Uniwersytet Wrocławski

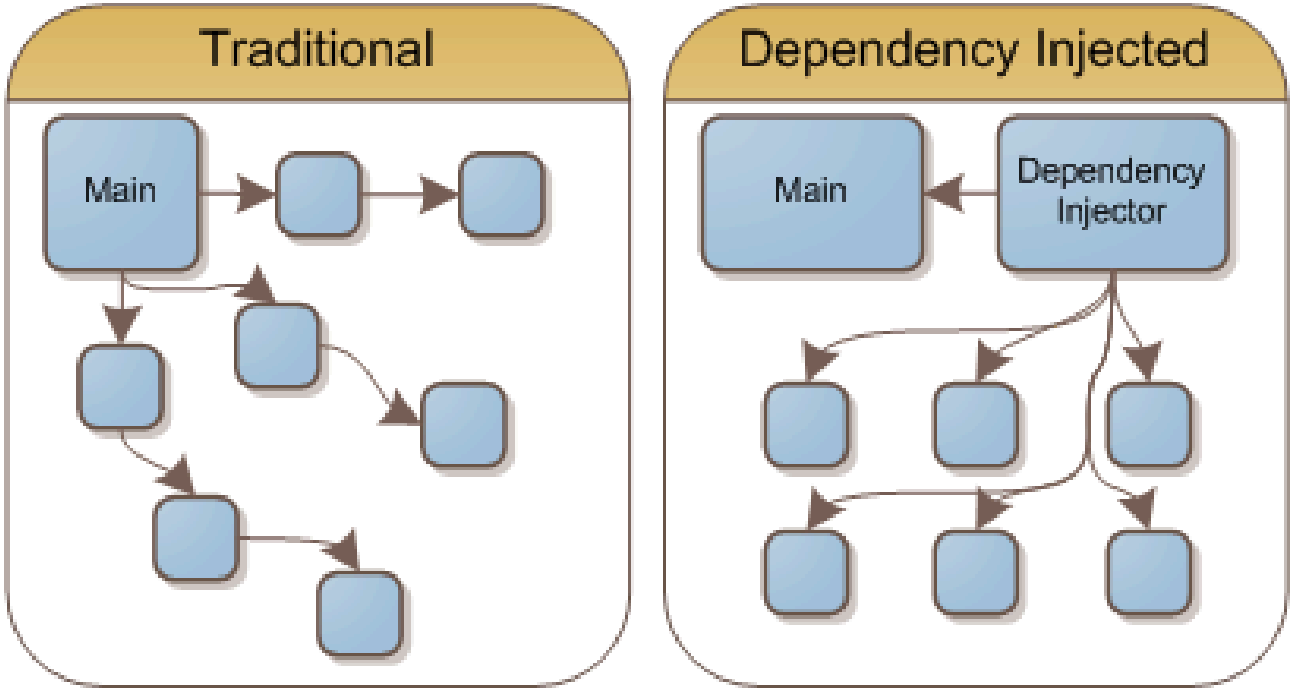
Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 14

- Wstrzykiwanie zależności
- Dagger-Hilt

Dependency Injection



https://www.google.com/url?sa=i&url=https%3A%2F%2Fblog.stackademic.com%2Friverpod-inversion-of-control-dependency-injection-dependency-inversion-and-service-locator-24a1c6972ed6

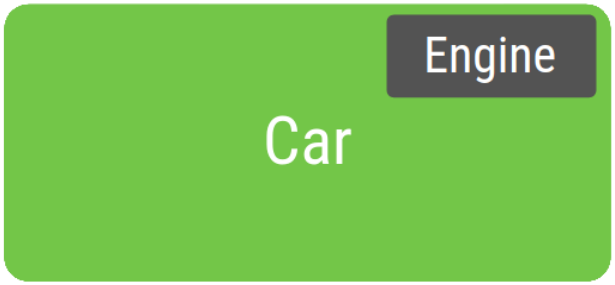
Dependency Injection

```
class Car {

    private Engine engine = new Engine();

    public void start() {
        engine.start();
    }
}


class MyApp {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
    }
}
```



Dependency Injection

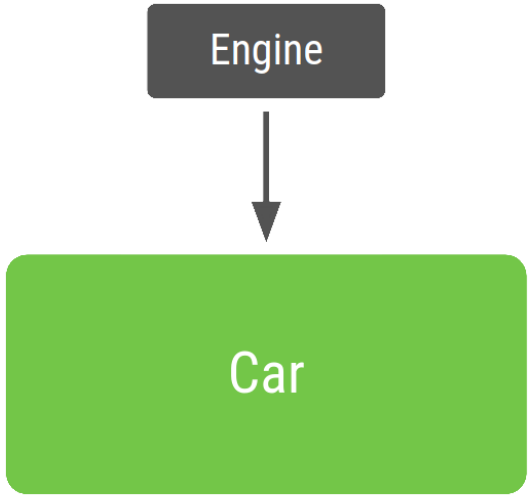
```
class Car {

    private final Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}


class MyApp {
    public static void main(String[] args) {
        Engine engine = new Engine();
        Car car = new Car(engine);
        car.start();
    }
}
```



Dependency Injection

```
class Computer {}
class Case {}
class CPU {}
class GPU {}
class Motherboard {}
class PowerSupply {}
```

```
class Computer (
    private val case: Case,
    private val gpu: GPU,
    private val cpu: CPU,
    private val motherboard: Motherboard,
    private val powerSupply: PowerSupply
        ) {
    fun work(): String{
        return "working"
    }
}
```

Dagger

```
class Case @Inject constructor()
class CPU @Inject constructor()
class GPU @Inject constructor()
class Motherboard @Inject constructor()
class PowerSupply @Inject constructor()
```

```
class Computer @Inject constructor (
    private val case: Case,
    private val gpu: GPU,
    private val cpu: CPU,
    private val motherboard: Motherboard,
    private val powerSupply: PowerSupply
        ) {
    fun work(): String{
        return "working"
    }
}
```

```
val component = DaggerComputerComponent.create()
computer = component.getComputer()
```

Dagger-Hilt

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

aktywuje generowanie kodu Hilt, w tym klasę bazową dla aplikacji, która pełni rolę kontenera zależności na poziomie aplikacji

Dagger-Hilt

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

aktywuje generowanie kodu Hilt, w tym klasę bazową dla aplikacji, która pełni rolę kontenera zależności na poziomie aplikacji

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

generuje indywidualny komponent Hilt

Dagger-Hilt

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

```
<application
    android:name=".MyApp"
    ...
```

Dagger-Hilt

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {}
```

- Adnotacja ta jest używana do oznaczania klas, które definiują jakie obiekty zależności powinny zostać utworzone i jak one powinny zostać połączone ze sobą.
- Klasy oznaczone adnotacją @Module to tzw. moduły, które służą do definiowania zależności i określają jakie obiekty powinny być tworzone w celu ich dostarczenia.

@InstallIn - deklaruje do których komponentów opisana klasa/obiekt powinna zostać dodana podczas generacji obiektów przez Hilt.

Dagger-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`

Dagger-Hilt

```
@Provides
@Singleton
fun provideAppRepository(api: PlaceholderApi) : AppRepository{
    return AppRepositoryImpl(api)
}
```

@Singleton - jeżeli PlaceholderApi zostanie wstrzyknięty do kilku klas, będzie to ta sama instancja. Bez tej adnotacji, przy wstrzykiwaniu do kilku klas, za każdym razem tworzona jest nowa instancja.

```
@HiltViewModel
class AppViewModel @Inject constructor (
    private val repository: AppRepository
) : ViewModel() {
```