Tervezési minták az Amőba játékban

1. TEMPLATE METHOD – Sablon metódus örökléssel

Probléma: Emberi és számítógépes játékosoknak vannak közös tulajdonságai, de különböző viselkedéseik. Külön implementálás duplikációt okozna.

Megoldás: Absztrakt ősosztály definiálja a sablont, alosztályok implementálják a specifikus részeket. Közös kód az ősben marad.

Projekt példa:

- Osztályok: `Player` (absztrakt), `HumanPlayer`, `ComputerPlayer`

- Hol látszik: `Player` tartalmazza a név és szimbólum kezelést, míg a lépésgenerálás az alosztályok feladata

- Előny: Új játékostípusok könnyen hozzáadhatók, kódtömörítés

2. MEMENTO – Állapotmentés és visszaállítás

Probléma: Játékmenet mentése és folytatása igényel komplex állapotkezelést. Belső struktúra nem szabad, hogy közvetlenül látható legyen.

Megoldás: Objektum állapotának kinyerése külső tárolóba, későbbi visszaállítással. Implementáció el van különítve a játéklogikától.

Projekt példa:

- Osztályok: `Game`, `Board`

- Hol látszik: `saveGameToFile()` és `loadGameFromFile()` metódusok kezelik a tábla állapotát fájlban

- Előny: Játék folytatható, mentési mechanizmus független, fájlformátum cserélhető

3. STRATEGY – Változtatható algoritmusok

Probléma: Számítógépes játékos jelenleg csak véletlenszerűen lép. Jövőbeli nehézségi szintek támogatásához rugalmasabb architektúra szükséges.

Megoldás: Algoritmusok külön osztályokba zárása, közös interfészen keresztüli használat. Dinamikus stratégiaválasztás.

Projekt példa (javaslat):

- Osztályok: `MoveStrategy` interfész, `RandomStrategy`, `SmartStrategy`
- Hol látszik: `ComputerPlayer` delegálhatna a stratégia objektumnak a lépésgenerálást
- Előny: AI nehézség változtatható, új stratégiák egyszerűen hozzáadhatók, könnyű tesztelés

4. VALUE OBJECT – Értékobjektumok

Probléma: Pozíciók kezelése primitív típusokkal nehezíti a kód olvashatóságát és biztonságát. Két int külön kezelése hibák forrása.

Megoldás: Kis, változtathatatlan osztály létrehozása, amely egyetlen logikai értéket reprezentál. Tartalom alapú összehasonlítás.

Projekt példa:

- Osztály: `Position`
- Hol látszik: `Position` osztály final mezőkkel, felülírt `equals()` és `hashCode()` metódusokkal
- Előny: Típusbiztonság, kollekciókban való használat, jobb kódstruktúra, logikai csoportosítás

5. SINGLETON – Egységes erőforráskezelés

Probléma: High score kezeléshez minden játékpéldány új adatkezelőt hoz létre. Ez erőforrás-pocsékolás és inkonzisztenciához vezethet.

Megoldás: Osztályból csak egy példány létezhet, globális hozzáférési pont biztosítása. Lazy initializationnel.

Projekt példa (javaslat):

- Osztály: `DatabaseManager` Singleton változatban

- Hol látszik: Privát konstruktor, statikus `getInstance()` metódus, egyszeri példányosítás

- Előny: Egységes adatállapot, erőforrás-hatékonyság, könnyű mockolás, garantált konzisztencia

6. FACTORY METHOD – Objektumlétrehozás centralizálása

Probléma: Játékosok létrehozása szétszórtan történik, nehéz konfigurálni és bővíteni. Konstruktor hívások közvetlenül a kódban.

Megoldás: Létrehozási logika külön osztályba szervezése. Egységes interfész különböző típusú objektumok generálásához.

Projekt példa (javaslat):

- Osztály: `PlayerFactory`

- Hol látszik: `PlayerFactory.createHuman()` és `PlayerFactory.createComputer()` metódusok

- Előny: Létrehozási logika centralizálva, konfiguráció egyszerűbb, könnyű új típusok hozzáadása

7. OBSERVER – Eseményvezérelt architektúra

Probléma: Játék események (győzelem, lépés) kezelése közvetlen hívásokkal történik. Új funkciók (naplózás, statisztika) hozzáadása nehézkes.

Megoldás: Közzétevő-feliratkozó modell. Eseményeket közvetítő objektum továbbítja a regisztrált figyelőknek.

Projekt példa (javaslat):

- Osztályok: `GameEventPublisher`, `GameObserver` interfész

- Hol látszik: Győzelem esetén a `Game` értesítené a regisztrált observer-eket

- Előny: Laza csatolás, könnyű új figyelők hozzáadása, több kimeneti csatorna