6. Tervezési minták alapjai

Célja

- Nagyobb rendszereknél alkalmazzuk.
    - Komplex alkalmazások
    - Átláthatóság
    - Későbbi kiegészítés lehetősége
    - Platformfüggetlen
- **Megoldás**
    - Tervezési minták (design patterns)
- El kell dönteni, hogy mi a fontosabb
    - Kódhossz, megírási idő, futásidő **VAGY** olvashatóság, újrafelhasználhatóság és karbantarthatóság

SOLID elvek

Single Responsibility

- Minden osztály egy dologért legyen felelős és azt jól lássa el.
- **Ha nem követjük, akkor:**
    - Spagetti kód, átláthatatlanság
    - Nagy méretű objektumok
    - Mindenért felelős alkalmazások és szolgáltatások

Open/Closed elv

- Egy osztály legyen nyitott a bővítésre és a zárt módosításra, vagyis nem írhatunk bele, de származtathatunk tőle.
- **Ha nem követjük, akkor:**
    - Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetőek.
    - Leszármazott megírásakor módosítani kell az ősosztályt, ami tilos.
    - Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában.

Liskov substitutable

- Ősosztály helyett utódpéldány legyen mindig használható.
- Compiler supported, hiszen OOP elv (polimorfizmus)
- Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.

Interface seggregation

- Sok kis interfészt használjunk egy hatalmas mindent előíró interfész helyett.
- **Ha nem követjük, akkor:**
    - Egy osztályt létrehozunk valamilyen célból, megvalósítjuk az interfészt és rengeteg üres, fölösleges metódusunk lesz.
    - Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással.

Dependency Inversion

- A függőségeket ne az őket felhasználó osztály hozza létre.
- Várjuk kívülről a példányokat interfészeken keresztül.
- **Példány megadására több módszer is lehetséges:**
    - Dependency Injection
    - Inversion of Control (IoC) container
    - Factory tervezési minta
- **Ha nem követjük, akkor**
    - Egymástól szorosan függő osztályok végtelen láncolata.
    - Nem lehet modularizálni és rétegezni.
    - Kód újrahasznosítás lehetetlen

Egyéb elvek

- DRY, Don't Repeat Yourself
- DDD = Domain Driven Design

Architekturális tervezési minták

Klasszikus háromrétegű architektúrák

- Adat, logika és megjelenítési rétegből áll.
- A háromrétegű architektúrában a felső rétegek mindig a közbenső rétegeken keresztül érik el az alsó rétegeket.

Megjelenítési réteg

- Az alkalmazás legfelső rétege.
- Fő funkciója a rétegnek, hogy lefordítsa a feladatokat az alsóbb rétegek felé és hogy a visszajövő adatokat megjelenítse a felhasználó felé.

Logikai réteg

- Alkalmazás viselkedése található meg ebben a rétegben.
- Feldolgozza a felsőbb rétegtől jövő feladatokat és döntéseket hoz az üzleti logika alapján.
- Feladata még az adatmozgatás a két réteg között.

Adat réteg

- Tárolja az információkat.
- Feladata az adatbázissal és a fájl rendszerrel való kommunikáció megvalósítása és a megfelelő adatok kigyűjtése.
- Az információt amit kinyer, azt vissza is küldi a logikai rétegnek.

Felhasználói felület alkalmazása a Model-View-ViewModel tervezési mintát használva.

Model-View-ViewModel – MVVM

- A **View** tud a **ViewModel**-ről.
- A **ViewModel** tud a **Model**-ről.
- A **Model** nem ismeri a **ViewModel**-t és a **ViewModel** nem tud a **View-ról**.

ViewModel

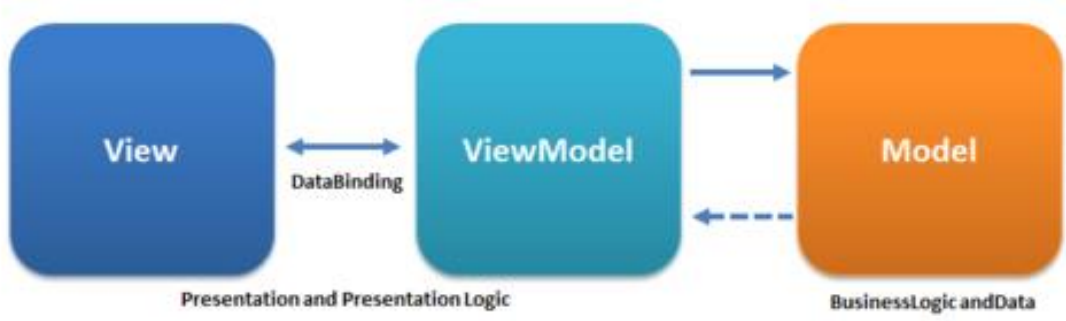
- Olyan tulajdonságokat és parancsokat implementál, amikhez a **View** adatkötést (Binding) végezhet.
- A változásértesítési eseményeken keresztül értesíti az állapotváltozások nézetét.

MVVM előnyök és hátrányok

- **Előnyök**
  - Tesztelés lehetősége
  - Bővíthetőség
  - Karbantarthatóság
- **Hátrányok**
  - A sok binding miatt bonyolult a debuggolás.
  - Egyes esetekben nehéz megtervezni egy komplexebb ViewModel-t.

Felhasználói felület alkalmazása

1. Létrehozunk egy **Model**-t, ami tartalmazza az alkalmazás logikáját és az adatmodellt.
2. Létrehozunk egy **View**-t, ami megjeleníti az adatokat, amiket a **Model** tartalmaz.
   a. Kezelheti a felhasználói interakciókat is.
3. Létrehozunk egy **ViewModel**-t, ami kapcsolódik a **Model**-hez és a **View**-hoz.
4. Összekötjük a **View**-t és a **ViewModel**-t adatkötés segítségével.
   a. Az adatkötés teszi lehetővé a kommunikációt a két réteg között.



Felhasználói felület alkalmazása a Model-View-Controller tervezési mintát használva.

Model-View-Controller – MVC

- A **View** réteg frissíti a **Controller** réteget.
- A **Controller** réteg frissíti a **Model**-t.
- A **Model** réteg közvetlenül visszahat a **View** rétegre.
- Ismertebb MVC web keretrendszerek
    - Ruby on Rails, Django, ASP.NET, Symphony

Controller

- A **Controller** felelős azért, hogy értelmezze a felhasználói interakciókat és módosítsa a **Model**-t ennek megfelelően.

MVC előnyök és hátrányok

- **Előnyök**
    - Közös munkát lehetővé teszi a többi fejlesztő számára.
    - Hibakeresés viszonylag könnyebb, mert több szint van.
    - Tesztelhető minden komponens külön-külön.
- **Hátrányok**
    - Nehezen újrahasználhatóak a modellek.
    - Fejlesztés során több technológia ismeretére lehet szükség.

Felhasználói felület alkalmazása

- Felhasználói felület elkészítésekor a felhasználói interakciókat és azok hatását a **Model**-re és a **Controller**-re kell szétválasztani.

MVC folyamata

1. Felhasználó csinál valamit a UI-on (**View**).
2. A **View** tájékoztatja a **Controller**-t a végrehajtott műveletről.
3. A **Controller** frissíti a **Model**-t.
4. A **Model** új adatokat szolgáltat.
5. **Controller** frissíti a **View**-t.