Vîlculescu Mihai-Bogdan Grupa 232
Martac Dana-Maria
Jurcăneanu Ionuţ-Adrian
Drăguţescu Mihai-Valentin
Borş Andrei

Arhitectura produsului software
Restaurant
Admin

Cuprins

1. Introducere

1.1. Scurtă descriere

Aplicaţia *Restaurant-Admin* a fost realizată folosind IDE-ul *Android Studio.* Limbajul de programare utilizat a fost *Java,* iar partea de design a fost realizată utilizând *XML.*

Aplicaţiile Android au la bază conceptul de *Activitate*. O activitate nu este nimic altceva decât o ”fereastră” din aplicaţiile desktop. Orice activitate moşteneşte clasa *Activity.*

Java este un limbaj OOP, iar din această cauză schema relaţiilor dintre clase este destul de complexă. În următoarele secţiuni, vom trece prin clasele importante ale programului.

1.2. Server local

Pentru dezvoltarea aplicaţiei, am hotărât utilizarea atât a unui server local, cât şi a unui extern. Serviciul folosit pentru dezvoltarea locală a bazei de date a fost *SQLite.* Acesta este un sistem bazat pe SQL pus la dispoziţie de către dezvoltatorii Android. Acesta permite modelarea unei baze de date sub forma unei *clase Java,* care conţine metode pentru a facilita comunicarea cu această bază de date.

```java
/**
 * Clasa care creeaza baza de date locala SQLite cu care lucreaza aplicatia.
 */
public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "PLANIFICARE_RESTAURANT.db";
    private static final String TABLE_NAME = "TABEL_CELULA";
    private static final String COL_1 = "CELULA_ID";
    private static final String COL_2 = "VECIN_SUS";
    private static final String COL_3 = "VECIN_STANGA";
    private static final String COL_4 = "CONTINUT";
    private static final String COL_5 = "NUMAR";

    public DatabaseHelper(Context context) { super(context, DATABASE_NAME, factory: null, version: 1); }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " + TABLE_NAME + " (CELULA_ID TEXT,VECIN_SUS TEXT,VECIN_STANGA TEXT,CONTINUT TEXT,NUMAR INTEGER)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }

    public boolean insertData(String celula_id, String vecin_sus, String vecin_stanga, String continut, int numar) {

        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(COL_1, celula_id);
        contentValues.put(COL_2, vecin_sus);
        contentValues.put(COL_3, vecin_stanga);
```

Figura 0.1. Clasă SQLite

Prin crearea diverselor metode de *inserare, selectare, update* se modelează o bază de date obişnuită. Aceasta este creată la prima rulare a aplicaţiei, urmând ca apoi să fie doar actualizată.

1.3. Server extern

Serviciul folosit pentru stocarea externă a datelor este *Firebase.* Acesta este un sistem dezvoltat de *Google* în scopul uşurării comunicării dintre o aplicaţie şi un server. Un *API* de *Firebase* poate fi implementat în mai multe feluri de aplicaţii: *Android, IOS, Web.*

Acest serviciu foloseşte un sistem *NoSQL,* bazat pe memorarea datelor ca obiecte de tip *JSON.* Lucrul cu acest *API* este mult simplificat, întrucât este necesară doar declararea unei referinţe către un obiect de tip Firebase, iar apoi se pot apela metode de tip *get, update, insert, ...*

Cum datele sunt memorate ca obiecte, atunci pentru ca lucrul cu acest server să fie mai uşor, am creat clase model care conţin exact datele pe care dorim să le găsim în baza de date. Vom vorbi de ele mai jos.

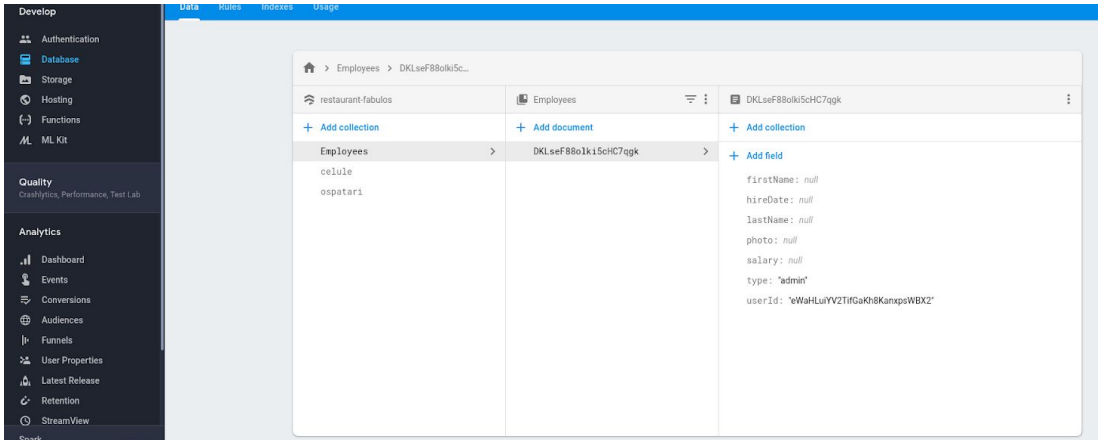

Figura 0.2. Interfaţa Firebase

În Firebase, datele sunt stocate astfel:

★ ***Tabel = Colecţie;***
★ ***Rând din tabel = Document;***
★ ***Coloană = Câmp (Field).***

Această distincţie provine din faptul că datele sunt transmise ca obiecte, ceea ce pentru un limbaj bazat pe OOP este ideal.

2. Pachetul *Models*

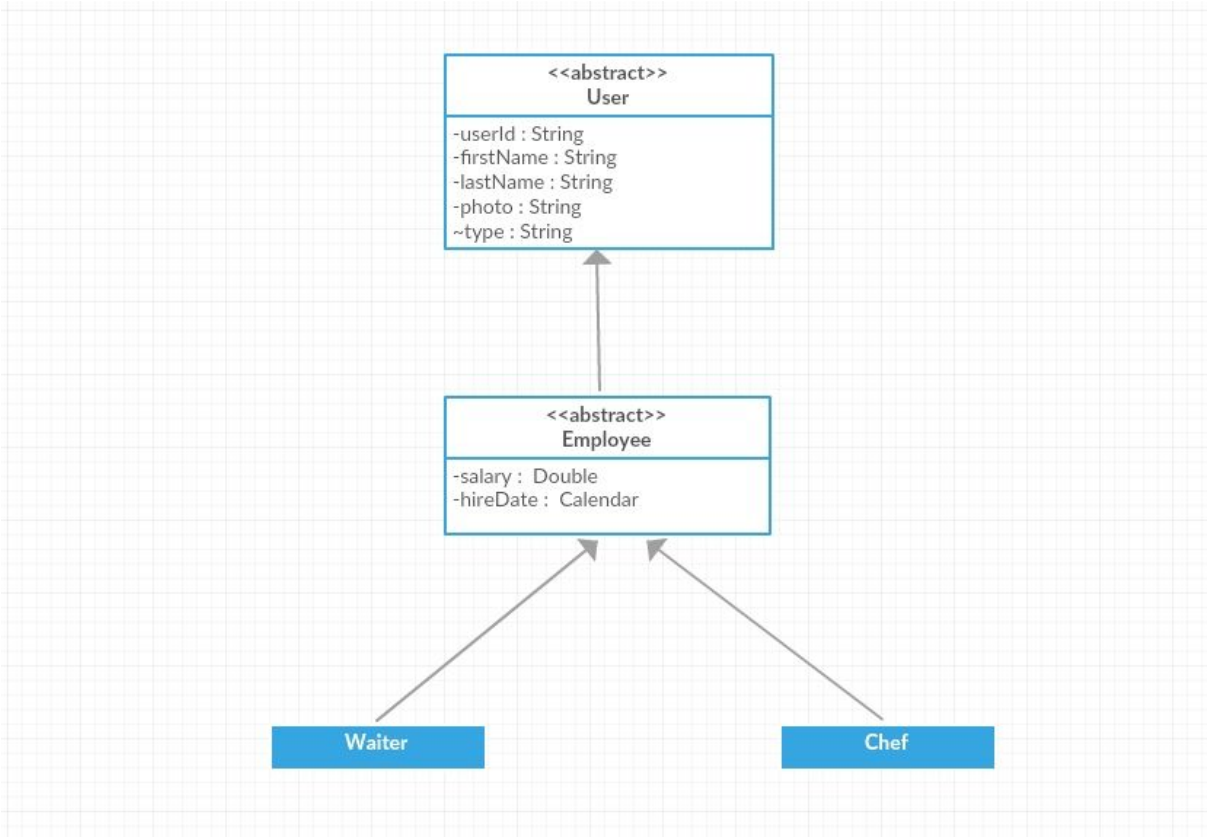

Figura 1. UML clase din pachetul models

În cadrul pachetului *models,* se regăsesc modelele utilizate în aplicaţie. Cum aplicaţia este despre un restaurant, clasele importante sunt tipurile de angajaţi. Astfel, ierarhia claselor este următoarea:

★ ***public abstract class*** *User* = clasa cea mai de bază. Aceasta modelează un utilizator obişnuit. Am ales un model abstract pentru că, pe viitor, vor mai fi create clase

derivate ale sale pentru a implementa mai multe tipuri de utilizatori. Conţine drept câmpuri:

- ***private*** *String userId* = id-ul utilizatorului. Va fi folosit în mod special pentru comunicarea cu baza de date externă;
- ***private*** *String firstName, lastName* = numele şi prenumele utilizatorului;
- ***private*** *String photo* = url către o poză a utilizatorului. Momentan, nu este implementat niciun feature legat de poză, dar în versiunile următoare vom implementa un fel de ”*poză de profil*”;

- `String type` = tipul utilizatorului (*"waiter", "chef"* sau *"admin").* Acesta este singurul câmp de date care nu este *private,* deoarece este accesat din subclase.
- Fiecare câmp are un *setter* şi un *getter*.

★ `public class Employee extends User implements Serializable` = clasa care modelează conceptul de angajat la modul abstract. Conţine 2 câmpuri de date: `private Double salary` şi `private Calendar hireDate` care sunt sugestive prin denumire. Nu conţine vreo altă metodă în afară de setteri şi getteri şi, bineînţeles un constructor parametrizat. Clasa *implementează* interfaţa marker *Serializable,* deoarece într-o activitate a aplicaţiei, este nevoie să se transmită un obiect de tip *Employee (*mai exact o subclasă de a sa*)* către o altă activitate, iar pentru ca acest lucru să fie posibil, clasa *Employee* trebuie să fie serializabilă (astfel toate clasele derivate devenind şi ele serializabile).

★ `public class Waiter extends Employee` = clasă concretă care modelează conceptul de *ospătar.* Nu are niciun câmp de date în plus faţă de clasele de bază. În corpul este declarat un constructor parametrizat care apelează constructorul parametrizat din clasa de bază. Pe lângă acest detaliu, tot în constructor se realizează modificarea parametrului *type,* astfel: `this.type = "waiter";` . Acum se face diferenţierea între tipurile de angajaţi, pentru a simplifica lucrul în interiorul aplicaţiei, mai ales în lucrul cu serverul.

★ `public class Chef extends Employee` = clasă concretă care modelează conceptul de *bucătar.* Aceeaşi situaţie ca şi în clasa *Waiter,* cu singura menţiune că în constructor apelul de modificare al câmpului *type* arată astfel: `this.type = "chef";` .

3. Pachetul *Activities*

În cadrul pachetului *activities* se află toate clasele java corespunzătoare activităţilor aplicaţiei. Aceste activităţi se ocupă de interfaţa grafică, afişată utilizatorului.

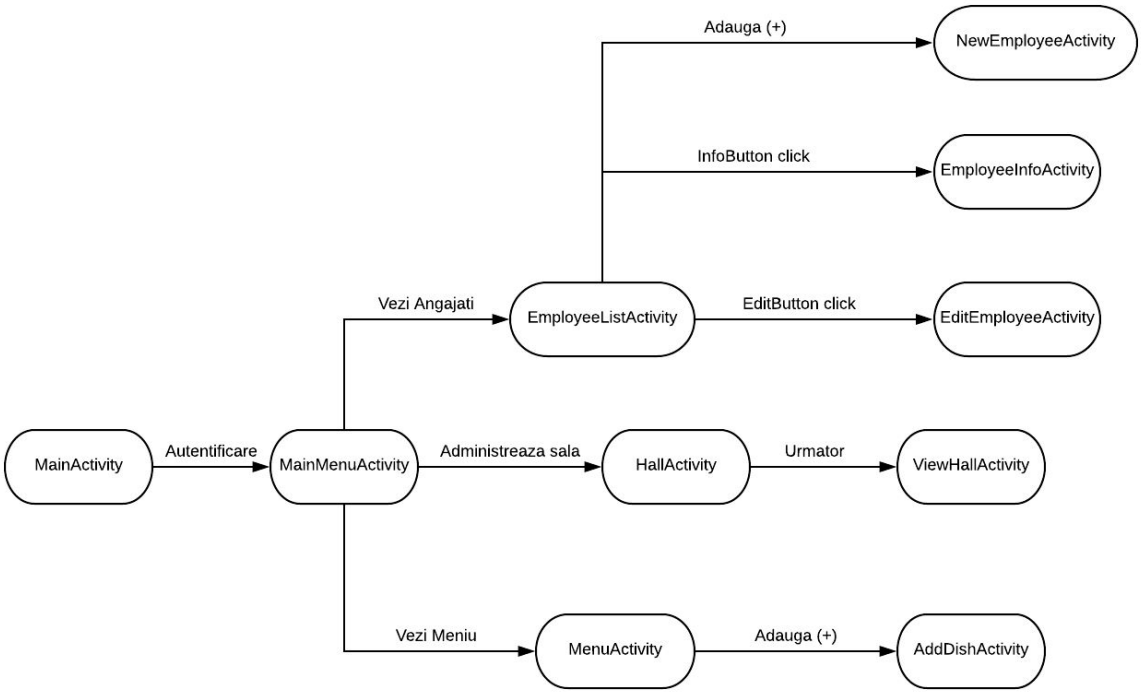

Figura 2.1. UML activităţi

3.1. Clasa MainActivity

★ ***public class*** *MainActivity* ***extends*** *AppCompatActivity* = activitatea principală a aplicaţiei. Aceasta constă într-un meniu de autentificare cu email şi parolă. Conţine următoarele câmpuri şi metode:

- ***private*** *EditText mEmailEditTxt, mPwdEditTxt* **=** câmpurile de introdus text pentru email, respectiv parolă. Prin aceste variabile se vor obţine datele introduse de utilizator;

- ***private*** *Button mLoginBtn* = butonul de autentificare. La selectarea sa se va încerca autentificarea utilizatorului prin serverul extern. Dacă procesul se

încheie cu succes, utilizatorul este redirecţionat către meniul principal, altfel este afişat un mesaj de eroare;

- ***private*** *void getSignedUser(FirebaseUser user)* = metodă care este apelată în *onStart (*adică la fiecare pornire a activităţii). Aceasta primeşte ca parametru un utilizator *Firebase* şi, dacă acest obiect nu este null, înseamnă că este un utilizator deja conectat, astfel se schimbă activitatea şi utilizatorul este transmis către activitatea nouă prin *intent.putExtra("user", user);* . Această operaţie trimite obiectul *user* către activitatea nouă înregistrată în *intent* (MainMenuActivity.class).

3.2. Clasa MainMenuActivity

★ ***public class*** *MainMenuActivity* ***extends*** *AppCompatActivity* = activitatea unde se află meniul principal al aplicaţiei. Constă, pentru admin, în 3 butoane care îl redirecţionează către alte activităţi astfel:
- ***private*** *Button* ***mSeeEmployeesBtn*** *-> EmployeesListActivity;*
- ***private*** *Button* ***mSeeLayoutBtn*** *-> HallActivity;*
- ***private*** *Button* ***mSeeMenuBtn*** *-> MenuActivity;*

★ Activitatea această va fi una dinamică odată cu implementarea utilităţilor pentru celelalte 2 tipuri de utilizator: *Ospătar* şi *Bucătar*.

3.3. Clasa EmployeesListActivity

★ ***public class*** *EmployeesListActivity* ***extends*** *AppCompatActivity* = activitatea unde sunt afişaţi toţi angajaţii adăugaţi până în momentul curent de către admin. Aceasta constă în următoarele:
- ***private*** *RecyclerView* ***mRecyclerView*** -> aici va fi afişată lista. Pentru ca un recyclerview să funcţioneze are nevoie de anumite metode, despre care vom vorbi mai în detaliu mai jos.
- ***private*** *List<Employee>* ***mEmployeeList*** -> colecţie care reţine lista de anagajaţi. Ea va fi populată de metoda ***private void*** *initData()* care parcurge baza de date Firebase şi extrage toţi angajaţii găsiţi.
- ***private*** *Button* ***mAddEmployeeBtn*** -> la apăsarea sa, va fi deschisă activitatea *NewEmployeeActivity* care permite adăugarea unui nou angajat.

3.4. Clasa NewEmployeeActivity

★ ***public class*** *NewEmployeeActivity* ***extends*** *AppCompatActivity* = activitatea unde se poate adăuga un nou angajat. Pentru a realiza acest lucru, este necesară completarea mai multor câmpuri, precum:

- *Numele complet* = trebuie respectat formatul ”Nume Prenume”;
- *Email-ul;*
- *Parola* = trebuie să aibă minim 6 caractere. Odată cu implementarea funcţionalităţilor specifice unui angajat, aceasta va fi putea fi schimbată la prima accesare a contului*;*
- *Job* = trebuie să selectată 1 dintre cele 2 opţiuni oferite: *Ospătar* sau *Bucătar.*

★ După completarea acestor câmpuri, se va selecta butonul *”Adauga angajat”*. Acesta va introduce în baza de date noul angajat, în cazul completării corespunzătoare a câmpurilor. Se va crea întâi contul angajatului prin *API-ul Firebase* utilizând email-ul şi parola furnizate. Apoi, se va adăuga tot prin *Firebase* în tabelul *Employees,* care arată astfel:

userId	***firstName***	***lastName***	***type***	***salary***	***hireDate***

Figura 2.2. Tabelul Employees

3.5. Clasa HallActivity

★ ***public class*** *HallActivity* ***extends*** *AppCompatActivity* = activitatea unde se poate modifica planificarea restaurantului folosindu-se diverse imagini sugestive. Există 3 butoane importante:

- ***private*** *Button* ***start*** -> porneşte posibilitatea de a modifica planificarea. Mai exact, se crează o matrice a cărei dimensiuni depinde de rezoluţia dispozitivului mobil. De asemenea, este apelată metoda ***public void*** *AddData()* care inserează în baza de date (locală şi externă) matricea nou creată, urmând ca aceasta să poată fii apoi actualizată în funcţie de preferinţele utilizatorului.
- ***private*** *Button* ***button_sterge*** -> şterge toate datele din bazele de date, apelând metoda ***public void*** *DeleteData()*.
- ***private*** *Button* ***button_next*** -> porneşte activitatea *ViewHallActivity.*

★ ***private final class*** *MyTouchListener* ***implements*** *View.OnTouchListener* = clasa care conţine metoda *onTouch* care detectează începutul evenimentului *drag and drop.* Mai exact, când este selectată o imagine pentru a fi adăugată în planificare, această clasă detectează acest eveniment şi salvează id-ul imaginii selectate.

★ ***class*** *MyDragListener* ***implements*** *View.OnDragListener* = clasa care conţine metoda *onDrag* care detectează mai multe evenimente de pe parcursul unei operaţii *drag and drop*. Aici, este tratat cazul *drop.* Când o imagine este lăsată deasupra unei celule, celula este actualizată cu imaginea respectivă, iar baza de date este şi ea actualizată cu acea imagine.

3.6. Clasa ViewHallActivity

★ ***public class*** *ViewHallActivity* ***extends*** *AppCompatActivity* ***implements*** *PopupMenu.OnMenuItemClickListener* = activitatea unde se poate reîncărca ultima planificare a restaurantului. Metoda *ReincarcaPlanifıfcarea()* este apelată la selectarea butonului *Reincarca.* Această metodă caută în baza de date toate celulele una câte una şi le afişează într-un RelativeLayout.

3.7. Clasa MenuActivity

★ ***public class*** *MenuActivity* ***extends*** *AppCompatActivity* = activitatea unde sunt afişate preparatele din meniu. Interfaţa este foarte asemănătoare cu cea din clasa *EmployeesListActivity,* fiindcă amândouă folosesc un *RecyclerView* unde sunt afişate toate elementele. La finalul listei, se află un buton ”+” care deschide o activitate unde se va putea adăuga un nou preparat. Momentan, activitatea de adăugare preparat nu este implementată.

4. RecyclerView

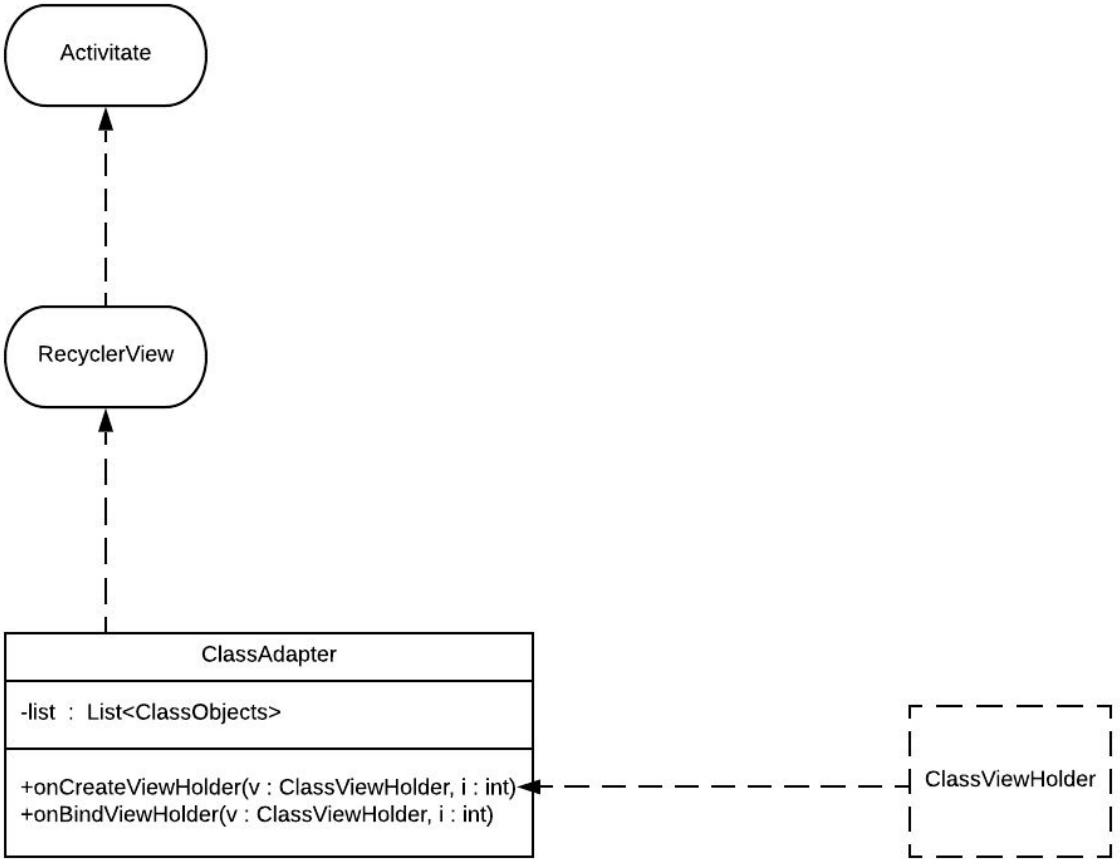

Figura 3.1. UML RecyclerView

RecyclerView este un instrument vizual specific Android, care permite afişarea unei liste de elemente numite *Card-uri.* Procesul de creare a unui *RecyclerView* este următorul:

1) ***Design-ul unui card.*** Se creează un fişier *xml* în care se stabileşte un design al unui card obişnuit care urmează a fi refolosit pentru mai multe card-uri. Fiecare card este specific, de obicei, unui anumit tip de model (*Employee, Dish, ...).*

2) ***Clasa de tip ViewHolder.***
   - ***a)*** Această clasă extinde clasa *ViewHolder* din clasa *RecyclerView* şi conţine drept câmpuri elementele vizuale corespunzătoare *unui singur card (*TextView, Button, …).
   - ***b)*** Ca metode nu are decât getteri pentru acele elemente vizuale (*views*).

3) Clasa de tip Adapter.

a) Această clasă extinde clasa generică *Adapter* din *RecyclerView* şi corespunde unui anumit tip de ViewHolder, astfel: `public class DishAdapter extends RecyclerView.Adapter<DishViewHolder>.`

b) Conţine o listă de un tip model (*Employee, Dish…);*

c) Implementează nişte metode abstracte din clasa de bază care creează câte un card corespunzător fiecărui obiect din listă. Pentru a face asta, se foloseşte de elementele vizuale din *ViewHolder.*

4) Activitatea care conţine RecyclerView.

a) Este creată o activitate care conţine, pe lângă alte elemente, un *RecyclerView.*
b) Se crează o listă de obiecte ca în *Adapter.*
c) Acea listă, după ce este populată cu date, este transmisă către un obiect de tip Adapter.
d) Se setează *adapter-ul* RecyclerView-ului ca fiind obiectul declarat la *pasul c).*