Готовим DI

Петропавлов Глеб

```kotlin
class Engine {
    fun start() {
        println("Врум-врум")
    }
}

class Car {
    private val engine: Engine = Engine()
    fun start() {
        engine.start()
    }
}

fun main(args: Array<String>) {
    val car = Car()
    car.start()
}
```

Автомобиль сам управляет созданием двигателя

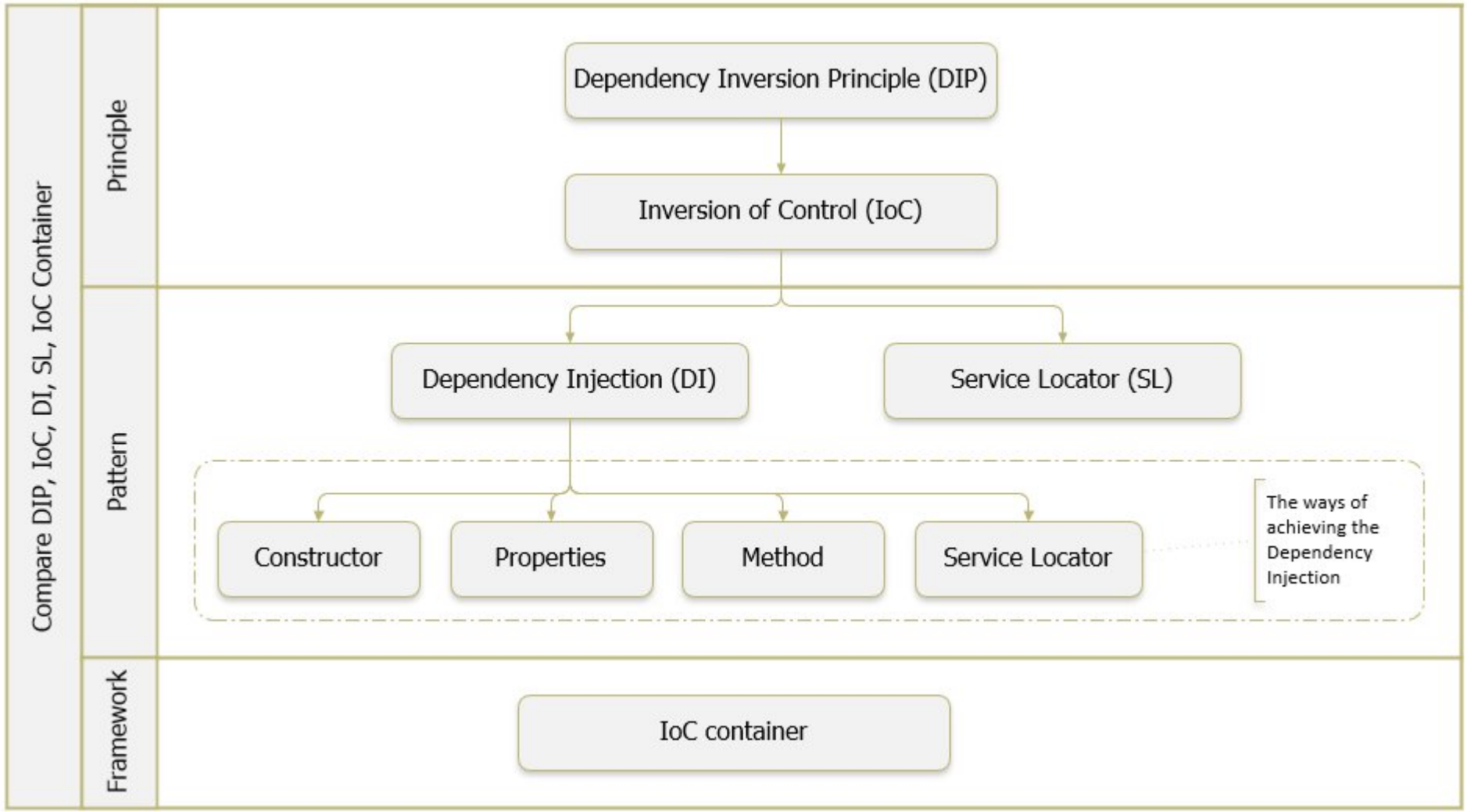
Мы можем создавать машины только с одним видом двигателя

Из-за сильной связанности сложно тестировать

IoC

DI

ServiceLocator


Compare DIP, IoC, DI, SL, IoC Container
Principle
Dependency Inversion Principle (DIP)
Inversion of Control (IoC)
Pattern
Dependency Injection (DI)
Service Locator (SL)
Constructor
Properties
Method
Service Locator
The ways of achieving the Dependency Injection
Framework
IoC container

ServiceLocator - Это некий реестр, который по запросу может предоставить нужный объект

```
object ServiceLocator {
  fun <reified T> register(factory: () -> T) { ... }
  fun <reified T> resolve(): T { ... }
}
```

```
interface Engine
class DefaultEngine: Engine

class Car {
  private val engine: Engine = ServiceLocator.resolve()
}

fun main() {
  ServiceLocator.register<Engine> { DefaultEngine() }
  val car = Car()
}
```

Класс Car получает зависимость извне, не зная о реализации

Мы ничего не знаем о зависимостях класса Car, и есть ли они вообще

Немного улучшений

```
interface ServiceLocator {
  fun <reified T> register(factory: () -> T)
  fun <reified T> resolve(): T
}

class DefaultServiceLocator: ServiceLocator { ... }

class Car(private val serviceLocator: ServiceLocator) {
  private val engine = serviceLocator.resolve<Engine>()
}

fun main() {
  val serviceLocator = DefaultServiceLocator()
  serviceLocator.register<Engine> { DefaultEngine() }
  val car = Car(serviceLocator)
}
```

Знаем, что у Car
есть зависимости

Но всё ещё не
знаем какие

Еще немного улучшений

```
class Car(private val engine: Engine)

fun main() {
  ServiceLocator.register<Engine> { DefaultEngine() }
  val car = Car(ServiceLocator.resolve<Engine>())
}
```

Знаем что у Car есть зависимости

Знаем какие

Подставляем их извне

И мы пришли к Dependency Injection (DI) в чистом виде

Dependency Injection

Field Injection (or Setter Injection)

```
3   class Engine {
4       fun start() {}
5   }
6
7   internal class Car {
8       private var engine: Engine? = null
9
10      fun setEngine(engine: Engine?) {
11          this.engine = engine
12      }
13
14      fun start() {
15          engine?.start()
16      }
17  }
18
19  fun main(args: Array<String>) {
20      val engine = Engine()
21      val car = Car().apply { this: Car
22          setEngine(engine)
23      }
24      car.start()
25  }
```

Constructor Injection

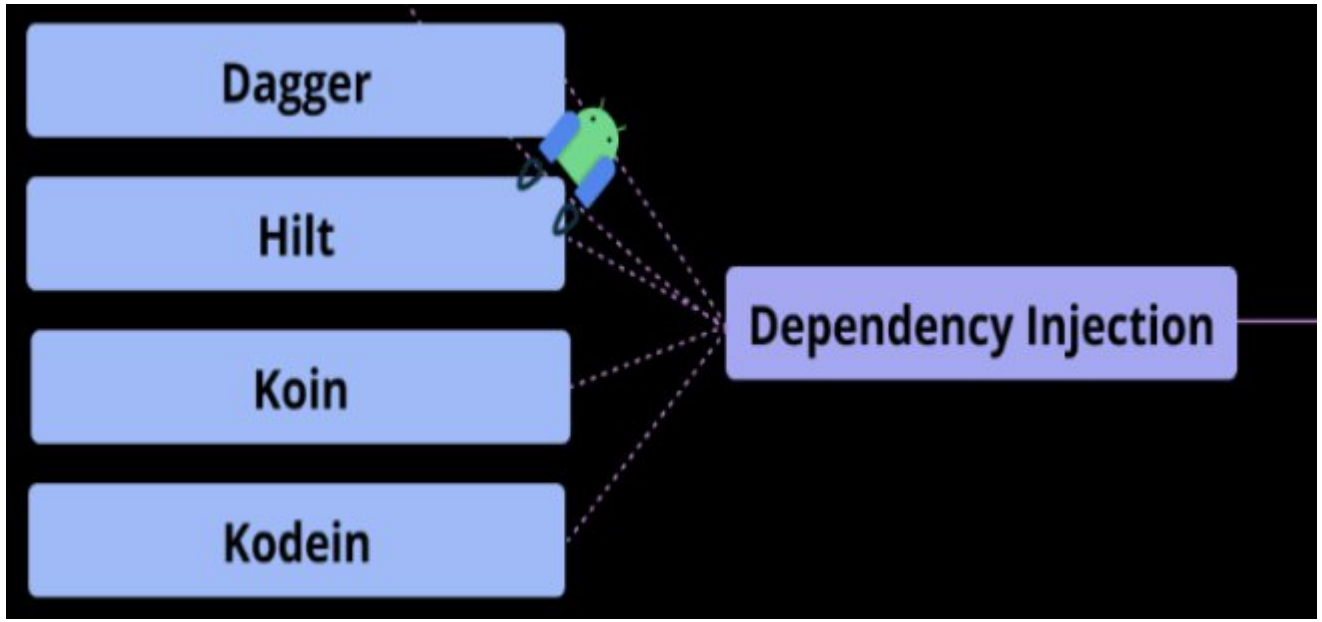
```
3   class Engine {
4       fun start() {}
5   }
6
7   class Car(private val engine: Engine) {
8       fun start() {
9           engine.start()
10      }
11  }
12
13  fun main(args: Array<String>) {
14      val engine = Engine()
15      val car = Car(engine)
16      car.start()
17  }
```

ElectricEngine & GasEngine

```
interface Engine {
    fun start()
}

class ElectricEngine : Engine {
    override fun start() {
        println("вшшшшуууус")
    }
}

class GasEngine : Engine {
    override fun start() {
        println("врррррууум")
    }
}
```

```
class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array<String>) {
    val electricCar = Car(ElectricEngine())
    val gasCar = Car(GasEngine())
    electricCar.start()
    gasCar.start()
}
```

Автоматизация DI



```kotlin
@Module
class EngineModule {

    @Provides
    @Named( name: "Gas")
    fun provideGasEngine(): Engine {
        return GasEngine()
    }

    @Provides
    @Named( name: "Electric")
    fun provideElectricEngine(): Engine {
        return ElectricEngine()
    }
}
```

```kotlin
@Module
class CarModule {

    @Provides
    @Named( name: "Electric")
    fun provideElectricCar(
        @Named( name: "Electric") engine: Engine,
    ): Car {
        return Car(engine)
    }

    @Provides
    @Named( name: "Gas")
    fun provideGasCar(
        @Named( name: "Gas") engine: Engine,
    ): Car {
        return Car(engine)
    }
}
```

```kotlin
@Component
interface CarComponent {

    fun inject(taxiPark: TaxiPark)

    fun provideElectricCar(): Car
}
```

```kotlin
fun main() {
    val taxiPark = TaxiPark()
    DaggerCarComponent.Builder().build().inject(taxiPark)
    taxiPark.taxiOrdered()
}
```

```kotlin
class TaxiPark {
    @Inject
    @Named( name: "Electric")
    lateinit var electricCar: Car

    @Inject
    @Named( name: "Gas")
    lateinit var gasCar: Car

    fun taxiOrdered() {
        if (electricCar.isReady()) {
            electricCar.start()
        } else if (gasCar.isReady()) {
            gasCar.start()
        } else {
            println("Нет свободных машин")
        }
    }
}
```

Hilt

```
plugins {
  ...
  id 'com.google.dagger.hilt.android' version '2.44' apply false
}
```

```
plugins {
  id 'kotlin-kapt'
  id 'com.google.dagger.hilt.android'
}

android {
  ...
}

dependencies {
  implementation "com.google.dagger:hilt-android:2.44"
  kapt "com.google.dagger:hilt-compiler:2.44"
}

// Allow references to generated code
kapt {
  correctErrorTypes true
}
```

Inject dependencies into Android classes

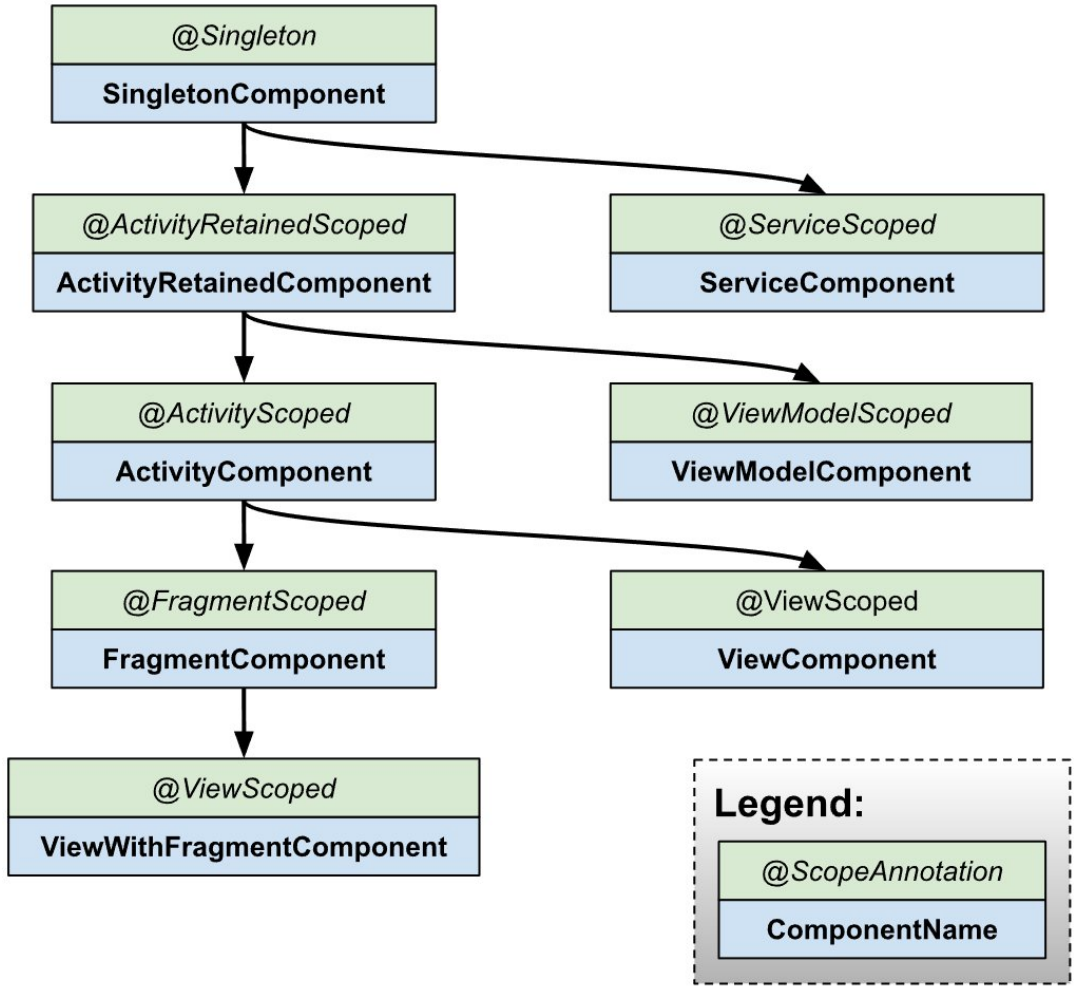
```
@HiltAndroidApp
class ExampleApplication : Application() { ... }
```

```
@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() { ... }
```

- `Application` (by using `@HiltAndroidApp`)
- `ViewModel` (by using `@HiltViewModel`)
- `Activity`
- `Fragment`
- `View`
- `Service`
- `BroadcastReceiver`

Componentы в Hilt

Hilt component	Injector for
`SingletonComponent`	`Application`
`ActivityRetainedComponent`	N/A
`ViewModelComponent`	`ViewModel`
`ActivityComponent`	`Activity`
`FragmentComponent`	`Fragment`
`ViewComponent`	`View`
`ViewWithFragmentComponent`	`View` annotated with `@WithFragmentBindings`
`ServiceComponent`	`Service`



Singleton
&
Local Singleton

Паттерн Singleton

Singleton - порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

```
enum class ColorEnum {
    RED,
    BLUE
}
```

```
object Color {
    val RED = ColorEnum.RED
    val BLUE = ColorEnum.BLUE
}
```

Double Check Locking Singleton

```
public class DclSingleton {
    private static volatile DclSingleton instance;
    public static DclSingleton getInstance() {
        if (instance == null) {
            synchronized (DclSingleton .class) {
                if (instance == null) {
                    instance = new DclSingleton();
                }
            }
        }
        return instance;
    }

    // private constructor and other methods...
}
```

Доп материал

Dagger Android - https://dagger.dev/dev-guide/android.html

Dependency injection with Hilt - https://developer.android.com/training/dependency-injection/hilt-android

Правильный Singleton в Java - https://habr.com/ru/articles/129494/

Далее для совершенствования потребуется: Много практики, гугления, изучения

Вопросы

Поправить код в соответсвие с DI

```
fun main() {
    AuthLogic().login("login", password: "password")
}


class AuthLogic {
    val authMethod = GoogleAuth()

    fun login(login: String, password: String) {
        authMethod.login(login, password)
    }
}

class GoogleAuth {
    fun login(login: String, password: String) {
        println("LOGGED WITH GOOGLE")
    }
}
```

Спасибо за внимание

SOLID

Android это все еще ООП

SOLID — это аббревиатура, помогающая определить пять основных принципов объектно-ориентированного проектирования:

- **Single Responsibility** – Принцип единой ответственности
- **Open-Closed Principle** – Принцип открытости-закрытости
- **Liskov Substitution Principle** – Принцип подстановки Лисков
- **Interface Segregation** – Принцип разделения интерфейса
- **Dependency Inversion Principle** – Принцип инверсии зависимостей

Принцип инверсии зависимостей

- A. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- B. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.