How we use no-container dependency injection to develop testable components in our modularized Android app

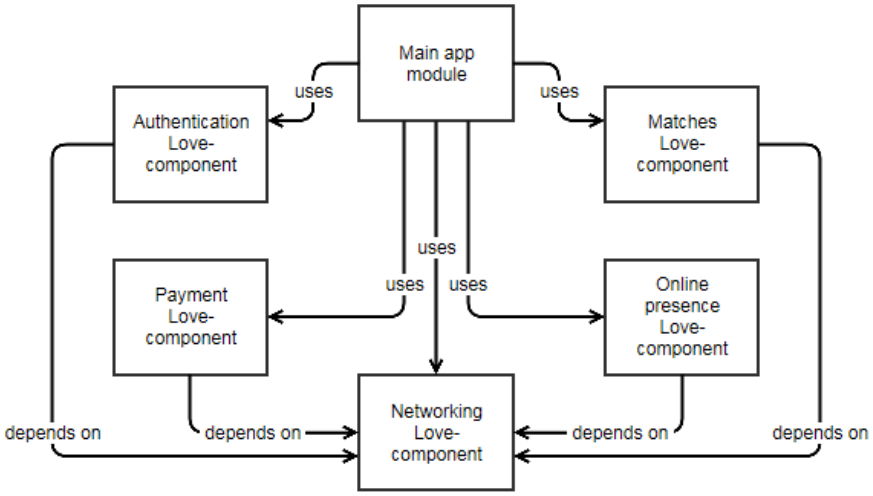
March 28, 2019

Dependency injection frameworks, like dagger2, use containers to instantiate and configure objects. But having a DI container at Android library level can bring several ugly consequences. That's why it's worth looking at no-container alternatives.

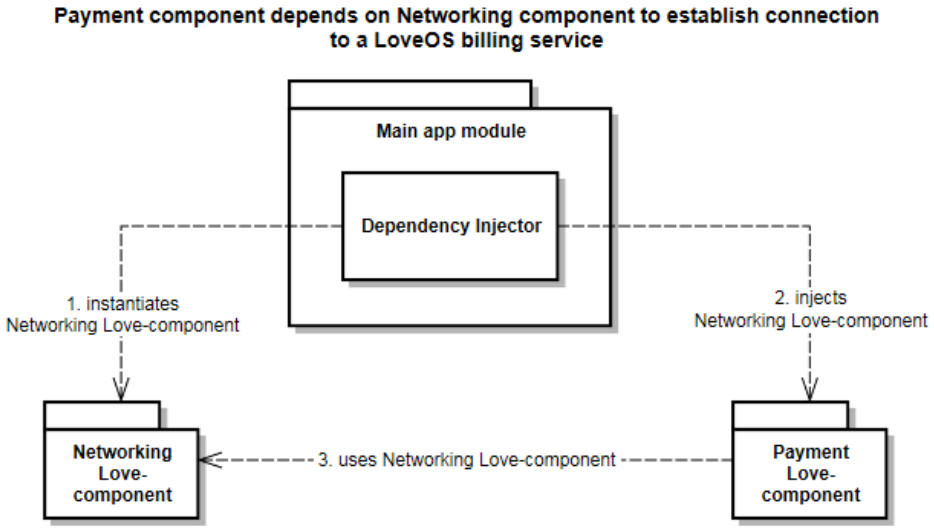
A large percentage of Spark's tech team is working on **loveOS,** a new platform for dating services that is built upon vertical components. Its modular approach means that components can be added at will to our various platforms, according to required function. Whether it be chat, conversion funnel, matchmaking or open search, building the individual vertical components of **loveOS** will allow the quick shipping of new features across platforms.

Every **loveOS** frontend part, web, iOS and Android, consists of components that talk to **loveOS** endpoints. Components represent features, adjustable and interchangeable, so we build each brand from the same codebase using only necessary features in a brand's own colors.

Therefore, Android **loveOS** app uses a lot of components. Each of them is represented by a separate module and is built with SOLID principles in mind, so it is, from the architectural point of view, a closed module. Also, each component is covered with tests, JUnit fur business logic and Espresso for user interaction.



A component relies on other components, let's call them love-components to distinguish, to ensure high cohesion and low coupling. That's why, for example, payment component only knows the name of an endpoint of the billing service but does not know how to establish a secure connection; instead, it depends on the other component that is responsible for handling the networking inside the **loveOS** platform.

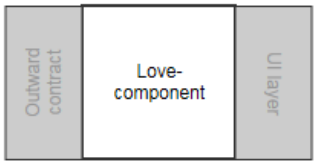


How do we provide these dependencies to the love-component? Of course, with a dependency injection. It is a technique that decouples the usage of an object from its creation, simplifies testing and makes managing dependencies easier. Any love-component is built as a part of the framework, and that means implemented Inversion of Control principle. Specifically, it means that each component provides a feature or a service and an interface that should be used to communicate with that service. To implement dependency injection there, we needed only an injector.

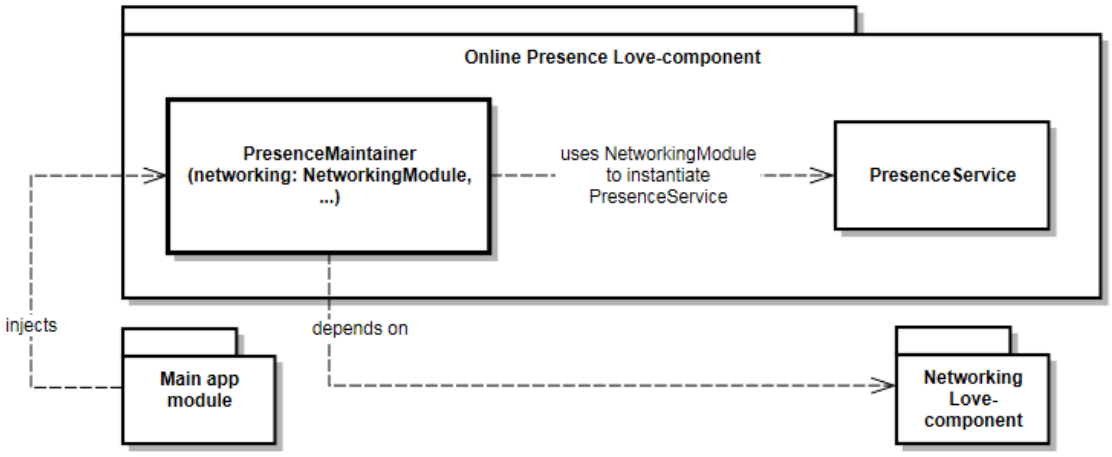
As we already used a DI framework in the main app, dagger2, naturally, the first option was to treat every love-component as a dagger2 subcomponent. Though, we found out a severe problem: subcomponents always have only one parent component that they inherit from and extend the object graph. But the parent component should know about all the subcomponents that are used in the project; otherwise, a runtime error would occur. As a number of love-components used in the built app depend on the brand configuration and is decided during build time, going the subcomponent way can bring ugly consequences, weird constructions to auto-generate code or unwanted circle dependencies. And, thus, we came to the conclusion that a DI framework should only be used on the application level, while on a component's level we should implement a no-container option.

Let's elaborate more on the love-component structure. There are two types of components: service- and UI-based. Service component does not include any graphical representation, e.g., online presence component that is responsible for polling the server with heartbeats once a customer logs in in the app. UI components, as the name implies, do include graphical representation. Imagine a user looking through the list of matches and noticing a couple of very interesting profiles. Then they go to the profile to get more detailed information about a person and see a "wave" button there. A Wave is a non-binding way to start an interaction with another user. Its widget is a part of the engagement component, named so because it propagates user engagement within the app. The engagement component now consists of a wave widget but can be further extended with other widgets serving the same purpose, like smiles or likes.

A typical component can have up to two external points of interaction: the outward contract and the UI layer.



The component provides the outward contract to its clients. These are usually interfaces to implement and resources to override [and works as an interface from the injection scheme for other love-components – depends on illustrations]. The online presence component mentioned earlier provides a contract in the form of a class that the main app can instantiate. And the main app creates a singleton instance of this class right when the user registers or logs in using app's DI framework of choice.




```
open class PresenceMaintainer(
    networkingModule: NetworkingModule,
) : Application.ActivityLifecycleCallbacks {
    internal val presenceService =
            networkingModule.createSecureServiceEndpoint(PresenceService::class.java)
    private val foregroundSubject = PublishSubject.create<Boolean>()
    private val polling = presenceService.sendHeartbeat()
            .subscribeOn(Schedulers.io())

    private var connected by Delegates.observable(false) { ... }

    init {
        foregroundSubject
 .debounce(TIMEOUT, TimeUnit.SECONDS)
                .distinctUntilChanged()
                .switchMap {
                    if (it && connected) polling.toObservable<Unit>() else Observable.never()
                }
                .subscribe()
    }

    override fun onActivityPaused(activity: Activity?) {
        foregroundSubject.onNext(false)
    }

    override fun onActivityResumed(activity: Activity?) {
        foregroundSubject.onNext(true)
    }

    override fun onActivityStarted(activity: Activity?) = Unit
    override fun onActivityDestroyed(activity: Activity?) = Unit
    override fun onActivitySaveInstanceState(activity: Activity?, outState: Bundle?) = Unit
    override fun onActivityStopped(activity: Activity?) = Unit
    override fun onActivityCreated(activity: Activity?, savedInstanceState: Bundle?) = Unit
}
```

Here we use a constructor injection method when the dependencies are provided through a class constructor

```
@Provides
@Singleton
open fun providePresenceMaintainer(networkingModule: NetworkingModule) =
PresenceMaintainer(networkingModule)
```

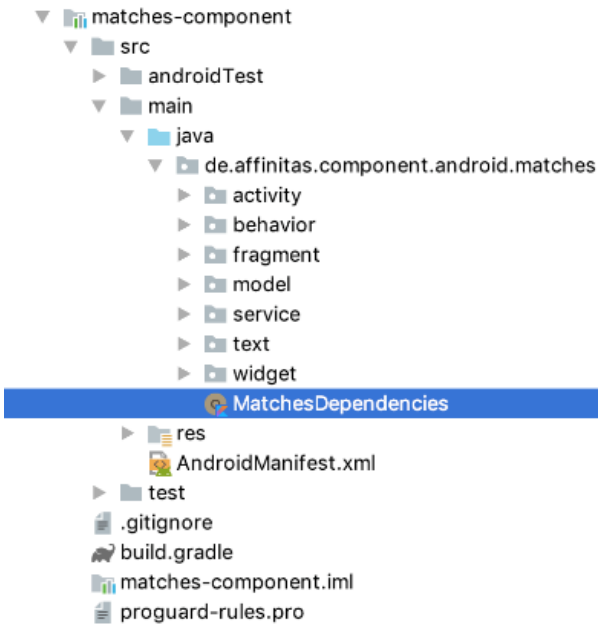
This way it's easy to mock the component's dependencies during testing, both in the component and in the main app.

```
@Test
internal fun presenceService_askedForFrequency() {
    val heartbeat = PublishSubject.create<Presence>()
    val mockPresenceService = mock<PresenceService> {
 on { sendHeartbeat() } doReturn heartbeat
    }
 val mockNetworkingModule = mock<NetworkingModule> {
 on { createSecureServiceEndpoint(PresenceService::class.java) }
 .doReturn(mockPresenceService)
    }
 val presenceMaintainer = PresenceMaintainer(mockNetworkingModule)
    heartbeat.onNext(Presence(1))
    presenceMaintainer.onActivityResumed(any())
    eventually { verify(mockPresenceService).sendHearbeat(true) }
}
```

The UI layer exists only in UI-components and consists of Views, Fragments, and Activities. Let's call them external because despite the fact that app instructions start them, the objects' creation is beyond the app's control. In service components we injected dependencies into a constructor in the main app's module. But here it's impossible to satisfy dependencies (if any) of these objects at the time of object's creation. Therefore, we decided to provide a dependency object for a love-component that will contain a method to initialize the dependencies before the component starts and also a method to provide the implementation of the outward contract. For the sake of simplicity, we decided that a life scope of this dependency object and all of its properties should be as long as the life scope of the component, so dependencies would be available from the moment of the love-components creation till its destruction.

Such a simple approach is, of course, not 100% perfect. It is, indeed, straightforward, readable and easy to implement. It can be incorporated smoothly in any DI framework (with or without a container) that is used in the main app without any constraints to use the dagger2, and it is easy to understand even for a person not involved in the project. However, directness of the method turns into excessive verboseness, and, thus, results in slightly bigger amount of maintaining code, both production and tests. But let's take a closer look at how we implemented this technique in one of our most important components; the matches component.

Matchmaking is a core part of any dating app, it is a reason why the user installed the app. Showing matches in a correct way is tremendously important, whether it's a classic grid or modern like and swipe. LoveOS matches component collection currently consists of a matches component with a highly customizable grid. As a love-component, it depends on the networking love-component and the implementation of an outward contract by a main app module. Dependencies are provided by a dependencies object, called MatchesDependencies, that has a very straightforward API: one method to initiate dependencies and one method to provide an implementation of an outward contract.




```
object MatchesDependencies {
    internal lateinit var matchMakingService: MatchMakingService
        private set

 fun init(networkingModule: NetworkingModule) {
        matchMakingService =
                networkingModule.createSecureServiceEndpoint(MatchMakingService::class.java)
    }
```

```
    internal lateinit var matchesBehavior: MatchesBehavior

    fun setBehavior(behavior: MatchesBehavior) { //behavior can be changed throughout the life scope of
the matches component
        matchesBehavior = behavior
    }
}
```

One can argue that properties in the dependencies object can be instantiated from the main app directly, outside the initiation method, but in that way not only would we violate encapsulation and Open-closed principle, the client would also have to track what dependencies should be satisfied themselves and, what's more, it would be impossible for the compiler to check those dependencies at compile time. The MatchesDependencies object is used in Activities and Fragments inside the matches component, as well as its tests.

```
class MatchListFragment : Fragment() {
    private val matchMakingService = MatchesDependencies.matchMakingService

 ...

}
```

Dependencies are referenced directly, so it's vitally necessary to resolve them before the Activity or Fragment is created. This also applies to tests: dependencies should be replaced by mocks or stubs before the activity under test is created.

```
class UsersListTabFragmentTest {
    val matchesService = mock<MatchMakingService> {
 on { matches(MatchMakingService.MatchMakingSearchType.Active) } doReturn ...
 on { matches(MatchMakingService.MatchMakingSearchType.New) } doReturn ...
 val networkingModule = mock<NetworkingModule> {
 on {
 createSecureServiceEndpoint(MatchMakingService::class.java) //we use retrofit for creating endpoints
        } doReturn matchesService
 }

 @Rule
 @JvmField
 val activityRule = object : ActivityTestRule<MainActivity>(MainActivity::class.java) {
        override fun beforeActivityLaunched() {
            MatchesDependencies.init(networkingModule)
             ...
    }

 }
```

Using this approach to build love-components, we develop them as testable as if we were using a DI framework with a container and leave ourselves freedom to implement any of a wide range of DI frameworks on the level of the main app instructions. So, if we'd decide tomorrow to use Koin instead of dagger2, we'll only have to make changes in one module and won't be required to change all love-components we now have thus leveraging the potential of **loveOS.**

- Margarita Litkevych

Android Developer