Bachelor's thesis

Information Technology

2018

Mark Vasiv

FUNCTIONAL REACTIVE PROGRAMMING FOR IOS

– with Objective-C and ReactiveCocoa

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology

2018 | 134

Mark Vasiv

FUNCTIONAL REACTIVE PROGRAMMING FOR IOS

- with Objective-C and ReactiveCocoa

The logic of most mobile applications is based on events coming from either user interface or network. Object-Oriented Programming (OOP), which is used in most of the client-side applications, is not designed to build software based on events. The existing related mechanisms in most of Object Oriented (OO) languages include the use of delegate methods, callbacks, and other language-specific techniques. These techniques abstractly serve the same functionality but use completely different syntax. This thesis focuses on iOS application development using Objective-C language, which has the above mentioned limitations concerning event-based programming. Firstly, the thesis introduces the language as well as iOS SDK basics and common patterns, highlighting core features and problems. Secondly, it provides a possible solution for the discussed issues – the use of the Functional Reactive Programming (FRP) paradigm. To achieve the FRP paradigm while developing iOS applications, the thesis suggests using ReactiveCocoa framework. The framework is a good compromise between abstract declarative programming style and relatively good performance level. The thesis also includes a fully-functional messenger User Interface (UI) component. The component is firstly built with the help of OOP and then refactored to expose FRP paradigm. The development process is documented and discussed with attention to every aspect of the process including design and actual programming.

KEYWORDS:

programming, iOS, Objective-C, ReactiveCocoa, FRP

CONTENTS

LIST OF ABBREVIATIONS (OR) SYMBOLS 6

1 INTRODUCTION 7

2 NATIVE IOS APP DEVELOPMENT 9

2.1 Software development kit 9

2.2 Objective-C 10

2.2.1 Class definition 10

2.2.2 Method declaration 11

2.2.3 Messages 12

2.2.4 Inheritance 12

2.2.5 Protocols 13

2.2.6 Categories 14

2.2.7 Properties 15

2.2.8 Primitive types 17

2.2.9 Memory management and ARC 17

2.2.10 Runtime 18

2.3 Standard frameworks and tools 24

2.3.1 Foundation 24

2.3.2 UIKit 25

2.3.3 Concurrency and libdispatch 27

2.4 Common patterns 29

2.4.1 Key-value observing/coding 29

2.4.2 IBOutlets and IBActions 32

2.4.3 NSNotification 33

2.4.4 Delegation 34

2.4.5 Blocks 35

2.4.6 MVC 36

2.5 Common problems 37

2.5.1 MVC vs MVVM 37

2.5.2 Event-based patterns 39

3 FUNCTIONAL REACTIVE PROGRAMMING 40

3.1 Functional programming 40

3.2 Reactive programming 40
3.3 Composition of paradigms 41
3.4 Support in iOS 41
3.5 ReactiveCocoa 41
3.5.1 Basic classes 42
3.5.2 Basic operators 43
3.6 Syntactical difference between ReactiveCocoa and iOS SDK 46

4 MESSENGER DESIGN 48
4.1 Contacts list 48
4.2 Chats list 49
4.3 Chat view 51
4.3.1 List of messages 51
4.3.2 Footer 53
4.3.3 Navigation bar actions 54
4.4 Shared media 56
4.5 Model 56

5 NATIVE APP WITH MVC 58
5.1 Managers 58
5.1.1 Contact manager 59
5.1.2 Chat manager 60
5.1.3 File manager 64
5.1.4 Database manager 66
5.2 Controllers 66
5.2.1 Chat list controller 66
5.2.2 Chat controller 72
5.3 Native app problems 88

6 REFACTORING WITH MVVM AND REACTIVECOCA 90
6.1 Managers 90
6.1.1 Contact manager 90
6.1.2 Chat manager 91
6.1.3 File manager 93
6.2 Views and view models 93
6.2.1 Chats list 93
6.2.2 Chat controller 104

7 POSSIBLE PROBLEMS WITH REACTIVECOCOA 120
7.1 Performance of object creations 120
7.1.1 RACSignal 121
7.1.2 RACObserve 121
7.1.3 RAC 122
7.1.4 RACCommand 122
7.2 Speed of events propagation 122
7.2.1 KVO and RACObserve 123
7.2.2 RACSignal 123
7.2.3 NSNotification and RACSignal wrapper 123
7.3 Implications 124

8 CONCLUSION 125

REFERENCES 126

LIST OF ABBREVIATIONS (OR) SYMBOLS

Abbreviation	Explanation of abbreviation (Source)
API	Application Programming Interface
ARC	Automatic Reference Counting
FIFO	First-In, First-Out
FP	Functional Programming
FRP	Functional Reactive Programming
GCD	Grand Central Dispatch
GUI	Graphical User Interface
IB	Interface Builder
IDE	Integrated Development Environment
KVC	Key-Value-Coding
KVO	Key-Value-Observing
MVC	Model View Controller
MVVM	Model View ViewModel
OOP	Object Oriented Programming
SDK	Software Development Kit
XML	Extensible Markup Language

1 INTRODUCTION

Functional Reactive Programming (FRP) is a declarative paradigm of programming, which already has shown its benefits in many programming fields [1]. The paradigm is built on data streams that are common to any reactive application. The term reactive here means that the application interferes with many external inputs. That is the case with any mobile app because mobile apps heavily work with user interactions and usually require network handling. Despite that FRP can be seen as a suitable choice for mobile programming, most of the client side software is still written with the use of imperative programming due to historical reasons.

Imperative and declarative programming fundamentally differ by approach that is used when designing software. The declarative programming is based on the idea that the program is written by telling the computer what to do instead of how to accomplish the result. The latter approach is used in imperative programming. The declarative programming abstracts away many small non-crucial details from the programmer, making it easy to focus on more complex tasks.

This thesis focuses on iOS application development using Objective-C language. The chosen language and the Software Development Kit (SDK) are object-oriented and imperative. To take advantage of FRP, the thesis presents the RectiveCocoa framework. The ReactiveCocoa provides needed data stream interfaces as well as additions to the SDK making it easier for developers to integrate the framework into the app. Overall, the solution has a fine balance between abstraction level and performance, which is described later in more detail in the thesis.

As an illustration of both paradigms, the thesis presents a messenger iOS app. It is firstly developed using pure iOS SDK and Objective-C and then refactored to use ReactiveCocoa framework. The documentation of the development process covers design process, programming techniques and patterns that are commonly used when developing iOS applications. It also highlights differences between OOP and FRP principles, as well as, provides results of ReactiveCocoa performance measurements. The structure of the thesis is as follows: Chapter 2 presents technology used to build iOS applications as well as basics of the Objective-C language and the iOS SDK. It also discusses the weaknesses of the language and the SDK. The source for this technology overview is based on official Apple documentation, public source files and articles of experts in the iOS programming field. Chapter 3 introduces FRP principles in a more detail and the ReactiveCocoa framework. The ReactiveCocoa overview includes a description of basic data types and operators commonly used. Chapter 4 is dedicated to the design process of the discussed messenger app. It presents UI layout, as well as, basic constrains for future model design. Chapter 5 describes in full detail the development process of the app using Objective-C and MVC. Chapter 6 continues the description of the development process and is dedicated to the refactoring of the app to expose the FRP paradigm and MVVM pattern using the ReactiveCocoa framework.

Chapter 7 outlines possible problems associated with the use of the ReactiveCocoa. More attention is paid to the potential performance issues. Finally, chapter 8 discusses the possible future use of the app in addition to outlining most essential conclusion that could be made from the thesis.

2 NATIVE IOS APP DEVELOPMENT

A native app is an app written specifically for the platform in which it is intended to use. Such an app can be developed only with the use of tools and technology integrated into the platform. Because of this, it can interact with and take advantage of all operating system and hardware features, which are publicly available in the Software Development Kit. The main benefits of native applications are their high performance and platform-friendly user experience. Obviously, target-specific application development is expensive as soon as most of the projects still need to support several platforms. Usually, this results in the need for more developers with a specific knowledge required to develop an app for a particular platform. Therefore, the cost of development correlates with the quantity of supported platforms.
In contrast, hybrid app development is targeted to support multiple platforms with a single codebase. This approach can drastically increase the speed and decrease the cost of development; however, the performance of hybrid apps is much lower than the performance of native apps. There are multiple frameworks and technologies which allow creating hybrid apps, each with its own benefits and drawbacks. This latter approach may be suitable for rapid prototyping or creation of apps with a relatively small functionality. Controversially, often for the development of commercial products, the native app development is used. As Mark Zuckerberg stated at TechCrunch Dusrupt conference, "The biggest mistake we made as a company was betting too much on HTML5 instead of native... We burnt two years." [1]
Native app development for iOS – the mobile application platform introduced by Apple Inc. in 2007 – is only possible with the help of software provided by Apple. The operating system was introduced with a demonstration of first iPhone, however, back then iOS was a closed system without public SDK, and there were no 3rd party applications. As was promised by Steve Jobs in the message posted on apple.com [2], the situation changed in 2008 with the release of iOS SDK and publishing of official documentation, enabling developers to create and distribute their application to the App Store.

2.1 Software development kit

As stated in the Xcode release notes [3], tools used to develop iOS apps include Xcode Integrated Development Environment (IDE), Xcode developer instruments, and iOS Software Development Kit (SDK). iOS SDK updates several times a year with each of major iOS releases, however, its core usually remains unchanged.
Xcode includes text editor, Interface Builder (IB) and Xcode developer tools, which are command line utilities used by Xcode to compile source code to the executable binary and extend some debugging functionality provided by Xcode. The compiler of choice is Clang.

Initially, Objective-C was the primary language of the platform - the successor of the C language that extends its abilities with the object-oriented paradigm. However, recently the Swift language was introduced [4] as a possible replacement of the old and verbose Objective-C. Assuming the fact that Objective-C is a child of C language, the use of C is also possible. Moreover, most of the low-level Application Programming Interfaces (APIs) actually have a C style. With a recent update, the usage of Swift is highly encouraged; however, the thesis only deals with Objective-C language because its syntax is easier to understand and the language itself does not limit actions regarding the selected topic. The use of Objective-C also ensures a good illustration of difference between imperative and declarative programming, because Objective-C is an object-oriented language and Swift is a multi-paradigm language. Moreover, the use of Objective-C adds some benifits from the practical point of view – still many iOS apps are written using Objective-C and are not migrating to the Swift because of the migration cost and immaturity of the Swift.

iOS SDK contains standard frameworks that enable developers to use system APIs and structures defined by Apple, from database management to user interface drawing and touch recognition. Along with the frameworks, the SDK contains the iPhone Simulator, a program used to simulate the look and feel of the iPhone on the developer's desktop. iPhone Simulator is not an emulator and runs code generated for an x86 target rather than ARM which is used in real devices. Because of that, the testing of production apps and perfomance measurements should always be carried out using real mobile devices.

2.2 Objective-C

Objective-C is an object-oriented language. It supports inheritance, polymorphism, data encapsulation and introduces some unique patterns of code structuring. These patterns, as well as some syntax examples and peculiarities of the language implementation, are described in this chapter.

2.2.1 Class definition

According to the Objective-C documentation [5], a class definition in Objective-C is split into two separate chunks of code - interface, and implementation. Interface holds a declaration of all public methods and invariants that the class undertakes to support. Code snippet 1 contains an example of an interface of a class called *Person*, which holds two invariants - *firstName* and *lastName*, and supports a method that gets a full name.

Code snippet 1. Interface of a Person class.

```
@interface Person : NSObject  {
    NSString *firstName;
    NSString *lastName;
}
- (NSString *)getFullName;
@end
```

Here *@interface* is a keyword that indicates the start of an interface declaration and *@end* is a keyword that indicates the end of a declaration block. *NSString* is a standard class used for storing textual strings.
The second part of a class definition in Objective-C - implementation, holds information about how actually methods work. An example of the same class implementation is shown in Code snippet 2.

Code snippet 2. Implementation of a Person class.

```
@implementation Person
- (NSString *)getFullName {
    return [firstName stringByAppendingString:lastName];
}
@end
```

Here *@implementation* indicates the start of the implementation block, and *@end* again shows the end of a block. The class contains only one method that is needed to be implemented - *getFullName*.

2.2.2 Method declaration

Methods in Objective-C can be of instance and class types. Instance method belongs to an instance of a class and is declared with `-` sign in the beginning. Each instance method in Objective-C takes a pointer implicitly to a data holding an object, for which the method was called. In the methods body, this pointer can be referenced by the use of reserved key *self*. The pointer to an instance of a base class is accessible through keyword *super*. Class methods belong to a class itself and are declared with `+` sign in the beginning. They do not reference any instance object, instead keyword *self* is used to access a class (classes are also objects in Objective-C). An example of different method declarations and their description is shown in Code snippet 3.

Code snippet 3. An example of different methods' declarations.

```
+ (Class)class;
// class method that returns value of Class type
- (id)init;
// instance method that returns value of id type
- (void)addObject:(id)anObject;
// instance method which takes id parameter and returns value of id type
+ (NSString *)stringWithCString:(const char *)aCString
                  usingEncoding:(NSStringEncoding)encoding;
// class method which takes (const char*) (NSStringEncoding) arguments and returns
// value of NSString type
```

2.2.3 Messages

The syntax of method invocation in Objective-C is shown in Code snippet 4. Here *receiver* is the object which will execute the method, and *methodOne* is a method to be invoked.

Code snippet 4. An example of message send.

```
[receiver methodOne];
```

If the method takes any argumets, these are declared just after the method name and each argument is written after the colon. An example of such methods is shown in Code snippet 5.

Code snippet 5. An example of methods with parameters.

```
[receiver methodWithParameter:1];
[receiver methodWithFirstArgument:10 andSecondArgument:20];
```

If method returns any object, it is possible to use nested syntax, so that the return value of the first invocation is used as receiver for the next method. This syntax can be seen from Code snippet 6.

Code snippet 6. An example of nested syntax

```
[label setText:[person getFullName]];
```

Nested construct shown in Code snippet 5 can be rewritten using two separate lines of code, which will better explain the order of execution that is taking place when such syntax is used. Cosider Code snippet 7, where such equivalent is shown.

Code snippet 7. Decomposition of nested syntax.

```
NSString *fullName = [person getFullName];
[label setText:fullName];
```

In fact, a more accurate name for the method's invocation in Objective-C is message sending. In the previous example shown in Code snippet 7, *setText:* is called a method selector and *setText:fullName* is the actual message. Such terminology is explained by the peculiarities of the runtime library. The library itself and messaging will be explained in a more detail in Chapter 2.2.10 Runtime.

2.2.4 Inheritance

The parent of most of the classes in Objective-C is *NSObject*. *NSObject* implements some basic behaviors that are needed for the correct program's execution. The interface of *NSObject* declares the *isa* invariant and some basic methods for

performing selectors, finding parent class, and related to memory management. These methods are needed in any Objective-C class, so the *NSObject* is used as a root for the most class hierarchies. Inheritance is declared in the class interface using the syntax shown in Code snippet 8.

Code snippet 8. Inheritance syntax.

```
@interface MyClass : NSObject
@end
```

Objective-C does not support multi-inheritance, which means that classes can not have multiple parents. However, very similar functionality can be achieved by the use of protocols.

2.2.5 Protocols

A protocol is a mechanism for extending class functionality, by supporting extra methods. In fact, a protocol is an anonymous interface that can be used by any class whishing to support methods listed in this interface. This mechanism hides the actual class of the object, exposing only protocol interface. If some class wants to utilize a protocol, it should implement methods declared in the protocol. If a class does so, this is called conforming to a protocol. The syntax of protocol declaration is shown in Code snippet 9.

Code snippet 9. Protocol declaration syntax.

```
@protocol ProtocolName
- (void)firstMethod;
- (void)secondMethod;
@end
```

If a class wants to indicate that it conforms to a protocol, it should state it in the interface by placing name of the procotol after the class name. An xample of such declaration is shown in Code snippet 10.

Code snippet 10. Protocol conformance syntax.

```
@interface MyClass <ProtocolName>
@end
```

Class *MyClass* used in the previous example should also implement methods *firstMethod* and *secondMethod* normally in its implementation to successfully conform to the protocol.

The benefits of protocols usage comes from the possibility to substitute a class name in variable definition with an object conforming to a protocol. This enables us to use protocols in multiple patterns, where the actual class name of the object is not known or irrelevant. Even though in such situation the class of the object is not known, we still

know that it conforms to a protocol and provides a desired functionality. Consider the example of such protocol usage in Code snippet 11.

Code snippet 11. An example of protocol usage.

```
@protocol PresenterProtocol
- (void)present;
@end
@interface ViewController : UIViewController
@property (strong, nonatomic) id<PresenterProtocol> presenter;
@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];

    [self.presenter present];
}
@end
```

Here ViewContoller holds a reference to an object of type *id,* which represents any type *(void *)*, however, this object is indicated to conform to a protocol *PresenterProtocol*. Therefore, it is safe to call method *present* declared in the protocol. *Presenter* in fact*,* could be of any class, this information is irrelevant for the *ViewController* as long as the object conforms to a protocol. This helps to uniform APIs and reuse code.

2.2.6 Categories

Category is a structure, which is used to split a single class definition into multiple files. Its goal is to ease the burden of maintaining large code bases by modularizing a class. That prevents source code from becoming monolithic 10000+ line files that are impossible to navigate, and makes it easy to assign specific, well-defined portions of a class to individual developers. The syntax for declaring a category is shown in Code snippet 12.

Code snippet 12. Category declaration syntax.

```
@interface MyClass (CategoryName)
- (void)additionalMethod;
@end
```

Here *MyClass* is the name of class that is being extended and *CategoryName* is the name of the category. Parameter *CategoryName* is optional and can be omitted, this results in a creation of so-called anonymous category or class extension. Class' extenssions are often used in *.m* files to hide the internal part of class' interface. The syntax of implementation part of a category is a mixture of normal class implementation and category declaration. Category implementation from previous example can be seen in Code snippet 13.

Code snippet 13. Category implementation syntax.

```
@implementation MyClass (CategoryName)
- (void)additionalMethod {...}
@end
```

Any class can have as many categories as needed. However, it is important to remember that if a category overrides a method declared in the class, during runtime will be used the implementation declared in the category. If the class has multiple categories that override the same method, runtime will use the implementation from the latest imported category. That is not easily predictable behaviour, so overiding methods inside of categories is usually avoided.

2.2.7 Properties

It is a common practice in object-oriented languages to use special methods – a setter and a getter to access an invariant, where the first one sets the value of the invariant and the latter returns the value of the invariant. That ensures data encapsulation by hiding internal invariants. The properties were designed to lighten up the definition of setters and getters by using short syntax, which is interpreted by the compiler to automatically generate described above methods.

Properties are declared in the interface of a class by declaring the name, class, and attributes, which will be covered shortly. The example of property declaration is shown in Code snippet 14.

Code snippet 14. Property declaration syntax.

```
@property (strong, nonatomic) NSString *firstName;
```

This notation will tell the compiler to generate an invariant of type *NSString* with the name *_firstName*. The compiler will also generate a getter and a setter for this invariant with corresponding names and signatures. The methods generated for the property in the previous example are shown in Code snippet 15.

Code snippet 15. Methods automatically generated for a property.

```
- (NSString *)firstName;
- (void)setFirstName:(NSString *)firstName;
```

The attributes to be specified when declaring properties determine the manner, in which a getter and a setter are generated. These attributes can be divided into the following groups:
- accessibility attributes (readonly / readwrite),
- ownership attributes (strong / copy / assign / weak),
- atomicity attributes (atomic / nonatomic).
Explicitly or implicitly the attributes of all types are used in each property.

Accessibility attributes

readwrite - indicates that the property is available for read and write, therefore a setter and a getter are generated. This attribute is set by default to all properties.

readonly - indicates that the property is read-only, only a getter is generated.

Ownership attributes

strong - indicates that the generated setter will increase the reference count by one on the assigned object and decrease the reference count by one on the object that previously was referred by the property. This is the default value when using Automatic Reference Counting (ARC). Memory management is described in more detail in Chapter 2.2.9 Memory management and ARC.

copy - indicates that in the generated setter the value of a corresponding instance variable will be assigned to the return value of *copy* message sent to the assigned object. The use of this attribute is only possible if the class is not mutable and conforms to protocol *NSCopying*, which guarantees that it can respond to the *copy* message.

weak - ensures that the property only stores the address of the assigned object, without any effect to the reference count. When the object is deleted from the memory, the value of such property will become *nil*, which will prevent the app from crashes if any message is further being sent to this property.

assign - the same as *weak*, however, the value of the property will not become *nil* when the referenced object is released from the memory. This attribute is usually used with non-ARC types or primitive data types.

Atomicity attributes

atomic - ensures that the getter and the setter for the property are generated in a thread-safe way. If two threads simultaneously will call the setter and the getter methods, one of them will be blocked until another one is executed. This is the default value.

nonatomic - opposite to *atomic*, auto-generated getter and setter are not thread safe. Due to the absence of additional thread-safe logic, the getter and the setter work faster.

2.2.8 Primitive types

As soon as Objective-C is a superset over the C language, it supports all the raw types from C. This includes *int*, *double*, *float*, etc. Even though all these types are supported, it is discouraged to use them in favor of Objective-C types which are typedefs on these C types. For example, *NSInteger* is an Objective-C raw type which is bridged to *int* under the hood.

2.2.9 Memory management and ARC

As stated in the official Apple documentation [6], memory management is the process of alllocating memory during program's runtime. More generally speaking, in terms of Objective-C it is a way of distributing ownership among multiple pieces of data. The following chapter is mostly based on the related topic in the documentation.

Object creation

All the objects in Objective-C are created in the dynamic memory, therefore the creation of the object consists of two steps - memory allocation and initialization of its invariants. To allocate the memory for the object message *alloc* is sent, and message *init* is used for object initialization. These two methods can be combined together by using method *new*. Code snippet 16 is used to illustrate these methods.

Code snippet 16. An example of object creation.

```
MyClass *firstObject = [[MyClass alloc] init];
//Object is created and ready for use
MyClass *secondObject = [MyClass new];
//Object is created and ready for use
```

Lifetime of an object

Objective-C uses reference count paradigm to manage object creation and destroy. Any object has an associated integer counter, which represents how many different pointers reference it. Objects which reference the object are responsible for incrementing and decrementing the counter. When the counter is decreased to the value of 0, the object receives message *dealloc*, and runtime assumes that it is released from the memory.

When class method *alloc* is sent, the object reference counter automatically increments to the value of 1. Method *retain* increases this value by one, *release* decreases the value by one. These methods are implemented by *NSObject*. The correct

work with reference counter ensures that the app does not have memory leaks and overall behaves correctly. Usually, this work is done in getter and setter methods. Code snippet 17 shows the example of working with reference counter. The method shown in the snippet is the setter for the invariant of type *NSString* (which is an object).

Code snippet 17. An example of working with reference counter.

```
- (void)setText:(NSString *)text {
    [text retain];
    //increment reference counter of the received object
    [_text release];
    //decrement reference counter of current invariant
    _text = text;
    //set the value of the invariant to a new pointer
}
```

Retain method here ensures that *text* variable will not be released immediately, and *release* decrements reference counter of the invariant to release it from memory.

Retain cycle

Retain cycle is a condition when two objects keep a reference to each other and are retained. It creates a retain cycle since both objects try to retain each other, making it impossible to release any of them. After releasing one of the objects its retain count will not be zero as it is being retained by another object and vice versa. This causes memory management issue. To resolve a retain cycle, one of the retained objects should use a weak pointer to another, this way the release of these objects will be possible.

Automatic reference counting

Starting from Xcode 4.2 Apple LLVM compiler comes with a mechanism of Automatic Reference Counting (ARC). As described in the LLVM documentation [7], It is not a garbage collection mechanism, controversially ARC works in the build time during code analysis. During compilation, ARC analyzes the code and automatically places *retain* and *release* calls where is needed. This simplifies the memory management process, reducing it to the correct use of property attributes. It does not provide a cycle collector, therefore lifetime of the objects must be explicitly managed by the programmer, breaking cycles manually or with *weak* or *unsafe* references.

2.2.10 Runtime

Objective-C was conceived as an add-in to the C language, adding to it the support of the object-oriented paradigm. In fact, Objective-C has a fairly small set of keywords

and control structures over ordinary C. As stated in the documentation guide [8], the main power of the language is the runtime library, which provides a set of functions that extends it, realizing its dynamic capabilities and running OOP. The runtime library is open-source and can be found on Apple's open source website [9].

The functions and structures of the runtime library are defined in several header files: objc.h, runtime.h, and message.h. Firstly, let us look at the file objc.h and determine what the object is in terms of the runtime. Relevant header part is shown in the Code snippet 18.

Code snippet 18. Object representation in objc.h.

```
/// An opaque type that represents an Objective-C class.
typedef struct objc_class *Class;

/// Represents an instance of a class.
struct objc_object {
    Class isa  OBJC_ISA_AVAILABILITY;
};
```

The object in the process of the program is represented by an ordinary C-structure. Each Objective-C object has a reference to its class - *isa* pointer. In turn, the class also represents a similar structure. Relevant header part is shown in Code snippet 19.

Code snippet 19. Class representation in objc.h.

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;
};
```

A class in Objective-C is an object and it also has an isa-pointer to the "class of a class", the so-called metaclass in terms of Objective-C. Similarly, C-structures are defined for other entities of the language. Corresponding lines in the objc.h header are shown in Code snippet 20.

Code snippet 20. Other C-structures from objc.h.

```
/// An opaque type that represents a method in a class definition.
typedef struct objc_method *Method;

/// An opaque type that represents an instance variable.
typedef struct objc_ivar *Ivar;

/// An opaque type that represents a category.
typedef struct objc_category *Category;

/// An opaque type that represents an Objective-C declared property.
typedef struct objc_property *objc_property_t;

/// An opaque type that represents a method selector.
typedef struct objc_selector *SEL;
```

In addition to defining the basic structures of the language, the library includes a set of functions that work with these structures. They can be conditionally divided into several groups:

- Manipulating classes
- Creating new classes
- Introspection
- Manipulating objects

Messaging

As described in the documentation [8], messages in Objective-C are not bound to methods' implementation until runtime. The compiler converts a message expression into a call on a messaging function *objc_msgSend*. This function takes the receiver and the name of the method mentioned in the message (the method selector) as its two principal parameters. If method takes any arguments, these are also passed to the *objc_msgSend*. The prototype of the function is defined in Code snippet 21.

Code snippet 21. Prototype of objc_msgSend.

```
id objc_msgSend (id self, SEL _cmd, id arg1, ...);
```

The call to *objc_msgSend* initiates the process of finding the implementation of the method corresponding to the selector passed to the function. The implementation of the method is searched in the so-called class dispatching table. The possible simplified implementation of the *objc_msgSend* is shown in Code snippet 22.

Code snippet 22. Theoretical implementation of objc_msgSend.

```
id objc_msgSend (id self, SEL _cmd, id arg1) {
    IMP methodFunction = [[self class] methodForSelector:_cmd];
    return methodFunction(self, _cmd, arg1);
}
```

Since a process of finding a method in the dispatch table can be quite long, each class has an associated methods cache. After the first call to any method, the result of the search for its implementation is cached. This is schematically shown in Figure 1.

Selector index	Selector string
14	addObject:
318	insertObject:atIndex:
418	count

Figure 1. Methods' cache illustration.

If the implementation of the method is not found in the class itself, the search continues up the inheritance hierarchy - in the superclasses of that class. This process is illustrated in Figure 2.

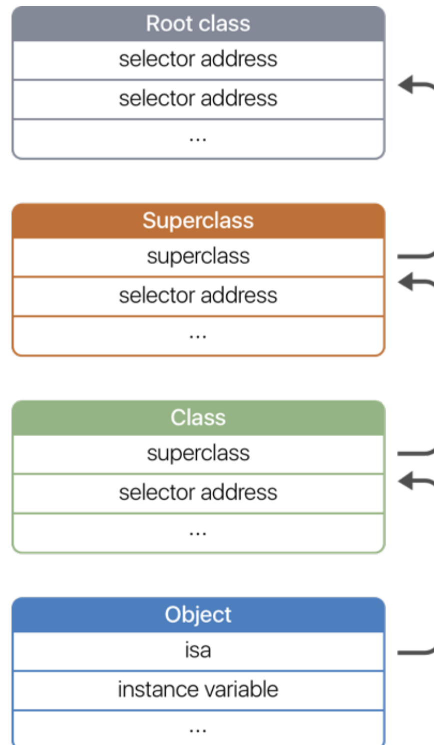

Figure 2. Querying classes' hierarchy for method's implementation.

If the result is not found when searching the hierarchy, the dynamic search mechanism is used - one of the following methods is called: *resolveInstanceMethod* or *resolveClassMethod*. That enables a class to dynamically add a method implementation during runtime. To do that, the class should implement one of these methods, in which add a method using runtime library and return *YES*, indicating that runtime should restart the message send. An example of this is shown in Code snippet 23.

Code snippet 23. An example of dynamic method addition.

```
+ (BOOL)resolveInstanceMethod:(SEL)aSelector {
    if (aSelector == @selector(myDynamicMethod)) {
        class_addMethod(self, aSelector, (IMP)myDynamicIMP, "v@:");
        return YES;
    }

    return [super resolveInstanceMethod:aSelector];
}
```

If runtime can not find a method implementation using described methods, it switches to its last resort – message forwarding. This enables the receiver to forward method execution to a different object.

The runtime sends the object a *forwardInvocation:* message with an *NSInvocation* object, which encapsulates the original message and the arguments that were passed with it. If the class wants to redirect the message, it should implement a *forwardInvocation:* method. An example of this redefinition is shown in Code snippet 24.

Code snippet 24. An example of forwarding an invocation.

```
- (void)forwardInvocation:(NSInvocation *)anInvocation {
    if ([someOtherObject respondsToSelector:[anInvocation selector]]) {
        [anInvocation invokeWithTarget:someOtherObject];
    } else {
        [super forwardInvocation:anInvocation];
    }
}
```

Method swizzling

As explained by Mattt Thompson in his article ”Method Swizzling” [10], method swizzling is the process of changing the implementation of an existing selector during a runtime. It is especially useful with the classes, which implementation is hidden during compile time. This technique uses the peculiarities of the runtime described in the previous chapter. Consider Code snippet 25 which holds a *NSMutableArray* category and adds a logging functionality when the object is added to the array.

Code snippet 25. An example of method swizzling.

```
@implementation NSMutableArray (Logging)
+ (void)load {
    Class class = [self class];

    SEL originalSelector = @selector(addObject:);
    SEL swizzledSelector = @selector(custom_addObject:);

    Method originalMethod = class_getInstanceMethod(class, originalSelector);
    Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);

    class_addMethod(class, originalSelector,
                    method_getImplementation(swizzledMethod),
                    method_getTypeEncoding(swizzledMethod));

    class_replaceMethod(class, swizzledSelector,
                        method_getImplementation(originalMethod),
                        method_getTypeEncoding(originalMethod));
}

- (void)custom_addObject:(id)object {
    [self custom_addObject:object];
    NSLog(@"Add object: %@",  object);
}
@end
```

First of all, we need to add our custom method to a class, this is done using *class_addMethod* function, and then we can replace methods in the dispatch table using *class_replaceMethod*. Another approach of doing that is by using function

method_exchangeImplementations. Lastly, we need to implement our swizzled method *custom_addObject*. We need to call *custom_addObject* in its body, so that the original method is invoked. It can be slightly confusing, however, such method invocation would call original *addObject:* method as soon as we already changed their implementations. It is important to notice that the swizzling method affects global class' state and, therefore, we need to minimize the possibility of race conditions. Doing swizzling in the *load* method guarantees that it would be done just after the class initialization (just before it was used for the first time).

Associated objects

Categories in Objective-C can extend class functionality by adding extra methods, however, they are incapable of adding extra invariants, which can be quite useful sometimes. As described by Colin Wheeler in his article "Understanding the Objective-C runtime" [11], it is possible to achieve such behaviour with the use of the runtime library. Consider that we want to add a new property to a standard class *UITableView* - a reference to the placeholder that will be displayed when the table is empty. The possible implementation of that with the help of runtime is shown in Code snippet 26.

Code snippet 26. An example of associated objects use.

```
@implementation UITableView (Placeholder)
- (void)setPlaceholderView:(UIView *)placeholderView {
    objc_setAssociatedObject(self, @selector(placeholderView), placeholderView,
                             OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (UIView *)placeholderView {
    return objc_getAssociatedObject(self, @selector(placeholderView));
}
@end
```

Here we use *objc_setAssociatedObject* and *objc_getAssociatedObject* functions to respectively set and get the needed object. We pass *@selector(placeholderView)* as a key to these functions. The key basically is a *const char ** value (array of chars in C language) and can be whatever we want, however, it should uniquely identify the associated object, therefore the use of selector in this case is quite handy.

2.3 Standard frameworks and tools

According to Apple documentation, iOS technologies can be represented as layers. Lower layers represent fudamental services and technologies, and higher layers are absract constructs built on these services. This allows developers to implement many complex tasks on the high abstraction level, reducing the amount of work that they would do if they worked on the lower level. However, some low level APIs are also available for the developers, which is especially useful for specific tasks when working with graphics, network and other complex areas. Technology layers are shematically shown in Figure 3.

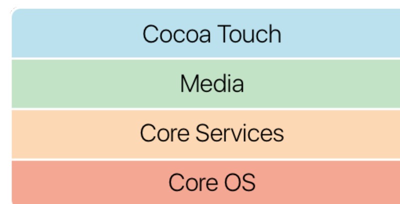

Figure 3. Layers of iOS technologies.

Most of system interfaces are delivered in the form of frameworks – packages containing dynamic shared library and associated resources. The complete list of supported frameworks is long and we are not going to cover all of them. Instead, we are going to focus on two most commonly used frameworks – Foundation framework and UIKit framework. The former provides Objective-C classes for basic data management, and the latter provides key infrastructure for implementing Graphical User Interface (GUI).

2.3.1 Foundation

The Foundation framework defines a base layer of Objective-C classes. That includes the root object class, classes representing basic data types such as strings and byte arrays, collection classes for storing other objects, classes representing system information such as dates, and classes representing communication ports. Most of the classes have intuitive API, and it would be pointless to describe how to add an object to an array or how to create a substring from a string. Methods' names in Objective-C are long and rich, enabling the reader to understand the meaning of the method without knowing it beforehand.

2.3.2 UIKit

The UIKit framework is the central UI framework in the iOS SDK. It provides the classes needed for the user interface and UI related events handling. It encapsulates all the objects required to support the UI of the app – windows, views, buttons, labels, controllers, and much more. Abstractly all the classes can be divided into two categories – views and controllers. Views provide the functionality of drawing and handling user interaction. Controllers are used to manipulate views. Developing an app usually requires subclassing of UIKit classes to override the standard behavior and inject effects into the lifecycle of these objects. Some basic controllers and views are used more frequently than others; they will be covered in this chapter.

UIViewController

As stated in the documentation [12], a view controller is responsible for managing a set of views that build up the portion of the app's UI. Usually, each view controller is assigned an independent full-size screen of the app, for example, a login screen would require a distinct view controller. A view controller has an associated view (stored in the property *view*). This view serves as a container for any other UI components. A view controller manages the position of views stored in its view, fills them with data, and responds to user events coming from them. It also handles a data coming from other objects, communicates with other view controllers and creates new ones if needed. Subclassing *UIViewController* enables the programmer to receive view-related system notifications. That is done through the overriding of standard UIViewController's methods responsible for view's lifecycle. For example, methods *viewWillAppear* and *viewWillDissappear*. The full list of these methods and a related diagram are shown in Figure 4.

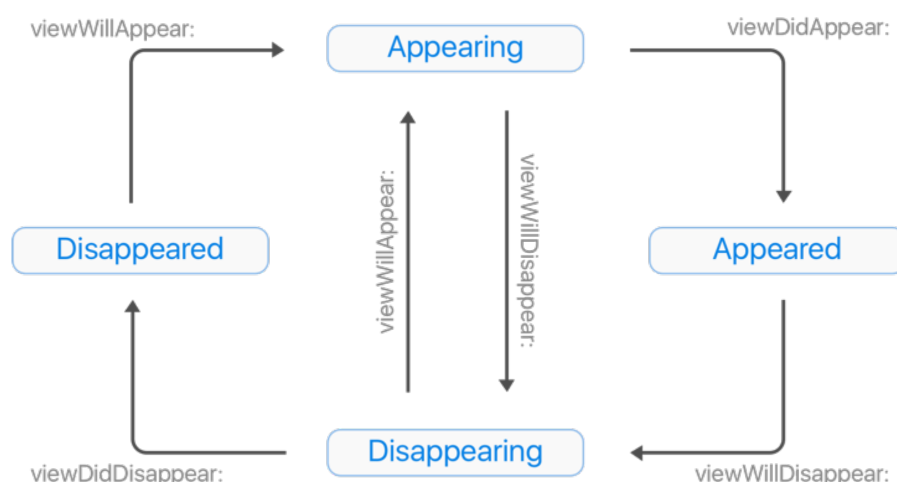

Figure 4. View lifecycle.

This methods' list is not exhaustive. *UIViewController* supports a significant amount of different notification-like methods and multiple protocols. Listing all of these methods would be pointless, however, the overall concept of its behavior and purpose should now be apparent.

UINavigationController

As stated in the documentation [13], a navigation controller is used to set up a navigation between view controllers in a hierarchical manner. To support this, the first view controller in the app should be set up as the root view controller of the navigation controller. Then navigation controller can be used to insert new controllers by using method *pushViewController:animated:.* After the method invocation, pushed view controller is added to the navigation stack and displayed with a preferred animation. Hierarchy, in this case, means that the navigation inside a navigation controller works like a stack – it is linear, it has root view controller, and it supports addition of new view controllers as well as removal of the last view controller currently in the stack. Navigation stack is accessible through the property *viewControllers* of a navigation controller; it is an array of displayed controllers. Naturally, storing view controllers inside an array leads to their retention in the memory, so even not currently displayed controllers can still receive notifications, delegate methods invocations and thus they continue to live and work.

Navigation controller also supports a navigation bar – a control displayed on top of the view, storing the title of currently displayed view and buttons at left and right sides. All the elements of navigation bar are customizable. Usually, the left button is reserved to perform a back or cancel action – popping currently displayed view controller.
Apart from displaying view controllers with the help of a navigation controller, it is also possible to use modal presentation. Instance method *presentViewController:animated:* of *UIViewController* class is used to accomplish that. In case of modal presentation, currently used navigation stack is not modified. This type of presentation is mainly used to show small controllers like menus or to support particular kinds of animations. In general, the choice regarding needed presentation type is based on the navigation logic of the app, UI style, and content that each view controller stores.

UITableView

A table view in contrast to objects described before is not a controller but a view. As described in the documentation [14], It is used to display content in a single column in a form of rows. As soon as *UITableView* is a subclass of *UIScrollView* it automatically provides scrolling behavior. A table view is usually meant to display lists of similar data – for example, a list of contacts. Each data entry in the table view is a separate view called a cell – a subclass of *UITableViewCell* class. *UITableView* has a powerful performance optimization built-in – the mechanism of cells' reuse. Each table view has

an associated pool of reusable cells. Before working with a table view, it is needed to register cells’ subclasses or Interface Builder files to be associated with that table view by the use of reuse identifiers. A reuse identifier is a string, which uniquely describes a sub-pool of cells. Reuse identifiers are used to obtain cells from a table view’s pool. The internal logic of a table view regulates how cells are rendered and when. Cells that go off screen are removed from the table view and are added to the reusable pool. When the user starts scrolling, and the new cell is needed, it is not being created from scratch, and instead, table view obtains it from the reusable pool. The cell returned from the pool is just an old cell that was previously used and went off screen. If all the cells standing behind one reuse identifier are composed in the same way, the only job to display a new cell is to fill it with data. A table view does not need to create view objects, perform a complete layout of cell’s views, and calculate their sizes every time, which makes the process much easier and quicker.

The API used to fill a table view is based on the delegate pattern. There are two major protocols – *UITableViewDataSource* and *UITableViewDelegate*. The first one is used to provide a table view with data, and the second one is used to customize the look of a table view and to inject some custom behaviors based on events happening to a table view.

2.3.3 Concurrency and libdispatch

With the proliferation of multicore CPUs increases the need of making the app concurrent, taking all advantages of modern processors. As explained in the documentation [15], although iOS is capable of running multiple programs in parallel, most of these programs run in the background mode, which eventually blocks their execution from active system use. Only the app that is in the foreground and keeps user's attention is allowed to fully load the hardware. Even though only the active app is allowed to take most of the hardware resources, if it has a lot of work to do but keeps only a fraction of the available cores occupied, these extra processing resources are wasted. Each application in iOS is made up of one or more threads, each of which represents a single path of execution through the application's code. Every application starts with a single thread, which runs the application's *main* function. This thread is called a *main thread* and it is only destroyed when the application is terminated. Applications can spawn additional threads to balance work between CPU cores. When an application spawns a new thread, that thread becomes an independent entity inside of the application's process space. Each thread has its own execution stack and is scheduled for runtime separately by the kernel. There are multiple APIs in iOS that allow working with threads directly - for example, *POSIX* threads or *NSThread*, however, these APIs are not frequently used in favor of more abstract APIs working with thread pools. Threads are a low-level tool that must be managed manually. Given that the optimal number of threads for an application can change dynamically based on the current system load and the underlying hardware, implementing a correct threading solution becomes extremely difficult. In addition, the synchronization

mechanisms typically used with threads add complexity and risk to software designs making it more error-prone. The use of dispatch queues abstracts away the idea of threads by letting the system to perform their creation and management and leaving only the payload for the app developer.

Grand central dispatch

Grand Central Dispatch (GCD) is a part of the open source libdispatch library, supporting thread pools or dispatch queues. A dispatch queue is an object-like structure that manages the tasks that are submitted to it. All dispatch queues are first-in, first-out (FIFO) data structures. Thus, the tasks added to a queue are always started in the same order that they were added. There are two types of dispatch queues - serial and concurrent. Serial queues execute one task at a time, delaying execution of any further tasks until the previous is completed. The main queue which is created during the app start is a serial queue. Concurrent queues execute multiple tasks simultaneously, however tasks execution still begins according to FIFO order. There are some predefined concurrent queues created by the system, which differ in the execution priority.

The task is submitted to a queue using block objects, which define a self-contained units of work. There are few possibilities to submit a block to a dispatch queue. All of them are done by use of C functions from the *dispatch* family. There are two functions that are often used - *dispatch_async* and *dispatch_sync*. When *dispatch_async* is called it submits the block to a queue and immediately returns. Controversially *dispatch_sync* waits for the dispatched block to be executed and returns only after that, this blocks the caller thread until submitted block is executed. Code snippet 27 shows the syntax of basic dispatch operations - creating a queue, obtaining a global queue and dispatching a block to a queue.

Code snippet 27. Syntax of basic operations on dispatch_queue objects.

```
dispatch_queue_t myQueue = dispatch_queue_create("com.example.MyQueue",
                                                DISPATCH_QUEUE_SERIAL);
//creates serial queue

dispatch_sync(myQueue, ^{
    //body of dispatched block
});
//dispatches a block synchronously to the created serial queue

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    //body of dispatched block
});
//obtains a default concurrent queue with priority default
//and then dispatches a block asynchronously to it
```

Grand central dispatch is a preferred API for multithreading in iOS since it maintains the high performance, scalability and ease of use which is not possible with raw threads. There many other possibilities for thread management that GCD offer,

however, they are not described here because they are not needed in the context of the thesis.

2.4 Common patterns

iOS development has some unique mechanisms and techniques provided by the system frameworks and Objective-C lanaguage. Some of them, especially related to event handling, are going to be desribed in this chapter.

2.4.1 Key-value observing/coding

Key-value observing (KVO) and key-value coding (KVC) are highly bound together, that is why we are going to cover them together.

Key-value coding

As stated in the documentation [16], key-value coding is a mechanism for accessing an object’s properties indirectly, using strings to identify properties, rather than through invocation of an accessor method or accessing them through instance variables. There is an informal protocol, named *NSKeyValueCoding*, which should be implemented for any class which wants to support *KVC*. *NSObject* class supports this protocol, therefore all other classes which inherit from *NSObject* also conform to *NSKeyValueCoding*.

The syntax for setting any invariant of an object using *KVC* mechanism is shown in Code snippet 28. Here *propertyName* string is the name of the invariant that should be changed and parameter *value* is the value that should be set to this invariant.

Code snippet 28. Syntax for setting a property with KVC.

```
[object setValue:value forKey:@"propertyName"];
```

There is also a way to set values for nested invariants. For example, an object of a class *Employee* has a property of class *Address* named *address*, which in order has a property of class *NSString* named *postalCode*. If it is needed to change a *postalCode* of some particular *Employee* instance, the dot syntax in key path should be used. This example is shown in Code snippet 29. Here dot syntax shows that we are accessing property *postalCode* of the *address* property.

Code snippet 29. Dot syntax for setting nested properties with KVC.

```
[employee setValue:@"193080" forKeyPath:@"address.postalCode"];
```

KVC serves as a backbone for key-value observing.

Key-value observing

Documentation [17] states that key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects. If to recall the example from the previous sub-chapter, *Employee* object may need to be aware of when certain aspects of *Address* instance change, such as the *postalCode*.
If these attributes are public properties of *Address*, the *Employee* could periodically poll the *Address* to discover changes, but this is, of course, inefficient, and often impractical. However, this is a perfect example of the KVO usage.

In order to observe such changes of an object, some requirements should be met:

1. The observed object, the *Address*, in this case, should be KVO compliant
2. The class that will be used to observe the property of another class should be set as an observer.
3. A method named *observeValueForKeyPath:ofObject:change:context:* should be implemented in the observing class.

All *NSObject* children are KVC and KVO compliant by default, so it can be assumed that *Address* inherits from *NSObject*. To register for KVO notification, *Employee* object should call method *addObserver:forKeyPath:options:context:*. This is illustrated in Code snippet 30.

Code snippet 30. Registering for a KVO notification.

```
[self.address addObserver:self forKeyPath:@"postalCode" options:0 context:nil];
```

Here *addObserver:forKeyPath:options:context* is sent to the *address* object, indicating that it should send notifications to *self* (which is the *employee* object) when *postalCode* invariant changes. The *options* parameter, specified as a bitwise OR of option constants, affects both the content of the change dictionary supplied in the notification, and the manner in which notifications are generated, however, it is not relevant for the example. The context pointer in the message contains arbitrary data that will be passed back to the observer in the corresponding change notifications. In this example it is safe to specify *nil* and rely entirely on the key path string to determine the origin of a change notification, however, this approach may cause problems for an object whose superclass is also observing the same key path.

After *Employee* object have sent message *addObserver:forKeyPath:options:context:* to the *address* object, every time *postalCode* changes, *Employee* will receive *observeValueForKeyPath:ofObject:change:context:* message with the information about the change, therefore this method should be implemented. The method is shown in Code snippet 31.

Code snippet 31. Receiving a KVO notifications.

```
-(void)observeValueForKeyPath:(NSString *)keyPath
                     ofObject:(id)object
                       change:(NSDictionary *)change
                      context:(void *)context {
    if ([keyPath isEqualToString:@"postalCode"]) {
        //logic
    }
}
```

Here *keyPath* identifies which property have changed, *object* identifies the object on which this change occurred, and the *change* is a dictionary that describes the changes that have been made to the value of the property. Parameter *context* is the same object which was sent as a *context* attribute when registering for observation.

Dictionary *change* contains multiple values, the supported keys for the dictionary and corresponding values are as follows:

1. *NSKeyValueChangeKindKey* - An *NSNumber* object that contains a value corresponding to one of the *NSKeyValueChange* enum values, indicating what type of change has occurred.
2. *NSKeyValueChangeNewKey* - If the value of the kindKey entry is setting, and *new* was specified when the observer was registered, the value of this key is the new value for the attribute.
3. *NSKeyValueChangeOldKey* - If the value of the kindKey entry is setting, and *old* was specified when the observer was registered, the value of this key is the value before the attribute was changed.
4. *NSKeyValueChangeIndexesKey* - If the value of the kindKey entry is insertion, removal, or replacement, the value of this key is an *NSIndexSet* object that contains the indexes of the inserted, removed, or replaced objects.
5. *NSKeyValueChangeNotificationIsPriorKey* - If the *prior* option was specified when the observer was registered, this notification is sent prior to a change.

Each object registered for notifications should also de-register when it no longer wants to receive notifications. To do so, it should send a message *removeObserver:forKeyPath:* to the object it was listening. Such call related to the previous example is shown in Code snippet 32.

Code snippet 32. Removing a KVO observer.

```
[self.address removeObserver:self forKeyPath:@"postalCode"];
```

Implementation details

Automatic key-value observing is implemented using a technique called isa-swizzling or class-swizzling, a feature possible because of objective-c runtime.
The *isa* pointer points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements,

among other data. When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result, the value of the *isa* pointer does not necessarily reflect the actual class of the instance.

2.4.2 IBOutlets and IBActions

As described in the Xcode help manual [18], Xcode has a tool for building user inteface called Interface Builder (IB). The process of interface designing basically consists of creating different view objects and controls using the builder. This structure then is stored in the Extensible Format Language (XML) format. To connect view objects to the code developer drags a desired control to the text editor. Such connection to a view object is called IBOutlet. Figure 5 illustrates the process of creating an IBOutlet connection.

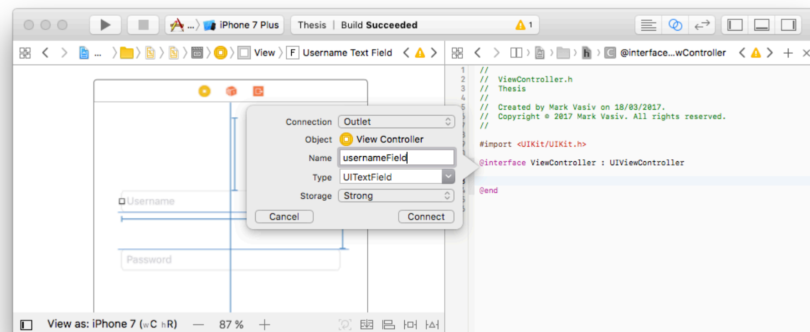

Figure 5. Creating of IBOutlet.

It is also possible to generate a method, which will be fired when a certain control is accessed by the user, for example, a button is being pressed. Such a connection is called IBAction. Figure 5 shows how IBAction is set up.

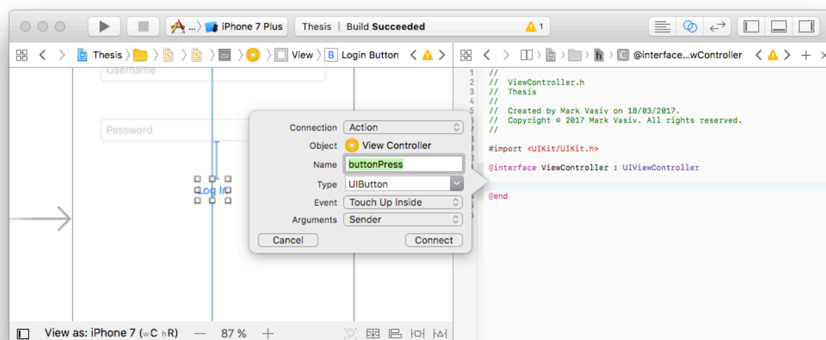

Figure 6. Creating of IBAction.

During runtime the XML is parsed and all the view objects are created automatically by the SDK. XML also stores the references to all IBOutlets and IBActions and fill them when needed. A syntax for declaring IBOutlet and IBAction in code is shown in Code snippet 33.

Code snippet 33. Syntax for IBOutlet and IBAction defenitions.

```
@property (strong, nonatomic) IBOutlet UITextField *usernameField;
- (IBAction)buttonPress:(id)sender;
```

2.4.3 NSNotification

As described in the documentation [19], *NSNotification* provides a mechanism for sending events from one object to multiple. Notification encapsulates information about the occured event. Objects that want to know about specific events should register using *NSNotificationCenter*, which is resposible for handling this type of events. When the event happens, a notification is posted to the notification center, which immediately broadcasts the notification to all registered objects. The mechanism abstractly is similar to the KVO, however, notifications are used for custom events instead of value changes.

To register an object to receive a notification we need to invoke method *addObserver:selector:name:object:* on the notification center, specifying the observer, the selector, which the center should invoke on the observer, the name of the notification it wants to receive, and from which object. If notification name is not specified, the observer will receive all notifications from that object. If object is not specified, the observer will receive notifications with name corresponsing to the specified, regardless of the object associated with it. There is a default notification center used for exhanging events within one process. It can be accessed using the class method *defaultCenter*. To remove the observer from the notification center we need to call method *removeObserver:* or *removeObserver:name:object:*.
To create a notification we need to call *notificationWithName:object:* or *notificationWithName:object:userInfo:* methods. Then we can pass these notification objects to the notification center using *postNotification:*. Consider example of NSNotification usage in Code snippet 34.

Code snippet 34. An example of NSNotification posting and receiving.

```
//ViewController
static NSString *SignUpNotificationKey = @"UserSignedUp";
- (IBAction)signupButtonPress:(id)sender {
    BOOL success = [LoginManager signupWithUsername:self.usernameField.text
                                        andPassword:self.passwordField.text];

    NSNotification *notification = [NSNotification
                              notificationWithName:SignUpNotificationKey
                                            object:nil
                                          userInfo:@{@"Success":@(success)}];

    [[NSNotificationCenter defaultCenter] postNotification:notification];
}


/Somewhere in a different class
- (void)registerForNotifications {
 [[NSNotificationCenter defaultCenter] addObserver:self
                                          selector:@selector(userSignedUp:)
                                              name:SignUpNotificationKey
                                            object:nil];
}

- (void)userSignedUp:(NSNotification *)notification {
    //Handle notification
}
```

Here we create and post a notification when the user presses signup button. We encapsulate a boolean indicating a success of signup into the notification object. And then we register for this notifications. From now, every time user presses the signup button, the method *userSignedUp:* will be fired, containing the success of the signup operation.

2.4.4 Delegation

As explained in the documentation [20], delegation is used when an object needs to communicate with another object to send it some data or to query for some data. The delegation pattern is implemented using protocols. In delegation an object outwardly expresses certain behavior but in reality transfers the responsibility for implementation of this behaviour to the linked object – its delegate. This helps to build systems with clear reposibility scopes.

For example, let us take a look at view controller, which holds a table view and serves as a delegate to this table view. The relevant code is shown in Code snippet 35.

Code snippet 35. An example of the delegate pattern.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.tableView.delegate = self;
    self.tableView.dataSource = self;
}

-(NSInteger)tableView:(UITableView *)tableView
                                        numberOfRowsInSection:(NSInteger)section {
    return 10;
}

-(UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
                                     dequeueReusableCellWithIdentifier:@"cellId"];
    [self fillCellData:cell forIndexPath:indexPath];
    return cell;
}

- (void)tableView:(UITableView *)tableView
                                didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    //Handle cell tap
}
```

First of all, we set *self* as a *delegate* and *dataSource* for the table view. If we take a look at the *UITableView* interface we will see that both of those propoerties have id type but conform to some protocols – *UITableViewDataSource* and *UITableViewDelegate*. These properties are shown in Code snippet 36.

Code snippet 36. *dataSource* and *delegate* properties of UITableView.

```
@property (nonatomic, weak, nullable) id <UITableViewDataSource> dataSource;
@property (nonatomic, weak, nullable) id <UITableViewDelegate> delegate;
```

These protocols store methods used to control the behaviour of the table view. In the view controller we implement all required methods of the *UITableViewDataSource* protocol – *tableView:numberOfRowsInSection:* and *tableView:cellForRowAtIndexPath:*. We also implement an optional method from the *UITableViewDelegate* to handle cell taps. All these methods will be invoked by the table view when it needs to query some data or to provide some data to its delegate.

2.4.5 Blocks

As described by Matt Gallagher in his article ”How blocks are implemented (and the consequences)” [21], block is a self-contained, autonomous code fragment, existing always into the scope of another programming structure, as for example the body of a method. The code on the block can interact with the scope out of it, but what takes place in the block is not visible to the scope out of it.

Blocks are objects, so they can be stored to data structures, as well as be returned from methods, or assigned to variables. The syntax of declaring blocks as local

variables, properties and method argument differs, however, is recognizable and similar. Let us have a look at how to declare a block as a local variable. That is shown in Code snippet 37.

Code snippet 37. An example of block declaration as a local variable.

```
void (^simpleBlock)(void) = ^{
    NSLog(@"Block called!");
};
```

Here from left to right: *void* is a return type of the block, *simpleBlock* is the block name and *void* is the type of input argumetns. ^{ } notation contains a body of the block, which will be excuted when the block is called. The syntax of block calling is very similar to a C function call. A call to a block described in the previous example is shown in Code snippet 38.

Code snippet 38. Syntax of a block call.

```
simpleBlock();
```

An example of a block, which takes parameters and returns a value is shown in Code snippet 34. This block takes two *int* arguments and returns their sum as an *int*.

Code snippet 39. An example of block declaration with input parameters.

```
int (^calculateSum)(int, int) = ^(int a, int b){
    return a + b;
};
```

2.4.6 MVC

As described in the documentation [22], the Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, but also the way objects communicate with each other. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries. Schematically these objects and relations between them are shown in Figure 7.

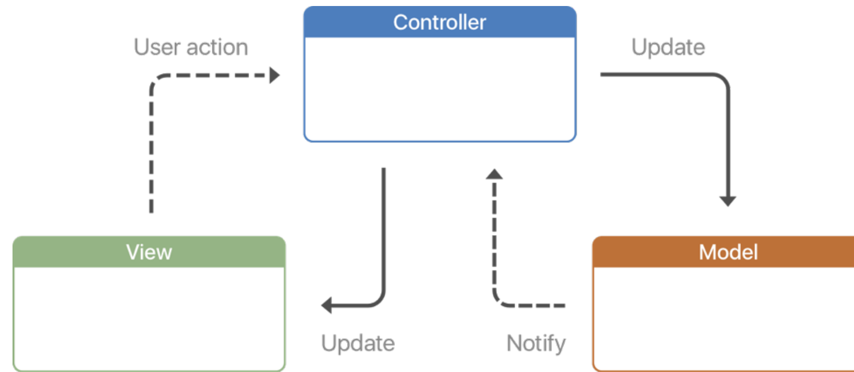

Figure 7. MVC diagram.

Model objects encapsulate the data specific to the application and define the logic and computation that manipulate and process that data. A model object can have to-one and to-many relationships with other model objects.
A view object is an object in the application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data.

A controller object acts as an intermediary between one or more of the application's view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa. Controller objects can also perform setup and coordinating tasks for the application and manage the life cycles of other objects. Conceptually, controller layer serves as a ”brain” for the app connecting view and model layers.

2.5 Common problems

Objective-C and iOS SDK introduce two big arcitecture problems that can be already seen without even building an app. First one is the overload of the controller instance in the MVC pattern (so-called Massive View Controller problem). And second one is the variety of event-based mechanisms with different syntax.

2.5.1 MVC vs MVVM

The strict use of the MVC pattern can lead to the overload of a controller instance. A controller is responsible for multiple not related to each other functionalities, incuding view creaction and handling, view animations, user actions handling and accessing model for quering and saving the data. This clearly overloads a single instance. Even though there are techniques in Objective-C to divide a class into multiple files, an abstractly controller instance still remains overloaded with resposibilities. Another approach that can be used to avoid this problem is to modify the MVC pattern. There

are many other architecture patterns that can be used when developing iOS apps, however, we are not going to go through all of them. The pattern we are going to focus on and oppose to the MVC is Model-View-ViewModel (MVVM).

That is due to the following reasons:
- MVVM is a slight modification of the MVC, which makes it easier to refactor MVC app to use MVVM
- MVVM can expose benefits of the ReactiveCocoa bindings, which are going to be described later

According to the msdn article [23], the MVVM pattern was developed by Microsoft architects in 2005 to simplify event-driven programming of user interfaces. Abstractly the pattern is similar to MVC because it separates model, view and presenter/controller logic. However, there are some important changes that are making this pattern a solution to the discussed controller overload. In the MVVM a model is responsible for data handling, a view is responsible for drawing the interface and performing animations, and a ViewModel serves as a representation of a view from the model perspective. It queries a model for the data, processes it and provides a view with a convenient interface for getting this data. The important aspect of the pattern is that ViewModel is not aware of its view. All the entities and relations are shown in Figure 8.

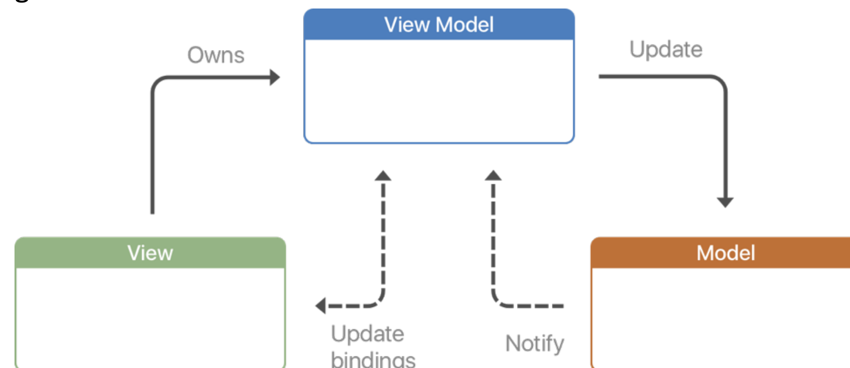

Figure 8. MVVM diagram.

The implementation of MVVM usually suggests that *UView* and *UIViewController* instances represent a view entity. Then, a ViewModel is assigned to each of the views – it can be a subclass of *NSObject*. All the logic related to data processing is then moved from *UIViewController* instance to the ViewModel. That is illustrated in Figure 9.

Figure 9. MVVM in iOS.

2.5.2 Event-based patterns

Another arcitecture problem in Objective-C is the existence of multiple patterns for serving events. These patterns include IBActions, delegates, callbacks, NSNotifications and KVO. All of them abstractly serve the same functionality – firing events when some data or user input changes. However, they have a very different syntax, which creates a problem in the code readability and ease of use. Multiple similar and related events are treated differently in a different parts of the program. iOS SDK does not have an API to uniform these patterns. However, it is clearly possbile to create some layer of abstraction over these patterns to unify them. To acomplish that we are going to use some interfaces of the ReactiveCocoa framework that are going to be covered in the next chapter.

3 FUNCTIONAL REACTIVE PROGRAMMING

Functional Reactive Programming (FRP) is a declarative programming paradigm that abstracts away the idea of mutable value with the help of signals. Signals encapsulate the semantics of a mutable value that changes over time. This compound data type allows capturing the temporal aspect of mutability better. Signals can also be transformed with the help of functional operators.

In contrast to a traditional imperative programming, the declarative style of FRP hides many not-important details, allowing programmers to do more with less code. The original formulation of FRP can be found in the paper by Paul Hudak and Connan Elliot [24], where they built an FRP system using Haskell language. Later works by Hudak and others at Yale [25] [26] [27] demonstrate the evolution of the concept, modifying some of the original ideas aiming to resolve inefficiencies of classical FRP. One of the ideas proposed in these works affects the behavior of signals. The signals are described as discrete entities that only change on events. That transforms the paradigm to be event-driven, which may seem a too restrictive approach. However, as it turns out many potential applications of FRP are event-oriented. That includes UI interfaces as well as mobile apps more general. FRP combines aspects of more specific functional and reactive programming paradigms.

3.1 Functional programming

Functional Programming (FP) tries to minimize the amount of the state by the immutability of objects and to avoid side effects. Each function is designed to be a clean mathematical function – it does not change any external state. That means that each execution of the function with the same input will produce the same output. It is common with the OOP that methods modify the state of the object, and this manipulation often cannot be predicted by the method’s user. The result of these two global differences in the FP is the ability to write maintainable, clear, self-definitive and safe code. However, it is a problematic way to program mobile applications since mobile apps heavily work with the user interaction, which ultimately leads to needing to maintain and mutate a state. That is why using clear functional programming in most of the client apps is not possible. And that is why we need reactive programming.

3.2 Reactive programming

The main building block of reactive programming is a data stream, which represents a value change over time. That stream is an asynchronous mechanism that is used for event propagation. Whenever a value of the variable changes (event happens), that value is being transmitted to another part of the program using operators specified by

the programmer. That means, for example, that the change of user interface can automatically be propagated to the model, changing some of its properties.
The possibility to express the intention in a short and declarative way enables developers to minimize the amount of mutable state. Objects can automatically change bound objects, without the need to explicitly command them to do so in the body of functions.

3.3 Composition of paradigms

A composition of functional and reactive paradigms let us use reactive data streams with short functional operators, making it easy to modify them achieving the desired data flow. As soon as the resulting paradigm is based on events, it is possible to overcome the problem of multiple event-based patterns in Objective-C. Moreover, reactive bindings let us automatically bind variables in ViewModel and view entities of the MVVM pattern.

3.4 Support in iOS

FRP is an abstract concept that can be used as a paradigm for building any software, however, the support in any specific language or platform differs. Unfortunately, iOS SDK cannot provide tools for creating such software, and we apparently cannot modify the SDK's code. Therefore, the only possible way to imply the paradigm is to build a new layer of abstraction over the SDK hiding its object-oriented nature. Multiple frameworks are providing this functionality for many systems and languages – Rx family, ReactNative, ReactiveCocoa. When we choose one, it is essential to understand that the abstraction level and the overall app performance usually correlate: the more abstract the code - the lower the performance. Hence we need to find an optimal balance between these attributes. ReactiveCocoa framework integrates with the SDK modifying some of its public interfaces and provides its own interfaces for building reactive streams. Even though it extends standard classes, the framework does not serve as the complete proxy between the app logic and the SDK. It is done with the help of Objective-C dynamic nature – its ability to seemingly modify existent classes without the need to access their source code. The whole language is powered by the runtime library, so using it to add some functionality should not impact the performance. Therefore, it can be considered as a good option for iOS development using Objective-C.

3.5 ReactiveCocoa

As stated by Mattt Thompson [28], ReactiveCocoa is an open source library that brings Functional Reactive Programming paradigm to Objective-C. It was created by Josh

Abernathy and Justin Spahr-Summers in the development of GitHub for Mac. The framework was inspired by .Net Reactive Extensions library.

3.5.1 Basic classes

As explained in the ReactiveCocoa documentation [29], the central class of the ReactiveCocoa is *RACSignal*. This class represents a data stream in the app. The signal encapsulates information about mutable data change over time. More simply, it sends events to its subscribers. There are three possible types of events: *next*, *completed* and *error*. The convention says that any signal can send multiple *next* events, and only one *completed* or *error* event. After *completed* or *error* is sent no more *next* events will be sent. The objects that want to receive events coming from a signal should subscribe to that signal with the help of multiple methods of *subscribe* family, for example, *subscribeNext:*.

The signals can unify all event-driven patterns in the general iOS development. Indeed, many standard classes have corresponding methods to provide a *RACSignal* interface instead of conventional one. For example, a category of *UIButton* supplied by RAC has method *rac_signalForControlEvents:* which serves as a replacement for a standard IBAction.

Apart from obtaining *RACSignal* instances from categories of standard classes, the instance can be created manually with the help of class method *createSignal:*. That is usually done to create a request-like object. Consider Code snippet 40 with an example of manual signal creation. The method *createSignal:* takes a block argument, which is going to be the body of the signal. This block is going to be called for each subscription. In the example, the signal is designed to handle login operation. The signal block is used to make an actual request and to send response to the subscriber, which it receives as a parameter. The retun vaue of the method is *RACDisposable* object. It is an object that holds instructions on how resources should be cleaned up after the signal's deallocation. In this case we want to cancel the request if it did not finish upon deallocation of the signal.

Code snippet 40. An example of manual RACSignal creation.
```
RACSignal *loginSignal = [RACSignal
                          createSignal:^RACDisposable *(id<RACSubscriber> subscriber) {
    NSError *error;
    NSString *requestId = [self loginUser:username password:password error:&error];
    if (!error) {
        [subscriber sendNext:@YES];
        [subscriber sendCompleted];
    } else {
        [subscriber sendError:error];
    }
    return [RACDisposable disposableWithBlock:^{
        [self cancelRequest:requestId];
    }];
}];
```

Code snippet 40 (continued).

```
[loginSignal subscribeNext:^(id x) {
    //handle login success
} error:^(NSError *error) {
    //handle login error
}];
```

Another exciting class coming with ReactiveCocoa is *RACTuple*. It is a representation of a tuple in many other languages. The tuple ships with two associated macros – *RACTuplePack* and *RACTupleUnpack*, which are used to create a tuple or to unwrap its values. *RACTuple* behind the scenes is powered by *NSArray* class, and it supports obtaining elements by index. Despite *NSArray*, *RACTuple* has built-in security optimizations, for example, not allowing to get an out-of-index exception.

ReactiveCocoa has many macros for common operations. Two the most used are *RAC* and *RACObserve*. *RAC* is used to set up a reactive binding theoretically described earlier – it automatically updates a value of a variable standing by passed key path according to the next value sent by a signal. *RACObserve* is used as a replacement for KVO API. It uses the same KVO but provides a *RACSignal* interface instead. *RACObserve* also takes keypath as a parameter and creates a signal which sends values when the property value changes. Consider an example of these macros use in Code snippet 41. Here the value of label's *text* is bound to the model's *title* variable. That means that every time string title is changed, the text of the label is updated.

Code snippet 41. An example of RAC and RACObserve macros.

```
RAC(self.titleLabel, text) = RACObserve(self.viewModel, title);
```

Keypaths that are passed to described macros are not *NSString* keypaths that are used with Cocoa Touch API's. Instead, they are variable names that are then translated by the ReactiveCocoa to *NSString* keypaths. That ensures type-safety, enabling the programmer to identify the desired object correctly.

Apart from signal and data representation, ReactiveCocoa has a class for handling multithreading. The class is called *RACScheduler* and is used to schedule work encapsulated in the block. The mechanism is similar to the one used with GCD. Indeed under the hood *RACScheduler* stores a pointer to a *dispatch_queue*, which is used to perform the needed work. Schedulers are useful when working with *RACSignal's* operators to easily balance different blocks between different queues.

3.5.2 Basic operators

The power of signals comes with functional operators that can be applied to them. Operators are used to modify the behavior of a signal. Syntactically they are *RACSignal's* instance methods, which return a new *RACSignal* instance. That enables us

to chain operators together, building a complex data stream handlers. There are many operators, some of which are common to other functional-style frameworks, and some are not. We will not go through all of them but will examine the most common ones.

Operator *filter* is respectively used to filter *next* events going through the signal. A block passed to the operator receives a *next* event and returns a boolean indicating whether this event should be passed. The example in Code snippet 42 shows how the *filter* operator is used to pass only even numbers.

Code snippet 42. Example of filter operator use.

```
RACSignal *filtered = [signal filter:^BOOL(NSNumber *number) {
    return (number.integerValue % 2 == 0);
}];
```

Another commonly used operator is *map*. It is used to map each *next* value of the signal to a new object. Code snippet 43 shows how it can be used to map all even numbers to the string “Even” and all odd numbers to the string “Odd”.

Code snippet 43. Example of map operator use.

```
RACSignal *mapped = [signal map:^id(NSNumber *number) {
    if (number.integerValue % 2 == 0) {
        return @"Even";
    } else {
        return @"Odd";
    }
}];
```

The operator *flattenMap* is somehow similar to map operator, but it can only return an instance of a *RACSignal* in its block. The operator is mainly used to chain independent operations together. For example, we can use it to perform multiple network operations. Code snippet 44 shows how it is done by chaining *login* and *loadContacts* operations.

Code snippet 44. An example of the flattenMap operator use.

```
[[[[self loginUser:user password:pass]
    flattenMap:^__kindof RACSignal *(NSString *userId) {
        return [self loadContactsOfUser:userId];
    }]
    deliverOnMainThread]
    subscribeNext:^(NSArray *contacts) {
        //Display contacts
    }];
```

Here method *loginUser:password:* returns a signal, which is going to perform login operation when subscribed to and will send *userId* after successful login. Method *loadContactsOfUser:* also returns a signal, and is used to perform loading of contacts, it sends an array of loaded contacts. We chain two signals together with a help of *flattenMap* operator so that when the first signal succeeds we automatically switch to the next signal. We also use operator *deliverOnMainThread*, which will pass all the

next events to the main queue. And finally, we subscribe to the resulting signal to handle loaded contacts. This short declaration enables us to chain independent operations, perform them on different queues and then pass the result to the main thread automatically.

In Code snippet 44, we also used operator *deliverOnMainThread*. That is a specific operator derived from the *deliverOn*. Operator *deliverOn* takes arbitrary *RACScheduler* as an argument and is used to pass the events of the resulting signal to the scheduler. *deliverOnMainThread* is the same *deliverOn* with the main thread scheduler.
Some operators are applied not to one signal but to multiple. For example, a *merge* operator is used to combine values from multiple signals. If operator *merge* is applied, the resulting signal will send events from all passed signals. Example 45 shows the syntax of *merge* operator use. Here the resulting merged signal will send values from *signal1*, *signal2*, and *signal3* exactly when they do.

Code snippet 45. An example of merge operator use

```
RACSignal *merged = [RACSignal merge:@[signal, signal2, signal3]];
```

Operator *combineLatest* is similar to merge operator – it also takes multiple signals and combines their values. The difference is that combineLatest sends *RACTuple* objects with the latest values from all signals. The value is emitted from the signal whenever each of the source signals sends new value. The resulting signal would send a value if only each of combined signal sent at least one *next*. The syntax for *combineLatest* operator is shown in Code snippet 46.

Code snippet 46. Example of combineLatest operator use.

```
RACSignal *combined = [RACSignal combineLatest:@[signal, signal2, signal3]];
```

Another abstract subset of operators are the ones responsible for injecting side effects. Visually they look similar to subscription methods, however, they are not performing any subscription at all. Instead, they are used to inject work in the form of the block that is executed when corresponding event occurs. The side effect is performed only in case of the subscription. The goal of these operators is to isolate side effects into a definitive structure, which explicitly points that this is a side effect. In the FRP style programming this is especially helpful because we try to minimize side effects. Because that is not entirely possible with client programming based on OOP language, we can use the mechanism of these operators to distinguish side effects from regular functions' behavior making code clearer. The example of these operators usage is shown in Code snippet 47.

Code snippet 47. An example of doNext operator usage.

```
[[signal doNext:^(id x) {
    //perform side effects
}] subscribeNext:^(id x) {
    //perform actual work
}];
```

Here we use operator *doNext* to perform side effects. The block passed to the operator will be called for each *next* event. After the use of the operator, we subscribe to signal to perform a standard behavior.

3.6 Syntactical difference between ReactiveCocoa and iOS SDK

Even though ReactiveCocoa is an addition to the iOS SDK and is built on top of the same frameworks and Objective-C, it can provide a clear functional reactive code style. Let us have a look at the example of form validation logic shown in Code snippet 48. This code is written without the help of ReactiveCocoa. Here the logic is fragmented across multiple methods. In a standard lifecycle callback *viewDidLoad*, we set our view controller as a delegate for the input text fields. Then we listen for a delegate's method *textFieldDidEndEditing:*, use method *isFormValid* for validating text fields and enabling *signupButton*. And finally, we use *signupButtonPress:* to signup a new user.

Code snippet 48. Conventional validation of a signup form.

```
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    self.usernameField.delegate = self;
    self.passwordField.delegate = self;
    self.passwordConfirmationField.delegate = self;
}

- (BOOL)isFormValid {
    return self.usernameField.text.length > 0 &&
           self.passwordField.text.length > 0 &&
           [self.passwordField.text isEqual:self.passwordConfirmationField.text];
}

- (void)textFieldDidEndEditing:(UITextField *)textField {
    self.signupButton.enabled = [self isFormValid];
}

- (IBAction)signupButtonPress:(id)sender {
    [LoginManager signupWithUsername:self.usernameField.text
                         andPassword:self.passwordField.text];
}
@end
```

Now let us re-make the whole logic in terms of the ReactiveCocoa. Code snippet 49 shows how it could be done. Here, all of the logic for validating form input is contained in a single signal's chain. Each time any of the text fields are updated, their inputs are reduced to a single boolean value. The resulting signal then is used to enable or disable the signup button automatically. And just after that declaration we can handle the button tap. All of the logic is contained in a single method *viewDidLoad*. That is making the logic clear, declarative, self-contained and easy to modify.

Code snippet 49. Reactive validation of a signup form.

```
RACSignal *formValid = [RACSignal
  combineLatest:@[
    self.usernameField.rac_textSignal,
    self.passwordField.rac_textSignal,
    self.passwordConfirmationField.rac_textSignal]
  reduce:^id(NSString *name, NSString *pass, NSString *confirm){
      return @(name.length>0 && pass.length>0 && [pass isEqual:confirm]);
  }];

RAC(self.signupButton, enabled) = formValid;

[[self.signupButton rac_signalForControlEvents:UIControlEventTouchUpInside]
  subscribeNext:^(id x) {
    [LoginManager signupWithUsername:self.usernameField.text
                         andPassword:self.passwordField.text];
  }];
```

4 MESSENGER DESIGN

The app development process usually starts with the design of user interface and architecture. To better understand the overall concept of the app let us start by defining the user interface and possible user stories.
As schematically illustrated in Figure 10, the app is going to have a tab bar with three tabs – contacts, chats, and settings. Contacts and chats tabs encapsulate lists of corresponding objects, whether settings tab stores controls used for data generation.

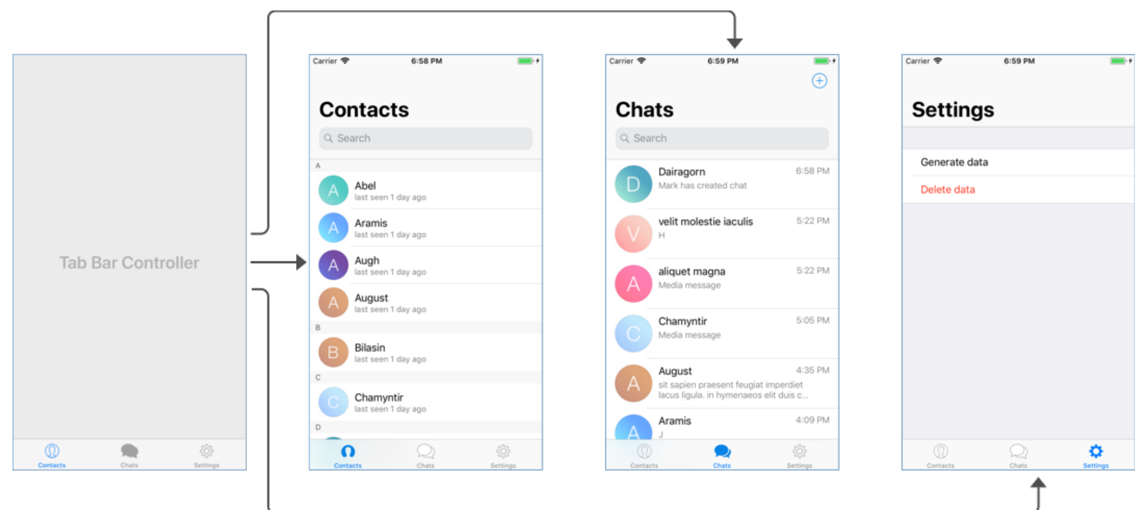

Figure 10. Root views of the app.

Now let us have a closer look on each of the tabs and views that could be accessed from them.

4.1 Contacts list

Contacts tab is going to be called contacts list from this moment. The list contains a table view showing a list of the user’s contacts, and a search bar. Table view cell contains multiple UI elements that display the contact’s information – name, avatar, and date when the contact was last seen in the app. All the cells are divided into sections by the first letter in the contact‘s name. Cell tap opens a chat with the selected contact or creates one if it does not exist and then opens it. When text is entered to the search bar, search results view is displayed with all the contacts matching the search query. Table view cells in the search display view are almost the same as in the contacts list. However, they lack the last seen date and are not divided into sections. Cell tap, as previously, opens the chat with the corresponding contact. Complete UI and described above actions are shown in Figure 11.

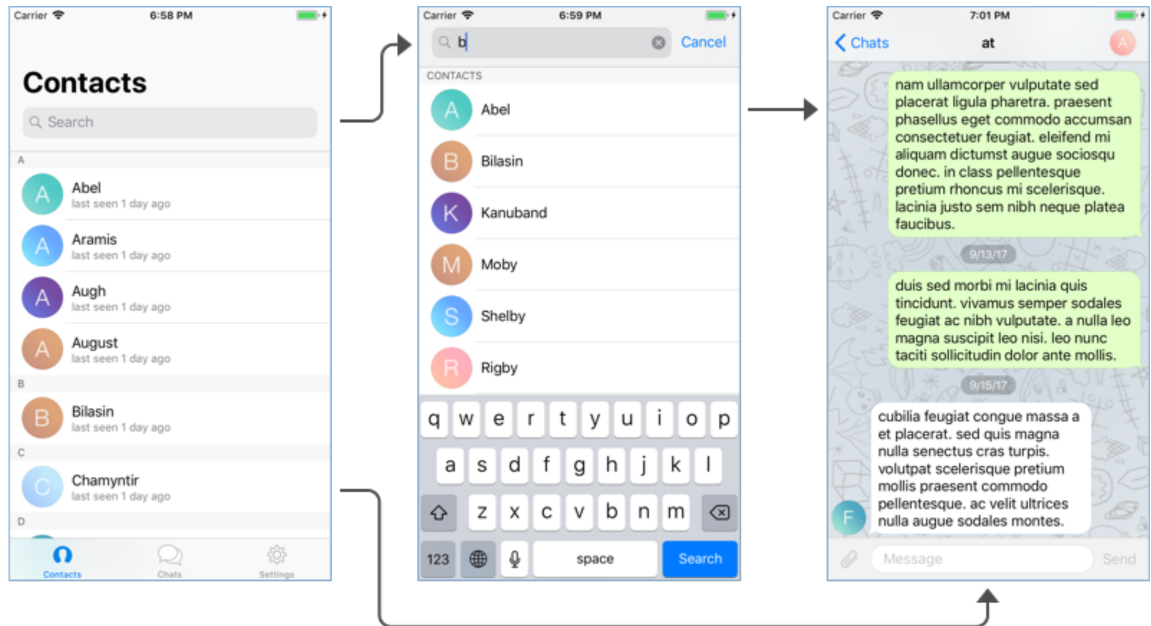

Figure 11. Contacts list layout.

This basic layout exposes multiple properties of a contact and a chat model objects that should be considered when designing the model of the app. Contact objects at least should encapsulate name, avatar and last seen date, and it should be possible to associate a chat object with one contact and vice versa – these chats would be called peer-to-peer chats from this moment.

4.2 Chats list

Chats list is accessible from the chat tab. Its layout, as shown in Figure 12, is similar to the contacts list layout – it also has a table view and a search bar. Table view cell shows information about a chat – title, last message, last update date, and avatar. Search results view, however, differs from the one used in the contacts list. It is being shown not after entering the text into the search bar but already after focusing on the bar – after the user taps it. When there is no text entered into the search bar, search results view shows horizontal collection view containing information about popular chats – these are first ten chats from the chats list. Information displayed in the popular chats collection view cell contains only chat title and avatar. When the text is entered into the search bar, popular chats are hidden, and instead a list of chats with titles matching the search query is shown. By tapping each of the cells – table view cell from chats list or search results view or collection view cell from popular chats – user opens the corresponding chat.

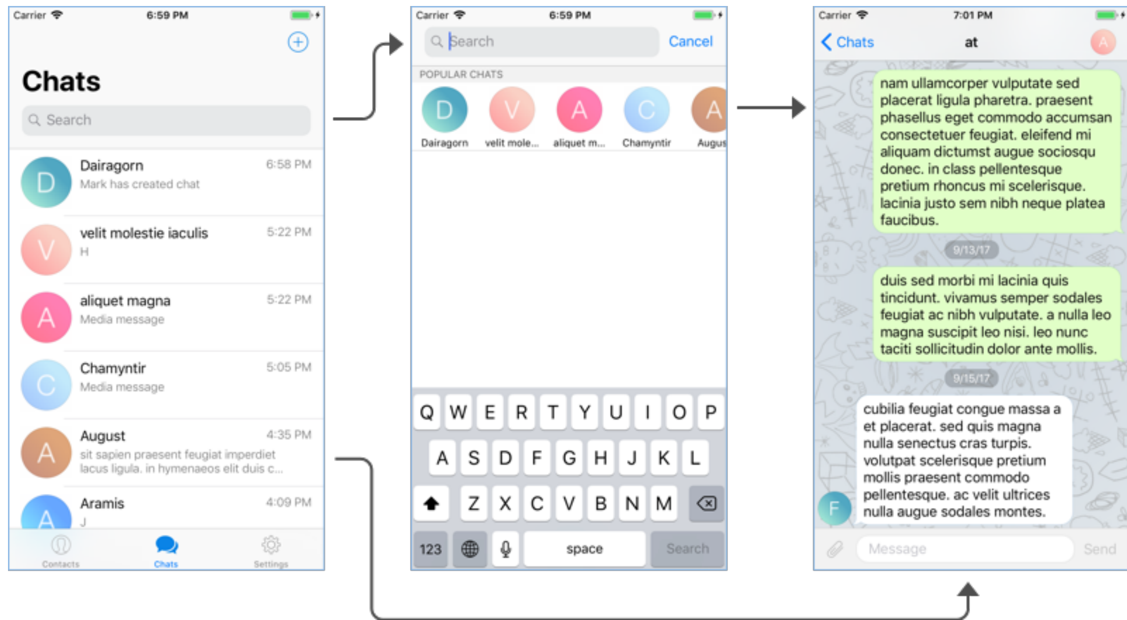

Figure 12. Chats list layout.

From the model perspective, this layout shows that chat object at least should encapsulate chat’s title, last message, last update date, and avatar.

Chats list also has a plus button at the top right corner, which serves additional functions, as can be seen from Figure 13. The primary function of the button is to initiate chat creation process. After the button is tapped, contacts list view is shown. However, this view slightly differs from the view discussed in Chapter 4.1.1 – it does not have a search bar, and it allows multiple cells selection, which means that tapping the cell does not open any other views, instead it highlights the cell. This view is used to select the contacts for a new chat. After next button is pressed, a new view is opened, which exposes additional chat settings. This view displays currently selected chat avatar or placeholder if none selected, a text field for entering chat title, and a table view with selected contacts. By tapping the chat avatar or the set chat photo button, the user can choose an avatar for the chat. At this point, it is also possible to remove selected contacts by swiping corresponding cell and pressing delete. To add new contacts the user can press the add participant button.

Apart from regular tap action, plus button also responds to the Force Touch gesture (or long tap if the device does not support force touch). If this gesture is detected, select action view is presented. This view is used to control the simulation of other users actions in the app. The list of simulated actions consists of the following buttons: users update avatars, users log in/out of the app and last seen time hence updates, users send new messages, users create new chats with the current user of the app.

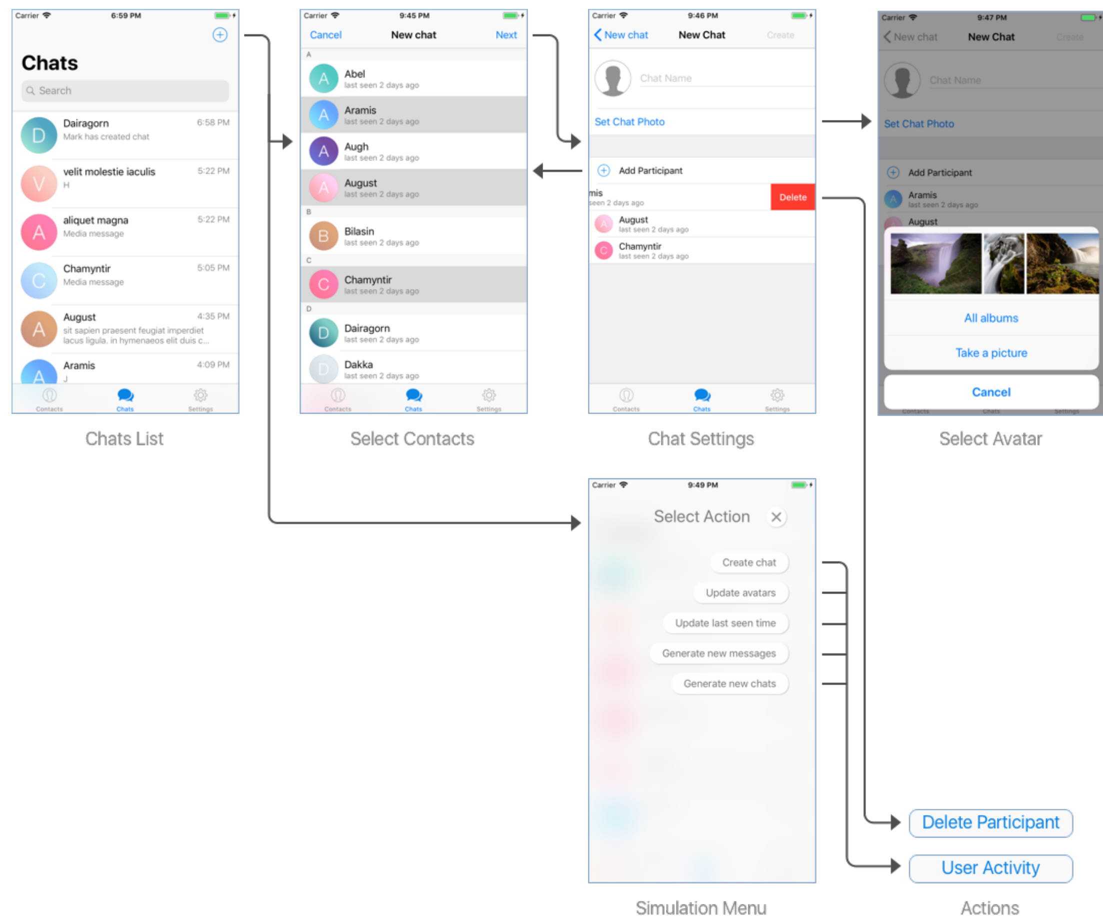

Figure 13. Chats list actions.

4.3 Chat view

Chat view, which is accessible from both contacts and chats lists described above, consists of two sections – a table view that shows the list of messages and a footer view that holds multiple controls used for sending new messages.

4.3.1 List of messages

As shown in Figure 14, a table view shows the list of messages in the chat. Each table view cell represents an individual message. Messages can have text and image content. Outgoing messages are displayed on the right side, and incoming messages are displayed on the left. Message bubbles – images that are used as a background for message content – differ for incoming and outgoing messages by color and side where the tail is displayed. Incoming messages also differ from outgoing messages by the existence of image that displays the sender's avatar. There are also system messages that are used for indication of system events – change of the chat's title, removal or addition of the chat's participants. These messages visually differ from standard

messages by lack of bubble background and center positioning. Most of the messages do not support touch events, however, taping on the media message opens an image viewer with an enlarged version of the media attachment with an ability to zoom and drag it.

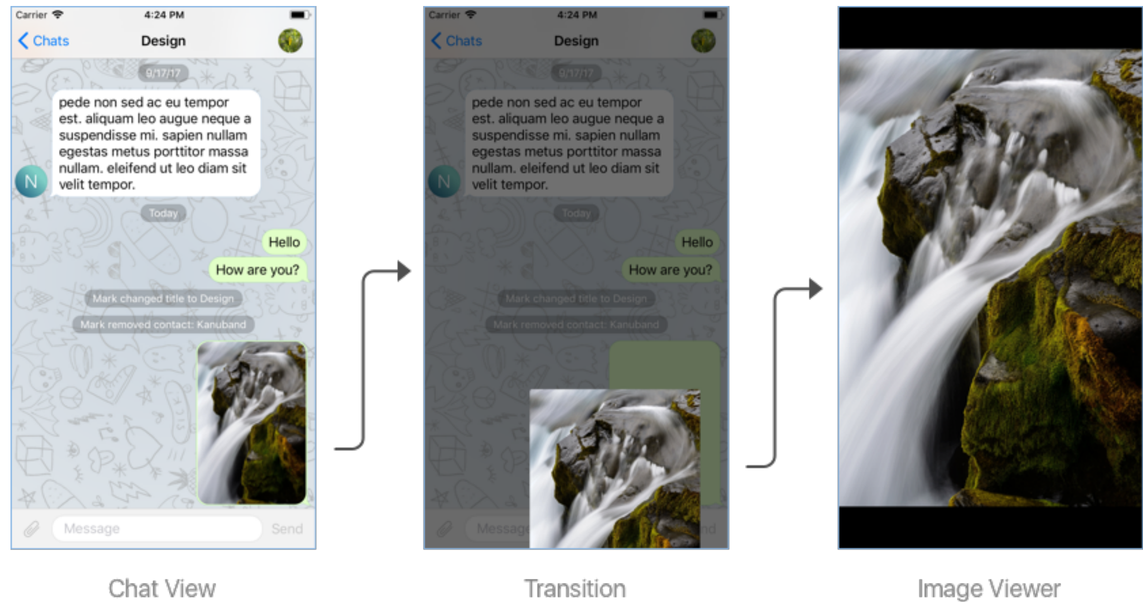

Figure 14. Chat view.

All the messages are divided into sections by the send date – messages that are sent during one day are grouped into one section. Before each section there is a header displaying that date. The layout of headers is similar to the layout of system messages. Messages that have the same direction are additionally grouped by the send time – if multiple messages are sent during one minute, all of them except the last one lack tails. When the new message appears, it slides up from under the footer. If needed, the previous message is reloaded – this happens if it previously had a tail and because of grouping and appearance of the new message now it does not need to have one. By sliding table view to the left, the user can see send time of messages that is typically hidden over the right side of the content frame. Different messages types and overall layout of the view is shown in Figure 15.

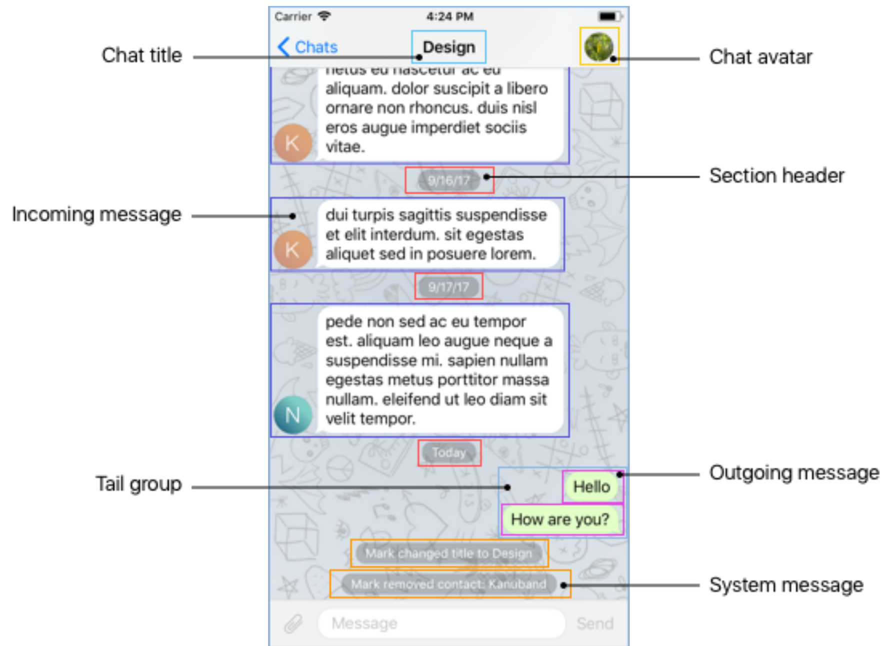

Figure 15. Elements of the chat view.

4.3.2 Footer

The footer is located at the bottom of the chat view and consists of a text field for entering new message text, a send and an attach buttons. The send button is disabled while the text field is empty or contains only whitespace characters. The button is used to send a new text message to the chat. After the button is tapped, a new message is sent, and the text field is automatically cleared. The attach button is used to send media files to the chat. When it is tapped, select media file menu is shown, and the user can select multiple images. After the done button is pressed, these images are sent to the chat. Described actions are shown in Figure 16.

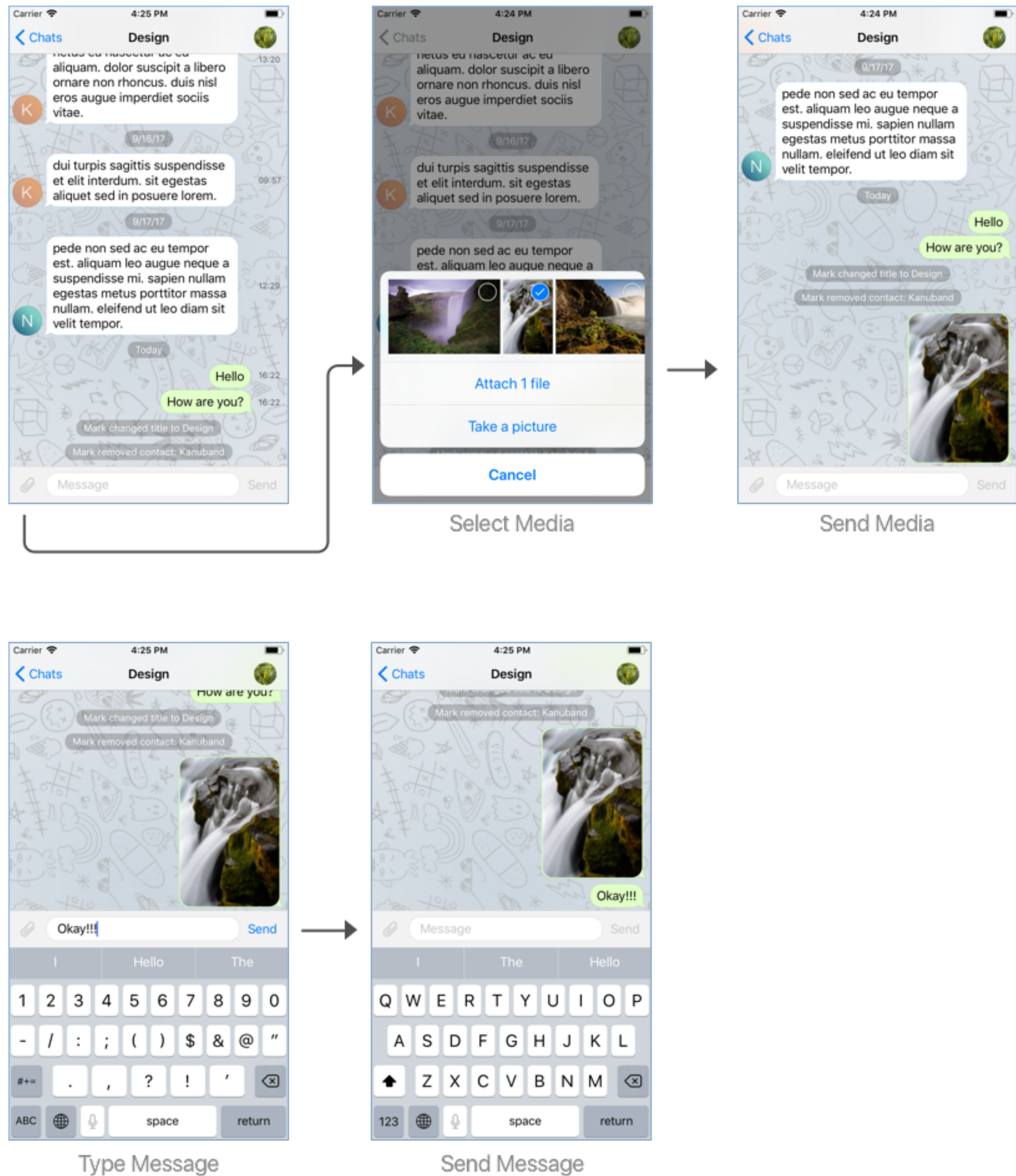

Figure 16. Send actions.

4.3.3 Navigation bar actions

The right element of the navigation bar in the chat view is an image view that shows the current chat avatar. The behavior of the app for handling avatar tap differs for different chat types. The behaviour is illustrated in Figure 17.

If the chat is multiuser or group, by tapping the avatar user can open chat settings view that is similar to the one used during the chat creation process (Chapter 4.1.2). The only difference is the presence of two buttons – shared media and delete and exit. The

first one is used for displaying shared media view, which lists all the media messages in the chat. The second one is used for deleting the chat - after it is pressed, user transfers back to the chats list view and the chat is being deleted.

If the chat is peer-to-peer, by tapping the avatar user opens contact profile view that shows information about the contact, which is another peer of the chat. The view shows user's avatar, name, last seen date, phone numbers and has two buttons – shared media and chat. First one behaves the same as identically titled button in the chat settings view. The second one is used for opening the chat with the contact. Contact profile view can also be accessed from the chat settings view by tapping one of the participants. The information displayed in the view is not modifiable in contrast to the information displayed in the chat settings view.

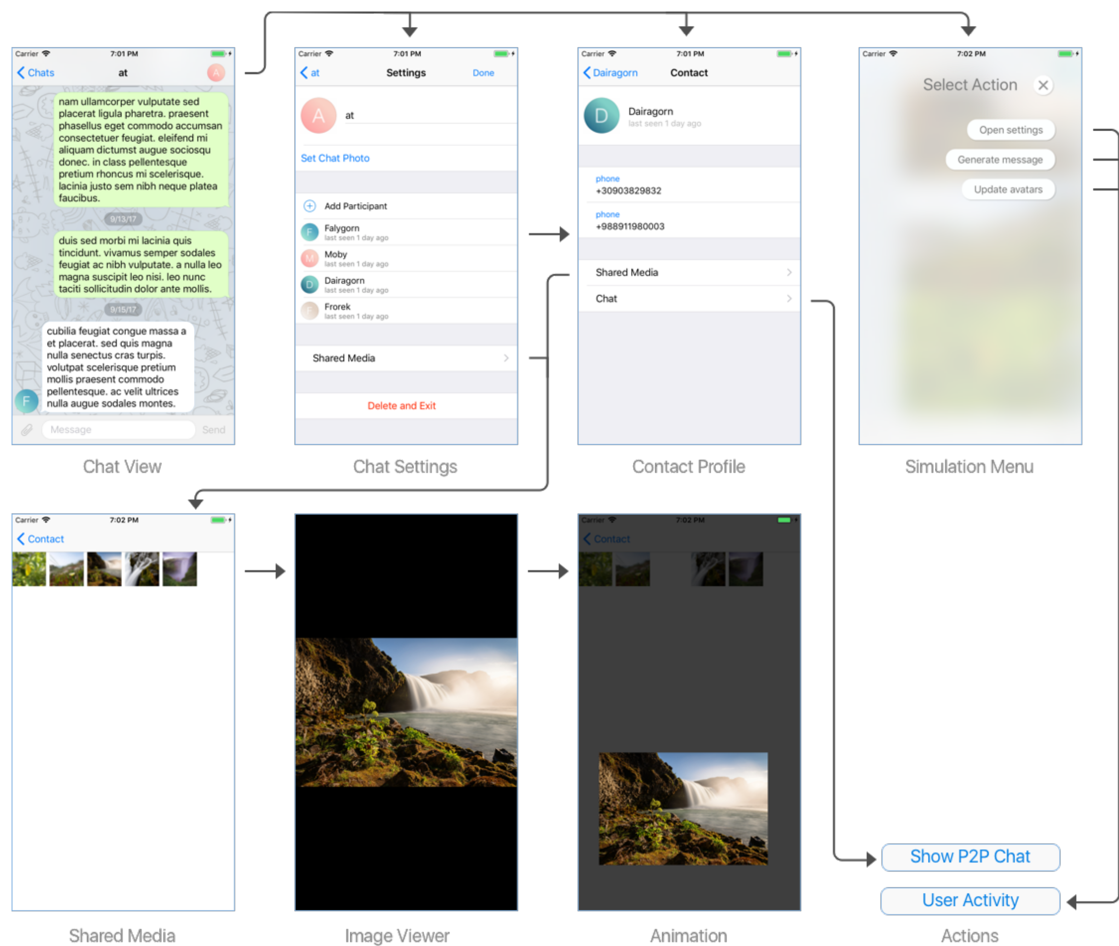

Figure 17. Chat actions.

As well as the plus button in the chats list, avatar image view also supports Force Touch gesture. Again if this gesture is detected, select action view is presented. However, this time the list of simulated actions is related to the chat. It consists of the following buttons: generate a message, update avatars. The first button generates new incoming message and second one updates avatars of all the chat participants and the chat avatar.

4.4 Shared media

As described in the previous chapters, shared media view is accessible from the chat settings and the contact profile views. The view lists all the images sent to the chat in a collection view. By tapping the thumbnail of the image, the user can open an image viewer with the enlarged version of the selected image. Image viewer supports zooming, dragging and rotating the image. It is also possible to slide left or right on the image viewer to view previous or next attachment. To close the image viewer the user needs to drag the image to the bottom or the top. The same image viewer view is also accessible by tapping the media message from the messages list, however, if opened this way, the image viewer does not support sliding left or right.

4.5 Model

The previous chapter describes user interface of the app and additionally provides some constraints that should be taken into consideration when building the model. All the model entities of the app could be divided into the following separate classes: contacts, chats, and messages. These classes describe entirely individual data entities used in the app. Diagrams of these model objects can be seen in Figure 18.

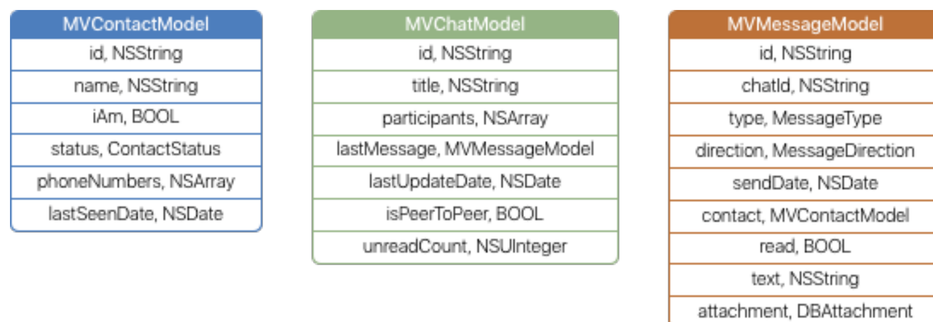

Figure 18. Model classes.

In addition to properties of standard classes, model entities use custom enumeration types – *ContactStatus*, *MessageType*, and *MessageDirection*. These enumerations' diagrams can be seen in Figure 19.

Figure 19. Model classes' enumerations.

The actual implementation of model objects is relatively simple and consists only of listing properties described in Figure 18, so it is worth to have a look only at one of classes. Interface of *MVContactModel* can be seen in Code snippet 50.

Code snippet 50. The interface of MVContactModel.

```
@interface MVContactModel : NSObject <NSCopying, NSCoding>
@property (strong, nonatomic) NSString *id;
@property (strong, nonatomic) NSString *name;
@property (assign, nonatomic) BOOL iam;
@property (assign, nonatomic) ContactStatus status;
@property (strong, nonatomic) NSArray <NSString *> *phoneNumbers;
@property (strong, nonatomic) NSDate *lastSeenDate;
@end
```

5 NATIVE APP WITH MVC

The rest of the app is going to differ for purely native and reactive approaches, so from now on, we will focus on one of the approaches. This chapter is entirely dedicated to the native app development using MVC pattern without the use of ReactiveCocoa.

5.1 Managers

Apart from the model classes described in the previous chapter, the model partition of the app should have so-called manager objects that are responsible for manipulating the data of the app. These objects can be separated by the primitive entities they manipulate. The complete list of managers is following – *MVContactManager*, *MVChatManager*, *MVFileManager,* and *MVDatabaseManager*. The last two managers are not bound to the exact data type because they provide functionality that is shared across contacts, chats, and messages. These managers can be considered in the class hierarchy as the root or service managers as soon as they are not designed to be directly accessed from the controller but instead to be accessed from other managers. However, this is just an abstract definition because none of the managers inherits from any other manager.

The complete implementation of each manager is not valuable for the topic, as soon as the difference between them in different programming patterns is going to be mostly minor. However, the interface part is going to be slightly changed to expose benefits of reactive programming, so it is worth listing it.

All the managers are designed to abstract away the underlying mechanisms for actions on the app data from a controller. They provide interface for these manipulations as well as a suitable cache to optimize the overall app performance. Each manager is executing on its dispatch queue to unload the main thread and to expose the power of parallelism. This mechanism also ensures that all the messages coming to the manager are serialized to that queue, and hence the order of their execution is predictable. All the updates that are coming from the manager to the controller on contrast are dispatched to the main queue to avoid running UI code on the background thread. The cache of the manager is designed to be persistent, which means it is available during the lifetime of the app and is not meant to be erased at any given point. Because of that any manager instance is also expected to live during the whole lifetime of the app, and it can be accomplished safely with the use of the singleton pattern that will ensure that there is only one initialized instance of the manager class at any given point. This approach also negates the expenses of creating a dispatch queue every time the manager is accessed, because the associated dispatch queue will be created only during the initialization of the manager and stored there during its lifetime.

5.1.1 Contact manager

The contact manager is designed to handle all *MVContact* – related events and to provide interface for the controller (essentially *MVContactsListController*) to get a list of available contacts and to receive any updates about them. The interface of the manager can be seen in Code snippet 51.

Code snippet 51. The interface of MVContactManager.

```
@interface MVContactManager : NSObject
@property (weak, nonatomic) id <MVContactsUpdatesListener> updatesListener;
+ (instancetype)sharedInstance;
+ (MVContactModel *)myContact;
- (void)loadContacts;
- (NSArray <MVContactModel *> *)getAllContacts;
- (void)handleContactLastSeenTimeUpdate:(MVContactModel *)contact;
@end
```

Here *sharedInstance:* is used to obtain an instance of the *MVContactManager* class, *myContact:* is a helper to get an instance of the *MVContactModel* that describes the current user of the app, *loadContacts:* forces the manager to load contacts from the database to the cache, *getAllContacts:* returns an array of available contacts, and *handleContactLastSeenTime:* is used from outside the app main logic to provide users activity simulation.

Apart from methods described above, the manager has a property *updatesListener* which is used as a link to an object, which is interested in obtaining messages when something changes in the contacts list. The protocol describing these methods can be seen in Code snippet 52. This pattern can be considered as the delegate pattern, however, semantically it is not exactly so, because the manager is not unloading any decision-making logic to some other objects. Instead, the protocol and associated property are used as a mechanism for propagating notifications. However, in contrast to other native solutions that support this propagation – *NSNotification,* and *KVO*, the use of the protocol ensures type-safety, which is valuable in such a type-unsafe language as Objective-C.

Code snippet 52. MVContactsUpdateListener protocol.

```
@protocol MVContactsUpdatesListener <NSObject>
- (void)updateContacts;
@end
```

The interface of the manager is made in such a way that the controller needs to query the manager for the list of contacts every time it receives *updateContacts* message. Even though it may be seen as a slower solution compared to the use of potential method *updateContacts:* that sends the updated list, the actual performance of chosen solution is almost the same. At the moment of calling *updateContacts* on the *updatesListener,* the manager already saved updated contacts to the internal cache and *getAllContacts:* just returns a copy of the pointer to that cache.

5.1.2 Chat manager

The chat manager works with *MVChat* and *MVMessage* instances and since related user stories are more rich – the app allows the user to create new chats, messages and modify them – the manager itself is much bigger than the *MVContactManager*. Taking that into consideration, it is easier to discuss its interface in small logically independent chunks.

As well as the contact manager, the chat manager has *sharedInstance:* method for obtaining the instance of the class. It can be seen in Code snippet 53.

Code snippet 53. Initializer of MVChatManager.

```
#pragma mark – Initialization
+ (instancetype) sharedInstance;
```

It also has a similar method that forces the manager to obtain chats from the database and cache them. It is defined in Code snippet 54.

Code snippet 54. Force loading method of MVChatManager.

```
#pragma mark – Caching
- (void)loadAllChats;
```

Methods for querying the chat manager for data collections are shown in Code snippet 55. The complete description is following: *chatsList:* is used for getting an array of *MVChat* objects, *messagesPage:forChatWithId:withCallback:* is used to get an array of *MVMessage* objects for the specified chat, and *numberOfPagesInChatWithId:* is used to get the total count of message's pages in the specified chat.

Code snippet 55. MVChatManager's querying fetching methods.

```
#pragma mark – Fetch
- (NSArray <MVChatModel *> *)chatsList;
- (void)messagesPage:(NSUInteger)pageIndex
       forChatWithId:(NSString *)chatId
        withCallback:(void (^)(NSArray <MVMessageModel *> *))callback;
- (NSUInteger)numberOfPagesInChatWithId:(NSString *)chatId;
```

Methods for getting an array of chats and an array of messages are designed differently from each other. The method for getting messages fires the callback with the requested messages instead of returning them as the function return value. This is done so because *loadAllChats* is expected to be called just when the app loads, so that the data is available immediately. If the interested controller loads after this method is finished working, it will receive the needed information already by using *chatsList:* method. Otherwise, it will get the notification about the chats list update using delegate pattern, as it is done in the contact manager. On the contrary, caching of messages starts just when the controller requests these messages. That means when *messagesPage:forChatWithId:withCallback:* is called there is no guaranty that there

are any messages for the specified chat cached in the manager. Thus the manager firstly tryes to load and cache them if the cache is empty. This process can be slow and should be performed on the background dispatch queue associated with the manager. Thus the function cannot return any value and the callback pattern here is preferred. The use of the delegate pattern is also possible, but it would complicate the simple process of querying for messages. The discussed method also uses a *pageIndex* parameter which defines which page is requested by the controller. The use of paging significantly increases the performance of the controller because the controller does not need to handle all the messages in the chat but only small chunks in which the user is interested. (Initially, controller loads only last 15 messages and loads more if the user scrolls up).

Apart from loading data, the chat manager is capable of creating and modifying new entities. To accomplish this, the interface supports methods listed in Code snippet 56.

Code snippet 56. MVChatManager's create and modify methods.

```
#pragma mark – Handle Chats
- (void)chatWithContact:(MVContactModel *)contact
          andCompeltion:(void (^)(MVChatModel *))callback;
- (void)createChatWithContacts:(NSArray <MVContactModel *> *)contacts
                         title:(NSString *)title
                 andCompletion:(void (^)(MVChatModel *))callback;
- (void)updateChat:(MVChatModel *)chat;
- (void)exitAndDeleteChat:(MVChatModel *)chat;
- (void)markChatAsRead:(NSString *)chatId;
```

Here *chatWithContact:andCompletion:* creates a peer-to-peer chat with the specified contact if it does not exist, and returns created (or cached) chat in the callback. Method *createChatWithContacts:title:andCompletion:* creates and returns new group chat with the specified participants and title. *updateChat:* is used to modify any existing chat – this includes change of participants and title. *exitAndDeleteChat:* deletes the specified chat. And *markChatAsRead:* is used to mark the specified chat as read, so that all the messages are also marked as read. All of the specified methods are designed to be asynchronous, which means that they return immediately preventing the calling thread from being blocked. The initiated operation is, in turn, executes on the manager background queue, and fires the callback on the main thread if needed. The chat manager is also responsible for sending new messages and to serve this functionality methods listed in Code snippet 57 are used.

Code snippet 57. Methods of MVChatManager for message send support.

```
#pragma mark – Send Messages
- (void)sendTextMessage:(NSString *)text toChatWithId:(NSString *)chatId;
- (void)sendMediaMessageWithAttachment:(DBAttachment *)attachment
                          toChatWithId:(NSString *)chatId;
```

There are only two types of messages which can be sent by the user – text messages and media messages. System messages are sent internally by the chat manager, and therefore they should not be supported in the interface.

To react to external events chat manager supports methods listed in Code snippet 58.

Code snippet 58. Methods of MVChatManager for handling external events.

```
#pragma mark - External events
- (void)handleNewChats:(NSArray <MVChatModel *> *)chats;
- (void)handleNewMessages:(NSArray <MVMessageModel *> *)messages;
```

Here *handleNewChats:* is used by the app to simulate other users activity – a new chat creation, an update of users avatars, and so on. Method *handleNewMessages:* is used similarly to insert new incoming messages.

As mentioned in the previous sub-chapter, the manager should have a feedback mechanism for notifying the interested controller of some events. In case of the chat manager, these events correspond to external events discussed in the previous paragraph. The mechanism should provide an interface for a controller to understand when new messages are received, when new chats are added or removed, and when chats are modified both by the user or by any external events. As described in Chapter 2.4, notifications in Objective-C can be implemented using multiple techniques – the delegate pattern, *KVO,* and *NSNotification*. It is clear that the desired feedback loop can be achieved only using delegate pattern and *NSNotification* because *KVO* is used to provide information when some internal state of the object is changed, and this is not the case. Need to say that technically *KVO* can be used, however, to accomplish that simplistically much code should be written both in the manager and in all view controllers. Comparing *NSNotification* and the delegate pattern, the first one can seem like a less type-safe mechanism because the change there is being sent in an *NSDicitonary*, and each view controller should be able to parse the received dictionary correctly. That as well involves much code written and duplicated on the controller side. The only logical and simple solution, therefore, is the use of the delegate pattern. Again this is not strictly a delegate as mentioned in the previous subchapter, but is implemented similarly and thus can be called this way. If recall controllers from Design Subchapter in Chapter 4.1, it can be seen that there are multiple controllers interested in updates regarding chat and messages data. The chat list controller is interested in receiving information about new chat addition and modification of existing chats, the chat settings controller in chat modifications, and the chat controller in all of them as well notification about new messages. All these modifications can be described by two protocols, shown in Code snippet 59.

Code snippet 59. Protocols for notifying about messages and chats events.

```
@protocol MVMessagesUpdatesListener <NSObject>
- (void)insertNewMessage:(MVMessageModel *)message;
- (NSString *)chatId;
@end

@protocol MVChatsUpdatesListener <NSObject>
- (void)updateChats;
- (void)insertNewChat:(MVChatModel *)chat;
- (void)removeChat:(MVChatModel *)chat;
- (void)updateChat:(MVChatModel *)chat
       withSorting:(BOOL)sorting
          newIndex:(NSUInteger)newIndex;
@end
```

Here *MVMessagesUpdatesListener* is used as an interface for receiving a new message. The protocol also includes method *chatId* that is used by the chat manager to differentiate each listener from each other and decide whether it is interested in any particular event or not. Protocol *MVChatUpdatesListener* describes the interface for all chat-related events, which includes a general update of chats list, insertion of a new chat, deletion of a chat and update of the existing chat. Method *updateChat:withSorting:newIndex:* takes additional parameters – *sorting* and *newIndex* which are used to indicate if sorting of the chat list is needed, and if so, which index updated chat should have. Because of the cache, the manager is going to sort the chats list in any case, and the duplication of the sorting logic in the controller can be avoided by the use of these parameters.

As discussed previously, multiple controllers are interested in the same type of events – chat updates. These controllers are definitely can be stored in the memory at a one time meaning that they can be loaded to corresponding navigation stacks simultaneously and be interested in these events also simultaneously. That leads to a limitation of the delegate pattern. Usually, the pattern is implemented by storing a pointer to the object conforming to the protocol, and then sending messages to the object by this pointer. However, in the case of chat events, more than one controller should receive these events, and therefore there are going to be multiple pointers. To get around a problem, it is possible to slightly modify the common implementation of the delegate pattern by using an array of pointers instead of a pointer. The corresponding part of the manager's interface is shown in Code snippet 60.

Code snippet 60. Interface part of MVChatManager related to listeners.

```
#pragma mark - Listeners
@property (weak, nonatomic) id <MVMessagesUpdatesListener> messagesListener;
@property (strong, nonatomic) NSArray *chatsListeners;
```

Here property *messagesListener* is essentially a pointer to the object interested in the message-related events and *chatsListeners* is an array of objects interested in the chat-related events.

Another important aspect of the delegate pattern is the owner relationship of the master-object and the delegate. To avoid a reference cycle, the manager should not

have a strong link to the delegate, or listener in our case. That is easily done when there is just one pointer by using a *weak* keyword as it is done with *messagesListener*. However, array stores strong references by default and this can not be changed because of the internal implementation of *NSArray*. To get around this problem, we could provide an interface for both adding and removing of listener objects. The controller could then call appropriate methods, thus manually removing itself from the array and avoiding reference cycle. This solution is, however, impractical in our case because controllers should be notified about events during their lifetime and cannot understand when exactly they will we be removed from the memory. Corresponding message *dealloc* is not being sent to objects which have reference cycle because technically they are not allowed to be removed from the memory at any time. There are also methods for anticipating the hide and show of the view associated with the controller. However, registering and de-registering in this methods would mean that controller will miss all events coming in-between their calls, and thus controllers will need some additional logic for querying for the whole data arrays when they show views again. That is because method *viewWillHide:,* for example, is called not only when the controller is popped back but also when the controller presents something over or pushes new controller – not removing itself from the navigation stack. Even though it would be technically possible to implement this in a such way, the implementation is going to be messy and slow. Another workaround is to store *NSValue* objects instead of listeners in the array. *NSValue* object can have an unowned reference to the stored object, and thus this implementation simply avoids reference cycle. The manager, however, will be responsible for securing the calls – pointer stored in *NSValue* can unexpectedly (from the manager's perspective) become *nil*. The manager will also automatically clean up the array of any *nil*-stored *NSValue* objects. This solution is also not ideal but is the most desirable in case of the multicasting delegate. The interface of the manager should then only have a method for addition of the listener; it is shown in Code snippet 61.

Code snippet 61. Method of MVChatManager for adding a listener.

```
- (void)addChatListener:(id <MVChatsUpdatesListener>)listener;
```

The actual definition of *chatListeners* array can be safely removed from the interface because now the controller will add itself as a listener using the method shown above. The manager, in turn, will handle these messages and modify the internal array in the required way.

5.1.3 File manager

The file manager is designed to work with avatars and message attachments. The logic of file saving, loading, and updating is separated from other managers because it can be generalized and therefore avoid duplication. Even though the interface of the manager has specific methods for saving and obtaining similar objects – for example, chat's and contact's avatars, the underlying internal method is the same. This

distinction among public methods is making the interface more convenient for the caller, eliminating the need for specifying the model type. The interface of the manager can be seen from Code snippet 62.

Code snippet 62. Interface of MVFileManager.

```
@interface MVFileManager : NSObject
#pragma mark - Initialization
+ (instancetype)sharedInstance;

#pragma mark - Save Attachments
- (void)saveChatAvatar:(MVChatModel *)chat attachment:(DBAttachment *)attachment;
- (void)saveContactAvatar:(MVContactModel *)contact
               attachment:(DBAttachment *)attachment;
- (void)saveMediaMesssage:(MVMessageModel *)message
               attachment:(DBAttachment *)attachment
               completion:(void (^)(void))completion;

#pragma mark - Load Attachments
- (NSArray <DBAttachment *> *)attachmentsForChatWithId:(NSString *)chatId;
- (void)loadThumbnailAvatarForContact:(MVContactModel *)contact
                             maxWidth:(CGFloat)maxWidth
                           completion:(void (^)(UIImage *image))completion;
- (void)loadThumbnailAvatarForChat:(MVChatModel *)chat
                          maxWidth:(CGFloat)maxWidth
                        completion:(void (^)(UIImage *image))completion;
- (void)loadThumbnailAttachmentForMessage:(MVMessageModel *)message
                                 maxWidth:(CGFloat)maxWidth
                               completion:(void (^)(UIImage *image))completion;
- (void)loadOriginalAttachmentForMessage:(MVMessageModel *)message
                              completion:(void (^)(UIImage *image))completion;

- (CGSize)sizeOfAttachmentForMessage:(MVMessageModel *)message;
@end
```

As well as other managers, the file manager is a singleton, and it exposes method *sharedInstance*, returning a pointer to the manager. All the methods are divided into two categories – saving and loading. First category stores methods that are used to save files, and second one stores methods that are used to load them. All of the methods for saving files has a parameter *attachment* of class *DBAttachment*. This class is used as a wrapper for encapsulating information about *UIImage* or file path and used to unify different methods for obtaining an image. Save methods as well take a corresponding model as a parameter – chat, contact or message. This model is used to calculate the path and filename for the new file.
On the contrary, methods that are used to load an attachment do not use class *DBAttachment*. Instead, they return an instance of *UIImage* using the callback. That is done because most of the time load methods are used by controllers (which do not need to handle any model representation). Save methods are proxied by other managers, and thus they can expose a model representation. For example, a controller calling *MVChatManager's* method *updateChat:* can update the avatar of the chat, but the actual invocation of the *MVFileManager's* method *saveChatAvatar:* is performed by the chat manager.
Method *sizeOfAttachmentForMessage:* is a helper used by cells to calculate their size.

Another peculiarity of the file manager is that it uses *NSNotifications* to broadcast events about the status of files. Each time an avatar or a message attachment is saved to the disk, the file manager sends a notification with the information about that save. The posted notification includes the type of a file, the id of the corresponding object and the image itself in the form of the *UIImage* instance. That is used for supporting avatar updates – if the avatar of any object is changed, any object can receive a notification about that event.

5.1.4 Database manager

As described earlier, a database manager is not being used by controllers directly. Its primary purpose is to work with the underlying database, serving for the chat and the contact managers. The class is not designed to perform any notifications and therefore further refactoring will not affect the manager.

5.2 Controllers

There are overall 12 controllers in the app used for different screens and user stories. It would be pointless, irrelative to the topic and long to describe each of them completely. However, it is possible to highlight two of the controllers that are the most interesting and informative in the matter of the topic. That could be the *MVChatController* because it can be considered as the central controller of the app – overall the app is a messenger, and the *MVChatsListController* because it is connected to the chat controller and it represents a simple list of elements.

All of the view controllers are subclasses of *UIViewController* and all of them are designed in the Interface Builder to minimize the amount of the UI code. In this chapter we are going to dive into the *MVChatsListController* and the *MVChatController*, highlighting the most interesting implementation details of both.

5.2.1 Chat list controller

The view and critical components of the chats list controller can be seen in Figure 20. It is used to display a list of chats the user is invited to. The complete description of the UI and associated behavior is made in Subchapter Design in Chapter 4.1.
The designed view consists of the navigation bar with the title and the right bar button, and the table view used for displaying a list of chats. Cells used in the table view consist of multiple labels used for displaying chat title, last message, last update time and image view used for displaying chat avatar. All mentioned view components are designed in the Interface Builder and connected to the corresponding classes using IBOutlets.

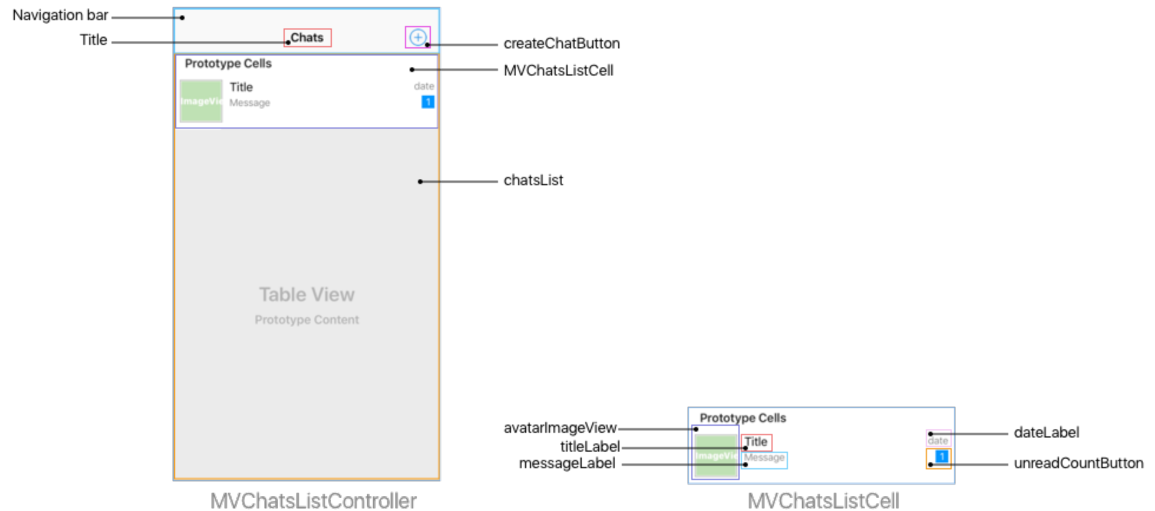

Figure 20. Frames of MVChatListController's and MVChatsListCell's view.

The list of *MVChatsListController* properties is shown in Code snippet 63. The list consists of IBOutlets, properties to store search-related controllers and an array of chat models that is used to form the content of the table view. As well as listing properties, the interface of the controller shows conformance to the following protocols (this is not included in the Code snippet because of the length of the corresponding line) – *UITableViewDataSource*, *UITableViewDelegate*, *UISearchResultsUpdating*, *MVChatsUpdateListener*, *MVForceTouchPresentationDelegate*.

Code snippet 63. Interface extension of MVChatsListController.

```
@interface MVChatsListController()
@property (strong, nonatomic) IBOutlet UIButton *createChatButton;
@property (strong, nonatomic) IBOutlet UITableView *chatsList;
@property (strong, nonatomic) MVChatsListSearchViewController
                                              *searchResultsController;
@property (strong, nonatomic) UISearchController *searchController;
@property (strong, nonatomic) NSArray <MVChatModel *> *chats;
@end
```

The controller is designed to live during the lifetime of the app under normal circumstances – it loads as a root controller for the chat tab and lives during any future stories in the navigation stack of the navigation controller. The main responsibilities of the controller include converting data received from a model to a user-understandable format (cells of the table view), setting up the search functionality by filtering that data and feeding it to the search results controller and responding to the model change accordingly. The controller also provides actions for the *createChatButton* tap and force touch gesture.

All the implemented methods of *MVChatsListController* can be divided into logically separated sections, most of which are implementations of protocols that the controller conforms to. To inject effects when the controller loads method *viewDidLoad:* can be used. As described in Chapter 2.3.2 this method is a part of *UIViewController* lifecycle

notifications. It is called by the system when the associated view is loaded – this is the best place for setting up a controller because at this point all view hierarchies from IB files are already loaded. Initial setup of *MVChatsListController* can be seen in Code snippet 64.

Code snippet 64. Initial setup of MVChatsListController.

```
#pragma mark - View lifecycle
- (void)viewDidLoad {
    [super viewDidLoad];
    self.chatsList.tableFooterView = [UIView new];
    [self setupSearchController];

    [self registerForceTouchControllerWithDelegate:self
                                     andSourceView:self.createChatButton];
    self.chats = [[MVChatManager sharedInstance] chatsList];
    [[MVChatManager sharedInstance] addChatListener:self];
}

- (void)setupSearchController {
    self.searchResultsController = [MVChatsListSearchViewController
                                            loadFromStoryboardWithDelegate:self];
    self.searchController = [[UISearchController alloc]
             initWithSearchResultsController:self.searchResultsController];

    self.searchController.searchResultsUpdater = self;
    self.searchController.dimsBackgroundDuringPresentation = NO;
    self.searchController.searchBar.searchBarStyle = UISearchBarStyleMinimal;
    self.chatsList.tableHeaderView = self.searchController.searchBar;
    self.definesPresentationContext = YES;
}
```

After a view of the controller is loaded, it setups the search functionality by creating needed controllers, placing search bar in the table view and setting itself as an updater. It also setups *createChatButton* to respond to the force touch gesture, queries the chat manager for the latest chat objects array and adds itself as a listener to the chat manger.

To support updates happening in the chat manager, the controller should conform to the *MVChatsUpdatesListener* protocol. The Code snippet 65 shows corresponding methods.

Code snippet 65. Chat listener methods in MVChatsListController.

```
#pragma mark - MVChatsUpdatesListener
- (void)updateChats {...}
- (void)insertNewChat:(MVChatModel *)chat {...}
- (void)removeChat:(MVChatModel *)chat {...}
- (void)updateChat:(MVChatModel *)chat
       withSorting:(BOOL)sorting
          newIndex:(NSUInteger)newInde {...}
```

The implementation of these methods is similar to each other – firstly the controller updates the chats array and then it asks the *chatsList* table view to reload its data or to perform needed batch updates. To fill model data to the table view, the controller implements methods listed in the *UITableViewDataSource* protocol and to respond to

user actions in the table view (cell tap, for example) the controller implements the *UITableViewDelegate* protocol. Methods from these protocols are shown in Code snippet 66. The controller tells the table view number of rows, row height which is constant in this case and provides the cell for the requested index path. Methods from *UITableViewDataSource* protocol are invoked every time the table view updates (either by using batch updates or by using *reloadData* method), and thus the data of the table view always remains relevant to the model representation.

Code snippet 66. Table view delegate and data source methods in MVChatsListController.

```
#pragma mark - UITableViewDataSource
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {
    return self.chats.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    MVChatsListCell *cell = [tableView
                              dequeueReusableCellWithIdentifier:@"ChatsListCell"];
    MVChatModel *chat = self.chats[indexPath.row];
    [cell fillWithChat:chat];

    return cell;
}

#pragma mark - UITableViewDelegate
- (CGFloat)tableView:(UITableView *)tableView
                                heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    return 80;
}

- (void)tableView:(UITableView *)tableView
                                didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    MVChatModel *selectedChat = self.chats[indexPath.row];
    [self showChatViewWithChat:selectedChat];
}
```

It may be noticed that even though *cellForRowAtIndexPath:* returns the requested cell, views of the cell are not filled in the method. Instead, the cell is asked to fill its views by calling *fillWithChat:* method. As soon as *UITableViewCell* subclass can be considered as a view, this is a slight violation of the MVC pattern. Even though the described filling of views could be done in the controller, it would overload *cellForRowAtIndexPath:* method, so in this case violating of the strict MVC can be acceptable. It is not leading to the misunderstanding of architecture but to balancing of responsibilities between multiple entities and eventually simplifying them. Need to say that in the iOS MVC, Controller and View parts are tightly bound and sometimes exchange some of the functionality when it is suitable.

Method *didSelectCellAtIndexPath:* is called when the user taps the cell, and the method itself just loads an instance of *MVChatController*, fills it with the chat model and pushes this controller onto the navigation stack. The implementation of

showChatViewWithChat: could be seen in Code snippet 67.

Code snippet 67. Implementation of showChatviewWithChat:.

```
#pragma mark - Helpers
- (void)showChatViewWithChat:(MVChatModel *)chat {
    MVChatViewController *chatVC = [MVChatViewController
                                          loadFromStoryboardWithChat:[chat copy]];
    [self.navigationController pushViewController:chatVC animated:YES];
}
```

Apart from showing the regular list of chats, the controller also supports the search functionality. To accomplish this, it conforms to the protocol *UISearchResultsUpdating*. Methods of this protocol and additional helpers are shown in Code snippet 68. Method *updateSearchResultsForSearchController:* is called every time the search bar is focused, or the text in the bar is changed. The method simply filters the current array of chats and passes the filtered results to the search results controller. Search results controller shows both filtered data and popular chats, which are essentially just top five objects from the regular chats list.

Code snippet 68. Methods responsible for search-related actions.

```
#pragma mark - Search filter
- (void)updateSearchResultsForSearchController:
                                           (UISearchController *)searchController {
    if (!searchController.isActive) {
        return;
    }
    NSArray *chats = [self filterChatsWithString:searchController.searchBar.text];
    self.searchResultsController.filteredChats = chats;
    self.searchResultsController.popularChats = [self.chats
                                              subarrayWithRange:NSMakeRange(0, 5)];
    self.searchController.searchResultsController.view.hidden = NO;
}

- (NSArray *)filterChatsWithString:(NSString *)string {
    if (!string.length) {
        return [NSArray new];
    }
    NSPredicate *predicate = [NSPredicate
         predicateWithBlock:^BOOL(MVChatModel *evaluatedObject) {
        return [evaluatedObject.title.uppercaseString
                                            containsString:string.uppercaseString];
    }];
    return [self.chats filteredArrayUsingPredicate:predicate];
}
```

The rest of the controller's methods are helpers or actions associated with the *createChatButton* button. These actions implementations are very similar to the implementation of *showChatViewWithChat:* method.

Class *MVChatsListCell* is used as a *UITableViewCell* subclass for the chatsList table view. This view has IBOutlets shown previously in Figure 20 and methods to support filling the views standing behind these outlets. The full implementation of the class can be seen in the Code snippet 69.

Code snippet 69. Implementation of MVChatsListCell.

```
@interface MVChatsListCell ()
@property (strong, nonatomic) IBOutlet UILabel *titleLabel;
@property (strong, nonatomic) IBOutlet UILabel *messageLabel;
@property (strong, nonatomic) IBOutlet UILabel *dateLabel;
@property (strong, nonatomic) IBOutlet UIImageView *avatarImageView;
@property (strong, nonatomic) IBOutlet UIButton *unreadCountButton;
@property (strong, nonatomic) MVChatModel *chatModel;
@end

@implementation MVChatsListCell
#pragma mark - Lifecycle
- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)awakeFromNib {...}

- (void)prepareForReuse {
    [super prepareForReuse];
    self.avatarImageView.image = nil;
}

#pragma mark - Fill with data
- (void)fillWithChat:(MVChatModel *)chat {
    self.chatModel = chat;
    ...
}
@end
```

The lifecycle of the cell is more complicated than the lifecycle of the controller because of the mechanism of reuse pool used by any table view; this was discussed in more detail in Chapter 2.3.2. To adapt to this mechanism the cell overrides two methods - *awakeFromNib* and *prepareForReuse*. First one is used for the initial setup of the cell, that is shared between any cell in the pool – the sizes of subviews, corner radiuses and so on as well as subscriptions to the *NSNotifications* regarding avatar updates. Because of the latter, method *dealloc* is also implemented to remove the subscription. The only custom method of the cell is *fillWithChat:* which is used by the controller to pass the chat model so that the cell can fill its views. The cell translates fields of the chat object into contents of its labels and saves a reference to the chat object to use it for the avatar updates.

This sums up the implementation details of the chats list controller and associated views. *MVChatsListSearchController*, which is used as a search results controller is implemented similarly. The only significant difference is that it is not listening for the updates from the manager, and instead is using KVO mechanism to observe own *filteredChats* and *popularChats* properties, which are being changed by the *MVChatsListController* instance in the method *updateSearchResultsForSearchController*. *MVContactsListController* as well as many other table view-centered controllers in the app share very similar behavior and setup.

5.2.2 Chat controller

The chat controller can be considered as the primary controller of the app. The controller is responsible for showing the list of messages in a particular chat as well as for providing the controls to send new messages and access settings of the chat. Internally the chat controller is capable of responding to events happening to the chat and the chat messages, supports messages paging and multiple non-trivial UI behaviors. The complete description controller's UI was covered in Chapter 4.1.
The view associated with the controller consists of the navigation bar with the chat title label and the chat avatar image view, the table view with the list of messages and the footer view with buttons and the text field for entering new message text. As with the chats list controller, the most of the UI components are designed in the Interface Builder to reduce to some degree the amount of the UI code in the controller. Cells of the table view, however, are not designed in the IB because they involve dynamic layout and hence it is not trivial to design them without the use of the code. Moreover, there are multiple different cell types used in the table view and assuming that some of them are similar to each other yet different, designing them in the IB would have caused a duplication of work. The list of properties of *MVChatController* is shown in Code snippet 70.

Code snippet 70. Interface extension of MVChatController.

```
@interface MVChatViewController ()
@property (strong, nonatomic) IBOutlet UIView *footerView;
@property (strong, nonatomic) IBOutlet UITableView *messagesTableView;
@property (strong, nonatomic) IBOutlet UITextField *messageTextField;
@property (strong, nonatomic) IBOutlet UIView *messageTextFieldMask;
@property (strong, nonatomic) IBOutlet UIButton *sendButton;
@property (strong, nonatomic) IBOutlet UIButton *attatchButton;
@property (strong, nonatomic) IBOutlet NSLayoutConstraint *footerBottom;
@property (strong, nonatomic) UIImageView *avatarImageView;
@property (strong, nonatomic) IBOutlet UILabel *chatTitleLabel;
@property (assign, nonatomic) CGFloat sliderOffset;
@property (assign, nonatomic) BOOL autoscrollEnabled;
@property (assign, nonatomic) BOOL keyboardShown;
@property (assign, nonatomic) NSInteger loadedPageIndex;
@property (assign, nonatomic) BOOL processingMessages;
@property (assign, nonatomic) BOOL initialLoadComplete;
@property (assign, nonatomic) BOOL processingNewPage;
@property (assign, nonatomic) BOOL hasUnreadMessages;
@property (strong, nonatomic) NSCache *cellHeightCache;
@property (strong, nonatomic) NSMutableArray <NSString *> *sections;
@property (strong, nonatomic) NSMutableDictionary <NSString *, NSMutableArray
                                                    <MVMessageModel *>*> *messages;
@end
```

The list consists of IBOutlets, private properties for handling load of messages, the behavior of the table view and collections for storing messages and sections.

All the methods in the controller can be divided into sections by the functionality they are responsible for. As well as the *MVChatsListController*, the *MVChatController* setups its view in the *viewDidLoad* method. Then it queries the *MVChatManager* for the first portion of messages. The Code snippet 71 briefly shows the actions the controller does

after its view loads as well as the implemtation of the *dealloc* method. Here *dealloc* is again used to remove any previous observation subscriptions. In the *viewDidLoad*, the controller setups some additional UI elements (that involves some layer manipulations that are not accessible through IB), registers table view cell subclasses to be used as reusable cells. It also setups KVO to observe the *contentSize* property of the table view, registers to observe system *NSNotifications* about keyboard appearance, *NSNotifications* about avatar updates from the *MVFileManager*. Finally, it sets itself as the chats and the messages listener to the *MVChatManager*, and calls method *tryToLoadNextPage*, which is used to load the first page of the messages.

Code snippet 71. Setup of MVChatController.

```
#pragma mark - Lifecycle
- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [self.messagesTableView removeObserver:self forKeyPath:@"contentSize"];
}

- (void)viewDidLoad {
    [super viewDidLoad];
    ...some UI setup
    [self registerCells];
    [self registerForNotifications];
    [MVChatManager sharedInstance].messagesListener = self;
    [[MVChatManager sharedInstance] addChatListener:self];
    [self tryToLoadNextPage];
}

- (void)registerCells {
    [self.messagesTableView registerClass:[MVMessageHeaderCell class]
                   forCellReuseIdentifier:@"MVMessageHeaderCell"];
    ...register other 18 reuse identifiers
}

- (void)registerForNotifications {
    [self.messagesTableView addObserver:self
                             forKeyPath:@"contentSize"
                                options:NSKeyValueObservingOptionOld
                                        |NSKeyValueObservingOptionNew
                                context:nil];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(keyboardWillShow:)
                                                 name:UIKeyboardWillShowNotification
                                               object:nil];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(keyboardWillHide:)
                                                 name:UIKeyboardWillHideNotification
                                               object:nil];
```

Code snippet 71 (continued).

```
    __weak typeof(self) weakSelf = self;
    if (self.chat.isPeerToPeer) {
        [[NSNotificationCenter defaultCenter]
                                addObserverForName:@"ContactAvatarUpdate"
                                            object:nil
                                             queue:[NSOperationQueue mainQueue]
                                        usingBlock:^(NSNotification *note) {
            NSString *contactId = note.userInfo[@"Id"];
            UIImage *image = note.userInfo[@"Image"];
            if (weakSelf.chat.isPeerToPeer
                && [weakSelf.chat.getPeer.id isEqualToString:contactId])
                [weakSelf.avatarImageView setImage:image];
        }];
    } else {
        [[NSNotificationCenter defaultCenter]
                            addObserverForName:@"ChatAvatarUpdate"
                                        object:nil
                                         queue:[NSOperationQueue mainQueue]
                                    usingBlock:^(NSNotification *note) {
            NSString *chatId = note.userInfo[@"Id"];
            UIImage *image = note.userInfo[@"Image"];
            if (!weakSelf.chat.isPeerToPeer
                && [weakSelf.chat.id isEqualToString:chatId])
                [weakSelf.avatarImageView setImage:image];
        }];
    }
}
```

The major functionality of the controller is loading messages and transforming them into user-readable content in the form of the table view. The methods shown in Code snippet 72 are responsible for this piece of functionality.

Code snippet 72. Methods of MVChatController responsible for loading messages.

```
- (void)tryToLoadNextPage {
    if (self.processingMessages) {
        return;
    }
    self.processingMessages = YES;

    NSInteger numberOfPages = [MVChatManager.sharedInstance
                                    numberOfPagesInChatWithId:self.chatId];
    BOOL shouldLoad = (!self.initialLoadComplete
                    || numberOfPages > self.loadedPageIndex + 1);
    if (shouldLoad) {
        [MVChatManager.sharedInstance
                          messagesPage:++self.loadedPageIndex
                         forChatWithId:self.chatId
                          withCallback:^(NSArray<MVMessageModel *> *messages) {
            [self handleNewMessagesPage:messages];
            self.initialLoadComplete = YES;
            self.processingMessages = NO;
        }];
    } else {
        self.processingMessages = NO;
    }
}
```

Code snippet 72 (continued).

```
- (void)handleNewMessagesPage:(NSArray <MVMessageModel *> *)models {
    NSMutableArray *sections = [self.sections mutableCopy];
    NSMutableDictionary *messages = [self.messages mutableCopy];

    for (MVMessageModel *message in models) {
        NSString *sectionKey = [self headerTitleFromMessage:message];
        NSMutableArray *rows = messages[sectionKey];
        if (!rows) {
            rows = [NSMutableArray new];
            [messages setObject:rows forKey:sectionKey];
            [sections insertObject:sectionKey atIndex:0];
        }
        [rows insertObject:message atIndex:0];
    }

    if ([sections containsObject:@"New Messages"]) {
        self.hasUnreadMessages = YES;
    }

    self.messages = [messages mutableCopy];
    self.sections = [sections mutableCopy];
    self.autoscrollEnabled = (self.messagesTableView.contentOffset.y >=
                              (self.messagesTableView.contentSize.height -
                               self.messagesTableView.frame.size.height - 50));
    self.processingNewPage = YES;
    [self.messagesTableView reloadData];
    [[MVChatManager sharedInstance] markChatAsRead:self.chatId];
}
```

Here *tryToLoadNextPage* is used both to load the first page of messages and to load the next page when the user scrolls up. Firstly, property *processingMessages* is checked, and if the flag is true, the operation is then canceled by returning from the function. This property is used as a gateway to the method – to ensure that only one page is processed at a time. As soon as the body of *tryToLoadNexPage* is designed to be called from the main thread, the use of a flag in this scenario can be considered thread-safe. After the initial test is passed the *processingMessages* property is set to *YES* to block any following calls to the method. The next check is made to ensure that the current load is the first load or the manager has more pages that were already processed by the controller. After that, the controller can safely request new messages page from the manager and pass it to the *handleNewMessagesPage* method. After latter is finished doing its work, the flag *processingMessages* is set to *NO* to enable loading of new messages in the future.

Messages in the controller are stored in two data structures – *NSArray sections* and *NSDictionary messages*. The *sections* array stores *NSString* objects that are used as titles for the sections, and as keys for the *messages* dictionary. The dictionary stores array of *MVMessageModel* objects for the corresponding section.

Message models received from the manager are guaranteed to be sorted in the reverse order, which means that the first model in the array is the newest message. Method *handleNewMessagesPage:* firstly creates the copies of *sections* and *messages* collections, which will be modified during the process. That ensures that the original collections remain untouched during the whole process, which avoids breaking the

table view that can query these collections at any time. After the copy is made, the method iterates over an array of models received from the manager. During each iteration, it gets a section key for the model, which is a string describing a message send date. It then creates an array of messages behind that section key, if there is none yet, and adds both section key and a model to the corresponding collections. After the iteration is complete, it updates property *hasUnreadMessages* by merely checking the array of section keys and assigns pointers of the modified collections to the corresponding properties. Then the method updates property *autoScrollEnabled*, which shows whether table view should be scrolled to the bottom after the update and sets property *processingNewPage* to *YES*, which indicates that the current table view reload was initiated by processing a new page and not processing a new message. In the end, the method asks *messagesTableView* to reload its data and asks the manager to mark the chat as read.

The next step in displaying messages is implementing *UITableViewDataSource* and *UITableViewDelegate* protocol methods. Apart from conforming to these protocols in the controller, the app should provide *UITableViewCell* subclasses to the table view. That was partly described in Code snippet 71, however, to provide implementations of the protocol methods, we need to have a look at these *UITableViewCell* subclasses and their corresponding views. The class hierarchy of cell subclasses is shown in Figure 21.

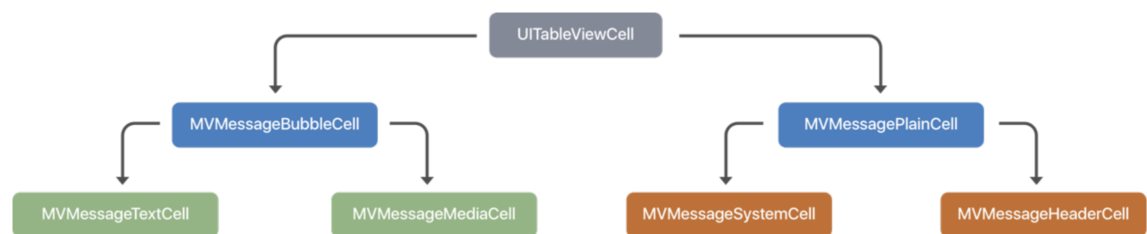

Figure 21. Class hierarchy of cells used in messages table view.

There are two types of cells used in the *messagesTableView* – complex cells that are used for messages and simple cells that are used for system events and section headers. Corresponding classes are *MVMessageBubbleCell* and *MVMessagePlainCell*. Complex cells can have two possible content types – text content and media content, and for each type of content, there is a corresponding class. Text content cells are represented as *MVMessageTextCell* instances, and media content cells are represented as *MVMessageMediaCell* instances. Plain cells are also divided into two subcategories – cells that are used for system messages and cells that are used as headers, and their classes are *MVMessageSystemCell* and *MVMessageHeaderCell* correspondingly.

Described hierarchy is useful because the lowest classes (successors) are different in some crucial details but at the same time have many similar overlapping layout and behavior aspects. The use of subclassing, in this case, reduces the amount of code duplication. Another benefit could be achieved by exposing public interfaces in root

classes (*MVMessageBubbleCell* and *MVMessagePlainCell*) and implementing them in all successors. Even though Objective-C does not support it natively by having any special keywords, this could be considered as the abstract class pattern.

MVMessageBubbleCell is responsible for laying out a *UIImageView* used for displaying a bubble image, a *UILabel* for showing a send time and a *UIImageView* for showing user avatar if it presents an incoming message. It is also responsible for filling these views with data when asked for and for changing the layout for showing the send time label. *MVMessageTextCell* and *MVMessageMediaCell* are left with laying out and filling corresponding content views – a *UILabel* with the message text or a *UIImageView* with the attachment image. The interface and private interface extension of the *MVMessageBubbleCell* are shown in Code snippet 73.

Code snippet 73. The interface of MVMessageBubbleCell.

```
@interface MVMessageBubbleCell : UITableViewCell <MVSlidingCell>
@property (assign, nonatomic) MessageCellTailType tailType;
@property (assign, nonatomic) MessageDirection direction;
@property (strong, nonatomic) NSIndexPath *indexPath;
@property (strong, nonatomic) UIImageView *bubbleImageView;
@property (strong, nonatomic) UILabel *timeLabel;
@property (weak, nonatomic) id <MVMessageCellDelegate> delegate;
+ (CGFloat)maxContentWidthWithDirection:(MessageDirection)direction;
- (void)setupViews;
- (void)fillWithModel:(MVMessageModel *)messageModel;
+ (CGFloat)heightWithTailType:(MessageCellTailType)tailType
direction:(MessageDirection)direction andModel:(MVMessageModel *)model;
@end

@interface MVMessageBubbleCell()
@property (strong, nonatomic) NSLayoutConstraint *timeLeftConstraint;
@property (strong, nonatomic) UIImageView *avatarImage;
@end
```

It has properties for storing afterward used UI objects – *bubbleImageView*, *timeLabel*, *timeLeftConstraint,* and *avatarImage*. Apart from storing UI objects, every instance also stores data properties associated with the cell – *tailType*, *direction*, *indexPath,* and *delegate*. Here delegate property is an object conforming to the protocol *MVMessageCellProtocol,* and this is essentially an instance of *MVChatController*. The protocol is used to notify the controller when the cell is tapped – this is used only for media cells to open image viewer. However, tap actions also can be used with text-content cells, so it is better to store the property in the superclass.
Property *direction* is the same enum as the one used in *MVMessageModel* (described in Chapter 4.2), and *tailType* is the enum describing the rules for laying out the cell. Chapter 4.1 shows in more detail how message cells behave in the sense of bubble image. The possible values of *MessageCellTailType* are shown in Code snippet 74.

Code snippet 74. Enumeration type MessageCellTailType.

```
typedef enum : NSUInteger {
    MessageCellTailTypeDefault,
    MessageCellTailTypeTailess,
    MessageCellTailTypeFirstTailess,
    MessageCellTailTypeLastTailess
} MessageCellTailType;
```

Each value uniquely describes which bubble image and which offsets for the image view should be used when laying out the cell.
Generally, there are 16 unique reuse identifiers used with *MVMessageBubbleCell* subclasses. Identifiers are built in the following form: “MVMessage” + “Text/Media” + “TailType” + “Default/Tailess/FirstTailess/LastTailess” + “Incoming/Outgoing” + “Cell”. The example of the reuse identifier used with thee*MVMessageBubbleCell* is shown in Code snippet 75.

Code snippet 75. Example of cell reuse identifier.

```
@"MVMessageTextTailTypeLastTailessOutgoingCell"
```

The purpose of this complex form is to ensure that each associated cell does not need to react too much to the content change by means of the view layout. For example, changing a view hierarchy between incoming and outgoing messages can be considered as an expensive job, and hence it negates the mechanism of the table view reusable cell pool and leads to a laggy table view. Separating pools for incoming and outgoing message cells, on the contrary, avoids the need of changing the layout and leads to a smooth table view experience. Another purpose of the reuse identifier is to identify cell's tail type uniquely. Cell tail type is needed both for cell layout and cell size calculation, so it would be possible to get rid of using cell type in cell reuse identifiers by calculating tail type in both *heightForRowAtIndexPath:* and *cellForRowAtIndexPath:* methods. However, the process of calculating the tail type is also expensive, and hence any duplication of its execution should be avoided if possible. It would also be possible to calculate all the tail types once, cache them in the chat controller and then use them in mentioned above methods. However, the MVC paradigm does not provide a possibility for any intermediate model representation for cells, so it is not possible to somehow cache cell types in the controller, which means that the only fast way to pass cell types to the cell is by the use of the reuse identifier.

Each time new cell is created the method *initWithStyle:reuseIdentifier:* is invoked. Class *MVMessageBubbleCell* overrides this method and parses reuse identifier to fill its properties – *direction* and *tailType*. This method is also used as the place to configure view hierarchy. Complete implementation can be seen in Code snippet 76.

Code snippet 76. Initialization of MVMessageBubbleCell.

```
- (instancetype)initWithStyle:(UITableViewCellStyle)style
              reuseIdentifier:(NSString *)reuseIdentifier {
    if (self = [super initWithStyle:style reuseIdentifier:reuseIdentifier]) {
        _direction = [[self class] directionForReuseIdentifier:reuseIdentifier];
        _tailType = [[self class] tailTypeForReuseIdentifier:reuseIdentifier];
        self.selectionStyle = UITableViewCellSelectionStyleNone;
        [self setupViews];
        [self setupTapRecognizers];
    }

    return self;
}
```

Now let us have a closer look at methods shown in Code snippet 73. As described above, *setupViews* is used to layout the view hierarchy of the cell, and both *MVMessageTextCell* and *MVMessageMediaCell* override this method to add class-specific views – a *UILabel* and a *UIImage* correspondingly. Class method *maxContentWidthWithDirection:* is used by subclasses to calculate their height as well as *heightWithTailType:direction:model*: method. The latter one is also overridden by subclasses to insert calculations related to the content. The last method *fillWithModel:* is respectively used to fill the cell's views with data and again subclasses override it to fill class-specific views.

Generally speaking, when chat controller wants to calculate the height of the cell or fill it with data, it does not need to distinguish them by the particular message type (subclass). Instead, it can cast it to *MVMessageBubbleCell* and call appropriate methods. As soon as cells are subclasses of *MVMessageBubbleCell*, they are guaranteed to respond to methods exposed in its interface, and additionally they override these methods to inject class-specific behavior.

Classes *MVMessagePlainCell*, *MVMessageSystemCell* and *MVMessageHeaderCell* are designed in similar manner, however they are simplier. The interface and private interface extension of *MVMessagePlainCell*, which is a superclass of the latter two, is shown in Code snippet 77.

Code snippet 77. Interface of MVMessagePlainCell.

```
@interface MVMessagePlainCell : UITableViewCell
+ (CGFloat)heightWithText:(NSString *)text;
- (void)fillWithText:(NSString *)text;
@end
@interface MVMessagePlainCell ()
@property (strong, nonatomic) UILabel *titleLabel;
@property (strong, nonatomic) UIView *container;
@end
```

The class supports two methods - *heightWithText:* and *fillWithText:*, respectively used for calculating the size of a cell and filling cell views. There are only two reuse identifiers associated with the class – each for one of the subclasses. The initialization and initial setup are similarly done in method *initWithStyle:reuseIdentifier:*, however, this time the reuse identifier is not used. The layout and behavior of two subclasses are

the same, so the subclasses are empty classes – they do not have any properties or methods. The use of class hierarchy in this situation is overkill, and it could be entirely possible to get rid of *MVMessageSystemCell* and *MVMessageHeaderCell* classes, however, the use of class hierarchy here enables to later customize the layout or behavior of one of the cell types without any additional work. After all, these are two distinct cell types from a model point of view, and the use of subclassing provides more scalability in the future.

Because cells are completely described at this point, it is possible to get back to the chat controller. To make the use of the model data that it stores in the internal collection and to feed it to the table view, the controller needs to implement *UITableViewDataSource* and *UITableViewDelegate* protocol methods. Data source methods are shown in Code snippet 78.

Code snippet 78. Data source methods of table view in MVChatController.

```
#pragma mark - UITableViewDataSource
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return self.sections.count;
}

- (NSInteger)tableView:(UITableView *)tableView
                numberOfRowsInSection:(NSInteger)section {
    return self.messages[self.sections[section]].count + 1;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *cellId = [self cellIdForIndexPath:indexPath];
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cellId];

    if (indexPath.row == 0) {
        MVMessagePlainCell *headerCell = (MVMessagePlainCell *)cell;
        NSString *sectionTitle = self.sections[indexPath.section];
        [headerCell fillWithText:sectionTitle];
    } else {
        NSString *section = self.sections[indexPath.section];
        MVMessageModel *model = self.messages[section][indexPath.row - 1];
        if (model.type == MVMessageTypeSystem) {
            MVMessagePlainCell *systemCell = (MVMessagePlainCell *)cell;
            [systemCell fillWithText:model.text];
        } else {
            MVMessageBubbleCell *bubbleCell = (MVMessageBubbleCell *)cell;
            [bubbleCell fillWithModel:model];
            bubbleCell.indexPath = [NSIndexPath indexPathForRow:indexPath.row - 1
                                                      inSection:indexPath.section];
            bubbleCell.delegate = self;
        }
    }

    return cell;
}
```

Methods *numberOfSectionsInTableView:* and *numberOfRowsInSections:* simply return the number of elements in the related collections. Method cellForRowAtIndexPath: firstly gets the cell reuse identifier by using method *cellIdForIndexPath:* and then uses it to obtain a cell from the reusable pool. The cell is then casted to the corresponding

class, filled and returned from the method. Because header cells do not have representation in the *messages* dictionary, we simply return headers as the first cell for every section. That complicates a little the further process of handling other cells because now we need to decrement the row index by 1, but this is not critical. It could be possible to use system mechanism for returning headers using method *viewForHeaderInSection:*, but the use of this method changes the scrolling behavior of headers, which is not customizable, so the only way to provide the UI described in Chapter 4.1 is to handle them as regular cells. Apart from the *UITableViewDataSource*, the chat controller also implements some methods from the *UITableViewDelegate*. These are shown in Code snippet 79.

Code snippet 79. Delegate methods of table view in MVChatController.

```
#pragma mark - UITableViewDelegate
- (CGFloat)tableView:(UITableView *)tableView
                       heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *section = self.sections[indexPath.section];
    MVMessageModel *model;
    MVMessageCellTailType tailType = 0;
    if (indexPath.row != 0) {
        model = self.messages[section][indexPath.row - 1];
        if (model.type != MVMessageTypeSystem) {
            NSIndexPath *modelIndexPath = [NSIndexPath
                                            indexPathForRow:indexPath.row - 1
                                                  inSection:indexPath.section]
            tailType = [self messageCellTailTypeAtIndexPath: modelIndexPath];
        }
    }

    NSString *cacheKey = [NSString stringWithFormat:@"%@_%@_%lu",
                                                     section,
                                                     model.id,
                                                    (unsigned long)tailType];
    NSNumber *cachedHeight = [self.cellHeightCache objectForKey:cacheKey];

    if (cachedHeight) {
        return [cachedHeight floatValue];
    }

    CGFloat height;
    if (indexPath.row == 0) {
        height = [MVMessageHeaderCell heightWithText:section];
    } else if (model.type == MVMessageTypeSystem) {
        height = [MVMessageSystemCell heightWithText:model.text];
    } else if (model.type == MVMessageTypeText){
        height = [MVMessageTextCell heightWithTailType:tailType
                                              direction:model.direction
                                               andModel:model];
    } else {
        height = [MVMessageMediaCell heightWithTailType:tailType
                                               direction:model.direction
                                                andModel:model];
    }
    [self.cellHeightCache setObject:@(height) forKey:cacheKey];
    return height;
}
```

Code snippet 79 (continued).

```
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
                                        forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (![cell conformsToProtocol:NSProtocolFromString(@"MVSlidingCell")]) {
        return;
    }

    UITableViewCell <MVSlidingCell> *slidingCell =
                                            (UITableViewCell <MVSlidingCell> *)cell;
    CGFloat oldSlidingConstraint = slidingCell.slidingConstraint;

    if (oldSlidingConstraint != self.sliderOffset) {
        [slidingCell setSlidingConstraint:self.sliderOffset];
        [slidingCell.contentView layoutIfNeeded];
    }
}
```

Method *heightForRowAtIndexPath:* is used to calculate and return the height of the cell. To accomplish that, the method uses cell's class method, passing this responsibility to the corresponding cell class. It uses methods *heightWithTailType:direction:andModel*: and *heightWithText:* depending on the cell type. Apart from that, the chat controller has a cache of cell heights - *cellHeightCache*, which stores all the values previously calculated in this method. This enables *heightForRowAtIndexPath:* to work faster and leads to a smother table view scrolling. The cache works similarly to a dictionary, storing values for keys. It is not possible to use index path as thet key because cell height for a particular index path can change when it changes the tail type or when new messages are added at the beginning of the table view. That could happen when a new message is received, or a new page is loaded. To identify the particular cell height record, we use a string, which encapsulates section index, model's id and tail type. That is the shortest and the easiest way to calculate a unique identifier for a cell height. The determination of cell's tail type, as described during overlook of cell subclasses, is expensive and should be avoided when possible, but the actual calculation of cell size is much more expensive in comparison, and thus the use of cache is still doing a better performance when compared to the implementation without the cache. Moreover, because the MVC does not provide any intermediate model representation of cells and caching of cell tail type is hence not possible, the determination of cell tail type in *heightForRowAtIndexPath:* is needed in both implementations. That leads to comparing a solution without caching, where both cell tail type determination and cell height calculation are performed, to a solution with caching, where sometimes (actually quite often) cell calculation step can be omitted. It is clear that the solution with caching then is much desirable, assuming that both operations are expensive and querying cache is nearly immediate.

Method *wiiDisplayCell:* is fired for each cell that is about to be displayed. We use this point to inject behavior related to showing and hiding of a cell's time label. The chat controller stores the current offset in the property *sliderOffset* every time user drags the table view horizontally. It then asks each cell for the current value of a constraint attached to the time label, compares that value to the one stored in the *sliderOffset* property and if needed asks cell to update its layout. To distinguish cells that react to

the user dragging the table view, we use protocol *MVSlidingCell*. Cells that are not conforming to the protocol are ignored in *willDisplayCell:* method. The protocol has only two methods - for getting the current constraint value, and for setting the constraint value. The only class that conforms to the protocol at this point is *MVMessageBubbleCell*. Another part of supporting the drag gesture and showing the cell's time label is the use of pan gesture recognizer. The action fired when the gesture recognizer's state is changed shown in Code snippet 80.

Code snippet 80. IBAction associated with pan gesture.

```
- (IBAction)handlePanGesture:(UIPanGestureRecognizer *)panRecognizer {
    NSMutableArray<id <MVSlidingCell>> *visibleCells = [NSMutableArray new];

    for (UITableViewCell *cell in self.messagesTableView.visibleCells) {
        if ([cell conformsToProtocol:NSProtocolFromString(@"MVSlidingCell")]) {
            [visibleCells addObject:(id <MVSlidingCell>)cell];
        }
    }

    if (!visibleCells.count) {
        return;
    }

    if (panRecognizer.state == UIGestureRecognizerStateEnded ||
        panRecognizer.state == UIGestureRecognizerStateFailed ||
        panRecognizer.state == UIGestureRecognizerStateCancelled) {
        CGFloat constant = 0;
        for (MVMessageTextCell *cell in visibleCells) {
            [cell setSlidingConstraint:constant];
        }

        self.sliderOffset = constant;

        [UIView animateWithDuration:0.2 animations:^{
            [self.view layoutIfNeeded];
        }];

        return;
    }

    CGFloat oldConstant = [visibleCells[0] slidingConstraint];
    CGFloat constant = [panRecognizer translationInView:self.view].x;
    CGFloat velocityX = [panRecognizer velocityInView:self.view].x;

    if (constant > 0) constant = 0;

    if (constant < -40) constant = -40;

    if (oldConstant != constant) {
        CGFloat path = ABS(oldConstant - constant);
        NSTimeInterval duration = path / velocityX;
        for (MVMessageTextCell *cell in visibleCells) {
            [cell setSlidingConstraint:constant];
        }

        self.sliderOffset = constant;

        [UIView animateWithDuration:duration animations:^{
            [self.messagesTableView layoutIfNeeded];
        }];
    }
}
```

First of all, we filter visible cells to work only with cells which conforms to protocol *MVSlidingCell*. If there is none of them, the method just returns as soon as no additional work is needed. Otherwise, the method checks the state of the recognizer and handles the behavior when it is canceled (when the user touches up). In this case, we update the time label related constraints, update chat controller's property *sliderOffset* and animate the constraints to hide the time label. If the state is not canceled, the method computes the translation and velocity of the pan gesture and then use these values to again update the constraint and *sliderOffset* property with the animation duration corresponding to the velocity.

This far we covered how chat controller loads messages, how it displays them in the table view and how it handles some of UI actions. Another part of the the data handling functionality is reacting to receiving a new message. That is implemented in the same way for new incoming and new outgoing message and lays in supporting method *insertNewMessage:* from *MVMessagesListener* protocol. The implementation of this method is shown in Code snippet 81.

Code snippet 81. Support of message listener in MVChatController.

```
- (void)insertNewMessage:(MVMessageModel *)message {
    self.processingMessages = YES;
    [self handleNewMessage:message];
    self.processingMessages = NO;
}

- (void)handleNewMessage:(MVMessageModel *)message {
    NSMutableArray *sections = [self.sections mutableCopy];
    NSMutableDictionary *messages = [self.messages mutableCopy];

    NSString *sectionKey = [self headerTitleFromMessage:message];
    NSMutableArray *rows = messages[sectionKey];

    BOOL insertedSection = NO;
    if (!rows) {
        insertedSection = YES;
        rows = [NSMutableArray new];
        [messages setObject:rows forKey:sectionKey];
        [sections addObject:sectionKey];
    }

    [rows addObject:message];

    self.messages = [messages mutableCopy];
    self.sections = [sections mutableCopy];

    self.autoscrollEnabled = (self.messagesTableView.contentOffset.y >=
                             (self.messagesTableView.contentSize.height -
                             self.messagesTableView.frame.size.height - 50));
    self.processingNewPage = NO;
```

Code snippet 81 (continued).

```
    if (insertedSection) {
        NSIndexSet *insert = [NSIndexSet
                                          indexSetWithIndex:self.sections.count - 1];
       [self.messagesTableView insertSections:insert
                              withRowAnimation:UITableViewRowAnimationBottom];
    } else {
        NSIndexPath *indexPathToInsert = [NSIndexPath indexPathForRow:rows.count
                                                      inSection:sections.count - 1];
        NSIndexPath *indexPathToReload =
                                   [NSIndexPath indexPathForRow:rows.count - 1
                                                      inSection:sections.count - 1];
        [self.messagesTableView insertRowsAtIndexPaths:@[indexPathToInsert]
                                  withRowAnimation:UITableViewRowAnimationBottom];
        [self.messagesTableView reloadRowsAtIndexPaths:@[indexPathToReload]
                                  withRowAnimation:UITableViewRowAnimationNone];
    }
    [[MVChatManager sharedInstance] markChatAsRead:self.chatId];
}
```

Method *insertNewMessage:* is called by the chat manager every time a new message is added to the chat. The method implementation modifies flag *processingMessages* to ensure unique access to messages-related collections and calls method *handleNewMessage:*, which performs the main work. Method *handleNewMessage:* works similarly to the method *handleNewMessagesPage:* discussed earlier. It again creates copies of controller's collections, inserts new data into them and finally assigns their pointers to original properties, as well as, updates flags *autoscrollEnabled* and *procesingNewPage*. The only difference is that it does not reload the table view after the data is changed. Instead it performs batch updates - inserts a new section or a new row with the reload of the previous. At the end of the execution, it asks the chat manager to mark the chat as read as it as done in *handleMessagesPage:* method.
To support updates of the chat object, the controller implements method *updateChat:withSorting:newIndex:* from protocol *MVChatsUpdatesListener*. The method simply compares chat ids to filter updates only about the currently opened chat and then updates the chat object and string displayed in *chatTitleLabel* in the navigation bar. The implementation is shown in Code snippet 82.

Code snippet 82. Support of chat listener in MVChatController.

```
- (void)updateChat:(MVChatModel *)chat
       withSorting:(BOOL)sorting
          newIndex:(NSUInteger)newIndex {
    if ([chat.id isEqualToString:self.chat.id]) {
        self.chat = chat;
        self.chatTitleLabel.text = chat.title;
    }
}
```

To react to avatar changes, the controller subscribes to *NSNotifications* coming from *MVFileManager*. Even though updates are related to the contact model or chat model, avatar changes do not affect the actual model object, and thus another mechanism is used.

In the *viewDidLoad* method, the controller also registers to receive *NSNotifications* about keyboard appearance and sets up a KVO observation on the content size of the *messagesTableView*. These observations are needed to correctly adapt the interface to the size of the keyboard and data addition to the table view. Consider methods shown in Code snippet 83.

Code snippet 83. Support for keyboard appearance in MVChatController.

```
#pragma mark - Keyboard
- (void)keyboardWillShow:(NSNotification *)notification {
    if (!self.keyboardShown) {
        [self adjustContentOffsetDuringKeyboardAppear:YES
                                          withNotification:notification];
        self.keyboardShown = YES;
    }
}
- (void)keyboardWillHide:(NSNotification *)notification {
    if (self.keyboardShown) {
        [self adjustContentOffsetDuringKeyboardAppear:NO
                                          withNotification:notification];
        self.keyboardShown = NO;
    }
}

- (void)adjustForKeyboardAppear:(BOOL)appear
               withNotification:(NSNotification *)notification {...}
```

Methods *keyboardWillShow:* and *keyboardWillHide:* are called when appropriate *NSNotifications* are received. They both call *adjustForKeyboardAppear:withNotification:*, which in turn changes the value of *footerBottom* constraint and move footer appropriately, as well as, recalculate *contentOffset* and *contentInset* of the table view accordingly to the size of the keyboard.
Code snippet 84 shows other methods related to the update of *contentOffset* and *contentInset* properties of the table view.

Code snippet 84. Other methods related to the update of content offset and inset.

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary<NSKeyValueChangeKey,id> *)change
                       context:(void *)context {
    if (object != self.messagesTableView
       || ![keyPath isEqualToString:@"contentSize"]) {
        return;
    }
    CGSize oldSize = [[change objectForKey:NSKeyValueChangeOldKey] CGSizeValue];
    CGSize newSize = [[change objectForKey:NSKeyValueChangeNewKey] CGSizeValue];

    if (CGSizeEqualToSize(oldSize, newSize)) {
        return;
    }

    NSTimeInterval duration = (!self.processingNewPage
                               && self.autoscrollEnabled)? 0.2 : 0;
```

Code snippet 84 (continued).

```
    [UIView animateWithDuration:duration animations:^{
        [self updateContentOffsetForOldContent:oldSize
                                 andNewContent:newSize
                             processingNewPage:self.processingNewPage
                             autoScrollEnabled:self.autoscrollEnabled];

        [self updateContentInsetForNewContent:newSize
                            frame:self.messagesTableView.frame.size.height];
        }];
    }
}

- (void)updateContentInsetForNewContent:(CGSize)contentSize
                                  frame:(CGFloat)frameHeight {...}

- (void)updateContentOffsetForOldContent:(CGSize)oldSize
                           andNewContent:(CGSize)newSize
                       processingNewPage:(BOOL)processingNewPage
                       autoScrollEnabled:(BOOL)autoScroll {...}
```

Method *observeValueForKeyPath:* is fired when any of KVO observed properties are changed. In case of the chat controller, we are interested only in the change of the *contentSize* property of the *messagesTableView*, so we can safely omit any other notifications by filtering them by the *object* and the *keyPath*. Moreover, we do not want to execute anything if the content size is just reassigned without the actual change of the value, so we can filter that by the use of the macro *CGSizeEqualToSize*. If we met all of the above-menthioned conditions, we conclude that there has been an update on the content size of the table view, which should be considered to update the table view's *contentOffset* and *contentInset*. That serves multiple aims – to scroll down the table view when the scroll position was near the bottom and a new message was received, and to preserve the scroll offset after loading a new page. These methods are also called from *adjustForKeyboardAppear:* described in Code snippet 83. The rest of methods in the chat controller are either actions to handle UI events or helpers, which aim is to calculate any values mentioned in all the other methods or to create other controllers, which will be presented after appropriate UI actions. The complete list is shown in Code snippet 85.

Code snippet 85. The rest of MVChatController methods.

```
- (NSString *)headerTitleFromMessage:(MVMessageModel *)message {...}
- (NSString *)cellIdForIndexPath:(NSIndexPath *)indexPath {...}
- (MVMessageCellTailType)messageCellTailTypeAtIndexPath:(NSIndexPath *)indexPath
                                                                             {...}
- (id<MVForceTouchControllerProtocol>
                   *)forceTouchViewControllerForContext:(NSString *)context {...}
- (void)showChatSettings {...}
- (void)showContactProfile {...}
- (void)chatAvatarTapped {...}
- (IBAction)tableViewTapped:(id)sender {...}
- (void)cellTapped:(UITableViewCell *)cell {...}
- (IBAction)attatchButtonTapped:(id)sender {...}
```

Code snippet 85 (continued).

```
- (IBAction)messageTextFieldChanged:(id)sender {
    NSCharacterSet *whiteSpace = [NSCharacterSet whitespaceCharacterSet];
    NSString *trimmedText = [self.messageTextField.text
                                 stringByTrimmingCharactersInSet:whitespace];
    self.sendButton.enabled = trimmedText.length > 0;
}

- (IBAction)sendButtonTapped:(id)sender {
    self.sendButton.enabled = NO;
    [[MVChatManager sharedInstance] sendTextMessage:self.messageTextField.text
                                       toChatWithId:self.chatId];
    self.messageTextField.text = @"";
}
```

Implementation of these methods is irrelevant; it is only worth mentioning that the last two IBActions – *messageTextFieldChanged:* and *sendButtonTapped:* are used to send a new message to the chat. The first one is used to enable or disable the send button depending on the content of the text field, and the second one disables the send button, sends a text message to the chat manager and clears the text field.

5.3 Native app problems

The app described in the previous chapter introduces some problems specific to the iOS development that were discussed before. A chapter dedicated to the managers showed how many similar event-handling mechanisms are treated differently. That includes the use of the delegate pattern in the *MVContactManager* and the *MVChatManager*, *NSNotification* in the *MVFileManager*. It also shows the limitation of the delegate pattern (protocol and a link actually), when multiple objects are interested in a particular update – the use of an array of *NSValues* is definitely a workaround and not an altogether desirable solution. The chapter about controllers highlighted these limitations and introduced other event-based mechanisms based on *NSNotification* and KVO, which again serve the very similar functionality but are implemented in a completely different manner. That chapter also showed that controller and view entities are tightly bound together in Apple's MVC, which leads to an overload of the controller with responsibilities. Schematically the real relations in the Apple's MVC can be seen in Figure 22.

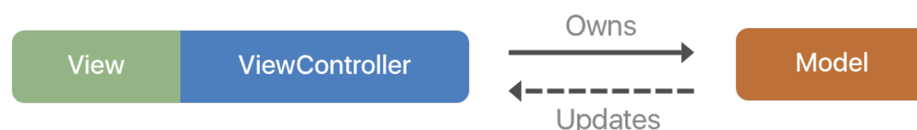

Figure 22. Massive view controller.

The chat controller covered in Chapter 4.3.2 consists of about 800-900 lines of code, which is too many for one class. Even though the number of lines in a particular file is not a metric, it is clear that chat controller is overloaded – it is responsible for handling model data, building views, responding to user actions and system events. Limitations of the MVC were in particular reflected in the chat controller functionality related to

the table view handling. Inability to store a convenient cache for some intermediate cells’ models leads to an over-computation and duplication of work, lowering the performance of the app.

6 REFACTORING WITH MVVM AND REACTIVECOCA

Considering the facts mentioned above, we can conclude that to resolve these problems we can slightly modify the Apple's MVC to allow it to have another entity, which is not so tightly bound to the view, as well as, provide a mechanism for unifying event-based mechanisms in the app. The following chapter will cover the refactoring of the app with the help of the MVVM pattern and the ReactiveCocoa framework, which are exactly the solutions we are looking for.
As was done with the development of the app, refactoring will start from changing model objects and then will continue on to the controller's modifications.

6.1 Managers

All the managers will be modified by transforming all of the event-based mechanisms to *RACSignal's*. That, at first glance a small change, is actually a robust solution, solving the problem of multicast delegates as well as providing a concise interface for similar mechanisms.

6.1.1 Contact manager

The contact manager previously had a property *updatesListener* confirming to the protocol *MVContactsUpdatesListener*. The internals of the manager are changed in a way to expose two *RACSignal's – listUpdateSignal* and *lastSeenTimeSignal*. The updated interface is shown in Code snippet 86.

Code snippet 86. Interface of MVContactManager.

```
@interface MVContactManager : NSObject
@property (strong, nonatomic) RACSignal *lastSeenTimeSignal;
@property (strong, nonatomic) RACSignal *listUpdateSignal;
+ (instancetype)sharedInstance;
- (void)loadContacts;
- (NSArray <MVContactModel *> *)getAllContacts;
+ (MVContactModel *)myContact;
- (void)handleContactLastSeenTimeUpdate:(MVContactModel *)contact;
- (void)clearAllCache;
@end
```

The rest of the interface and overall behavior remains untouched. Signal *listUpdateSignal* is used to notify the subscriber that contacts list needs to be updated and sends boolean values, but in this case, any value can be used because it does not provide any meaning to the listener, the only purpose of the signal is to send any value when the list should be updated. Signal *lastSeenTimeSignal*, on the contrary, sends meaningful data. It sends *MVContact* model object each time last seen time of the contact is changed. The subscriber then can check model's id and update the value of the *lastSeenTime* appropriately. Not only signals provide a concise interface here, but

support multicasting which enables multiple subscribers to independently listen to the signals without any additional work on the manager side.

6.1.2 Chat manager

The chat manager had a property *messagesListener* and array for chat listeners with public method *addChatListener:*. Both properties are as well replaced with *RACSignal* instances: *chatUpdateSignal* and *messageUpdate* signal correspondingly. The interface of *MVChatManager* is too long, and it is not modified in any other aspects, so the Code snippet 87 shows only occured changes.

Code snippet 87. MVChatManager interface.

```
interface MVChatManager : NSObject
@property (strong, nonatomic) RACSignal *chatUpdateSignal;
@property (strong, nonatomic) RACSignal *messageUpdateSignal;
@property (strong, nonatomic) RACScheduler *viewModelScheduler;
@property (strong, nonatomic) dispatch_queue_t viewModelQueue;
- (RACSignal *)messagesPage:(NSInteger)pageIndex forChatWithId:(NSString *)chatId;

...other methods
@end
```

Signal *chatUpdatesSignal* is used to notify subscribers about any change happening to the chat's list or any chat in particular. The signal, as well as others, just substitutes the protocol that was used previously, completely re-implementing its functionality. To support this, *chatUpdateSignal* sends small intermediate objects that are also defined in the header of *MVChatManager* class. The Code snippet 88 shows the interface of that object.

Code snippet 88. Interface of MVChatUpdate.

```
typedef enum : NSUInteger {
    ChatUpdateTypeReload,
    ChatUpdateTypeInsert,
    ChatUpdateTypeDelete,
    ChatUpdateTypeModify
} ChatUpdateType;

@interface MVChatUpdate
@property (assign, nonatomic) ChatUpdateType updateType;
@property (strong, nonatomic) MVChatModel *chat;
@property (assign, nonatomic) BOOL sorting;
@property (assign, nonatomic) NSInteger index;
@end
```

Class *MVChatUpdate* wraps update type and *MVChatModel* instance, allowing a type-safe access to the value received by the signal's subscriber. The use of such small intermediate objects is preferable over using a dictionary type because it is more transparent on the receiver side, which kind of object a signal sends. The value of the *chat* property in the class is optional and will be empty when *updateType* is going to store value *ChatUpdateTypeReload*. Objective-C does not have support for optional

types, but it is clear from the context that when the whole list is reloaded, there is no need to use a *chat* value and even potentially it can not store any meaningful data. Properties *sorting* and *index* are also optional and are used only with the update type *ChatUpdateTypeModify*.

Signal *messageUpdatesSignal* is only responsible for notifying observers when a new message is added to the internal collection, and thus it does not need any wrapper class for the sent values. Instead, it sends *MVMessageModel* objects. The sent messages are the messages that are added to the chat.

Apart from signal properties, *MVChatManager* interface also got two similar properties for storing thread-related objects. Property *viewModelQueue* is used to store a dispatch queue designated to handle work of any chat-related ViewModel. Property *viewModelScheduler* is just a wrapper of that queue to be used in the ReactiveCocoa world.

The concept of having not only queues associated with each manager but also queues associated with ViewModels helps to unload the main thread even more. ViewModel is a separate object not modifying any of UI components, so it safely can execute its code on the background thread. Even though it is not modifying the UI of the app, it works together with the controller, which accesses the UI-related parts of the app and executes on the main thread, so their synchronization should be considered on the ViewModel or view side. The queue used for encapsulating ViewModel's work should be serial because most of the time ViewModel's work cannot be paralleled and the order of different function invocation is relevant. Moreover, not all ViewModel really need to have a dedicated queue, and it depends entirely on the amount of work they are doing. In our case, any ViewModel associated with contacts can safely execute on the main thread without the necessity of using a background queue. The use of background thread, in this case, can be considered as a premature optimization when there is actually no visible problem, and optimization is just making things more complicated without any actual benefit. On the contrary, ViewModels that are going to be derived from *MVChatsListController* and *MVChatController* are likely to be in need of background queue, because of the type of work and computation complexity that they are facing.

Method *messagesPage:forChatWithId:* is changed to return a signal instead of calling a passed callback as it was done previously. Callbacks in operations where they are used to return requested data asynchronously can be changed to *RACSignal* interfaces to unify API and make handling more natural on the user side. Usability comes from consolidating interfaces for working with related objects and from the ease of use of *RACSignal's* operators related to scheduling and switching between different schedulers.

6.1.3 File manager

The mechanism of sending notifications previously used in the file manager is changed to *RACSignal* interface. This minor change not only explicitly exposes the behavior of the manager in its interface, but also make it easier on the controller side to parse notifications. Instead of parsing a dictionary, a controller now receives a wrapper object which holds all the needed information. The interface change of file manager is shown in Code snippet 89.

Code snippet 89. Interface change of MVFileManager.

```
#pragma mark - Signals
@property (strong, nonatomic) RACSignal *avatarUpdateSignal;
```

Signal *avatarUpdateSignal* send events when any avatar is being changed. The object that is being sent is of class *MVAvatarUpdate*. Its interface is shown in Code snippet 90.

Code snippet 90. Interface of MVChatAvatarUpdate.

```
@interface MVAvatarUpdate : NSObject
@property (assign, nonatomic) MVAvatarUpdateType type;
@property (strong, nonatomic) NSString *id;
@property (strong, nonatomic) UIImage *avatar;
@end
```

The class stores the same data that was previously enclosed in a dictionary of *NSNotification*. The use of the wrapper object, in this case, ensures type safety on the receiver's side as well as provides a clear understanding of which data to expect from the signal.

6.2 Views and view models

As discussed in Chapter 2.5, the MVVM pattern does not have a concept of controllers. Instead, it introduces a ViewModel entity that is responsible for handling the data for the view. In the case of the iOS, the composition of the *UIViewController* and the *UIView* is going to be considered together as a view entity, and ViewModel (which is going to be *NSObject* subclass) is going to take some of the responsibilities previously loaded onto the controller.

6.2.1 Chats list

Class *MVChatsListController* is divided into 2 classes – *MVChatsListController* and *MVChatsListViewModel*. The word controller used in the name of the *MVChatsListController* just illustrates that the class is still a subclass of the *UIViewController* but it has nothing to do with controller in the MVC pattern. Cells

used in the table view also obtain associated ViewModels – *MVChatsListCellViewModel*. This model is going to serve as intermediate representation between *MVChatModel* and the view in the form of *MVChatsListCell*. The interface and private interface extension of *MVChatsListViewModel* are shown in Code snippet 91.

Code snippet 91. Interface of MVChatsListViewModel.

```
@interface MVChatsListViewModel : NSObject <UISearchResultsUpdating>
@property (strong, nonatomic, readonly) NSArray <MVChatsListUpdate *> *listUpdates;
@property (assign, nonatomic, readonly) BOOL shouldShowPopularData;
@property (strong, nonatomic) MVChatsListCellViewModel *recentSearchChat;
@property (strong, nonatomic, readonly)
                                  NSArray <MVChatsListCellViewModel *> *chats;
@property (strong, nonatomic, readonly)
                              NSArray <MVChatsListCellViewModel *> *filteredChats;
@property (strong, nonatomic, readonly)
                               NSArray <MVChatsListCellViewModel *> *popularChats;
@end

@interface MVChatsListViewModel ()
@property (strong, nonatomic, readwrite) NSArray *chats;
@property (strong, nonatomic, readwrite) NSArray *filteredChats;
@property (strong, nonatomic, readwrite) NSArray *popularChats;
@property (assign, nonatomic, readwrite) BOOL shouldShowPopularData;
@property (strong, nonatomic, readwrite) NSArray *listUpdates;
@end
```

The class stores properties for all data-related objects that were previously stored in the controller as well as such properties of the search controller. The ViewModel is going to provide data not only to the *MVChatsListController* but also to the *MVChatsListSearchController*. There are some duplicates of properties in the public interface, and its private extension. These properties are not meant to be modified outside of the *MVChatsListViewModel* class, and thus they are marked with the *readonly* attribute. That makes the intent clear, and to support changing the properties in the class, we need to override their definition in the interface extension with a property attribute *readwrite*. Apart from the invariants migrated from the controller, the ViewModel also has a new property – *listUpdates* array. This array is used to store updates happening in the chat's array in the form that is the most convenient for the view. The objects that are stored in this array are of type *MVChatsListUpdate*. The interface of this class in also defined in the header of the MVChatsListViewModel and can be seen in Code snippet 92.

Code snippet 92. Interface of MVChatsListUpdate.

```
typedef enum : NSUInteger {
    MVChatsListUpdateTypeReloadAll,
    MVChatsListUpdateTypeReload,
    MVChatsListUpdateTypeInsert,
    MVChatsListUpdateTypeDelete,
    MVChatsListUpdateTypeMove
} MVChatsListUpdateType;
```

Code snippet 92 (continued).

```
@interface MVChatsListUpdate : NSObject
@property (assign, nonatomic) MVChatsListUpdateType updateType;
@property (strong, nonatomic) NSIndexPath *startIndexPath;
@property (strong, nonatomic) NSIndexPath *endIndexPath;
@property (strong, nonatomic) NSIndexPath *insertIndexPath;
@property (strong, nonatomic) NSIndexPath *removeIndexPath;
@property (strong, nonatomic) NSIndexPath *reloadIndexPath;
@end
```

This class stores information about the update that should be translated to the table view. Its property *updateType* uniquely identifies the type of a batch update, or a complete table view reload, and multiple *NSIndexPath's* properties are used to pass data relevant to the specified update. It could be possible to store only two *NSIndexPath* properties because this is the most amount that is needed for any particular update, however, the use of descriptive naming associated with each update type makes the semantics of the property clearer.

The objects stored in the *chats* array also changed their class, comparing to the same array in the native app. Now the array does not store *MVChat* objects but instead stores cell's ViewModels of class *MVChatsListCellViewModel*. These objects partially mirror the data stored in the *MVChatModel*, but with some extra tuning, which makes them easier to use in the view. The point here is to maximally simplify the code in the view instances, taking all the responsibility of model handling to the ViewModel. The interface of *MVChatsListCellModel* can be seen in Code snippet 93.

Code snippet 93. Interface of MVChatsListCellViewModel.

```
@interface MVChatsListCellViewModel : NSObject
@property (strong, nonatomic) MVChatModel *chat;
@property (strong, nonatomic) NSString *title;
@property (strong, nonatomic) NSString *message;
@property (strong, nonatomic) NSString *unreadCount;
@property (strong, nonatomic) NSString *updateDate;
@property (strong, nonatomic) UIImage *avatar;
@end
```

MVChatsListCellViewModel is a pure data class; it does not support any methods but entirely mirrors the view hierarchy of the cell, with which it is associated. Cell in turn needs only to obtain the data from its ViewModel and fill the corresponding views, which is trivial.

Now let us look at the *MVChatsListController* interface and methods. The Code snippet 94 shows the private interface extension of the class, the public interface remains empty.

Code snippet 94. Interface of MVChatsListViewController.

```
@interface MVChatsListViewController () <UITableViewDelegate,
UITableViewDataSource, MVForceTouchPresentaionDelegate, UICollectionViewDelegate>
@property (strong, nonatomic) IBOutlet UITableView *chatsList;
@property (strong, nonatomic) IBOutlet UIButton *createChatButton;
@property (nonatomic) MVChatsListSearchViewController *searchResultsController;
@property (strong, nonatomic) UISearchController *searchController;
@property (strong, nonatomic) MVChatsListViewModel *viewModel;
@end
```

Now the controller does not store any other properties than ones related to the view objects and property for storing its ViewModel. The initialization of the controller is also simplified. Corresponding methods are shown in Code snippet 95. The controller is responsible for initialization of its ViewModel and it does it even before the view is fully loaded. As soon as the ViewModel is mainly working on the background queue, this can improve the overall performance of the view. The data potentially can be already processed by the ViewModel at the time when *viewDidLoad* is called and controller just left with displaying it.

Code snippet 95. Initialization related methods of MVChatsListViewController.

```
#pragma mark - Initialization
- (instancetype)initWithCoder:(NSCoder *)aDecoder {
    if (self = [super initWithCoder:aDecoder]) {
        _viewModel = [MVChatsListViewModel new];
    }

    return self;
}

#pragma mark - View lifecycle
- (void)viewDidLoad {
    [super viewDidLoad];
    ...allold view setups

    [self bindAll];
    [self.chatsList reloadData];
}
```

In the *viewDidLoad*, the controller does all the view setups related to the search controller and force touch recognition that was done previously, setups binding to the ViewModel and reloads the table view. The code responsible to bind to the ViewModel is shown in Code snippet 96.

Code snippet 96. Binding of MVChatsListViewController.

```
- (void)bindAll {
    @weakify(self);
    [[[RACObserve(self.viewModel, listUpdates) skip:1] deliverOnMainThread]
        subscribeNext:^(NSArray *updates) {
            @strongify(self);
            for (MVChatsListUpdate *update in updates) {
                switch (update.updateType) {
                    case MVChatsListUpdateTypeReloadAll:
                        [self.chatsList reloadData];
                        break;

                    case MVChatsListUpdateTypeInsert:
                        [self.chatsList
                             insertRowsAtIndexPaths:@[update.insertIndexPath]
                                   withRowAnimation:UITableViewRowAnimationBottom];
                        break;

                    case MVChatsListUpdateTypeDelete:
                        [self.chatsList
                          deleteRowsAtIndexPaths:@[update.removeIndexPath]
                                withRowAnimation:UITableViewRowAnimationAutomatic];
                        break;

                    case MVChatsListUpdateTypeMove:
                        [self.chatsList moveRowAtIndexPath:update.startIndexPath
                                               toIndexPath:update.endIndexPath];
                        break;

                    case MVChatsListUpdateTypeReload:
                        [UIView setAnimationsEnabled:NO];
                        [self.chatsList
                                reloadRowsAtIndexPaths:@[update.reloadIndexPath]
                                withRowAnimation:UITableViewRowAnimationAutomatic];
                        [UIView setAnimationsEnabled:YES];
                        break;

                    default:
                        break;
                }
            }
        }];

    [[self.createChatButton rac_signalForControlEvents:UIControlEventTouchUpInside]
                                         subscribeNext:^(__kindof UIControl *x) {
        @strongify(self);
        [self createNewChat];
    }];
}
```

The binding consists of using *RACObserve* macro to create a signal from a KVO on the ViewModel's property *listUpdates* and then subscribing to that signal. In the subscription block controller responds to different types of updates by calling appropriate methods of the table view. The signal has two modifying operators – *skip:1* and *deliverOnMainThread*. The first one is used to skip the first value of the signal, which is irrelevant to the controller as soon as it is calling table view's *reloadData* in the *viewDidLoad* in any case, and the second is used to schedule the subscription block on the main queue, which is obligatory for the UI manipulations.

Another signal used in the *bindAll* method is related to the button tap handling. That is accomplished by the use of rac interface added to all the *UIControlls* –

rac_signalForControlEvents. The returned signal is sending values when the specified control events are happening. This subscription block is an equivalent of IBAction use. However, it provides a concise *RACSignal* interface.

Macros *@weakify* and *@strongify* used in the previous snippet are used to create correspondingly weak and strong references to the object passed to the macro. As soon as these signals are eternal – they last the lifetime of the associated objects, and subscription's blocks are just Objective-C blocks, retaining a strong link to *self* in the block is causing a reference cycle. To break this cycle, these macros are used.

Another method that is somehow related to the major functionality of the view is *cellForRowAtIndexPath:* from *UITableViewDataSource*. The method is shown in Code snippet 97.

Code snippet 97. Method cellForRowAtIndexPath: from MVChatsListViewController.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    MVChatsListCell *cell = [tableView
                              dequeueReusableCellWithIdentifier:@"ChatsListCell"];
    MVChatsListCellViewModel *model = self.viewModel.chats[indexPath.row];
    [cell fillWithModel:model];

    return cell;
}
```

Implementation of the method is rather simple – it just obtains a cell by the reuse identifier, gets a cell's ViewModel from the controller's ViewModel and fills the cell with its ViewModel. The cell, in turn, is responsible for filling out its views. The implementation of *fillWithModel:* method is shown in Code snippet 98.

Code snippet 98. Method fillWithModel: from MVChatsListCell.

```
- (void)fillWithModel:(MVChatsListCellViewModel *)model {
    self.titleLabel.text = model.title;
    self.messageLabel.text = model.message;
    self.dateLabel.text = model.updateDate;

    if (model.unreadCount) {
        [self.unreadCountButton setTitle:model.unreadCount
                                forState:UIControlStateNormal];
        self.unreadCountButton.hidden = NO;
    } else {
        self.unreadCountButton.hidden = YES;
    }

    RAC(self.avatarImageView, image) = [[RACObserve(model, avatar)
                    deliverOnMainThread] takeUntil:self.rac_prepareForReuseSignal];

    self.chatModel = model.chat;
}
```

Here cell fills all the views as well as subscribes to the signal related to the avatar updates. As soon as an avatar of a particular cell can be changed, this change can be handled by the main ViewModel, propagated to the cell's ViewModel and eventually

be determined by the cell itself. Such solution efficiently balances responsibilities between different entities, leaving the cell with the only one – listening to its own ViewModel without full understanding how the underlying change is occurring. The mechanism is built on using two macros – *RAC* and *RACObserve*. Macros *RACObserve* as described earlier, setups a KVO over passed object and returns the signal, which sends values each time the object is changed. Macros *RAC*, in turn, is used to create an automatic subscription to a signal and to save all the sent values to the passed property – image of the image view in this case. The signal also uses operator *deliverOnTheMainThread* to ensure that image view is updated on the main queue and *takeUntil: self.rac_prepareForReuseSignal*. The latter is used to terminate the signal subscription and eventually the signal itself as soon as the cell is added to the reuse pool. That helps to avoid reference cycle and unpredictable behavior of the cell's avatar image view. Because every time the cell is reused it is being filled and associated with another ViewModel, it should forget about all the bindings related to the previously used ViewModel.

The cell does not have any other interesting methods, as well as the view. All the rest methods just migrated from the native implementation and are responsible for view setup and trivial support of the UI protocols.

The *MVChatsListViewModel* initialization is shown in Code snippet 99. Method *setupBindings* is large but contains independent blocks of execution that are going to be listed separately.

Code snippet 99. Initialization of MVChatsListViewModel.

```
#pragma mark - Lifecycle
- (instancetype)init {
    if (self = [super init]) {
        [self setupBindings];
    }

    return self;
}

- (void)setupBindings {
    MVChatManager *chatManager = [MVChatManager sharedInstance];
    RACScheduler *viewModelScheduler = chatManager.viewModelScheduler;
    RACSignal *chatUpdateSignal = [chatManager.chatUpdateSignal
                                               deliverOn:viewModelScheduler];

    self.chats = [self viewModelsForChats:chatManager.chatsList];
    self.listUpdates = @[[MVChatsListUpdate reloadAllUpdate]];

    ...other bindings
}
```

During initialization the ViewModel stores frequently used instances to variables – this includes *chatManager*, *viewModelScheduler* and *chatUpdatesSignal*. Signal *chatUpdatesSignal* obtained from the manager is also modified to schedule all the events to the *viewModelSheduler*. After that, the ViewModel obtains the current list of chats from the chat manager and stores the initial value to the *listUpdates* property.

The data handling initiated by the *chatUpdatesSignal* is set up later in the method and separated to independent execution blocks by the update type. The Code snippet 100 shows the setup of the *reloadSignal* and the *insertSignal*, responsible for handling updates of *ChatUpdateTypeReload* and *ChatUpdateTypeInsert* types.

Code snippet 100. Setup of reloadSignal and insertSignal inside of setupBindings method.

```
@weakify(self);
RACSignal *reloadSignal = [[[[chatUpdateSignal filter:^BOOL(MVChatUpdate
*listUpdate) {
    return listUpdate.updateType == ChatUpdateTypeReload;
}] map:^id(MVChatUpdate *listUpdate) {
   @strongify(self);
   return [self viewModelsForChats:chatManager.chatsList];
}] doNext:^(NSArray *models) {
   @strongify(self);
   self.chats = models;
}] map:^id (NSArray *models) {
   return @[[MVChatsListUpdate reloadAllUpdate]];
}];

RACSignal *insertSignal = [[[[chatUpdateSignal filter:^BOOL(MVChatUpdate
*listUpdate) {
    return listUpdate.updateType == ChatUpdateTypeInsert;
}] map:^id (MVChatUpdate *listUpdate) {
    @strongify(self);
    return [self viewModelForChat:listUpdate.chat];
}] doNext:^(MVChatsListViewModel *model) {
    @strongify(self);
    NSMutableArray *chats = [self.chats mutableCopy];
    [chats insertObject:model atIndex:0];
    self.chats = [chats copy];
}] map:^id (MVChatsListViewModel *model) {
    NSIndexPath *insertIndexPath = [NSIndexPath indexPathForRow:0 inSection:0];
    return @[[MVChatsListUpdate insertUpdateWithIndex:insertIndexPath]];
}];
```

Both signals first filter the value received from *chatUpdateSignal* by the *updateType* to handle only events, in which they are interested. After *reloadSignal* receives a value, it obtains a current list of chats from the chat manager, creates cells' ViewModels and stores them in the *chats* property. As the final steps, it maps the received value to an object of class *MVChatsListUpdate*, which later will be used by the view. Signal *isnertSignal* does similar actions – it generates cell's ViewModel for the received chat and inserts it into the *chats* array. As well as the *reloadSignal*, it maps the value to the *MVChatsListUpdate* object. Code snippet 101 contains two other signals responsible for updates of *ChatUpdateTypeDelete* and *ChatUpdateTypeModify* types.

Code snippet 101. Setup of deleteSignal and modifySignal inside of setupBindings method.

```
RACSignal *deleteSignal = [[[[[chatUpdateSignal filter:^BOOL(id listUpdate) {
    return listUpdate.updateType == ChatUpdateTypeDelete;
}] map:^id(MVChatUpdate *listUpdate)
    @strongify(self);
    return @([self indexOfChat:listUpdate.chat]);
}] filter:^BOOL(NSNumber *index) {
    return index.integerValue != NSNotFound;
}] doNext:^(NSNumber *index) {
    @strongify(self);
    NSMutableArray *chats = [self.chats mutableCopy];
    [chats removeObjectAtIndex:index.integerValue];
    self.chats = [chats copy];
}] map:^id (NSNumber *index) {
    NSIndexPath *deleteIndexPath = [NSIndexPath indexPathForRow:index.integerValue
                                                      inSection:0];
    return @[[MVChatsListUpdate deleteUpdateWithIndex: deleteIndexPath]];
}];
RACSignal *modifySignal = [[[[chatUpdateSignal filter:^BOOL(id listUpdate) {
    return listUpdate.updateType == ChatUpdateTypeModify;
}] map:^id (MVChatUpdate *listUpdate) {
    @strongify(self);
    return RACTuplePack(@(listUpdate.sorting),
                        [self viewModelForChat:listUpdate.chat],
                        @([self indexOfChat:listUpdate.chat]),
                        @(listUpdate.index));
}] doNext:^(RACTuple *tuple) {
    RACTupleUnpack(NSNumber *sorting, MVChatsListCellViewModel *model,
                   NSNumber *oldIndex, NSNumber *newIndex) = tuple;
    @strongify(self);
    NSMutableArray *chats = [self.chats mutableCopy];
    if (sorting.boolValue) {
        [chats removeObjectAtIndex:oldIndex.integerValue];
        [chats insertObject:model atIndex:newIndex.integerValue];
    } else {
        [chats replaceObjectAtIndex:oldIndex.integerValue withObject:model];
    }
    self.chats = [chats copy];
}] map:^id (RACTuple *tuple) {
    RACTupleUnpack(NSNumber *sorting, MVChatsListCellViewModel *model,
                   NSNumber *oldIndex, NSNumber *newIndex) = tuple;
    NSMutableArray *updates = [NSMutableArray new];
    if (sorting.boolValue) {
        NSIndexPath *oldIndexPath = [NSIndexPath
                                     indexPathForRow:oldIndex.integerValue
                                           inSection:0];
        NSIndexPath *newIndexPath = [NSIndexPath
                                     indexPathForRow:newIndex.integerValue
                                           inSection:0];
        MVChatsListUpdate *move = [MVChatsListUpdate
                                   moveUpdateWithStartIndex:oldIndexPath
                                                   endIndex:newIndexPath];
        [updates addObject:move];
    }
    NSInteger rowToReload = sorting.boolValue?
                            newIndex.integerValue : oldIndex.integerValue;
    NSIndexPath *reloadIndexPath = [NSIndexPath indexPathForRow:rowToReload
                                                      inSection:0]
    MVChatsListUpdate *reload = [MVChatsListUpdate
                                 reloadUpdateWithIndex:reloadIndexPath];
    [updates addObject:reload];
    return updates.copy;
}];
```

These signals, as well as the ones, discussed earlier firstly filter the update received from the chat manager to receive only events of the corresponding type. Signal *deleteSignal* then finds the index of the needed chat, removes associated cells' ViewModel from the *chats* array and eventually maps the value to an *MVChatsListUpdate* object, passing there the index of the deleted row. Signal *modifySignal* creates a cell's ViewModel, finds the current index of the modified chat and passes all these data as well as some of the data encapsulated in the *MVChatUpdate* object to the next block with the use of the *RACTuple* object. Macros *RACTuplePack* and *RACTupleUnpack* here are used correspondingly to create a *RACTuple* with passed objects and to unwrap a tuple and store its values in the passed variables. The signal modifies the *chat* array according to the *sorting* flag and creates *MVChatsListUpdate* objects, which are then added to the array and returned from the last map operator.

Described above signals inject side effects with the use of such operators as *map*, *doNext,* and *filter*. All the signals are designed to do some work using these side effects and eventually map the values to an array of *MVChatsListUpdate* instances. The code encapsulated into side-effect blocks of a signal is going to be executed only when there is a subscription to that signal, so the next step is to subscribe to all update-related signals. That is also done in the body of the *setupBindings* method and showed in Code snippet 102, as well as another not-related subscription finishing the method.

Code snippet 102. Final setup of bindings inside setupBindings method.

```
RAC(self, listUpdates) = [RACSignal merge:@[reloadSignal, insertSignal,
                                             deleteSignal, modifySignal]];

RAC(self, shouldShowPopularData) = [[RACObserve(self, filteredChats)
map:^id(NSArray *chats) {
    return @((BOOL)chats);
}] not];
```

Signals described previously are combined using *merge* operator, which creates a new signal that will send all the values from the passed signals at the same time they send values. The resulting signal then writes all the values to the property *listUpdates* with the help of the *RAC* macros. That inherently creates the subscription needed to perform the work encapsulated in the signals as side effects and stores update objects to the ViewModel's property *listUpdates* that is then observed by the view to update its contents. Essentially, the binding process setups a tunnel from the chat manager to the view, modifying events coming from the manager and performing some additional work as these events occur. The signal's side-effects work is serialized on the *viewModelScheduler* which ensures that the view will receive a value only when side effects work is finished.

Another binding shown in the previous listing is mapping the value of *filteredChats* array to the *shouldShowPopularData* property – the property will be *YES* if only *filteredChats* pointer is *nil* and vice versa.

The rest of *MVChatsListViewModel* methods are mostly helpers used to create cell's ViewModels and filter them to support search functionality. The complete list is shown in Code snippet 103.

Code snippet 103. The rest of MVChatsListViewModel methods.

```
- (MVChatsListCellViewModel *)viewModelForChat:(MVChatModel *)chat {...}
- (NSArray <MVChatsListCellViewModel *> *)viewModelsForChats:
                                            (NSArray <MVChatModel *> *)chats {...}
- (void)updateSearchResultsForSearchController:(UISearchController
*)searchController {...}
- (NSArray *)filterChatsWithString:(NSString *)string {...}
- (NSUInteger)indexOfChat:(MVChatModel *)chat {...}
- (NSString *)textFromMessage:(MVMessageModel *)message {...}
- (NSString *)textFromUpdateDate:(NSDate *)date {...}
```

Implementation of listed above methods is trivial and mostly was previously done in the code of *MVChatsListController*. Implementation of *viewModelForChat:* method is partially interesting because it does not only create a cell's ViewModel but also setups a binding for the avatar updates that was described when discussing cells. The implementation is shown in Code snippet 104.

Code snippet 104. Implementation of viewModelForChat: method.

```
- (MVChatsListCellViewModel *)viewModelForChat:(MVChatModel *)chat {
    MVChatsListCellViewModel *viewModel = [MVChatsListCellViewModel new];
    viewModel.chat = chat;
    if (chat.isPeerToPeer) {
        viewModel.title = chat.getPeer.name;
    } else {
        viewModel.title = chat.title;
    }
    viewModel.message = [self textFromMessage:chat.lastMessage];

    if (chat.unreadCount != 0) {
        viewModel.unreadCount = [NSString stringWithFormat:@"%lu",
                                 (unsigned long)chat.unreadCount];
    }
    viewModel.updateDate = [self textFromUpdateDate:chat.lastUpdateDate];

    [[MVFileManager sharedInstance] loadThumbnailAvatarForChat:chat
                                                      maxWidth:50
                                                    completion:^(UIImage *image) {
        viewModel.avatar = image;
    }];
    RAC(viewModel, avatar) =
    [[[[MVFileManager sharedInstance].avatarUpdateSignal filter:^BOOL(id update) {
        if (chat.isPeerToPeer) {
            return (update.type == MVAvatarUpdateTypeContact
                    && [update.id isEqualToString:chat.getPeer.id]);
        } else {
            return (update.type == MVAvatarUpdateTypeChat
                    && [update.id isEqualToString:chat.id]);
        }
    }] map:^id (MVAvatarUpdate *update) {
        return update.avatar;
    }] takeUntil:viewModel.rac_willDeallocSignal];

    return viewModel;
}
```

The method is responsible for mapping *MVChatModel* to *MVChatsListCellViewModel*, which is more convinient for the cell to display. Apart from the object creation and filling the properties, the method setups a binding for supporting avatar updates. The signal exposed by the *MVFileManager* is filtered to pass only events about a particular chat, mapped to send an actual *UIImage* value and eventually binded to the *avatar* property of the cell's ViewModel. This property is then going to be observed by the cell to update the contents of the corresponding *UIImageView* object. Operator *takeUntil:* is also applied to the signal to discard it when associated ViewModel is removed from the memory to avoid a memory leak in that place.

Refactoring of the *MVChatListController* reduced its size, shifted some of its responsibilities to the ViewModel and introduced a clearer and more concise implementation. Now every object has an area of responsibilities which is sharply different from another. All of the data flow in this part of the app are described in a declarative way and thus are easy to read and understand. The performance of the controller is also improved because ViewModel as an independent object can work on a dedicated background queue.

6.2.2 Chat controller

The chat controller is refactored similarly – by adding the *MVChatViewModel* as the view's ViewModel and the *MVMessageCellModel* as the cell's ViewModel. In contrast to the chats list controller, ViewModels related to the chat controller are going to take much more responsibility and overall be capable of much more functionality because the refactored controller is much complex in turn. The interface of the *MVChatViewModel* as well as its private interface extension are shown in Code snippet 105.

Code snippet 105. Interface of MVChatViewModel.

```
@interface MVChatViewModel : NSObject
- (instancetype)initWithChat:(MVChatModel *)chat;
@property (strong, nonatomic) NSArray <MVMessageCellModel *> *rows;
@property (strong, nonatomic) UIImage *avatar;
@property (strong, nonatomic) NSString *title;
@property (strong, nonatomic) NSString *messageText;
@property (strong, nonatomic) RACSignal *updateSignal;
@property (strong, nonatomic) RACCommand *sendCommand;
@property (assign, nonatomic) CGFloat sliderOffset;
@property (strong, nonatomic) NSString *chatId;
@property (strong, nonatomic) NSArray *chatParticipants;
- (void)tryToLoadNextPage;
- (UIViewController *)relevantSettingsController;
- (UIViewController *)attachmentPicker;
- (void)imageViewerForMessage:(MVMessageCellModel *)model
                fromImageView:(UIImageView *)imageView
                   completion:(void (^)(UIViewController *))completion;
@end
```

Code snippet 105 (continued).

```
@interface MVChatViewModel()
@property (strong, nonatomic) MVChatModel *chat;
@property (strong, nonatomic) RACScheduler *scheduler;
@property (strong, nonatomic) dispatch_queue_t queue;
@property (strong, nonatomic) NSMutableArray <MVMessageCellModel *> *messages;
@property (assign, nonatomic) BOOL processingMessages;
@property (assign, nonatomic) BOOL initialLoadComplete;
@property (assign, nonatomic) NSInteger loadedPageIndex;
@property (assign, nonatomic) NSInteger numberOfProcessedMessages;
@property (strong, nonatomic) RACReplaySubject *updateSubject;
@property (nonatomic, copy) BOOL (^insertMessage)(MVMessageModel *,
                                                  MVMessageModel *,
                                                  MVMessageModel *,
                                   NSMutableArray <MVMessageCellModel *> *, BOOL);
@end
```

The interface stores data-related properties migrated from the *MVChatController* as well as some new properties. It provides a simple interface for the view to obtain desired data objects. For example, avatar and title are not queried from some other objects; they are exposed in a one-step way. Apart from these simple data properties, the ViewModel stores *updateSignal*, which is used by the view to receive update events and *sendCommand*, which is used as *RACCommand* for the send button. Interestingly, but the ViewModel has two properties for storing cell's ViewModels – array *rows* and mutable array *messages*. This separation will be covered shortly when we will get to the view's code. Briefly, it is used to provide better thread-safety for the controller. Another interesting property, which was not presented before, is *insertMessage*. It is a block taking multiple parameters and returning a boolean. The block is used internally by the ViewModel to insert new messages into the collection passed to the block. The same functionality could be achieved by using the method, but if the operation is designed to be performed very frequently, the use of block is faster compared to the use of the method. The class also has a method for loading a new page of messages and obtaining viewcontroller's instances requested by its view. There is one designated initializer – *initWithChat:*, which takes chat object. Implementation of this method is shown in Code snippet 106.

Code snippet 106. Implementation of initWithChat: method.

```
- (instancetype)initWithChat:(MVChatModel *)chat {
    if (self = [super init]) {
        _loadedPageIndex = -1;
        _sliderOffset = 0;
        _chat = chat;
        _messages = [NSMutableArray new];
        _updateSubject = [RACReplaySubject replaySubjectWithCapacity:1];
        _scheduler = [MVChatManager sharedInstance].viewModelScheduler;
        _queue = [MVChatManager sharedInstance].viewModelQueue;
        _chatId = chat.id;
        _updateSignal = [_updateSubject deliverOnMainThread];

        @weakify(self);
        self.insertMessage = ^BOOL (MVMessageModel *previous,
                                    MVMessageModel *current,
                                   MVMessageModel *next,
                                   NSMutableArray *rows,
                                   BOOL reverse) {
```

Code snippet 106 (continued).

```
            @strongify(self);
            if (previous && !previous.read && !reverse) {
                current.read = NO;
            }

            NSString *sectionKey = [self headerTitleFromMessage:current];
            NSString *previousSectionKey = [self headerTitleFromMessage:previous];
            MVMessageCellModel *viewModel = [self viewModelForMessage:current
                                                  previousMessage:previous
                                                      nextMessage:next];
            MVMessageCellModel *headerViewModel = [self
                                                 viewModelForSection:sectionKey];
            BOOL shouldInsertHeader = ![previousSectionKey
                                                    isEqualToString:sectionKey];

            NSInteger insertIndex = reverse? 0 : rows.count;

            [rows insertObject:viewModel atIndex:insertIndex];
            if (shouldInsertHeader) {
                [rows insertObject:headerViewModel atIndex:insertIndex];
            }

            return shouldInsertHeader;
        };

        [self setupAll];
        [self tryToLoadNextPage];
    }

    return self;
}
```

The initialization consists of simple filling all the properties with the initial values. The passed value of the chat object is saved so that it can be used during the lifetime of the ViewModel. Properties *queue* and *scheduler* are the queue and associated scheduler provided by the chat manager. Property *updateSubejct* is used to be a provider of events for the *updateSignal*, it is in some sense an interface between non-reactive and reactive worlds. Its semantics allows to manually send values to the subject, which are then tunneled to the corresponding signal. The subject is of class *RACReplaySubject* and is initialized with capacity 1, which means it will save the last value that it sent and re-send it to all the new subscribers when they are subscribed. This subject is used to send update values to the view, and we use here replay subject because at the time when the signal is subscribed (the view is loaded, in our case), the ViewModel potentially could be fast enough to handle messages and send an update to the subject just before the subscription. Replay subject ensures that this initial value will be received by the subscriber (the view in our case). Initialization also has initialization of the *insertMessage* block. As mentioned previously, the block is used to insert a message to the passed collection. The block expects to receive a message, which is needed to be inserted – the *current* parameter, messages that go before that and after that – the *previous* and the *next* parameters, mutable collection where to insert a message – the *rows* parameter, and a flag indicating whether this message should be inserted at the end or the beginning of the collection – the *reverse* parameter. Parameters *previous* and *next* are optional and can be *nil*. When the block is invoked, it firstly handles the *read* property of the message – all the messages that come after

unread messages should be stored in the same section according to the UI, so it tweaks the *read* property to ensure this. Then it generates cell's ViewModel for the message itself and the section header, and inserts them to the array. Viewmodel of the section header is inserted only if the currently processing message and the previous message have different section keys. The block then returns a boolean, indicating whether section header was inserted or not.

Important to note at this point that the storing of data is changed. Previously we stored two collections – an array of section keys and a dictionary that maps this section keys to the array of messages. That was the only possible implementation in terms of the MVC, but lead to some complications in handling of the section headers in the *UITableViewDataSource* methods. MVVM can provide an intermediate model – cell's ViewModel and thus we can represent both a message and a header with the same model class. That is why ViewModel does not have a dictionary of messages and array of section keys. Instead, it stores all the cell's ViewModels in the array. The logic for special handling of section headers, decreasing the index path and so on is then can be omitted. Technically, messages table view has only one section and multiple rows in that section, that are actually from multiple abstract sections.

After filling properties the initialization method transfers control to *setupAll* and *tryToLoadNextPage* methods, which finish the initialization. Afterwards instance is successfully returned from the method. Method *setupAll* is shown in Code snippet 107.

Code snippet 107. Implementation of setupAll method.

```
- (void)setupAll {
    self.title = self.chat.title;
    self.chatParticipants = self.chat.participants;
    [[MVFileManager sharedInstance] loadThumbnailAvatarForChat:self.chat
                                                      maxWidth:50
                                                    completion:^(UIImage *image) {
        self.avatar = image;
    }];

    @weakify(self);
    RAC(self, avatar) =
    [[[[MVFileManager sharedInstance] avatarUpdateSignal]
filter:^BOOL(MVAvatarUpdate *update) {
        @strongify(self);
        if (self.chat.isPeerToPeer) {
            return [update.id isEqualToString:self.chat.getPeer.id];
        } else {
            return [update.id isEqualToString:self.chat.id];
        }
    }] map:^id (MVAvatarUpdate *update) {
        return update.avatar;
    }];

    RACSignal *messageTextValid = [RACObserve(self, messageText)
                                                        map:^id (NSString *text) {
        NSCharacterSet *whitespace = [NSCharacterSet
                                          whitespaceAndNewlineCharacterSet];
        return @([text stringByTrimmingCharactersInSet:whitespace].length > 0);
    }];
```

Code snippet 107 (continued).

```
    self.sendCommand = [[RACCommand alloc] initWithEnabled:messageTextValid
                                               signalBlock:^RACSignal *(id input) {
        @strongify(self);
        return [self sendCommandSignal];
    }];

    [[[MVChatManager sharedInstance].messageUpdateSignal
        deliverOn:self.scheduler]
        subscribeNext:^(MVMessageModel *message) {
            @strongify(self);
            [self insertNewMessage:message];
        }];

    NSString *chatId = [self.chat.id copy];
    [[[MVChatManager sharedInstance].chatUpdateSignal
                                  filter:^BOOL(MVChatUpdate *chatUpdate) {
        return chatUpdate.updateType == ChatUpdateTypeModify
               && [chatUpdate.chat.id isEqualToString:chatId];
    }] subscribeNext:^(MVChatUpdate *chatUpdate) {
        @strongify(self);
        self.chat = chatUpdate.chat;
        self.title = chatUpdate.chat.title;
        self.chatParticipants = chatUpdate.chat.participants;
    }];

    [self.rac_willDeallocSignal subscribeCompleted:^{
        [[MVChatManager sharedInstance] markChatAsRead:chatId];
    }];
}
```

The method fills the rest of the properties with the data obtained from the chat object, loads avatar and setups bindings, which is the primary purpose of the method. First binding is a subscription to the avatar update signal which saves new values to the avatar property. Signal *messageTextValid* is used to validate the *messageText* property, which previously was done in the textfield's IBAction. The signal determines whether the string is valid by checking if it contains any values except whitespace characters, and sends booleans indicating this validity. The signal is then used to create a *RACCommand* object as an *enabled* option. Command *sendCommand* is going to handle tap of the send button in the view, disabling and enabling the button according to the *enabled* signal. The signal which is returned in the *signalBlock* of the command encapsulates the work which needs to be performed when the command is executed – button is pressed. Viewmodel also subscribes to the *chatUpdatesSignal* to update its data when the chat is updated, and to the *rac_willDiallocSignal* to ask the chat manager to mark the chat as read. The latter signal sends a value when ViewModel is about to be removed from the memory – in our case, it happens when the associated view is being closed.

Apart from the method *setupAll*, initialization method calls *tryToLoadNextPage* before returning. The method should be familiar from the native implementation, it is used to obtain a new messages page. Even though the same method was used in the native approach, now it has a different implementation. It is shown in the Code snippet 108.

Code snippet 108. Implementation of tryToLoadNextPage method.

```
- (void)tryToLoadNextPage {
    if (self.processingMessages) {
        return;
    }
    self.processingMessages = YES;

    BOOL shouldLoad = (!self.initialLoadComplete ||
            ([[MVChatManager sharedInstance]
            numberOfPagesInChatWithId:self.chatId]) > self.loadedPageIndex + 1);

    if (!shouldLoad) {
        self.processingMessages = NO;
        return;
    }

    @weakify(self);
    [[[[[MVChatManager sharedInstance]
                        messagesPage:++self.loadedPageIndex
                       forChatWithId:self.chatId] map:^id (NSArray *models) {
        @strongify(self);
        NSMutableArray <MVMessageCellModel *> *messages = self.messages;
        MVMessageModel *lastLoadedMessage = [messages
                                    optionalObjectAtIndex:1].message;
        MVMessageModel *beforeLastLoadedMessage = [messages
                                    optionalObjectAtIndex:2].message;
        RACTuple *firstTuple = RACTuplePack(beforeLastLoadedMessage,
                                        lastLoadedMessage);
        return [[models.rac_sequence scanWithStart:firstTuple
                             reduceWithIndex:^id (RACTuple *running,
                                                   MVMessageModel *next,
                                                   NSUInteger index) {
            if (index == 0) {
                return RACTuplePack(running.first, running.second, next, @(index));
            } else {
                return RACTuplePack(running.second, running.third, next, @(index));
            }

        }] map:^id (RACTuple *tuple) {
            return RACTuplePack(messages, tuple.first, tuple.second,
                                tuple.third, tuple.fourth, @(models.count));
        }];
    }] flattenMap:^__kindof RACSignal *(RACSequence *sequence) {
        @strongify(self);
        return [sequence signalWithScheduler:self.scheduler];
    }] subscribeNext:^(RACTuple *tuple) {
        RACTupleUnpack(NSMutableArray <MVMessageCellModel *> *rows,
                       MVMessageModel *nextModel, MVMessageModel *currentModel,
                       MVMessageModel *previousModel, NSNumber *idx,
                       NSNumber *count) = tuple;
        @strongify(self);
        if (idx.integerValue == 0 && currentModel) {
            [rows removeObjectAtIndex:0];
            [rows removeObjectAtIndex:0];
        }
        if (currentModel) {
            self.insertMessage(previousModel, currentModel, nextModel, rows, YES);
        }
        if (idx.integerValue == count.integerValue - 1) {
            self.insertMessage(nil, previousModel, currentModel, rows, YES);
        }
    }
```

Code snippet 108 (continued).

```
    completed:^{
        @strongify(self);
        self.processingMessages = NO;
        self.initialLoadComplete = YES;
        self.numberOfProcessedMessages += self.messages.count;
        MVMessagesListUpdate *update = [[MVMessagesListUpdate alloc]
                                        initWithType:MVMessagesListUpdateTypeReloadAll
                                           indexPath:nil
                                                rows:[self.messages copy]];
        [self.updateSubject sendNext:update];
    }];
}
```

As well as the previous implementation, method firstly secures its execution with the help of *processingMessages* flag. If the execution proceeds, and this is the case for the very first call, it obtains an array of messages from the chat manager. As discussed earlier the corresponding chat manager's method is changed in a way, so that it returns a signal that sends a requested value instead of calling a callback with that value. Viewmodel then applies multiple operators to that signal to handle the value and notify the view about it. The first operator is *map*, which transforms the received array of *MVMessageModel* objects into the *RACSequence* of *RACTuple* objects. The algorithm of the mapping enclosed in the *map* block firstly maps each *MVMessageModel* object into 4-spaced *RACTuple* that holds references to the current processing model, the previous model, and the next model as well as the index of the current model in the initial array. The order of objects in the tuple is following – *nextModel*, *currentModel*, *previousModel*, *index*. Then each of these *RACTuples* is mapped to the 5-spaced *RACTuple* by adding a pointer to the messages array (internal mutable collection of the ViewModels) at index 0. The process of mapping obtains a *RACSequence* obtained from the *models* array and modifies it to send described above *RACTuple* objects instead of *MVMessageModel* instances. It is done with the help of *scanWithStart:reduceWithIndex:* and *map* operators. The map operator performed on the top-level signal then returns a modified *RACSequence* instance. On the next step operator *flattenMap* is used to create a signal that is going to send each value from the passed sequence. The construction of *RACTuples* makes the process of actual data handling rather easy because it provides *MVMessageModel* objects in order that is the most convenient for the underlying algorithm as well as other relevant data.

Subscription to the resulting signal involves using *subscribeNext:completed:* operator. Here *next* block is going to be executed for every value sent by the signal and the *completed* block is only executed once after all the *next* values are processed. The first value sent by the signal stores the last message from the ViewModel's *messages* array as the *current* message. The first *if* condition in the *next* block is designed to handle this object – it removes first two ViewModels if they were presented before. We need to remove the very first section header in the messages list and the very first message model because this header can potentially move up because of the new models, and the message model can change its properties due to the same reasons. The message model and optionally its header are added back to the array in the second *if* condition. Here we use *insertMessage* block discussed previously to insert currently processed

message. Another peculiarity of the data sent by the signal is that it is not going to send the last value of the array as the *current* message – this is because we substituted it with the value of the last message in the *messages* array in the very first map operator. To handle the last object we use the last *if* condition in the *subscribeNext* block. It simply processes the last object, shifting objects of the *RACTuple* one position left. The *completed* block is executed when signal finishes sending its *next* values and completes. In this block, the ViewModel updates relevant properties and creates an update object, which is then sent to the *updateSubject* and eventually is received by the view.

The method can be seen relatively complicated due to the amount of blocks, but the underlying algorithm is simple and clear. The object, which is sent to the *updateSubject* subject is of *MVMessagesListUpdate* class. The interface of that class is shown in Code snippet 109.

Code snippet 109. Interface of MVMessagesListUpdate.

```
typedef enum : NSUInteger {
    MVMessagesListUpdateTypeReloadAll,
    MVMessagesListUpdateTypeInsertRow
} MVMessagesListUpdateType;

@interface MVMessagesListUpdate : NSObject
- (instancetype)initWithType:(MVMessagesListUpdateType)type
                   indexPath:(NSIndexPath *)indexPath
                        rows:(NSArray *)rows;
@property (assign, nonatomic) MVMessagesListUpdateType type;
@property (strong, nonatomic) NSIndexPath *indexPath;
@property (assign, nonatomic) BOOL shouldReloadPrevious;
@property (assign, nonatomic) BOOL shouldInsertHeader;
@property (strong, nonatomic) NSArray *rows;
@end
```

There are two types of updates that chat view should support – complete reload of the table view and insertion of a new row to the end of the table view. To tune the later, update *MVMessagesListUpdate* also has properties for defining its aspects – the index path of the inserted row, and flags regulating whether the previous row should be reloaded and whether the section should also be inserted. The object also encapsulates an array of cell ViewModels. This array is a snapshot of the *messages* array stored in the *MVChatViewModel* made just after this particular update. When the view receives the update, it saves this snapshot to the *rows* property of its ViewModel and this collection is then used by the table view delegate and data source methods. Such handling ensures that data remains persistent during the update of the view. Viewmodel works on the background thread and can modify *messages* array, for example, after receiving a new message, while the view is still handling the previous update on the main thread. That can cause race condition which will mostly result in the crash of the app. The mechanism of taking snapshots of modified collection and then using that snapshot only on one thread is the way to avoid that problem.
As described previously, during the initialization the *MVChatViewModel* subscribes to the *messageUpdateSignal* exposed by the chat manager. For each value passed by that

signal, it calls *insertNewMessage:* method. Its implementation is shown in Code snippet 110.

Code snippet 110. Implementation of insertMessage: method.

```
- (void)insertNewMessage:(MVMessageModel *)message {
    self.processingMessages = YES;
    [self handleNewMessage:message];
    self.numberOfProcessedMessages++;
    if (self.numberOfProcessedMessages % MVMessagesPageSize == 0) {
        self.loadedPageIndex++;
    }
    self.processingMessages = NO;
}

- (void)handleNewMessage:(MVMessageModel *)message {
    NSMutableArray <MVMessageCellModel *> *rows = self.messages;
    MVMessageCellModel *previousModel = rows.optionalLastObject;

    BOOL reloadPrevious = NO;
    if (previousModel.message && [self messageModel:previousModel.message
                        hasEqualDirectionAndTypeWith:message]) {
        MVMessageModel *beforeLastMessage = [rows
                                     optionalObjectAtIndex:rows.count - 2].message;
        MVMessageCellModel *updatedModel = [self
                                            viewModelForMessage:previousModel.message
                                                previousMessage:beforeLastMessage
                                                    nextMessage:message];
        if (updatedModel.tailType != previousModel.tailType) {
            [rows replaceObjectAtIndex:rows.count - 1 withObject:updatedModel];
            reloadPrevious = YES;
        }
    }

    BOOL insertHeader = self.insertMessage(previousModel.message,
                                           message, nil, rows, NO);
    NSIndexPath *insertPath = [NSIndexPath indexPathForRow:rows.count - 1
                                                 inSection:0];
    MVMessagesListUpdate *insert = [[MVMessagesListUpdate alloc]
                                     initWithType:MVMessagesListUpdateTypeInsertRow
                                        indexPath:insertPath
                                              rows:[rows copy]];
    insert.shouldReloadPrevious = reloadPrevious;
    insert.shouldInsertHeader = insertHeader;
    [self.updateSubject sendNext:insert];
}
```

Method *insertNewMessage:* updates *processingMessage* flag, recalculates a number of already processed messages and adjusts *loadedPageIndex* property if needed. That happens when a number of processed messages is a multiple of a page size. To process the message it calls *handleNewMessage:* method. Its algorithm is pretty straightforward – it creates a *MVMessageCellModel* instance for the inserted message, if needed creates a ViewModel for the header, inserts both of them to the messages array with the help of *insertMessage* block, and finally creates an update object for the view.

Objects of *MVMessageCellModel* class are used as a ViewModels for cells in the table view of messages. These ViewModels not only serve as intermediate data models for cells with convenient interface exposed but also take some of the functionality that

previously was done by cells and the controller. The interface of *MVMessageCellModel* can be seen in Code snippet 111.

Code snippet 111. Interface of MVMessageCellModel.

```
typedef enum : NSUInteger {
    MVMessageCellModelTypeHeader,
    MVMessageCellModelTypeSystemMessage,
    MVMessageCellModelTypeTextMessage,
    MVMessageCellModelTypeMediaMessage
} MVMessageCellModelType;

typedef enum : NSUInteger {
    MVMessageCellTailTypeDefault,
    MVMessageCellTailTypeTailess,
    MVMessageCellTailTypeFirstTailess,
    MVMessageCellTailTypeLastTailess
} MVMessageCellTailType;

typedef enum : NSUInteger {
  MVMessageCellModelDirectionIncoming,
    MVMessageCellModelDirectionOutgoing
} MVMessageCellModelDirection;

@interface MVMessageCellModel : NSObject
@property (assign, nonatomic) MVMessageCellModelType type;
@property (assign, nonatomic) MVMessageCellTailType tailType;
@property (assign, nonatomic) MVMessageCellModelDirection direction;
@property (strong, nonatomic) MVMessageModel *message;
@property (assign, nonatomic) CGFloat height;
@property (assign, nonatomic) CGFloat width;
@property (strong, nonatomic) NSString *text;
@property (strong, nonatomic) NSString *sendDateString;
@property (strong, nonatomic) UIImage *avatar;
@property (strong, nonatomic) UIImage *mediaImage;
- (void)calculateSize;
- (NSString *)cellId;
+ (CGFloat)bubbleWidthMultiplierForDirection:
(MVMessageCellModelDirection)direction;
+ (CGFloat)bubbleBottomOffsetForTailType:(MVMessageCellTailType)tailType;
+ (CGFloat)bubbleTopOffsetForTailType:(MVMessageCellTailType)tailType;
+ (CGFloat)contentOffsetForMessageType:(MVMessageCellModelType)type
tailType:(MVMessageCellTailType)tailType tailSide:(BOOL)tailSide;
+ (MVMessageCellModelDirection)directionForReuseIdentifier:(NSString *)reuseId;
+ (MVMessageCellTailType)tailTypeForReuseIdentifier:(NSString *)reuseId;
@end
```

Enumeration types that are used as types for *type*, *tailType* and *direction* properties are similar to those that were used in the cell subclasses. The only difference is that *MVMessageCellModelType* now has a value for header cells. The rest of properties include data values, in which cells are interested such as *text* or *sendDataString* and properties for storing the calculated size of the cell – *width* and *height* properties. The list of instance methods include *calculateSize* and *cellId*. The first one is used to calculate the size of the cell depending on its data and the second one is used to determine the cell's reuse identifier. Class methods are used as helpers for cells and include determination of some layout paramenters and cell types according to reuse identifiers. To calculate the size, the ViewModel use hardcoded layout rules – the same as used for drawing cells – and appropriate methods for calculating size of the content.

These methods include *NSString's boundingRectWithSize:attributes:context:* method and *AVFoundation's* function *AVMakeRectWithAspectRatioInsideRect()*.
Methods responsible for the creation of cell ViewModels that were used when processing a new page of messages or a new message are shown in Code snippet 112.

Code snippet 112. Creation of cell's ViewModel.

```
- (MVMessageCellModel *)viewModelForMessage:(MVMessageModel *)message
                            previousMessage:(MVMessageModel *)previousMessage
                                nextMessage:(MVMessageModel *)nextMessage {
    MVMessageCellModel *viewModel = [MVMessageCellModel new];
    viewModel.type = [self modelTypeForMessageType:message.type];
    if (message.type != MVMessageTypeSystem) {
        viewModel.tailType = [self messageCellTailTypeForModel:message
                                                 previousModel:previousMessage
                                                     nextModel:nextMessage];
    }
    viewModel.message = message;
    viewModel.text = message.text;
    if (message.type == MVMessageTypeMedia) {
        [message.attachment thumbnailImageWithMaxWidth:viewModel.width
                                            completion:^(UIImage *resultImage) {
            viewModel.mediaImage = resultImage;
        }];
    }
    viewModel.direction = [self
                             modeldirectionForMessageDirection:message.direction];
    viewModel.sendDateString = [NSString messageTimeFromDate:message.sendDate];
    [viewModel calculateSize];

    if (message.direction == MessageDirectionIncoming) {
        [[MVFileManager sharedInstance]
                                loadThumbnailAvatarForContact:message.contact
                                                     maxWidth:50
                                                   completion:^(UIImage *image) {
            viewModel.avatar = image;
        }];

        RAC(viewModel, avatar) =
        [[[[MVFileManager sharedInstance].avatarUpdateSignal
                                           filter:^BOOL(MVAvatarUpdate *update) {
            return (update.type == MVAvatarUpdateTypeContact
                    && [update.id isEqualToString:message.contact.id]);
        }] map:^id(MVAvatarUpdate *update) {
            return update.avatar;
        }] takeUntil:viewModel.rac_willDeallocSignal];
    }
    return viewModel;
}

- (MVMessageCellModel *)viewModelForSection:(NSString *)section {
    MVMessageCellModel *viewModel = [MVMessageCellModel new];
    viewModel.text = section;
    viewModel.type = MVMessageCellModelTypeHeader;
    [viewModel calculateSize];
    return viewModel;
}
```

Apart from the regular filling of the model with data, the method setups binding for an avatar in case of the incoming message, subscribing to the signal coming from the *MVFileManager* instance. During the creation, the ViewModel is also asked to calculate the size for the future used cell with the help of the *calculateSize* method.

Calculated size later will be accessed by the view to pass it to the table view delegate method. The calculation of cell's size at this point loads this quite massive task on the ViewModel and subsequently on the background queue, which leads to a smoother interface and better performance in general.

During initialization *MVChatViewModel* also setups a *sendCommand* with the use of the *sendCommandSignal* method. This command as described is used with the send button. The implementation of *sendCommandSignal* is shown in Code snippet 113. The method just creates a *RACSignal* passing it the block, which will be executed once the button is pressed. The button press will trigger as previously, the send of the message, and will clear the text field (*messageText* property in this case).

Code snippet 113. Creation of sendCommandSignal.

```
- (RACSignal *)sendCommandSignal {
    @weakify(self);
    return [RACSignal createSignal:^RACDisposable *(id<RACSubscriber> subscriber) {
        @strongify(self);
        [[MVChatManager sharedInstance] sendTextMessage:self.messageText
                                             toChatWithId:self.chatId];
        self.messageText = @"";
        [subscriber sendCompleted];
        return nil;
    }];
}
```

Viewmodel contains many other methods, some of which were just copied from the controller's implementation, and some of which even appear in some of the method implementations listed above. However, these methods are helpers without any regard to the ReactiveCocoa or the MVVM described in this chapter. This sums up the code of the *MVChatViewModel*, and it is time to get back to the view – the *MVChatViewController* class.

The *MVChatViewController* class has a helper method for loading from the storyboard. This method is the only one used to initialize the controller. It calls a superclass method for loading from the storyboard, creates a ViewModel and passes it to the generated instance. The implementation is shown in Code snippet 114.

Code snippet 114. Initialization of MVChatController.

```
+ (instancetype)loadFromStoryboardWithViewModel:(MVChatViewModel *)viewModel {
    MVChatViewController *instance = [super loadFromStoryboard];
    instance.viewModel = viewModel;
    return instance;
}
```

Initialization of the ViewModel at this step makes it possible to the ViewModel to process messages even before the view is loaded. However, the bindings responsible for the connection between the view and the ViewModel are still made in the *viewDidLoad* method, because most of them work with the UI elements which otherwise may be absent. Implementation of the *viewDidLoad* has all the setup previously presented in the controller's code – setting up the navigation bar, the table

view and all the others views. The only difference that now it also calls the *bindAll* method responsible for bindings. The implementation of this method is shown in Code snippet 115. It will be divided into small chunks because the whole implementation is quite long.

Code snippet 115. Setup of bindnings for handling user data.

```
RAC(self.navigationItem, title) = RACObserve(self.viewModel, title);
RAC(self.viewModel, messageText) = [self.messageTextField rac_textSignal];
@weakify(self);
[RACObserve(self.viewModel, messageText) subscribeNext:^(NSString *text) {
    @strongify(self);
    self.messageTextField.text = text;
}];

[RACObserve(self.viewModel, avatar) subscribeNext:^(UIImage *image) {
    @strongify(self);
    [self.avatarButton setImage:image forState:UIControlStateNormal];
}];
```

Firstly it setups bindings to all the visible elements used to display or input data. Each component has a corresponding data property in the ViewModel. Binding to the *messageTextField* and the *messageText* property is bidirectional – the property is updated when the textfield has a new value and vice versa, so they are entirely synchronized nevertheless the update's origin. The next step is supporting user interaction on buttons, which is shown in Code snippet 116.

Code snippet 116. Setup of bindings for handling user interaction.

```
self.sendButton.rac_command = self.viewModel.sendCommand;
[[[self.attatchButton rac_signalForControlEvents:UIControlEventTouchUpInside]
    map:^id (UIControl *value) {
        @strongify(self);
        return self.viewModel.attachmentPicker;
    }]
    subscribeNext:^(DBAttachmentPickerController *controller) {
        @strongify(self);
        [controller presentOnViewController:self];
    }];

[[[self.avatarButton rac_signalForControlEvents:UIControlEventTouchUpInside]
    map:^id (UIControl *value) {
        @strongify(self);
        return [self.viewModel relevantSettingsController];
    }]
    subscribeNext:^(UIViewController *viewController) {
        @strongify(self);
        [self.navigationController pushViewController:viewController animated:YES];
    }];
```

The *sendButton* is set up with the help of the *RACCommand*. This very short assignment regulates two aspects of the button's behavior – what happens when it is tapped, and when it should be enabled or disabled. Button *avatarButton* is handled differently – view is listening to its signal which sends values when the button is tapped, queries the ViewModel to create and configure relevant settings controller and finally pushes it onto the navigation stack. That could also be done with the help of

the *RACCommand* interface, but as soon as the view is responsible for presenting or pushing new view controllers, this implementation is more explicit.

After that, the view setups the UI-related behavior – update of view's layout to support the appearance of the keyboard and change of the table view's content size. That is shown in Code snippet 117.

Code snippet 117. Setup bindings for handling layout updates.

```
__block BOOL processingNewPage = NO;
__block BOOL autoscroll = YES;
__block NSValue *oldSize;
__block BOOL keyboardShown = NO;

[[[[NSNotificationCenter defaultCenter]
                             rac_addObserverForName:UIKeyboardWillShowNotification
                                             object:nil]
    filter:^BOOL(NSNotification *value) {
        return !keyboardShown;
    }]
    subscribeNext:^(NSNotification *x) {
        @strongify(self);
        keyboardShown = YES;
        autoscroll = NO;
        [self adjustContentOffsetDuringKeyboardAppear:YES withNotification:x];
    }];

[[[[NSNotificationCenter defaultCenter]
                              rac_addObserverForName:UIKeyboardWillHideNotification
                                              object:nil]
    filter:^BOOL(NSNotification *value) {
        return keyboardShown;
    }]
    subscribeNext:^(NSNotification *x) {
        @strongify(self);
        keyboardShown = NO;
        autoscroll = NO;
        [self adjustContentOffsetDuringKeyboardAppear:NO withNotification:x];
    }];

[[RACObserve(self.messagesTableView, contentSize) distinctUntilChanged]
    subscribeNext:^(NSValue *newSize) {
        @strongify(self);

        [UIView animateWithDuration:(!processingNewPage && autoscroll)? 0.2 : 0
                                                                 animations:^{
            [self updateContentOffsetForOldContent:oldSize.CGSizeValue
                                     andNewContent:newSize.CGSizeValue
                                 processingNewPage:processingNewPage
                                 autoScrollEnabled:autoscroll];

            [self updateContentInsetForNewContent:newSize.CGSizeValue
                                           frame:self.messagesTableView.frame.size.height];
        }];

        oldSize = newSize;
    }];
```

First two signals are used to transform *NSSNotifications* into *RACSignal* interfaces and the last one to do the same with the KVO. The actions the view performs for these events is almost the same as before, the only change is unifying of different API's.

The last step is a subscription to ViewModel's *updateSignal* which regulates when and how table view should be updated. The corresponding code is shown in Code snippet 118.

Code snippet 118. Setup of bindings related to the model data flow.

```
[self.viewModel.updateSignal subscribeNext:^(MVMessagesListUpdate *update) {
    @strongify(self);
    processingNewPage = (update.type == MVMessagesListUpdateTypeReloadAll);
    autoscroll = (self.messagesTableView.contentOffset.y >=
                 (self.messagesTableView.contentSize.height -
                  self.messagesTableView.frame.size.height - 50))
                  || !self.viewModel.rows.count;

    self.viewModel.rows = update.rows;
    if (update.type == MVMessagesListUpdateTypeReloadAll) {
        [self.messagesTableView reloadData];
    } else if (update.type == MVMessagesListUpdateTypeInsertRow) {
        NSIndexPath *previousIndexPath = [NSIndexPath
                                          indexPathForRow:update.indexPath.row-1
                                                inSection:0];
        NSArray *insertIndexPaths = @[update.indexPath];
        if (update.shouldInsertHeader) {
            insertIndexPaths = @[update.indexPath, previousIndexPath];
        }
        [self.messagesTableView insertRowsAtIndexPaths:insertIndexPaths
                                     withRowAnimation:UITableViewRowAnimationNone];
        if (update.shouldReloadPrevious) {
            [self.messagesTableView reloadRowsAtIndexPaths:@[previousIndexPath]
                                         withRowAnimation:UITableViewRowAnimationNone];
        }
    }
}];
```

The handling on the view side setups appropriately properties regulating the scrolling of the table view, copies received rows update to a persistent collection and performs needed updates on the table view.

Table view delegate's and data source's methods have almost the same implementation that they had previously. The biggest change occurred in the *heightForRowAtIndexPath:* and the *cellForRowAtIndexPath:* methods. They are shown in Code snippet 119.

Code snippet 119. Changed table view delegate and data source methods.

```
- (CGFloat)tableView:(UITableView *)tableView
                               heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    return self.viewModel.rows[indexPath.row].height;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
                                 cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    MVMessageCellModel *model = self.viewModel.rows[indexPath.row];
    UITableViewCell <MVMessageCell> *cell = [tableView
                                 dequeueReusableCellWithIdentifier:model.cellId];
    [cell fillWithModel:model];

    @weakify(self);
    [[[[cell.tapRecognizer.rac_gestureSignal
        map:^id (UIGestureRecognizer *value) {
            return cell.model;
        }]
        doNext:^(id  _Nullable x) {
            @strongify(self);
            [self.messageTextField resignFirstResponder];
        }]
        filter:^BOOL(MVMessageCellModel *model) {
            return model.type == MVMessageCellModelTypeMediaMessage;
        }]
        subscribeNext:^(MVMessageCellModel *model) {
            @strongify(self);
            MVMessageMediaCell *mediaCell = (MVMessageMediaCell *)cell;
            [self showImageViewerForMessage:model
                                  fromImageView:mediaCell.mediaImageView];
        }];

    return cell;
}
```

First one now does not have any code specific to the size calculation as well as does not support any caching. The caching is now naturally implemented by the ViewModel because cell's ViewModels store size of the cell and controller's ViewModel persistently stores cell's ViewModels. Delegate's method now simply returns a value stored in the corresponding cell's ViewModel. That is much faster than calculating the size of on the main thread by request.

Method *cellForRowAtIndexPath:* as previously, obtains a cell from the table view's reusable pool and asks it to fill itself with the model. The difference here in the handling of tap events. Previously it was done with the use of the delegate – controller conformed to a protocol and passed a weak link to itself, the cell then could notify the controller about the tap event with the help of the method defined in that protocol. Now cell exposes its tap recognizer, and the controller can handle it with the help of the *RACSignal* interface. That places cell-related code into one method making it easier to navigate through the code.

Generally speaking, *MVChatViewController* got rid of functionality related to data handling and multiple workarounds for improving the performance. This functionality is now transferred to the ViewModel, and logic behind workarounds is tightly integrated into the programming pattern that is used.

7 POSSIBLE PROBLEMS WITH REACTIVECOCOA

The use of the ReactiveCocoa introduced declarative syntax, mechanisms for unifying data streams, and abstractions over some native Objective-C components. Declarative syntax and richness of the framework not only help to build clear self-descriptive code but also introduces a language barrier for developers not familiar with the ReactiveCocoa. That can lead to a violation of the conventions, improper use of the classes and overall low quality of the app. The abstraction level of the framework is too high to make decisions regarding its use intuitively – after all, all abstract objects underneath are standard Objective-C classes and mechanisms that can show low performance if not used correctly. This problem is naturally solved if some developers in the team know the basics of the ReactiveCocoa and can point out these improper uses of the framework, share their experience and knowledge. Moreover, the framework has a comprehensive documentation in its header files, making it easier to understand how each component behaves. Community on the GitHub is very active, providing the support with any issues and storing information about known pitfalls. After all, the framework is not bigger than CocoaTouch thus making it possible for developers to pick up a technology in a reasonable time.

Even when using the framework properly according to the documentation, conventions and common sense, the performance should be taken into consideration. As described earlier, the use of any technology that builds a layer of abstraction leads to more or less lower performance. We can use code from Chapter 4 to identify possible bottlenecks and then measure the difference between similar operations in the ReactiveCocoa and pure Objective-C.

7.1 Performance of object creations

The common pattern for many iOS application is setting up a controller inside the method *viewDidLoad:*. Normally this method is called when the view controller's view is about to move onscreen – just before the *viewWillAppear:* is called. If we stop just before the call to the *viewDidLoad:* method, the user interface will still display the previous controller. The method is always called on the main thread, as well as most of the UI-related operations. That means that if we block the *viewDidLoad:* execution, the interface will become unresponsive and the user will be stuck on the previous screen. Blocking here means not a complete block of the thread (or a deadlock), but costly operations performed on the thread. Summing up, if we put something high-intensive in terms of computing in the *viewDidLoad:*, the interface will be stuck until this work is done.

In case of the ReactiveCocoa, we usually place the *RAC* macro, create new KVO signals with *RACObserve*, and subscribe to existing signals inside the *viewDidLoad:* method.

We can measure how much time each of these operations usually takes and then make a conclusion whether they are really lowering the performance or not.
To measure the performance of these events we can use GCD function *dispatch_benchmark()*, which automatically makes an iterator and schedules work. We will use 100000 as a number of iterations to make the results more precise. All the measurements are going to be performed on iPhone 6S running iOS11; the app is going to be executed in the production mode to reduce the amount of the debug code. That is not crucial where to perform tests but to make them relevant, we should run all of them on the same system.

7.1.1 RACSignal

According to measurements a creation of custom *RACSignal* using the method *createSignal:* is cheap and takes about 700 ns. There are other types os signals (internally they are represented as different classes), which initialize even faster – they are separated to provide these optimizations. For example, internal class *RACReturnSignal* is used to represent a signal which returns specified value and immediately completes. As soon as that is a common and easy to implement functionality, the class is separated from more general *RACDynamicSignal*, allowing it to be implemented in a simplistic and fast manner.

About half of time spent on the creation of a signal goes to the process of block copying – the block which is passed to the method *createSignal:* ultimately needs to be copied internally.

When working with signals, it is a common operation to modify their behavior with the use of operators. Measurements show that adding a *filter* operator can increase creation cost to 2100 ns. The same number is valid for a *map* operator. Chaining multiple operators, therefore, can dramatically increase the cost of a signal creation.

Another part of working with any signal is a subscription to the signal using *subscribeNext:* method. That costs approximately 4550 ns. The most time again is spent on block’s copying. Subscribing involves creation and handling of multiple *RACDisposable* objects internally, which also takes a significant portion of the execution time.

7.1.2 RACObserve

Setting up a *RACObserve* signal is much more costly and takes 4.8 ms. We can examine its source code to find out why this operation is relatively expensive. It turns out that the underlying mechanism for *RACObserve* – essentially performing *addObserver:* selector – is quite expensive itself, and takes about 2.3 ms to execute. That is only a

part of the answer because it is still twice less than the cost of setting up a *RACObserve*. The essence lies in the internal mechanisms inside the *RACObserve* macro. The macro is mapped to a method *rac_valuesAndChangesForKeyPath:options:observer:*. The method creates multiple signals and uses *NSRecursiveLock* for thread safety. Recursive locks in any C-like language are known to be slow. And, as we can see from the previous chapter, creating of complex signals is also relatively slow. That easily can add up 2.3 ms to the *RACObserve* creation.

7.1.3 RAC

Using *RAC* binding is also costly, but no as using *RACObserve*. Its takes about 7600 ns to create a binding from an existing signal. The most time-consuming operation beneath the macro is a subscription to a signal, which takes 4550 ns. Other 3000 ns are spent on setting up multiple *RACDisposables* and using *OSAtomicCompareAndSwapPtrBarrier* to cancel binding in case the variable becomes *null*. The actual mechanism for setting a variable value uses method *setValueForKey:*, which takes about 360 ns to execute. That is also costly compared to the regular pointer assignment that takes 30 ns.

7.1.4 RACCommand

According to measurements, a creation of *RACCommand* takes about 0.58 ms when doing 1000 iterations and about 18 ms when doing 100000 iterations. That is extremely slow, and it is clear that by increasing the number of iterations we somehow slow down the creation process. The latter is due to the fact that *RACCommand* builds a complex system of signals and uses multicast connections during initialization. Because some of these singnals have *replayLast* operator applied and because of multicast connections, initialization uses scheduling extremely. The default thread for schedulers is the main thread, and it eventually gets slowed down because of excessive use of the schedulers blocking each other, delaying execution and performing context switches. This finding means that it is not recommended to create *RACCommand* objects frequently – it is better to reuse one or substitute it with another ReactiveCocoa mechanism. We can check that by placing a *RACCommand* creation to a table view’s data source method *cellForRowAtIndexPath:*, for example, to support button tap actions. And indeed, if we do so, we will notice a significant decrease in table view’s scrolling performance, because the method is called frequently.

7.2 Speed of events propagation

Another aspect of measuring ReactiveCocoa performance is defining the speed of events propagation. We can observe how different entities of the framework behave

in the sense of the speed of reaction. Basically, this is a measurement of the amount of time passed between the send of the event and receival of the same event.

7.2.1 KVO and RACObserve

KVO is a common way to share and expose data when using ReactiveCocoa. To measure the time of KVO event propagation we firstly need to set up an observation. After that, we can take time sample just before setting the observable property, as well sample when a corresponding method or the *subscribeNext's* block is called. Measurements show that time for event propagation when using *RACObserve* is about 14300 ns. When using pure KVO, the time is reduced to 1650 ns. That shows that *RACObserve* creates an overhead that costs about ten times more than the standard handling mechanism. However, these numbers are small considering that KVO events are usually distributed across the timeline and are not happening close to each other.

7.2.2 RACSignal

As briefly mentioned before, there are multiple types of *RACSignals* and to measure performance we are focusing only on the most abstract one – *RACDynamicSignal*, which should show the worst performance possible. It is pointless to measure the time difference between method calls inside signal's block ([subscriber sendNext:]) and *subscribeNext's* block execution, because this way we are going to measure the time of a standard message sending process. Instead, we can use *RACSubject* to send events manually, and then create a *RACSignal* from that subject, subscribe to it and perform observation on its *subscribeNext's* block. That will show us how much time it takes to send a custom event through a *RACSignal*. That can be compared to a message sending process because this particular pattern was used in the previous chapter to substitute delegates.

Measurements show that single event propagation from *RACSubject* to *subscribeNext's* block takes about 2465 ns. That is nearly twice as much than using regular message sending, which takes 1183 ns. If we add the usage of method *conformsToProtocol:* to the regular message sending (which is commonly used when working with delegates), the execution time will increase to 1820 ns. That shows that using *RACSubject* and signals for general event handling is not making any significant performance decrease.

7.2.3 NSNotification and RACSignal wrapper

Another mechanism used for sending events in Objective-C is *NSNotification*. This technique could be substituted by a wrapper signal, added to the *NSNotificationCenter*. We can measure the difference between the speed of a standard *NSNotification* listening and doing the same with the *RACSignal*.

It takes 2670 ns for *NSNotification* to propagate to a default block handler, the propagation to *RACSignal's subscribeNext's* block takes 3370 ns. Although it is 1.5 times slower, the overall time is still short making *RACSignal* a suitable choice.

7.3 Implications

Measurements discussed above lead us to the conclusion that using ReactiveCocoa always creates an overhead in the sense of computing speed. However, most of the times this overhead can be neglected because of relatively small numbers – it is really not a big difference between 2000 ns and 5000 ns in the real world.

Some of the objects in the Reactive world are extremely expensive – for example, *RACCommand*. That always should be taken into consideration, when building software using the framework. Previously mentioned example of using *RACCommand* with *UITableView* is a good demonstration of using the ReactiveCocoa without an understanding of its internals. Debugging of the laggy table view, in that case, could result in a long and tedious process. Otherwise, all objects and techniques exposed by ReactiveCocoa are powerful and yet fast when used in the right context.

8 CONCLUSION

FRP allows programmers to abstract away many non-crucial details and focus on essential tasks making the programming process more abstract. The use of the FRP while developing iOS apps can make the code more clear and definitive by providing reactive interfaces and unifying event-based interfaces of Objective-C. The ReactiveCocoa framework offers needed data types and mechanisms to implement FRP while developing iOS apps. The framework has a relatively good performance when applied in the right way. However, some problems may arise if the user of the framework is not familiar with the internals of the ReactiveCocoa.

The app developed in this thesis can be used not only to demonstrate the beauty of the ReactiveCocoa but also to serve some practical use. More and more apps are taking advantage of the messenger functionality – that not only includes conventional social networks, but all sort of apps for supporting prompt customer service. If we look at the UI of the most of the messengers, we can see that they all look and behave similarly. However, the process of creating one can be quite long. The app developed throughout the thesis can serve as an external UI component, enabling developers to integrate messaging logic to their apps quickly. A brief analysis of existing libraries of that type shows that although some exist, all of them are either old (not using new iOS APIs), slow, or not much extensible (not letting the developer to customize the look and behavior in a full manner). We can overcome all these limitations of existing libraries, and provide an up-to-date customizable solution. In order to achieve that, the project should transform. We need to remove irrelevant pieces of code (contacts list, contact profile) to provide only a chat view functionality. This part of the app should be extended to give the developers an easy way to customize different aspects of the view. Even though that is a long process of trial and error, the project can serve as a backbone for that library.

REFERENCES

[1] D. Bohn, ”Mark Zuckerberg promises a native Android app, says betting on HTML5 for mobile was a 'mistake',” 11 September 2012. [Online]. Available: https://www.theverge.com/2012/9/11/3317230/mark-zuckerberg-betting-on-html5-for-mobile-was-a-mistake-hints-at. [Retrieved 21 November 2017].

[2] S. Jobs, ”Third Party Applications on the iPhone,” 17 October 2017. [Online]. Available: http://fortune.com/2007/10/17/steve-jobs-apple-will-open-iphone-to-3rd-party-apps-in-february/. [Retrieved 21 November 2017].

[3] Apple, ”Xcode Release Notes,” 31 October 2017. [Online]. Available: https://developer.apple.com/library/content/releasenotes/DeveloperTools/RN-Xcode/Chapters/Introduction.html#//apple_ref/doc/uid/TP40001051. [Retrieved 21 November 2017].

[4] Apple, ”Swift Has Reached 1.0,” 9 September 2014. [Online]. Available: https://developer.apple.com/swift/blog/?id=14. [Retrieved 21 November 2017].

[5] Apple, ”Programming with Objective-C,” 17 September 2014. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011210-CH1-SW1. [Retrieved 21 November 2017].

[6] Apple, ”Advanced Memory Management Programming Guide,” 17 July 2012. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html. [Retrieved 21 November 2017].

[7] The Clang Team, ”Objective-C Automatic Reference Counting (ARC),” [Online]. Available: http://clang.llvm.org/docs/AutomaticReferenceCounting.html. [Retrieved 21 November 2017].

[8] Apple, ”Objective-C Runtime Programming Guide,” 19 October 2009. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html. [Retrieved 21 November 2017].

[9] Apple, ”Apple Open Source,” [Online]. Available: https://opensource.apple.com. [Retrieved 21 November 2017].

[10] M. Thompson, ”Method Swizzling,” 17 February 2014. [Online]. Available: http://nshipster.com/method-swizzling/. [Retrieved 21 November 2017].

[11] C. Wheeler, ”Understanding the Objective-C Runtime,” 20 January 2010. [Online]. Available: http://cocoasamurai.blogspot.ru/2010/01/understanding-objective-c-runtime.html. [Retrieved 21 November 2017].

[12] Apple, ”UIViewController,” [Online]. Available: https://developer.apple.com/documentation/uikit/uiviewcontroller. [Retrieved 21 November 2017].

[13] Apple, ”UINavigationController,” [Online]. Available: https://developer.apple.com/documentation/uikit/uinavigationcontroller. [Retrieved 21 November 2017].

[14] Apple, 18 September 2013. [Online]. Available: https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/TableView_iPhone/AboutTableViewsiPhone/AboutTableViewsiPhone.html. [Retrieved 21 November 2017].

[15] Apple, ”Concurrency Programming Guide,” 13 December 2012. [Online]. Available: https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html. [Retrieved 21 November 2017].

[16] Apple, ”Key-Value Coding Programming Guide,” [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueCoding/index.html. [Retrieved 21 November 2017].

[17] Apple, ”Key-Value Observing Programming Guide,” 13 September 2016. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html. [Retrieved 21 November 2017].

[18] Apple, ”Xcode Help,” [Online]. Available: http://help.apple.com/xcode/mac/8.0/#/dev31645f17f. [Retrieved 21 November 2017].

[19] Apple, ”Notification Programming Topics,” 18 August 2009. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Notifications/Introduction/introNotifications.html. [Retrieved 21 November 2017].

[20] Apple, ”Concepts in Objective-C Programming,” 09 January 2012. [Online]. Available: https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html. [Retrieved 21 November 2017].

[21] M. Gallagher, ”How blocks are implemented (and the consequences),” 18 October 2009. [Online]. Available: http://www.cocoawithlove.com/2009/10/how-blocks-are-implemented-and.html. [Retrieved 21 November 2017].

[22] Apple, ”Model-View-Controller,” 21 October 2015. [Online]. Available: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. [Retrieved 21 November 2017].

[23] J. Smith, ”Patterns - WPF Apps With The Model-View-ViewModel Design Pattern,” February 2009. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx. [Retrieved 30 November 2017].

[24] C. E. a. P. Hudak, ”Functional Reactive Animation,” 1997. [Online]. Available: http://conal.net/papers/icfp97/icfp97.pdf. [Retrieved 30 November 2017].

[25] W. T. Z. W. Paul Hudak, ”Real-Time FRP,” Yale, 2001.

[26] P. Hudak, ”Conception, Evolution, and Application of Functional Programming Languages,” Yale University, Department of Computer Science, New Haven, 1989.

[27] A. C. J. P. Henrik Nilsson, ”Functional Reactive Programming, Continued,” Yale University, Department of Computer Science, Pittsburgh, 2002.

[28] M. Thompson, ”ReactiveCocoa,” NSHipster, 18 February 2013. [Online]. Available: http://nshipster.com/reactivecocoa/. [Retrieved 30 November 2017].

[29] ReactiveCocoa, ”The 2.x ReactiveCocoa Objective-C API: Streams of values over time,” ReactiveCocoa, [Online]. Available: https://github.com/ReactiveCocoa/ReactiveObjC. [Retrieved 30 November 2017].