Pràctica introductòria #6 - Arquitectura neta

Projecte Integrat de Software (2024/25)
Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

1 Introducció

L'arquitectura neta (*clean architecture*) és un enfocament de disseny de programari que organitza els components en capes concèntriques segons el seu nivell d'abstracció, seguint la *regla de dependència*. Aquesta regla estableix que les capes interiors, més estables i independents dels detalls concrets, no han de dependre de les capes exteriors, que són més volàtils i específiques.

Per exemple, la interfície d'usuari (UI) i els mecanismes de persistència són elements que poden canviar amb freqüència, per la qual cosa es col·loquen a les capes més externes. En canvi, els objectes del domini i els casos d'ús, que encapsulen la lògica fonamental del sistema, es mantenen a les capes interiors, protegits de modificacions externes. Si un component d'una capa interior ha de comunicar-se amb un de l'exterior, ho fa mitjançant una abstracció que la capa exterior implementa.

Aquesta guia estructurarà el codi en mòduls i paquets que reflecteixin aquesta organització, assegurant un compliment estricte de la regla de dependència.

Requeriments

Haver fet les pràctiques introductòries anteriors (PI1 - PI5).

Objectius i temps

1. Repassar els conceptes claus de l'arquitectura neta.
2. Saber aplicar l'arquitectura neta en context d'una aplicació Android.
3. Saber fer injecció de dependències (manual) en una aplicació Android.
4. Veure una manera de propagar errors a través de l'arquitectura neta sense vulnerar la regla de dependència.

Objectius i temps

- `CleanArchitectureExample.zip`: projecte base d'Android Studio descarregable del campus virtual, per fer el seguiment d'aquest guió. Està basat en el codi de la solució de la PI5 (FirebaseRepositoryExample.zip) per a fer-ho incremental.

2 Arquitectura Neta

Proposada per Robert C. Martin, l'**arquitectura neta** [2] estableix una organització clara dels components en capes circulars i concèntriques, seguint una estructura similar a una ceba. Aquesta arquitectura es compon de quatre capes principals, ordenades de l'exterior cap a l'interior:

1. Regles del negoci de l'empresa.
2. Regles de negoci de l'aplicació.
3. Adaptadors d'interfície.
4. *Frameworks* i connectors.

Vegeu la Fig. 1.

2.1 Nivell d'abstracció i estabilitat

Les capes exteriors, amb un nivell d'abstracció més baix, solen ser més inestables que les capes interiors, que tenen un nivell d'abstracció més alt. Un exemple clar de component inestable és la Interfície d'Usuari (UI). Les modificacions a la UI són habituals i necessàries, però han de poder realitzar-se sense afectar la part estable de l'aplicació, com la lògica del negoci. Un altre exemple de component inestable és la capa de persistència, que depèn d'infraestructura externa (per exemple, la llibreria *Firestore*), la qual no controlem com a desenvolupadors.

2.2 Regla de dependència i DIP

L'arquitectura neta es basa en la **regla de dependència**, que estableix que *les dependències que travessen fronteres entre capes han de fer-ho seguint el sentit de l'estabilitat*. Això implica que els components de les capes exteriors poden dependre dels components de les capes interiors, però no a l'inrevés.

Tot i això, les capes interiors podrien necessitar comunicar-se amb les exteriors. És a dir, es vol permetre el flux de control de dins cap a fora, però mantenint la dependència en sentit contrari, per no vulnerar la regla de dependència. Per aconseguir-ho, apliquem el **d'inversió de dependències (DIP)**: la capa interior defineix una abstracció (com una interfície en Java) que la capa exterior ha d'implementar. Així, el component de la capa interior pot interactuar amb

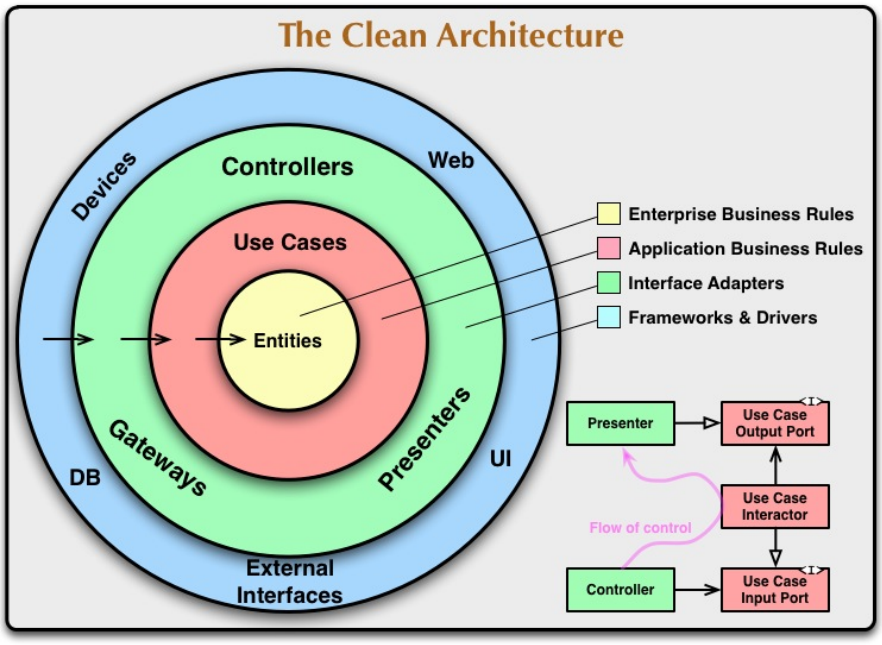

Figure 1: Divisió en capes circulars concèntriques de l'arquitectura neta. Les fletxes indiquen el sentit de les dependències.

l'abstracció sense dependre directament de la seva implementació a la capa exterior.

A continuació, veiem com aplicar aquests conceptes a una aplicació pràctica amb una UI basada en *Android* i una capa de persistència amb *Firestore*.

3 Aplicant l'arquitectura neta

Per tenir més control del sentit de les dependències entre capes, implementarem les capes mitjançant la definició de mòduls. A continuació, veurem quins mòduls tindrem i com podem crear-los. Més tard, concretarem quins components formaran part de cada mòdul.

3.1 Estructuració del projecte Android en mòduls

Els mòduls ens permetran una més clara separació i control sobre les dependències entre capes. Dins de cada mòdul, podrem – òbviament – tenir-hi paquets per separar els components de la mateixa capa.

Tal com mostrem a la Fig. 2, estructurarem el codi en els mòduls **app** (o

presentation), **features**, **domain** i **data**. Seguim la codificació de colors de l'arquitectura neta per aclarir a quina capa de l'arquitectura correspon cada component de cadascun dels mòduls:

- **presentation** i **data** són frameworks (és a dir, llibreries) i connectors.
- **features** són regles de negoci de l'aplicació.
- **domain** són regles de negoci de l'empresa.

Amb les fletxes de la mateixa figura, ja veiem el sentit de les dependències i com es respecta la regla de dependència.

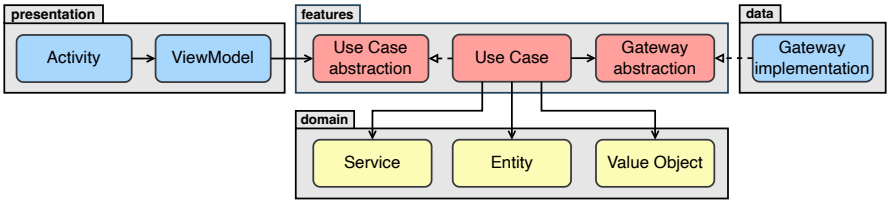

Figure 2: Una aplicació d'Android amb mòduls: **presentation**, **features**, **domain** i **data**. La divisió per colors indica el nivell d'abstracció dels components continguts en els mòduls.

3.1.1 Creació de mòduls amb Android Studio

Per a crear un mòdul, fem `File > New > New Module...` per obrir la finestra de la Fig. 3. A l'esquerra hi veureu el tipus de mòdul. En funció del que hagi de contenir el mòdul, haurà de ser d'un tipus o d'un altre:

- **app** (reanomenat **presentation**) serà un mòdul de tipus "Phone & Tablet". El mòdul ja ve creat quan creem el projecte. Per canviar-li el nom, cliquem el botó dret a sobre del nom del mòdul i fent `Refactor > Renam e > Rename module`.
- **features** i **domain** seran mòduls de tipus **Java o Kotlin Library**, ja que, no contenen codi Android. El nom del mòdul s'especifica en el camp "Library name" (per defecte, "lib") i al "Package name" li incloem el nom de l'aplicació (p. ex. `edu.ub.pis2425.\textbf{nomdelprojecte }.features`).
- **data** és de tipus **Android Library** perquè, en el nostre cas, farà ús de *Firestore*, un *framework* que depèn de llibreries d'Android.

Cada mòdul tindrà el seu propi fitxer gradle. Veure la Fig. **??**. Això permet afegir dependències als mòduls de manera individual.

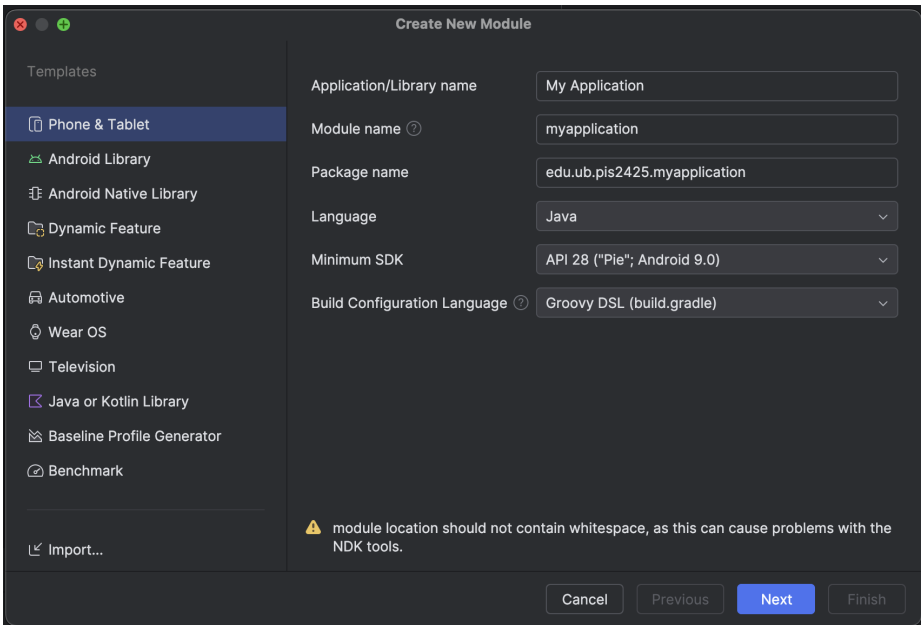

Figure 3: Finestra de creació de mòduls. Veure text pels detalls.

3.1.2 Dependències entre mòduls

A més de les dependències de llibreries externes, a cada mòdul li hem d'afegir les dependències dels altres mòduls. A la mateixa secció `dependencies` dels gradle a nivell de mòdul, hi podem afegir les dependències dels mòduls.

Per exemple, si al mòdul **features** li volguéssim afegir la dependència del mòdul **domain**, modificaríem la secció `dependencies` del fitxer `build.gradle (Module :features)` afegint-hi la línia:

```
dependencies {
    implementation project(:domain)

    ...
}
```

A continuació, llistem totes les dependències entre els mòduls per tal de poder muntar l'arquitectura que es mostrava a la Fig. 2:

- `domain` no depen de cap altre mòdul.
- `features` depèn de `domain`.
- `data` depèn de `domain`.

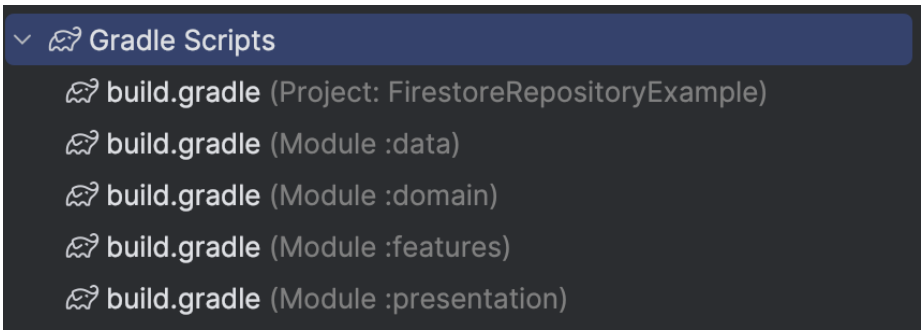

Figure 4: Fitxers gradle del nostre projecte. Tenim el fitxer gradle general i un per cadascun dels mòduls.

- `presentation` depèn de tota la resta de mòduls (`domain`, `features` i `data`), perquè és el mòdul que farà la injecció de dependències – tal com expliquem a la Sec. 3.3.

3.2 Components dels mòduls

A continuació, expliquem què hauria de contenir cadascun dels mòduls.

3.2.1 Domini

Conté els *objectes del domini*. Quins? L'arquitectura neta sovint s'aplica juntament amb alguns patrons tàctics del *disseny guiat pel domini* (o *domain-driven design*) [1]. Un d'ells és de distingir els objectes del domini en *entitats*, *objectes de valor*, *agregats* i `serveis`. En aquesta secció, fem un repàs de què són cadascun d'ells.

Entitats. Tenen una identitat única representada per un identificador immutable, que permet que si una entitat veu modificat qualsevol dels seus altres atributs, pugui seguir sent reconeguda pel sistema com la mateixa identitat. Pensem en l'exemple d'un `Client` que canviï el seu correu, nom d'usuari o adreça postal.

👉 Les **referències d'objecte no són identificadors d'entitat vàlids** perquè canvien quan les entitats es guarden i es recarreguen de persistència (es crea un objecte nou i, per tant, una referència nova).

Objectes de valor. No tenen identificador, sinó que la seva identitat ve donada pel valor conjunt de tots els seus atributs. Per exemple, un `Price(double amount, String currency)` seria un clar objecte de valor. Dos objectes amb la mateixa quantitat d'una certa divisa són, essencialment, el mateix preu. I si canvia un dels atributs, la quantitat o la divisa, estem parlant d'un objecte

fonamentalment diferent. Per tant, els objectes de valor són immutables, no permetent canviar-los cap del valor dels seus atributs.

👉 Si tenim un objecte `Preu` i haguéssim de canviar-li la quantitat, crearíem un nou objecte. Això "embruta" el codi client (el que fa ús de l'objecte de valor). En lloc d'això, podríem delegar la creació del nou objecte a la instància de l'objecte de valor antic. Per exemple: `public Price addAmount(double amount) { return new Price(this.amount + increment, this.currency); }`.

Agregats. Un agregat és una agrupació conceptual d'entitats i objectes de valor que formen una unitat de consistència i integritat dins del domini. Cada agregat té una entitat arrel que actua com a punt d'entrada i és responsable d'orquestrar les operacions que afectin l'estat de les dades dels objectes agrupats en l'agregat. Cal evitar, doncs, exposar fora de l'agregat qualsevol entitat que no sigui l'entitat arrel o objectes de valor. A més, els agregats són unitats de persistència i, per tant, tampoc hem de permetre que guardin referències d'altres agregats. Si un agregat ha de tenir accés a un altre agregat, sempre podem fer ús d'una referència transitiva (passant-li-la per un mètode i descartant-la després).

👉 Els objectes de valor es poden exposar fora dels agregats, precisament, per la seva condició immutable i formar part de diversos agregats alhora.

Serveis. Implementen lògica de coordinació entre agregats. Els serveis poden invocar altres serveis i s'implementen com a façanes sense estat.

3.2.2 Features

Inclou, principalment, els *casos d'ús*, abstraccions dels casos d'ús i abstraccions de *gateway* que les capes externes implementen. A vegades també tindrem *models de petició* i/o *models de resposta*, que són objectes que s'utilitzen per a facilitar la comunicació amb els casos d'ús.

Casos d'ús. Són serveis exposats a l'usuari, els quals poden – alhora – delegar en altres casos d'ús, serveis, agregats, etc. Fem la distinció amb els serveis per seguir el paradigma de l'*Screaming Architecture*[1], que vol dir que mirant el codi tindrem una idea clara i ràpida de què fa.

Abstraccions dels casos d'ús. Definirem abstraccions també pels casos d'ús. En aquest cas, l'abstracció no les definim per a complir amb la regla de dependència de l'arquitectura neta, ja que, qui crida els casos d'ús és un model de vista. Ho fem per a millorar la testabilitat dels casos d'ús. Hem dit que els casos d'ús poden delegar (dependre) d'altres casos d'ús i, per a testejar-los, podrem substituir les abstraccions per implementacions mock dels casos d'ús.

[1] Article sobre l'*Screaming Architecture* per Robert C. Martin: https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

Model de petició o resposta. Els models de petició són objectes de tipus contenidor per a encapsular el pas de paràmetres als casos d'ús. D'altra banda, els models de resposta serveixen per a encapsular les dades de retorn del cas d'ús. Pot valer la pena si s'han de passar o retornar tipus de dades més complexes que variables primitives, col·leccions o objectes del domini.

Abstraccions dels *gateways*. Les abstraccions que permeten el control de les implementacions dels *gateways*. Els gateways són mecanismes de sortida, per tant, repositoris o APIs externes.

3.2.3 Presentació

El mòdul de presentació contindrà els components de la UI (activitats o fragments[2]) i els d'adaptació que tenen a veure amb la UI (adaptadors de vista reciclable, models de vista i objectes de presentació). No entrarem a explicar què són cadascuna d'aquestes coses perquè ja ho heu vist en pràctiques introductòries anteriors. Potser cal parlar del que anomenem *objectes de presentació*.

Objectes de presentació (POs). Els objectes de presentació són classes sense comportament que utilitzem per representar informació dels objectes del domini, sense exposar-los fora de les capes del domini (`domain`) i de l'aplicació (`features`). *Quines implicacions tindria fer-ho?* La primera implicació és la possibilitat d'inconsistències en les dades. Les activitats podrien causar canvis en els objectes del domini que no farien persistir, perquè no tenen capacitat de fer-ho, podent generar discrepàncies entre els objectes en la memòria i la persistència. Una altra implicació és la vulneració de la regla de dependència. Els objectes de presentació es poden passar entre activitats o fragments, però això obliga a fer-los implementar la interfície `Parcelable`. Aquesta interfície és pròpia d'*Android SDK* i crea una dependència que no volem pels objectes del domini.

Els encarregats de transformar objectes del domini en POs, en el context de la programació Android, seran els models de vista. Un cop feta la transformació, el model de vista li podrà – si cal – guardar-se el PO i, també, fer-li arribar a l'activitat corresponent.

👉 Per convertir d'objectes del domini a objectes de presentació, podem utilitzar – com fèiem per convertir DTOs a objectes del domini – un mapejador genèric `ModelMapper`. Veure la classe `presentation.pos.mappers.DomainToPOMapper` en el codi base.

3.2.4 Data

Contindrà les implementacions de *gateways* (p. ex. repositoris) i els *objectes de transferència de dades* (DTOs).

[2]Introduïts més endavant, a la Sec. 4.1.

Implementacions de passarel·les (o *gateways*). Un clar exemple de passarel·la és – com bé sabeu – un repositori. En l'exemple del repositori de productes, tindreu `ProductRepository` al mòdul **domain** i `ProductRepositoryImpl`[3] aquí, al mòdul **data**.

👉 En el cas de fer ús de bases de dades relacionals, els repositoris podrien delegar en DAOs. Ara bé, assumint que feu servir *Firestore*, no els necessitareu. L'accés a la *Firestore* l'implementarà la passarel·la (p. ex. `ProductFirebaseRepository`) directament, gràcies a la capacitat de l'API de *Firestore* de mapejar automàticament els objectes de Java a documents i viceversa.

DTOs. Els DTO són classes contenidores que s'utilitzen per carregar-hi les dades de persistència i, en cas de necessitat, transferir-les entre components de persistència relacionats. Quines implicacions tindria no utilitzar DTOs? Si carreguéssim directament objectes del domini, aquests estarien condicionats per les dades i la seva estructura. L'altra és, de nou, la vulneració de la regla de dependència. Com ara, quan el *framework* utilitzat per la persistència, ens obliga a que els objectes on carreguem les dades depenguin d'ell. És el cas de *Firestore* amb el decorador *@DocumentId*, el qual utilitzem per a carregar l'identificador del document en l'atribut de l'objecte carregat. Per tant, triem que siguin els DTOs qui tinguin aquesta dependència i no els objectes del domini directament.

👉 La conversió de DTO → objectes del domini la fan els mateixos repositoris amb la classe de mapeig genèrica `data.dtos.firestore.mappers.DTOToDomainMapper`, explicada en detall a la pràctica introductòria de *Firebase*.

Arribats a aquest punt, seria un bon moment per a revisar el codi base i intentar trobar cadascun dels elements descrits fins ara.

3.3 Injecció de dependències a Android

La **injecció de dependències** és la pràctica de crear els components i passar-los les seves dependències (altres components) com a paràmetres del constructor. El que normalment s'injecta són abstraccions, permetent que els components que estan sent injectats no estiguin acoblats a implementacions concretes. Això ens serà útil per:

- Injectar repositoris als casos d'ús.
- Injectar casos d'ús a altres casos d'ús.
- Injectar casos d'ús als models de vista.

[3] Si preferiu dir-li `IProductRepository` a l'abstracció i `ProductRepository` a la implementació també podeu fer-ho. Només us demanem ser consistents amb la nomenclatura que feu servir.

Aquestes injeccions hauran de tenir efecte el més aviat possible just després de l'inici de l'execució de l'aplicació i, per tant, abans que l'activitat principal iniciï el seu cicle de vida i es cridi el mètode `onCreate` de l'activitat.

Fins ara hem obviat que existís cap altre objecte amb un cicle de vida que s'iniciés abans que el de la pròpia activitat principal. Bé doncs, sí que n'hi ha un, el que representa la pròpia aplicació. Es tracta d'un objecte de la classe `Application`, el qual té també els seus mètodes de callback (un d'ells `onCreate`).

El que farem, doncs, serà estendre la classe `Application` de l'*Android Sdk* i sobrecarregar-li el mètode `onCreate` per a crear un objecte d'una classe pròpia `AppContainer` on hi inicialitzarem tots els nostres components i hi farem la injecció de dependències:

```
1  import android.app.Application;
2
3  /**
4   * To be able to perform manual dependency injection.
5   * Source:
6   * https://developer.android.com/training/dependency-injection/
        manual
7   */
8  public class MyApplication extends Application {
9
10   public AppContainer appContainer;
11
12   @Override
13   public void onCreate() {
14     super.onCreate();
15     // Initialize the AppContainer when the application starts
16     appContainer = new AppContainer();
17   }
18
19   @SuppressWarnings("unused")
20   public AppContainer getAppContainer() {
21     return appContainer;
22   }
23
24   @SuppressWarnings("unused")
25   public AppContainer.ViewModelFactory getViewModelFactory() {
26     return appContainer.viewModelFactory;
27   }
28 }
```

Fixeu-vos que l'únic que fem és crear un objecte de la classe `AppContainer` i afegir-lo com atribut de `MyApplication`. També definim un parell de *getters* que ja veurem més tard perquè ens serviran. Centrem-nos en la implementació d'`AppContainer`:

```
public class AppContainer {
  /* Dependency injection */
  // Repositories
  private final ClientRepository clientRepository = new
      ClientFirestoreRepository();
  private final ProductRepository productRepository = new
      ProductFirestoreRepository();
  // Use cases
  private final LogInUseCase logInUseCase = new
      LogInUseCaseImpl(clientRepository);
  private final SignUpUseCase signUpUseCase = new
      SignUpUseCaseImpl(clientRepository);
  private final GetAllProductsUseCase getAllProductsUseCase =
      new GetAllProductsUseCaseImpl(productRepository);
  private final GetProductsByNameUseCase
      getProductsByNameUseCase = new
      GetProductsByNameUseCaseImpl(productRepository);
  private final RemoveProductUseCase removeProductUseCase = new
       RemoveProductUseCaseImpl(productRepository);

  ...
}
```

Veieu com creem i injectem els objectes seguint l'ordre de les dependències: si un cas d'ús depèn d'un repositori, primer creem el repositori i després el cas d'ús (passant-li el repositori pel constructor). Com hem dit abans, injectem abstraccions.

Ara faltaria **injectar els casos d'ús als models de vista**. Però com que els models de vida tenen el seu cicle de vida independent (que no controlem nosaltres), no té sentit crear-los i injectar-los ara mateix aquí. El que farem serà crear una *factory* de models de vista que permeti, de manera reflexiva, obtenir-los injectats. Aquesta és la resta de l'`AppContainer` on fem això:

```
public class AppContainer {
  /* Dependency injection */
  ...
  // ViewModel factory: initialization of internal class (see
      below)
  public ViewModelFactory viewModelFactory = new
      ViewModelFactory(this);

  /* Internal class definitions */
  // ViewModel factory
  public static class ViewModelFactory implements
      ViewModelProvider.Factory {
    private final AppContainer appContainer;

    public ViewModelFactory(AppContainer appContainer) {
```

```
13          this.appContainer = appContainer;
14      }
15
16      @SuppressWarnings("unchecked") // This would be helpful for
            lint warnings for casts.
17      @NonNull
18      @Override
19      public <T extends ViewModel> T create(Class<T> modelClass)
            {
20          if (modelClass.isAssignableFrom(SignUpViewModel.class)) {
21              return (T) new SignUpViewModel(appContainer.
                    signUpUseCase);
22          } else if (modelClass.isAssignableFrom(LogInViewModel.
                class)) {
23              return (T) new LogInViewModel(appContainer.logInUseCase
                    );
24          } else if (modelClass.isAssignableFrom(ShoppingViewModel.
                class)) {
25              return (T) new ShoppingViewModel(
26                  appContainer.getAllProductsUseCase,
27                  appContainer.getProductsByNameUseCase,
28                  appContainer.removeProductUseCase
29              );
30          } else if (modelClass.isAssignableFrom(
                ViewProductDetailsViewModel.class)) {
31              return (T) new ViewProductDetailsViewModel();
32          }
33          throw new IllegalArgumentException("ViewModel Not Found")
                ;
34      }
35  }
36  }
37  }
```

Aquesta és la *factory* que passarem als `ModelViewProvider` que fem anar a les activitats per inicialitzar els seus corresponents models de vista. De fet, aquesta ***factory* global**, substitueix la definició de *factories* individuals que definíem als models de vista (explicat a la pràctica introductòria dels models de vista). Anem a veure el cas concret de la creació del `LogInViewModel` a una `LogInActivity`:

```
1   public class LogInActivity extends Activity {
2
3     /* ViewModel */
4     private LogInViewModel logInViewModel;
5
6     ...
7
8     /**
9      * Initialize the viewmodel and its observers.
```

```
    */
  private void initViewModel() {
    /* Init viewmodel */
    logInViewModel = new ViewModelProvider(
        this,
        ((MyApplication) getApplication()).getViewModelFactory
            ()
    ).get(LogInViewModel.class);

    initObservers();
  }

  ...
}
```

I el constructor del `LogInViewModel` rebria el cas d'ús que injectem reflexivament des de la *factory* global:

```
public class LogInViewModel extends ViewModel {
  /* Use cases */
  private final LogInUseCase logInUseCase;

  ...

  /* Constructor */
  public LogInViewModel(LogInUseCase logInUseCase) {
    this.logInUseCase = logInUseCase;
    ...
  }

  ...
}
```

D'aquesta manera, ens quedaria ja tot injectat i lligat: l'activitat `LogInActivity` tindria assignat el model de vista `LogInViewModel`, el model de vista el cas d'ús `LogInUseCase` injectat i, alhora, el cas d'ús `LogInUseCase` tindria el repositori `ClientRepository`.

4 Observacions addicionals

A més de l'arquitectura neta, el codi proporcionat també inclou altres novetats. Concretament, l'ús de *fragments* i del *controlador de navegació* per la UI, que seran **OPCIONALS** pel desenvolupament del vostre projecte. Per tant, no els cobrirem en profunditat a les pràctiques introductòries. Tot i això, us en fem unes pinzellades ràpides i us proporcionem recursos per investigar-ho pel vostre compte amb més profunditat.

4.1 Fragments

Els *fragments*, igual que les activitats en Android, tenen una disposició associada, definida mitjançant arxius `XML`, i una lògica de control, implementada en codi Java. Aquests fragments disposen d'un cicle de vida propi, però són sensibles al cicle de vida de l'activitat que els conté. Els fragments tenen la capacitat de transitar entre ells, és a dir, poden canviar de manera dinàmica, enviar i rebre informació, entre d'altres operacions. Per tant, poden considerar-se com a *"activitats petites"* dins de les activitats que proporcionen diversos avantatges:

- **Modularitat**: Permeten dividir activitats grans en fragments més petits i manejables.
- **Reusabilitat**: Un mateix fragment pot ser utilitzat en diverses activitats.
- **Integració amb el controlador de navegació**: Els fragments poden ser fàcilment gestionats mitjançant el sistema de navegació d'Android.

Avui en dia, és habitual trobar aplicacions d'una sola activitat que actuen com a contenidors de fragments i que inclouen una barra de navegació, com ara una `BottomNavigationView`, que permet als usuaris seleccionar quin fragment es mostra en el contenidor.

Trobeu informació sobre fragments a la pàgina de desenvolupadors: https://developer.android.com/guide/fragments.

4.2 Controlador de navegació

El controlador de navegació ofereix diverses característiques. Entre les quals, el maneig transparent de transaccions entre fragments. Les transaccions són les transicions de canvi d'un fragment per un altre en un contenidor de fragments, les quals es poden fer manualment amb *transaccions* (veure https://developer.android.com/guide/fragments/transactions), similar a com fan les *intencions explícites* de les activitats, o utilitzant el controlador de navegació (veure `NavController` a https://developer.android.com/guide/navigation).

El controlador de navegació és recomanable, sobretot, quan hi ha una vista de navegació: `BottomNavigationView` (barra inferior) o `NavigationDrawer` (panell lateral que es mostra i s'oculta).

Trobareu el codi i fitxers relacionats amb aquesta part al mòdul **presentation**, principalment:

- `java/edu.ub.pis2425.cleanarchitectureexample.presentation.ui.activities/MainActivity.java` i el seu layout a `res/layout/activity_main.xml`. Mireu també l'AndroidManifest.xml per veure com ha quedat de net.

- `java/edu.ub.pis2425.cleanarchitectureexample.presentation.ui.fragments/*.java` i els seus layouts a `res/layout/fragment_*.xml`.
- `res/navigation/nav_graph_main.xml`
- `res/menu/bottom_navigation_menu.xml`
- `res/values/dimen.xml`
- `res/values/strings.xml`

5 Exercicis

Aquesta pràctica introductòria no inclou exercicis específics. El seu objectiu és que el codi base us serveixi com a recurs i guia per a la implementació del vostre projecte.

Podeu executar l'aplicació i accedir-hi amb les credencials `"admin/admin"`. Tanmateix, cal tenir en compte que el permís d'escriptura de la base de dades (*Firestore*), a la qual està connectat el projecte, està deshabilitat. Així doncs, la funcionalitat d'esborrar un producte de la llista no funcionarà si no teniu connectada la vostra pròpia base de dades, tal com es detalla a la pràctica introductòria sobre *Firestore*.

References

[1] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. 6

[2] Robert C Martin. Clean architecture, 2017. 2