DESIGNING A MOBILE APPLICATION ON THE EXAMPLE OF A SYSTEM FOR DIGITAL PHOTOS WATERMARKING

Paweł Kaczmarek*[a], Zbigniew Piotrowski[a]
[a] Military University of Technology, gen. Sylwestra Kaliskiego 2 str., 01-476 Warsaw, Poland

ABSTRACT

The development of mobile technologies implies the increase of the popularity of applications – programs dedicated for these devices. The production of an IT system intended for many end users must include the stages of analysis, design and development of the mobile application in order to achieve commercial success. The content created for a mobile device can have the form of a responsive website or a mobile application. The article presents the process of designing a native mobile application for the Android operating system on the example of a system for marking digital photos with a resistant watermark. The use of the Inversion of Control design pattern used in the application and the use of the programming paradigm - reactive programming - was discussed. The article describes the disadvantages and advantages of technology and libraries used when designing a mobile application on the example of a system for identifying digital photos and their owners on the Internet using data hiding techniques.

Keywords: Android application, mobile application design, dependency injection, reactive programming

1. INTRODUCTION

The development of mobile technologies implies the increase of popularity of applications – programs dedicated to these devices. Production of an IT system intended for many end users must include the stages of analysis, design and development of the mobile application in order to achieve commercial success. Content created for a mobile device may take the form of a responsive website or a mobile application. The article presents the process of designing a native mobile application for Android operating system based on an example of a system used to affix resistant watermarks to digital photos.

2. DESCRIPTION OF ANDROID SYSTEM ARCHITECTURE

2.1 Description of Android operating system

Android operating system should be understood as a software with an open source, created on the basis of Linux system and dedicated for a wide range of devices and electronic equipment. Figure no. 1 presents the Android program stack. The platform is based on the Linux kernel responsible for low-level management of drivers and management of multithreading. The next overlay is the Hardware Abstraction Layer (HAL), which maps the assets and functions of hardware devices to appropriate interfaces of the high-level programming language – Java API. When Java API refers to a specific hardware asset, this layer loads a library module for this component. Two parallel layers are run one after another – Android Runtime (ART) and C/C++ native libraries, respectively. Devices that use Android system in version 5.0 and higher start applications in separate processes using own ART stack. ART launches many virtual machines by executing DEX files, produced as a result of development of the application, on them. DEX files are consolidated files that contain a byte code created as a result of Java compilation. Before version 5.0, Dalvik was the virtual machine. and the runtime for Android. Operation of Dalvik is based on compiling a byte code at each start of application (just-in-time method), while ART compiles a byte code during installation of an application (ahead-of-time method). Because it is based on the Linux kernel, Android requires natively written libraries using C/C++ programming language. When writing an application, one may use Android NDK (Native Development Kit) to utilise the source code contained therein. The functions of Android operating system are available to users via API interfaces written in Java. They constitute another layer of the system stack. Java API Framework layer provides solutions such as: creation of

* pawelk.kaczmarek@wat.edu.pl; phone: 48 261 839 715; https://itk.wel.wat.edu.pl

Radioelectronic Systems Conference 2019, edited by Piotr Kaniewski, Jan Matuszewski, Proc. of SPIE Vol. 11442, 114420V · © 2020 SPIE · CCC code: 0277-786X/20/$21 · doi: 10.1117/12.2565225

application views using a view system, enabling access to graphics or files, access to data of other applications and management of a life cycle of an application or a single activity. The last element of the stack are system applications.

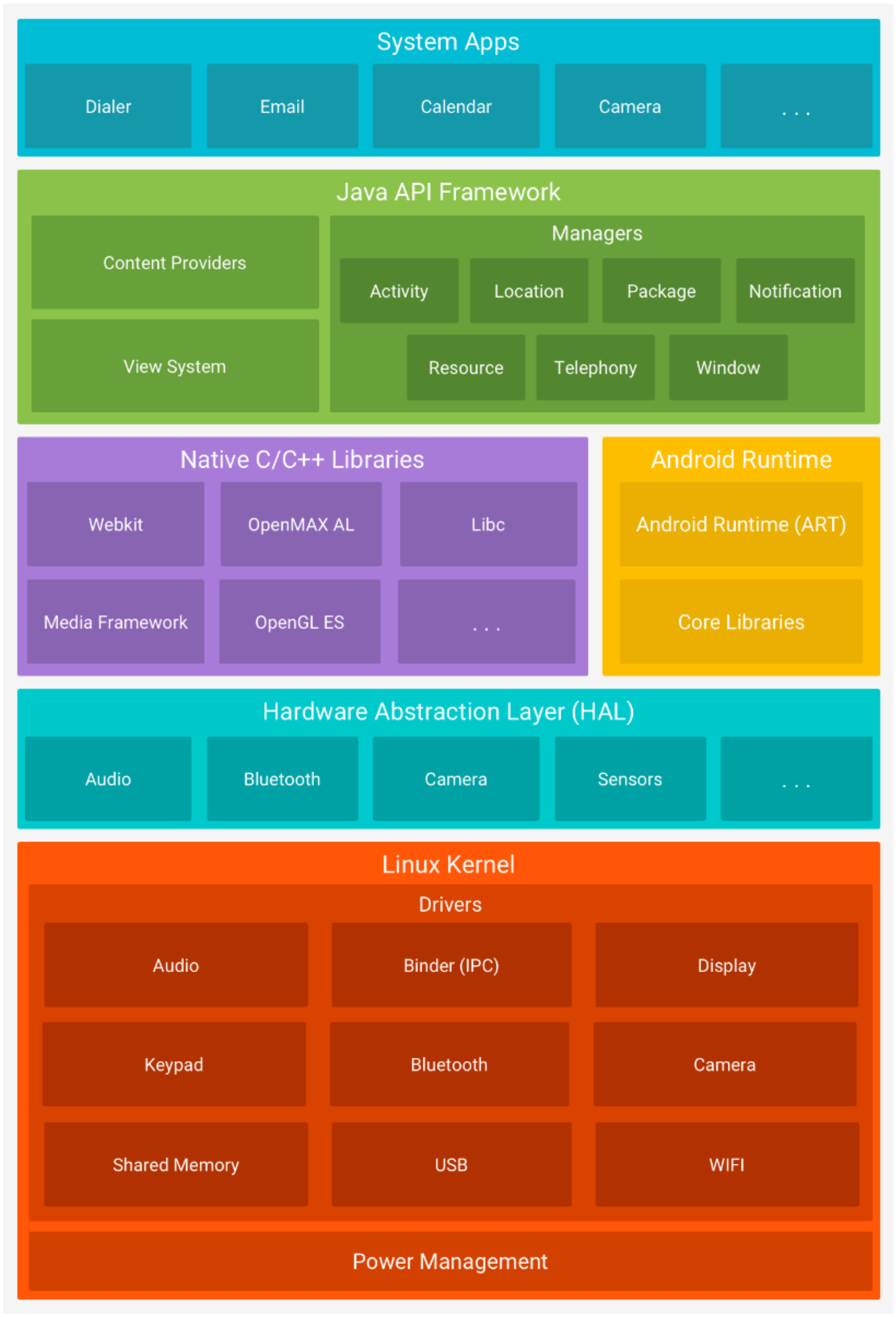
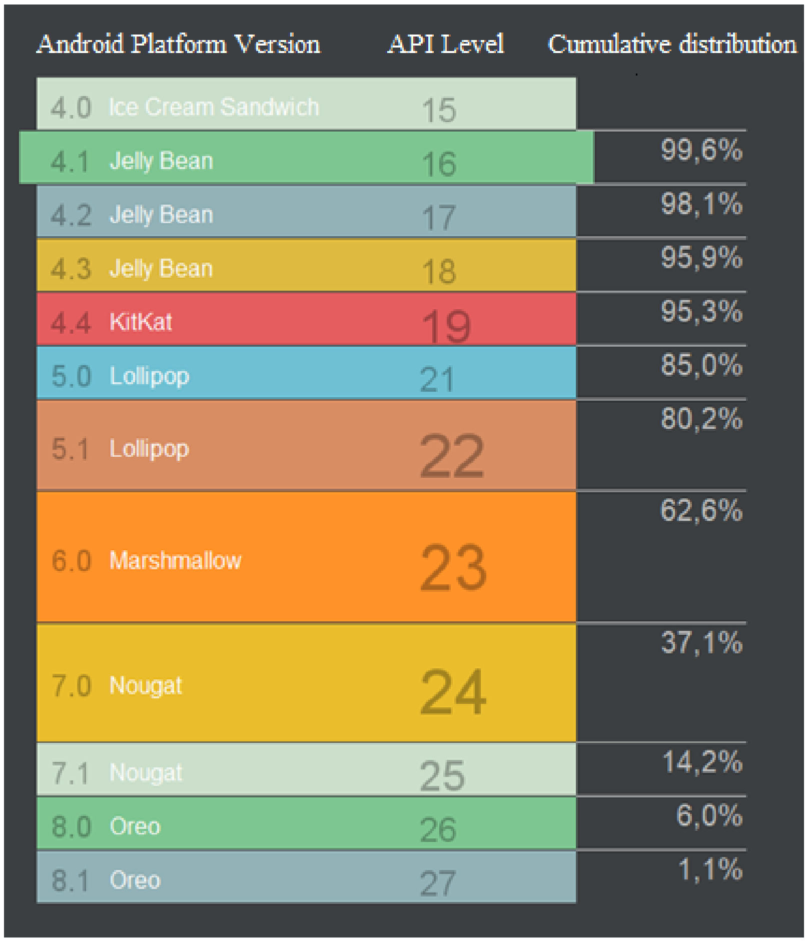


Figure 1. Android program stack [1].

These applications are installed on a physical device by default and are used to support basic services such as calls, e-mail or camera [1].

2.2 Components of a mobile application for Android operating system

Before starting to design an application, you should choose a programming language. Available programming languages include: Kotlin, Java, and C++. Android SDK (Software Development Kit) intended for programmers compiles a code including all assets, thus creating APK file, which is distributed to end devices via various methods. A system or a user may access an installed application via one of the four basic elements that create this application.



Android Platform Version		API Level	Cumulative distribution
4.0	Ice Cream Sandwich	15	
4.1	Jelly Bean	16	99,6%
4.2	Jelly Bean	17	98,1%
4.3	Jelly Bean	18	95,9%
4.4	KitKat	19	95,3%
5.0	Lollipop	21	85,0%
5.1	Lollipop	22	80,2%
6.0	Marshmallow	23	62,6%
7.0	Nougat	24	37,1%
7.1	Nougat	25	14,2%
8.0	Oreo	26	6,0%
8.1	Oreo	27	1,1%

Figure 2. Percentage coverage of devices with Android system depending on API version [3].

These baseline elements are: activity, service, broadcast receiver, and content providers. Each of them has own life cycle, which determines the method of its creation and destruction. Activity is a single application screen used to interact with the user. Activity may include various views or fragments that group these views in order to allow their reusability in various screen resolutions. Background activity is ensured by the service. Services have no user interfaces. Broadcast receivers enable delivery of system events, thus allowing to support system messages in the application. Content provider is a component used to manage a data set by sharing these data with other applications via files, databases or other mechanisms. For the system, content provider is an entry point to the application, enabling publishing of named data elements identified by the scheme of the Uniform Resource Identifier (URI) [2]. Three out of four types of components – activities, services, and broadcast receivers – are activated by an asynchronous message known as intent. Intents bind together particular components when they are executed.

They are messages that demand actions from other components, regardless of the application they are a part of. When starting a component of an application, Android operating system must know all elements included in the application. Therefore, the application must declare all its components in the manifest file called AndroidManifest.xml, which must be present in the main directory of the application project. Apart from a source code, an application also includes assets such as images, audio files or animations. Identifying an asset by adding it to a project in a specific place in /res directory makes SDK generate a unique identifier – an integer that may be used to refer to an asset from a source code or other assets identified in XML files.

Creation and management of a project requires an Integrated Development Environment (IDE). The official IDE used to create applications for Android is Android Studio based on IDE IntelliJ IDEA by JetBrains. Besides the code editor and IntelliJ programming tools, Android Studio offers a compilation system based on Gradle tool used to manage dependencies, an emulator of many systems and physical devices, a unified development environment for all devices with Android system – phones, tablets, watches, TVs, cars with Android Auto system, as well as Android Things systems based on the Internet of Things (IoT). Each project in Android Studio includes one or more modules with source code files and assets files. When creating a new project, the minimum version of Android's API must be specified. Currently, API 16 version allows to run the developed application on approximately 99.6% of devices, as shown in figure no. 2 [3].

3. DESCRIPTION OF THE DESIGN PROCESS OF A MOBILE APPLICATION FOR ANDROID OPERATING SYSTEM

3.1 Specification of the dependency of an application project

Knowing the business requirements and functions realised by the mobile application, we may start the design process. The system used to affix watermarks to digital photos has API defined, created and described in RESTfulAPI architecture. It has been assumed that the system will collect photos from the user, mark them, and allow to view and edit them. The system will authenticate users in two stages, using e.g. a signed photo. The services will be published as commercial services. The aforementioned assumptions allow to start a new project in Android Studio. Based on the specification provided above, the following set of technologies and design tools was selected:

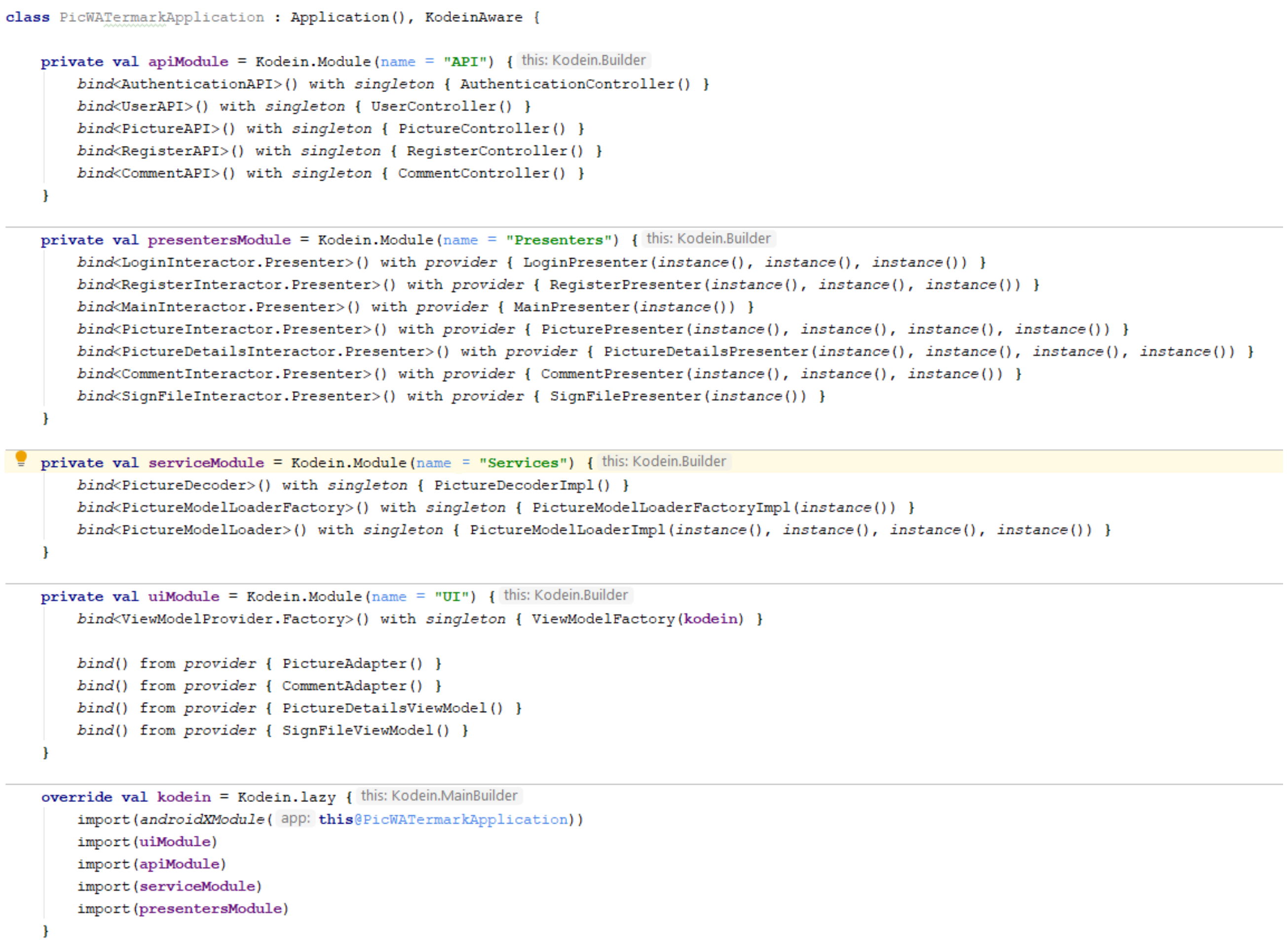
- Programming language: Kotlin version 1.30,
- Minimum Android SDK 16,
- Maximum Android SDK 29,
- Jetpack package libraries – libraries prepared by the authors of Android, which allow to speed up the process of software development by providing complete components and tools that facilitate the assumed tasks,
- RxJava2 library – a library that includes extensions for reactive programming,
- Kodein library – a dependencies container,
- Glide library – tools that ensures full processing of multimedia – decoding, buffering, and saving to local cache,
- Fast Android Networking library with Rx extension – a library that allows to execute all sorts of network connections in an application.

Such a complete set of tools is a basis for further considerations associated with the application project. Additional libraries needed to implement specific function will not impact the developed architecture of the application, e.g. libraries for unit or automatic testing, since the employed patterns and paradigms allow to create a fully functional and reliable native application for Android system. Android operating system should be understood as a software with an open source, created on the basis of Linux system and dedicated for a wide range of devices and electronic equipment. Figure no. 1 presents the Android

Assembling all tools and libraries in the first iteration of development of software is impossible, as functionalities or the scope may be subject to change after presenting the end results to a business client or a new tool or library may be developed that may increase the quality and safety of the application, and thus new possibilities must be taken into account. Moreover, the design team should use tools that will support their teamwork. Examples and uses of such tools are described in [4][5].

3.2 Implementation of dependencies injection as an implementation of reverse control

Languages used to create applications for Android system are object-oriented programming languages. As a paradigm, the object-oriented software identifies objects as elements connecting data with maintenance of the software. Inversion of control (IoC) of dependencies consists of transfer of responsibilities related to creation and connection of components outside the objects.

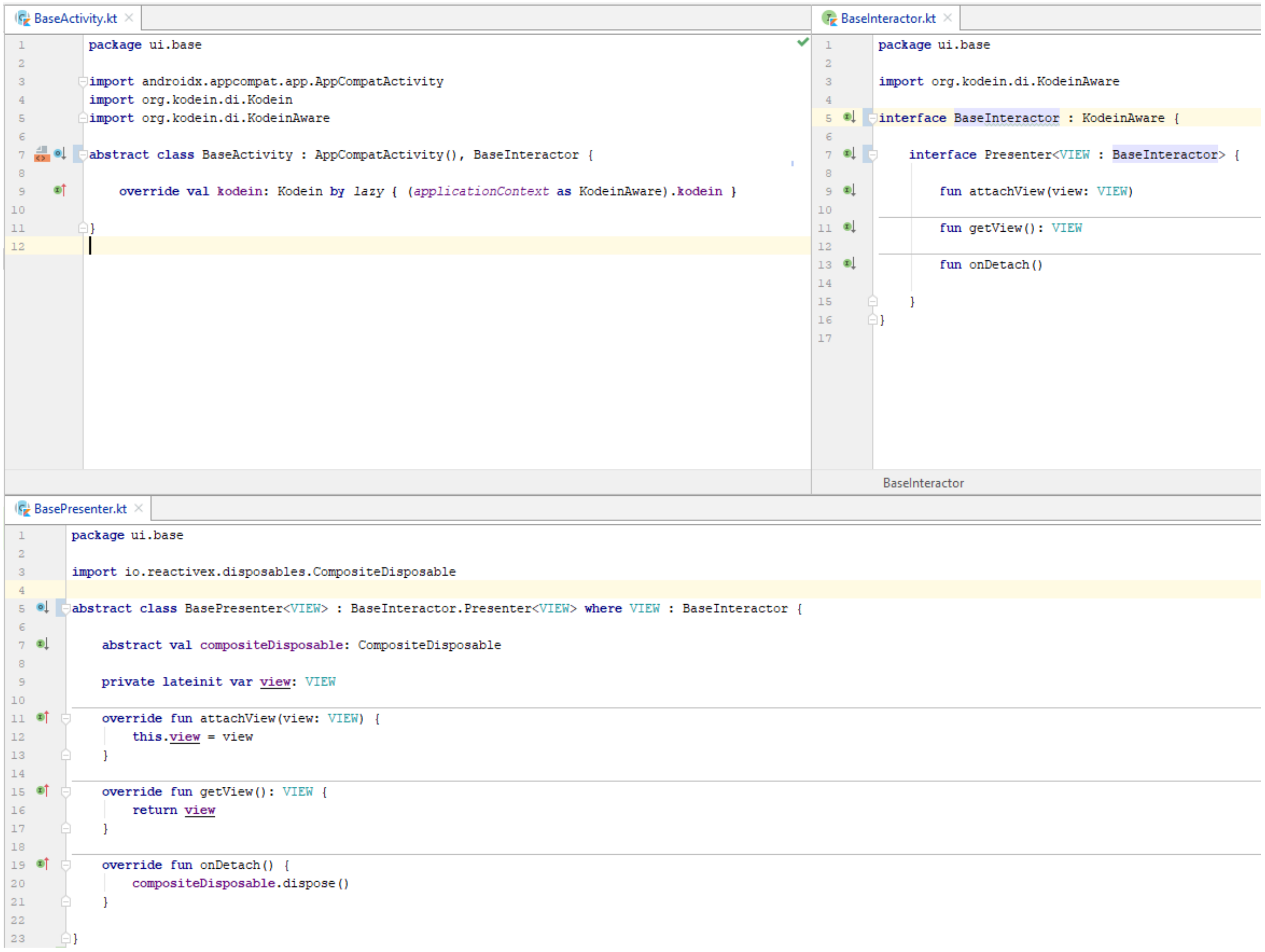


```
class PicWATermarkApplication : Application(), KodeinAware {

    private val apiModule = Kodein.Module(name = "API") { this: Kodein.Builder
        bind<AuthenticationAPI>() with singleton { AuthenticationController() }
        bind<UserAPI>() with singleton { UserController() }
        bind<PictureAPI>() with singleton { PictureController() }
        bind<RegisterAPI>() with singleton { RegisterController() }
        bind<CommentAPI>() with singleton { CommentController() }
    }

    private val presentersModule = Kodein.Module(name = "Presenters") { this: Kodein.Builder
        bind<LoginInteractor.Presenter>() with provider { LoginPresenter(instance(), instance(), instance()) }
        bind<RegisterInteractor.Presenter>() with provider { RegisterPresenter(instance(), instance(), instance()) }
        bind<MainInteractor.Presenter>() with provider { MainPresenter(instance()) }
        bind<PictureInteractor.Presenter>() with provider { PicturePresenter(instance(), instance(), instance(), instance()) }
        bind<PictureDetailsInteractor.Presenter>() with provider { PictureDetailsPresenter(instance(), instance(), instance(), instance()) }
        bind<CommentInteractor.Presenter>() with provider { CommentPresenter(instance(), instance(), instance()) }
        bind<SignFileInteractor.Presenter>() with provider { SignFilePresenter(instance()) }
    }

    private val serviceModule = Kodein.Module(name = "Services") { this: Kodein.Builder
        bind<PictureDecoder>() with singleton { PictureDecoderImpl() }
        bind<PictureModelLoaderFactory>() with singleton { PictureModelLoaderFactoryImpl(instance()) }
        bind<PictureModelLoader>() with singleton { PictureModelLoaderImpl(instance(), instance(), instance(), instance()) }
    }

    private val uiModule = Kodein.Module(name = "UI") { this: Kodein.Builder
        bind<ViewModelProvider.Factory>() with singleton { ViewModelFactory(kodein) }

        bind() from provider { PictureAdapter() }
        bind() from provider { CommentAdapter() }
        bind() from provider { PictureDetailsViewModel() }
        bind() from provider { SignFileViewModel() }
    }

    override val kodein = Kodein.lazy { this: Kodein.MainBuilder
        import(androidXModule( app: this@PicWATermarkApplication))
        import(uiModule)
        import(apiModule)
        import(serviceModule)
        import(presentersModule)
    }
```

Figure 3. Declarations of bindings between interfaces and implementations of particular classes [Own elaboration].

This responsibility is most often transferred to IoC container. The rules of inversion of control specify the abstraction, according to which objects should not have specific dependencies and may not obtain them in any other way than by receiving them from outside. Dependency injection (DI) is a design pattern that implements the inversion of control principle. Software written using DI is strictly structured into configuration parameters. Components are configured via process of searching the space of dependencies without modifying the source code of the components. The traditional operation of DI container consists of creating connections between constructors of dependent objects (also knows as “object graphs”) by checking a configuration object or a file that includes a list of associations between abstract types and their constructor arguments. Then, the container selects the target class and delivers dependencies to an appropriate constructor of the target class. Certain DI containers may not provide dependencies if an object graph includes ambiguities, such as selection of many abstract class subtypes or implementation of interfaces. Therefore, if the class of object A requires another class of object B in order to execute functionalities, there are several means to trigger an appropriate action. First of all, A class object may directly create class B instance – this is simple, but it makes replacement of alternative class B implementations more difficult. Secondly, A class object may obtain class B instance from another service, such as a localiser of factory or a service, allowing use of alternative implementations, but makes class A strictly dependant on the resolution strategy. Finally, by using DI, A class may require provision of B object using a constructor argument. This means that a dependency (B class) is injected to A class object. Therefore, each component that creates A class object may replace an alternative implementation or change class B configuration by any means [6][7][8].

In the original solution, Kodein library is the implementation of DI [9]. Dagger2 library may be used as an alternative, but it was not selected due to the employed programming language – Kotlin. Figure no. 3 presents declaring of connections between interfaces uploaded to the constructor and their particular implementations. Such a solution allows to change implementation of a given class without modifying the code of the class that uses it. The class that represents the application (by extending the Application class from Android SDK) has all dependencies managed by DI container, allowing to change them in a single configuration file. A container called kodein is available from other classes within the application. It allows to apply the Model View Presenter (MVP) design pattern, in which the presenter acts as an intermediary between the view and the model. The view is responsible for interaction and response to the user events.



```kotlin
package ui.base

import androidx.appcompat.app.AppCompatActivity
import org.kodein.di.Kodein
import org.kodein.di.KodeinAware

abstract class BaseActivity : AppCompatActivity(), BaseInteractor {

    override val kodein: Kodein by lazy { (applicationContext as KodeinAware).kodein }

}
```

```kotlin
package ui.base

import org.kodein.di.KodeinAware

interface BaseInteractor : KodeinAware {

    interface Presenter<VIEW : BaseInteractor> {

        fun attachView(view: VIEW)

        fun getView(): VIEW

        fun onDetach()

    }
}
```

```kotlin
package ui.base

import io.reactivex.disposables.CompositeDisposable

abstract class BasePresenter<VIEW> : BaseInteractor.Presenter<VIEW> where VIEW : BaseInteractor {

    abstract val compositeDisposable: CompositeDisposable

    private lateinit var view: VIEW

    override fun attachView(view: VIEW) {
        this.view = view
    }

    override fun getView(): VIEW {
        return view
    }

    override fun onDetach() {
        compositeDisposable.dispose()
    }

}
```

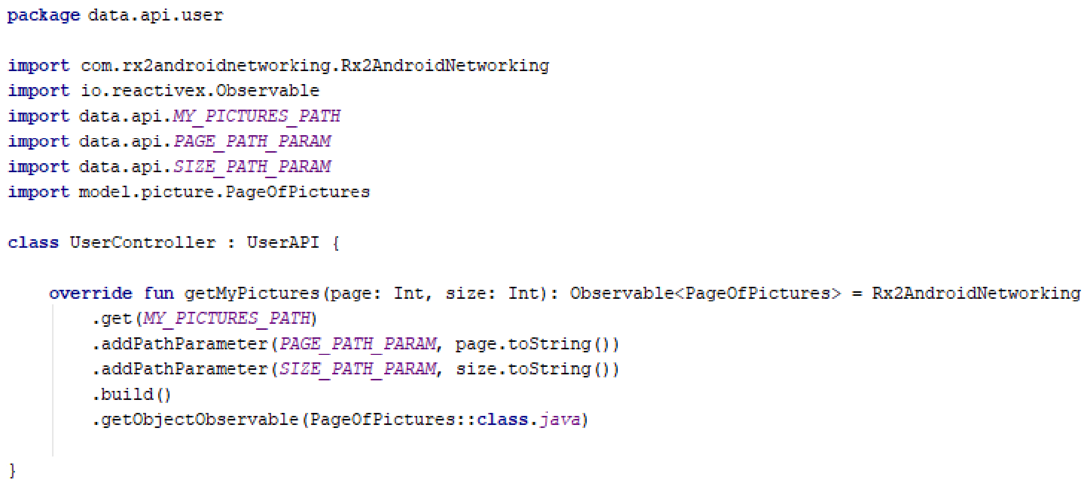
Figure 4. Separation of classes from an activity component as per MVP pattern [Own elaboration].

Figure 4 shows division into three classes that correspond to the aforementioned statement. Due to the fact that they inherit properties of BaseActivity, BaseInteractor and BasePresenter classes, the component classes are managed by DI Kodein. The dependency component has a delivered implementation of BaseInteractor.Presenter interface through the injected dependency. The presenter classes obtain dependencies to other objects via the constructor, into which DI container also injects class object instances. This solution allows to structure dependencies configuration in a single file and used them in classes that require them [9].

3.3 Use of the reactive programming paradigm

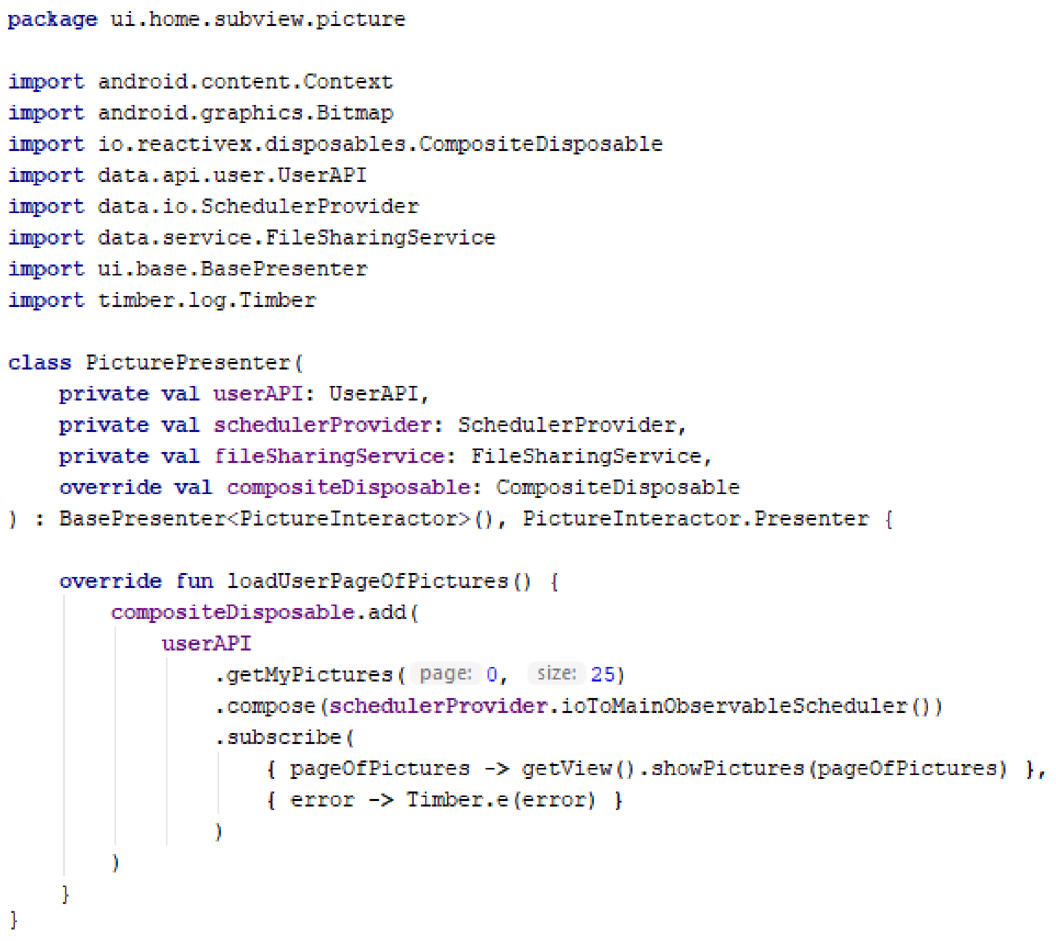
The user interface (UI) seen by the end user is created in the activity component. After launching, the application has programmed listeners, which are directly assigned to specific controls – basic graphic elements of the interface.

The original application uses RxJava tools to fully support requests of REST HTTP, including integration with the server’s API. Longer operations, such as processing of photos into a coded string in Base64 or writing to a file, are also executed and managed by RxJava. Figure 5 presents the user’s implementation of API. getMyPictures method returns the Observable object, which is then registered by the observer.



```
package data.api.user

import com.rx2androidnetworking.Rx2AndroidNetworking
import io.reactivex.Observable
import data.api.MY_PICTURES_PATH
import data.api.PAGE_PATH_PARAM
import data.api.SIZE_PATH_PARAM
import model.picture.PageOfPictures

class UserController : UserAPI {

    override fun getMyPictures(page: Int, size: Int): Observable<PageOfPictures> = Rx2AndroidNetworking
        .get(MY_PICTURES_PATH)
        .addPathParameter(PAGE_PATH_PARAM, page.toString())
        .addPathParameter(SIZE_PATH_PARAM, size.toString())
        .build()
        .getObjectObservable(PageOfPictures::class.java)

}
```

Figure 5. The class that implements downloading of the user photos using RxJava [Own elaboration]



```
package ui.home.subview.picture

import android.content.Context
import android.graphics.Bitmap
import io.reactivex.disposables.CompositeDisposable
import data.api.user.UserAPI
import data.io.SchedulerProvider
import data.service.FileSharingService
import ui.base.BasePresenter
import timber.log.Timber

class PicturePresenter(
    private val userAPI: UserAPI,
    private val schedulerProvider: SchedulerProvider,
    private val fileSharingService: FileSharingService,
    override val compositeDisposable: CompositeDisposable
) : BasePresenter<PictureInteractor>(), PictureInteractor.Presenter {

    override fun loadUserPageOfPictures() {
        compositeDisposable.add(
            userAPI
                .getMyPictures( page: 0,  size: 25)
                .compose(schedulerProvider.ioToMainObservableScheduler())
                .subscribe(
                    { pageOfPictures -> getView().showPictures(pageOfPictures) },
                    { error -> Timber.e(error) }
                )
        )
    }
}
```

Figure 6. PicturePresenter class [Own elaboration].

The listeners respond to user actions, forcing business actions such as: sign in, attachment of a photo for affixing or requesting affixing of a watermark on a photo. The application displays the UI in the main thread, and each code called from an activity is implemented there, thus blocking and preventing further interaction. Such an operation is inconsistent with the guidelines for designing applications for Android system [1]. REST HTTP or other actions that require longer duration may be called by a service component, however, this is insufficient in large applications, and thus event stream processing solutions should be used. An asynchronous approach based on data sequences and observers constitutes reactive programming. They may be treated as an extension of an observer pattern. Objects that are observed (observables) are asynchronous, and information regarding operation is emitted via an event. It is delivered to each observer, which is an object interested in a given event. Each observable type object is able to emit zero or more events within its life cycle. RxJava library is Reactive Extensions implementation – an observer pattern implementation. It allows to create asynchronous programs, based on events, using observable sequences. It extends the observer pattern to allow it to support data sequences and events by adding operators, enabling declarative compilation of sequences, and at the same time eliminating problems such as: low-level threading, synchronisation, safety of threads and concurrent data structures.

This method is called by PicturePresenter, which is illustrated in figure no. 6. loadUserPageOfPictures() method uses UserAPI by calling getMyPictures() method in a separate thread and observes the results in the main thread – this is ensured by compose() method. Then, subscribe() method is called, which subscribes the created Observable object and ensures return call to support all elements and any errors [10]. The presented use of RxJava tools is implemented using the dependency injection pattern [10].

4. CONCLUSION

Before starting to design a mobile application, the main requirements and assumptions for the application must be compiled. One must understand the architecture of Android system and the principle of operation of particular applications. Selecting a modern programming language and current libraries that improve and speed up creation of a source code is of key importance. Using patterns and paradigms of programming allows to create an efficient, complete application that will achieve a commercial success. The digital photos watermarking system has full REST API specification and a website that uses this API, which allowed to single out libraries such as: Android Jetpack, Kodein, RxJava and Fast Android Networking. Using an open source code significantly accelerates the process of production of a source code. Dependencies injection design patterns and Model ViewPresenter pattern allow to create a simple and clear architecture. Such an architecture is easy to maintain and develop. Use of the reactive programming paradigm enables interactions with the model in an asynchronous and quick manner that does not block the screen. The entirety ensures a complete collection of solutions for designing a mobile application for Android system.

REFERENCES

[1] Google Developers, Android Platform, https://developer.android.com/guide/platform
[2] Google Developers, Android Platform, https://developer.android.com/guide/components
[3] Google Developers, Android Platform, https://developer.android.com/studio/
[4] Kaczmarek P., Wojtczak M., 2019, "GitLab - a tool for team work of programmers", Elektronika – konstrukcje, technologie, zastosowania, 6/2019, s.25-28, DOI 10.15199/13.2019.6.5.
[5] Kaczmarek P., Wojtczak M., 2019, "Architecture and configuration of tools for working in a programming team in the Scrum methodology", Elektronika – konstrukcje, technologie, zastosowania, 7/2019, s.34-36, DOI 10.15199/13.2019.7.10
[6] Jóźwiak, I.; Sasnal, P., 2012, "Inversion of control of dependencies strategy in Go programming language",
[7] Ekstrand, M. D., Ludwig, M., 2016, "Dependency injection with static analysis and context-Aware policy", Journal of Object Technology, Vol.15(1), pp.1-31
[8] Kocsis Z., Swan J., 2017, "Dependency Injection for Programming by Optimization", arXiv.org
[9] Kodein Framework, https://kodein.org
[10] ReactiveX Project, http://reactivex.io/