International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 17 (2015) pp 38562-38566


# «Dependency injection» on Android and its application in the development of a mobile multifunction hardware-software complex of long cardiomonitoring and ergonometry

**Denis NikolaevichAkolzin**
Scientific and Technical Center "Technocenter" Southern Federal University,
347900, 81 Petrovskaya street

**Ivan AlexandrovichGrinko**
Scientific and Technical Center "Technocenter" Southern Federal University,
347900, 81 Petrovskaya street

Abstract- The article discusses the use of the mechanism of "Dependency Injection" allowing you to dynamically describe dependencies in a code, separating the business logic into smaller blocks. This is useful primarily to the fact that you can later replace these small blocks with the tested ones, and reduce the test zone. Dependency injection is the process of adding external dependence to a software component. Is a specific form of "Inversion of Control", where a change in the order of communication is carried out by obtaining the necessary dependencies. Dependency Injection is more flexible because it easier allows to create alternative implementations of service, and then specify what kind of implementation should be used, for example, the configuration file, without any changes in the objects that use this service. This is especially useful in unit testing because the implementation of insert "stub" service in the object under test is very simple. On the other hand, excessive use may make implementation dependencies applications more complex and difficult accompanied: to understand the behavior of the application, developer should look not only at the source code, but also in the configuration, which generally invisible for IDE, that supports analysis of references and refactoring, unless explicitly configured to support dependency injection frameworks. This mechanism has allowed the testing of the application and conduct a parallel application development without the portable unit complex. In many cases, the introduction of the designer is much easier and more reliable, but there are situations where the introduction of a property is the right choice.

Keywords: Java, Android, Bluetooth.

## Introduction

The mechanism of dependences injection (Dependency Injection or DI) belongs to the list of the most incorrectly perceived concepts of object-oriented programming. This confusion is widespread and concerns the terminology, purposes and mechanics.

This paper will discuss about the use of the mechanism of dependences injection on Android platform within the work on the project of a mobile multipurpose hardware-software complex of long cardiomonitoring and ergonometry.

First it is necessary to explain what dependence injection is. The mechanism of dependence injection is a set of the principles and templates of the design of software which allow developing a code in which it is possible to modify components easily, or to replace the whole component with its analog almost without touching other part of the project [1 - 3].

The development of such code in this project is necessary for several reasons:

- The transfer and data acquisition in the application from the module must depend on the chosen Bluetooth protocol as little as possible. Because the transition should support not only Bluetooth Serial Port, but also Bluetooth Low Energy, which in difference in connection realization, transfer and data acquisition. Dependency injection will allow using at once two protocols of data transmission in the application depending on the protocol of the module of the system;

- When testing this appendix, it is useful to simulate the module of the system creating a fictitious service for the work with it. In strong dependence of components of the application it is impossible, or is too complicated;

- For the development of the application without the module of the system it is also useful to emulate its work. For example, if it will be necessary to check the display work when processing the wrong data, the emulation of the module on the device, without change of an initial code, will be made.

The use of dependences injection provides some advantages including the showed below:

- Weakening of connection between classes. Dependences are accurately defined in each class. Data on the configuration and comparisons between interfaces or basic classes and actual concrete types are stored in the container used by the mechanism of dependences injection. If it will be necessary they can be updated without making any changes into a code during the performance;

- Creation of a code, which undergoes a check better. From types of constructors, properties or methods of the user classes it is possible easy to know what objects they use and what dependences are between them. While creation of copies with the use of a code in classes it is much more difficult to trace dependences. It is recommended to allow dependences in the field of a class by the indication of types

or interfaces which it demands and to use the advantage of dependences injection;

- Testing simplification. If a user allows or gets objects with the use of a code in classes, it is necessary to specify in a right way the adjusted container for the use during the modular testing of these classes. If to take advantage of dependences injection, it is possible to create simple models of test objects for classes which are applied by a user.

The mechanism of dependency injection allows distributing responsibility between classes. As soon as responsibility of each class becomes obviously certain and limited, the process of support of the whole application becomes easier. This advantage of the principle of the only responsibility, which claims that each class must have just one responsibility, is well-known. The process of adding new opportunities in the application becomes simpler because it is clear where it is necessary to make changes [4]. Usually it is not even necessary to change an existing code, instead of it, new classes should be added and it is unnecessary to compile the application all over again. Here the principle of one responsibility is entered the game. The search of malfunctions also becomes less tiresome because the area of possible culprits for malfunctions is narrowed. Thanks to obviously certain responsibilities there is an accurate understanding of that wherefrom it is necessary to begin the search of a root reason of problem formation [5 - 7].

With the use of dependency injection it is possible to realize the concept of the separate development. The concept of division makes the development of a code by parallel commands possible. When the project reaches a certain size, it becomes necessary to divide the developer's team into some teams of a manageable size. Each team has the responsibility over any area of the application. To differentiate the responsibility, each team will develop one or more than one module which will be embedded into the finished application [8]. Only when spheres of the activity of each team will obviously depend from each other, the consistent development will be necessary.

For dependences injection the following patterns are used:

- Injection into the constructor (Constructor Injection);

- Injection into the property (Property Injection).

Injection into the constructor works in the following way: the class, which demands the object, must provide the open constructor which accepts realization of the necessary object as an argument of the constructor. In most cases, it must be only the available constructor. If more than one object is necessary, additional arguments of the constructor can be used. A good practice is to note the field containing the object as "readonly" - it guarantees that as soon as the initialization logic of the constructor will be done, the field cannot be changed. It is not obligatory from the point of view of dependences injection, but it will protect you from a casual change of the field (for example, its installation on "null") somewhere in other place of a code dependent on a class.

## Choice by default for dependences injection

This approach has to be a choice by default for dependences injection. He allows saying to a class with confidence that injection exactly is, and it is simple for realization.

In the approach of injection into the property the class which uses dependence, must provide the property of the object type which is open, available to write down. This approach is very easy in use. But representation can be deceptive, and injection into the property is interfaced to difficulties. It is difficult to realize it in a right way. Clients can forget (or not want) to provide realization of the object, or by mistake to appropriate "null" as a value. Besides, what must happen if a client will try to change the object in the middle of life cycle of a class? It can lead to inconsistent or unexpected behavior therefore you can want to protect yourself from this event [10 - 14].

With injection into the constructor it is possible to protect a class against such incidents using a keyword "readonly" to the field, but it is impossible when the object is opened as the written-down property. In many cases injection into the constructor is much easier and more reliable, but there are situations when injection into the property is a right choice [15 - 17]. It is in that case when providing the object realization is optional because there is a good localization of this class. Injection into the property also ties to a library which is used for realization of the mechanism of dependences injection. Injection into the constructor allows to disconnect these libraries during the testing and to transfer obviously test objects in constructors.

For understanding of this mechanism and problems solved by it is necessary to consider an example below. The example is the initial approach for the Android application work with the module of the system on Bluetooth. Figure 1 presents the interrelation between classes and the main components for interaction with the module of the system with the use of Bluetooth. In effect, "application" processes events of user actions. For data transmission it uses the BluetoothService object which encapsulates the work with data transmission on the Bluetooth protocol using BluetotohSocket for direct sending of data.

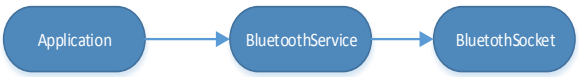


Fig. 1. The scheme of classes for data transmission

Figure 2 shows how it looks in a code. For descriptive reasons and explanations of the work of dependences injection, there is only one EnableModule method, which sends a command of inclusion to the module of the system. In this code programming is conducted concerning a class abstraction. Many guides on the object-oriented design are focused on interfaces as on the main mechanism of abstractions while the guides on the design on the JAVA basis support a superiority of abstract classes over interfaces. Is it necessary to use interfaces or abstract classes? Concerning the mechanism of dependences injection the consolatory answer to this question is that it does not matter what to use. It is only important that programming is conducted concerning any sort of abstraction. The choice

between interfaces and abstract classes is important in other contexts but not in this.

```
public class Application
{
    private BluetoothService api;
    public Application()
    {
        this.api = new BluetoothService();
    }

    public void EnableModule(String date)
    {
        api.sendCommand(date);
    }
}

public class BluetoothService
{
    private BluetoothSocket bluetoothSocket;

    public BluetoothService()
    {
        this.bluetoothSocket = new BluetoothSocket();
    }

    public void sendCommand(String message)
    {
        //отправляем данные
        bluetoothSocket.Send(message);
    }
}
```

Fig. 2. The example of realization

Figure 3 shows the use of this code. It is necessary to create the class Application and to cause in it a method of sending an inclusion command.

```
Application  application= new Application();
application.EnableModule("Enable");
```

Fig. 3.The use of the Application class.

In this code everything works well and also a code is quite simple. But if it is necessary to test this code, there is at once a need to replace BluetoothSocket on test one which will generate messages. In this case we need to relieve responsibility from the BluetoothService class, to create BluetoothSocket and to transfer it to the higher classes. At introduction of these completions the Aplication class starts changing. Figure 4 shows the received code. Transfer of the BluetothSocket object is done through constructors of classes.

```
public class Application
{
    private BluetoothService api;
    public Application(BluetoothSocket bluetoothSocket)
    {
        this.api = new BluetoothService(bluetoothSocket);
    }

    public void EnableModule(String date)
    {
        api.sendCommand(date);
    }
}

public class BluetoothService
{
    private BluetoothSocket bluetoothSocket;

    public BluetoothService(BluetoothSocket bluetoothSocket)
    {
        this.bluetoothSocket =bluetoothSocket;
    }

    public void sendCommand(String message)
    {
        //отправляем данные
        bluetoothSocket.Send(message);
    }
}
```

Fig. 4. Realization for testing

Now in order to use the Application class during the creation of this object it is necessary to transfer to it BlueoothSocket. There is an opportunity to transfer the test object instead of it.

```
Application  aplication= new Application(new BluetoothSocket());
aplication.EnableModule("Enable");
```

Fig. 5.The use of a code with the injection of the Bluetooth Socket class through the constructor.

It is not difficult. Actually the pattern dependency injection is by hand organized. But if there is a need for the use of Bluetooth Service in other place, it is also necessary to create BluetoothSocket and to transfer it by the underlying class [18]. Thereby to carry out the duplication of a code and also to connect the application classes by the unified constructor. Of course, it is possible to add constructors by default which will create a real BluetoothSocket and to use the second constructor only when testing, but after all it does not solve the problem but only masks it. It is possible to improve this problem having realized the library for "Dependences injection".

With company resources it was possible to realize the own library for this purpose [19]. But such approach also has disadvantages, this is quite expensive in comparison with costs for the project, and also the testing of this library is necessary. That is why it was decided to use a ready library Dagger 2.

Dagger 2 is the library with an open source code from the developersokhttp, retrofit, picasso and many other remarkable libraries known to many Android developers.

The main advantages of Dagger 2:

- The static analysis of all dependences;

- The definition of errors of the configuration at a compilation stage (not only in “runtime”);

- The lack of “reflection” that considerably accelerates the process of the configuration;

- A quite small load on the memory.

For the configuration of a library it is necessary to create modules which will provide essences on demand. For this system the figure 6 shows the realization of such module. The module provides two essence- BluetoothSocket and BluetoothService. When granting BluetoothService it automatically will provide the realization of BluetoothSocket for it [20 - 22]. For this purpose in the constructor of BluetoothService essence the @Inject attribute is specified.

```
@Module
public class BluetothModule {
    @Provides
    @Singleton
    BluetoothSocket provideBluetoothSocket()
    {
        return new BluetoothSocket();
    }

    @Provides
    @Singleton
    BluetoothService provideBluetoothService(BluetoothSocket socket)
    {
        return new BluetoothService(socket);
    }
}
```

Fig. 6. The module of granting essences

Further it is necessary to create the interface to which the external application will address. Figure 8 presents its initial code. It gives a chance not to provide the BluetoothSocket essence owing to its local use in the module.

```
@Singleton
@Component(modules = {BluetothModule.class})
public interface BluetoothComponent {
    BluetoothService provideBluetoothService();
}
```

Fig. 7. The interface of granting essences to external components

Now, to receive the BluetothService essence the external component needs to execute a code presented in the figure 7. In this code with the use of the interface which the figure 8 has earlier shown, there is a receiving of the necessary essence.

```
BluetoothComponent daggerBluetoothComponent = DaggerBluetoothComponent.builde
        .bluetothModule(new BluetothModule()).build();
this.api = daggerBluetoothComponent.provideBluetoothService();
```

Fig. 8. Receiving of the essence of BluetotohService

An impression may be arise that the use of Dagger 2 has only complicated a primary code. But this representation is false. The small by the size bases of a code which are similar to the given above example, are in essence supported thanks to the size; that is why the mechanism of dependences injection is perceived in simple examples as the excess development [23]. Than more by the size there is a base of a code, especially visible are advantages of the mechanism of dependences injection.

The mechanism of dependences injection by its nature is means of the result achievement, but not the purpose. It is the best way of dependences reduction of components from each other which is an important component of a supported code. The advantages which are provided by the use of this mechanism, are not always immediately obvious, but become noticeable eventually when the complexity of a code increases. The primary code, where components are strongly dependent from each other eventually will become difficult for understanding, meanwhile at rather weak dependence of components it can remain supported [24]. To reach the real "flexible design", it is necessary not only to carry out easy dependences between components, but programming on the basis of the interface that is a necessary condition of the "flexible design".

The DI mechanism is more than just a set of the principles and patterns. It is more likely a way of inventing and designing of a code than means and receptions – an important point of weak binding, it is also the mechanism of dependences injection, and for efficiency it must be used in all components of the developed applications.

## Conclusion

The results of researches, which this paper has presented, were got with the financial support of the Ministry of Education and Science of the Russian Federation within realization of the "Creation of Advanced technology production on fabrication of a mobile multipurpose hardware-software complex of long cardiomonitoring and ergonometry" project according to the governmental resolution № 218 (9 April, 2010). Researches were conducted in FSAEI of HE of the SFU.

## References

[1] Dependency Injection, Annotations, and why Java is Better Than you Think it is // Hetzer. 2013. URL:http://www.objc.io/issue-11/dependency-injection-in-java.html(accessed date: 29.03.2015).

[2] Mark Seemann Dependency Injection in .NET. Manning Publications. 2011.

[3] Dominic Betts, Grigori Melnik Dependency Injection with Unity. Microsoft. 2014.

[4] Freeman, J. 2013. Who nailed the principles of great UI design? Microsoft, that's who // InfoWorld. URL: http://www.infoworld.com/article/2614315/application-development/who-nailed-the-principles-of-great-ui-design--microsoft--that-s-who.html?page=3 (accessed date: 29.03.2015).

[5] Dependency injection on Android: Dagger (Part 1) // Antonio Leiva.2015. URL: http://antonioleiva.com/dependency-injection-android-dagger-part-1/ (accessed date: 29.03.2015).

[6] Johan, T.J. JuergenBocklage-Ryannel and Johan Thelin. 2014.

[7] Dependency injection on Android: Dagger (Part 2) // Antonio Leiva.2015. URL: http://antonioleiva.com/dependency-injection-android-dagger-part-2/ (accessed date: 29.03.2015).

[8] Zharko,Miyailovich D.M. 2013,Tehnologiirazrabotkipolzovatelskihinterfeisov // Otkrytyesistemy. URL: http://www.osp.ru/os/2013/10/13039072/ (accessed date: 29.03.2015).

[9] Dependency injection on Android: Dagger (Part 3) // Antonio Leiva.2015. URL: http://antonioleiva.com/dependency-injection-android-dagger-part-3/ (accessed date: 29.03.2015).

[10] Android: Inversion of Control, Dependency Injection, Dagger - Part 1 // Leftshift.io. 2015. URL: http://leftshift.io/android-inversion-of-control-dependency-injection-dagger-part-1 (accessed date: 29.03.2015).

[11] Android: Inversion of Control, Dependency Injection, Dagger - Part 2 // Leftshift.io. 2015. URL: http://leftshift.io/android-inversion-of-control-dependency-injection-dagger-part-2 (accessed date: 29.03.2015).

[12] Dependency Injection Options for Java // Java Code Geeks. 2015. URL: http://www.javacodegeeks.com/2015/02/dependency-injection-options-for-java.html (accessed date: 29.03.2015).

[13] Dependency Injection on Android // DroidCon NL. 2013. URL: http://www.slideshare.net/JoanPuigSanz/dependency-injection-on-android (accessed date: 29.03.2015).

[14] Dependency Injection: anti-patterns // Habrahabr. 2013. URL: http://habrahabr.ru/post/166287/ (accessed date: 29.03.2015).

[15] Design pattern – Inversion of control and Dependency injection // Code project. 2015. URL: http://www.codeproject.com/Articles/29271/Design-pattern-Inversion-of-control-and-Dependency (accessed date: 29.03.2015).

[16] Dependency injection в Java EE 6 // Habrahabr. 2015. URL: http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html (accessed date: 29.03.2015).

[17] Patternyproektirovaniya // Qt Official Site. 2015. URL: http://doc.qt.io/qt-5/qtqml-cppintegration-topic.html (accessed date: 29.03.2015).

[18] Goloshchapov,A. 2010, Google Android: programmirovaniedlyamobilnykhustroistv. Spb.: BKHV-Petersburg.pp. 448.

[19] Molen, Brad (2012-01-14). "Samsung Gear 2 smartwatches coming in April with Tizen OS". Engadget.com. Retrieved 2014-07-22.

[20] Komatineni, S.,Maklin D. &Kheschimi S. 2011, Google Android: programmirovaniedlyamobilnykhustroistv = Pro Android 2. 1-st edition. Spb.: Saint-Petersburg, 2011. pp. 736.

[21] The Qt Company Ltd. Qt Bluetooth // Qt Official Site. 2015. URL: http:// doc.qt.io/qt-5/qtbluetooth-index.html (accessed date: 29.03.2015).

[22] Young, D.G. 2014, A Solution for Android Bluetooth Crashes // Radius Networks. URL: http://developer.radiusnetworks.com/2014/04/02/a-solution-for-android-bluetooth-crashes.html (accessed date: 29.03.2015).

[23] Satiya,Komatineni&DeivMaklin. Android 4 dlyaprofessionalov. Sozdanieprilozhenydlyaplanshetnykhkompyuterov I smartfonov = Pro Android 4. M.: Vilyams. pp.880.

[24] Erik,Frimen, Elizabeth Frimen, Kate S'erra, Bert Beits, Patternyproektirovaniya, 2011. 1-st edition. Spb.: Saint-Petersburg.pp. 736.