УДК 004.043

Krivuliak V. V., Android developer

DEPENDENCY INJECTION IN ANDROID DEVELOPMENT

The cry of “You must do dependency injection!” is heard throughout modern software development teams. With such an imposing name, Dependency Injection (or DI for short) can put a fright into any software developer. Dependency Injection is a design pattern which is originally mentioned in Martin Fowler’s article. [3] It turns out that Dependency Injection is nowhere near as complex as its name implies, and is a key tool for building software systems that are *maintainable* and *testable*. In the end, relying on dependency injection will simplify your code quite a bit and also allow for a clearer path to writing testable code.

Any non-trivial software program is going to contain components that pass information and message calls back and forth between one another.
For example, when using an Object-Oriented Programming language (such as Java on Android), objects will call methods on other objects that they have references to. A *dependency* is simply when one of the objects depends on the concrete implementation of another object. [2]
From a practical perspective in Java code, you can identify a dependency in your code whenever you use the new keyword to instantiate one object within another. In such a case, you are fully responsible for creating and properly configuring the object that is being created. For example, in the following class A:

```
public class A {
  private B b;
  public A() {
    b = new B();
  }
}
```

An A instance creates its b field in its constructor. The A instance is fully dependent on the concrete implementation of B and on configuring the b field in order to use it properly.
This presents a *coupling* or *dependency* of the A class on the B class. If the setup of a B object is complex, all of that complexity will be reflected within the A class as well. Any changes necessary to configure a B object will have to be made within class A.
Should the B class depend itself on class C, which in turn depends on class D, all of that complexity will propagate throughout the code base and cause a tight coupling between the components of the application.

Dependency Injection is the term used to describe the technique of loosening the coupling just described. In the simple example above, only one tiny change is needed:

```
public class A {
  private B b;
  public A(B b) {
    this.b = b;
  }
}
```

That’s dependency injection at its core! Rather than creating the b object in the A constructor, the b object is passed into or injected into A ‘s constructor. The responsibility for configuring b is elsewhere, and the A class is simply a consumer of the B class.

DI is often discussed in conjunction with one of the five SOLID principles of Object-Oriented Design: The Dependency Inversion principle. [1]
The gist of the Dependency Inversion principle is that it is important to depend on abstractions rather than concrete implementations. In the simple example above, this means changing B to a Java interface rather than a Java class. With this change, many different types of concrete B type objects that adhere to the B interface can be passed into the A constructor. This presents several key benefits:

- Allows for the A class to be tested with various kinds of B objects.
- Mock B objects can be used as needed in certain test scenarios.
- Testing of A is independent of the implementation of B.

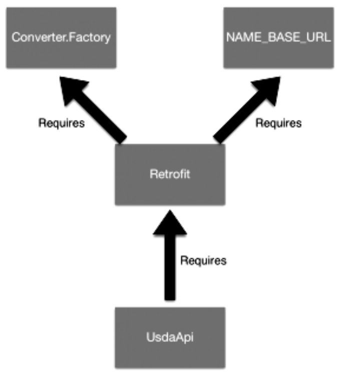
While often discussed together, dependency injection and the Dependency Inversion principle are not the same. Essentially, dependency injection is a technique that is used as part of adhering to the Dependency Inversion Principle.

In the Java world, there are a number of frameworks that have been created to simplify the application of dependency injection. The frameworks remove a lot of the boilerplate code that can occur, and also provide a systematic way to apply dependency injection across a software system.

The team at *Square* developed the Dagger framework, targeted primarily at Android. While a fantastic accomplishment, the initial Dagger framework had a few downsides. For example, performance issues due to runtime reflection and difficulty working with ProGuard.

As a result, the updated Dagger 2 processor was born, which produces simpler generated code and solves the performance issues by having injection occur at compile time.

The name "Dagger" is inspired in part by the nature of dependencies in software development. The web of dependencies that occur between objects such as A, B, C, …, create a structure called a *Directed Acyclic Graph*. Dagger and Dagger 2 are used to simplify the creation of such graphs in your Java and Android projects.



In a typical app that uses Dagger 2 and Retrofit together, Retrofit will be provided by dependency injection.

Here you will see some of the many advantages of using dependency injection and Dagger 2, including:

- Eliminating code duplication.
- Eliminating the need for dependency configuration.
- Automatic construction of a dependency graph.

Sources

1. Joe Howard, [Web – resource], DI injection with Dagger 2 – access https://www.raywenderlich.com/146804/dependency-injection-dagger-2 open access.
2. Quang Nguyen, [Web – resource] Android MVP architecture with DI – access https://android.jlelse.eu/android-mvp-architecture-with-dependency-injection-dee43fe47af0 open access.
3. Martin Fowler, [Web – resource] Inversion of Control Containers and the Dependency Injection pattern – access https://www.martinfowler.com/articles/injection.html open access.