Arhitectura MVC

Model:

- Incapsuleaza datele si accesul la acestea
- Gestioneaza starea aplicatiei
- Notifica **View-ul** cu privire la modificarile aparute

Responsibilities:

- *Managing data: CRUD (Create, Read, Update, Delete) operations.*
- *Enforcing business rules.*
- *Notifying the View and Controller of state changes.*

View

- Interfata utilizator
- Logica prezentarii
- Gestioneaza interactiunea cu utilizatorul

Responsibilities:

- *Rendering data to the user in a specific format.*
- *Displaying the user interface elements.*
- *Updating the display when the Model changes.*

Example: *In a bookstore application, the View would display the list of books, book details, and provide input fields for searching or filtering books.*

Controller

- Gestioneaza interactiunile
- Coreleaza actiunile utilizatorului cu datele asociate
- Conecteaza **modelul** si **interfata**

Responsibilities:

- *Receiving user input and interpreting it.*
- *Updating the Model based on user actions.*
- *Selecting and displaying the appropriate View.*

Example: *In a bookstore application, the Controller would handle actions such as searching for a book, adding a book to the cart, or checking out.*

Model -> Clasele din aplicatie

View -> afiseaza in consola

Controller -> primeste input-ul de la user

➔ Contine o instanta View, cere date de la model si actualizeaza View-ul daca este nevoie

```
public class Controller {
    private List<Bursa> burse;
    private ConsoleView view;
//++metode
```

Main -> creeaza instantele pentru View si Controller:

```
public class Main {
    public static void main(String[] args) {
        ConsoleView view = new ConsoleView();
        Controller controller = new Controller(view);
        controller.start();
    }
}
```

Arhitectura MVP

Model:

- Nivelul datelor aplicatiei (acces la baza de date, preluare retea etc.)

View:

- Nivelul interfata utilizator
- Afiseaza datele
- Notifica **Presenter** cu privire la actiunile utilizatorilor

Presenter:

- Legatura intre data si interfata utilizator
- Preia datele din model
- Aplica logica interfetei utilizator
- Gestioneaza starea View-ului
- Reactioneaza la interactiunea utilizatorilor cu **View-ul**

MODEL -> clasele din aplicatie

VIEW - > ConsoleView (afiseaza in consola si notifica presenter-ul de input-ul user-ului)

```
public interface View {
    void displayMessage(String message);
    void displayBurse(List<Bursa> burse);
    etc.
}
```

```
public class ConsoleView implements View {
    private static final Scanner scanner = new Scanner(System.in);

    public void setPresenter(Presenter presenter) {
        this.presenter = presenter;
    }
// ++ metode care notifica presenter-ul despre input-ul userului
```

Presenter -> primeste notificarile de la view si le gestioneaza

```
public class Presenter {
    private List<Burse> burse;
    private View view;

    public Presenter(View view) {
        this.burse = Bursa.incarcaBurse();
    }
    public void attachView(View view){
        this.view = view;
    }


// ++ metode care primesc input-ul de la view si interactioneaza cu modelul
```

Main -> creeaza instantele pentru view si presenter:

```
public class Main {
    public static void main(String[] args) {
        Presenter presenter = new Presenter();
        ConsoleView view = new ConsoleView();
        presenter.attachView(view);
        view.setPresenter(presenter);
        //++metode de afisare din view daca e nevoie
    }
}
```

Diferențe între MVP și MVC

- **Rolul View-ului**: În MVP, view-ul este mai pasiv, recepționând comenzi de la presenter pentru a actualiza interfața. În MVC, view-ul poate avea un rol mai activ în solicitarea datelor de la utilizator, dar tot controllerul gestionează logica.
- **Manipularea inputului**: În MVP, presenterul este responsabil pentru preluarea și procesarea inputului, în timp ce în MVC, controllerul preia această responsabilitate, lucrând direct cu view-ul pentru a răspunde la acțiunile utilizatorului.
- **Decuplarea componentelor**: MVP oferă o decuplare și mai mare între view și logica de business prin utilizarea unui mediator (presenter), ceea ce poate facilita testarea și mentenanța. MVC permite un flux mai direct și poate fi mai intuitiv în aplicații cu interacțiuni complexe între view și model.

MVVM (Model-View-ViewModel)

Model

- Datele aplicatiei

View

- Interfata utilizator

ViewModel

- Legatura intre date si interfata
- Uzual, se utilizeaza legarea dinamica a datelor (data binding)
- Genereaza evenimente la modificari ale datelor

MVT (Model View Template)

Model

- Incapsuleaza datele si accesul la date
- Notifica View-ul cu privire la modificarile aparute

View

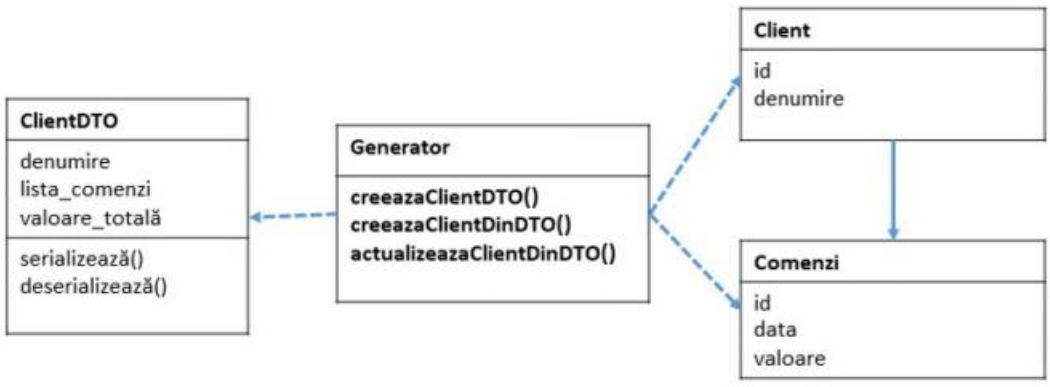
- Logica aplicatei
- Interactioneaza cu modelul

Template:

- Nivelul de prezentare
- Gestioneaza interactiunea cu interfata aplicatiei

Modele pentru aplicatii de interprindere

DTO



Active Record -> asociat cu o inregistrare dintr-o tabela

- ➔ Accesul la baza de date + logica domeniului pt datele reprezentate
- ➔ Incalca SRP

Data Mapper -> separa obiectele in memorie de baza de date

- ➔ Izoleaza cele 2 niveluri (aplicatie si persistenta)

Exemple proiect:

Filter studenti dupa nume:

```
students.stream()
        .filter(student -> student.getName ().equalsIgnoreCase(nume))
        .collect(Collectors.toList());
```

Filter burse dupa medie:

```
students.stream()
        .filter(student -> student.getPret() > pretMinim)
        .collect(Collectors.toList());
```

Citire Burse din fisier txt

```
public static List<Bursa> loadBurse(){
    List<Bursa> bursa = new ArrayList<>();

    try(BufferedReader br = new BufferedReader(new FileReader(FISIER))){
        br.lines().forEach(linie -> {
            String parts[] = linie.split(",");

            String numeStudent = parts[0].trim();
            double total = Double.parseDouble(parts[1].trim());

            BursaState state = BursaStateFactory.getState(parts[2].trim());
            burse.add(new Bursa(numeStudent, total, state));
        });
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return burse;
}
```

Scriere Burse in fisier txt

```
public static void saveAll(List<Bursa> burse){
    try(BufferedWriter bw = new BufferedWriter(new FileWriter(FISIER))){
        for(Bursa bursa : burse){
            bw.write(bursa.toString());
            bw.newLine();
        }
```

```
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```