TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Dmitri Bogatenkov, 121963 IABMM

REACTIVE PROGRAMMING ON ANDROID IN AN AGILE ENVIRONMENT

Master’s thesis

Supervisor Ants Torim
Raul Liivrand

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia Teaduskond
Tarkvarateaduse instituut

Dmitri Bogatenkov, 121963 IABMM

REAKTIIVNE PROGRAMMEERIMINE ANDROIDIL AGIILSES KESKKONNAS

Magistritöö

Juhendaja	Ants Torim
	Raul Liivrand

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Dmitri Bogatenkov

05.12.2017

Acknowledgements

I would like to take this opportunity to thank those people who supported me during my studies and thesis. All the Professors and personnel of TTU for their support and assistance during the times of studies.

Firstly, I would like to gratitude my supervisors Mr. Raul Liivrand and Mr. Ants Torim for the support and guidance during the development of this work.

Secondly, I would like to thank members of the thesis defense panel for their assistance in improving this research.

Finally, I would like to thank members of my family, who had supported me throughout my studies.

Abstract

The current Master’s thesis describes the significance of using reactive programming while developing native applications for Android in an agile environment.

The purpose of this work is to demonstrate that the usage of a reactive programming paradigm for Android application development while solving concrete problems can help to improve the code quality and make the development process easier in the environment, where requirements evolve and change quickly throughout the project.

Firstly, the determination process of the most typical obstacles that Android developers overcome every day is described, and the basic requirements for an Android development in the agile environment are analyzed.

Secondly, the key concepts of reactive programming are disassembled, and different paradigms are compared.

Lastly, on the concrete project example, detected problems are combined and solutions suggested by using two different paradigms: Imperative and Reactive.

As a result, solutions to the identified problems are compared against defined criteria set, and the most efficient approach is selected.

This thesis is written in English and is 50 pages long, including 6 chapters, 10 figures, 2 tables.

Annotatsioon

Käesolev magistritöö kirjeldab reaktiivse programmeerimise olulisust Android rakenduste arendamise seisukohalt pidevalt muutuvas keskkonnas.

Magistritöö eesmärgiks on näidata, et reaktiivse programmeerimise kasutamine Android rakenduste arendamiseks võib parandada koodi kvaliteeti ja teha arendamise protsessi lihtsamaks tihti muutuvate nõuetega.

Teoreetilises osas magistritöö autor toob välja kõige kriitilisemad kohad, läbiviidud uuringu tulemuste põhjal, millega Android rakenduste arendajad igapäevaselt kokku puutuvad ning analüüsib Android rakenduste arendamise vajadusi.

Töö empiirilises osas autor võrdleb ja analüüsib reaktiivse programmeerimise erinevaid kontseptsioone, mida kasutatakse teoreetilises osas tuvastatud probleemide lahendamiseks ning selle järel toob välja tuvastatud probleemidele lahendused, kasutades selleks kahte erinevat paradigmat: Imperatiivne ja Reaktiivne.

Magistritöö viimases osas võrdleb autor probleemidele saadud lahendusi määratletud kriteeriumide kogumiga ning selgitab välja kõige tõhusamat lähenemisviisi.

Magistritöö on kirjutatud inglise keeles ja on 50 lehekülge pikk, sisaldab 6 peatükki, 10 jooniseid, 2 tabelit.

List of abbreviations and terms

RP	Reactive Programming
FP	Functional Programming
FRP	Functional Reactive Programming
RX	Reactive Extensions
PHP	Hypertext Preprocessor
UX	User Experience
UI	User interface
AHP	Analytic hierarchy process
JVM	Java Virtual Machine

List of figures

List of tables

Table of contents

1. Introduction

The majority of the development teams today are using agile techniques in their projects. Application needs have dramatically changed in the recent years. Users expect quick response times and 99.9% uptime. Software architecture rapidly becomes outdated as new demands are placed.

1.1 Background

Agile software development teams welcome changes, accepting the idea that requirements will evolve throughout a project. More often happens that the functionality of the application cannot be fully predicted at the beginning of the project. The project needs change frequently thus a streamlined and flexible approach is required for requirements change management. Systems should be more robust, more resilient, more flexible and better positioned to meet the modern demands. The code should be easily maintainable. [1]

1.2 Problem statement

Basically, on the day to day basis, developers make the code changes and add new things. Unfortunately, most of the developers are focused on what they want the program to do today thus they forgot that what the system does today is only a part of the story and tomorrow will be a new day with some new requirements. So, this is where the author believes that Reactive Programming (RP) can help software engineers to make their life easier with a possibility to continue developing at speed and in this thesis the author tests this hypothesis. RP is a general programming term that is focused on reacting to changes, such as data values or events. [2]

1.3 Purpose

Nowadays more and more people appreciate talks about Reactive programming but are still not able to find the proper place where to make an appropriate use of it in their Android projects.

This master thesis focuses on the usage of reactive programming while developing native Android applications in an agile environment.

The purpose of this thesis is to demonstrate that reactive programming paradigm is an essential part of the Android applications development. The author believes that while solving concrete problems it can help to reduce complexity of the code and make it more readable. In addition, based on the case study and the comparison against chosen criteria provide an opportunity to understand when the Reactive approach will be the most efficient and will help to improve the code quality and make the development process easier in the environment, where requirements evolve and change quickly throughout the project.

1.4 Overview

The author analyzed the necessary requirements for a native Android development in the agile environment and defined criteria set. Through the survey research among others Android developers, the author has determined the most typical problems they are experiencing every day.

On the concrete project example, the author managed to combine all detected problems that Android developers experience every day and suggested possible solutions by using two different paradigms: Imperative and Reactive.

The comparison against defined criteria set provides an opportunity to understand when the Reactive approach will be most effective and worth considering for the upcoming projects.

As a result, solutions to the identified problems are compared against defined criteria set, and the most efficient approach is selected.

2. Methodology

This chapter describes the research process and provides an outline of the methodology used to determine certain typical obstacles that Android developers experience every day while doing Android development.

2.1 Research question

The focus of the research was to measure people's opinions and judgements about using the reactive approach while dealing with an Android development in the commercial or pet projects. The research question was to identify what kind of most common challenges Android developers, with different experience level, trying to solve while doing Android applications development in the agile environment. Research objective was to specify what common challenges Android developers face with in the environment where requirements evolve and change quickly throughout the project.

2.2 Research approach

To establish several of the biggest general problems that Android developers experience every day the quantitative research method was used.

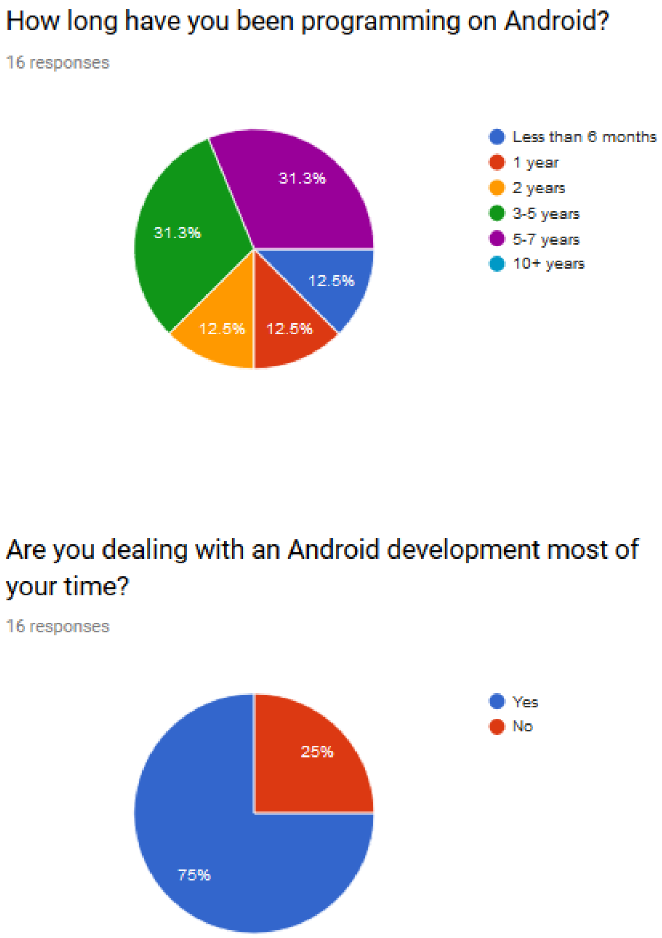
A questionnaire construction was used as a data collection technique to produce a reliable and valid result. The author tried to design questions to be good measures. Good questions are reliable thus providing consistent measures in comparable situations and questions answers correspond to what they are intended to measure. [3]

Two different types of questions were used while composing a questionnaire: free response questions and closed questions.

With the help of Google Forms information was collected, organized and survey data was analyzed and visualized.

2.3 Research process

The respondents of the study were 20 people dealing with an Android development most of their time. The distribution of respondents based on their Android programming skills is showed on the Figure 1.



How long have you been programming on Android?

16 responses

Are you dealing with an Android development most of your time?

16 responses

Figure 1. Distribution of respondents based on involvement in Android development.

The author contacted with the potential survey respondents via telephone or email. Due to the efficiency of data collection and convenience for respondents, online survey method was used. Unfortunately, only 16 respondents had a will to participate in this survey.

The online survey took place from the period between 12 September 2017 and 26 September 2017. The data has been collected and updated contemporaneously with each new respondent's answer. The survey results have been organized in the Google Forms and visualized as a chart. [Appendix 1-3]

3. Reactive Programming

3.1 Reactive and Functional Reactive programming overview

Reactive programming is a programming paradigm, oriented around data flows and the propagation of change. [4]

Reactive programming is programming with asynchronous data streams. A stream is a sequence of ongoing events ordered in time. It can emit three different things: a value of some type, an error, or a "completed" signal. These emitted events are captured asynchronously, by defining different functions that will execute when signals are emitted. The function for values must be defined, others can be omitted. The listening activity to the stream is called subscribing. The defined functions are called observers. The stream is the subject or observable being observed. This is almost the Observer Design Pattern. [4]

It extends the observer pattern to support sequences of events and adds operators that allow composing sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures. [5]

Reactive programming is based on the four guiding principles:

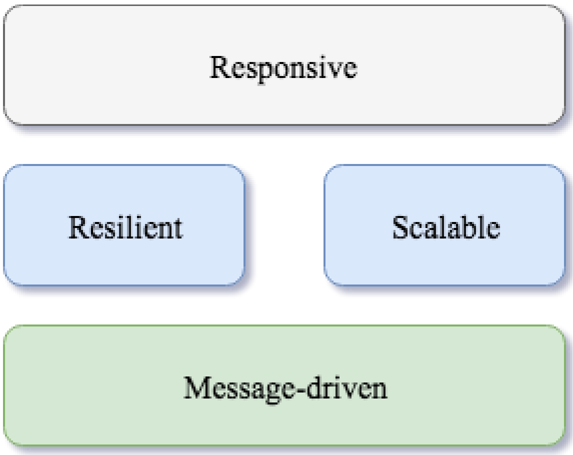


Figure 2. Principles of Reactive Programming.

The primary goal of any reactive application is to be responsive. As the application grows more complex it is still able to quickly react on the user actions thus providing consistently the better user experience.

Achieving responsiveness without both resilience and scalability is impossible. A scalable system is easily upgraded on demand to ensure responsiveness under various load conditions. A resilient system uses proper design and architecture principles to ensure responsiveness under a variety of real-world, less than ideal conditions. [6]

Scalability and resiliency are closely related while creating consistently responsive applications.

As we know the world is asynchronous thus a message-driven approach is the basis of scalable, resilient and responsive applications. A message-driven architecture provides us with an asynchronous boundary that decouples from time and space and is the basis for reactive applications. All the principles must be applied together to develop quality software in a modern context. [6]

Functional Reactive Programming combines reactive and functional programming.

Functional Programming (FP) is one of the programming paradigms, which does computation like mathematical functions without changing state and mutating data. Functional Reactive Programming is a modification of Reactive Programming that follows Functional Programming principles such as transparency and seeks to be purely functional. [4]

Based on the statements given above it is easy to form the following equation:

Imperative programming + Declarative programming with lazy evaluation = Reactive programming

Reactive programming + higher order functions and composition = Functional reactive programming

3.2 Comparisons between imperative programming and declarative programming

Typical applications are developed in an imperative style. The way where operations are ordered sequentially and based on a call stack. Applications nowadays are frequently asynchronous and imperative programming is not enough for the application logic.

Event-driven applications are focused on triggering events. Instead of components making requests when they need something, components raise events when things change. Other components then listen to events and react appropriately. [7]

Events can be encoded as a queue of messages that is observed by observers. The number of observers could be zero or more. The significant difference between event-driven and imperative style is that the caller does not block and hold onto a thread while waiting for a response. [6]

As a result, with the use of the event-driven architecture, it is simple to avoid nested callbacks problem called callback hell. Functional programming allows to follow SOLID principles thus makes code cleaner and less coupled.

3.3 Rx, RxJava and RxAndroid

The Reactive Extensions (Rx) is a library for designing event-based and asynchronous programs using observable sequences and operators, developed by Microsoft. [5]

RxJava is a library for declaratively composing event-based and asynchronous programs by using observable sequences for the Java virtual machine (JVM). [5] In the world of RxJava, everything can be represented as a stream. Each stream with some single or multiple items emitted can be consumed. For example, click events, location updates, push notifications and so on.

Main blocks of a Rx program are Observable, Observer and Operator. Observable emits values on changes. Observer or several Observers can subscribe to a single Observable to receive its emitted events and transform or modify data streams using powerful Rx Operators.

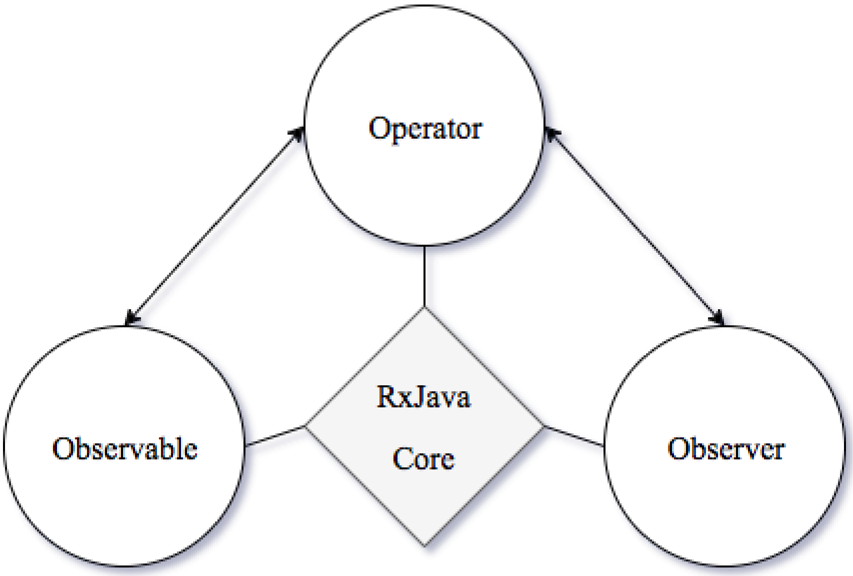


Figure 3. Basic building blocks of stream concept in Rx program.

RxJava toolset contains a big variety of built-in operators. Each operator is visualized in the documentation. The visual explanation of how certain operator works, called Marble Diagram. For example, a marble diagram of the filtering operator named “skip” is showed on the Figure 4.

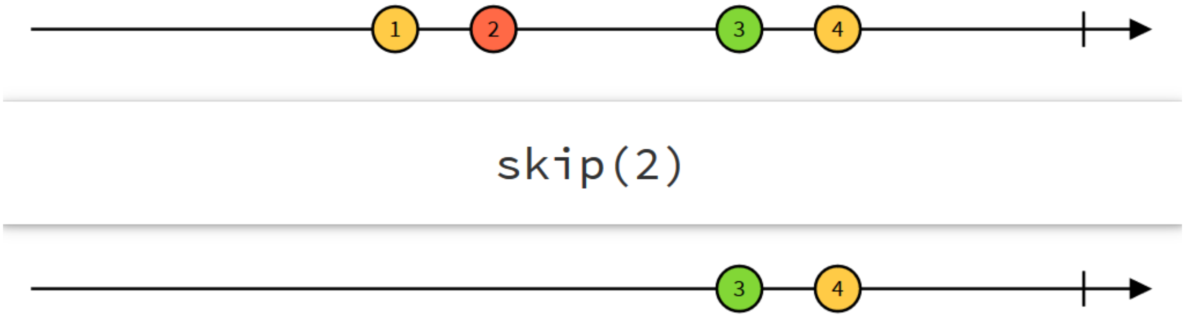
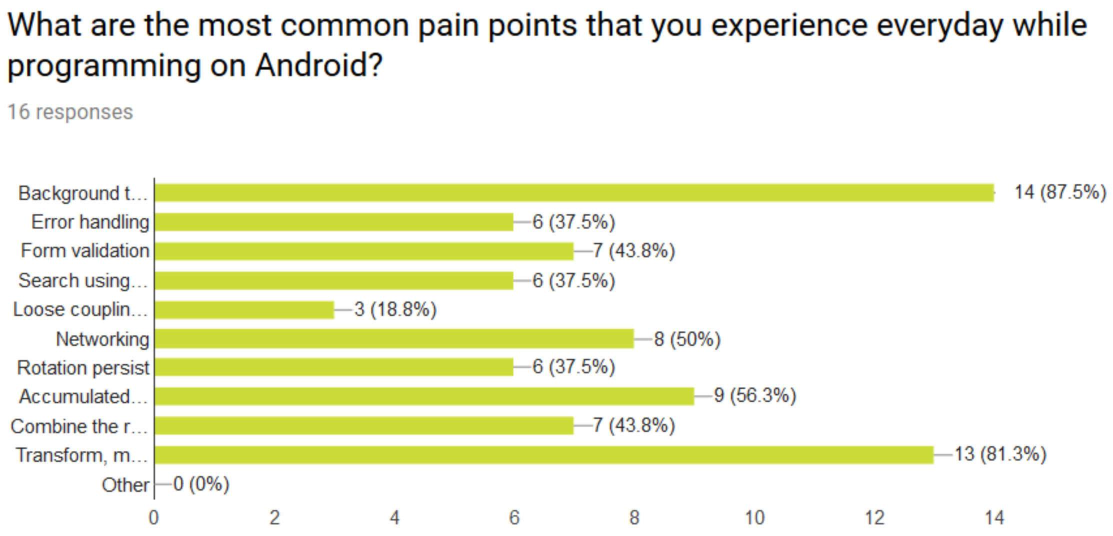


Figure 4. Marble diagram for the skip operator. [8]

RxAndroid is a Reactive Extensions for Android that consist of Android specific bindings for RxJava. This module simplifies usage of the reactive components in Android applications.

4. Problems and solutions

With the help of the survey results author figured out some of the most typical problems that Android developers solve every day.



What are the most common pain points that you experience everyday while programming on Android?

16 responses

Figure 5. Survey results of some common obstacles in Android development.

Extracted problems were prioritized based on the frequency of mention.

Table 1. Prioritized survey results of some typical obstacles in Android development.

Typical obstacles in Android development	**Frequency of mention**
Background tasks and callback-hell (nested callbacks)	87,5 %
Transform, modify and filter data	81,3 %
Accumulated calls	56,3 %
Combine the result into a single data point after making the parallel network calls	43,8 %
Form validation	43,8 %
Error handling	37,5 %

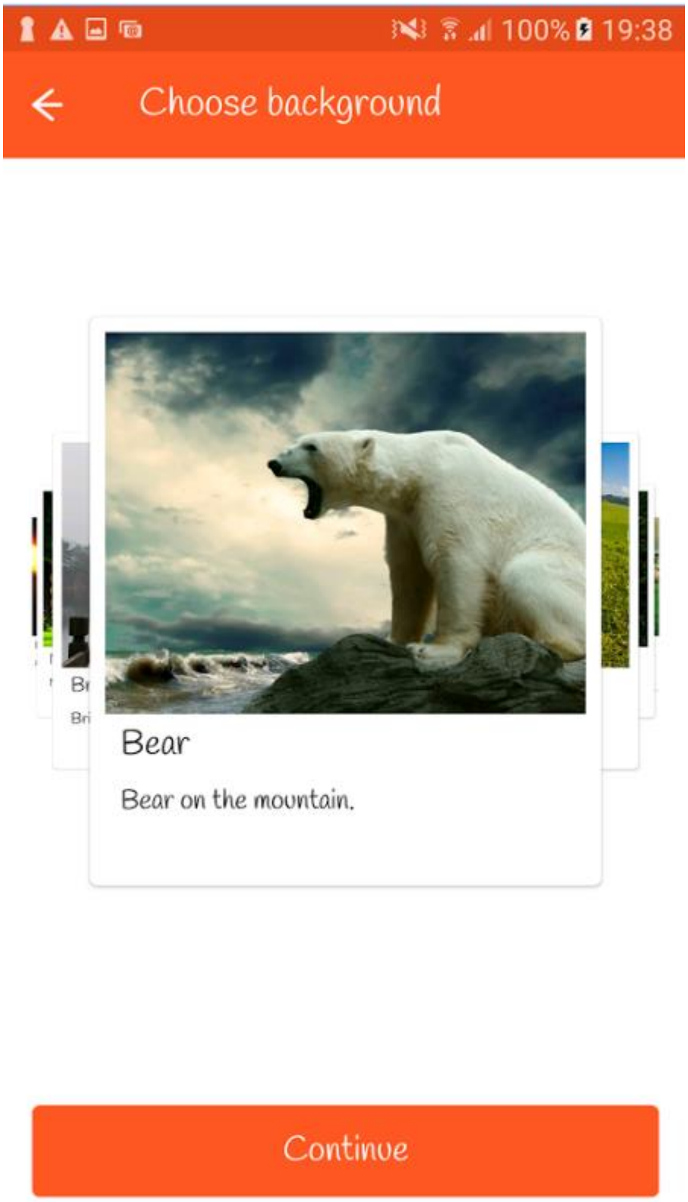
Search using autocomplete	37,5 %
Rotation persist	37,5 %
Loose coupling, isolation	18,8 %

Based on the project described below, specifically the PxMile Android application, all listed problems were analyzed and solved by comparing two different programming paradigms: Imperative and Reactive.

4.1 Project description

As an example, let's take an IoT project named PxMile Photobooth. PxMile is a contemporary green-screen photo booth solution that allows customers to make pictures with various backgrounds. Users can control the whole process with their mobile device by using the PxMile mobile application. This application lets users control PxMile photo booth - choose backgrounds, take, watch and share pictures.

To take a picture the user just need to download the application from the market, choose a suitable background, pair with a photo booth, click a couple of buttons and as a result the picture appears on the user's phone.

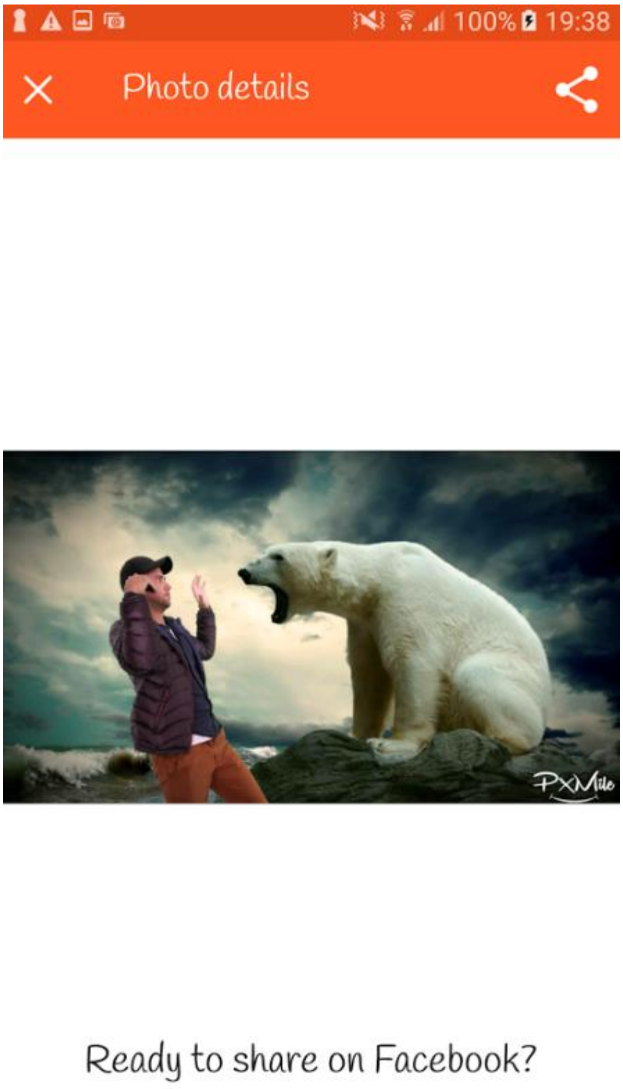


Choose background

Bear

Bear on the mountain.

Figure 6. Background selection screen.



Ready to share on Facebook?

Figure 7. Result screen.

4.2 Problem 1: Long-running background tasks

The first and the most worrying topic for all participants in the survey was the background tasks and nested callbacks problem. The question is how to efficiently execute heavy tasks on the background threads and deliver the result to the UI thread. [Appendix 1-5]

In the domain of Android, parallel execution allows processing data without freezing the UI thus responding to ongoing user interactions.

Some of the Android developers may say that this is trivial and straightforward. Everything that is needed is the background and UI thread and possibility to organize communication among different threads, for example by using AsyncTask. Unfortunately, by using it, there is a chance that implementation will be overly complicated or not all problem situations will be handled.

Based on the project described above, specifically the PxMile Android application, let's try to analyze the two following use case.

For that particular problem to compare two different approaches (Imperative and Reactive) the following criteria set was defined:

Criteria 1: Combining multiple web requests
Criteria 2: Activity/ Fragment lifecycle
Criteria 3: Caching
Criteria 4: Error handling
Criteria 5: Testability

4.2.1 Use case 1

As a user, I want to see a detailed list of the backgrounds.

To fulfill this requirement, PxMile API should be queried first to get a list of the backgrounds and then the detailed information should be requested for each background with a help of the next endpoints:

{base_url}/backgrounds – to get the list of backgrounds

`{base_url}/backgrounds/{background_id} –` to get detailed information for the specific background

Imperative approach:

The standard way to perform some simple asynchronous tasks in Android is to make use of Java’s low-level concurrency primitives. The result of the incorrect use of them are threading risks, for example race hazard or deadlock.

```
@Override
public void getDetailedBackgrounds(@NonNull final
LoadBackgroundsCallback callback) {
    Call<BackgroundsResponse> backgroundsCall =
RestService.getInstance().getPxMileApi()
      .fetchBackgrounds(PreferencesManager.getUserToken(context));
    backgroundsCall.enqueue(new Callback<BackgroundsResponse>() {
        @Override
        public void onResponse(Call<BackgroundsResponse> call,
Response<BackgroundsResponse> backgroundsResponse) {
            if (backgroundsResponse != null &&
backgroundsResponse.code() == 200) {
                final List<DetailedBackground> detailedBackgrounds =
new ArrayList<DetailedBackground>();
                for (Background background :
backgroundsResponse.body().getBackgrounds()) {
                    Call<DetailedBackgroundResponse>
detailedBackgroundCall = RestService.getInstance().getPxMileApi()
.fetchDetailedBackground(PreferencesManager.getUserToken(context),
background.getId());
                    detailedBackgroundCall.enqueue(new
Callback<DetailedBackgroundResponse>() {
                        @Override
                        public void
onResponse(Call<DetailedBackgroundResponse> call,
Response<DetailedBackgroundResponse> response) {
                            if (response != null && response.code() ==
200) {
                                detailedBackgrounds.add(response
.body().getDetailedBackground());
                                if (detailedBackgrounds.size() ==
backgroundsResponse.body().getBackgrounds().size()) {
                                    if (callback != null) {
                                        callback
.onDetailedBackgroundsLoaded(detailedBackgrounds);
                                    }
                                }
                            }
                        }
                        @Override
                        public void
onFailure(Call<DetailedBackgroundResponse> call, Throwable throwable)
{
                            Logger.e(Logger.TAG,
"getDetailedBackground onFailure " + " exception: " +
                                    Throwable
.getMessage().toString());
                        }
                    });
                }
            }
        }

        @Override
        public void onFailure(Call<BackgroundsResponse> call,
Throwable throwable) {
            Logger.e(Logger.TAG, "getDetailedBackgrounds onFailure " +
" exception: " +
                    throwable.getMessage().toString());
        }
    });
}
```

The example above shows what most of the Android developers might already be familiar with.

The first web service is called to request all available backgrounds. As a successful result there will be a list of available backgrounds. With the use of the first service callback each list item is passed to the second web service that is responsible for the background details. When all calls are done and there is an information for each item in the list, the second callback is used to update the application UI. As a result, we have to manage nested callbacks.

Android assures that this code will not execute in the main user-interface thread.

Though it is not the worst code, but adding more web service calls will increase illegibility. Each following call will be dependent of the previous one and will add levels of callbacks and the code complexity. Thus, it occurrence is known as callback hell. For example, a developer would like to improve the application performance and make a few web-service calls in parallel. For that, the results should be merged and returned back to the UI. There is no common solution for that problem. One of the ways to do that is to create a custom executor and to coordinate parallel threads.

Also, unfortunately, there is no ready-made solution for the situation where something can go wrong. There is a solution to surround code with a try/catch block. It helps, but developers have to pay a proper attention and correctly handle all exceptions based on the project needs. Custom solutions are not consistent and predictable for new coming developers with a bunch of extra code.

If developers would like to unit test their code it will be really difficult and as a result hard to maintain in the future. So definitely third-party library should be used.

Reactive approach:

```
public void rxFetchDetailedBackgrounds(@NonNull final
LoadBackgroundsCallback callback) {
    RestService.getInstance()
        .getPxMileApi().fetchBackgroundsRx()
        .concatMap(Observable::from)
        .concatMap((Background background) ->
            RestService.getInstance()
                .getPxMileApi()
                .rxFetchBackgroundDetails(background.getId()))
        .toList()
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(detailedBackgrounds-> {
    if (callback != null) {
        callback.onDetailedBackgroundsLoaded(detailedBackgrounds);
    }
    });
}
```

Fortunately, all the issues discussed above have an elegant solution with RxJava and RxAndroid library. The reactive approach helps to avoid accumulating callbacks or “callback hell” and make the code more clear and legible. To merge the final results `concatMap` Rx Operator is used.

By using Observables, no additional work needed. Both, error and success cases are already handled by default in a concise way. As an additional feature, it is possible to specify where the results will be handled. For example, the results of the request could be processed on the main thread.

Testing is clear, maintainable and straightforward with a reactive approach. By using `toBlocking()` method, any method could be turned from asynchronous to synchronous one. Thus, there is no need to fragile things by using sleep methods.

4.2.2 Use case 2

As a user, I want to view the taken photo when it will be fully processed on the server-side and uploaded to an Amazon S3 bucket.

To fulfill this requirement, PxMile API should be queried first to get the current photo status.

If the photo status equals "0" let's repeat the first call after a specific timeout, in the other case the photo is successfully uploaded and the image should be requested.

This will involve two PxMile web service endpoints and one Amazon S3 bucket endpoint:

`{base_url}/users/{user_id}/photos/{code}/status-` retrieve the photo status
`{base_url}/users/{user_id}/photos/{code} -` retrieve the photo hash
`{bucket}.s3.amazonaws.com/images/{hash}.jpg -` retrieve the photo

To fulfill the user needs based on the described use case, multiple web service calls should be composed. Let's omit the nested callbacks problem as it was discussed above and focus on the long-running task – photo download process.

Imperative approach:

The most of the Android developers use AsyncTask class to perform long-running background operations. AsyncTask must be subclassed and should override at least one method `(doInBackground(Params...))`. Let's ascertain what happens if the user presses back button finish Activity or change the device orientation while having long-running task.

If nothing additional is added to prevent that situation the application will crash. The NullPointerException will be fired, because the Activity is not accessible anymore. To avoid this crash the task have to be referred. Another way is to use `isFinishing()` method, to check the Activity state and cancel the task.

Thus, it is really hard to get everything right because there is no any proper approach provided and solutions that are used, differ from project to project

Reactive approach:

In the world of the reactive programming the lifecycle and the memory leaking problems are solved by the usage of subscriptions.

A common pattern is to use a `CompositeSubscription`. It helps to hold all Subscriptions and unsubscribe all at once at ease in `onDestroy()` or `onDestroyView()` methods.

```
private CompositeSubscription compositeSub = new
CompositeSubscription();

private void initAutocompleteField() {
    AutocompleteService autocompleteService = new
AutocompleteService();
    compositeSub.add(autocompleteService
            .autocomplete(startLocation, locationService)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(updateAutocompleteList()));
}

@Override
public void onDestroyView() {
    compositeSub.unsubscribe();
    super.onDestroyView();
}
```

The context leaking is also prevented due to the unsubscription.

There is also another way to resolve that problem, by using Trello's RxLifecycle lightweight library.

Mentioned library allows for automatic completion of sequences based on Activity or Fragment lifecycle events. It determines the appropriate time to end the sequence.

```
myObservable.compose(RxLifecycleAndroid.bindActivity(lifecycle))
.subscribe();
```

4.3 Problem 2: Responding to user interaction

One of the key elements of the most Android applications is the user interface and a reaction to the user interactions. The majority of Android developers solve issues concerning UI in the traditional way, but the reactive approach could be also applied.

Criteria 1: Reuse individual pieces
Criteria 2: The code length and readability
Criteria 3: Separation of view and model logic
Criteria 4: Memory leaks

4.3.1 Use case 1

As a user, I want to see the dropdown list with countries suggestions when I stop typing without pressing any button.

To fulfill this requirement, the AutoCompleteTextView could be used and auto-complete action should be triggered when the user stops typing. The problem is how to get known when to trigger it.

As user type the word "Estonia", developer does not want to execute searches for E, Es, Est ... etc. But rather wait for a couple of seconds, make sure the user has finished typing the whole word, and then make a call.

Imperative approach:

One of the solutions is to use `onFocusChangeListener` to observe when the user begins editing text in this text field and when ends. Another possible solution is to start timer with a delay. With any text change timer restarts and waits for the next change in the text field. If there is no changes the required action should be triggered. It will be really hard to cut the complete solution down to about 30 lines of code. There is also a risk of the potential memory leaks.

Reactive approach:

RxJava/RxAndroid library could be used not just for background operations. It is not yet fully featured, but is already really useful for making a responsive UI. It also helps to separate view and model logic.

With RxJava, Subject can be used to automatically update the UI. Subjects are observables that can both subscribe to and trigger updates on. With Subjects a reference is not required to an observer, just emit the data on the subject itself. To optimize event handlers debounce and throttle methods could be used.

```
public Observable<List<SuggestedLocation>>
autocomplete(CustomAutocompleteTextView view, final LocationService
locationService) {
    return getInputTextObservable(view)
    .debounce(DEBOUNCE_TIMEOUT_IN_MILLISECONDS, TimeUnit.MILLISECONDS)
    .switchMap(new Func1<String,
Observable<List<SuggestedLocation>>>() {
        @Override
        public Observable<List<SuggestedLocation>> call(final String
query) {
            return fetchSuggestedLocations(LOCALE, query,
locationService.getCurrentLocation())
    .timeout(REQUEST_TIMEOUT_IN_SECONDS, TimeUnit.SECONDS)
    .retry(RETRY_COUNT_FOR_REQUEST);
        }
    });
}
```

As an addition the throttle method could be also applied to avoid button multiple clicks.

```
RxView.clicks(view).throttleFirst(500,
TimeUnit.MILLISECONDS).subscribe(empty -> { // action on click });
```

4.4 Problem 3: Isolation, complex list filtering and data transformation

Based on the survey results another significant topic for all participants in the survey was the data transformation. The question is how to make it easier to convert, filter, combine and transform data with no pain, fewer lines of code and the ability to reuse individual pieces.

For that particular problem to compare two different approaches (Imperative and Reactive) the following criteria set was defined:

Criteria 1: Code length
Criteria 2: Reusability
Criteria 3: Complexity

4.4.1 Use case 1

As a user, I want to filter out all black-and-white backgrounds and display background names in the lowercase.

Imperative approach:

```
ArrayList<String> tempBackgroundNames = new ArrayList<>();
ArrayList<String> filteredBackgroundNames = new ArrayList<>()
ArrayList<String> backgroundNames = new ArrayList<>();

for (String backgroundName : tempBackgroundNames) {
    if (backgroundName.startsWith("BW")) {
        filteredBackgroundNames.add(backgroundName);
    }
}
for (String backgroundName : filteredBackgroundNames) {
    backgroundNames.add(backgroundName.toLowerCase());
}
```

Reactive approach:

The great power of the reactive approach lies in the operators. They allow manipulating, transforming, filtering and combining objects emitted by the Observables. There is a wide range of operators that could make the developers life easier. Multiple operators could be used at once. The complete list of Rx operators is available in the official document.

Applying an operator to Observable returns a new Observable, leaving the original one untouched. Thus, is it easy to isolate and decompose the tasks.

```
Observable<String> backgroundNamesObservable =
Observable.from(backgroundNames)
     .flatMap(new Func1<String, Observable<String>>() {
         @Override
         public Observable<String> call(String background) {
             return Observable.from(background.toUpperCase());
         }
     })
     .filter(new Func1<Observable<String>, Boolean>() {
         @Override
         public Boolean call(String background) {
             return background.startsWith("BW");
         }
     });
```

In the suggested solution the `filter()` operator is used, which takes a predicate and either passes events further or discards them.

For the better understanding how `filter()` operator works let's have a look on the visual explanation of how it, by the help of so-called marble diagram:

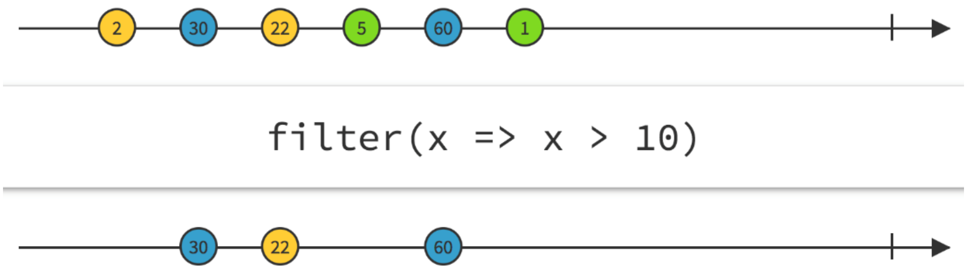


Figure 8. Marble diagram for the filter operator. [8]

4.4.2 Use case 2

As a user, I want registration form to be validated when all input fields are filled.

The best practice is to make sure that all fields are filled before submitting the form. The question is how to make sure that all form fields are filled.

Imperative approach:

There is at least two possibilities. The first one is to listen to changes made within field and fire up the field validation. The second one is to listen to the submit-button click and start validatin all fields in the from on submit.

Each form input field should be registered and `textChangeListener` added.

```
EditText emailField = (EditText) findViewById(R.id.edit_text_email);
emailField.addTextChangedListener(new TextWatcher() {
            public void onTextChanged(CharSequence charSeq, int start,
int before, int count) {}
            // with each change, check if email is valid
            public void afterTextChanged(Editable emailText) {
                isEmailAddress(emailText.toString(), true);
            }
            public void beforeTextChanged(CharSequence charSeq, int
start, int count, int after) {}
});
```

`TextWatcher` makes it possible to validate field ad-hoc.

After the form submission the validation class will check for any errors and display them if needed on the respective fields.

Handling it input fields with a bunch of booleans makes the code cluttered and kind of difficult to follow.

Reactive approach:

Traditionally, validation methods return a boolean that tells us if it succeeded or not, so we started with that.

However, using combineLatest Rx Operator developer could monitor the state of multiple observables at once compactly at a single location and as a result has a small block of code.

```
Observable.combineLatest(emailObservable, countryObservable
(emailValid, countryValid) -> emailValid && countryValid)
.distinctUntilChanged()
.subscribe(valid -> submitButton.setEnabled(valid));
```

Button state, enabled or disabled depends on the combined single emitted value and changes relatively. Rx Operator combineLatest emits value only when all Observables have at least one value that is emitted.

This technique becomes more apparent when there is more than one input field in a form.

4.5 Evaluation criteria

The author analyzed top three of the most worrying problems, while doing Android development, for all participants in the survey. Firstly, long-running background tasks. Secondly, immediate and continuous user interaction and finally, complex list filtering with data transformation.

Each obstacle was analyzed based on the different suite of use cases. For each problem specific criteria set was defined to compare imperative and reactive approaches.

Table 2. Criteria set for each defined problem.

Criteria	**Problems**
Combining multiple web requests	Long-running background tasks
Activity/ Fragment lifecycle	Long-running background tasks
Caching	Long-running background tasks
Error handling	Long-running background tasks

Testability	Long-running background tasks
Reuse of individual pieces	Responding to user interaction, Isolation, complex list filtering and data transformation
The code length and readability	Responding to user interaction, Isolation, complex list filtering and data transformation
Separation of view and model logic	Responding to user interaction
Memory leaks	Responding to user interaction
Complexity	Isolation, complex list filtering and data transformation

5. Analysis and evaluation

The author analyzed the requirements of an Agile environment and identified basic criteria.

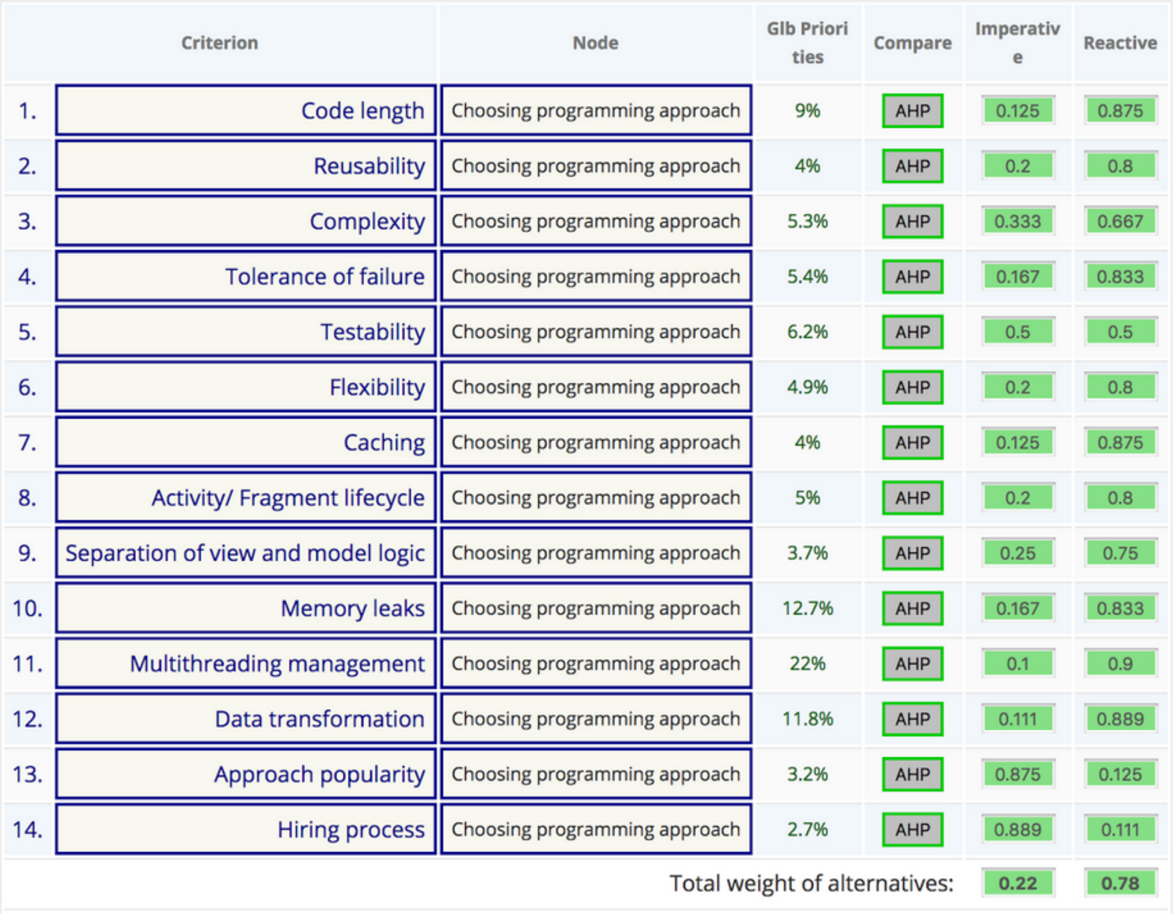
The factors to be considered are the key factors of the fast-paced work environment: tolerance of failure, capability, reusability, testability and flexibility.

Based on the survey results author figured out some typical problems that Android developers experience every day. Extracted problems were prioritized based on the frequency of mention.

Based on the identified criteria provided in the Table 2 and the list of the key factors most common problems were solved by comparing two different programming paradigms: Imperative and Reactive.

The goal was to demonstrate that Reactive programming paradigm from a field of two main paradigms: Imperative and Reactive is the most suitable while solving several of the biggest general problems that Android developers experience every day that were discovered with the help of the survey research method.

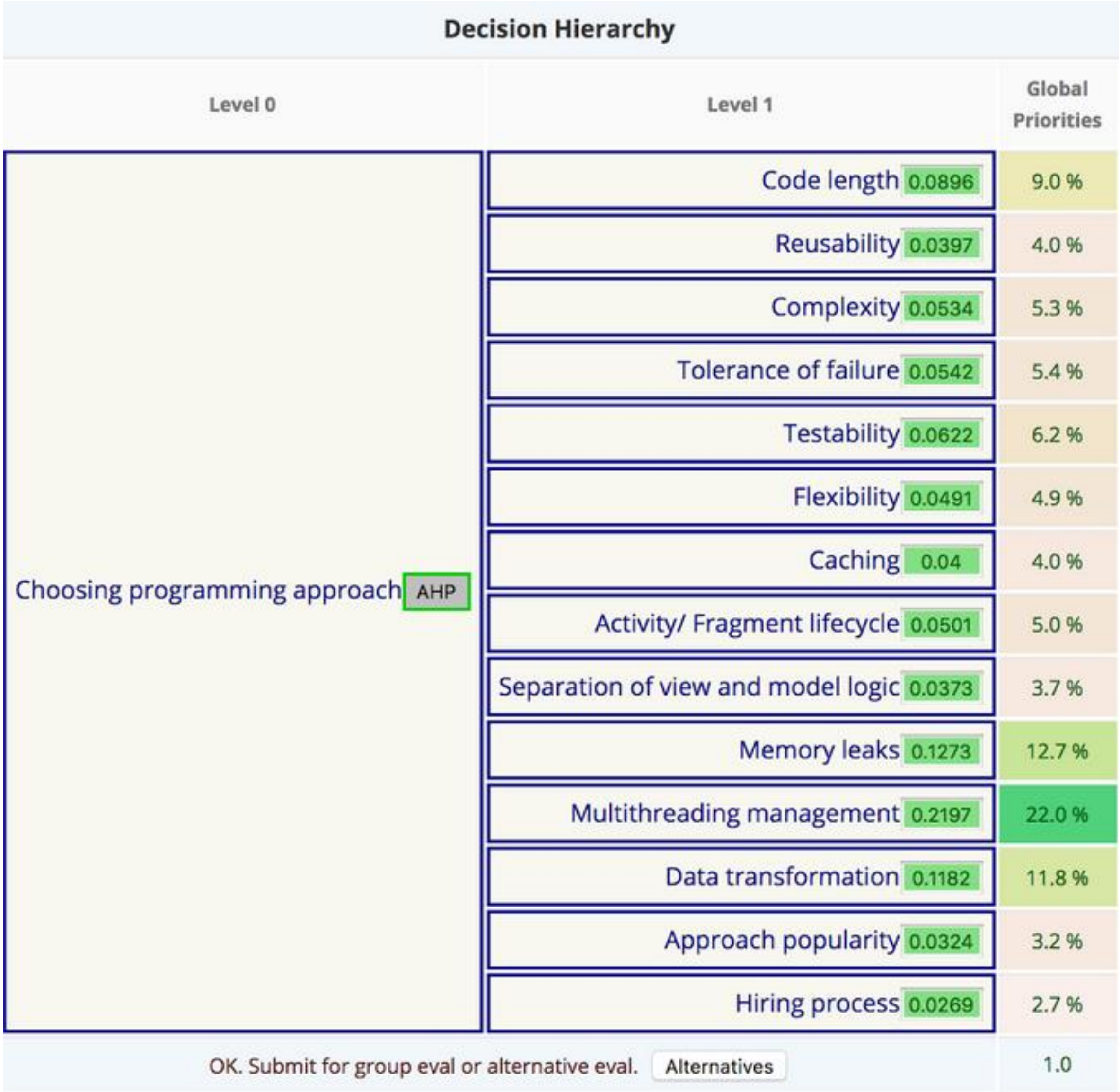
For analyzing a complex decision, the analytic hierarchy process was used. AHP provided a rational framework for structuring a decision problem and for evaluating alternative solutions. The author analyzed independently every sub-problem in the hierarchy.



Criterion		Node	Glb Priorities	Compare	Imperative	Reactive
1.	Code length	Choosing programming approach	9%	AHP	0.125	0.875
2.	Reusability	Choosing programming approach	4%	AHP	0.2	0.8
3.	Complexity	Choosing programming approach	5.3%	AHP	0.333	0.667
4.	Tolerance of failure	Choosing programming approach	5.4%	AHP	0.167	0.833
5.	Testability	Choosing programming approach	6.2%	AHP	0.5	0.5
6.	Flexibility	Choosing programming approach	4.9%	AHP	0.2	0.8
7.	Caching	Choosing programming approach	4%	AHP	0.125	0.875
8.	Activity/ Fragment lifecycle	Choosing programming approach	5%	AHP	0.2	0.8
9.	Separation of view and model logic	Choosing programming approach	3.7%	AHP	0.25	0.75
10.	Memory leaks	Choosing programming approach	12.7%	AHP	0.167	0.833
11.	Multithreading management	Choosing programming approach	22%	AHP	0.1	0.9
12.	Data transformation	Choosing programming approach	11.8%	AHP	0.111	0.889
13.	Approach popularity	Choosing programming approach	3.2%	AHP	0.875	0.125
14.	Hiring process	Choosing programming approach	2.7%	AHP	0.889	0.111
				Total weight of alternatives:	**0.22**	**0.78**

Figure 9. AHP structure for the evaluation of alternatives based on different criteria.

With the usage of analytic hierarchy process, alternative solutions were evaluated. The author created a decision hierarchy based on the defined criteria that could be helpful for other developers to select the most suitable programming approach.



Decision Hierarchy

Level 0	Level 1	Global Priorities
Choosing programming approach AHP	Code length 0.0896	9.0 %
	Reusability 0.0397	4.0 %
	Complexity 0.0534	5.3 %
	Tolerance of failure 0.0542	5.4 %
	Testability 0.0622	6.2 %
	Flexibility 0.0491	4.9 %
	Caching 0.04	4.0 %
	Activity/ Fragment lifecycle 0.0501	5.0 %
	Separation of view and model logic 0.0373	3.7 %
	Memory leaks 0.1273	12.7 %
	Multithreading management 0.2197	22.0 %
	Data transformation 0.1182	11.8 %
	Approach popularity 0.0324	3.2 %
	Hiring process 0.0269	2.7 %
OK. Submit for group eval or alternative eval. Alternatives		1.0

Figure 10. Decision table on choosing the most suitable programming approach.

Current Master's thesis objectives were:

- To highlight the significance of Reactive programming while developing native applications for Android in an agile environment.
- To demonstrate how or why to make use of the Reactive approach.
- To verify on the concrete project example that problem solutions with the usage of Reactive programming can help to improve the code quality and make the development process easier.

The following actions were taken to accomplish the purpose:

- The most typical problems that Android developers experience every day were determined with the help of survey that was conducted to get a better overview.
- The key concepts of Reactive programming were disassembled, and different paradigms were compared.
- On the concrete project example, the detected problems were prioritized based on the frequency of mention and combined by criteria.
- By using two different paradigms, Imperative and Reactive, possible solutions were suggested and compared against defined criteria.

The first and the most worrying topic for all participants in the survey was the long-running background tasks and nested callbacks problem. The author tried to efficiently execute heavy tasks on the background threads and deliver the result to the UI thread. Two use cases were analyzed. At first thought, solutions based on the imperative approach seemed to be trivial and straightforward. However, the comparison with the reactive approach revealed that not all the edge cases were handled.

It also proved that without using of RxJava/ RxAndroid libraries each following call will be dependent of the previous one and will add levels of callbacks and the code complexity. On the dropdown list example, the author also showed how the reactive approach could be applied while solving issues concerning the user interface and a reaction to the user interactions. As a result, the reactive approach also helped to separate view and model logic, optimize event handlers and prevent memory leaks.

The author was also able to demonstrate the power of the Rx operators while validating forms that allow manipulating, transforming and combining objects issued by the Observables.

RxJava and RxAndroid libraries are still rather new. Even today many developers are still figuring out why to use a Reactive approach and how it could help to overcome some typical problems while developing applications for Android.

6. Summary

In the world of software systems, the four elements are fulfilled to tag it as reactive:

- React to events – event driven / message driven
- React to system load – scalable
- React to failure – resilient and robust
- React to user – responsive

Based on the survey results author demonstrated the most typical problems that Android developers experience every day. Extracted problems were prioritized based on the frequency of mention. Based on the concrete project example, the author found some disadvantages of using Imperative approach while solving extracted problems:

- Multithreading management is very basic.
- No easy way to synchronize nested asynchronous calls.
- Callbacks are not universal and reuse of methods is almost impossible.
- Difficult to transform data.

With the usage of AHP, alternative solutions were evaluated. The author created a decision hierarchy table based on the chosen criteria set that could be helpful for other developers to select the most suitable programming approach for the Android development.

From the decision hierarchy it is possible to conclude, that using the Reactive approach, helped to improve the code quality and made the development process easier. The author also found that the biggest advantage of reactive programming is that it lets write clean, concise and readable code. It is easy to focus on solving the problem rather than the required procedure with a reactive approach. Individual pieces could be reused, multithreading management is simplified, and operators make data transformation easier.

As a result of this thesis the author demonstrated on the real-life project with some real cases that the Reactive programming can help to solve some most common cases with a no pain, less lines of code and easy refactoring. As a result, it will be a much smaller program that will consist of isolated components, will be loosely-coupled, flexible and will stay responsive in the face of failure. It offers some fresh expectations on solving the recent programming problems and makes developers think differently and in most cases using it can help to improve the application speed.

The author hopes that this thesis has provided an opportunity to understand when the Reactive approach could be the most efficient, help to improve the code quality and could make the development process easier in the environment, where requirements evolve and change quickly throughout the project.

Kokkuvõte

Antud magistritöö eesmärgiks oli:

- Rõhutada reaktiivse programmeerimise kasutamise olulisust Android rakenduste arendamisel pidevalt muutuvas keskkonnas.
- Näidata kuidas ja miks oleks mõistlik kasutada Reaktiivset lähenemist.
- Konkreetse projekti näitel kontrollida, et reaktiivse programmeerimise kasutamine probleemide lahendamiseks parandab koodi kvaliteeti ning teeb arendamise protsessi lihtsamaks.

Eesmärgi täitmiseks töö autor tegi järgmised sammud:

Küsitluse abil olid välja selgitatud kõige teravamad kohad, millega Android rakenduste arendajad igapäevaselt kokku puutuvad.

Reaktiivse programmeerimise mõiste oli lahti seletatud ning võrreldud erinevate teiste paradigmatega.

Konkreetse projekti näitel probleemid olid järjestatud olulisuse alusel ning grupeeritud kriteeriumite järgi.

Kasutades Reaktiivset ja Imperatiivset paradigmat pakkus autor välja probleemidele lahendused, mis omakorda olid võrreldud eelnevalt määratletud kriteeriumitega.

Esimene ja kõige murettekitavam teema kõigi uuringus osalenute jaoks olid kaua kestvad taustategevused ja astmeliste funktsioonide pärimisega seotud probleemid.

Autor üritas tõhusalt täita koormavaid ülesandeid tausta harudes ja kuvada tulemusi kasutajaliideses. Kahte kasutusjuhtu võeti analüüsimiseks.

Esmapilgul tundus, et imperatiivse lähenemise tulemused paistsid liiga triviaalsed ja lihtsad, kuid võrreldes reaktiivse lähenemisega selgus, et kõik erijuhud ei olnud kaetud.

Samuti tuli välja, et ilma RxJava/ RxAndroid teegita iga järgmise päring hakkab sõltuma eelnevast ja lisab tasemeid ja koodi keerukust.

Rippmenüü näitel, demonstreeris autor kuidas reaktiivset lähenemist võiks kohaldada lahendamaks probleeme, mis puudutavad kasutajaliidest ja kasutajamugavust.

Selle tulemusena reaktiivne lähenemine aitas eraldada vaadete ja mudelite loogikat, optimeerida sündmustetöötlejaid ja vältida mälu lekkeid.

Autor näitas ka Rx operaatorite efektiivsust vormide valideerimisel. Samuti ka manipuleerides, filtreerides, muutes ja ühendades objekte.

Autor kasutas analüütilist hierarhilist mudellähenemist (AHP) ning tegi valmis otsustuspuu imperatiivse ja reaktiivse paradigmade võrdlemiseks. Autor loodab, et otsustuspuu, saab lihtsustada teiste Android arendajate elu ning aidata õigesti valida, kas kasutada imperatiivse või reaktiivse lähenemise vastavalt määratletud kriteeriumitele.

RxJava ja RxAndroid teegid on veel vähe tuntud. Endiselt leidub palju arendajaid, kes ei tea kuidas ja miks nad peavad neid kasutama konkreetsete ülesannete lahendamiseks.

Autor loodab, et käesolev magistritöö aitab mõista, millal Reaktiivne lähenemine võiks olla kõige tõhusam, aidates parandada koodi kvaliteeti ja lihtsustada arendusprotsessi keskkonnas, kus nõuded arenevad ja muutuvad kiiresti kogu projekti vältel.

References

[1] J. Bonér, D. Farley, R. Kuhn and M. Thompson, "The Reactive Manifesto," 16 September 2014. [Online]. Available: https://www.reactivemanifesto.org. [Accessed 20 December 2016].

[2] T. Nurkiewicz and B. Christensen, Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications, O'Reilly Media, 2016.

[3] F. J. Fowler, "Designing Questions to Be Good Measures," in *Survey Research Methods*, 1984, pp. 76-79.

[4] A. Staltz, "The introduction to Reactive Programming you've been missing," 2016. [Online]. Available: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. [Accessed 3 November 2017].

[5] "RxJava: Reactive Extensions for the JVM," ReactiveX, 2015. [Online]. Available: https://github.com/ReactiveX/RxJava. [Accessed 2 March 2017].

[6] K. Webber, "What is Reactive Programming?," RedElastic, 19 August 2014. [Online]. Available: https://blog.redelastic.com/what-is-reactive-programming-bc9fa7f4a7fc. [Accessed 13 November 2017].

[7] M. Fowler, "Event Collaboration," 19 June 2006. [Online]. Available: https://www.martinfowler.com/eaaDev/EventCollaboration.html. [Accessed 3 May 2017].

[8] A. Staltz, "Interactive diagrams of Rx Observables," 2015. [Online]. Available: http://rxmarbles.com/#skip. [Accessed 10 December 2017].

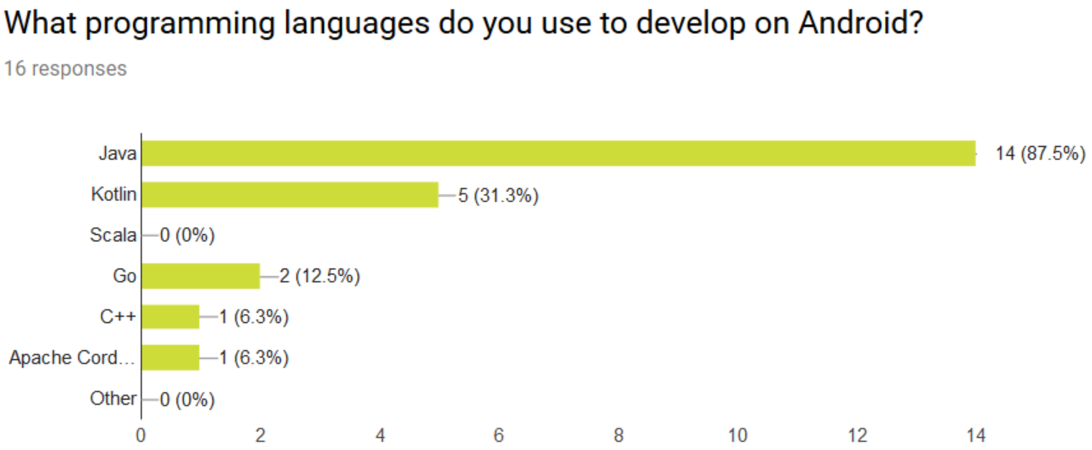
[9] A. Huang and C. Arriola, Reactive Programming on Android with RxJava, MYNAH, 2017.

[10] *Meetings with Reactive Programming evangelists on the Droidcon London conference.* [Interview]. 2017.

[11] T. Nield, Learning RxJava: Reactive, Concurrent, and responsive applications, Packt Publishing, 2017.

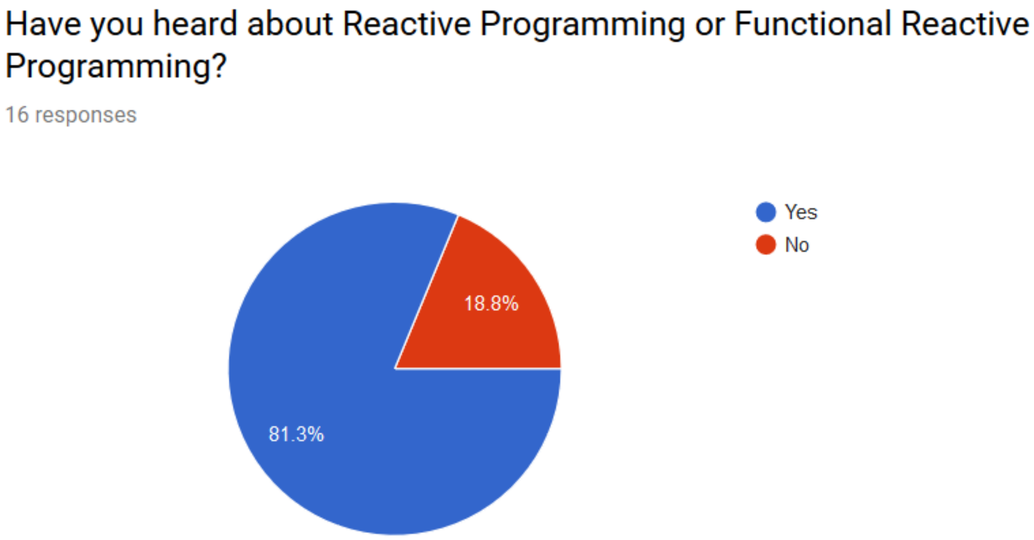
[12] K. D. Goepel, "AHP online system," 2016. [Online]. Available: https://bpmsg.com/academic/ahp-hierarchy.php. [Accessed 21 September 2017].

Appendix 1 - Survey results



What programming languages do you use to develop on Android?

16 responses



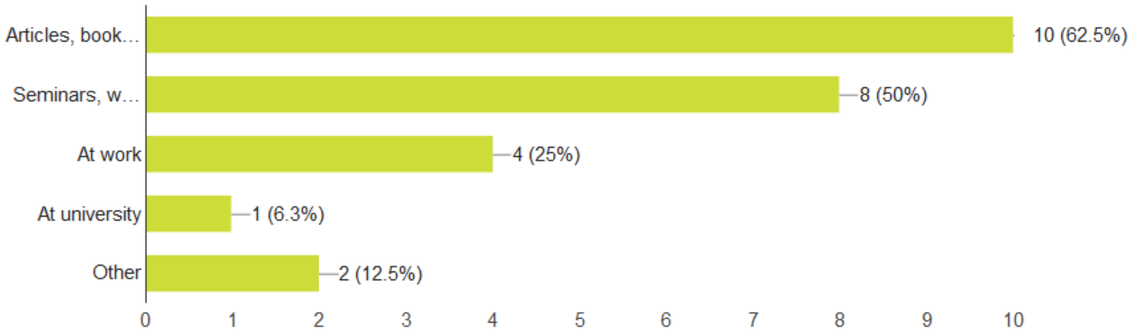
Have you heard about Reactive Programming or Functional Reactive Programming?

16 responses

Appendix 2 - Survey results

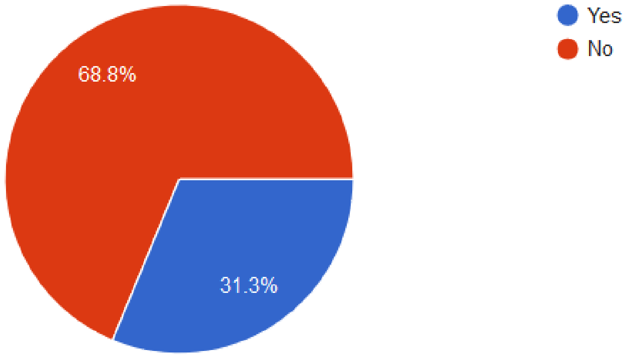
Where have you heard about it?

16 responses



Have you used it in the real project/ projects?

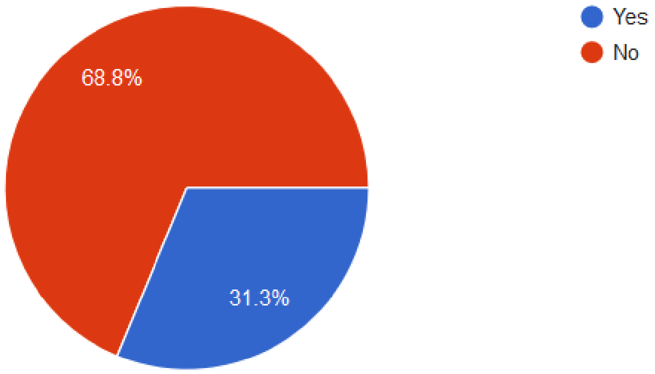
16 responses



Appendix 3 - Survey results

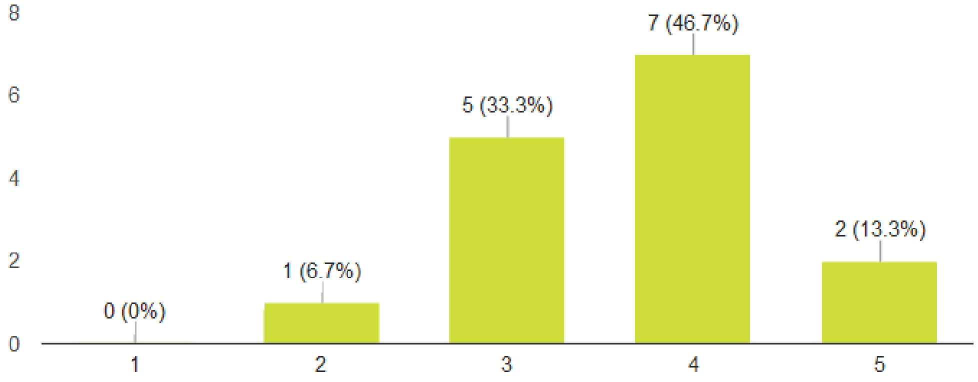
Have you used it in the pet projects?

16 responses



How challenging it was to start using Reactive Programming?

15 responses



8
6
4
2
0
0 (0%)
1 (6.7%)
5 (33.3%)
7 (46.7%)
2 (13.3%)
1
2
3
4
5