UDC 004
DOI https://doi.org/10.32782/2663-5941/2023.3.1/31

Nikitenko Ye.V.
National University of Life and Environmental Sciences of Ukraine

Guida O.G.
V.I. Vernadsky Taurida National University

ARCHITECTURAL FRAMEWORK FOR THE FUNCTIONALIZATION OF MOBILE APPLICATION FOR IOS BASED ON REACTIVE EXTENSIONS

As the demand for services via mobile applications grows, users' expectations of the quality and speed of the application are increasing significantly. Another serious problem is the pressure from competitors. As a result, the complexity of the programs being developed, the size of the code base, and the number of different specialties and teams for one project are increasing. It also makes it harder to manage, report, and hire new staff.

The main goal of this work is to develop a framework for the iOS operating system that will provide a convenient API for developing application functions, limiting the developer in setting up communications between architectural components, focusing him on the details of implementing the functions of the business domain.

The software product under development is an architectural framework for templating the functionality of an iOS mobile application based on Reactive Extensions.

The result of the work is the ability of the framework to provide the following capabilities:

- *create multi-module applications;*
- *business logic of applications should be reusable;*
- *applications should be easy to test;*
- *application code should be easily modifiable;*
- *an application using the framework should be easy to maintain;*
- *an application using the framework should be easily extensible.*

In the future, this framework can be used in software for designers and managers, which will allow creating scenarios at the software design stage and code generators that reduce human impact on the quality of the resulting work and the speed of its execution. Such software tools have a large economic impact on the mobile development and software development industry as a whole.

Key words: *software architecture, iOS operating system, application, framework.*

Problem statement. One of the oldest ways to organize software code is to create a software architecture. A software architecture is a set of the most important decisions about the organization of software systems. It facilitates communication between several developers, allows testing individual parts of the code, reusing them, easily modifying them without undesirable impact on other systems, and adding new system modules to increase the program's functionality [1].

The problem with implementing such a solution is the possible inaccuracy of team members' understanding of the correct approach to complying with certain architectural rules. For example, one and the same architectural rule can be understood in several ways, which leads to a conflict in understanding, the choice of wrong solutions, and system disruption.

Formulation of the article's objectives. The aim of the work is to develop a framework that should include a set of tools that:

– allow you to describe scenarios for using the application functionality in a declarative way;

– automatically manage the data flow of a scenario based on its description;

– allow you to interact with the iOS Cocoa Touch Framework, considering the life cycle of its components.

Also, the framework should provide a simple, clear, and unambiguous interface for work, and facilitate the writing of understandable, testable code that complies with the SOLID principles.

Outline of the main material. Reactive Extensions or ReactiveX is a combination of the Observer, Iterator, and functional programming design pattern created by Netflix for the JavaScript programming language [2]. The implementation for the Swift programming language, which is the official programming language for the iOS platform, inherits all the original characteristics, adapting the API to a familiar form for its developers.

This programming methodology promotes application development in a declarative manner. It consists of three main ideas:

– datasourceintheformofObservable<Element>;
– subscriber in the form of Observer<Element>;
– functional operators that create new Observable<Element> based on others.

RX has many advantages over the classical imperative approach to programming. Because it is one of the implementations of the Observer pattern, the application always operates on up-to-date data. A close observation shows that this approach meets the other two great hallmarks of good design, namely unidirectionality and a single source of truth. Information is always moving away from the Observable and traveling toward the Observer. Taking into account that unidirectionality is always realized through pure functions and immutable state, we can notice these two important features inherent in this implementation. Each Observable has operators that allow you to modify the data it operates on. However, after each operator is used, it creates a new object with the modified data, which guarantees the immutability of the state. This is one of the possible implementations of the Builder design pattern. Operators usually act as pure functions, but there are some exceptions that exist to combine imperative code into reactive code.

Scenario description

Each scenario changes the system state during and after its execution, which may affect the execution of other scenarios. This is a very important note, since the system functionality does not exist completely separately from it, which, in the case of free and uncontrolled modifications of data in the storage, creates a possible violation of the DRY principle.

When executing consecutive requests in a scenario, it is possible that the data from the result of the previous request is needed for the next request, or the data that was delivered by the actor at the beginning is needed to form a non-first request, which means that within one scenario, all the necessary data must be stored in a special storage that does not change the general state of the system, but exists locally for the proper functioning of the scenario. Such a storage will be called ActionState, and the class responsible for processing the state of this storage and creating new requests will be called ActionProcessor. In Figure 1, you can see an extended data flow diagram that describes all the processes involved in managing a scenario.

Since each scenario changes the state of the system, the framework must implement an interface that explicitly declares possible changes. This is necessary to ensure that the execution of the scenario does not cause undesirable effects that are very difficult to track. Another important necessity is a special class whose responsibility is to receive the ActionState at the end of the script execution for further updating the repository. The name of this class is Repository.

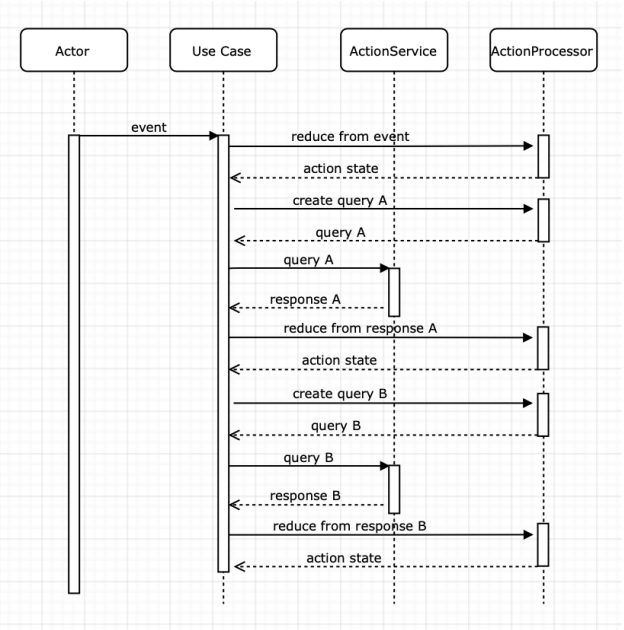

Fig. 1. Data flow diagram using the ActionProcessor

In Figure 2, you can see an example of a data flow diagram of a successful scenario that updates the state of the overall system. The state of the repository directly affects the ability to execute a particular use case. This means that every time the state changes after all requests are completed or an erroneous response is received, all scenarios that are not configured to work properly in the new state will stop receiving messages from actors. A data flow diagram that shows the checking of the current state before responding to an actor event and subsequent script execution and stopping is shown in Figures 3 and 4, respectively.

The number of use cases that can operate in parallel can be more than one, which means a common system state for each of them. Because they exist at the same moment in time and have access to the same data, the result of executing one script can change the data for the other, namely the ActionState. Changes in the scenario state should be predictable, should not lead to conflicts in the execution of any of the requests of any of the scenarios at any time. Therefore, considering all the conditions, each scenario start should begin with the formation of an initial ActionState, which is created based on the general state, which acts as the default state of the scenario and is used in the first reducer. Figure 5 shows a diagram of the data flows of a scenario with several queries.

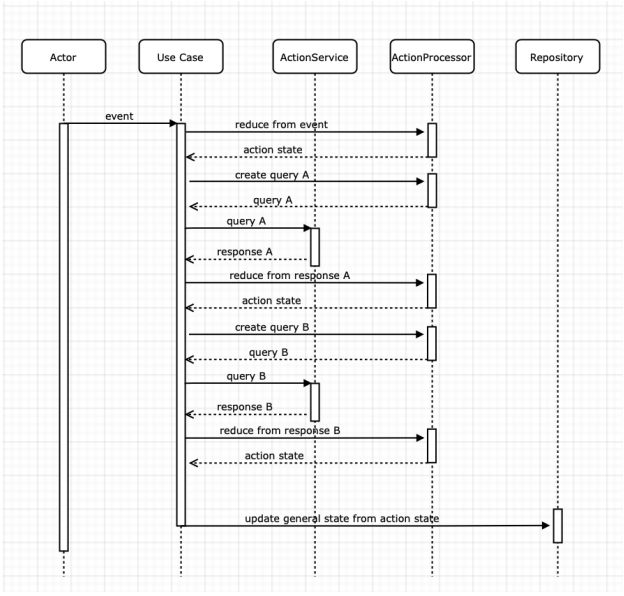

Fig. 2. Data flow diagram with status updates using Repository

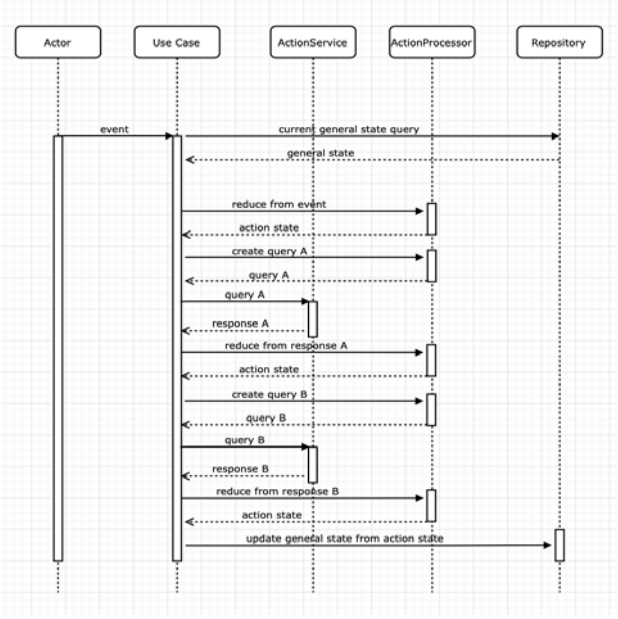

Fig. 3. Data flow diagram with checking the general state of the system and scenario execution

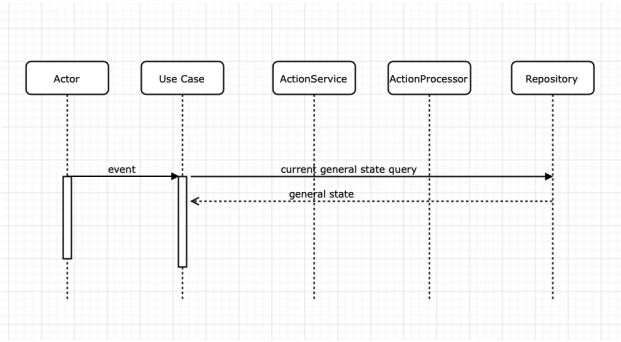

Fig. 4. Data flow diagram with checking the general state of the system and stopping the scenario

This way, each scenario can receive updated data from a common repository and use it in a timely manner.

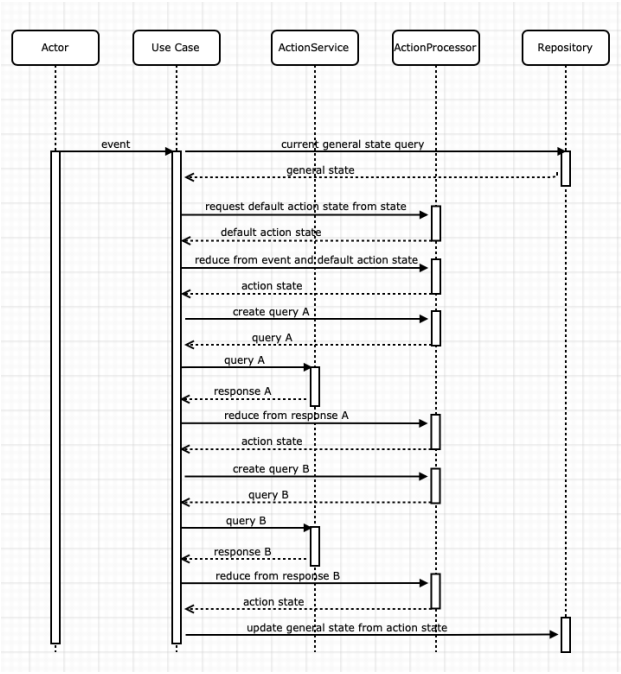

Fig. 5. Data flow diagram with the implementation of the initial ActionState based on the general state

Designing the user interface

Because components such as ActionProcessor, Repository, EffectMapper, ActionService, and Actor can be represented as protocols, it would be appropriate to describe the requirements for them in the basic implementation of the Core constructor, but such protocols will probably use generalizations that do not allow them to be used as parameters in functions due to the peculiarities of the Swift programming language. Therefore, it is necessary to create basic implementations of these protocols. Class names will have the form of the prefix “Base”, which are combined with the name of the corresponding protocol.

The Actor protocol will consist of a property that provides the ability to subscribe to a reactive message stream, and its BaseActor implementation will encapsulate the implementation of combining different streams from the external environment into a single common stream to which Core will subscribe. In order to conveniently and easily configure different threads into one, it is necessary to create a special API that will meet all the requirements, namely, have a clear and simple interface.

To do this, we will create a special method in the base class, the purpose of which is to be overridden in the inheritors and act as a provider of a special facade for configuring threads. This solution is necessary because when using methods

to configure threads, you can do it in the wrong place, which will lead to uncertainty and possible unexpected bugs when the program runs. To prevent heirs from interfering with the merge process, you need to prohibit them from overriding the events property In the Swift programming language, this is realized by using the final keyword before the field name, and in our case, **events** [3]. The Facade class will implement a special provide(events:) method that will take a message source as input and combine it with other sources in its implementation. In Figure 6, you can see a diagram of the Actor and BaseActor protocol classes.

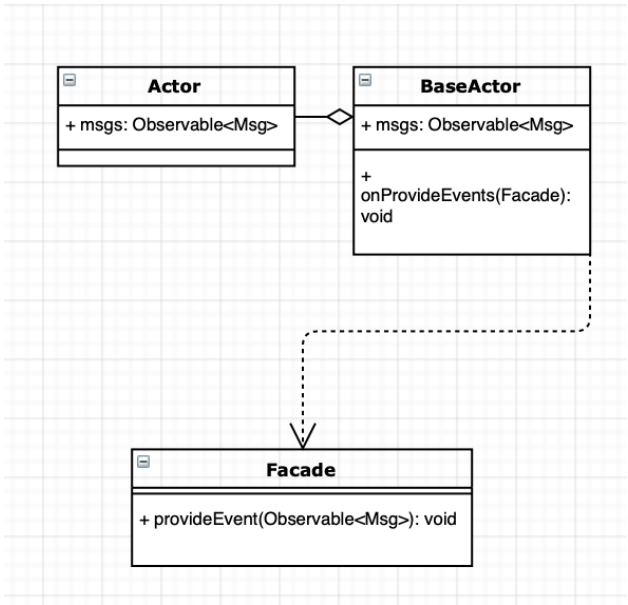

Fig. 6. Diagram of Actor and BaseActor protocol classes

This way, we designed a simple and clear interface for connecting notification sources from the external environment and encapsulated its implementation from the framework users, reducing the risk of unexpected bugs by focusing developers' attention on data sources rather than on their management.

The Repository class is responsible for updating the state from the ActionState. Since the states of the scenarios are separate for each of them, and their data can be managed with reference to the ActionState type, it is necessary to develop a convenient API for updating the general state of the repository. The user will provide the type of ActionState and the closure to store this state in a declarative manner, which allows us to shift the focus to what type of scenario state we are managing and how we update the overall state [4]. Figure 7 shows the class diagram for the Repository protocol and its basic implementation.

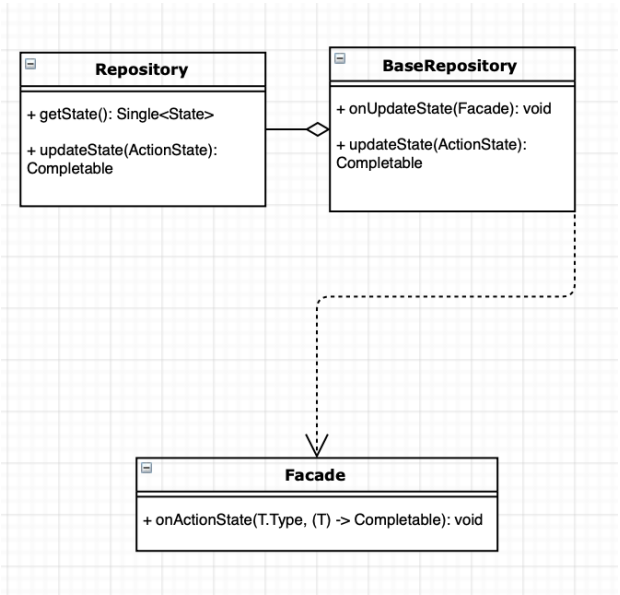

Fig. 7. Diagram of the Repository and BaseRepository protocol classes

It is worth noting that in this API we use a special reactive type – Completable. This is a special case in the implementation of the previously discussed Observable, but Completable returns either a flag indicating the state of successful completion of the thread execution or its error, compared to Observable, which can also provide a sequence of elements.

The next protocol and its corresponding class is ActionService and BaseActionService. It has only one method – execute(query:), the main purpose of which is to convert the request object into a specific call to external methods and convert their responses into a special Msg object that is configured to work in Core. However, this is not enough, since the main purpose of designing a framework is to separate the implementation from the abstract description of the behavior of program functions. It is necessary to consider the cases that are used in every application but are usually ignored: the request timeout and the number of retries in case of an error. Each request should be terminated if the result of its execution does not return long enough to be considered a failure. For this reason, the basic implementation should explicitly require the developer to specify the time to timeout. Repeated requests are not a common phenomenon in mobile software development, but in some cases it is a very important nuance that is usually not mentioned. Because of this, BaseActionService should provide the ability to specify the number of repeated requests, which will attract the attention of developers and encourage them to design functions in more detail. Another important thing is the fact that the result of the request execution can return some data or a simple flag that signals that the request was completed successfully. In RxSwift, this is implemented

through the already discussed Completable type and Single, a special Observable type that either returns an element from the stream and terminates it or terminates the stream with an error. The BaseActionService API should provide the ability to comfortably use both types without additional settings or conversions. Therefore, a method will be created that will receive the type of request, a special loop to call the proper function to the external environment, a loop to convert the successful and failed response to an Msg object, and variables for the timeout and the number of possible retries.

It is important to note that escapes for requests and response transformations must be of a special type. In Swift, escapes that can be stored in fields or called later than the function in which they are passed as a parameter completes must have the @escaping keyword. This way, the Swift compiler checks all types in advance and performs additional necessary optimizations. A diagram of the ActionService and BaseActionService classes can be seen in Figure 8.

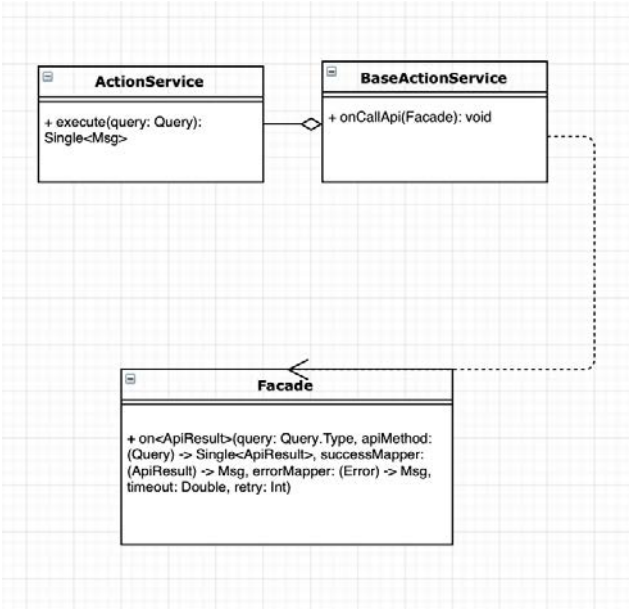

Fig. 8. Diagram of the ActionService and BaseActionService classes

The most difficult protocol and its base class to design is the ActionProcessor and its BaseActionProcessor. This protocol has three independent functions that are united by a common responsibility – working with ActionState. Because of this, the API for each method will be designed separately.

The first task of the processor is to generate the initial state of the script at the start of its execution. This method will receive the general state that will be passed to it from the Repository and the type of ActionState to be created. In its basic implementation, two levels of facades will be used. The first-level facade is a class that provides a method for specifying the type of the general state and creates the second-level facade. The second-level facade has a method that receives the type of ActionState to be created and a closure of type @escaping, which creates a scenario state object based on the previously specified general state.

The next task of the ActionProcessor is to reduce the existing scenario state after receiving a message from the actor or responding to a request. To do this, the protocol will create the reduce(currentActionState:newMsg:) method, which receives the current state of the execution scenario and the received message from the external environment. It is important to note that a message is considered to be both a classic message from an actor and the result of a scripted request. To implement this method, special facades of several levels will be created in the base class. The first-level facade provides a method that receives an ActionState type and returns an object of the second-level facade. The second-level facade has a method that receives a message type and a reduction function that generates a new state of the execution scenario. Thus, you can list all possible changes for one scenario state in a declarative style.

The last responsibility of the ActionProcessor is to create request objects, which are then passed to the ActionService. Let's describe the create(queryType:actionState) method that will receive the type of the query to be created and the corresponding ActionState. For the basic implementation, we will use the already familiar structure of a two-tiered facade. The method of the first level will receive the type of the scenario state, and the second level will receive the type of the query and the closure for its creation.

So, we designed the ActionProcessor protocol and its basic implementation in the BaseActionProcessor class. The diagram of the ActionProcessor and BaseActionProcessor classes can be seen in Figure 9.

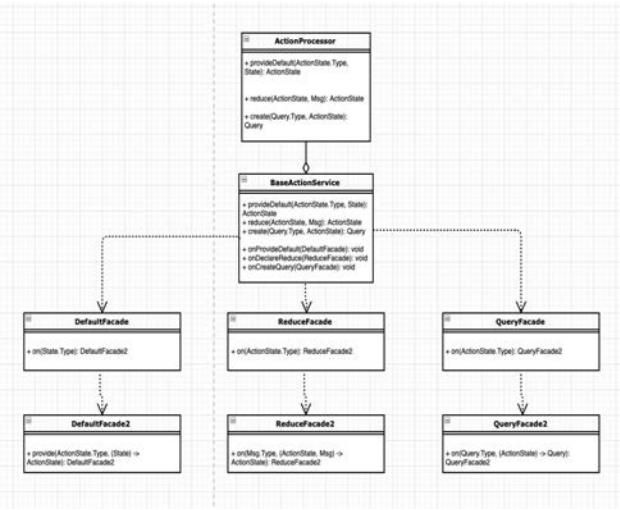

Fig. 9. Diagram of classes with ActionProcessor and BaseActionProcessor

The next entity to be designed is the EffectMapper. The main task of the EffectMapper and its BaseEffectMapper is to receive State, Msg, or Query objects and convert them to an Effect that is described by the framework user. Let's design the corresponding methods map(state:), map(msg:), and map(query:), each of which returns an Effect object. The basic implementation of the EffectMapper protocol will provide a convenient interface for declarative description of transformations. It will have an onDeclareMap method that receives a special facade where the user can configure the appropriate transformations. The facade will offer a special method for such a conversion. This method will receive the type of object to be converted as the first parameter, and the second parameter will be the closure that will be called for the conversion. Figure 10 shows a diagram with the EffectMapper and BaseEffectMapper.

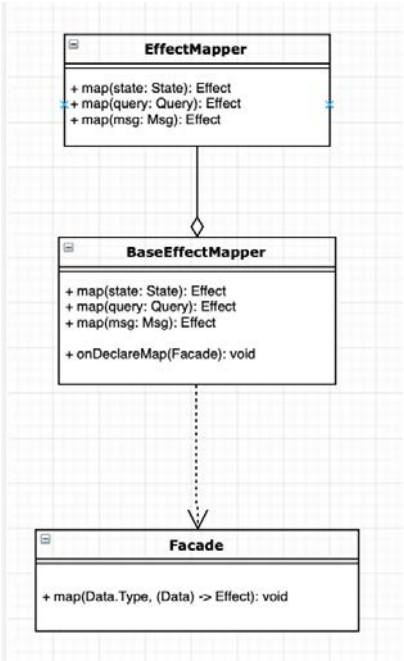

Fig. 10. Diagram of the EffectMapper and BaseEffectMapper classes

The last entity in the main module of the framework that needs to be designed is the Core. This is the most complex class compared to the others, but its API is the easiest to use due to the lack of any implementation in the inherited classes. Following the general design style of the framework, a special method was created that will receive a facade for configuring scenarios as a parameter. The beginning of the description of any list of scenarios for one general state will be the start(from:) method, which takes as input the type of the general state and returns the Starter class to describe the scenarios.

Starter provides the react(to:) method, which begins the description of a specific use case. The argument to this method is the type of message that will start the execution of the scenario, and this method returns a special Reactor class that describes possible requests and their responses.

Reactor has two methods:

– skip();
– with(call:expecting:).

The first method is called when the user explicitly ignores the possibility of executing any requests for the received message and returns an object of type SkipProcessor with a single process() method. The second method has the opposite responsibility and takes as input the type of request that will be created after receiving the message and the type of expected response. This is very important, since all other types of responses that will come from the ActionService to this request will be considered erroneous and will stop the script from executing. This method returns a special object of type QueryProcessor. It provides the ability to repeatedly call the with(call:waiting) method to list multiple requests and the process() method.

A common characteristic for both SkipProcessor and QueryProcessor is the presence of the process() method. This method completes the description of the use case and accepts the ActionState type to which it corresponds. This method returns a Starter object to allow you to declaratively describe other scenarios that belong to the same general state [5].

It is important to note that one Core can have a list of scenarios for several general states, if necessary. Sometimes one scenario can change a general state, which will make it impossible to use other scenarios for this state, but will disable the ability to run those scenarios that belong to other states and are specified in this Core. The diagram of Core classes is shown in Figure 11.

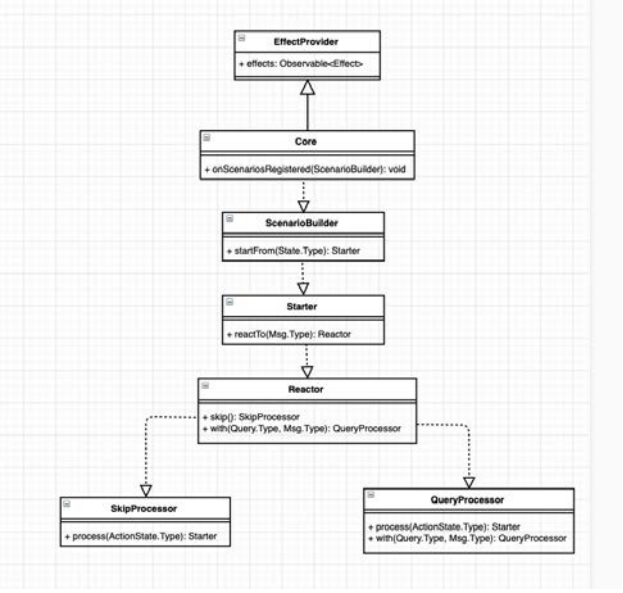

Fig. 11. Diagram of Core classes

Conclusions. We designed and implemented an architectural framework for templating the functionality of an iOS application that provides all the necessary tools for software development with a minimum number of errors in setting up data flows, requires attention to important details when developing functionality, and prevents unexpected and improper use.

We also developed an ideology of proper architecture that combines all the advantages of different patterns and focuses on designing and developing business functionality, analyzed how important the right approach to designing mobile application architecture is and the role of architectural frameworks in this.

The framework was divided into protocols and their basic implementations, each of which was displayed using UML tools. The framework was implemented using the Swift programming language in the Xcode development environment based on the RxSwift reactive framework with the Reactive Extensions methodology.

The developed framework meets the goal, templates the functionality of the iOS application with the ability to reuse it in different projects, provides a convenient API that allows you to fully describe scenarios at a high level.

In the future, this framework can be used in software for designers and managers, which will allow creating scenarios at the software design stage and code generators, which reduces the human impact on the quality of the resulting work and the speed of its execution. Such software tools have a great economic impact on the mobile development industry and software development in general.

Bibliography:

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software - Addison-Wesley, 1994. – 395 c.
2. ReactiveX: An API for asynchronous programming with observable streams. URL: http://reactivex.io – Title from the screen.
3. 5 Best Swift IDE. URL: https://www.dunebook.com/5-best-ide-swift-programming/ – Title from the screen.
4. Jacobson, Ivar; Spence, Ian; Bittner, Kurt (December 2011). "Use Case 2.0: The Guide to Succeeding with Use Cases". Ivar Jacobson International. Retrieved 5 May 2014.
5. Architectural Styles and Architectural Patterns. *Medium*. URL: https://medium.com/@mlbors/architectural-styles-and-architectural-patterns-c240f7df88a0 – Title from the screen.

Нікітенко Є.В., Гуйда О.Г. АРХІТЕКТУРНИЙ ФРЕЙМВОРК ДЛЯ ШАБЛОНІЗАЦІЇ ФУНКЦІОНАЛУ МОБІЛЬНОГО ДОДАТКА ДЛЯ iOS НА БАЗІ REACTIVE EXTENSIONS

З ростом попиту на послуги через мобільні додатки помітно зростає очікування користувачів щодо якості та швидкості роботи програми. Також іншою серйозною проблемою є тиск з боку конкурентів. В наслідок цього збільшується складність розроблюваних програм, розмір кодової бази, кількість різних спеціальностей та команд для одного проекту. Крім того, ускладнюється керування, звітування та введення нових кадрів.

Головна мета даної роботи - розробити фреймворк для операційної системи iOS, який буде надавати зручний API для розробки функцій додатку, обмежуючи розробника у налаштуванні комунікацій між архітектурними компонентами, фокусуючи його на деталях реалізації функцій предметної області бізнесу.

Розроблюваний програмний продукт – це архітектурний фреймворк для шаблонізації функціоналу мобільного додатку для iOS на базі Reactive Extensions.

Результатом роботи є можливість фреймворку надавати наступні можливості:

- *створювати додатки багатомодульними;*
- *бізнес-логіка додатків повинна мати змогу бути використаною повторно;*
- *додатки повинні легко піддаватися тестуванню;*
- *код додатків повинен мати можливість легко модифікуватися;*
- *додаток, що використовує фреймворк, повинен бути легко підтримуваним;*
- *додаток, що використовує фреймворк, повинен мати можливість легко розширюватися.*

У перспективі даний фреймворк можна використовувати у програмних забезпеченнях для дизайнерів та менеджерів, який дозволятиме створювати сценарії на етапі проектування програмного забезпечення та генераторах коду, який знижує людський вплив на якість результуючої роботи та швидкість її виконання. Такі програмні засоби мають великий економічний вплив на індустрію мобільної розробки та розробки програмного забезпечення в цілому.

Ключові слова: *архітектура програмного забезпечення, операційна система iOS, додаток, фреймворк.*