Tervezési minták egy OO programozási nyelvben

Bevezetés

Az objektumorientált programozás egyik alapgondolata, hogy a programot kisebb, jól elkülöníthető részekből építjük fel, amelyek együtt adják ki a teljes működést. Ahogy egy projekt bonyolultabb lesz, egyre több probléma ismétlődik, és kiderül, hogy sok feladat megoldására már léteznek bevált módszerek. Ezeket hívjuk tervezési mintáknak. A tervezési minták nem konkrét kódot jelentenek, hanem olyan általános megoldásokat, amelyeket többféle alkalmazásban és nyelvben is újra fel lehet használni.

A beadandó célja bemutatni néhány fontos tervezési mintát, amelyekre igyekszem példákat is hozni: több valós projektből, emiatt nyilván korlátozottan.

Az elmúlt időszakban rengetegszer használtam a PHP alapú Laravel keretrendszert munkáim során, amely a legtöbb rétegében tervezési mintákból áll. Persze ez alapvetően már előre megépített mintákat foglal magába, de a tervezési minták bemutatásához, mint azt a jelentése is mutatja, nem kód, hanem általános megoldás tervezése szükséges. Ettől függetlenül természetesen saját magunk is írhatunk mintába illő metódusokat is Laravelen belül.

A tervezési minták fogalma és szerepe

A tervezési minta lényegében egy sablon. Nem konkrét kód, hanem egy olyan gondolkodási keret, amely segít bizonyos típusú problémák megoldásában.

Előnyei:

- nem kell újra „kitalálni” bizonyos funkciókat minden projektnél
- tisztább, könnyebben karbantartható kódot eredményez
- más fejlesztők is könnyebben megértik

A tervezési mintákat általában három kategóriába sorolják:

- **Kreációs minták** (például Singleton, Factory),
- **Strukturális minták** (Adapter, Facade),
- **Viselkedési minták** (Observer, Strategy).

Ebből a Laravel a legtöbbet alkalmazza, amelyek bemutatásra fognak kerülni.

Az MVC tervezési minta

A legelterjedtebb minták közé tartozik az MVC, vagyis a Model–View–Controller. Ezt rengeteg keretrendszer használja, különösen a webfejlesztésben. Az MVC célja az, hogy különválassza az adatot, a megjelenítést és az irányítást.

Model

A Model felelős az adatok kezeléséért és az üzleti logikáért. Webes környezetben ide tartoznak az adatbázis műveletek is. Laravelben ez általában egy Eloquent model osztályt jelent, például User, Post stb.

A hozott példát megnézve láthatjuk, hogy a **Scan osztály a Laravel Eloquent Model ősosztály gyermeke.** Megörökli a Model tulajdonságait, ami például lehet query builder, vagy a képen láthato **fillable kezelés, kapcsolatok.**

A $fillable öröklött tulajdonság felel a tömeges feltöltésért, a $dates dátumot konvertál, a legalsó metódus pedig a kapcsolatot írja, az idegen kulcsot, persze ezt a Location modelben is jellemezni kell.

```php
class Scan extends Model
{
    use HasFactory;

    protected $fillable = [
        'location_id',
        'plate',
        'driver_name',
        'from',
        'to',
        'status',
    ];

    protected $dates = [
        'from',
        'to',
    ];

    public function location()
    {
        return $this->belongsTo(Location::class);
    }
```

Controller

Az MVC szerint most a View következne, de véleményem szerint az MVC magyarázata
M-C-V sorrendben sokkal érthetőbb.

A Controller köti össze a Modelt és a View-t. Kezeli a kéréseket, hívja a megfelelő model metódusokat, és előkészíti az adatot a nézet számára.

Mondhatjuk úgy, hogy ez képviseli a **backend logikát.** Kinek, mikor, mit és hogyan, minden kérdésre itt adunk választ, ezzel korlátozhatjuk a view adathoz való hozzáférését, annak kezelését. Akár mi adunk adatot, vagy a view-ból tér vissza.Még érthetőbb lenne ha az előző modellel kapcsolatos controllert hoznám példának, de az nyilvánvaló okokból nem lenne helyes. Ehelyett bemutatok egy cache kezelést.

A controller ősosztály gyermeke a LocationController. Itt egy részlete látható, egy metódus, ami a manuális kiléptetésért felel. A view-ban ez gombként jelenik meg, rányomva az adott scanId alapján a rekord kiléptetésre kerül. A $scan változóba lekérjük az id-hez tartozó rekordot, ahol pár ellenőrzést végzünk, és „left” állapotba tesszük. Gyakorlatilag ez egy view oldalról jövő adatbázis változtatás.

```php
public function manualExit($scanId)
{
    $scan = Scan::findOrFail($scanId);

    if ($scan->to === null) {
        $scan->to = now();
        $scan->status = 'left';
        $scan->save();
```

View

A View a megjelenítésért felel. Itt nincs, vagy kevés a logika, csak a felhasználó felé látható HTML, esetleg dinamikus változókkal. Laravelben erre a Blade sablonok szolgálnak.

```blade
@foreach($scans as $scan)
    <tr>
        <td>{{ $scan->id }}</td>
        <td>{{ $scan->plate }}</td>
        <td>{{ optional($scan->location)->name ?? '-' }}</td>
        <td>{{ $scan->from ? $scan->from->format('Y-m-d H:i') : '-' }}</td>
        <td>{{ $scan->to ? $scan->to->format('Y-m-d H:i') : '-' }}</td>
        <td>
```

Olyat hoztam, amin a blade lényege is látszik, a HTML-t már mindenki ismeri. Itt a tr/td felel egy táblázatszerű elrendezésért. Amit látunk egy **dashboard részlet.** Itt egy foreach-el végig megyünk a controllerben a view-nak átadott scaneken, és ezeket kiiratjuk. Az eloquent model része a $model->oszlopnév. Ez a usernek csak a DB cella tartalmát fogja mutatni, mint vizuálisan, mint konzol szerint is.

Miért jó az MVC?

- Átláthatóbb lesz a projekt.
- Külön emberek is tudnak dolgozni a model, a view vagy a controller rétegen.
- A kód könnyebben bővíthető.

Ez aránylag jól szemlélteti az MVC működését. A model egy példányt azonosít, ami jelenleg az adatbázishoz köthető. Leírja a feltöltésének módját, a kapcsolatait, adatot alakít át. A controller ami a tényleges munkát végzi. Adatbázisműveleteket, mit adjon át a viewnak, műveleteket végez el. A view pedig a megjelenítést végzi. Ami nekünk egy kattintás, mint a példa szerint egy valami kiléptetése, az ezen a folyamaton fut végig a háttérben.

Persze ne feledjük, hogy az első lépés nem lett megemlítve, de ez nem is kapcsolódik rögtön az MVC modelhez: A routing, ami a böngészőből érkező kérést feldolgozza.

Tervezési minták

Singleton (Egyke)

A Singleton mintázat biztosítja, hogy egy osztálynak csak egyetlen példánya legyen az egész alkalmazásban, és mindenki ugyanahhoz az objektumhoz férjen hozzá. Például egy konfigurációs osztály esetén hasznos, ahol nem szeretnénk, hogy többször példányosítsák a beállításokat.

Factory Method (Gyártófüggvény)

A Factory Method mintázat segít az objektumok létrehozásában anélkül, hogy közvetlenül az osztályokat példányosítanánk. Ez rugalmasabbá teszi a kódot, mert könnyen lecserélhetünk egy típust a gyártófüggvény módosításával, anélkül hogy a fő logikát átírnánk.

Bridge (Híd)

A Bridge mintázat célja, hogy elválassza az absztrakciót a megvalósítástól, így mindkettőt külön tudjuk fejleszteni és bővíteni. Például egy rajzoló programban lehet külön a forma (kör, négyzet) és a megjelenítési technika (vonal, kitöltés), és a kettőt könnyen kombinálhatjuk.

Composite (Összetétel)

A Composite minta lehetővé teszi, hogy összetett objektumokat és az egyszerű objektumokat egységes módon kezeljük. Például egy fájlrendszerben a fájlokat és a mappákat ugyanúgy tudjuk kezelni, mert mindkettő “csomópont” az összetett struktúrában.

Template Method (Sablonfüggvény)

A Template Method egy algoritmus vázát adja meg, ahol bizonyos lépések konkrét megvalósítása a leszármazott osztályokra van bízva. Ez segít, hogy a közös logikát ne ismételjük, de a részleteket rugalmasan lehessen változtatni.

Strategy (Stratégia)

A Strategy mintázat lehetővé teszi, hogy egy algoritmus változatos megvalósításait cserélgetni tudjuk futásidőben. Például egy játékban különböző AI-stratégiákat alkalmazhatunk az ellenfelekre anélkül, hogy az alap logikát módosítanánk.

State (Állapot)

A State minta segít egy objektum viselkedését a belső állapotától függően változtatni. Például egy videólejátszó másképp reagál, ha “lejátszás” vagy “szünet” állapotban van, anélkül, hogy hatalmas if-else ágakat kellene írni.

Visitor (Látogató)

A Visitor mintázat lehetővé teszi, hogy egy objektum struktúrán új műveleteket hajtsunk végre anélkül, hogy magát az objektumot módosítanánk. Például egy fájlstruktúrában különböző jelentéseket készíthetünk a fájlokról és mappákról, anélkül, hogy a fájl vagy mappa osztályt változtatnánk.

Iterator (Bejáró)

Az Iterator mintázat lehetővé teszi, hogy egy összetett adatszerkezet elemein egységes módon végiglépkedjünk, anélkül, hogy tudnunk kellene az adatszerkezet belső felépítését. Például egy lista, halmaz vagy fa esetén ugyanazzal a bejáróval tudunk dolgozni.

Dependency Injection (Függőség befecskendezés)

A Dependency Injection mintázat lényege, hogy egy osztály ne saját maga hozza létre a függőségeit, hanem kívülről kapja meg azokat. Ez növeli a kód tesztelhetőségét és karbantarthatóságát, mert könnyen lecserélhetők az objektumok és csökkenti a szoros összekapcsoltságot.