Programozási Technológiák Beadandó

Farkas Balázs Alex HHNIHW

June 2020

1. Bevezető

A projekt egy nagyvonaló webshopot reprezentál, ahol egy fő raktár kommunikál két kisebb raktárral (élelmiszer és játék), és a felhasználók képesek termékeket rendelni úgy, hogy csak a fő raktárról van tudomásuk. A fő raktárba szállítanak be termékeket, ami jelez az alraktáraknak, hogy termék érkezett, és automatikusan szét szortírozza azokat. Két vásárlói kört különböztetek meg: normális vásárló és prémium vásárló. A kettő annyiban tér el egymástól, hogy a prémium vásárlónak nem kell általános forgalmi adót fizetnie. A vásárlás csak akkor történhet meg, ha az aktuális pillanatban a vásárlónak elegendő pénze van az "online kártyáján" a termékek teljeskörű kifizetésére.

A projektben a fő cél a tervezési minták logikus és indokolt használata volt, így csak annyi van elkészítve a projektből, ami feltétlenül szükséges a minták implementálására és bemutatására. A webshopnak nincsen felhasználói felülete, A tesztek bizonyítják a metódusok működését.

2. Tervezési minták

A tervezési minták gyakori programozói feladatokat oldanak meg, jól kiforrott válaszok a tipikus programozói feladatokra. A minták alkalmazásával könnyen bővíthető, módosítható osztályszerkezeteket kapunk, tehát rugalmas kódot. Az ár amit ezért fizetünk, a bonyolultabb, nehezebben átlátható kód és a nagyobb fejlesztési idő. A webshopban sincs ez másképp, a projektet sokkal egyszerűbben is el lehetett volna készíteni. Ugyanakkor elképzehető, hogy a rendszertelensége miatt a jövőben több időt fordítanánk a hibakeresésre, mint amennyi idő alatt mintákkal lefejlesztettük. Mivel minták alkalmazásával a kódunk rugalmasabb, így könynen bővíthető lesz. A továbbiakban a beadandóban használt mintákat említem meg.

2.1. Tesztvezérelt fejlesztés

A tesztvezérelt fejlesztés egy szoftverfejlesztési folyamat, amiben a követelményeket speciális teszt esetekként fogalmazzuk meg, a kódot pedig ehhez mérten írjuk meg, így az át fog menni a teszten. Ennek köszönhetően biztosak

lehetünk benne, hogy a programunk jól működik minden változtatás után. A projektet végig ebben a folyamatban készítettem, így megkönnyítettem a saját dolgom. Az úgynevezett piros-zöld-kék-piros mintát használtam, tehát először megírtam a tesztet, utána az üzemi kódot, utána pedig refaktoráltam.

```
MainWarehouse mw = MainWarehouse.getInstance();

@Test
public void TestMainWarehouseIsSingleton() {
    MainWarehouse mw2 = MainWarehouse.getInstance();
    Object expected = mw;
    Object actual = mw2;
    Assert.assertEquals(expected, actual);
}
```

2.2. Singleton

A singleton olyan programtervezési minta, amely egy objektumra korlátozza egy osztály létrehozható példányainak számát. Gyakori, hogy egy osztályt úgy kell megírni, hogy csak egy példány lehet belőle. A projektben a fő raktár (MainWarehouse) singleton-ként lett létrehozva, mivel csak egyetlen egyre lesz szükségünk. Ennek köszönhetően csak egyszer lehet példányosítani, a többi hivatkozás az egyetlen referenciát fogja visszaadni.

```
public final class MainWarehouse implements IObservable{

    static MainWarehouse mw = null;

    public static MainWarehouse getInstance(){
        if (mw == null)
            mw = new MainWarehouse();
        return mw;
    }
```

2.3. Observer

Az Observer minta egy olyan szoftvertervezési minta, melyben egy objektum listát vezet alárendeltjeiről, és automatikusan értesíti őket bármilyen állapotváltozásról, többnyire valamely metódusuk meghívásán keresztül. A beadandóban

a Fő raktár vezet listát az alraktárakról, és amint termék érkezik a fő raktárba, automatikusan értesíti a többi alraktárt, és szét szortírozza a termékeket.

2.4. Command

a mintában egy objektumot használunk arra, hogy reprezentáljunk és párosítsunk minden olyan információt, amely szükséges lehet egy metódus későbbi meghívásához. A projektben a rendelés metódusa van Command mintában lefejlesztve. A vásárló rendelése visszavonható, ami szintén a Command minta jellegzetessége. A feladó maga a vevő, a befogadó pedig a fő raktár. Én azonban ezt egy fokkal messzebbre vittem, és összekötöttem a Strategy mintával.

```
public class OrderCommand implements IOrderCommand{

    MainWarehouse mw;
    IOrderStrategy os;

    public OrderCommand(MainWarehouse mw, IOrderStrategy os) {
        this.mw = mw;
        this.os = os;
    }

    @Override
    public void order(Product p, int quantity, float balance)
       os.orderProduct(p,quantity,balance);
    }

    @Override
    public void undo() {
        os.undoOrder();
    }
}
```

2.5. Strategy

a stratégia minta lehetővé teszi, hogy egy algoritmus viselkedését a futás során válasszuk meg. Különböző viselkedéseket adhatunk meg hasonló objektumoknak. A projektben a stratégia mintával különítem el a két vásárlói szerepkört: a

normál vásárlót és a prémium vásárlót. Mindkét vásárló ugyanazt tudja, annyiban térnek el, hogy a prémium vásárlónak nem kell ÁFÁ-T fizetnie. Ezt a különböző viselkedést választja szét a stratégia minta, azonos objektumokon. Amint említettem, a Command mintával kötöttem össze a stratégiát, így a vásárló létrejöttekör a hozzá tartozó rendelési parancs úgy jön létre, hogy az figyelembe veszi a viselkedési stratégiát.

2.6. Iterator

Az objektumorientált programozásban az iterátor olyan tervezési minta, melyet arra használunk, hogy bejárjunk egy kollekciót és elérjük annak elemeit. Előnye, hogy nem csak végigiterálható adatszerkezeteken használható, hanem saját objektumon is. Szerencsére a legtöbb programozási nyelv keretrendszere támogatja az Iterator minta használatát, így előre elkészíti a terepet számunkra. Ez azt jelenti, hogy nem kell megírnunk a különböző interfészeket, mert már a keretrendszer magában tartalmazza. A projektben a terékek törlésénél használtam, ahol be kell járnom a Termék objektumokat tartalmazó listát. Olyan pontra jutottam, ahol elkerülhetetlenül használnom kellett az Iterator mintát, ugyanis egyéb esetben kivételt váltott ki a program.

```java
@Override
public void removeProduct(String name) {
    ArrayList<Product> list = mw.getToyWarehouse().toyList;

    for (Iterator<Product> iterator = list.iterator(); iterator.hasNext(); ) {
        Product p = iterator.next();
        if (p.getName().equals(name)) {
            iterator.remove();
            return;
        }
    }
}
```

2.7. Clean Code

A Clean Code a szoftverfejlesztés kifejezése. Előnyei a stabilabb és hatékonyan karbantartható programok. A projekt fejlesztése folyamán végig figyeltem az elveinek betartására, így egy összetett, jól olvasható forráskódot kaptam végeredményül.

2.8. Open-Closed Principle

Az open-closed principle a számítógép-programozásban a SOLID-alapelvek egyike. Azt mondja ki, hogy a program forráskódja legyen nyitott a bővítésre, de zárt a módosításra. A programomat úgy alakítottam, hogy megfeleljen ennek a kritériumnak. Így a jövőben a kódot nem kell módosítani, nem kell további

if/else-ágakat létrehozni. Ezt úgy értem el, hogy az új objektumok létrehozásakor, a konstruktorban kerülnek beállításra az efféle adatok.

2.9. GOF Alapelvek

A GOF, azaz a Gang Of Four alapelveit 4 híres programozó tette le, ami azóta is a programtervezés alapjául szolgál. Tételeik kimondják, hogy ha tudjuk, kerüljük az öröklődést, és helyette használjunk kompozíciót, azaz objektum összetételt. A programomban is így tettem, egyeltlán nem szerepel benne öröklődés, csak interface-k implementálása. Továbbá az osztályokat beszúrtam a főbb osztályba, ezáltal megfordítva a felelősséget (Függőség megfordításának elve). Tételeik azt is kimondják, hogy programozzunk felületre implementáció helyett. Ez tulajdonképpen az OCP-nek megfelel, így ezt is teljesíti a program.

2.10. OOP elvek

Az objektumorientált programozás főbb elvei közé tartozik az öröklődés, polimorfizmus, egységbezárás és az absztrakció. Azonban a GOF elvek kimondják, hogy kerüljük az öröklődést, ezért nem használtam a projektben. A főbb hangsúly a polimorfizmusra, azaz a többalakúságra, és az egységbezárásra került. Ezeket találtam a legjobban használható elveknek, így a program teljes egészében megtalálható a használatuk.