今日から始める依存性の注入

First Time Dependency Injection

Keisuke Kobayashi

DroidKaigi 2019 / Room1, 2019/02/08 14:50~15:20

自己紹介

- Keisuke Kobayashi
- Twitter: @kobakei122
- GitHub: @kobakei
- Merpay, Inc. / Engineering Manager
- Studyplus, Inc. / 技術顧問（副業）

merpay

Studyplus

今日のテーマ

- DI初心者、または「雰囲気でDagger2を使っている人」に、DIとはどういうものか、何のために導入するのかを解説するセッションです
- 個別のDIコンテナの詳しい使い方には時間の関係上深入りしません

アジェンダ

- Dependency Injection(DI)とは？
  - Androidアプリ開発を例に紹介
- DIコンテナの基本的な使い方
  - Dagger2 & Koin
- DI導入後のテストの書き方

Dependency Injection (DI)とは？

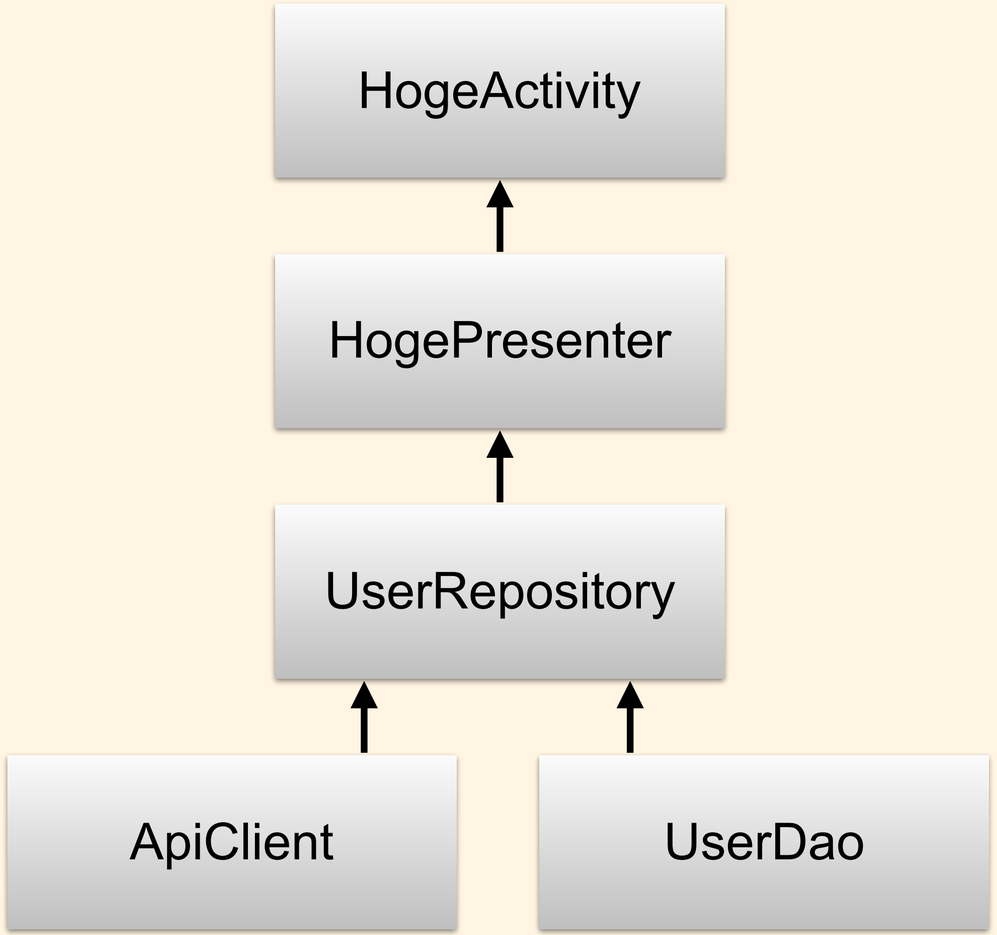
- コンポーネント間の依存関係をソースコードから排除し、外部から依存コンポーネントを注入させるデザインパターン
- 依存関係を排除することで、コンポーネント間が疎結合になる

DIのメリット

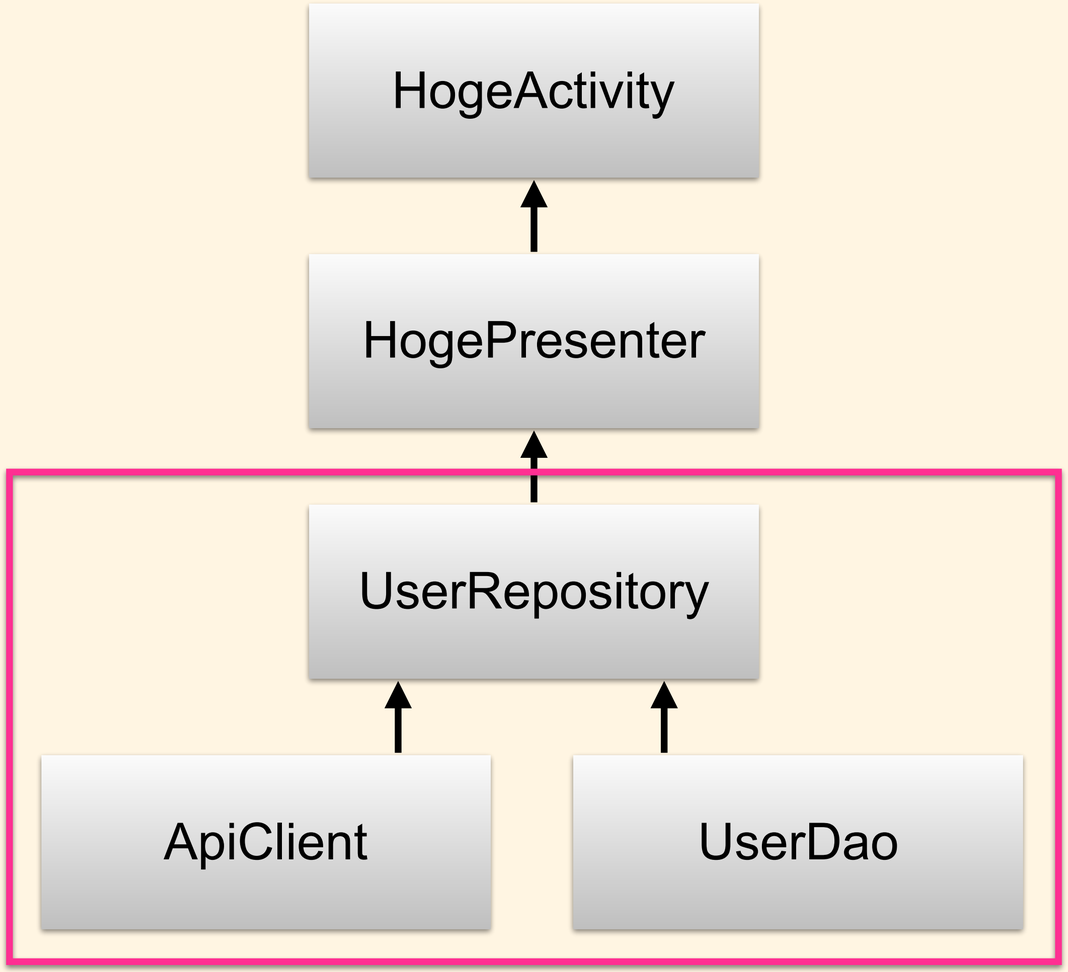
- コンポーネント間が疎結合になることで、以下のメリットがある
  - アプリケーションを拡張しやすくなる
  - テストが書きやすくなる
    - これは後で詳しく解説

Androidアプリ開発で
よくあるケースを考える

よくあるMVPの例


HogeActivity
HogePresenter
UserRepository
ApiClient
UserDao

よくあるMVPの例


HogeActivity
HogePresenter
UserRepository
ApiClient
UserDao

```
// DIを導入していないコード
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository {

  private val apiClient: ApiClient = ApiClientImpl()
  private val userDao: UserDao = UserDaoImpl()

  fun find(id:Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}

// UserRepositoryを使う側
val repo = UserRepository()
repo.find(123)...
```

```
// DIを導入していないコード
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository {

  private val apiClient: ApiClient = ApiClientImpl()
  private val userDao: UserDao = UserDaoImpl()

  fun find(id:Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}

// UserRepositoryを使う側
val repo = UserRepository()
repo.find(123)...
```

依存オブジェクトと

密結合している

```
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {

  fun find(id:Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}

// UserRepositoryを使う側
val apiClient = ApiClientImpl()
val userDao = UserDaoImpl()
val repo = UserRepository(apiClient, userDao)
repo.find(123)...
```

```
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {
  fun find(id:Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}

// UserRepositoryを使う側
val apiClient = ApiClientImpl()
val userDao = UserDaoImpl()
val repo = UserRepository(apiClient, userDao)
repo.find(123)...
```

コンストラクタで
依存オブジェクトを
受け取る

```
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {

  fun find(id:Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}
```

使う側が依存オブジェクトから組み立てている

```
// UserRepositoryを使う側
val apiClient = ApiClientImpl()
val userDao = UserDaoImpl()
val repo = UserRepository(apiClient, userDao)
repo.find(123)...
```

```
// 依存オブジェクトを解決するためのコンテナオブジェクト
object Container {

  val apiClient: ApiClient = ApiClientImpl()
  val userDao: UserDao = UserDaoImpl()

  val userRepo: UserRepository =
    UserRepository(apiClient, userDao)
}
```

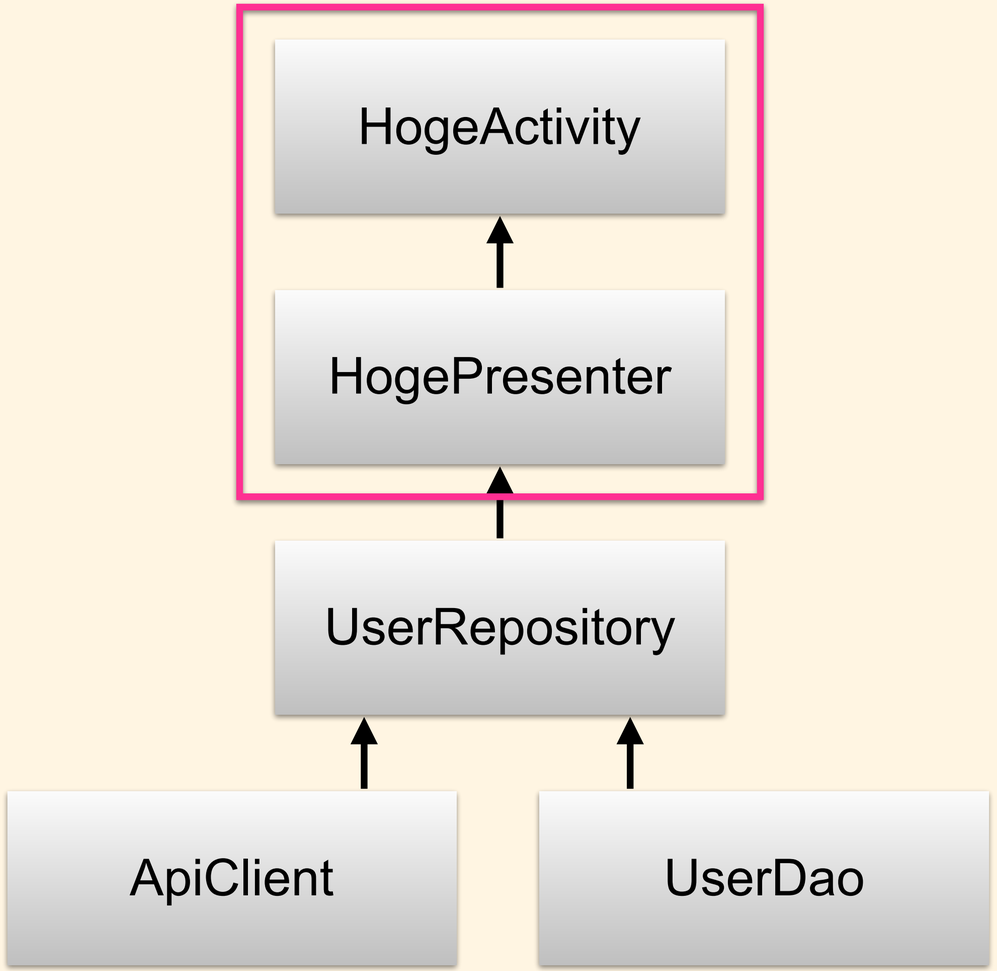
```
// ApiClientとUserDaoに依存したリポジトリクラス
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {

  fun find(id: Long): Single<User> =
    return if (userDao.cached) {
      userDao.find(id)
    } else {
      apiClient.getUser(id)
    }
}
```

```
// UserRepositoryを使う側
val repo = Container.userRepo
repo.find(123)...
```

UserRepositoryの依存関係が排除された

よくあるMVPの例


HogeActivity
HogePresenter
UserRepository
ApiClient
UserDao

Activityの場合

- コンストラクタでの注入ができない
- Fragment, Service, BroadcastReceiverなども同じ

```
// DIを導入していないコード
class HogeActivity : AppCompatActivity() {

  private val presenter = HogePresenter()

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

Activityが
HogePresenterに
密結合している

```
// DIを導入していないコード
class HogeActivity : AppCompatActivity() {

  private lateinit var presenter: HogePresenter

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略

    presenter = HogePresenter(applicationContext)
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

これも密結合している

```
// 依存オブジェクトを解決するためのコンテナオブジェクト
object Container {

  val apiClient: ApiClient = ApiClientImpl()
  val userDao: UserDao = UserDaoImpl()

  val userRepo: UserRepository =
    UserRepository(apiClient, userDao)

  private var hogePresenter: HogePresenter? = null
  fun resolveHogePresenter(context: Context): HogePresenter {
    if (hogePresenter == null) {
      hogePresenter = HogePresenter(context, userRepo)
    }
    return requireNotNull(hogePresenter)
  }
}
```

Contextを引数に取るメソッドを追加

by lazyを使うことで、

Application Contextを渡せる

```
// DIを導入後のコード
class HogeActivity : AppCompatActivity() {

  private val presenter: HogePresenter by lazy {
    Container.resolveHogePresenter(applicationContext)
  }

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

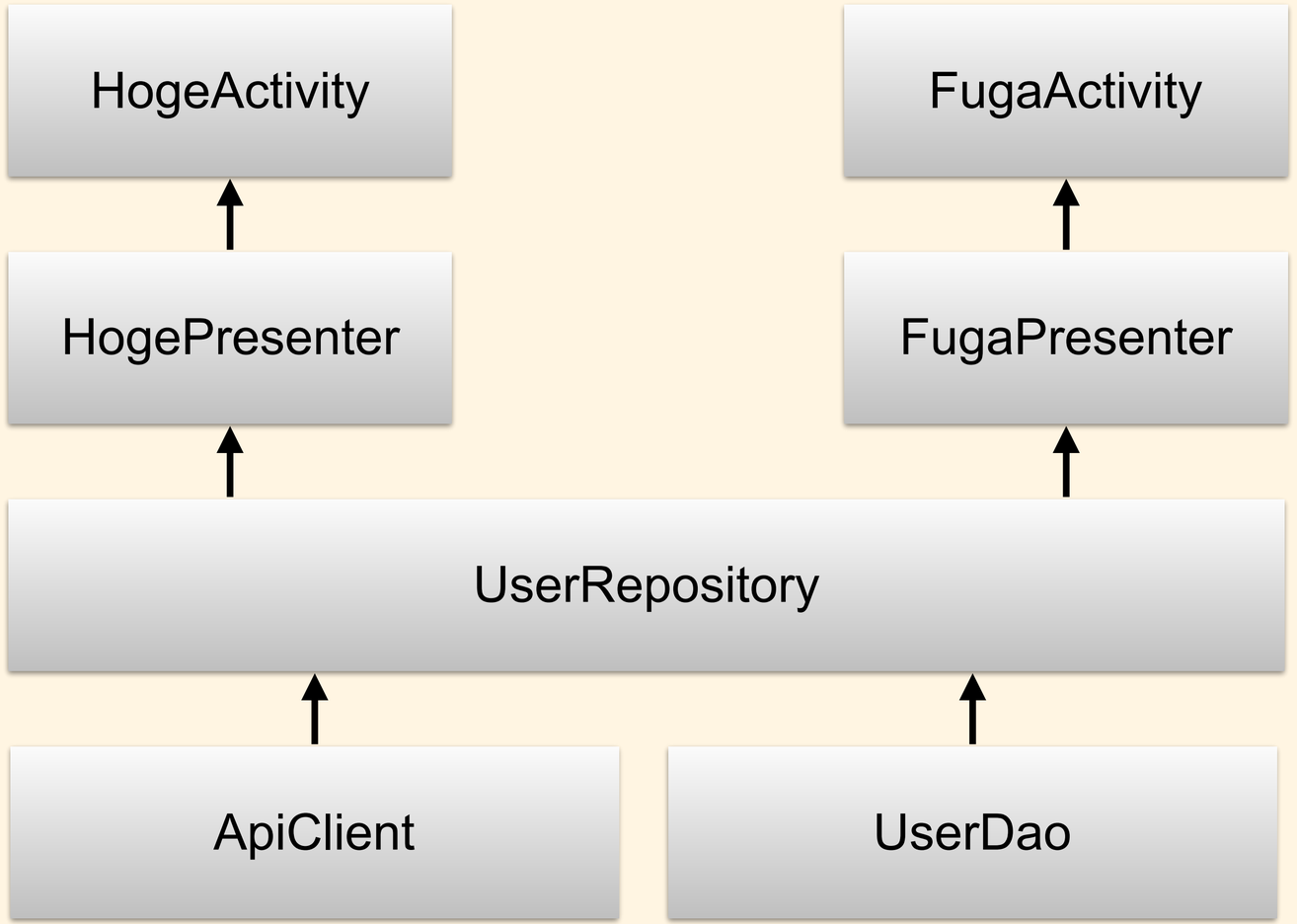
ここで気になることが🤔

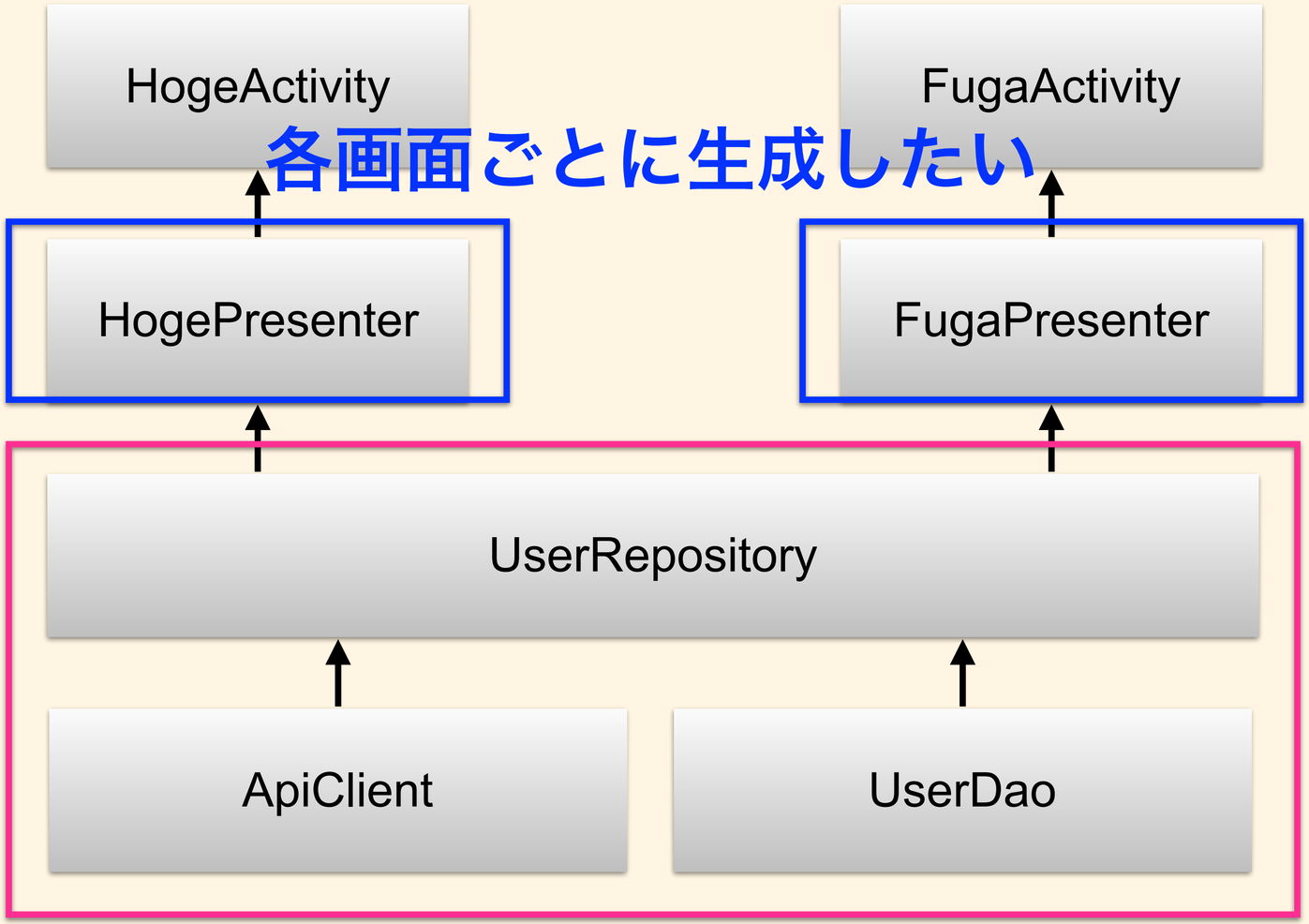
Presenterのライフサイクルこれで大丈夫？

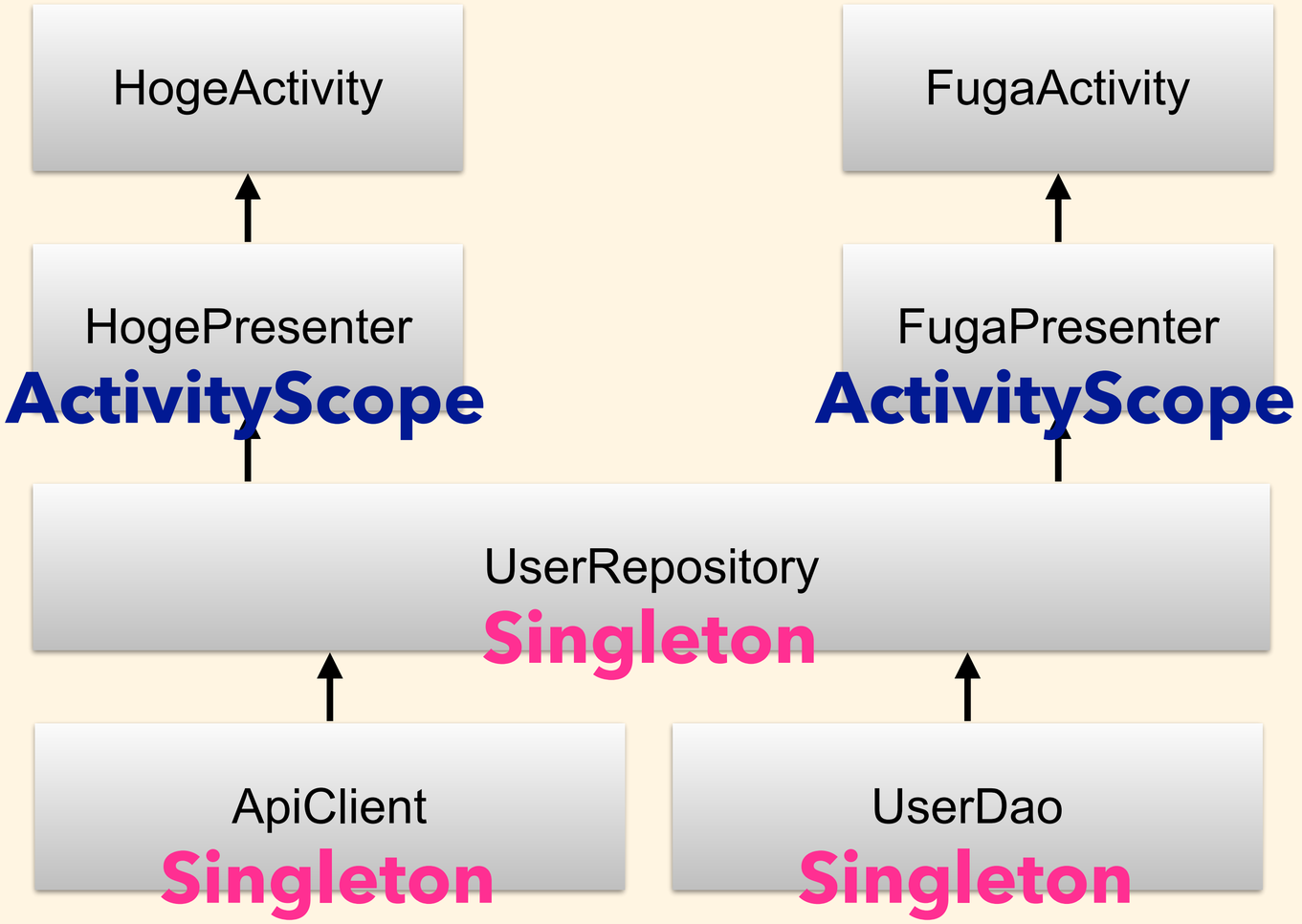
- PresenterはActivityと同じライフサイクルにしたい
  - Activityが生成されるとPresenterも生成され、Activityが破棄されると同時に破棄されるべき
  - Activityのインスタンスが2つあるときは、それぞれにPresenterがいてほしい

スコープ

- 依存オブジェクトの生存期間を表す
  - Singleton、Activityと同じ生存期間、など
- スコープごとに依存オブジェクトが生成され、使い回されるような仕組みが必要


HogeActivity
FugaActivity
HogePresenter
FugaPresenter
UserRepository
ApiClient
UserDao


HogeActivity
FugaActivity
各画面ごとに生成したい
HogePresenter
FugaPresenter
UserRepository
ApiClient
UserDao
画面共通で使いまわしたい


HogeActivity
FugaActivity
HogePresenter
ActivityScope
FugaPresenter
ActivityScope
UserRepository
Singleton
ApiClient
Singleton
UserDao
Singleton

```
object Container {
  // 省略

  val userRepo = UserRepository(apiClient, logger)
}

class HogeActivityScopeContainer {
  fun resolveHogePresenter(ctx: Context): HogePresenter {…}
}

class FugaActivityScopeContainer {
  fun resolveFugaPresenter(ctx: Context): FugaPresenter {…}
}
```

```
object Container {
  // 省略
  val userRepo = UserRepository(apiClient, logger)
}
```

Singletonな
クラスのみ

```
class HogeActivityScopeContainer {
  fun resolveHogePresenter(ctx: Context): HogePresenter {…}
}

class FugaActivityScopeContainer {
  fun resolveFugaPresenter(ctx: Context): FugaPresenter {…}
}
```

```
object Container {
  // 省略
  val userRepo = UserRepository(apiClient, logger)
}

class HogeActivityScopeContainer {
  fun resolveHogePresenter(ctx: Context): HogePresenter {…}
}

class FugaActivityScopeContainer {
  fun resolveFugaPresenter(ctx: Context): FugaPresenter {…}
}
```

HogeActivityと同じ
スコープのクラスを返す

```
// DIを導入後のコード
class HogeActivity : AppCompatActivity() {

  private val container = HogeActivityScopeContainer()

  private val presenter: HogePresenter by lazy {
    container.resolveHogePresenter(applicationContext)
  }

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

このActivityの
コンテナを初期化

```
// DIを導入後のコード
class HogeActivity : AppCompatActivity() {

  private val container = HogeActivityScopeContainer()

  private val presenter: HogePresenter by lazy {
    container.resolveHogePresenter(applicationContext)
  }

  override fun onCreate(savedInstanceSta
    // 省略
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

containerを使って注入

ここまでのまとめ

- DIとはコンポーネント間の依存関係を外部に追い出すパターン
- 依存コンポーネントを外部から受け取るようにする
- スコープはオブジェクトの生存期間を表し、スコープごとにコンテナを持つ
- ここまでの実装はあくまで解説用の雑なものなので注意！

  より実践的な実装方法は次以降

DIコンテナ

DIコンテナ

- DIを実現するためのライブラリ
- 先程自分で実装したContainer（よりもっといいもの）を自動で作ってくれるライブラリ

Androidで有名なDIコンテナ

- Dagger2
- Koin
- Kodein
- Toothpick
- ~~Roboguice~~

Androidで有名なDIコンテナ

- Dagger2
- Koin
- Kodein
- Toothpick
- ~~Roboguice~~

Dagger2
https://google.github.io/dagger/

Dagger2

- 元々Squareが開発 => 現在はGoogleがfork
- Annotation processorを使用
  - コンパイル時に依存ツリーに問題があると分かる🤦
  - コンストラクタが使えるクラスの場合、依存関係の解決を自動生成できる🤦
  - ビルドは遅くなる🙅
- ドキュメントが難しい🙅🙅🙅

注意！

- Dagger2はかなり多機能ですが、時間の関係上あっさりしか紹介できません
- Dagger2にはAndroidに限らない通常版とAndroidサポート版（dagger.android）がある
  - 今日はdagger.androidでのActivityへの注入を紹介
  - 「おまじない」が大量に出てくるので注意

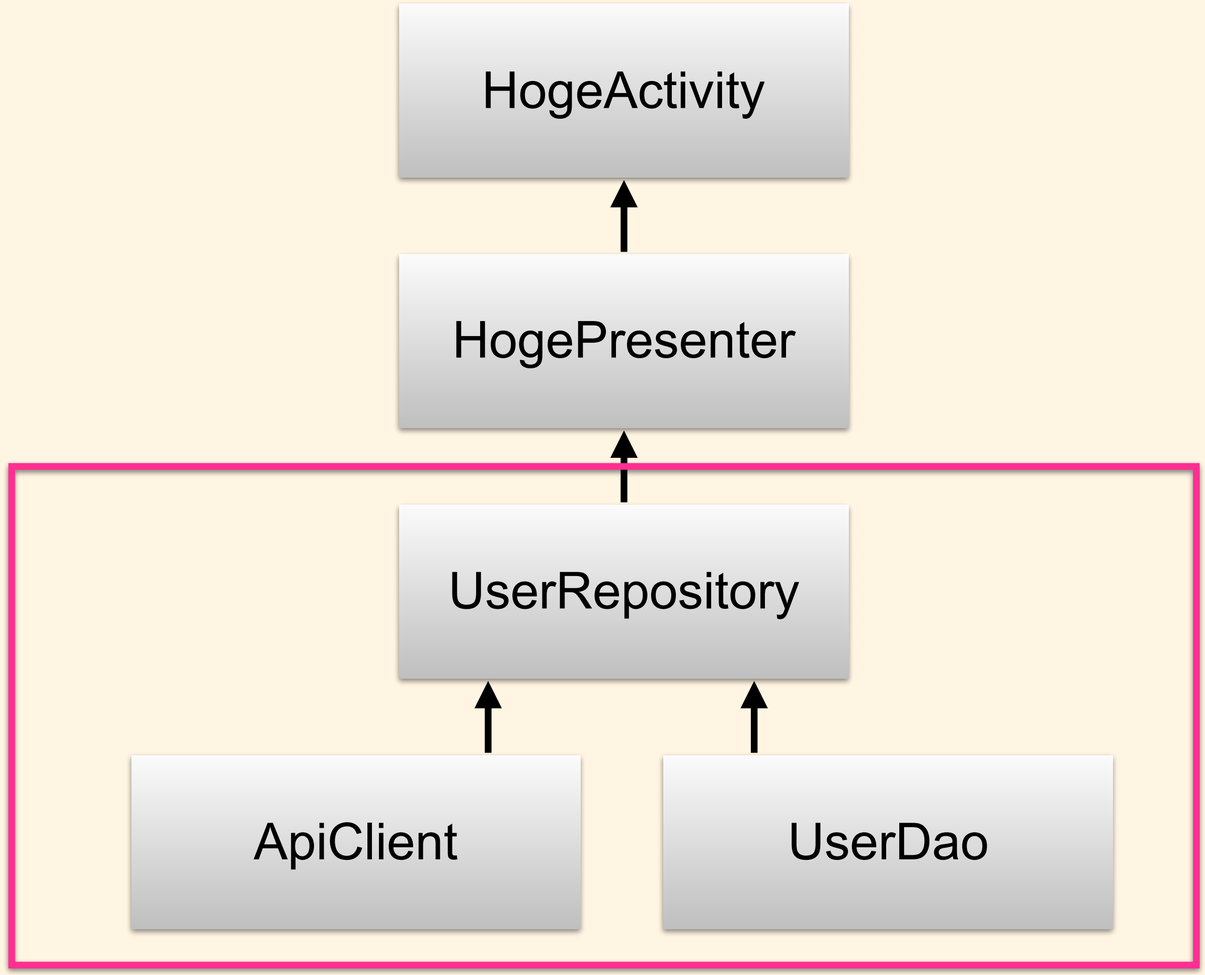
Dagger2の登場人物

- Module
- Component

Module

- 各依存クラスをどうインスタンス化するかを定義するクラス
- Daggerの場合コンストラクタを使ってインスタンス化するクラスは自動的に生成できる。
  コンストラクタが使えないクラスをどうやってインスタンス化するかだけ定義すればよい。

よくあるMVPの例


HogeActivity
HogePresenter
UserRepository
ApiClient
UserDao

```
class ApiClientImpl private constructor(): ApiClient {
  class Builder {
    fun build(): ApiClient {...}
  }
  override fun getUser(id: Long): Single<User> {...}
}


class UserDaoImpl(): UserDao {
  override fun find(id: Long): Single<User> {...}
}


class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {
  fun find(id: Long): Single<Hoge> {...}
}
```

```
class ApiClientImpl private constructor(): ApiClient {
  class Builder {
    fun build(): ApiClient {...}
  }
  override fun getUser(id: Long): Single<User> {...}
}
```

自動生成できない🙅‍♂️

=> Moduleに書く必要あり

```
class UserDaoImpl(): UserDao {
  override fun find(id: Long): Single<User> {...}
}
```

自動生成できる🤷‍♂️

=>Moduleに書く必要なし

```
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {
  fun find(id: Long): Single<Hoge> {...}
}
```

```
@Module
class AppModule {

  @Singleton
  @Provides
  fun provideApiClient(): ApiClient {
    return ApiClientImpl.Builder().build()
  }
}
```

Dagger2のモジュールを意味する

```
@Module
class AppModule {

  @Singleton
  @Provides
  fun provideApiClient(): ApiClient {
    return ApiClientImpl.Builder().build()
  }
}
```

```
@Module
class AppModule {

  @Singleton
  @Provides
  fun provideApiClient(): ApiClient {
    return ApiClientImpl.Builder().build()
  }
}
```

```
@Module
class AppModule {

  @Singleton
  @Provides
  fun provideApiClient(): ApiClient {
    return ApiClientImpl.Builder().build()
  }
}
```

このメソッドを依存解決に使う

```
@Module
class AppModule {

  @Singleton
  @Provides
  fun provideApiClient(): ApiClient {
    return ApiClientImpl.Builder().build()
  }
}
```

ApiClientのスコープ

```
class ApiClientImpl private constructor(): ApiClient {
  class Builder {
    fun build(): ApiClient {...}
  }
  override fun getUser(id: Long): Single<User> {...}
}
```

自動生成できない🤦

=>Moduleに書く必要あり

```
class UserDaoImpl(): UserDao {
  override fun find(id: Long): Single<User> {...}
}
```

```
class UserRepository(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {
  fun find(id: Long): Single<Hoge> {...}
}
```

自動生成できる🤦

=>Moduleに書く必要なし

```
@Singleton
class UserRepository @Inject constructor(
  private val apiClient: ApiClient,
  private val userDao: UserDao
) {
  fun find(id: Long): Single<User> {...}
}
```

このコンストラクタを使ってインスタンス化される

```
@Module
abstract class ActivityModule {
  @ActivityScope
  @ContributesAndroidInjector
  abstract fun contributeHogeActivity(): HogeActivity
}
```

dagger.androidを使うための特殊なモジュール

@Providesメソッドを持たない

```
@Module
abstract class ActivityModule {
  @ActivityScope
  @ContributesAndroidInjector
  abstract fun contributeHogeActivity(): HogeActivity
}
```

HogeActivityに注入するためのメソッド

Component

- Moduleを使って依存しているオブジェクトを生成・注入するクラス

```
@Singleton
@Component(modules = [
  AppModule::class,
  ActivityModule::class,
  AndroidInjectionModule::class
])
interface AppComponent: AndroidInjector<App> {

  @Component.Builder
  abstract class Builder: AndroidInjector.Builder<App>()
}
```

```
@Singleton
@Component(modules = [
  AppModule::class,
  ActivityModule::class,
  AndroidInjectionModule::class
])
interface AppComponent: AndroidInjector<App> {

  @Component.Builder
  abstract class Builder: AndroidInjector.Builder<App>()
}
```

このコンポーネントが使う
モジュールの配列

```
@Singleton
@Component(modules = [
  AppModule::class,
  ActivityModule::class,
  AndroidInjectionModule::class
])
interface AppComponent: AndroidInjector<App> {

  @Component.Builder
  abstract class Builder: AndroidInjector.Builder<App>()
}
```

Android support用のインタフェースを継承させる

```
class App : Application(), HasActivityInjector {

  @Inject
  lateinit var injector: DispatchingAndroidInjector<Activity>

  override fun onCreate() {
    super.onCreate()

    // Dagger2の初期化
    DaggerAppComponent.builder()
      .create(this)
      .inject(this)
  }

  override fun activityInjector(): AndroidInjector<Activity>
    = injector
}
```

```
class App : Application(), HasActivityInjector {

  @Inject
  lateinit var injector: DispatchingAndroidInjector<Activity>

  override fun onCreate() {
    super.onCreate()

    // Dagger2の初期化
    DaggerAppComponent.builder()
      .create(this)
      .inject(this)
  }

  override fun activityInjector(): AndroidInjector<Activity>
    = injector
}
```

Android support用のインタフェースを継承させる

```
class App : Application(), HasActivityInjector {

  @Inject
  lateinit var injector: DispatchingAndroidInjector<Activity>

  override fun onCreate() {
    super.onCreate()

    // Dagger2の初期化
    DaggerAppComponent.builder()
      .create(this)
      .inject(this)
  }

  override fun activityInjector(): AndroidInjector<Activity>
    = injector
}
```

AppにinjectorをInjectする

```
class HogeActivity : AppCompatActivity() {

  @Inject
  lateinit var presenter: HogePresenter

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略


    AndroidInjection.inject(this)
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

```
class HogeActivity : AppCompatActivity() {

  @Inject
  lateinit var presenter: HogePresenter

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略


    AndroidInjection.inject(this)
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

依存オブジェクトには
@Injectを付けておく

```
class HogeActivity : AppCompatActivity() {

  @Inject
  lateinit var presenter: HogePresenter

  override fun onCreate(savedInstanceState: Bundle?) {
    // 省略

    AndroidInjection.inject(this)
  }

  fun onButtonClick(view: View) {
    presenter.onButtonClick()
  }
}
```

依存オブジェクトを注入

なるほど、わからん🤯

- 大丈夫
  - 多分みんな最初は分かってない
- 今後Daggerを導入するときに、もう一度この資料や他のサンプルを見直せばOK

Koin
https://insert-koin.io/

Koin

- Kotlinで書かれたDIコンテナ
  - 移譲プロパティを使用して依存性を解決
- Annotation processor不使用
  - ビルド速度に悪影響はない🙆
  - 実行するまで依存ツリーに問題がないか分からない🙅
  - constructor injectionできるクラスでも、モジュールに書く必要あり🙅
- ドキュメントがわかりやすい🙆
- AAC ViewModelに対応🙆

使い方

1. moduleの定義
   - ここはDagger2と同じ
2. startKoinでコンテナ作成
3. 注入先のクラスで移譲プロパティを使って注入する

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

モジュールの作成

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

Singleton

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

オブジェクトを
毎回作成するクラス

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

getで依存解決

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      single<Logger> { LoggerImpl() }
      factory<ApiClient> {
        ApiClientImpl.Builder().build()
      }
      factory<HogeRepository> {
        HogeRepository(get())
      }
    }

    // 依存ツリーの作成
    startKoin(this, listOf(appModule))
  }
}
```

コンテナの作成

```
class HogeActivity : AppCompatActivity() {

    val logger: Logger by inject()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ...

        logger.v("Hello world!")
    }
}
```

委譲プロパティを使って注入

AAC ViewModelに対応

- ViewModelはFactory経由でインスタンスを生成する
  - Factoryを継承する仕組みがあり、DIと組み合わせるのは
    トリックが必要
- Koinは対応している！

```
class HogeViewModel(
  private val userRepository: UserRepository
) : ViewModel() {…}
```

```
class App : Application() {

  override fun onCreate() {
    super.onCreate()

    // Koinのモジュールの設定
    val appModule = module {
      // ...
      viewModel { HogeViewModel(get()) }
    }
    startKoin(this, listOf(appModule))
  }
}
```

ViewModelのインスタンス化

ViewModel用の
委譲プロパティを使う

```
class HogeActivity : AppCompatActivity() {

  val hogeViewModel: HogeViewModel by viewModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // ...
  }

  fun onButtonClick(view: View) {
    hogeViewModel.onButtonClick()
  }
}
```

ここまでのまとめ

- DIコンテナ = DIを簡単に実装するためのライブラリ
- Dagger2...多機能だが難しい。constructor injectionの場合はモジュール省略可能。
- Koin...わかりやすい。AAC ViewModelに対応している。

DIを導入して何が変わった？

DIのメリット（再掲）

- コンポーネント間が疎結合になることで、以下のメリットがある
  - アプリケーションを拡張しやすくなる
  - テストが書きやすくなる
    - これは後で詳しく解説

単体テスト

```
// テスト対象のクラス（DI導入前）
class UserRepository {

  private val apiClient: ApiClient = ApiClientImpl()

  fun find(id: Long): Single<User> =
    apiClient.getUser(id)
}
```

ApiClientに密結合している

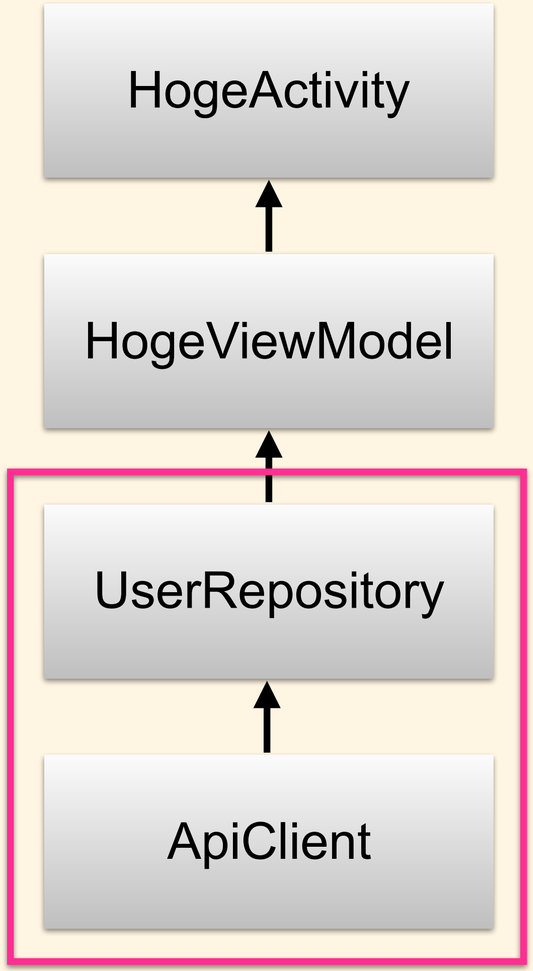
```
// UserRepositoryのテストコード
class UserRepositoryTest {

  @Test
  fun find_isSuccess() {
    val repo = UserRepository()
    repo.find(1)
      .test()
      .awaitCount(1)
      .assertValue(expected)
  }
}
```

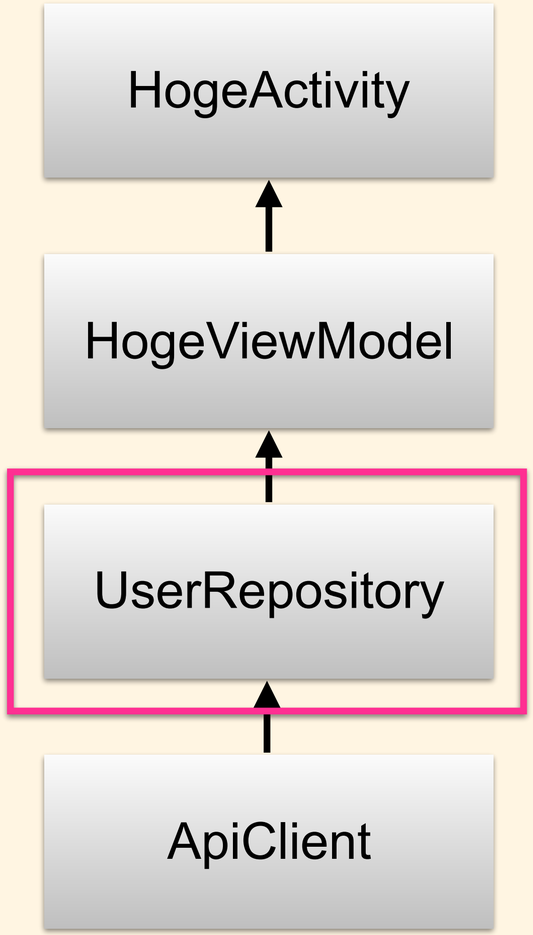
```
// UserRepositoryのテストコード
class UserRepositoryTest {

  @Test
  fun find_isSuccess() {
    val repo = UserRepository()
    repo.find(1)
        .test()
        .awaitCount(1)
        .assertValue(expected)
  }
}
```

ApiClientImplの
コードも実行している



ここがテスト対象に
なってしまっている



本当にテストしたいのは
ここだけ

DIを導入している場合

- 依存コンポーネントを注入するときに、テストに都合のいいオブジェクトを使うことで、対象のクラスのみ検証可能
  - Mockk, Mockitoなどでモックにすることが多い

```
// テスト対象のクラス（DI導入後）
class UserRepository(
  private val apiClient: ApiClient
) {

  fun find(id: Long): Single<User> =
    apiClient.getUser(id)
}
```

ApiClientを
外部から注入可能

```
// UserRepositoryのテストコード
class UserRepositoryTest {

  @Test
  fun find_isSuccess() {
    val apiClient = mockk<ApiClient> {
      every { getUser(any()) } returns Single.just(user)
    }

    val repo = UserRepository(apiClient)
    repo.find(1)
      .test()
      .awaitCount(1)
      .assertValue(user)
  }
}
```

```
// UserRepositoryのテストコード
class UserRepositoryTest {

  @Test
  fun find_isSuccess() {
    val apiClient = mockk<ApiClient> {
      every { getUser(any()) } returns Single.just(user)
    }

    val repo = UserRepository(apiClient)
    repo.find(1)
      .test()
      .awaitCount(1)
      .assertValue(user)
  }
}
```

ApiClientのモックを作成

```
// UserRepositoryのテストコード
class UserRepositoryTest {

  @Test
  fun find_isSuccess() {
    val apiClient = mockk<ApiClient> {
      every { getUser(any()) } returns Single.just(user)
    }

    val repo = UserRepository(apiClient)
    repo.find(1)
      .test()
      .awaitCount(1)
      .assertValue(user)
  }
}
```

モックを注入して
テスト実行

結合テスト

結合テスト

- ???「どうせ結合した状態でテストするからDIでコンポーネント間を疎結合にしても意味ないんじゃないんの？」
  - そんなことはない

例: EspressoでUIテストを書く

- APIクライアントの応答が毎回違うので辛い
  - レスポンスが時間によって違う
  - 応答時間
  - よくAPIが壊れる（開発環境など）
- DIを導入していると、簡単にAPIクライアントをテスト用のモックに差し替えることができる

DIコンテナのモジュールをテスト用に差し替える

1. テスト用のApplicationを作成し、内部でテスト用モジュールを使って依存ツリーを作る

2. Instrumented test実行時に起動するApplicationクラスを、1で作ったクラスに差し替える

```
open class App : Application() {

  private val appModule = module {
    single<Logger> { LoggerImpl() }
    factory { UserRepository(get()) }
    viewModel { MainViewModel(get(), get()) }
  }

  // APIクライアント用モジュール
  protected open val apiModule = module {
    factory<ApiClient> { ApiClientImpl() }
  }

  override fun onCreate() {
    super.onCreate()

    // 依存ツリーを構築
    startKoin(this, listOf(appModule, apiModule))
  }
}
```

```
open class App : Application() {

    private val appModule = module {
        single<Logger> { LoggerImpl() }
        factory { UserRepository(get()) }
        viewModel { MainViewModel(get(), get()) }
    }

    // APIクライアント用モジュール
    protected open val apiModule = module {
        factory<ApiClient> { ApiClientImpl() }
    }

    override fun onCreate() {
        super.onCreate()

        // 依存ツリーを構築
        startKoin(this, listOf(appModule, apiModule))
    }
}
```

テスト用のAppクラスを
継承して作れるようにする

```
open class App : Application() {

  private val appModule = module {
    single<Logger> { LoggerImpl() }
    factory { UserRepository(get()) }
    viewModel { MainViewModel(get(), get()) }
  }

  // APIクライアント用モジュール
  protected open val apiModule = module {
    factory<ApiClient> { ApiClientImpl() }
  }

  override fun onCreate() {
    super.onCreate()

    // 依存ツリーを構築
    startKoin(this, listOf(appModule, apiModule))
  }
}
```

ここもオーバーライド
できるようにする

```
class TestApp : App() {

  override val apiModule = module {
    factory<ApiClient> {
      val user = User(123, “kobakei”)
      val apiClient = mockk<ApiClient> {
        every { getUser(any()) } returns Single.just(user)
      }
      apiClient
    }
  }

}
```

```
class TestApp : App() {

  override val apiModule = module {
    factory<ApiClient> {
      val user = User(123, “kobakei”)
      val apiClient = mockk<ApiClient> {
        every { getUser(any()) } returns Single.just(user)
      }
      apiClient
    }
  }

}
```

モックのAPIクライアントを返す

モジュールでオーバーライド

```
// アプリケーションを差し替えるためのランナー
class MyTestRunner : AndroidJUnitRunner() {

  override fun newApplication(cl: ClassLoader?, name: String?, c: Context?) =
    super.newApplication(cl, TestApp::class.java.name, c)
}
```

TestAppクラスを起動する

```
// app/build.gradle
android {
  defaultConfig {
    testInstrumentationRunner “your.app.MyTestRunner”
  }
}
```

Instrumented test時のランナーを指定

本日のまとめ

まとめ

- DI = コンポーネント間の依存関係を外部に追い出すデザインパターン
- DIを導入することで、コンポーネントが疎結合になり、コンポーネントを差し替えたり、テストが書きやすくなる
- Dagger2やKoinなどDIコンテナを利用しよう

Thanks!!!