Tervezési minták egy OO programozási nyelvben beadandó

Iterator

Az iterator egy viselkedési tervezési minta, amely lehetővé teszi egy gyűjtemény elemeinek bejárását anélkül, hogy felfedné annak belső megvalósítását (lista, verem, fa stb.).

A gyűjtemények az egyik leggyakrabban használt adattípusok a programozásban. Ennek ellenére egy gyűjtemény csupán egy tároló egy csoport objektum számára.

A gyűjtemények többsége egyszerű listákban tárolja az elemeit. Azonban egyesek vermeken, fákon, gráfokon és más összetett adatstruktúrák alapján működnek.

De bárhogyan is van felépítve egy gyűjtemény, biztosítania kell valamilyen módot az elemei elérésére, hogy más kódok használhassák ezeket az elemeket. Olyan módszernek kell lennie, amely lehetővé teszi az összes elem végigjárását anélkül, hogy ugyanazokat az elemeket újra és újra elérnénk.

Ez könnyű feladatnak tűnhet, ha egy lista alapú gyűjteményről van szó. Egyszerűen végig lehet menni az összes elemen. De hogyan lehet végigjárni egy összetett adatstruktúra, például egy fa elemeit? Például lehet, hogy egyik nap elegendő a fa mélységi bejárása. Másnap viszont lehet, hogy szélességi bejárásra van szükség. És egy hét múlva talán valami mást szeretnénk, például véletlenszerű hozzáférést a fa elemeihez.

Az Iterator minta fő ötlete, hogy a gyűjtemény bejárási viselkedését egy különálló objektumba, úgynevezett iteratorba szervezzük ki.

Az iterator objektum nemcsak az algoritmus megvalósítását tartalmazza, hanem az összes bejárási részletet is, például az aktuális pozíciót és azt, hogy hány elem van még hátra a végéig. Ennek köszönhetően több iterator is végigmehet ugyanazon a gyűjteményen egyszerre, függetlenül egymástól.

Általában az iterátorok egy fő metódust biztosítanak az elemek lekérésére a gyűjteményből. A kliens ezt a metódust addig hívhatja, amíg az nem ad vissza semmit, ami azt jelenti, hogy az iterator már végigjárta az összes elemet.

Minden iterátornak ugyanazt az interfészt kell megvalósítania. Ez lehetővé teszi, hogy a kliens kód kompatibilis legyen bármilyen gyűjteménytípussal vagy bármilyen bejárási algoritmussal, amennyiben megfelelő iterator létezik. Ha speciális módot szeretnénk egy gyűjtemény bejárására, egyszerűen létrehozhatunk egy új iterator osztályt anélkül, hogy változtatni kellene a gyűjteményen vagy a kliensen.

Implementáció

1. Deklaráljuk az iterátor interfészt. Legalább egy metódust kell tartalmaznia, amely lehetővé teszi a következő elem lekérését a gyűjteményből. A kényelem érdekében hozzáadhatunk néhány másik metódust is, például a korábbi elem lekérését, az aktuális pozíció nyomon követését és az iteráció végének ellenőrzését.
2. Deklaráljuk a gyűjtemény interfészt, és írjuk le a metódust az iterátorok lekérésére. A visszatérési típusnak meg kell egyeznie az iterátor interfész típusával. Hasonló metódusokat deklarálhatunk, ha több különböző iterátor csoportot is szeretnénk használni.
3. Implementáljunk konkrét iterátor osztályokat azokra a gyűjteményekre, amelyeket iterátorokkal szeretnénk bejárni. Egy iterátor objektumnak egyetlen gyűjtemény példányhoz kell kapcsolódnia. Általában ezt a kapcsolatot az iterátor konstruktorán keresztül hozzuk létre.
4. Implementáljuk a gyűjtemény interfészt a gyűjtemény osztályaiban. A fő cél az, hogy a kliens számára egy egyszerűsített módot biztosítsunk az iterátorok létrehozására, amelyek a konkrét gyűjtemény osztályhoz vannak szabva. A gyűjtemény objektumnak át kell adnia saját magát az iterátor konstruktorának, hogy létrejöjjön a kapcsolat közöttük.
5. Ellenőrizzük a kliens kódot, hogy minden gyűjtemény bejárásához kapcsolódó kódot iterátorok használatával helyettesítsünk. A kliens minden alkalommal új iterátor objektumot kér, amikor a gyűjtemény elemein szeretne iterálni.

Pro

1. Egységes Felelősség Elve (Single Responsibility Principle): Tisztábbá tehetjük a kliens kódot és a gyűjteményeket azáltal, hogy a nagy, bonyolult bejárási algoritmusokat külön osztályokba helyezzük.
2. Nyitott/Zárt Elv (Open/Closed Principle): Új típusú gyűjteményeket és iterátorokat valósíthatunk meg, és átadhatjuk őket a meglévő kódnak anélkül, hogy bármit is megsértenénk.

3. Párhuzamos Iterálás: Ugyanazt a gyűjteményt párhuzamosan is bejárhatjuk, mert minden iterátor objektum saját iterációs állapotot tartalmaz.

4. Késleltetett Iterálás: Ugyanezért az iterálást késleltethetjük, és szükség esetén folytathatjuk, amikor csak szükséges.

Contra

1. Túlzott Használat (Overkill): Ha az alkalmazásunk csak egyszerű gyűjteményekkel dolgozik, az iterátor minta alkalmazása túlzás lehet.

2. Teljesítmény Probléma: Az iterátor használata kevésbé lehet hatékony, mint közvetlenül végigmenni egyes speciális gyűjtemények elemein.

Command

A parancs (Command) egy viselkedési tervezési minta, amely egy kérést önálló objektummá alakít, amely tartalmazza az összes információt a kéréssel kapcsolatban. Ez az átalakítás lehetővé teszi, hogy a kéréseket metódus paramétereként továbbítsuk, késleltessük vagy sorba állítsuk a kérés végrehajtását, valamint támogassuk a visszavonható műveleteket.

Bár mindezek a gombok hasonlónak tűnnek, mindegyiknek más-más feladatot kell végrehajtania. Hol helyeznénk el a különböző kattintási kezelők kódját a gombokhoz? A legegyszerűbb megoldás az lenne, ha rengeteg alosztályt hoznánk létre minden egyes helyhez, ahol a gombot használják. Ezek az alosztályok tartalmaznák a kódot, amelyet egy gomb kattintásakor végre kell hajtani.

Rövid időn belül rájöhetünk, hogy ez a megközelítés súlyosan hibás. Először is, hatalmas számú alosztályunk lesz, és ez rendben is lenne, ha nem kellene minden alkalommal módosítanunk az alap Button osztályt, miközben nem kockáztatnánk, hogy a kódunk a különböző alosztályokban megsérüljön. Egyszerűen fogalmazva, a GUI kód kényelmetlenül függ a változékony üzleti logikától.

És itt van a legrosszabb rész. Néhány műveletet, mint például a szöveg másolása/beillesztése, több helyről is el kellene indítani. Például a felhasználó rákattinthat egy kis "Másolás" gombra az eszköztáron, vagy másolhat valamit a jobb gombos menü segítségével, vagy egyszerűen lenyomhatja a Ctrl+C billentyűkombinációt a billentyűzeten.

Kezdetben, amikor az alkalmazásunknak csak az eszköztár volt, még rendben volt, hogy a különböző műveletek megvalósítását a gombok alosztályaiba helyeztük. Más szóval, a szöveg másolásának kódja a CopyButton alosztályban nem volt probléma. De, amikor implementálni kell a kontextus menüket, gyorsbillentyűket és egyéb funkciókat, akkor vagy meg kell duplázni a művelet kódját sok osztályban, vagy a menüket kell a gombokhoz kötni, ami még rosszabb megoldás.

A jó szoftvertervezés gyakran a felelősségek szétválasztásának elvén alapul, ami rendszerint az alkalmazás rétegekre bontását eredményezi. A leggyakoribb példa: egy réteg a grafikus felhasználói felület (GUI), és egy másik réteg az üzleti logika számára. A GUI réteg felelős a gyönyörű kép megjelenítéséért a képernyőn, az inputok rögzítéséért, és az eredmények megjelenítéséért, amelyek azt mutatják, mit csinál a felhasználó és az alkalmazás. Azonban, amikor valami fontosat kell végrehajtani, például a Hold pályájának kiszámítását vagy egy éves jelentés összeállítását, a GUI réteg delegálja a munkát az alatta lévő üzleti logikai rétegre.

A kódban ez így nézhet ki: egy GUI objektum egy üzleti logikai objektum metódusát hívja meg, és átad neki néhány paramétert. Ezt a folyamatot általában úgy írják le, mint amikor egy objektum kérést küld egy másiknak.

A Parancs (Command) minta azt javasolja, hogy a GUI objektumok ne küldjék el közvetlenül ezeket a kéréseket. Ehelyett ki kell vonni az összes kérelem részletét, például a meghívott objektumot, a metódus nevét és az argumentumok listáját egy külön parancs osztályba, amely egyetlen metódussal rendelkezik, ami kiváltja ezt a kérést.

A parancs objektumok linkként szolgálnak a különböző GUI és üzleti logikai objektumok között. Mostantól a GUI objektumnak nem kell tudnia, hogy melyik üzleti logikai objektum fogja fogadni a kérést és hogyan lesz az feldolgozva. A GUI objektum egyszerűen kiváltja a parancsot, amely kezeli az összes részletet.

Implementáció

1. Deklaráljuk a parancs interfészt egyetlen végrehajtó metódussal.
2. Kezdjünk el kéréseket kivonni konkrét parancs osztályokba, amelyek megvalósítják a parancs interfészt. Minden osztálynak rendelkeznie kell mezőkkel, amelyek tárolják a kérés paramétereit, valamint egy hivatkozással a tényleges fogadó objektumra. Mindezeket az értékeket a parancs konstruktorán keresztül kell inicializálni.
3. Azonosítsuk azokat az osztályokat, amelyek küldőkként fognak működni. Adj hozzá mezőket a parancsok tárolására ezekhez az osztályokhoz. A küldőknek csak a parancs interfészen keresztül kell kommunikálniuk a parancsaikkal. A küldők általában nem hozzák létre maguk a parancs objektumokat, hanem a kliens kódból kapják meg őket.
4. Módosítsuk a küldőket úgy, hogy a parancsot hajtsák végre, ahelyett, hogy közvetlenül a fogadónak küldenének kérést.
5. A kliensnek a következő sorrendben kell inicializálnia az objektumokat:
    - Hozzuk létre a fogadókat.
    - Hozzuk létre a parancsokat, és ha szükséges, társítsuk őket a fogadókkal.
    - Hozzuk létre a küldőket, és társítsuk őket az egyes parancsokhoz.

Pro

1. Egyszeri felelősség elve: Elkülöníthetjük azokat az osztályokat, amelyek műveleteket hívnak meg azoktól az osztályoktól, amelyek ezeket a műveleteket végrehajtják.

2. Nyitott/Zárt elv: Új parancsokat adhatunk hozzá az alkalmazáshoz anélkül, hogy megsértenénk a meglévő kliens kódot.

3. Lehetőség van az undo/redo (visszavonás/ismétlés) megvalósítására.

4. Megvalósítható a műveletek késleltetett végrehajtása.

5. Egyszerű parancsokból összeállíthatunk egy összetett parancsot.

Contra

1. A kód bonyolultabbá válhat, mivel egy teljesen új réteget vezetünk be a küldők és a fogadók közé.

Proxy

A Proxy egy szerkezeti tervezési minta, amely lehetővé teszi, hogy egy másik objektum helyettesítésére vagy helykitöltőjeként szolgáló elemet biztosítsunk. A proxy szabályozza az eredeti objektumhoz való hozzáférést, így lehetőséget ad arra, hogy valamilyen műveletet végezzünk el a kérés eredeti objektumhoz történő eljutása előtt vagy után.

A Proxy minta azt javasolja, hogy hozzunk létre egy új proxy osztályt, amely ugyanazzal az interfésszel rendelkezik, mint az eredeti szolgáltatási objektum. Ezután frissítsük az alkalmazást úgy, hogy az eredeti objektum minden kliensének a proxy objektumot adja át. Amikor a proxy egy kérést kap egy klienstől, létrehozza a valódi szolgáltatási objektumot, és minden munkát tovább delegál annak.

Mi az előnye? Ha valamit végre kell hajtanunk az osztály elsődleges logikája előtt vagy után, a proxy lehetővé teszi ezt anélkül, hogy módosítanunk kellene az adott osztályt. Mivel a proxy ugyanazt az interfészt valósítja meg, mint az eredeti osztály, bármely kliens számára átadható, amely valódi szolgáltatási objektumot vár.

Implementáció

Ha nincs előre létrehozott szolgáltatási interfész, hozzunk létre egyet, hogy a proxy és a szolgáltatási objektumok felcserélhetők legyenek. Az interfész kinyerése a szolgáltatási osztályból nem mindig lehetséges, mert ez megkövetelné az összes kliens módosítását, hogy az interfészt használják. Alternatív megoldásként a proxy lehet a szolgáltatási osztály egy alosztálya, így örökli a szolgáltatás interfészét.

1. **A proxy osztály létrehozása**
   - Hozzunk létre egy proxy osztályt, amely tartalmaz egy mezőt a szolgáltatási objektum hivatkozásának tárolására. Általában a proxyk hozzák létre és kezelik a

szolgáltatási objektumaik teljes életciklusát. Ritka esetekben a kliens adhatja át a szolgáltatási objektumot a proxy konstruktorán keresztül.

2. **A proxy metódusok implementálása**
   - Implementáljuk a proxy metódusait a céljaiknak megfelelően. A legtöbb esetben a proxy valamilyen művelet elvégzése után a munkát továbbadja a szolgáltatási objektumnak.
3. **Egy létrehozási metódus bevezetése**
   - Fontoljunk meg egy olyan létrehozási metódus bevezetését, amely eldönti, hogy a kliens proxy objektumot vagy valódi szolgáltatási objektumot kapjon. Ez lehet egy egyszerű statikus metódus a proxy osztályban, vagy akár egy teljes értékű gyári (factory) metódus is.
4. **Késleltetett inicializálás megvalósítása**
   - Fontoljuk meg a késleltetett inicializálás megvalósítását a szolgáltatási objektum számára, hogy az csak akkor jöjjön létre, amikor valóban szükség van rá.

Pro

1. Ellenőrzés a szolgáltatási objektum felett: A proxy lehetővé teszi a szolgáltatási objektum irányítását anélkül, hogy a kliensek erről tudnának.

2. Életciklus-kezelés: A proxy kezeli a szolgáltatási objektum életciklusát, amikor a kliensek ezzel nem foglalkoznak.

3. Rugalmasság: A proxy akkor is működik, ha a szolgáltatási objektum még nincs készen vagy nem elérhető.
4. Nyitott/zárt elv (Open/Closed Principle): Új proxykat vezethetsz be anélkül, hogy módosítanod kellene a szolgáltatást vagy a klienseket.

Contra

1. Bonyolultság növekedése: A kód bonyolultabbá válhat, mivel sok új osztályt kell létrehozni.

2. Válasz késleltetése: A szolgáltatástól érkező válasz késhet a proxy működése miatt.

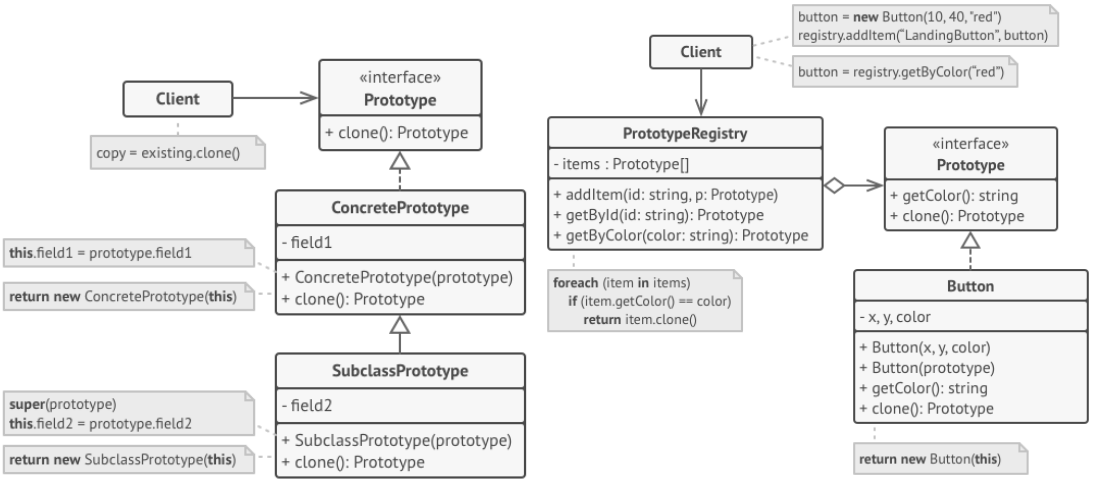
Prototype

A Prototype egy kreációs tervezési minta, amely lehetővé teszi meglévő objektumok másolását anélkül, hogy a kód az osztályaiktól függővé válna.

A Prototype minta a klónozási folyamatot magukra az objektumokra bízza, amelyek klónozva lesznek. A minta egy közös interfészt deklarál minden olyan objektum számára, amely támogatja a klónozást. Ez az interfész lehetővé teszi egy objektum klónozását anélkül, hogy a kódot az adott objektum osztályához kellene kötni. Általában egy ilyen interfész csupán egyetlen klónozó (clone) metódust tartalmaz.

A clone metódus megvalósítása nagyon hasonló az összes osztályban. A metódus létrehoz egy objektumot az aktuális osztályból, és átmásolja a régi objektum mezőinek értékeit az új objektumba. Még privát mezőket is másolhatunk, mivel a legtöbb programozási nyelv lehetővé teszi, hogy egy osztály objektumai hozzáférjenek más, ugyanahhoz az osztályhoz tartozó objektumok privát mezőihez.

Az objektumokat, amelyek támogatják a klónozást, prototípusoknak nevezzük. Amikor az objektumainknak tucatnyi mezőjük és több száz lehetséges konfigurációjuk van, azok klónozása az alosztályok létrehozásának alternatívája lehet.



Implementáció

1. Hozzuk létre a prototípus interfészt, és deklaráljuk benne a clone metódust. Vagy csak adjuk hozzá a metódust az összes osztályhoz, ha már van egy meglévő osztályhierarchiánk.
2. A prototípus osztálynak definiálnia kell egy alternatív konstruktort, amely az osztály egy objektumát fogadja paraméterül. A konstruktor feladata, hogy az osztályban definiált összes mező értékét átöltsese az átadott objektumból az újonnan létrehozott példányba. Ha alosztályt módosítunk, akkor a szülőosztály konstruktora is meghívásra kell kerüljön, hogy a szülőosztály kezelje a saját privát mezőinek klónozását.
3. A klónozó metódus jellemzően csak egy sort tartalmaz: egy új operátort hajt végre a prototípus verziójával a konstruktorból. Fontos megjegyezni, hogy minden osztálynak explicit módon felül kell írnia a klónozó metódust, és saját osztálynevével kell használnia az új operátort. Ellenkező esetben a klónozó metódus egy szülőosztály objektumát is létrehozhatja.
4. Opcionálisan létrehozhatunk egy központosított prototípus regisztert, hogy tárolja a gyakran használt prototípusok katalógusát.

A regisztert új gyári osztályként valósíthatjuk meg, vagy beágyazhatjuk a prototípus alap osztályába, egy statikus metódussal a prototípusok lekérésére. Ennek a metódusnak a kliens kódja által átadott keresési kritériumok alapján kell keresnie a prototípust. A kritérium lehet egyszerű szöveges címke vagy egy komplex keresési paraméterek halmaza. Miután megtalálta a megfelelő prototípust, a regiszternek klónoznia kell azt, és vissza kell adnia a másolatot a kliensnek.

Végül cseréljük le az alosztályok konstruktoraival való közvetlen hívásokat a prototípus regiszter gyári metódusának hívására.

Pro

1. Az objektumokat anélkül másolhatjuk, hogy azok konkrét osztályaihoz kellene kötődni.
2. Megszabadulhatunk az ismétlődő inicializáló kódtól a már elkészített prototípusok klónozásával.
3. Kényelmesebben hozhatunk létre összetett objektumokat.
4. Alternatívát kínál az öröklődéshez, amikor összetett objektumok konfigurációs előbeállításait kell kezelni.

Contra

1. Az összetett objektumok klónozása, amelyek körkörös hivatkozásokat tartalmaznak, nagyon trükkössé válhatnak.

Bridge

A Bridge egy szerkezeti tervezési minta, amely lehetővé teszi, hogy egy nagy osztályt vagy szorosan összefüggő osztályok halmazát két különböző hierarchiára bontsuk: az absztrakcióra és az implementációra, melyek egymástól függetlenül fejleszthetők.

A Bridge minta arra próbál megoldást találni, hogy az öröklődésről az objektum kompozícióra vált. Ez azt jelenti, hogy az egyik dimenziót egy külön osztályhierarchiába helyezzük, így az eredeti osztályok nem az összes állapotukat és viselkedésüket egyetlen osztályon belül tartalmazzák, hanem egy új hierarchia objektumára hivatkoznak.

Ezt a megközelítést követve a színhez kapcsolódó kódot külön osztályba vonhatjuk, két alosztállyal: Piros és Kék. A Shape osztály ezután egy referencia mezőt kap, amely a színobjektumok egyikére mutat. Így a Shape osztály bármilyen színhez kapcsolódó feladatot

átdelegálhat a hivatkozott színobjektumnak. Ez a referencia hidat képez a Shape és a Color osztályok között. Innentől kezdve új színek hozzáadása nem igényli a Shape hierarchia módosítását, és fordítva sem szükséges.

Absztrakció és Implementáció

A GoF könyv az Absztrakció és Implementáció kifejezéseket a Bridge minta részeként vezeti be.

Az absztrakció (más néven interfész) egy magas szintű vezérlő réteg egy entitás számára. Ennek a rétegnek nem szabad önállóan végeznie semmilyen tényleges munkát. A munkát az implementációs rétegre (más néven platformra) kell delegálnia.

Fontos megjegyezni, hogy itt nem a programozási nyelvekben használt interfészekről vagy absztrakt osztályokról beszélünk. Ezek nem ugyanazok.

Ha valódi alkalmazásokat nézünk, az absztrakciót reprezentálhatja egy grafikus felhasználói felület (GUI), míg az implementáció lehet az alatta futó operációs rendszer kódja (API), amelyet a GUI réteg hív meg a felhasználói interakciókra válaszul.

Általánosságban elmondható, hogy egy ilyen alkalmazást két független irányban bővíthetünk:

- Több különböző GUI-t alkalmazhatunk (például, amelyek a rendszeres felhasználókra vagy az adminisztrátorokra vannak szabva).
- Támogathatunk több különböző API-t (például, hogy képesek legyünk futtatni az alkalmazást Windows, Linux és macOS rendszereken).

A legrosszabb esetben az alkalmazás egy óriási spagetti tálhoz hasonlíthat, ahol százával kapcsolódnak különböző típusú GUI-k és API-k a kód minden egyes részén.

Hogyan implementáljuk?

1. Azonosítsuk az ortogonális dimenziókat az osztályaiban. Ezek független fogalmak lehetnek: absztrakció/platform, domain/infrastruktúra, front-end/back-end vagy interfész/implementáció.
2. Nézzük meg, milyen műveletekre van szüksége az ügyfélnek, és határozzuk meg őket az alap absztrakciós osztályban.
3. Határozzuk meg azokat a műveleteket, amelyek minden platformon elérhetők. Deklaráljuk azokat, amelyeket az absztrakciónak szüksége van az általános implementációs interfészben.
4. Minden platformra a domain-en belül hozzunk létre konkrét implementációs osztályokat, de ügyeljünk arra, hogy mindegyikük kövesse az implementációs interfészt.
5. Az absztrakciós osztályban adjunk hozzá egy referencia mezőt az implementáció típusának. Az absztrakció a legtöbb munkát az implementációs objektumnak delegálja, amelyet ebben a mezőben hivatkozunk.

6. Ha több változata van a magas szintű logikának, akkor hozzuk létre az egyes változatok finomított absztrakcióit az alap absztrakciós osztály kiterjesztésével.
7. Az ügyfélszoftvernek implementációs objektumot kell átadnia az absztrakció konstruktorának, hogy összekapcsolja a kettőt. Ezt követően az ügyfél elfelejtheti az implementációt, és csak az absztrakciós objektummal dolgozhat.

Pro

Létrehozhatunk platformfüggetlen osztályokat és alkalmazásokat.

Az ügyfélszoftver magas szintű absztrakciókkal dolgozik, így nincs kitéve a platform részleteinek.

Open/Closed Principle (Nyitott/Zárt elv): Új absztrakciókat és implementációkat vezethetünk be függetlenül egymástól.

Single Responsibility Principle (Egységes felelősség elve): Az absztrakcióban a magas szintű logikára, míg az implementációban a platform részleteire összpontosíthatunk.

Contra

Előfordulhat, hogy bonyolultabbá tesszük a kódot, ha ezt a mintát egy erősen koherens osztályra alkalmazzuk.