iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** IV **Month of publication:** April 2021

DOI: https://doi.org/10.22214/ijraset.2021.33608

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

Dependency Injection in Android Development

Kaajal Sharma[1], Abhishek Jagushte[2], Avadhoot Khedekar[3], Sourav Katkar[4]
[1, 2, 3, 4]*Computer Engineering Department, Mumbai University*

Abstract: Dependency Injection is a term for creating a class's dependencies outside of it and providing them to the class as parameters. Throughout this paper we'll see the necessity, benefits and the ways to achieve the same in Android Development as it plays a very crucial role in large scale applications due to its key advantages. We'll also see the implementation details for a most recent framework Dagger-Hilt developed as an abstraction over Google's Dagger library already in use by many large scale Android projects.
Keywords: Android Development, Dependency injection, Dagger, Hilt, Kotlin

I. INTRODUCTION

Object Oriented programming (OOP) is a programming methodology that consists of concepts called classes and objects. This is used to structure a software program into reusable pieces of code, which are used to create individual instances of objects. Designing an object-oriented software is a difficult task. First you have to identify the objects you want, then factor them into the classes. You have to define class interfaces and inheritance hierarchies, identify the relationships among them. You have to design your software such that it meets the requirements and also it is adaptable to the changes required in the future. It happens that developers use their useful previous solutions or approach for making a new software. Similarly, there are recurring patterns of classes and communicating objects in many object-oriented systems. One such problem occurs when an object has other objects as it's dependencies. In such case, the dependent object either needs to create a new instance of the required object or an already existing instance must be passed to it as parameter while creating the object.
This instantiation of the required class by the dependant class is a tedious task. There are patterns used to simplify this process. These recurring patterns which make design reusable, flexible to the future changes are known as design patterns.

II. THE SOLUTION

In object oriented programming it may happen that a common problem is occurring repeatedly. Programmers have been seen to use their previous programming logic to tackle the repetitive problem. This programming logic can be seen as a design pattern.
There are many design patterns that aim to solve a specific problem. With the help of these design patterns, our code can be flexible, maintainable, reusable, it can be tested quickly and thus it will make our software development process faster. Design patterns can be classified into three types - Creational, Structural and Behavioural [1].

A. Creational design patterns aim at how to create or instantiate an object.
B. Structural design patterns aim at identifying the relationships in classes and objects. Its aim is to simplify the structure of a class.
C. Behavioural design patterns deal with the interaction between different classes. They aim to have this interaction smooth with less dependencies.

III.NEED FOR SOLUTION IN ANDROID DEVELOPMENT

MVVM architecture in Android Development is used to separate Graphical User Interface (GUI) from the backend or business logic. Google recommends programmers to use this architecture in Android App Development. MVVM consists of Model, View and ViewModel. This architecture is used to remove tight coupling between the View and the data variables (Model).

1) *Model:* This component stores data and its related logic.
2) *View:* This component presents the data as an output to the user.
3) *View Model:* It links the model and view. Its function is to operate on models and to support the state of view. If the data or model changes, it is the job of ViewModel to update the View.

Consider the given scenario in Fig 1 wherein our MainActivity needs an instance of MainViewModel. Our MainActivity can create an instance of MainViewModel, but it will also have to manage the dependencies required by the MainViewModel class. In this case these dependencies are MyName, MyCity. Also, it may happen that other classes might also need an instance of MainViewModel, creating an instance of MainViewModel each time any class needs it will create a lot of duplicate code. If each class creates an instance, our application could become resource heavy.

©IJRASET: All Rights are Reserved

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

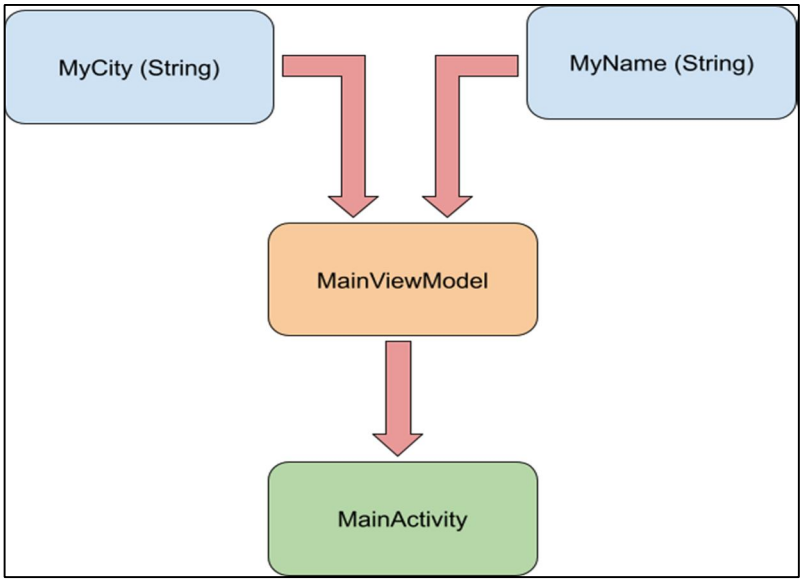


Fig. 1 Example for dependencies between classes

To solve the aforementioned problem, below methods can be used:

A. Service Locator

Service locator is a central registry. Whenever our class needs an instantiation of another class it asks for its implementation from the service locator. Service locator provides implementation of different interfaces. In this way our class doesn't need to instantiate the required class nor it needs to know which class is going to instantiate the required class.

Service locator has a cache to return the implementation of the interface that has already been used. It also has an initializer that is used when an interface has to be implemented for the first time.

B. Dependency Injection

In software development, classes have to interact with other classes to use their methods. We cannot use methods of a class directly; first we need to create an object of that required class. A class can get its object from the other class in three ways

1) The class itself creates an object by instantiating the class
2) Ask for interfaces or other sources to provide the implementation
3) To get the object as a parameter.

Getting the object as a parameter is dependency injection. In this way, the class doesn't have to worry about instantiating the required class. Instead the dependencies are provided directly to the class. Many have adapted to dependency injection because of its advantages over service locator pattern.

There are basically three types of dependency injection:

a) Constructor injection: In this approach, the dependencies are passed as parameters in the class constructor
b) Property injection: In this approach, the dependencies are passed in public property of the client class.
c) Method injection: In this approach, the client class implements an interface which declares the methods to supply the dependency and the injector uses this interface to supply the dependency to the client class.

C. Advantages of using Dependency Injection

1) Reusability of code - By separating the creation of an object from its usage, this method can replace a dependency without changing any code and it also reduces the boilerplate code in the business logic.
2) Ease of refactoring - Dependency Injection does not require any changes in the code behaviour. For a new and simple application refactoring is simply straightforward. For a complex application there are various types of dependency injection which could be used according to the situation to break it down to smaller refactoring.
3) Ease of testing - It brings flexibility in unit testing. For unit testing you can create a fake class of your own test data and methods and simply pass it on to test your different scenarios and configure that fake class for different tests.

©IJRASET: All Rights are Reserved

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

IV. DEPENDENCY INJECTION FRAMEWORKS IN ANDROID – DAGGER-HILT

Dagger-Hilt is built as an abstraction over the original *Dagger* that is infamously used for dependency injection in Android Development. It is ridden with some of the disadvantages which are essential to address especially when working with smaller projects. The whole setup is perplexing and uses a lot of boilerplate code. *Dagger-Hilt* is designed to overcome these disadvantages with the following goals:

1) To rationalize Dagger-related infrastructure for Android apps.
2) To create a standard set of components and scopes to ease setup, readability, and code sharing between apps.
3) To provide an easy way to provision different bindings to various build types viz. testing, debugging and release.

Hilt uses annotation processing in order to generate the *Dagger* code for the programmer so that the programmer can focus on the business logic more and less on unimportant details and boilerplate surrounding the usage of dependency injection with *Dagger*. *Hilt* generates the *Dagger* components and the code to automatically inject in Android classes like activities and fragments.
In order to use *Hilt* in an Android project, first the dependencies and plugins need to be set up in the project's build properties. The plugin, 'hilt-android-gradle-plugin' is added at the project's root build.gradle file. As an example:

```
buildscript {
   dependencies {
      classpath  "com.google.dagger:hilt-android-gradle-plugin:2.33-beta"
   }
}
```

Hilt needs two gradle plugins for successfully generating the required code, they are as follows:

```
apply plugin: 'kotlin-kapt'
apply plugin: 'dagger.hilt.android.plugin'
```

The kotlin-kapt plugin helps in annotation processing and the dagger.hilt.android.plugin is responsible for generating the necessary code for dependency injection. Finally, there are two dependencies that need to be added at the app level *build.gradle* file:

```
dependencies {
   implementation "com.google.dagger:hilt-android:2.33-beta"
   kapt "com.google.dagger:hilt-compiler:2.33-beta"
}
```

A. Hilt Application Class

In order to use *Hilt* in a project, the project needs to have an Application Class annotated with @HiltAndroidApp.

```
@HiltAndroidApp
class MyApplication: Application () {}
```

In Android, overriding the Application class is followed by adding the attribute *name* to the *application* element in the *AndroidManifest.xml.*

```
<application
  android:name = ".MyApplication"
…
```

In Android, certain classes are never instantiated by the programmer but are done by Android System. Dependency Injection in such classes can be done through *Hilt* too. It can provide dependencies to such Android classes that have the @AndroidEntryPoint annotation:

1) Application (by using @HiltAndroidApp)
2) ViewModel (by using @HiltViewModel)
3) Activity
4) Fragment
5) View
6) Service
7) BroadcastReceiver

©IJRASET: All Rights are Reserved

If an Android Class is annotated with @AndroidEntryPoint annotation, then all the classes that depend on it must also be annotated with @AndroidEntryPoint annotation. @AndroidEntryPoint generates an individual *Hilt* component for each Android class in a project.

At build time, *Hilt* generates *Dagger* components for Android classes. Then, *Hilt* performs the following steps:

a) Builds and validates dependency graphs, ensuring that there are no unsatisfied dependencies and no dependency cycles.

b) Generates the classes that it uses at runtime to create the actual objects and their dependencies.

B. Hilt Components

Dagger components are interfaces where programmers define functions and return the instances of the class that are needed. Unlike this, *Hilt* programmers have no need to define and instantiate the *Dagger* Components, *Hilt* already has a predefined set of components that can be used. These components are integrated with various Android Lifecycles like Activity Lifecycle, Fragment Lifecycle, etc. The figure 2 below shows the *Hilt* component hierarchy. The annotation above each component is used to scope bindings to the lifetime of that component. The arrow below a component is an arrow from parent component to child component. Normally, a binding in a child component can have dependencies on any binding in an ancestor component. [6]

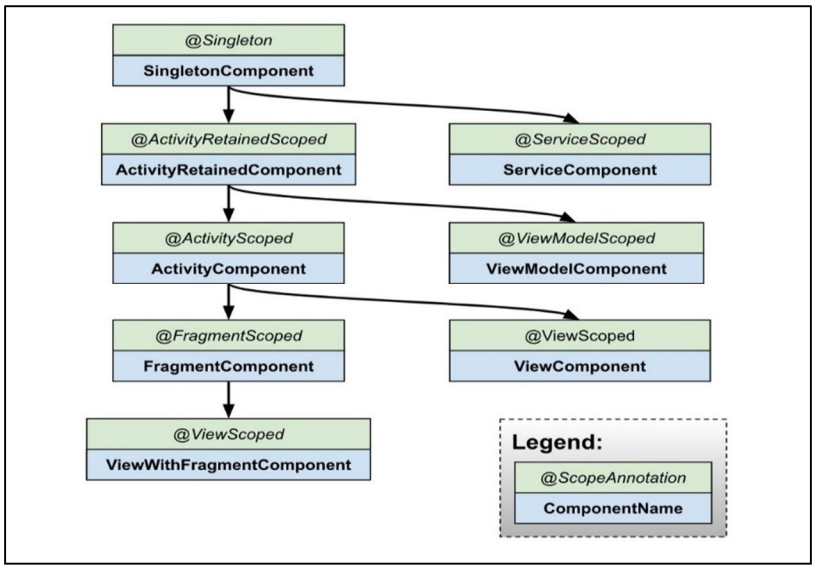


Fig. 2 Hilt Component Hierarchy

C. Hilt Modules

Hilt Module is a class annotated with @Module that informs *Hilt* how to provide instances of certain types. These are needed to be installed in *Hilt* components. This is done by using the @InstallIn annotation. Depending on the component the module is installed in, it has a specify scope. Table 1 shows all the different components and the times when they are created and destroyed. Continuing with the previously considered example as given in Fig 1,

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
  @Provides
  @Singleton
  fun provideMyName() = "Hilt Android"
}
```

@SingletonComponent used in the @InstallIn annotation makes sure that the AppModule has Application binding. This means that the scope of the objects provided through AppModule is application-wide. Below table shows all the possible options for @InstallIn and their respective lifecycles. [5]

©IJRASET: All Rights are Reserved

TABLE I
Lifecycle of components

Generated component	Created at	Destroyed at
SingletonComponent	Application#onCreate()	Application#onDestroy()
ActivityRetainedComponent	Activity#onCreate()	Activity#onDestroy()
ViewModelComponent	ViewModel created	ViewModel destroyed
ActivityComponent	Activity#onCreate()	Activity#onDestroy()
FragmentComponent	Fragment#onAttach()	Fragment#onDestroy()
ViewComponent	View#super()	View destroyed
ViewWithFragmentComponent	View#super()	View destroyed
ServiceComponent	Service#onCreate()	Service#onDestroy()

The method provideMyName() is annotated with @Provides and @Singleton annotation. @Provides is used to specify a provider method for the specific return type, String in the above case. The @Singleton annotation ensures that only one instance of the provided String “Hilt Android " is ever created throughout the life of the application.

D. Providing Multiple Bindings for Same Type

In cases where two objects of the same type need to be provided, for example MyName and MyCity in the following example both provide String, @Qualifier annotation is used. Here *Hilt* needs to be informed about how to provide an instance of the type that corresponds with each qualifier.

```
@Qualifier
@Retention (AnnotationRetention.BINARY)
annotation class MyNameString

@Qualifier
@Retention (AnnotationRetention.BINARY)
annotation class MyCityString
```

The provider methods will change as follows:

```
@Provides
@Singleton
@MyNameString
fun provideMyName () = "Hilt Android”
@Provides
@Singleton
@MyCityString
fun provideMyCity () = "Mumbai"
```

There are predefined qualifiers in *Hilt* which are used to provide common instances like ApplicationContext and ActivityContext provided using @ApplicationContext and @ActivityContext qualifiers.

©IJRASET: All Rights are Reserved

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

E. Injecting the Dependencies

Generally, dependency injection is done via Constructor Injection or Field Injection. Both the methods are described below.

F. Field Injection

In case of injection into an Android Activity, constructor injection is not possible as Activities are instantiated by Android OS. So, field injection is used in this case. For this to work, the activity class first needs to be annotated with @AndroidEntryPoint annotation.

```
@AndroidEntryPoint
class MainActivity: AppCompatActivity() {

  @Inject @MyNameString lateinit var myName: String

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     //myName can be used below now
  }
}
```

G. Constructor Injection

Dependencies can be injected directly to a class via the class constructor. Below code snippet shows how this can be achieved:

```
class StringUser @Inject constructor (
  @MyNameString val myName: String,
  @MyCityString val myCity: String
)
```

Now the programmer cannot instantiate this class but *Hilt* does it. In order to get an instance of this class, below line of code is used:

```
@Inject lateinit var stringUser: StringUser
```

Hilt instantiates the StringUser class by providing its dependencies viz. myName and myCity.

H. ViewModel Injection with Hilt

Google recommends using the MVVM Architecture for applications and using ViewModels to properly manage data variables in case of screen configuration changes. A *ViewModel* annotated with @HiltViewModel is available for creation by the dagger.hilt.android.lifecycle.HiltViewModelFactory and can be retrieved by default in an Activity or Fragment annotated with @AndroidEntryPoint. The HiltViewModel containing a constructor annotated with @Inject will have its dependencies defined in the constructor parameters injected by *Hilt*.

```
@HiltViewModel
class MainViewModel @Inject constructor (
     @MyNameString val myName: String,
     @MyCityString val myCity: String
): ViewModel () {}
```

This *ViewModel* can now be accessed from an activity using the fragment-ktx module added via adding implementation "androidx.fragment:fragment-ktx:1.3.2" in the app level *build.gradle* file. The Fragment KTX module provides a number of extensions to simplify the fragment API.

```
private val mainViewModel: MainViewModel by viewModels()
```

Thus the *ViewModel* can be accessed just by one line of code as opposed to writing multiple lines while using plain *Dagger* for Android.

©IJRASET: All Rights are Reserved

V. CONCLUSION

Dependency Injection is a necessity for medium to large sized Android Development Projects. Frameworks like *Dagger-Hilt* which and being developed and maintained by Google have played an important role in simplifying the overall experience of implementing Dependency Injection in projects.

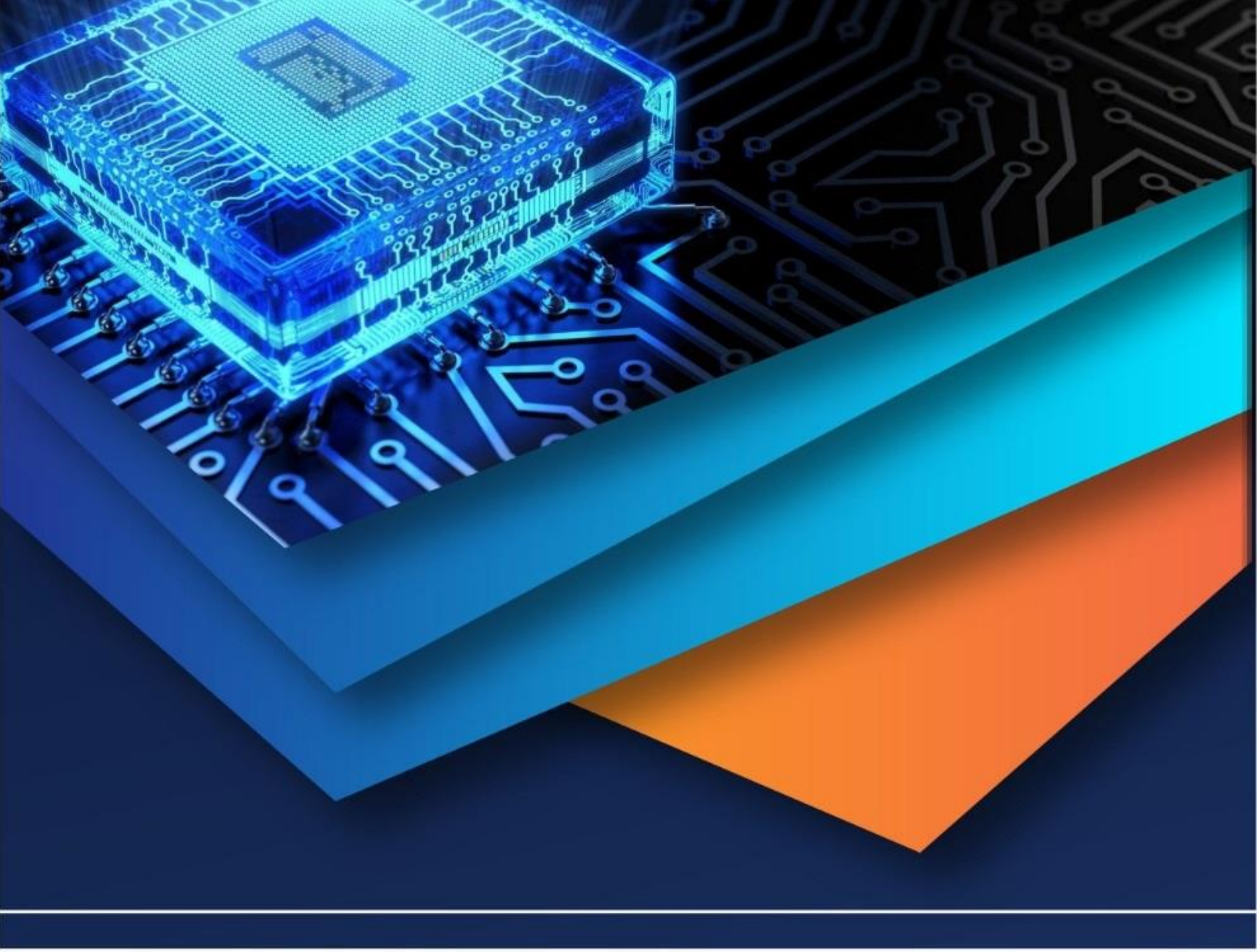
VI. ACKNOWLEDGMENT

We would like to express our gratitude to the people who have assisted us during this course of research. The support extended by Prof. Kaajal Sharma and Department of Computer Engineering, MCT's Rajiv Gandhi Institute of Technology, Mumbai is highly appreciated and acknowledged with due respect.

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (2015), Design Patterns Elements of Reusable Object-Oriented Software, 1st ed, Pearson Education
[2] Mateus, Bruno & Martinez, Matias. (2019). An empirical study on quality of Android applications written in Kotlin language. Empirical Software Engineering. 24. 10.1007/s10664-019-09727-4.
[3] Clow, Mark. (2018). Dependency Injection. 10.1007/978-1-4842-3279-8_13.
[4] Verdecchia, Roberto & Malavolta, Ivano & Lago, Patricia. (2019). Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study. 10.1109/ICSA.2019.00023.
[5] Dependency injection with Hilt | Android Developers
[6] Hilt Components (dagger.dev)
[7] View Models (dagger.dev)
[8] Modules (dagger.dev)
[9] Entry Points (dagger.dev)

©IJRASET: All Rights are Reserved



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)