Pràctica introductòria #3: Model – Vista – Model de Vista (MVVM)

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

Introducció

En les aplicacions Android, s'utilitza el patró *Model–View–ViewModel* (MVVM). Aquest patró estableix que el responsable d'emmagatzemar la informació que mostra la vista (activitat) – ja sigui derivada de la interacció de l'usuari amb els elements gràfics o retornada pel model – és el *model de vista*. En aquest context d'aquest patró, la vista és únicament responsable de detectar passivament els canvis en el model de vista i d'assegurar-se que els elements gràfics mostrin la informació actualitzada.

Aquest enfocament és especialment útil per evitar que els *canvis de configuració* (com ara la rotació del dispositiu) que provoquen la destrucció de les activitats, causin també la pèrdua d'informació.

Requeriments

- PI1 - Primers passos amb Android Studio
- PI2 - Activitats i d'altres components fonamentals

Objectius

1. Aprendre a crear i associar models de vista per a les activitats (vista).
2. Aplicar el patró observador per a establir comunicació des del model de vista cap a la vista (activitat) utilitzant els observadors/observables de l'*Android Sdk*, els quals són sensibles als cicles de vida de les activitats i models de vista.

Recursos

Us proporcionem:

- `MVVMExample.zip`: projecte base d'Android Studio descarregable del campus virtual, per fer el seguiment d'aquest guió i resoldre els exercicis proposats.

1 Model–Vista–Model de vista (MVVM)

A Android, les activitats actuen alhora com a vista, determinant la inclusió i l'aparença dels elements gràfics (mitjançant el seu fitxer de disposició), i com a controlador que caça i delega esdeveniments provocats per la interacció de l'usuari amb els elements gràfics (mitjançant el seu codi Java). De la comunicació amb les classes del model i de transformar la informació retornada pel model en un format ingerible per la vista, se n'encarreguen els **models de vista**. També, i sobretot, de fer el paper d'*expert en informació* del que hagi de mostrar la vista (activitat).

Pot ser raonable preguntar-se, en primer lloc, *si cal tenir guardades dades que la vista no estigui actualment mostrant.* Doncs bé, penseu que les interfícies gràfiques són dinàmiques i que, per tant, la vista pot haver de mostrar i ocultar informació segons la interacció de l'usuari amb la interfície gràfica. De fet, mentre l'usuari romangui en la mateixa activitat, hi ha una alta probabilitat que necessiti recuperar qualsevol informació que l'activitat pugui proporcionar tot i no ser sempre visible.

Per què no guardar les dades a les activitats i prescindir dels models de vista?

La raó principal per no emmagatzemar les dades directament a les activitats és evitar-ne la pèrdua quan es produeix un canvi de configuració. Exemples típics d'aquests canvis són la rotació del dispositiu, un canvi de tema o un canvi d'idioma (vegeu Fig.1). Quan això passa, Android atura i destrueix la instància de l'activitat per crear-ne una de nova amb la configuració actualitzada. Malauradament, la instància recreada només conservarà certs estats de la interfície, com el contingut d'un `EditText` o la posició d'un `ScrollView`, que Android desa automàticament en un objecte de tipus `Bundle`.

Un `Bundle` és un contenidor de dades dissenyat per passar informació entre components d'Android, incloent-hi la transmissió entre la instància original d'una activitat destruïda i la nova instància. Abans de la destrucció, el sistema invoca el mètode de callback `onSaveInstanceState` per fer el desat de la informació en el `Bundle` que serà, posteriorment, passat per paràmetre al mètode `onRestoreInstanceState` invocat en la nova instància de l'activitat.

Tot i que és possible sobreescriure `onSaveInstanceState` i `onRestoreInstanceState` per desar/carregar dades addicionals, aquest enfocament té

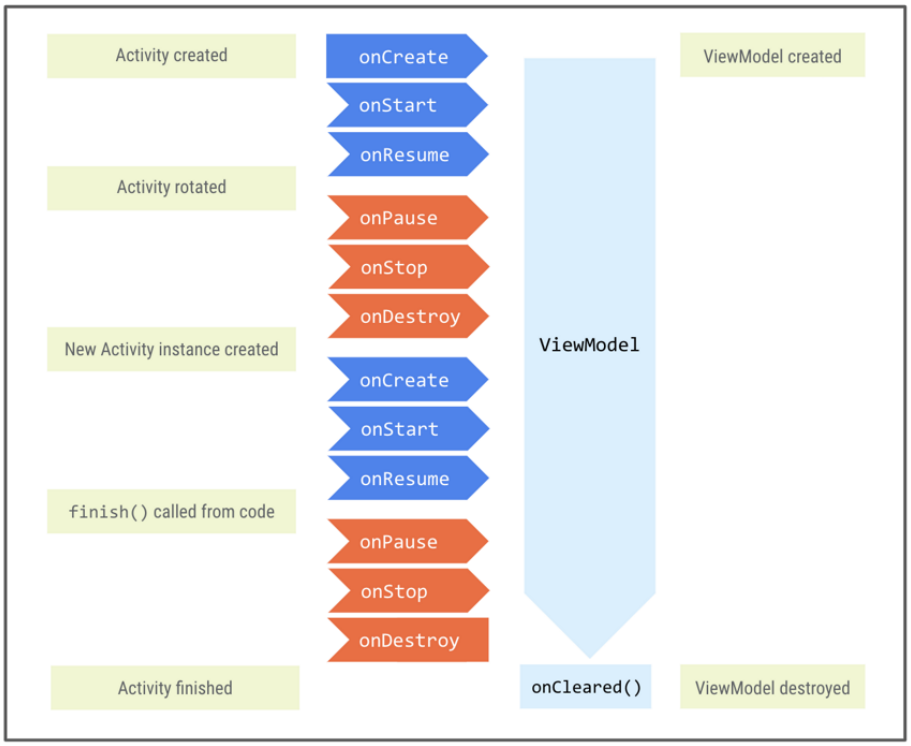

Figure 1: Els canvis de configuració destrueixen les activitats, però no els models de vista.

limitacions: (1) els `Bundle` tenen una mida màxima d'1 MB, ja que estan dissenyats per al seu ús en intencions (*intents*) i pas de dades lleugeres[1]. Això significa que no són adequats per emmagatzemar estructures grans, com llistes amb moltes imatges o objectes complexos; i (2), el procés de guardat requereix serialització o parcelització de les dades, fent-lo un mecanisme ineficient per grans quantitats de dades (+1 MB).

Per tant, deixarem que `onSaveInstanceState` i `onRestoreInstanceState` gestionin automàticament la preservació dels estats dels elements de la interfície, segons el que Android consideri oportú. I les dades que no estiguin directament vinculades als elements de la UI, sinó que provinguin del model (i haguessin de tornar a ser demanades en cas de destruir-se l'activitat), les emmagatzemarem en models de vista.

[1] En aquest context, no es crida l'`onRestoreInstanceState`, sinó només `onCreate`, que també té un paràmetre `Bundle` a partir del qual rebre les dades d'una activitat, en aquest cas d'una altra activitat diferent.

1.1 Models de Vista

Els models de vista s'han de crear i associar amb les activitats corresponents. Tenen un cicle de vida finit i acaben sent destruïts pels mateixos motius que les activitats, amb l'excepció del canvi de configuració. Veiem-ho més en detall.

Creació, associació i cicle de vida dels models de vista. Els models de vista d'Android s'implementen estenent (fent `extends`) de la classe base `ViewModel` i han de ser associades a les activitats corresponents en temps d'execució. A partir del moment de l'associació, els `ViewModel` seran conscients del cicle de vida de les activitats i perviuran als seus canvis de configuració. I no només això, sinó que quan una activitat sigui destruïda per un canvi de configuració, el seu model de vista serà automàticament re-associat a la nova instància de l'activitat de manera automàtica i transparent. Torneu a fer un cop d'ull a la Fig.1. Ara bé, els models de vista també tenen un cicle de vida finit que en algun moment podria acabar-se, destruint-lo.

Destrucció dels models de vista. El model de vista només es destrueix quan l'activitat a la que està associada és també destruïda, però per un motiu diferent al d'un canvi de configuració: (1) quan es passa a una nova activitat i el programador invoca explícitament el `finish()` de l'activitat actual just després de llençar l'intent de transició a la nova activitat o (2), quan es clica el botó de retrocés i es crida implícitament `finish()` de l'activitat actual. En ambdues situacions, l'usuari està abandonant intencionadament l'activitat i, llavors, té sentit acabar amb el model de vista. Podem dir, per tant, que mentre una activitat no sigui intencionadament destruïda, el model de vista estarà disponible i, amb ell, les seves dades accessibles per part de l'activitat.

Normalment, tindrem un `ViewModel` per cada `Activity`, perquè les activitats són inherentment porcions petites de la GUI de l'aplicació. Si l'activitat i el model de vista creixen massa, ens caldrà plantejar-nos la possibilitat de dividir l'activitat i el model de vista en d'altres activitats i models de vista més petits (amb menys responsabilitats).

1.2 Observadors sensibles al cicle de vida

Les aplicacions d'Android, per defecte, s'executen un únic fil d'execució (o *thread*). Per tant, si enlloc de permetre que el fil dediqui el seu temps als processos de la UI, el dediquem a altres coses que requereixin un temps de comput massa elevat, la UI quedarà penjada[2].

Aquest és el problema que ens podem trobar quan s'executen les crides que passen pel model: l'activitat demana de fer alguna cosa al model de vista i, indirectament, el model de vista li ho demana de fer al model. Això vol dir que

[2]Application Not Responding (ANR). Més informació a `https://developer.android.com/topic/performance/vitals/anr`.

l'activitat s'haurà d'esperar al retorn de la crida del primer mètode durant un temps indefinit. És per això que cal un mecanisme de comunicació asíncron, gràcies al qual l'activitat pugui invocar un mètode del model de vista i seguir amb la seva execució fins que el model de vista ens avisi de la completació de la tasca. D'altra banda, també voldríem que la feina del model es fes un fil d'execució separat, però això ho deixarem per més endavant. Per ara, podeu assumir que serà així.

El que farem per a que la vista pugui seguir amb la seva execució després de demanar alguna cosa al model de vista, és aplicar el *patró observable*. A diferència d'altres aplicacions del patró observador, aquí els observable i l'observador, no seran les classes que representen el model de vista i la vista (activitat) respectivament, sinó que seran cadascuna de les pròpies dades.

1.2.1 Observadors i observables

Afortunadament, l'SDK d'Android proporciona una sèrie de classes per implementar el patró observador i permetre **la comunicació asíncrona des del model de vista cap a la vista**. L'avantatge principal d'utilitzar aquestes classes, respecte d'un patró observador propi, és que aquestes ja implementen la consciència dels cicles de vida de les activitats / models de vista. Són les classes `LiveData<T>`/`MutableLiveData<T>` pels observables i la classe `Observer<T>` per l'observador. Són classes, no interfícies, perquè són instanciables i el tipus de dada es pot variar gràcies a la instanciació del *template*, és a dir, `T`.

En el moment d'iniciar una activitat (en el mètode `onStart`), inicialitzarem un atribut observador `Observer<T>` sobre un tipus de dada `T`, que es subscriurà a un observable `LiveData<T>`. L'observable `LiveData<T>` serà un atribut definit en el model de vista, accessible a través d'un *getter*. Un cop establer-ta l'observació, qualsevol canvi en la dada de observable `LiveData<T>`, causarà una notificació en `Observer<T>` de l'activitat per a que aquesta pugui actualitzar la UI amb el canvi. Feu un cop d'ull a la Fig.2 per veure el flux d'informació entre la vista, model de vista i model. De la **comunicació asíncrona des del model cap al model de vista** ens n'encarregarem en un altre moment, tot i que podeu assumir que serà un mecanisme d'observació similar (sense fer ús d'observadors/observables específics de l'*Android Sdk*).

Quan té lloc la destrucció de l'activitat, els `Observer<T>` informen als `LiveData<T>` als que estan subscrits de que cal cancel·lar la subscripció. La nova instància de l'activitat recreada inicialitza els seus propis observadors i els torna a subscriure als observables, evitant invocar mètodes d'instàncies destruïdes o fuites de memòria.

1.2.2 Mutabilitat dels observables

Fins ara hem mencionat que hi havia dos tipus d'observables, `LiveData<T>` i `MutableLiveData<T>`, però no hem entrat a discutir les seves diferències.

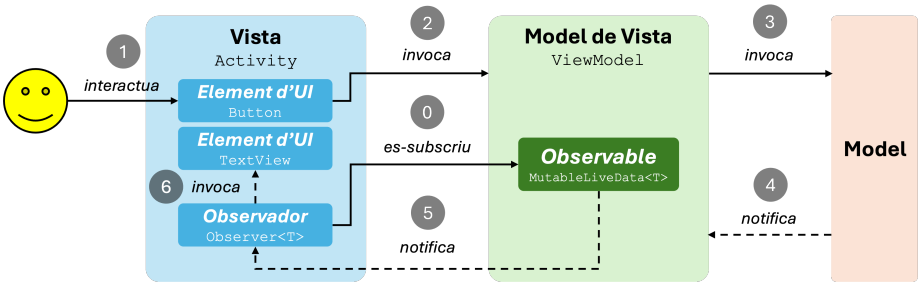

Figure 2: En el moment de creació de la vista (`Activity`), aquesta inicialitza un observador per observar un observable `MutableLiveData<T>` del model de vista (0). Quan l'usuari provoca un esdeveniment d'interacció amb un element de l'`Activity` (p. ex. un `Button`) (1), la vista delega la feina en el `ViewModel` (2) i, alhora per indirecció, el `ViewModel` en el model (3). Després d'un temps indeterminat i, per tant, de manera síncrona el model notifica i fa arribar la dada al `ViewModel` (4). Aquesta dada rebuda, l'emet el model de vista per l'observable `MutableLiveData<T>` de manera que l'observador de l'`Activity` sigui notificat i rebi també la dada (5). L'últim pas és que la vista presenti la dada en algun element de la UI (p. ex. un `TextView`) (7).

`MutableLiveData<T>` és una classe derivada de `LiveData<T>` que afegeix la capacitat de mutar la data de l'observable, per tant, la capacitat d'emetre dades. En canvi, `LiveData<T>` és immutable (només lectura) i, per tant, no se'n pot modificar la dada; només serveix per subscriure-s'hi i rebre dades.

Per a veure una implementació de l'MVVM amb aquestes eines, il·lustrem un exemple senzill d'una activitat de logueig amb el seu model de vista corresponent.

1.3 Exemple: logueig d'usuari

Després d'emplenar el nom d'un usuari i la contrassenya en un parell d'`EditText`, l'usuari fa clic a un botó "log-in" per loguejar-se. La vista detecta la interacció de clic amb el botó i, en el mateix mètode `onClick` de l'escoltador, arreplega el nom d'usuari i la contrassenya dels `EditText` i invoca un mètode del model de vista, `void logIn(username, password)`. Aquest mètode del model de vista delega la tasca de logueig a una façana (o servei[3]) d'autenticació. Un cop obtingut el resultat per part de la façana, el model de vista prepara una presentació del resultat útil per la vista i li passa, per a que aquesta li mostri a l'usuari a través d'un `Toast`.

En aquest exemple, al model de vista hi podriem definir un `MutableLiveData<Boolean>` que representés l'èxit/fracàs del logueig. Ara bé, en cas

[3]Amb *servei* ens referim al concepte més abstracte de la paraula (alguna cosa que serveix), no a la classe `Service` de l'Android Sdk.

d'èxit, voldrem passar la informació del client i, en cas de fracàs, un missatge d'error. El que fem, llavors, és definir una classe `LogInState` que encapsuli tota la informació i ens permeti passar-la de manera íntegra. Com que és una classe que no tindrà sentit fora de la lògica d'aquest model de vista (i activitat), la definirem com a classe interna del model de vista `LogInViewModel`. Així doncs, l'observable que definirem al model de vista finalment serà un `MutableLiveData<LogInState>`. També us podeu estalviar la implementació d'aquest tipus de classes d'estat utilitzant una classe `StateLiveData<T>` (consulteu l'apèndix A per més informació).

A continuació, teniu el `LogInViewModel` i la `LogInActivity` que implementen aquest escenari. Veiem primer el **`LogInViewModel`**:

```
1  public class LogInViewModel extends ViewModel {
2    /* Model */
3    private final LogInService logInService;
4    /* LiveData */
5    private final MutableLiveData<LogInState> logInState;
6
7    /* Constructor */
8    public LogInViewModel() {
9      logInService = new LogInService();
10     logInState = new MutableLiveData<>();
11   }
12   /**
13    * Returns the state of the log-in
14    * @return the state of the log-in
15    */
16   public LiveData<LogInState> getLogInState() {
17     return logInState;
18   }
19
20   /**
21    * Logs in the user
22    * @param username the username
23    * @param password the password
24    */
25   public void logIn(String username, String password) {
26     logInService.logIn(
27       username,
28       password,
29       new OnLogInListener() {
30         @Override
31         void onLogInSuccess(Client client) {
32           logInState.postValue(new LogInState(true, client));
33         }
34         @Override
35         void onLogInError(Exception e) {
36           logInState.postValue(
```

```
            new LogInState(false, e.getMessage())
          );
        }
      }
    );
  }

  /**
   * Inner data container class carrying log-in state
   */
  public static final class LogInState {
    public final boolean success;
    public final Client client; /* Cas d'èxit */
    public final String message; /* Cas de fracàs */

    public LogInState(boolean success, Client client) {
      /* Cas d'èxit */
      this.success = success;
      this.client = client;
      this.message = null;
    }

    public LogInState(boolean success, String message) {
      /* Cas de fracàs */
      this.success = success;
      this.client = null;
      this.message = message;
    }
  }
}
```

Si us heu fixat en el mètode `logIn(String username, String password)`, haureu vist que fa una crida a l'objecte del model que representa el servei d'autenticació, `LoginService`, i que li passa un escoltador per l'últim paràmetre. Tornem a remarcar, que no serà així com ho farem més endavant, però ara mateix ens serveix per simular l'asincronia: quan el model acabi, cridarà un dels dos mètodes de *callback* de l'escoltador (`onLogInSuccess` o `onLogInFailure`), el qual farà l'emissió pertinent a través del `MutableLiveData<LogInState> logInState`.

Veiem com queda la **`LogInActivity`**:

```
public class LogInActivity extends AppCompatActivity {
  /* ViewModel */
  private LogInViewModel logInViewModel;
  /* View binding */
  private ActivityLogInBinding binding;

  /**
   * Called when the activity is being created.
```

```
9       * @param savedInstanceState
10      */
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13       super.onCreate(savedInstanceState);
14       /* Set view binding */
15       binding = ActivityLogInBinding.inflate(getLayoutInflater())
             ;
16       setContentView(binding.getRoot());
17
18       /* Initializations */
19       initWidgetListeners();
20       initViewModel();
21     }
22
23     /**
24      * Initialize the listeners of the widgets.
25      */
26     private void initWidgetListeners() {
27       binding.btnLogIn.setOnClickListener(ignoredView -> {
28         // Delegate the log-in logic to the viewmodel
29         logInViewModel.logIn(
30           String.valueOf(binding.etLoginUsername.getText()),
31           String.valueOf(binding.etLoginPassword.getText())
32         );
33       });
34     }
35
36     /**
37      * Initialize the viewmodel and its observers.
38      */
39     private void initViewModel() {
40       /* Init viewmodel */
41       logInViewModel = new ViewModelProvider(
42           this
43       ).get(LogInViewModel.class);
44
45       initObservers();
46     }
47
48     /**
49      * Initialize the observers of the viewmodel.
50      */
51     private void initObservers() {
52       /* Observe the login state */
53       logInViewModel.getLogInState().observe(
54         this, // Awaraness of activity's lifecycle
55         new Observer<LogInViewModel.LogInState>() {
56           @Override
57           public void onChanged(
```

```
58            LogInViewModel.LogInState logInState
59          ) {
60            if (logInState.success) {
61              Intent intent = new Intent(
62                LogInActivity.this,
63                LoggedInActivity.class
64              );
65              intent.putExtra(
66                "CLIENT_ID",
67                logInState.client.getId()
68              );
69              startActivity(intent);
70            } else {
71              Toast.makeText(
72                this,
73                logInState.message,
74                Toast.LENGTH_SHORT
75              ).show();
76            }
77          }
78        }
79      );
80    }
81  }
```

Primer, noteu que quan `logInViewModel.getLogInState()` (línia 53), el que demanem és un `LiveData<LogInViewModel.LogInState>`, que és l'observable (de només lectura), no la dada en si, que sería el `LogInViewModel.LogInState`. Segon, que un cop obtingut l'observable (encara a la línia 53), n'invoquem el mètode `observe` passant-li dues coses: (1) una referència de l'activitat, `this`, per a fer l'observador sensible al cicle de vida de l'activitat i (2) la implementació anònima de l'observador, `Observer<LogInViewModel.LogInState>`. En la implementació anònima és definim el comportament dels callbacks; en aquest cas, un únic mètode: `void onChange(LogInViewModel.LogInState)`. Aquest callback és el que serà automàticament invocat quan al `LogInViewModel` es modifiqui del valor de l'observable (és a dir, es produeixi una emissió) amb la instrucció `logInState.postValue(...)`. Quan es cridi el callback, l'observador presentarà la informació encapsulada en l'objecte `LogInViewModel.LogInState` en un missatge `Toast`[4].

Per últim, fixeu-vos també amb la inicialització del model de vista. Es fa a l'activitat a través d'un `ViewModelProvider` (línia 41), enlloc d'utilitzar la instrucció `new`. El proveïdor és la manera d'assegurar-se d'enllaçar els cicles de vida de l'activitat i el model de vista. L'inconvenient del proveïdor és que ens impedeix passar-li paràmetres al constructor del model de vista. Tot i això, a vegades serà necessari el pas de paràmetres. Per saber com fer-ho, consulteu l'apèndix B.

[4] Els `Toast` són missatges de notificació flotants.

Podeu realitzar els l'**Exercici 1**.

2 Exercicis

Us donem l'aplicació *Xoping* amb quatre activitats: `LogInActivity` i `SignUpActivity`, `ShoppingActivity` i `ViewProductDetailsActivity`. Ens centrarem en aquesta última, assumint que la resta funcionen. Lues dues primeres les heu treballat en la PI anterior i pel nom ja intuïu què fan. La `ShoppingActivity` mostra la llista de productes disponibles a la botiga, els quals es poden clicar. La treballarem a la següent PI. En aquesta PI, ens centrarem en `ViewProductDetailsActivity`, que és l'activitat que s'obra quan s'ha clicat un producte de la llista de productes de la `ShoppingActivity`. La disposició l'activitat (`res/layout/activity_view_product_details`) us la donem feta per agilitzar el procés, però haureu de fer la resta. El model de vista corresponent a l'activitat, `ViewProductDetailsViewModel` també us el donem fet i només s'hi hauran d'afegir unes poques línies de codi.

Exercici 1. Feu que es mostrin els detalls del producte seleccionat a la `ViewProductDetailsActivity`. Els passos a seguir són (us podeu guiar també pels llocs del codi on diu "EXERCICI 1"):

1. Aneu a veure el mètode `startViewProductDetailsActivity(Product product)` de la `ShoppingActivity` només per veure com es llança l'intent explícit per a navegar cap a la `ViewProductDetailsActivity` i mostrar el producte seleccionat. No cal mirar res fora del mètode ara mateix, només cal que intenteu entendre com es passen les dades. a través de l'intenció: un `String`, que representa l'identificador de client, i un `Product`).

2. A la `ViewProductDetailsActivity`, fixeu-vos en com s'obté el producte passat per l'intent[5]. L'objecte `Product` recuperat el passarem als mètodes que implementarem a continuació.

3. Reviseu la disposició l'activitat (`res/layout/activity_view_product_details`) per conèixer els identificadors dels diversos elements gràfics.

4. Implementeu els mètode `void initWidgets(Product product)` per a emplenar els elements de la disposició amb els detalls del producte. La part de la imatge ja us la donem feta (amb la llibreria Picasso). Falta mostrar el nom del producte, el preu i la descripció.

[5] Per més detalls sobre el pas d'objectes (és a dir, tipus no-primitiu) entre activitats, reviseu l'apèndix C.

5. Implementeu `void initWidgetListeners(Product product)` per a establir escoltadors pels botons de "Buy", "+" i "-". Feu que aquests botons causin la crida dels mètodes `void incrementQuantity()`, `void decrementQuantity()` i `buyProduct()`. La seva implementació la fareu en l'**Exercici 2**.

Exercici 2. Inicialitzar el model de vista `ViewProductDetailsViewModel` a l'activitat i implementar els mètodes del model de vista que els escoltadors de l'activitat hagin d'invocar. Els passos són (us podeu guiar també pels llocs del codi on diu "EXERCICI 2"):

1. Implementar el mètode `void initViewModel()` de `ViewProductDetailsViewActivity`. Recordeu de fer servir el `ViewModelProvider`.
2. Implementeu els mètodes del model de vista per incrementar/decrementar quantitat:
   - `void increaseQuantity()`, per a que incrementi la quantitat de l'observable `quantityState`. Feu servir el mètode `postValue` per emetre el valor un cop incrementat.
   - `void decreaseQuantity()`, anàlogament a `increaseQuantity`, però fent el decrement. Controleu-hi que la quantitat només es pugui decrementar si resulta estrictament positiva.
3. Implementeu el mètode `void initObservers()` de `ViewProductDetailsActivity` per crear un observador de l'observable `quantityState` i que quan detecti el canvi actualitzi el valor del `TextView` que mostra la quantitat de producte.

Exercici 3. Afegiu el codi necessari per a confirmar la compra del producte quan es cliqui el botó "Buy" i afegiu la restricció de que no es puguin comprar menys de 1 unitat de producte o més de 10. En cas que tot vagi bé, sortiu de l'activitat dels detalls per a tornar a l'activitat anterior. Per aquesta ocasió, utilitzeu ubn `StateLiveData<Void> buyState` com a observable al model de vista. Us podeu guiar també pels llocs del "EXERCICI 3".

Appendix A Classe auxiliar `StateLiveData<T>`

Quan examineu el codi base trobareu que, en ocasions, fem ús d'un tercer tipus d'observable, anomenat `StateLiveData<T>`. Aquesta classe no és oficial de cap *Android Sdk*, sinó una classe implementada per un tercer. Ens estalvia la implementació de classes internes al model de vista per passar el booleà d'èxit/fracàs i el missatge d'error (del fracàs) en addició a la dada que voldríem retornar per l'observable en cas d'èxit.

`StateLiveData<T>` ofereix mètodes d'emissió alternatius al `postValue` i més específics: `postSuccess(T data)`, `postError(Throwable throwable)`, `postLoading()` i `postCompleted()`. En qualsevol cas, l'observador rebrà un objecte `StateData` amb un mètode `getStatus()` que ens permetrà filtrar pel tipus d'emissió (`CREATED`, `SUCCESS`, `ERROR`, `LOADING` i `COMPLETED`) s'ha realitzat amb un `switch` durant la recepció de l'emissió (al callback `onChanged` de l'observer). En cas de `SUCCESS`, invocarem el `getData()` de l'`StateData` i, en cas de `ERROR`, el `getError()`.

Trobareu el codi d'aquesta classe al paquet `utils` dins del mateix codi base. Podeu veure com s'ha aplicat per les activitats/models de vista de log-in i sign-up.

Appendix B Pas de paràmetres als models de vista (o *viewmodels*)

Per inicialitzar un model de vista, fem servir un proveïdor de models de vista (classe `ViewModelProvider`). En algun lloc de l'activitat, fem:

```
1  MyViewModel myViewModel = new ViewModelProvider(
2    this
3  ).get(MyViewModel.class);
```

Ara bé, l'única cosa que accepta el `ViewModelProvider`, a més del `this`, seria una instància de la classe `ViewModelProvider.NewInstanceFactory`. Si volem passar paràmetres, per tant, haurem de crear una *factory concreta* que estengui (*extends*) d'aquesta *factory genèrica* (`ViewModelProvider.NewInstanceFactory`) per a que creï els nostres models de vista amb els paràmetres que vulguem passar-los pel constructor.

La *factory concreta*, diem-li `Factory`, la podem definir com una classe interna del nostre model de vista:

```
1
2  public class MyViewModel extends ViewModel {
3    /* Attributes */
4    private MyParameterType myParameter;
```

```
5
6     /* Constructor */
7     public MyViewModel(MyParameterType myParameter) {
8       this.myParameter = myParameter;
9     }
10
11    ...
12
13    /*
14      Classe interna Factory de MyViewModel per a la creació
15      del viewmodel per poder fer pas paràmetres al constructor
16      del viewmodel.
17    */
18    public static class Factory
19      extends ViewModelProvider.NewInstanceFactory {
20
21      private final MyParameterType myParameter;
22
23      public Factory(MyParameterType myParameter) {
24        this.myParameter = myParameter;
25      }
26
27      @NonNull
28      @Override
29      @SuppressWarnings("unchecked")
30      public <T extends ViewModel> T create(
31        @NonNull Class<T> modelClass
32      ) {
33        return (T) new MyViewModel(myParameter);
34      }
35    }
36  }
```

I, llavors, per a la inicialització del `MyViewModel` a la l'activitat:

```
1   MyParameterType myParameter = ... // Obtenció del paràmetre
2
3   myViewModel = new ViewModelProvider(
4     this,
5     new MyViewModel.Factory(myParameter) // Pas de myParameter
6   ).get(MyViewModel.class);
```

Appendix C Pas d'objectes entre activitats

Per a poder passar objectes entre activitats, cal que els objectes implementin la interficie `Serializable` o `Parcelable`. Mentre que la primer alternativa és immediata i fàcil, afegint el típic `implements Serializable` a la capçalera de definició de la classe, també és una alternativa més lenta. Els serialitzables

s'han de convertir en un bytearray i després deserialitzar.

Els `Parcelable` són més ràpids perquè obliguen a especificar explícitament mètodes que s'utilitzen per la conversió a bytearray. I com que no és una solució genèrica, com en el cas dels `Serializable` la conversió i de-conversió resulten més ràpides (al voltant del doble). Afortunadament, a Android Studio teniu el plugin "Android Parcelable Code Generator" que permet implementar aquests mètodes de manera automàtica.

Un cop instal·lat el plug-in, aneu a la implementació de la vostra classe (dins del fitxer Java), feu clic amb el botó dret a sobre del nom de la classe (p. ex. `Product`), feu "Generate..." i, finalment, "Parcelable".

Per passar un objecte `Parcelable` a través d'un `Intent`, en l'activitat d'origen:

```
1  /**
2   * Starts the ViewProductDetailsActivity.
3   * @param product the product to be shown
4   */
5  private void startViewProductDetailsActivity(Product product)
       {
6    Intent intent = new Intent(this, ViewProductDetailsActivity
         .class);
7    intent.putExtra("CLIENT_ID", clientId);
8    intent.putExtra("PRODUCT", product); // Product class
         implements Parcelable
9    startActivity(intent);
10 }
```

I recuperar-lo des de l'activitat de destí (p. ex. a l'`onCreate(...)`):

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3    super.onCreate(savedInstanceState);
4    ...
5    /* Get Product from Intent that created this activity */
6    Product productModel = (Product) getIntent()
7        .getParcelableExtra("PRODUCT");
8    ...
9  }
```