // handle stream completion
10    }
11 }
```

The method `fetchResult()` now returns an `Observable` instead of an end result like in the synchronous example. Observer connects to `observable` by calling `Observable`'s `subscribe:` method. In the above example the observer is a *closure* that accepts a single `event` argument. In RxSwift, `event` is a generic *enumeration* type, which can be accessed using the `switch` expression [5]. The enumeration has three cases where `next` represents a new value emitted by the `Observable`, `error` stream termination to an error and `completed` stream completion after there are no more items and no error has occurred. RxSwift `Observable` also has `subscribe(onNext:)` method, accepting callbacks to resolved enumeration values as shown in Table 4.3.

`Disposable` returned by the `subscribe:` method can be used to unsubscribe from the `Observable`. The `Observable` can then choose to stop emitting items if there are no more remaining observers. RxSwift also implements `DisposeBag` which collects multiple disposables that are automatically disposed when the instance is deinitialised by ARC [5].

4.4 Operators

`Observables` can be declaratively composed together with *operators* while abstracting away concerns such as low-level threading, synchronisation, concurrent data-structures and asynchronous input and output [4]. Operator chaining is demonstrated in Table 4.3.

Operators can be used to transform `Observables` to another `Observable` types [4]. One such operator is called *map* that transforms items emitted by an `Observable` by applying a function to each item. Another commonly used operator is the *flatMap* operator that transforms items emitted by an `Observable` into `Observables`, that are then flattened into a single `Observable`. *flatMap* can be used for example to perform asynchronous tasks in the middle of an operator chain.

4.5 Subject

`Subject` is a proxy object that is both an *observer* and an `Observable`. Subjects can be observed by one or more `Observables` and pass through the events it receives [4]. This section introduces the `BehaviorSubject` and `Variable` specialisations of `Subject`.

4.5.1 BehaviorSubject

`BehaviorSubject` emits the last value emitted by the source `Observable` and all subsequent values following an observer's subscription [4]. The following examples shows a `BehaviorSubject` in action:

```
1 let subject = BehaviorSubject(value: "")
2 subject.on(.next("Hello"))
3
4 subject.subscribe(onNext: { value
5     print(value)
6 }) // prints "Hello"
7
8 subject.on(.next("World!")) // prints "World!"
```

An instance of `BehaviorSubject` is set to constant `subject`. Because the subject always emits the previous value on subscription, it is initialised to an empty string passed to the constructor. Events are emitted using the `on:` instance method, accepting an event enumeration instance. The value `"Hello"` emitted by the subject is not observed immediately, but once the observer subscribes to the subject, the last value `"Hello"` is repeated to the observer on subscription. The value `"World!"` is immediately observed by the observer because it is emitted after the observer is subscribed to the subject. Because `BehaviorSubject` emits only the previous and subsequent items following a subscription, it can be considered a *hot* `Observable`.

4.5.2 Variable

`Variable` is an RxSwift utility construct that wraps a `BehaviorSubject` [5]. `Variable` does not expose the subject in its public interface, but instead has a `value` property that can be used to emit a new value and get the last value of the subject. This essentially hides the `on:` method of the subject, so the subject can not be terminated from the outside. The following example shows a `Variable` in action:

```
1 let variable = Variable("Hello")
2
3 variable.asObservable()
4   .subscribe(onNext: { value
5     print(value)
6   }) // prints "Hello"
7
8 variable.value = "World!" // prints "World!"
```

A `Variable` is constructed with initial value "Hello". As the observer subscribes to the `Observable` returned by `variable.asObservable()`, the initial value "Hello" is observed by the observer. When the value changes to "World" it is observed by the observer just like in the `BehaviorSubject` example.

When a `Variable` is deinitialised, it will complete the `Observable` returned by the `asObservable()` method.

5. APPLICATION DESIGN

This chapter describes Superoperator iPhone application requirements and the background services which the application uses. First, an overview of the system in question is shown with the used APIs. Last, the key requirements of the application are presented.

5.1 Software Overview

The Superoperator iPhone application is a client application for customer registration, account management and car wash purchasing. The application is part the Superoperator car wash system which provides services for remote car wash automation, resource management and payments. The system has two public interfaces that are consumed by the application: Representational State Transfer (REST) [7] API and Socket.IO [19] API. A high-level overview of the system is shown in Figure 5.1.

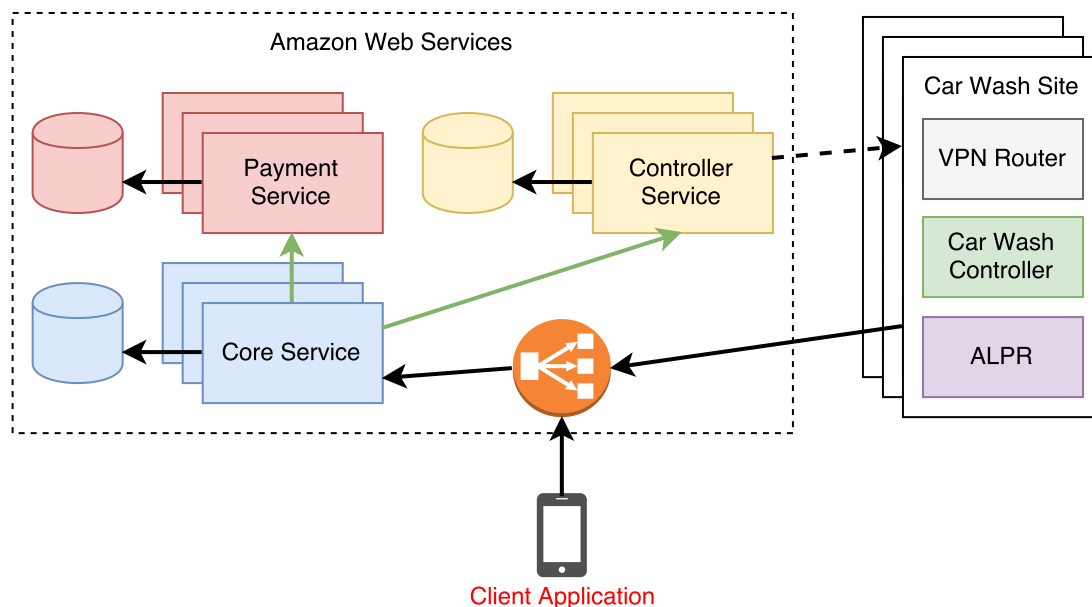


Figure 5.1 High-level overview of Superoperator car wash system.

Figure 5.1 shows a high-level overview of the Superoperator car wash system. The Superoperator iPhone application described in this chapter is represented in the

figure as *Client Application* highlighted in red text. The figure shows where each service is physically deployed in the system. Green arrows represent a *load-balanced* [20] connection between two services and a dashed line represents a *Virtual Private Network* (VPN) [21] connection.

The client application communicates with *Core Service* APIs via a load balancer (shown as orange). The APIs include REST API for resource access and Socket.IO API for real-time event delivery. Core Service is hosted in *Amazon Web Services* (AWS) [22] cloud platform, which also houses two internal services (*Payment* and *Controller*) and a database instances for each service.

Core Service handles authentication, access control and task delegation between the two internal services. Core Service APIs are the only APIs accessible for the client application.

Payment Service implements support for various payment service providers and handles tasks related to subscription payments and invoicing. Payment service is abstracted away from the client application as requests related to purchases are delegated through the Core Service.

Controller Service is responsible for observing car washer equipments' states and starting washes. The service communicates with car wash sites over VPN connection for additional security. Washer state changes are propagated through Core Service Socket.IO API to listening client applications, so the applications can notify users in real-time when a car washer becomes for example available or unavailable.

Each *Car Wash Site* has a VPN router for establishing a private connection to services running on AWS, a *Car Wash Controller* service implementing protocols for different car washers and an Automatic License Plate Recognition (ALPR) service for recognising vehicles arriving to the car washer. The site's services are configured over VPN connection as shown with the dotted line in Figure 5.1. Once configured, the site sends back events to Controller Service through the load balancer when a vehicle is recognised or the car washer state changes.

5.1.1 REST API

The application consumes the public REST API provided by the Core Service. The API enables create, read and update access to static resources such as user data, car wash site data and equipment data. The service uses JavaScript Object Notation (JSON) [23] data interchange format in both REST and Socket.IO APIs.

Access right to resources is determined by a JSON Web Token (JWT) [24] that is transported in the *X-Auth-Token* header in HTTP requests. The system generates a JWT token for response to login request made by the client application. The token contains fields such as *email*, *user identifier* and *customer identifier* to determine the principal of the request. Because the payload contains fields from database that may be changed for example when customer information is edited, the JWT token is expired and refreshed periodically. Requests made with an expired token result in a 401 HTTP status code, indicating authorisation failure.

5.1.2 Socket.IO API

The application also uses Socket.IO API [19] provided by the Core Service to receive real-time updates to resources, such as car wash equipment state changes and license plate recognitions. The Socket.IO protocol implements a WebSocket transport, allowing the client to receive events without polling the REST API. Frequent polling would consume additional resources and bandwidth by the Core Service, since each incoming request needs to be authenticated and authorised, while less frequent polling would cause too much delay to observed state changes.

In order to receive car wash equipment state changes and recognition updates, the client sends a `subscribe-equipment` message with `equipmentId` parameter. The parameter is the unique database identifier of the equipment of interest. The server will then respond with `equipment-state-changed` messages with the provided `equipmentId` and equipment data when the resource is updated, until cancelled by sending an `unsubscribe-equipment` message.

5.2 Requirements

The application requirements include customer registration and login, displaying a list nearby of car wash sites and purchasing washes for the listed car wash sites. The requirements are chosen because they depict common scenarios for interactive mobile applications requiring asynchronous event handling, such as touch input, network requests and user location handling.

5.2.1 Registration and Login

The first requirement is the ability to register a new account and login with an already registered email address. The application features *single sign-on* functionality, where the user does not have to login each time he or she uses the application.

Screens shown in Figure 5.2 depict the login use case while screens in Figure 5.3 depict the registration use case.

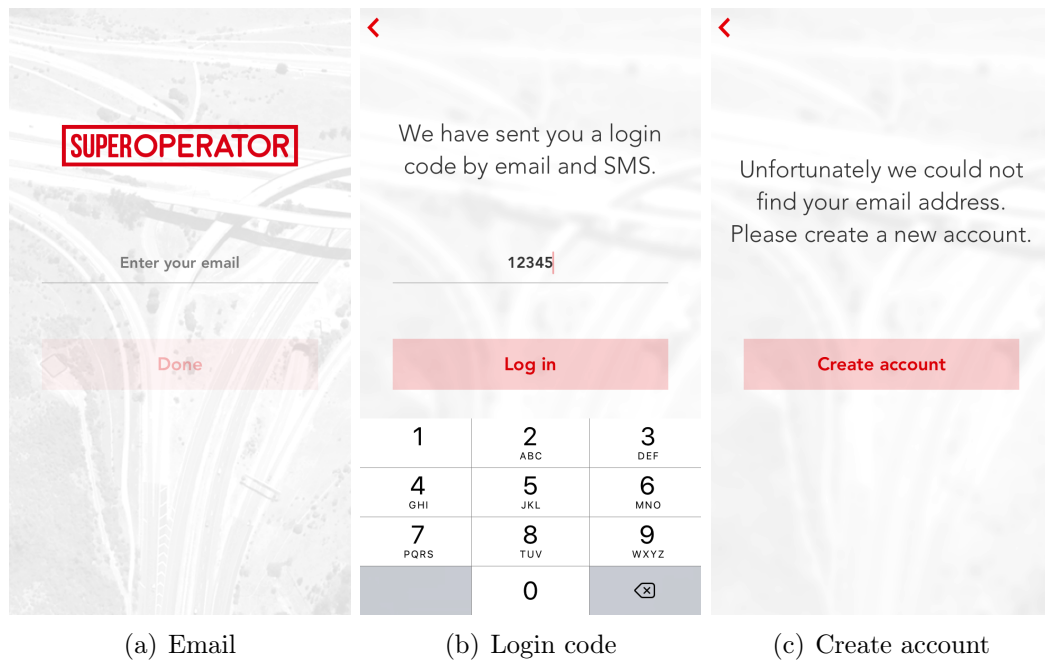


Figure 5.2 Login screens.

The application starts by determining whether the user has previously logged in to the application. If the user is logged in, then the list of nearby car wash sites is presented. Otherwise, the email form is shown as seen in screen (a) in Figure 5.2 where the user enters his or hers email address associated to the service. The REST API is used to query login code response for the provided email address. If the email address is found in the system, the user receives a text message with a five-digit login code that is input to screen (b). If the email address is not found, the user is navigated to the registration screen (c).

A new account is registered by tapping the *Create account* button in screen (c) in Figure 5.2. The user then chooses an *account type* in screen (a) in Figure 5.3 based on whether a household or a company account is created. Once the desired account type has been selected, the user navigates to screen (b) where payment card information is entered. The card information is tokenised by a trusted third-party service. Finally, the user provides his or hers and optionally the company's basic information in screen (c). By tapping the *Done* button, the entered information along with the tokenised payment information is sent to Core Service via the REST API. If the entered information is correct, the API returns a login code so the client can store the login information and display the list of nearby car wash sites. Otherwise if the information is incorrect, an error is shown to the user so the information can

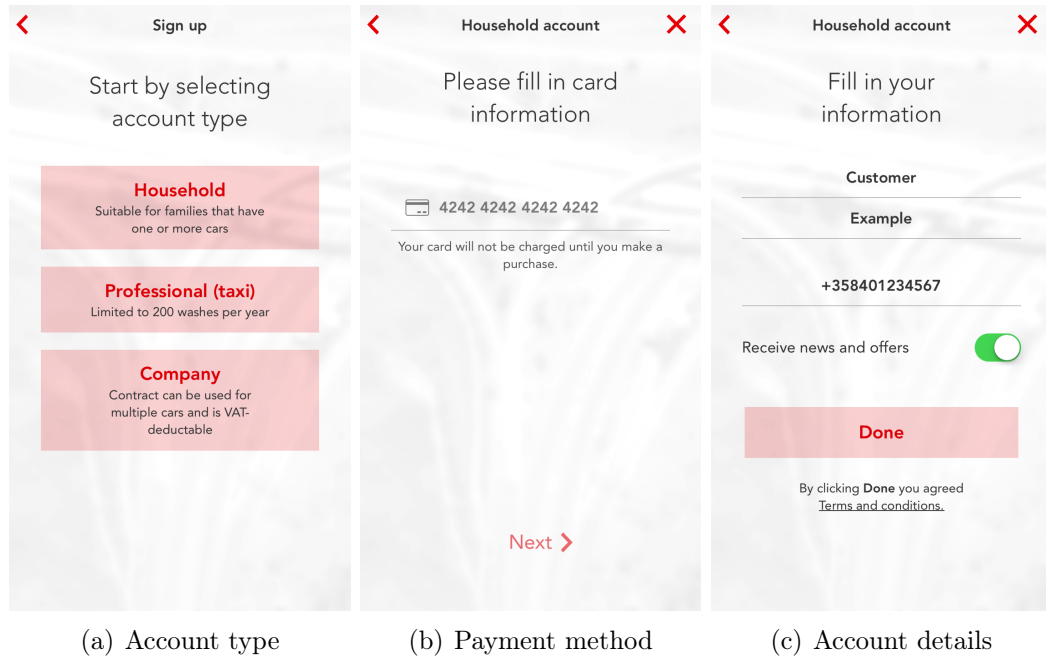


Figure 5.3 Registration screens.

be corrected and the request retried.

5.2.2 Listing Nearby Car Wash Sites

The main screen of the application shows a list of nearby car wash sites in ascending order sorted by user's distance to the site. The list of sites is returned by the REST API. By tapping a site, the user navigates to the car wash site screen where the state of the car washer can be seen. Figure 5.4 shows the two ways the user finds nearby car wash sites.

Screen (c) in Figure 5.4 shows a list of car wash sites sorted in ascending order by user's distance to the site. A row in the list contains the site's name in bold, the services provided by the site under the name, the site's icon and distance to the site in kilometres on the right.

Site's GPS coordinates are returned by the REST API, allowing the sites to be placed visually as pins on the map. User can expand the site list header by dragging the list down, exposing screen (b) in Figure 5.4 to the user. The user can then tap a pin and navigate to site detail screen.

The application listens for user location changes and updates the list of sites accordingly when the location changes. In addition, the list of sites returned by the API are refreshed periodically as sites may be removed and new sites added to the

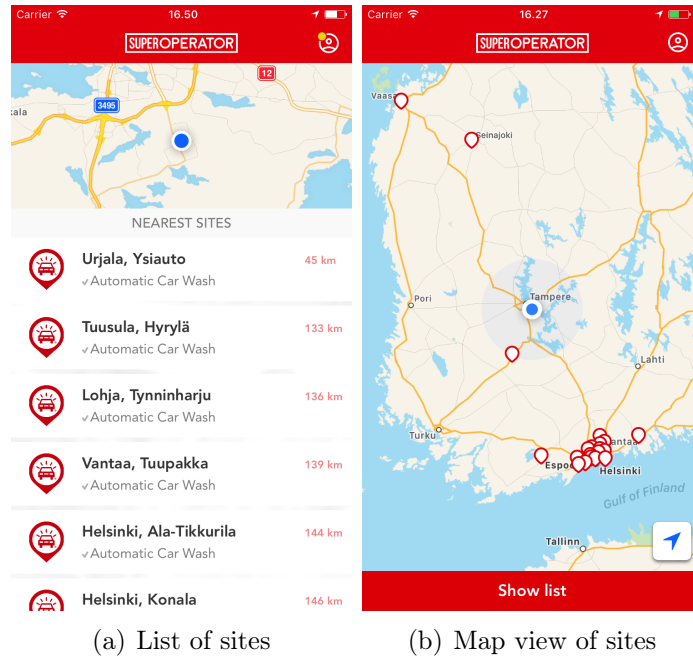


Figure 5.4 Car wash site list and map screens.

service.

5.2.3 Purchasing Washes

The final requirement is the ability to purchase and start car washes from the application. The application shows a purchase slider when a purchase can be made. The ability to purchase and start a wash is determined by multiple conditions which the application must update in real-time using the Socket.IO API. Figures 5.5 and 5.6 show the different states of the car washer when purchasing and starting washes.

User navigates to car wash site screen (a) in Figure 5.5 by tapping a site in Figure 5.4. The screen contains detailed information of the site, such as address and contact information of the owner. The bottom half of the screen is dedicated to interacting with car and wash program selections, displaying the washer state and starting washes.

When the car washer is ready to accept the next customer, screen (a) in Figure 5.5 is shown with the ability to choose a registered car and wash program. Once the user has driven in front the car washer's entrance as instructed, the car's license plate is read by the ALPR service. The plate is then sent to the Core Service and matched against registered car plates in the database. On a successful match, an event is propagated through the Socket.IO API to the client, notifying the client

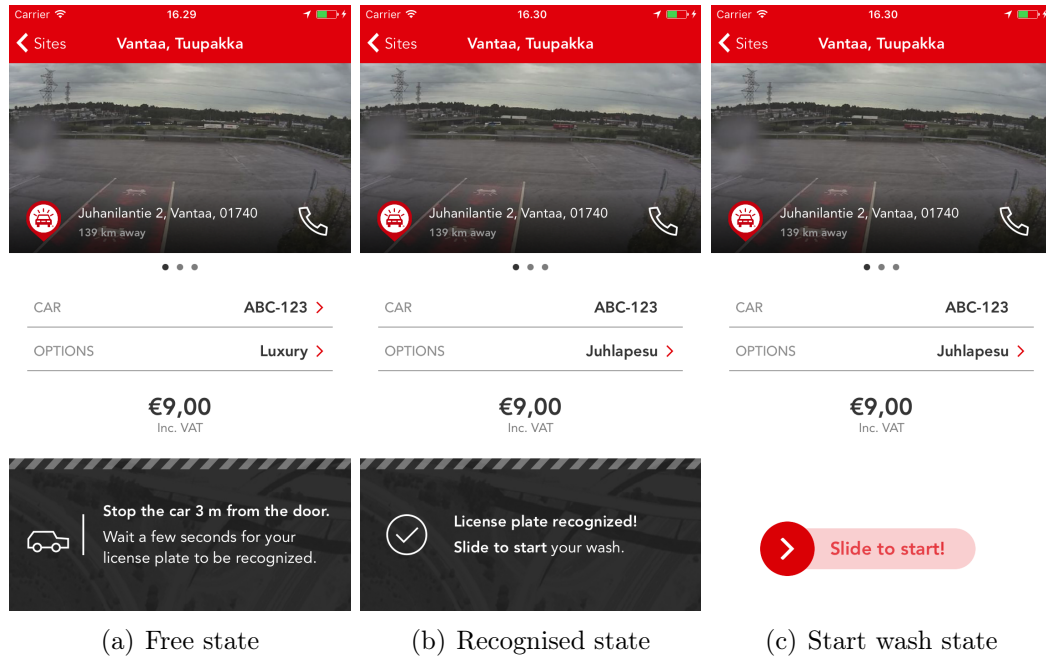


Figure 5.5 Car washer state screens.

that the washer is now able to accept the waiting customer. Screen (b) is shown briefly to the user when the license plate is recognised, requesting user attention. The user then confirms the purchase by dragging the *Slide to start!* slider all the way to the right or changes the wash program by tapping the *OPTIONS* button.

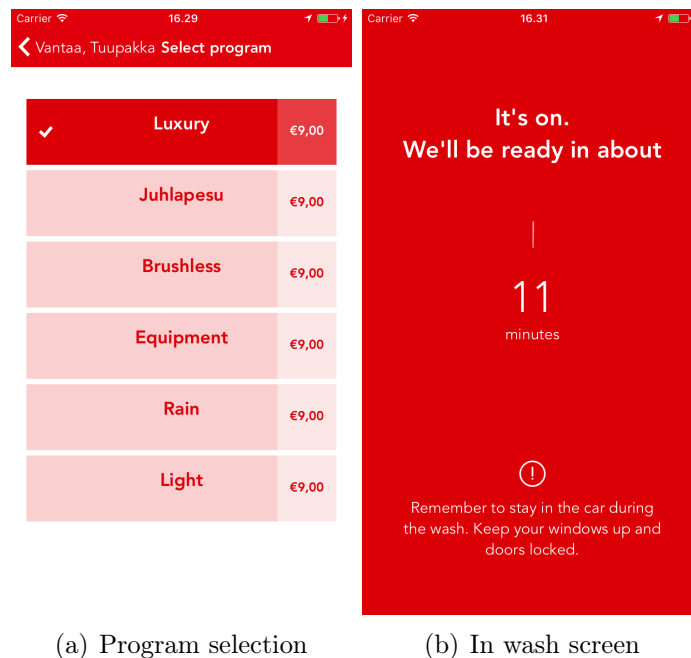


Figure 5.6 Program selection and wash progress screens.

The equipment supports one or more wash programs which are returned by the

REST API. The program selection is done in screen (a) in Figure 5.6. Wash prices are tied to the selected vehicle's subscription and must therefore be updated when the selected vehicle is changed. The price for the selected program is shown above the *Slide to start!* slider in screen (c) in Figure 5.5.

Once the payment has been processed and the car washer enters *occupied* state, screen (b) in Figure 5.6 is shown to the user. The screen displays detailed information of the wash progress, such as the remaining duration in minutes and additional instructions. When the wash ends, the application returns to screen (a) in Figure 5.5.

The Socket.IO API emits `equipment-state-changed` messages only when the equipment state changes. When a wash program starts, the Controller Service records the date, time and wash duration to equipment's state which is then propagated through the Socket.IO API the client. The client then periodically updates the remaining duration of the wash in screen (b) in Figure 5.6 by calculating the device's local time difference to the start time of the wash.

6. IMPLEMENTATION

This chapter describes how the Superoperator iPhone application requirements are implemented using reactive programming concepts. First, the application architecture combining the Model-View-ViewModel pattern and the Observable model is explained, followed by implementation of asynchronous services by transforming callback-based methods to Observables, and last, showing how application logic behind the screens presented in the requirements is implemented declaratively by chaining Observables with operators.

6.1 Model-View-ViewModel and Rx

In order to use MVVM architecture with UIKit framework, the *View* component depicted in Figure 2.4 needs to be defined. One definition would be to use an `UIView` subclass directly as the *View*, but that would cause incompatibility with the storyboard workflow, since storyboards can only work with `UIViewController`s [14]. `UITableViewController`s also provide many convenient lifecycle methods as shown in Table 6.1, which would have to be reimplemented. Therefore, the `UIViewController` is chosen as the *View* component, enabling the use of storyboards with all the benefits of the MVVM pattern. The layers of MVVM expose reactive interfaces with Observables, which enables operator chaining to be used to declaratively express application logic behind the layers.

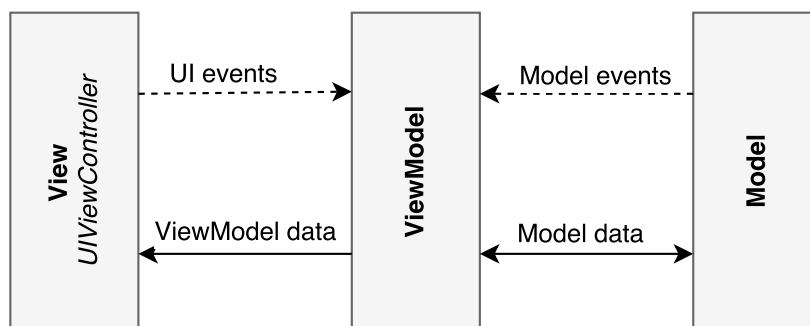


Figure 6.1 View, ViewModel and Model relationships.

Figure 6.1 shows how events and data are propagated in the described architecture. The *View* is implemented by subclassing an `UIViewController` as seen in the figure.

The View emits UI events, such as touch and input events to the ViewModel, and the Model emits internal events such as location change or network request result events to the ViewModel. The ViewModel then transforms the Model data to the View as depicted with the solid arrows. The ViewModel can also update the Model based on View events as illustrated with the two-headed arrow.

Derived classes of `UIViewController` implement *data-binding* logic to transform events emitted by the ViewModel's Observables to child `UIView` instances, such as text fields, buttons and labels. Data-binding logic also transforms asynchronous `UIView` events, such as tap and input events, to Observables that can be bound to the ViewModel. ViewModels contain most of the *business logic* of the application, meaning that ViewModels transform data from the Model, such as network request results, that are displayed in the UI. ViewModels respectively react to View events, such as button presses and update the Model.

6.1.1 Data and Event Binding

Data and event binding refers to how Observables from ViewModel are transformed to `UIView` property updates, and correspondingly `UIView` events are transformed to Observables and bound to the ViewModel. RxSwift provides convenient extensions for implementing data and event binding in a declarative way without having to rely on side-effects.

RxSwift includes various RxCocoa *extensions*, providing reactive wrappers for asynchronous UIKit events [5]. The extensions can be found under generated property named `rx` in `UIView` subclasses. For example, `UIButton`'s *tap* events are observed with the following extension:

```
1 let button = UIButton()
2 button.rx.tap.subscribe(onNext: { in
3     // handle button press
4 })
```

Often button tap events trigger some action in the associated ViewModel, such as perform a network request or navigate to another screen. One way to implement such functionality would be to call a method, such as `submit` from the above closure as a side-effect. To eliminate the use of side-effects and to make the code more declarative, Observable *binding* is utilised to implement the same functionality:

```
1 class ViewModel {
2     func submit(observable: Observable<Void>) -> Disposable {
3         return observable.subscribe(onNext: {
4             // Handle button press
5         })
6     }
7 }
8
9 let viewModel = ViewModel()
10
11 let button = UIButton()
12 button.rx.tap.bind(to: viewModel.submit)
```

ViewModel class is defined having `submit(observable:)` method which accepts an `Observable<Void>` instance and returns a `Disposable`. RxSwift `Observable` has `bind(to:)` method, which passes the `Observable` instance to the given function as a parameter. Last line's `subscribe(onNext:)` method call is replaced with `bind(to:)` call. Reference to `viewModel` instance's `submit(observable:)` method is passed to the `bind(to:)` method as argument.

Binding data from ViewModel to UIViews is also done using the `bind(to:)` method. RxSwift includes various extensions, which can be used to bind data from ViewModel to UIView properties. For example, `UILabel`'s `text` property can be bound as shown in the following example:

```
1 class ViewModel {
2     var greeting: Observable<String> {
3         return Observable.just("Hello")
4     }
5 }
6
7 let viewModel = ViewModel()
8
9 let label = UILabel()
10 viewModel.greeting.bind(to: label.rx.text)
```

Following the previous example, the `ViewModel` class contains a `greeting` computed property having `Observable<String>` type. After the class declaration, `viewModel` and `label` constants are declared and `viewModel`'s `greeting` `Observable` is bound

to `label`'s `text` extension method. The above code causes the `label`'s display text to update each time a `next` event is emitted by `greeting` Observable. In this case the label renders the text *Hello*.

Subscriptions to Observables are *disposed* when the associated `UIViewController` instance is deallocated. Because `UIViewController` holds strong reference to its `ViewModel` instance, the `ViewModel` is also deallocated along with the controller. A `DisposeBag` utility class is used to manage `Disposables` returned by subscriptions. One `DisposeBag` instance is instantiated for each `UIViewController` and the associated `ViewModel` instance to manage lifetime of internal observers' subscriptions. When `UIViewController` or `ViewModel` instance is deinitialised by ARC, the associated `DisposeBag` is also deinitialised and the added `Disposables` are disposed automatically. The following example shows how a `Disposable` is added to a `DisposeBag`:

```
1 let disposeBag = DisposeBag()
2 let button = UIButton()
3 button.rx.tap
4     .subscribe(onNext: {
5         // handle button press
6     })
7     .disposed(by: disposeBag)
```

Following the first example, a `disposeBag` constant is declared and `Disposable` returned by `subscribe(onNext:)` call is added to `disposeBag` using `disposed(by:)` method, accepting a `DisposeBag` as an argument.

6.1.2 Navigation

Navigating from one screen to another requires instantiating an `UIViewController` and adding it to a container view controller, such as `UINavigationController` or assigning it to the window's root view controller [14]. A `ViewModel` is also instantiated and bound to the view controller instance. When using storyboard *segue*s, the framework instantiates the `UIViewController` and notifies the initiating `UIViewController` by calling `prepare(for:target:)` lifecycle method [14].

`ViewModel` needs to be notified when the view is shown and when the view is hidden from the user to *lazily* fetch and refresh data displayed on the screen. To implement these events, certain `UIViewController` lifecycle events are intercepted

and propagated to the ViewModel. The lifecycle methods of interest are listed in Table 6.1.

Method	Description
<code>viewDidLoad</code>	The controller's view was loaded into memory.
<code>viewWillAppear:</code>	The controller's view is about to be added to the view hierarchy.
<code>viewWillDisappear:</code>	The controller's view was removed from the view hierarchy.
<code>prepare(for:target:)</code>	A segue is about to be performed.

Table 6.1 UINavigationController lifecycle methods needed to implement navigation. [14]

The ViewModel needs to know when the controller's view is attached to the view hierarchy and detached from it. To describe these events, a protocol called `Activable` is declared:

```

1 protocol Activable {
2     var isActive: Variable<Bool> { get }
3 }

```

The `Activable` protocol declares an `isActive` read-only constant `Variable` accepting boolean values. The variable is set to value `true` when the bound `UIViewController`'s `viewWillAppear:` is called, and respectively to `false` when `viewWillDisappear:` is called.

```

1 class ViewController: UIViewController {
2     private var activable: Activable?
3
4     func bind(activable: Activable) {
5         self.activable = activable
6     }
7     override func viewWillAppear(_ animated: Bool) {
8         activable?.isActive.value = true
9     }
10 }

```

A derived `ViewController` class is declared to delegate `UIViewController` lifecycle events to the associated `ViewModel`. A public `bind(activable:)` method is

declared which accepts an instance conforming to the `Activable` protocol. `UIViewController`'s `viewWillAppear:` and `viewWillDisappear:` (omitted for brevity) methods are then overridden in order to update the activated state of `activable`.

While `Activable`'s `isActive` provides a sequence of view hierarchy visibility states, it is more convenient to know *when* the view controller's view is attached to the view hierarchy. This event can be transformed with the following protocol extension:

```

1 extension Activable {
2     var activated: Observable<Void> {
3         return isActive.asObservable()
4             .distinctUntilChanged()
5             .filter { $0 }
6             .map { _ in }
7     }
8 }

```

Computed property `activated` returns an `Observable` of `Void` type, which means that the event has no value. The `Observable` emits a value when `isActive`'s value changes to `true` for the first time, because the `distinctUntilChanged` operator [5] only emits new values when the new value is different than the last one. The `Activable` protocol also offers flexibility to describe when the controller's view is *detached* from the view hierarchy, however the implementation is omitted for brevity. The following example shows how a `ViewModel` implementing `Activable` protocol is used together with the `ViewController` base class:

```

1 class ExampleViewModel: Activable {
2     let isActive = Variable<Bool>(false)
3
4     init() {
5         activated.subscribe(onNext: {
6             // Prepare initial data
7         })
8     }
9 }
10
11 class ExampleViewController: ViewController {
12     let viewModel = ExampleViewModel()

```

```

13
14     override func viewDidLoad() {
15         bind(activable: viewModel)
16     }
17 }

```

`ExampleViewModel` implements `Activable` protocol so it can be passed as argument to `ViewController`'s `bind(activable:)` method. `ExampleViewController` extends `ViewController` class and declares `viewModel` member constant. Once the view has been loaded into memory, `viewDidLoad` is called and `viewModel` is bound to the controller's lifecycle by calling `bind(activable:)` and passing the view model instance as argument. The view model can then start preparing initial data for the view when `activated` is triggered by `ViewController`'s `viewWillAppear:` method.

6.2 Asynchronous Services

ViewModels react to Model events and update Model state as shown in Figure 6.1. Asynchronous services are part of the Model layer and implement interfaces with Observables to system and third party frameworks. This section describes how HTTP client for performing REST API requests, Socket.IO client for car wash state observing and location service for obtaining user location are implemented.

6.2.1 HTTP Client

The HTTP client is used to perform requests to Core Service's REST API, exposing a reactive interface with Observables that can be consumed in ViewModels. The implemented service acts as a wrapper for Alamofire [25] that handles the client-server communication over HTTP protocol. Since RxSwift does not include Alamofire extension by default, the callback interface provided by Alamofire is transformed to Observable with `Observable.create`:

```

1 func fetch(request: URLRequestConvertible) -> Observable<Any> {
2     return Observable.create { observer in
3         let task = Alamofire.request(request)
4             .responseJSON { response in
5                 if let value = response.result.value {
6                     observer.on(.next(value))

```

```

7         observer.on(.completed)
8     } else {
9         observer.on(.error(ApiError.from(response)))
10    }
11 }
12 return Disposables.create {
13     task.cancel()
14 }
15 }
16 }

```

`fetch(request:)` is a function accepting an `URLRequestConvertible` which is an Alamofire protocol [25] used to map custom type instances, such as enumeration values to `URLRequest` objects. `fetch(request:)` returns an `Observable` of type `Any` since JSON objects can be represented as either arrays or dictionaries. The returned `Observable` emits a single value or an error before completing the sequence.

The closure passed to `Observable.create` as argument shows how a callback-based interface is translated to an `Observable`. The closure returns a `Disposable` that gets disposed when the `Observable` is disposed. In the above case, the dispose closure cancels the previously created task that is used to terminate the request. Alamofire begins executing the network request when the topmost closure is executed on `Observable` subscription. Response to the request is eventually delivered to the inner closure accepting a `response` object. In the inner closure, a `nil`-check is performed on the parsed JSON value. If `value` is non-`nil`, a `next` event is emitted along with a `completed` event. In case of a `nil` value, an `ApiError` object is created from `response` and `error` event is passed to the observer. In either case, the observable sequence is terminated when the Alamofire's request callback is called.

Alternatively, RxSwift includes extensions for `URLSession` [5], providing an easy way to make HTTP request if Alamofire usage is not required:

```

1 let req = URLRequest(url: URL(string: "http://www.tut.fi"))
2 URLSession.shared.rx.json(request: req)
3     .subscribe { event in
4         // Do something with result
5     }

```

In the above example, events are emitted from a *background thread*, which causes problems when the above `Observable` is bound directly to `UIView` objects since view

manipulation outside of the *main thread* is not supported in UIKit [14]. Events can be observed on the main thread with `observeOn:` operator [5]:

```

1 URLSession.shared.rx.json(request: req)
2   .observeOn(MainScheduler.instance)
3   .subscribe { event in
4     // Do something with result
5   }

```

Events emitted by `json(request:)` are scheduled to `MainScheduler`, causing any operators after `observeOn:`, including the `subscribe:` closure to be executed on the main thread.

6.2.2 Socket.IO Client

Socket.IO client implements reactive interface with Observables for asynchronous bi-directional real-time communication with Core Service's Socket.IO API. Socket.IO client is used to receive real-time equipment state change events in order to update the user interface to reflect car washer availability and customer wash progress shown in Figures 5.5 and 5.6. Similarly to Alamofire, Socket.IO client framework offers a callback interface which needs to be transformed to Observables using a wrapper.

Because Socket.IO is a *stateful* protocol, the client is constantly connected to the server, either using WebSockets or XMLHttpRequest (XHR) polling [19]. In mobile applications, network connectivity may be intermittently lost, so the client needs to re-establish subscriptions to equipment state changes when the connection is re-established. To abstract away connection handling, `SocketIOService` is introduced that provides an Observable of connected `SocketIOClient` objects. First, an enumeration describing the connection state is declared:

```

1 enum SocketState {
2   case connecting(socket: SocketIOClient)
3   case connected(socket: SocketIOClient)
4   case disconnected
5 }

```

`SocketState` enumeration holds the connecting or connected `SocketIOClient` instance. `SocketIOService` is then declared, exposing the connected client as an Observable (rest of the implementation is omitted for brevity):

```

1 class SocketIOService {
2     private let state = Variable<SocketState>(Disconnected)
3
4     var socket: Observable<SocketIOClient> {
5         return state.asObservable()
6             .map { state -> SocketIOClient? in
7                 switch state {
8                     case .connected(let socket):
9                         return socket
10                    default:
11                        return nil
12                }
13            }
14        .filterNil()
15    }
16    // ...
17 }

```

`SocketIOService` has a private member constant `state` that is assigned a value of `Variable<SocketState>` and is initialised to `Disconnected` state. Computed property `socket` returns an observable sequence of *connected* `SocketIOClient` instances, unwrapped from sequence of `SocketState` values. Only `connected` values are unwrapped and the associated socket instance is returned, otherwise `nil` is returned and filtered by `filterNil` extension. `filterNil` omits `nil` values from the sequence and unwraps non-`nil` optionals. The omitted portion of code updates `state` value to reflect state of the associated `SocketIOClient` instance and implements automatic connection recovery.

`SocketIOService` is used to implement higher-level interfaces for consuming equipment state change events. Equipment state change listener is implemented as follows:

```

1 let socketService = SocketIOService()
2 func listen(equipment: Equipment) -> Observable<EquipmentState> {
3     return socketService.socket
4         .flatMapLatest { socket in
5             return subscribe(socket, to: equipment)
6         }
7 }

```

`listen(equipment:)` takes `Equipment` struct containing database primary key identifier of the equipment (`id`) as argument. The function returns a *continuous* observable sequence of `EquipmentState` values that is completed only when the Observable is disposed. `flatMapLatest`: operator transforms the socket sequence into sequence of Observables and emits only items emitted by the most recently transformed Observable [5]. This means that each time the socket is connected, `subscribe(_:to:)` is executed to re-establish messaging with the server. `subscribe(_:to:)` is implemented with `Observable.create`:

```
1 return Observable.create { observer in
2     let id = socket.on("equipment-state-changed") { message in
3         let state = state(from: message)
4         observer.on(.next(state))
5     }
6
7     socket.emit("subscribe-equipment", [
8         "equipmentId": equipment.id
9     ])
10
11    return Disposables.create {
12        socket.off(id: id)
13        socket.emit("unsubscribe-equipment", [
14            "equipmentId": equipment.id
15        ])
16    }
17 }
```

`socket.on`: subscribes to incoming messages that are parsed and emitted to the observer. `socket.emit`: asks the server to send equipment state change events for the given equipment. `Disposables.create` block cancels the subscription by calling `socket.off(id:)` and notifies the server that the client is no longer interested in receiving state change events for the equipment.

The above implementation allows listening for equipment state changes regardless of the underlying socket connection state. When connection is recovered, the `Socket.IO` subscriptions are automatically negotiated without having to implement complex error handling in `ViewModels`.

6.2.3 Location Service

Location service implements reactive interface with Observables to emit user location updates using `CLLocationManager`. Up-to-date location information is required to calculate user distance to car wash sites returned by Core Service REST API and sort the list in ascending order by distance. Class named `LocationService` is declared to provide access to user location:

```
1 class LocationService {
2     private let disposeBag = DisposeBag()
3     private let locationManager = CLLocationManager()
4 }
```

`LocationService` contains private member constants `disposeBag` and `locationManager`. `CLLocationManager` manages delivery of location-related events and is imported from CoreLocation framework [26]. Observable of location values is then defined as follows:

```
1 var location: Observable<CLLocation?> {
2     return locationManager.rx.didUpdateLocations
3         .map { $0.last }
4         .startWith(locationManager.location)
5 }
```

`location` is a computed property, returning an Observable of `CLLocation?` instances. `didUpdateLocations` extension is an Observable of type `[CLLocation]`, acting as a proxy for `CLLocationManagerDelegate` [5]. `map`: operator is used to take the *last* array value because the most recent location update is at the end of the array [26]. Because `didUpdateLocations` only emits *future* updates, `startWith`: operator is used to emit the last known user location as first value of the sequence. Nil value in sequence indicates that the user location is not known at that point in time.

Applications must request user authorisation in order to access user location [26]. RxSwift provides extension for receiving authorisation status updates as an Observable [5]. Authorisation can be requested using `requestWhenInUseAuthorization`: method of `locationManager`. The authorisation status can be observed as follows:

```

1 locationManager.rx.didChangeAuthorization
2   .filter { $0 == .authorizedWhenInUse }
3   .subscribe(onNext: { [unowned self] _ in
4     self.locationManager.startUpdatingLocation()
5   })
6   .disposed(by: disposeBag)

```

`didChangeAuthorization` proxy extension is an Observable of type `CLLocationAuthorizationStatus` enumeration. `filter`: operator is used to select only `authorizedWhenInUse` enumeration values, which is the desired authorisation status for the application. When the user consents to the use of location, `authorizedWhenInUse` enumeration value is emitted bypassing the filter operator and starting location updates. Because the above code is placed in the class initialiser method, `unowned self` is used to prevent a cyclic strong reference to `self`.

6.3 Forms and Input Validation

Forms include one or multiple input elements, such as text fields or toggles and a button that causes the entered data to be processed when the button is tapped. Login and registration screens in Figures 5.2 and 5.3 consist mostly of these elements. This section describes how forms, input validation and error handling is generally handled in the application, using the login screen as an example for simplicity.

Figure 5.2 (a) shows one email text field and button element. The button is disabled and has reduced opacity when the *email* text field is empty or the entered email is not valid. Once a valid email has been entered, the button appears with normal opacity and can be tapped. ViewModel for the login screen is declared as:

```

1 class LoginViewModel {
2   let email = Variable<String?>(nil)
3   var canSubmit: Observable<Bool>
4   func submit(observable: Observable<Void>) -> Disposable
5 }

```

`LoginViewModel` has three properties: `email` to hold the value of the entered email address, `canSubmit` returning an Observable of boolean values indicating email address validity and `submit(observable:)` that performs login code request to the REST API for the entered email address. Controller for the login screen is then declared as:

```

1 class LoginViewController: UIViewController {
2     private let disposeBag = DisposeBag()
3     private let viewModel = LoginViewModel()
4
5     @IBOutlet weak var email: UITextField!
6     @IBOutlet weak var submit: UIButton!
7 }

```

`LoginViewController` has `disposeBag` and `viewModel` member constants and `email` and `submit` references to the *Email* text field and *Done* button views as seen in screen (a) in Figure 5.2. UIKit will have set references to @IBOutlet properties from storyboard when `viewDidLoad` is executed [14]. `disposeBag` constant is used to dispose subscriptions when ARC deinitialises the controller. `viewDidLoad` binds the two views to `ViewModel` properties:

```

1 override func viewDidLoad() {
2     email.rx.text.bind(to: viewModel.email)
3         .disposed(by: disposeBag)
4
5     viewModel.canSubmit.bind(to: submit.rx.isEnabled)
6         .disposed(by: disposeBag)
7
8     submit.rx.tap.bind(to: viewModel.submit)
9         .disposed(by: disposeBag)
10 }

```

`viewDidLoad` method contains data-binding code to bind `View` and `ViewModel` properties using RxSwift extensions. `email` text field's `text` sequence is bound to the `ViewModel`'s `email` property, `ViewModel`'s `canSubmit` `Observable` is bound to `submit` button's `isEnabled` property and finally, `submit` button's `tap` event `Observable` is bound to `ViewModel`'s `submit(observable:)` method. The `ViewModel`'s `canSubmit` computed property is implemented as:

```

1 var canSubmit: Observable<Bool> {
2     return email.asObservable()
3         .map { value in
4             guard let text = value else { return false }
5             return text.characters.count >= 3

```

```

6     }
7 }

```

`canSubmit` transforms `email` Observable to a boolean Observable using the `map:` operator, representing the enabled state of the submit button. The Observable emits `true` when the string value is not `nil` and is at least three characters long, and `false` otherwise. For example, as the user types characters `abc`, the following values are emitted by `canSubmit`:

$$(nil, "a", "ab", "abc") \rightarrow (false, false, false, true)$$

The values of `email` sequence are shown on the left, and transformed output of `canSubmit` is shown on the right. \rightarrow represents the `map:` operator. Text field is initialised to a `nil` `text` value, and as the user types the characters `abc`, the Observable emits a new value for each entered letter. Once the third letter is entered, `canSubmit` outputs `true` and the done button gets enabled. Last, the ViewModel's `submit(observable:)` method is implemented as:

```

1 func submit(observable: Observable<Void>) -> Disposable {
2     return observable
3     .flatMapLatest { email in
4         guard let email = self.email.value else {
5             return Observable.empty()
6         }
7         return self.login(as: email)
8     }
9     .subscribe { /* Process login response or handle error */ }
10 }

```

`submit` transforms tap event Observable to an Observable of login API response objects using `flatMapLatest:` operator and subscribes to it. `login(as:)` member function performs a REST API request for the given email and returns an Observable of response objects. `flatMapLatest` is used to cancel any overlapping requests if the user taps the done button multiple times while a previous request is still pending. The value of `email` is unwrapped and an empty sequence is returned if the value is `nil`. Login response and error handling are omitted for brevity.

The problem with the above `submit(observable:)` implementation is that the Observable is terminated when `login(as:)` emits an error. Error can be caused by

an unexpected API response or other network related error, in which case the user should be able to retry the request at a later time. However, since the Observable terminates to the first error event, tapping the done button shown in Figure 5.2 (a) does nothing once an error has occurred. To remedy this, the error event must be caught before it ends up to the `subscribe:` method. One way to catch error events is by using `catchError:` operator [5]:

```
1 // ...
2 .catchError { error in
3     // Handle error
4     return Observable.empty()
5 }
6 .subscribe(onNext: { response in
7     // Process login response
8 }
```

The above example replaces the `subscribe:` portion of `submit(observable:)` listing. `catchError` accepts a function accepting an `Error` object and returning an `Observable` that continues the sequence when an error occurs. The error handling portion of code is moved from `subscribe:` closure to `catchError:` closure. Now, when an error event is emitted by the source Observable, it is caught in the `catchError:` operator and the sequence is not terminated by returning an empty sequence, causing no `next` event to be emitted.

Alternatively, RxSwift `Driver` [5] concept can be used in place of `catchError`. `Driver` does more than just error recovery, such as emitting observed events from the main thread and sharing subscriptions.

```
1 // ...
2 .asDriver(onErrorRecover: { error in
3     // Handle error and continue sequence
4 })
5 .drive(onNext: { response in
6     // Process login response
7 }
```

`asDriver(onErrorRecover:)` replaces the `catchError:` operator from the previous example, transforming the `Observable` to `SharedSequence`. `subscribe(onNext:)`

is also replaced with `drive(onNext:)` because the transformed shared sequence does not have the same interface as the `Observable`. Depending on the use case, it may be more convenient to use one approach over the other.

6.4 Listing Nearby Car Wash Sites

Car wash site listing requirement shown in Figure 5.4 requires the app to fetch an array of `Site` objects from the REST API and sorting the array by ascending distance to the user. The `Site` objects are then transformed to *cell* `ViewModels` that expose the data for an individual row in the table. To implement the above requirement, two previously discussed `Observables` `fetch(request:)` and `LocationService.location` are combined. View model for the site list screen is declared as:

```
1 class SiteListViewModel: Activable {
2     private let locationService: LocationService
3     let isActivated = Variable<Bool>(false)
4     let siteCells: Observable<[SiteCellViewModel]>
5 }
```

Because sites can be added and removed from the service, the sites array is requested from the REST API each time the user navigates to the site list screen. In order to react to the activation event, the `ViewModel` conforms to the `Activable` protocol by declaring the `isActivated` constant. Private constant `locationService` is set to an instance of `LocationService` to observe changes to user location. Constant `siteCells` is set to an `Observable` of transformed `SiteCellViewModel` array that represents the screen's table row data. The site array is retrieved from the REST API as a reaction to the `Activable.activated` event:

```
1 let sites = activated
2     .flatMapLatest {
3         fetch(request: Router.ListSites)
4         .catchErrorJustReturn([])
5     }
```

Each time `activated` emits a value, one REST API request is made to retrieve the list of car wash sites visible to the user. `flatMapLatest` transforms the activation

event to a `Site` Observable, returned by `fetch(request:).catchErrorJustReturn:[5]` transforms the Observable to an empty array when the API request fails by any reason. Empty array is returned just as an example, in addition a descriptive error message would also be displayed to the user. Observable of sites sorted in ascending order by distance to the user is then defined as:

```
1 let nearbySites = Observable
2   .combineLatest(sites, locationService.location) { sites, location in
3     return sites.sorted {
4       // ...
5     }
6   }
```

`nearbySites` constant is an Observable of `Site` arrays that are sorted in ascending order by distance to the user. The `combineLatest:` operator takes two or more Observables and emits a new pair of values each time either Observable emits a value [5]. First pair is emitted once both Observables have emitted a value. `combineLatest:` takes a mapper function as the last argument that transforms the pair to a desired value. In this case, `sorted:` returns a sorted array of sites. The comparison closure passed as an argument to the `sorted:` method calculates a site pair's distance to `location` and returns a value indicating which site is closer (sorted before the second). The `Site` object contains coordinate of the site that is used for distance calculation. Last, the `siteCells` constant is defined as:

```
1 siteCells = nearbySites.map {
2   SiteCellViewModel(site: $0)
3 }
```

`nearbySites` is transformed to an Observable of `SiteCellViewModel` arrays. The View then subscribes to the Observable and updates the list and map each time sites are updated from the API or the user location changes.

6.5 Car Washer Operation

Car washer operation screens shown in Figures 5.5 and 5.6 include many view components having states that are dependant on the current state of the car washer. For example, such components are the *banner* that appears from the bottom of the

screen, the visibility of the *Slide to start!* button and the contents of selected car, operation and wash price labels. The car wash state is combined from the REST API and Socket.IO state change messages. Figure 6.2 shows dependency graph of view model properties for the car wash screen.

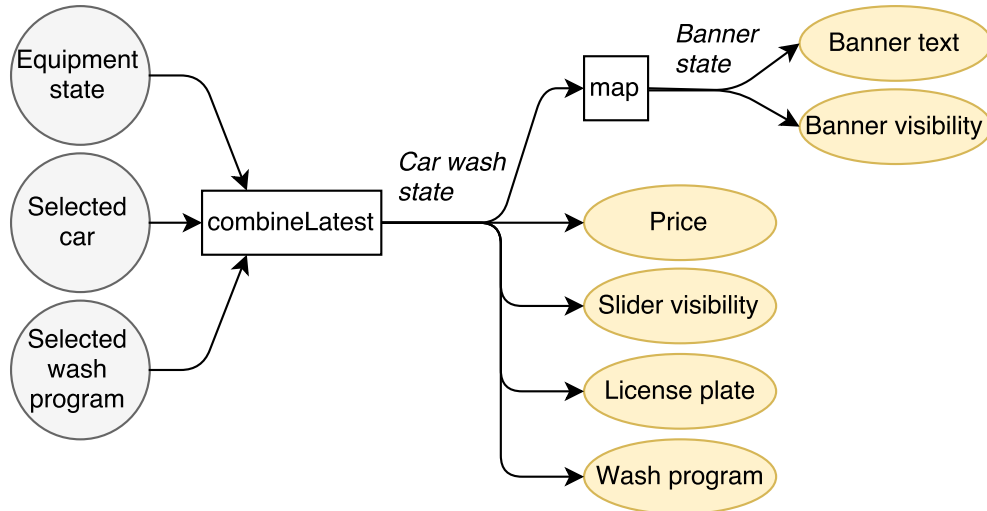


Figure 6.2 Car wash screen *ViewModel* property dependencies.

Equipment state received from the REST API and the Socket.IO API is not alone enough to implement all the view states shown in Figure 5.5. User selects the car and wash program using auxiliary screens that is then reflected on the car washer screen. The combined *car wash state* is transformed from the three source Observables *Equipment state*, *Selected car* and *Selected wash program* using `combineLatest:` operator as seen in dependency graph in Figure 6.2 on the left.

The dependency graph shows Observables dependant on the result of the *car wash state* Observable on the right. Additionally, `map:` operator is used to transform *banner state* that combines additional details, such as wash timer to transform banner text and visibility Observables.

In a naive implementation, the *ViewModel*'s Observables would be bound directly to view properties. The subscriptions would cause calculation of the entire Observable chain for each property, causing unnecessary computations and REST and Socket.IO API requests to be made. Binding this many views to *ViewModel* properties sharing a common dependency would therefore lead to performance problems.

RxSwift provides an operator for Observable result sharing named `shareReplay:` [5] that buffers last n values emitted by the source Observable and replays them for next operators in the chain. Since the result of `combineLatest:` is used by all of the following operators in the graph, `shareReplay:` operator is inserted after the

combineLatest: operator:

```
1 let carWashState = Observable.combineLatest(/* ... */)
2   .shareReplay(1)
```

In this case the last emitted *car wash state* value is replayed as indicated by the buffer size of 1 passed as argument. Each time any of the source Observables emit a new value, *car wash state* is transformed only once and the buffered result is replayed to all five Observables as shown in graph in Figure 6.2. Additionally, when views are bound to ViewModel properties, instead of executing the Observable chain six times for each property, the shared portion is executed only once, so only a single REST API and Socket.IO call is made to load resources and subscribe to equipment state updates.

7. EVALUATION

This chapter describes the benefits and problems related reactive programming techniques used in the application implementation. The benefits of reactivity in asynchronous programming and MVVM architecture are discussed by comparing the implementation to conventional asynchronous programming approaches. The problems discovered during the implementation are divided into both programming language and reactive extension library related problems, and maintainability problems caused by the complexity of the reactive extension model and API differences between reactive extension libraries.

7.1 Reactive Programming Benefits

Reactive programming allows implementing application logic declaratively as reactions to events while abstracting away difficult problems related to conventional asynchronous programming. Observable operator chaining introduces functional aspects to interaction modelling and enables convenient error handling, asynchronous task cancellation and threading management. RxSwift extensions enable binding Observables to `UIView` properties, allowing the UI to update declaratively without relying on side-effects, and transforming asynchronous UIKit events to Observables. The extensions are used together with the MVVM model shown in Figure 6.1 to implement data-binding between the layers.

7.1.1 Asynchronous Programming

Reactive programming techniques described in the implementation chapter show how commonly used asynchronous tasks in mobile development, such as network requests, threading and UI events can be managed with the Observable model. Handling these tasks using conventional programming approaches, such as callbacks, would require manual state management by the programmer, which can become complex and error prone [1, p. 52:2].

Observable model has many functional operators, such as `map:`, `filter:` and

`flatMap`: to transform Observables to other Observable types [5]. Application logic can be declaratively expressed by chaining together such operators, like shown in Table 4.3. In a callback-based approach, manual state management would be required, which often means declaring additional variables to describe the application state, and using side-effects to change the application state from callbacks [1, p. 52:2].

Transforming conventional callback-based methods, as discussed in the HTTP Client section, to Observables enables using any non-reactive framework as part of a reactive program. When all layers in the proposed MVVM model shown in Figure 6.1 provide interfaces with Observables, application logic can be declaratively expressed using operator chaining. In the proposed model, the ViewModel layer mostly implements business logic by transforming Observables between the View and Model layers.

Errors emitted by an Observable fall through the operator chain to the `subscribe`: block unless caught by an error handling operator, such as `catchError`: or `retry` [5]. Error handling operators allow continuing the sequence by returning a default value or resubscribing to the source Observable in hope it will complete without an error. Such operators are convenient when dealing with network requests that may fail intermittently, so the request can be retried automatically or some default data returned. Handling errors that fall to the `subscribe`: block is also more convenient than having to check for errors in multiple parts of asynchronous task execution.

Observables support multiple ways of cancelling asynchronous tasks with `Disposable` or operators such as `takeUntil`: [5]. This is especially convenient in the implemented application where pending network requests started by a ViewModel are cancelled when the user navigates away from the associated screen. Callbacks do not have built-in support for cancellations, so instead asynchronous tasks are often cancelled by calling some method returned by the asynchronous method, or with a side-effect.

Multiple views often depend on the same data returned by a network request, like seen in Figure 6.2. Instead of fetching the same data multiple times from the server, Observable sharing can be used to replay the last result to future subscribers. Observable sharing can be done by using the `shareReplay`: operator, or transforming the operator to a shared sequence with `asDriver` operator [5]. Multiple Observables can also be combined with `combineLatest`: and `zip`: operators, removing the need of manually tracking the order in which asynchronous tasks complete. Implementing similar functionality using callbacks can become difficult, since multiple

dependant asynchronous tasks need to keep track of the completion state of one or more asynchronous tasks.

Threading operators abstract away manual threading management, which would conventionally be done using the Dispatch framework [27]. Operators offering threading management include `observeOn:` and `subscribeOn:` operators [5]. Threading operators can be used to execute computationally expensive transformations in a background thread without blocking the UI thread.

7.1.2 Data-Binding

Data-binding refers to the act of updating `UIView` properties as a reaction to View-Model state changes, and updating the View-Model state as a reaction to `UIView` events. RxSwift offers `UIView` extensions [5] that Observables can be both bound to and from, achieving data-binding with minimal code as described in the implementation chapter. To understand how data-binding implemented using the described reactive extensions reduces program complexity, let's consider how data-binding is implemented using *delegate* pattern by declaring a delegate protocol:

```
1 protocol LoginViewModelDelegate {
2     func didChangeCanSubmit(isEnabled: Bool)
3 }
```

`LoginViewModelDelegate` declares a `didChangeCanSubmit(isEnabled:)` method which is used to notify the controller of changes to enabled state of the submit button as described in implementation. The controller then implements this protocol:

```
1 extension LoginViewController: LoginViewModelDelegate {
2     func didChangeCanSubmit(isEnabled: Bool) {
3         submit.isEnabled = isEnabled
4     }
5 }
```

`LoginViewController` implements the delegate method which assigns the argument to `submit` button's enabled state, implementing similar behaviour as in the implementation section. The view model is then implemented as:

```
1 class LoginViewModel: {
2     var delegate: LoginViewModelDelegate? {
3         didSet {
4             if let delegate = delegate {
5                 // Set initial view state through delegate
6             } else {
7                 // Cancel pending requests
8             }
9         }
10    }
11 }
```

`LoginViewModel` has `delegate` member variable, which holds the value of the controller's delegate. View model can assign the initial values through the delegate using the `didSet` block which is executed when a value is assigned to the variable. Pending requests are cancelled when the controller assigns `nil` value to the delegate as the controller's view is removed from the view hierarchy.

```
1 var email: String? {
2     didSet {
3         delegate?.didChangeCanSubmit(isEnabled: isEmailValid(email))
4     }
5 }
```

`LoginViewModel.email` property is implemented in a similar fashion. `didSet` block updates the submit button status to the controller based on entered email validity like shown in the implementation section. The controller then updates the `email` variable when `email` text field text is edited:

```
1 extension LoginViewController: UITextFieldDelegate {
2     func textField(_ textField: UITextField,
3                   shouldChangeCharactersIn range: NSRange,
4                   replacementString string: String) -> Bool {
5         viewModel.email = textField.text
6         return true
7     }
8 }
```

`LoginViewController` implements `UITextFieldDelegate` protocol and assigns `self` to `email` text field's `delegate` property. `UITextFieldDelegate` is used to manage editing and validation of `UITextField` objects [14]. The delegate then receives updates to the above method when the text field contents change. The updated text is assigned to the view model as a side-effect. The above handler becomes more complicated when form contains multiple text field elements since the edited text field needs to be determined by comparing view references.

The above pattern relies on the use of side-effects to update the UI and additional protocol declarations when compared the reactive extension's data-binding pattern described in the implementation section. The above listings are also incomplete and do not include button handling and request logic, which would require additional state management in the view model. Therefore, reactive programming makes it easier to implement data-binding logic used to glue together MVVM layers.

7.2 Problems

Problems discovered during the application implementation are divided into language related and maintainability related problems. Language related problems are strong reference cycles caused by reference counted garbage collection and type inference issues dealing with complex expressions. Maintainability concerns are raised by reactive extension libraries' API differences and high learning cost for developers unfamiliar with the reactive extension model.

7.2.1 Programming Language

Programming language related problems include strong reference cycles caused by reference counting used in the Swift programming language and complex type inference issues, requiring manual type definitions. The concerns are *language related*, because the described concerns are unique to the RxSwift implementation and the Swift programming language.

Strong reference cycles happen when two objects reference each other. The described situation happens often in situations where a reference to an Observable is set to member variable of an object that references `self` in an Observable chain's closure:

```
1 class ViewModel {
2     let subject = BehaviorSubject(value: "")
```

```

3
4  init() {
5      subject.subscribe { event in
6          self.doSomething()
7      }
8  }
9
10 func doSomething() {}
11 }

```

`ViewModel` has `subject` member constant, creating *strong* reference to the referenced object. In the class initialiser `init`, the `subject` is subscribed to and `self` is referenced in the closure, creating a *strong* reference from `subject` to `self`. This causes a *strong reference cycle* between `subject` and `self`, preventing neither from being deallocated during the program lifetime. When large amounts of memory become not reclaimable, it leads to performance problems and the application is terminated when more heap memory can not be allocated for the process by the operating system.

The above problem can be avoided by using *weak* and *unowned* references as described in the implementation section. However, this requires additional consideration from the programmer to determine when the use of *weak* or *unowned* references is required. For example, developers coming from *garbage collected* implementations of reactive extensions like RxJS [28] can be confused by when and why such language constructs are used.

The second consideration refers to *type inference* used in Swift programming language where the compiler may be unable to infer types from some commonly used RxSwift patterns. For example, consider the following `flatMap` operator:

```

1 .flatMap { result in
2     guard result.isSuccess else {
3         return Observable.empty()
4     }
5     return fetch(url: result.url)
6 }

```

The above operator tests whether `result.isSuccess` is `true` and returns an another `Observable` returned by `fetch(url:)`. When `result.isSuccess` is `false`, an empty

sequence is returned. The above pattern may lead to a confusing compiler error. To fix the issue, explicit type declaration is added for the closure return value:

```
1 .flatMap { result -> Observable<Result> in /* ... */ }
```

Closure return value type of `Observable<Result>` is declared to assist the type inference solver. Compiler errors caused by type inference issues related to RxSwift operator chaining may be difficult to diagnose unless the developer is familiar with such patterns.

7.2.2 Maintainability

Maintainability problems refer to potential issues regarding further development of the application. Introducing developers unfamiliar with reactive extensions to the project face additional learning curve before they can comfortably continue application development. The increased learning curve incurs additional development cost when compared to traditional iOS development. API differences between different reactive extension implementations also incur additional learning costs when the same developer switches from one project to another using another reactive extension implementation.

Reactive extension model has a lot of depth for someone who only has knowledge of callback-based asynchronous models used in iOS development. Functional aspects of reactive extensions such as `map:` and `filter:` may also be unfamiliar to a developer coming from imperative programming background. On the other hand, reactive extensions have implementations for many popular programming languages used in the software industry [4, 29], so a developer learning to use the reactive extension model in one project can start using the model in the next project with little learning cost.

While reactive extension implementations share a common asynchronous model and mostly the same API, the implementations have notable API differences. For example, RxSwift has `flatMapLatest:` operator as described in the implementation chapter, while in RxJS the operator has been renamed to `switchMap` [28]. The Driver concept introduced in RxSwift also has no counterpart in the RxJS library. These differences between reactive extension implementations incur additional learning cost when a developer for example moves from a project using RxSwift to a project using RxJS as a dependency or the other way around.

8. CONCLUSIONS

Reactive programming paradigm can be utilised in iOS application development using libraries that implement reactive programming models. In this work, a reactive extension library named RxSwift for the Swift programming language was used to implement the Superoperator iPhone application. RxSwift implements convenient extensions for transforming asynchronous UIKit events to Observables and binding Observables to view properties. When comparing RxSwift to other reactive extension implementations, such as RxJS, the Swift programming language introduces additional concerns with strong reference cycles and type reference issues. Dependency in RxSwift also raises maintainability concerns due to the complexity of the reactive extension model and implementation differences between libraries.

REST and Socket.IO APIs can be consumed in a reactive application by implementing wrappers for third party libraries, transforming callback-based asynchronous models to Observables. RxSwift also provides extensions for transforming asynchronous system services, such as location service events to Observables. Once the Model and View layers both provide Observable interfaces, the ViewModel layer can declaratively implement business logic by transforming Observables using operator composition.

Observable chaining with operators reduces program complexity when compared to conventional callback-based asynchronous programming models. Observable model and operator composition abstract away complex state management inherent to asynchronous programming when dealing with parallel and sequential asynchronous tasks. Error handling is also simplified, because errors fall through the Observable chain to the observer, enabling error handling to be done by the observer.

Reference counted garbage collection used in the Swift programming language requires additional consideration when common RxSwift patterns are used. Often a class subscribing to an Observable also holds a strong reference to the Observable, causing a strong reference cycle when `self` is referenced in the observer closure. Type inference also requires explicit type hinting when dealing with common operations, leading to difficult to debug compile time errors when omitted.

Complexity of reactive extension model incurs additional learning cost when compared to conventional asynchronous programming models used in iOS development. The increased learning cost leads to maintainability issues when developers unfamiliar with the reactive extension model are expected to continue the application development. Differences between libraries implementing the reactive extension model, such as RxSwift and RxJS, also require adaptation from a developer when switching between projects using different libraries implementing the same model.

The Model-View-ViewModel architecture described in the implementation chapter is also a good fit for reactive programming, because the ViewModel layer provides a convenient data abstraction for view properties that can be bound directly to views using RxSwift extensions. However, the used MVVM architecture is not a requirement when implementing reactive applications with Swift.

Future research could further compare the reactive extension model to other libraries implementing reactive programming patterns and investigate alternative models that even further reduce asynchronous programming complexity in iOS application development. Reactive programming used in alternative languages than Swift that target the iOS platform could also be examined, one example being JavaScript that can be used with React Native framework [30] to implement native iOS apps.

BIBLIOGRAPHY

- [1] E. Bainomugisha et al. “A survey on reactive programming”. In: *ACM Computing Surveys (CSUR)* 45.4 (2013), pp. 1–34.
- [2] K. Kambona, E. Boix, and W. D. Meuter. “An evaluation of reactive programming and promises for structuring collaborative web applications”. In: ACM, 2013, pp. 1–9. ISBN: 1450-320414.
- [3] E. Meijer. “Reactive extensions (Rx): curing your asynchronous programming blues”. In: ACM, 2010, p. 1. ISBN: 9781-450305167.
- [4] *ReactiveX*. URL: <http://reactivex.io/> (Accessed: 19.3.2017).
- [5] *Github - ReactiveX/RxSwift*. URL: <https://github.com/ReactiveX/RxSwift> (Accessed: 19.3.2017).
- [6] C. Anderson. “The Model-View-ViewModel (MVVM) Design Pattern”. In: Berkeley, CA: Apress, 2012, pp. 461–499. ISBN: 978-1-4302-3501-9.
- [7] R. T. Fielding. “Architectural styles and the design of network -based software architectures”. PhD thesis. Jan. 2000, p. 162.
- [8] A. Möller et al. “Update Behavior in App Markets and Security Implications: A Case Study in Google Play”. In: *Research in the LARGE: Proceedings of the 3rd International Workshop. Held in Conjunction with Mobile HCI*. 2012, pp. 3–6.
- [9] *iOS Technology Overview*. URL: <https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html> (Accessed: 6.3.2017).
- [10] *Programming with Objective-C*. URL: <https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html> (Accessed: 6.3.2017).
- [11] *The Swift Programming Language (Swift 3.1)*. URL: https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/ (Accessed: 6.3.2017).
- [12] S. Blackburn, P. Cheng, and K. McKinley. “Myths and realities: the performance impact of garbage collection”. In: *ACM SIGMETRICS Performance Evaluation Review* 32.1 (2004), pp. 25–36.
- [13] *Objective-C Automatic Reference Counting (ARC)*. URL: <https://clang.llvm.org/docs/AutomaticReferenceCounting.html> (Accessed: 14.3.2017).

- [14] *UIKit - Apple Development Documentation*. URL: <https://developer.apple.com/reference/uikit> (Accessed: 18.3.2017).
- [15] *Reactive Extensions*. URL: [https://msdn.microsoft.com/en-us/library/hh242985\(v=vs.103\).aspx](https://msdn.microsoft.com/en-us/library/hh242985(v=vs.103).aspx) (Accessed: 19.3.2017).
- [16] *Github - ReactiveCocoa/ReactiveSwift*. URL: <https://github.com/ReactiveCocoa/ReactiveSwift> (Accessed: 23.3.2017).
- [17] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Reading (MA): Addison-Wesley, 1994. ISBN: 0201633612.
- [18] *Futures*. URL: <https://msdn.microsoft.com/en-us/library/ff963556.aspx> (Accessed: 22.4.2017).
- [19] *Socket.IO*. URL: <https://socket.io/> (Accessed: 23.4.2017).
- [20] *Elastic Load Balancing*. URL: <https://aws.amazon.com/elasticloadbalancing/> (Accessed: 23.4.2017).
- [21] *Virtual Private Networking: An Overview*. URL: <https://technet.microsoft.com/en-us/library/bb742566.aspx> (Accessed: 23.4.2017).
- [22] *Amazon Web Services (AWS) - Cloud Computing Services*. URL: <https://aws.amazon.com/> (Accessed: 23.4.2017).
- [23] *The JavaScript Object Notation (JSON) Data Interchange Format*. URL: <https://tools.ietf.org/html/rfc7159> (Accessed: 25.7.2017).
- [24] *JSON Web Token (JWT)*. URL: <https://tools.ietf.org/html/rfc7519> (Accessed: 25.7.2017).
- [25] *Github - Alamofire/Alamofire*. URL: <https://github.com/Alamofire/Alamofire> (Accessed: 16.8.2017).
- [26] *Core Location*. URL: <https://developer.apple.com/documentation/corelocation> (Accessed: 17.8.2017).
- [27] *Dispatch - Apple Development Documentation*. URL: <https://developer.apple.com/documentation/dispatch> (Accessed: 2.9.2017).
- [28] *Github - ReactiveX/rxjs*. URL: <https://github.com/ReactiveX/rxjs> (Accessed: 19.8.2017).
- [29] *The 2017 Top Programming Languages*. URL: <http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages> (Accessed: 20.8.2017).
- [30] *React Native*. URL: <https://facebook.github.io/react-native/> (Accessed: 27.8.2017).