concrete solutions

Dagger 2

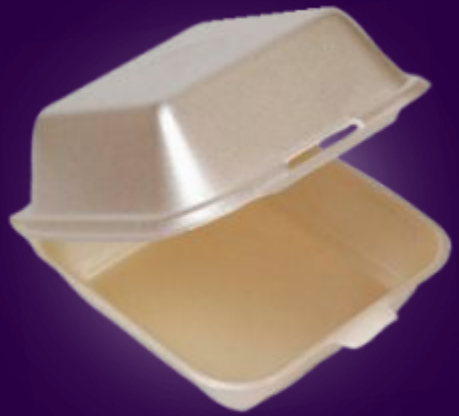
uso avançado em projetos Android

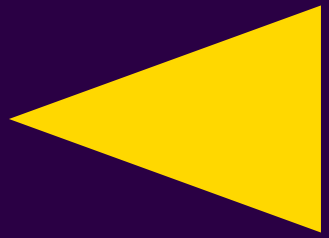
DI - Dependency Injection

Não crie, peça

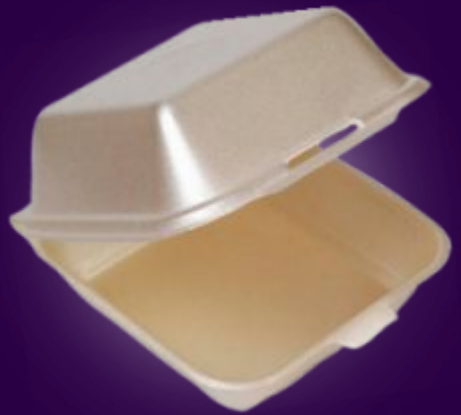
Dependa de abstrações, não dependa de classes concretas

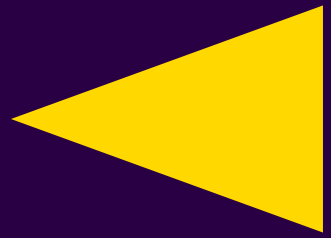
Abstração x Concreto





Abstração x Concreto





AVISO!

Cuidado com a quantidade de interfaces!

concrete solutions

Dagger 2

Qualifiers

Qualifiers - por quê?

Diferenciação de dependências de mesmo tipo / classe

Qualifiers - como?

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface ApiUrl {

}
```

```java
@Module
public class QualifiedModule {

    @Provides
    @ApiUrl
    public String provideApiUrl() {
        return "https://api.stackexchange.com/2.2/";
    }

    @Provides
    public String provideUnqualified() {
        return "Android!";
    }
}
```

```
public class InjectedClass {

    @Inject
    String value;

    @Inject
    @ApiUrl
    String url;

    ...
```

```java
@Module
public class AnotherModule {

    @Provides
    public MyClass provideClass(String value,
            @ApiUrl String url) {

        return new MyClass(value, url);
    }

    ...
```

Qualifiers - extras

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Named {

    String value() default "";
}
```

```java
@Module
public class QualifiedModule {

    @Provides
    @Named("One")
    public String provideV1() {
        return "one!";
    }

    @Provides
    @Named("Two")
    public String provideV2() {
        return "two!";
    }
}
```

```
public class InjectedClass {

    @Inject
    @Named("One")
    String value1; // one!

    @Inject
    @Named("Two")
    String value2; // two!
    ...
```

Qualifiers - extras

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface ComplexQualifier {

    String value() default "";

    int ordinal() default 1;
}
```

concrete solutions

Dagger 2

Custom Scopes

Custom Scopes - por quê?

Ciclo de vida de um objeto dentro de um escopo

É uma espécie de “Singleton local”

Custom Scopes - como?

```
@Scope
@Retention(RetentionPolicy.RUNTIME)
public @interface ActivityScope {

}
```

```java
@Module
public class ScopedModule {

    @Provides
    @ActivityScope
    public MyObject provideDependency() {
        // Singleton on this scope
    }

    @Provides
    public MyOtherObject provideOtherDependency() {
        // Each request, a new object
    }
}
```

Custom Scopes - como?

```
@ActivityScope
@SubComponent(modules = ScopedModule.class)
public interface ActivityComponent {

    ...

}
```

Custom Scopes - como?

```
@Singleton
@Component(...)
public interface MainComponent {

    ActivityComponent plus();
}
```

Custom Scopes - como?

```
@Singleton
@Component(...)
public interface MainComponent {

    ActivityComponent plus();

    // Shared dependency
    Application getApplication();
}
```

Custom Scopes - como?

```
@Singleton
@Component(...)
public interface MainComponent {

    // If module has no default constructor
    ActivityComponent plus(CustomModule m);
}
```

Custom Scopes - como?

```
ActivityComponent activityComponent =
    mainComponent.plus();

// ou
ActivityComponent activityComponent =
    mainComponent.plus(new MyModule(ctx));
```

Custom Scopes - importante

É importante sempre gerenciar o ciclo de vida do componente de acordo com o escopo desejado para evitar leaks

Explicitamente:

```
activityComponent = null;
```

Pro Tip: Escopo @Reusable

Escopo @Reusable

Otimização para dependências que podem ser reutilizadas mas não precisam necessariamente ser a mesma instância.

Muito útil para Utils e Helpers

Disponível a partir da versão 2.3

concrete solutions

Dagger 2

Multibindings

Multibindings - por quê?

Dependências que só fazem sentido juntas

São entregues juntas dentro de uma collection - Map ou Set

```java
@Module
public class MultibindingsSetModule {

    @Provides
    @IntoSet
    public String provideFirstValue() {
        return "One!";
    }

    @Provides
    @IntoSet
    public String provideSecondValue() {
        return "Two!";
    }
}
```

```
public class InjectedClass {

    @Inject
    Set<String> values; // One!, Two!

    ...
```

```java
@Module
public class MultibindingsMapModule {

    @Provides
    @IntoMap
    @StringKey("key1")
    public String provideFirstValue() {
        return "One!";
    }

    @Provides
    @IntoMap
    @StringKey("key2")
    public String provideSecondValue() {
        return "Two!";
    }
}
```

```
public class InjectedClass {

    @Inject
    Map<String, String> values;
    // key1 => One!
    // key2 =: Two!

    ...
```

Multibindings - Maps

As chaves podem ser:

IntKey
LongKey
StringKey
ClassKey

Multibindings - Dica

Caso a coleção possa, por algum motivo, ficar vazia, precisamos de um módulo abstrato declarando a coleção.

```
@Module
public abstract class MyModule {

    @Multibinds
    public abstract Set<String> possiblyEmpty();
}
```

concrete solutions

Dagger 2

Binds

Binds - por quê?

Servem para ligar interfaces a implementações dentro de um módulo abstrato

É um excelente aliado na criação de módulos de teste e na substituição de objetos reais por mocks

```java
@Module
public abstract class BindModule {

    @Binds
    abstract Bluetooth provideBluetooth(BluetoothImpl impl);

    ...
}

@Module
public class ConcreteModule {

    @Provides
    BluetoothImpl provideBluetoothImpl() {
        return new BluetoothImpl();
    }
}
```

```java
@Module
public abstract class BindTestModule {

    @Binds
    abstract Bluetooth provideBluetooth(BluetoothMock mock);

    ...
}

@Module
public class ConcreteTestModule {

    @Provides
    BluetoothMock provideBluetoothMock() {
        return new BluetoothMock();
    }
}
```

concrete solutions

Dagger 2

Producers

Producers - por quê?

São praticamente uma biblioteca à parte - de fato, são uma dependência à parte :)

É uma API para a injeção de dependência de forma assíncrona, ideal para dependências muito pesadas ou de inicialização lenta.

Producers - avisos

Não segue o padrão da JSR-330 - não usa anotações `@Inject`, por exemplo

Adiciona o Guava ao classpath do projeto, o que pode engordar o APK e contribuir para aumento do número de métodos... #MultidexDanger

Producers - passo 1 (Executor)

```
@Module
public class ExecutorModule {

    @Provides
    @Production
    public Executor provideExecutor() {
        return Executors.newCachedThreadPool();
    }
}
```

Producers - passo 2 (ProducerModules)

```
@ProducerModule
public class MyProducerModule {

    @Produces
    public HeavyDep produceHeavyDep() {
        // Sempre Singleton
        return new HeavyDep();
    }
}
```

Producers - passo 3 (ProductionComponent)

```
@ProductionComponent(
    modules = { MyProducerModule.class,
                ExecutorModule.class }
)
public interface ProducerComponent {

   ListenableFuture<HeavyDep> heavyDep();
}
```

Producers - passo 4

```
ProducerComponent component = ...
Futures.addCallback(component.heavyDep(),
    new FutureCallback() {
        @Override
        public void onSuccess(HeavyDep dep) {}

        @Override
        public void onFailure(Throwable t) {}
});
```

Producers - observações

Os Production Components podem depender de Components convencionais, e podem conter módulos que fornecem dependências pela anotação `@Provides`

concrete solutions

Dagger 2

Testing Tricks

Trick #1 - Só APT

```
compile 'com.google.dagger:dagger:2.6.1'

androidTestApt 'com.google.dagger:dagger-compiler:2.6.1'

testApt 'com.google.dagger:dagger-compiler:2.6.1'
```

Trick #2 - Facilidade de Mock

Crie o módulo recebendo as dependências *mockadas* no construtor. Assim, fica mais fácil controlar o objeto dentro de um teste!

```java
@Module
public class ConcreteTestModule {

    private BluetoothMock btMock;

    public ConcreteTestModule(BluetoothMock mock) {
        this.btMock = mock;
    }

    @Provides
    public Bluetooth provideBluetoothMock() {
        return btMock;
    }
}
```

```
@Before
public void setUp() {
    Bluetooth bt = mock(Bluetooth.class);
    MyComponent testComponent =
      DaggerTestComponent.builder()
        .concreteTestModule(new ConcreteTestModule(bt))
        .build();

    // RuntimeEnvironment.application
    // ou
    // InstrumentationRegistry.getTargetContext()
    //     .getApplicationContext()
    app.setComponent(testComponent);
}
```

Mais Dagger 2

google.github.io/dagger

frogermcs.github.io

medium.com/android-dev-br

Slack #dagger

concrete solutions

Ajudamos empresas a criar
produtos digitais de sucesso

@_rafaeltoledo

www.rafaeltoledo.net

blog.concretesolutions.com.br

concretesolutions.com.br/carreira

www.concretesolutions.com.br

Rio de Janeiro – Rua São José, 90 – cj. 2121
Centro – (21) 2240-2030

São Paulo - Av. Nações Unidas, 11.541
3º andar - Brooklin - (11) 4119-0449