Az Objektumorientált programozás és a Programtervezési minta

Tartalom

Bevezetés

Az objektumorientált programozás (OOP) a szoftverfejlesztés területén már évtizedek óta domináns szerepet tölt be, és a tervezési minták alkalmazása ezen paradigma keretében kiemelkedő jelentőséggel bír.

Az informatikában a **programtervezési mintának** (*Software Design Patterns*) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.

A tervminták nem nyújtanak kész tervet, amit közvetlenül le lehet kódolni, habár vannak hozzájuk példakódok, amiket azonban meg kell tölteni az adott helyzetre alkalmas kóddal. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.
A minták rendszerint osztályok és objektumok közötti kapcsolatokat mutatnak, de nem specifikálják konkrétan a végleges osztályokat vagy objektumokat. A modellek absztrakt osztályai helyett egyes esetekben interfészek is használhatók, habár azokat maga a tervminta nem mutatja. Egyes nyelvek beépítetten tartalmaznak tervmintákat.
A tervminták tekinthetők a strukturált programozás egyik szintjének a paradigma és az algoritmus között.
A legtöbb tervminta objektumorientált környezetre van kidolgozva. Mivel a funkcionális programozás kevéssé ismert és használt, arra a környezetre még csak kevés tervminta ismert,

A programtervezési minták fogalma Christopher Alexander építész ötlete nyomán született meg. Ő volt az, aki olyan, az építészetben újra és újra felbukkanó mintákat keresett, amelyek a jól megépített házakat jellemzik. Könyvében, a „The Timeless Way of Buliding”-ben olyan mintákat próbált leírni, amelyek segítségével akár egy kezdő építész is gyorsan jó épületeket tervezhet. A minták a létrehozó építészek sok éves tapasztalata miatt szebb, jobb vagy használhatóbb házakat eredményeztek, mintha a tervezőnek csupán saját erejére támaszkodva kellett volna megterveznie azokat.

Objektumorientált Programozás alapjai

Az **objektumorientált** vagy **objektumelvű programozás** (OOP) az objektumok fogalmán alapuló programozási paradigma. Az objektumok egységbe foglalják az adatokat és a hozzájuk tartozó műveleteket. Az objektum által tartalmazott adatokon általában az objektum metódusai végeznek műveletet. A program egymással kommunikáló objektumok összességéből áll. A legtöbb objektumorientált programozási nyelv osztályalapú, azaz az objektumok osztályok példányai, és típusuk az osztály.

Objektumorientált Programozás története

Az 1960-as években fejlesztették ki az első objektumorientált nyelvet, a Simulát. Ebben volt objektum, osztály, öröklődés és dinamikus kötés. Kezelte az adatok biztonságát is, és szemétgyűjtéssel is el volt látva, ami automatikusan kitakarította a nem használt objektumokat a RAM-ból. Láthatósági szintek nem voltak benne, mivel egy nem publikus adattaghoz nem férhetett volna hozzá publikus metódus.

Az 1980-as évek közepén újabb objektumorientált nyelveket fejlesztettek ki, Brad Cox az Objective-C nyelvet és Bjarne Stroustrup (Simulával írta a doktori értekezését) a C++ nyelvet. Az 1990-es évek elején és közepén az objektumorientáció vált a programozás fő paradigmájává. Ezt támogatták azok az eszközök, amelyekkel az objektumorientált nyelvek elterjedhettek és népszerűvé váltak.

A legfontosabb objektumorientált nyelvek: Java, C++, C#, Python, PHP, Ruby, Perl,

Objektumorientált Programozás alapeszközei

Példányosítás

Egy osztály példányosítása során az osztály definíció alapján az objektum belső állapotának tárolásához szükséges tárhely lefoglalásra kerül, illetve a műveletek hozzárendelődnek. Tehát a példány egy absztrakció konkrét megjelenése, amelyre az osztályban definiált metódusok alkalmazhatók, illetve van állapota, amellyel e metódusok hatásait képes tárolni.

Osztály, felület, alosztály, altípus, öröklés

Osztály: Egy osztállyal reprezentálhatjuk az objektumok egy halmazát, melyeket ugyanazon attribútumokkal írhatunk le, illetve viselkedésük, kapcsolataik egyeznek. Egy objektum osztálya megadja annak belső állapotleírását, illetve a rajta értelmezett műveleteket. Egy típus az adott objektum felületére, azaz a rajta értelmezett metódusokra utal.

Az objektumorientált tervezés kulcspontja az öröklődés. Az öröklődés jelentése, hogy egy már meglévő osztályban definiált adatstruktúrákat, metódusokat öröklik az adott osztályból származó alosztályok úgy, hogy kiegészíthetik azokat új adatstruktúrákkal, metódusokkal, illetve az öröklött tulajdonságokat felül is definiálhatják. Az öröklődés teszi logikusan átláthatóvá a tervet, és megszabadítja azt a kód redundanciáktól

Öröklésről beszélünk, mikor típusok, osztályok közös tulajdonságait kiemelve, azokat egy közös új típus-osztály tag változóiként és metódusaiként deklaráljuk, melyeket örökölni fognak a speciális kiegészítéseket adó altípusok, alosztályok. Altípus, azaz felületöröklés esetén azt adjuk meg, hogy egy objektumot mikor használhatunk egy másik helyett.

Osztályöröklés, alosztály esetén a szülőosztály szolgáltatásait tudjuk bővíteni, lehetőséget adva arra, hogy azok felé közös felületet biztosítsunk, ami a többalakúság alapja. Ha az osztályöröklés során az alosztály csak felülírhatja a műveleteket vagy újakat határozhat meg, de a szülőosztály műveleteit nem tudja elrejteni, akkor minden alosztály objektuma tudja fogadni a szülőosztálynak szánt kérelmeket, azaz a szülőosztály felületén értelmezett műveleteket, tehát az alosztály egyben altípus is.

Osztály hierarchiák, azaz öröklődés esetén „is a" kapcsolatról beszélünk. Az öröklés statikus megoldást nyújt a felületek kiegészítésére, működésének részleges felülírására. A kód újrahasznosításának kifinomult eszköze, mivel kiegészítheti, módosíthatja a szülőosztály viselkedését úgy, hogy utána ugyanazon felületen keresztül elérhetők annak szolgáltatásai. Azonban hátrány, hogy ez a statikus függőség nem oldható fel, azaz a szülőosztály által nyújtott megvalósítások futásidőben nem változtathatók meg. Szintén kritikus problémát jelent, hogy a szülőosztály megvalósításától való függés a szülőosztály változása esetén kockázatot jelenthet. Ennek megoldása az lehet, hogy csak absztrakt osztályoktól öröklünk, amelyek nem vagy csak részlegesen rendelkeznek megvalósítással. A többszörös öröklődés egy olyan osztályok közötti kapcsolat, mellyel egy osztály az adatstruktúráját és/vagy viselkedését meg tudja osztani más osztályokkal. Tehát ha egy alosztálynak két szülőosztálya van, akkor többszörös öröklésről beszélünk.

Polimorfizmus

A többalakúságon – polimorfizmuson azt értjük, hogy egy referencia által mutatott objektum típusa egy osztályhierarchián belül futásidőben dől el, azaz „többalakú". Ennek megfelelően egy osztályban definiált metódusok az alosztályokban felüldefiniálhatók, így csak futásidőben kerül meghatározásra (kései kötés, „Late Binding"), hogy ténylegesen melyik kód kapja meg a vezérlést. Tehát polimorfizmus esetén egy objektumnak több típusa is lehet, és az ősosztály által meghatározott felület, metódus, kérés az alosztály által felüldefiniált. Lehetséges, hogy az adott kérést különböző objektumok (szülő vagy alosztály objektuma) kapják meg, azaz ugyanarra a kérésre más-más megvalósítás fog érvényre jutni.

OOP előnyei és hátrányai

Az OOP előnyei közé tartozik a kód újrafelhasználhatósága, a moduláris felépítés, a könnyebb karbantartás és a hibák csökkentett kockázata. Az objektumorientált megközelítés támogatja a bonyolult programok strukturált és rendezett fejlesztését.

Az OOP hátrányai közé sorolható a potenciálisan nagyobb erőforrás-igény a több objektum létrehozása és kezelése miatt, valamint a tanulási görbe meredeksége, különösen azok számára, akik először találkoznak ezzel a paradigmával.

A programtervezési minták definíciója

A programtervezési minták „egymással együttműködő objektumok és osztályok leírásai, amelyek testre szabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben”.

A szoftvertervezésben egy-egy problémára végtelen sok különböző megoldás adható, azonban ezek között kevés optimális van.

Tapasztalt szoftvertervezők, akik már sok hasonló problémával találkoztak, könnyen előhúzhatnak egy olyan megoldást, amely már kipróbált és bizonyított.

Kezdő fejlesztőknek viszont jól jön, ha mindazt a tudást és tapasztalatot, amit csak évek munkájával érhetnek el, precízen dokumentálva kézbe vehetik és tanulhatnak belőlük A programtervezési minták ilyen összegyűjtött tapasztalatok, amelyek mindegyike egy-egy gyakran előforduló problémára ad általánosított választ.

Előnyük

- lerövidítik a tapasztalatszerzési időt. A programtervezési mintákat nem „feltalálták”, hanem a gyakran előforduló problémákra adott optimális válaszokat gyűjtötték össze, ezáltal olyan megoldásokat adtak, amelyekre előbb-utóbb a legtöbb fejlesztő magától is rájönne – csak esetleg jóval később.

- lerövidítik a tervezési időt. Az összes minta jól dokumentált, könnyen újrafelhasználható, így ha egyszer alkalmazzuk őket, jó eséllyel egy hasonló problémánál újra eszünkbe fognak jutni az összes előnyükkel és hátrányukkal együtt. Így azonnal hatékony, rugalmas megoldást adhatunk és megkímélhetjük magunkat sok tervezéstől és esetleges újratervezéstől.

- közös szótárat ad a fejlesztők kezébe. Ez megkönnyíti az egymás közti kommunikációt és a program dokumentálását is, hiszen könnyebb úgy beszélni egy probléma megoldásáról, ha van egy közös alap, ahonnan indulunk vagy amihez lehet hasonlítani az új terveket.

- magasabb szintű programozást tesz lehetővé. Mivel ezek a minták elterjedtségük miatt már kiállták nagyon sok programozó próbáját, feltehetőleg az optimális megoldást tartalmazzák a problémára.

Tervezési minták osztályozása

A tervezési minták a modulokat és kapcsolataikat szervezik. Alacsonyabb szintűek, mint az architekturális minták, amelyek a teljes rendszer általános felépítését jellemzik.

Több különböző tervminta létezik, például:

- Algoritmus stratégia minták: Magas szintű stratégiák, amelyek leírják, hogyan kell algoritmust szervezni egy adott architektúrára.

- Számítástervezési minták: A kulcsfontosságú számítások megkeresését célozzák.

- Végrehajtási minták: A végrehajtás alacsonyabb szintjén címzésekkel, feladatok végrehajtásának szervezésével, optimalizálásával, szinkronizálásával foglalkozó minták.

- Implementációs stratégia minták: A forráskód implementációjában támogatják a program szervezését, és a párhuzamos programozás számára fontos adatszerkezetek felépítését.

- Szerkezeti tervezési minták: Az alkalmazás globális struktúrájával foglalkoznak.

A tervmintákat három kategóriába csoportosítják:

- létrehozási minták
- szerkezeti minták
- viselkedési minták

Egyes szerzők elkülönítik az architekturális tervezési mintákat is, mint a modell-nézet-vezérlő.

Modell-Nézet-Vezérlő

Az MVC (Modell-Nézet-Vezérlő) egy széles körben alkalmazott tervezési minta az objektumorientált programozásban. Ez a minta három fő komponensre osztja a szoftverarchitektúrát:

- Modell: Az adatok és azokat kezelő logika reprezentációja. A modell felelős az adatok tárolásáért, kezeléséért és validálásáért, valamint az üzleti logika implementálásáért.

- Nézet: A felhasználói felület elemei, amelyek megjelenítik a modell adatait. A nézet felelős a modell adatok vizuális reprezentációjáért.

- Vezérlő: Az a komponens, amely összeköti a modellt és a nézetet. A vezérlő fogadja a felhasználói inputokat, feldolgozza azokat, és frissíti a modellt vagy a nézetet.

Az MVC minta előnyei közé tartozik az elkülönített felelősségi körök, amelyek elősegítik a fejlesztési folyamat modularitását és tesztelhetőségét. Az elkülönítés lehetővé teszi, hogy a fejlesztők külön dolgozhassanak a modell, a nézet, és a vezérlő különböző aspektusain, növelve ezzel a kód karbantarthatóságát és rugalmasságát. Az MVC különösen népszerű webalkalmazások és asztali alkalmazások fejlesztésénél, ahol a felhasználói interakciók és adatok megjelenítése kiemelten fontos.

Az MVC minta számos programozási nyelvben és keretrendszerben megtalálható. Például a Java Spring MVC keretrendszer, a Ruby on Rails, és az ASP.NET MVC, mind-mind az MVC mintát használják az alkalmazások struktúrájának meghatározására. Ezek a keretrendszerek biztosítanak sablonokat és könyvtárakat, amelyek megkönnyítik az MVC minta implementálását, így segítve a fejlesztőket a gyors és hatékony szoftverfejlesztésben.

Egyéb Tervezési Minták

Létrehozási minták

- *Absztrakt gyár* (Abstract factory): Létrehoz egy interfészt egymástól függő vagy valamilyen módon összekapcsolódó objektumok családjainak anélkül, hogy megadnánk a konkrét osztály típusát.

  Ennek a mintának a használata, lehetővé teszi egy rendszerben a konkrét típus implementációk kicserélését (még akár futásidőben is) anélkül, hogy az őket használó kódot módosítanánk. Mindazonáltal ennek a mintának a használata, szükségtelen komplexitást okozhat, és plusz munkát igényelhet a kezdeti kódkészítésben. Továbbá az absztrakció és a szétválasztás magasabb szintje olyan rendszert eredményezhet, amiben nehezebb a hibakeresés és a karbantartás.

- *Építő*: Egy objektum felépítési folyamattal több, különböző szerkezetű objektumok létrehozására. A létrehozás folyamata független az objektum ábrázolásától.

  Az Építő minta gyakran Összetétel tervezési mintát készít. Előfordul, hogy a minták Gyár mintát alkalmazva indulnak mely továbbfejlődik Absztrakt Gyárrá, Prototípussá vagy Építővé, ahogy a fejlesztő felfedezi, hogy hol van szükség nagyobb rugalmasságra. Néha a létrehozási minták kiegészítik egymást: az Építő minta használhat egy másik mintát, hogy megállapítsa, milyen más komponensek jönnek létre. Az Építők jól alkalmazhatók könnyed interfészekhez.

- *Gyártó metódus (Factory method)*: Meghatároz egy interfészt egy objektum létrehozására, de a leszármazottakra van bízva az objektum típusának meghatározása. Segítségével elkerülhető, hogy alkalmazás specifikus osztályokat rakjunk a kódba.

- *Lusta inicializáció*: Objektum létrehozás, költséges számítás, stb. késleltetése addig, amíg az objektum/érték nem válik szükségessé. Ez a programtervezési minta egy olyan taktika, amely szerint késleltetjük egy objektum létrehozását, vagy valamely számításigényes művelet elvégzését egészen addig, amíg az objektumra vagy a számítás eredményére először ténylegesen szükség lesz.

  Ezzel a viselkedéssel az objektum létrehozását „elhalasztják” az első használatig, amely bizonyos körülmények között, csökkenti a rendszer válaszidejét és gyorsítja az indítást azáltal, hogy elkerüli a nagyméretű objektumok előzetes létrehozását és a memóriafoglalását.

  Több szálon futó kód esetén, a lusta inicializációval használt objektumokhoz való hozzáférést szinkronizálni kell, a versenyhelyzet (race condition) elkerülése érdekében.

- *Többke*: Hasonlóan az egyke tervezési mintához, egy példány készítését teszi lehetővé, viszont nem alkalmazásonként hanem kulcsonként. A multiton osztály tartalmaz egy mapot a példányokról (kulcs-érték párok).

  Többke tervezési minta hasonlít az Egyke programtervezési mintára. Míg az Egyke programtervezési minta csak egy osztály példány létrehozását engedi, addig a Többke tervezési minta továbbfejleszti ezt a koncepciót: lehetővé teszi megnevezett példányok, mint kulcs-érték párok mappelését. Ahelyett, hogy az alkalmazásunkban egyetlen példányt alkalmaznánk, a Többke minta minden kulcshoz külön példányt biztosít.

  A Többke úgy tűnhet nem több, mint egy egyszerű hash tábla, szinkronizált hozzáféréssel. Az első fontos különbség, hogy a Többke nem teszi lehetővé a klienseknek a mappelést. A második, hogy sosem ad vissza null vagy üres referenciát. Ez a minta létrehoz egy Többke példányt az első, a hozzá tartozó kulcshoz kapcsolódó kéréssel. Az ilyen kulccsal rendelkező további kérések az eredeti példányt fogják visszaadni.

  Ezzel szemben a hash tábla használata pusztán végrehajtási részlet és nem az egyetlen lehetséges megközelítés. A minta leegyszerűsíti a megosztott objektumok lekérését egy alkalmazásban.

  Hátránya, hogy az Egyke programtervezési minta is, megnehezíti a unit tesztek használatát, mivel globális állapotot vezet be az alkalmazásban. A garbage collectort alkalmazó nyelveknél ez memóriavesztést eredményezhet, mivel bevezeti a globális erős kötést az objektumhoz.

- *Objektumkészlet*: Objektumok újrafelhasználása. Objektumok példányosítása és törlése költséges művelet lehet, ezért előre legyártott objektumokat használ az alkalmazás, és miután egy szükségtelenné vált, visszahelyezi azt az objektumkészletbe. Tipikus példája a szálkészlet, amely processzek által használatos szálakat tárol. Az **objektumkészlet** programtervezési minta egy létrehozási minta, amely inicializált objektum példányok egy csoportját tartja használatra az igények kiszolgálásához, ahelyett, hogy a keresletnek megfelelően folyamatosan újonnan létrehozná, illetve megsemmisítené azokat. A készlet egy kliense egy objektum példányt igényel a készletből és bizonyos műveleteket hajt végre a kapott objektum segítségével. Amikor a kliens ezekkel végzett, „visszateszi“ az objektumot a készletbe, ahelyett, hogy megsemmisítené azt.

  Az objektumkészlet minta elsősorban a teljesítménnyel kapcsolatos okok miatt használatos: bizonyos körülmények között, a minta alkalmazása jelentősen növeli a teljesítményt. Az objektumkészlet használata „megbonyolítja“ az objektum életciklusát, mivel az objektumok csak „ki vannak véve“ a készletből illetve „vissza vannak téve“ oda, nem pedig ténylegesen létrehozva és megsemmisítve.

  Az objektumkészlet használata jelentős teljesítmény növekedést nyújthat azokban a szituációkban, ahol egy objektum példány létrehozása költséges és az adott objektum osztályából nagy gyakorisággal hozunk létre illetve semmisítünk meg

példányokat. Ebben az esetben az objektum példányok sokszor újrahasznosíthatóak, és minden újrahasznosítással jelentős mennyiségű időt és teljesítményt spórolunk meg.

- *Prototípus (Prototype)*: Meghatároz egy előzetes mintát az objektumok létrehozásához, és később ez kerül másolásra. Gyakran használjuk, ha az objektum pontos típusa csak futásidőben derül ki.

  A minta lényege a klónozás, azaz az eredeti objektummal megegyező új példány létrehozása. Az egyszerű értékadás erre nem alkalmas, mivel az csak az objektum hivatkozását másolja le, melynek eredményeképpen az eredeti példány és másolata ugyanoda hivatkozik. Két típust különböztetünk meg, a sekély és a mély klónozást. A sekély klónozás esetében az osztály által hivatkozott objektumokat ugyanúgy másoljuk, mint elemi típusú tulajdonságait. A mély klónozásnál az osztály által hivatkozott objektumokat is klónozzuk.

- *Egyke (Singleton)*: Biztosítja, hogy az osztályból csak egy példány készül, és biztosít egy publikus hozzáférést ehhez a példányhoz.

  Gyakori, hogy egy osztályt úgy kell megírni, hogy csak egy példány legyen belőle. Ehhez jól kell ismerni az objektumorientált programozás alapelveit. Az osztályból példányt a konstruktorával lehet készíteni. Ha van publikus konstruktor az osztályban, akkor akárhány példány készíthető belőle, tehát publikus konstruktora nem lehet az egykének. De ha nincs konstruktor, akkor nem hozható létre a példány, amin keresztül hívhatnánk a metódusait. A megoldást az osztályszintű (statikus) metódusok jelentik. Ezeket akkor is lehet hívni, ha nincs példány. Az egykének tehát van egy osztályszintű metódusa, ami minden hívójának ugyanazt a példányt adja vissza. Természetesen ezt a példányt is létre kell hozni, ehhez privát konstruktort kell készíteni, amit a szerezPéldány az egyke osztály tagjaként meghívhat.

  Sokan erősen kritizálják az egyke mintát, és antimintának tekintik, mivel szükségtelen korlátozásokat és globális állapotokat helyez el az alkalmazásban, illetve csökkenti a tesztelhetőséget

Szerkezeti minták

- Illesztő minta: Egy felület átalakítása, illesztése egy másikra. Összeférhetetlen osztályok együttműködését teszi lehetővé.

- Híd: Lehetővé teszi az absztrakció (interfész) és az implementáció szétválasztását, így a kettő külön változtatható egymástól függetlenül.

- Összetétel: Az objektumok faszerkezetbe való ábrázolásának megvalósítása. Egységesíti a kezelését az objektumnak és objektumok összetételének.

- Díszítő: Lehetővé teszi az absztrakció változtatása nélkül további funkciók, felelősségi körök dinamikus hozzáadását.

- Homlokzat: Egy egységes magasabb szintű interfészt hoz létre egy alrendszer számos interfészének. Leegyszerűsíti a bonyolult rendszert, általa könnyebben használhatók az alrendszer funkciói.

- Pehelysúlyú minta: Sok egyforma objektum erőforrásaival hatékonyan gazdálkodik oly módon, hogy egy külön objektumban tárolja az újrahasznoítható (ismétlődő) funkciókat, tulajdonságokat.

- Front vezérlő: Web alkalmazásokhoz kapcsolódó tervezési minta. A beérkező kérések feldolgozásának és kiszolgálásának vezérléséhez használt módszer.

- Modul: Egy sor egymással összefüggő elemet (mint az osztályok, eljárások stb.) csoportosít egy közös konceptuális entitásba (modulba).

- Helyettes: Egy másik objektum elfedésére, helyettesítésére alkalmazott tervezési minta.

- Iker: Lehetőséget ad a többszörös öröklődés megvalósítására olyan programozási nyelvekben, ahol ez nem támogatott.

Viselkedési minták

- Blackboard: Egy általánosított observer, amely engedélyez több írót és olvasót. Rendszerszintű az információ áramlása.

- Felelősséglánc: Az üzenet vagy kérés küldőjének függetlenítése a fogadótól. Megvalósítása felelősségláncok kialakításával történik. Lényegében egy

láncolt lista, amin a kérelem végig halad mindaddig, amíg egy objektum le nem tudja kezelni.

- Parancs: Kérelmek objektumba ágyazása. Ezáltal a klienseknek különböző parancsokat adhatunk át, amit naplózhatunk, sorba rendezhetünk és a visszavonást kezelhetjük le. Azaz az egyes kérelmek teljesen kivizsgálhatóak és hatálytalaníthatók lesznek.

- Értelmező: Meghatározza az adott nyelv értelmezőjének a forráskód értelmezésének nyelvtanát, szabályait.

- Iterator: Lehetővé teszi egy aggregált objektum elemeinek szekvenciális elérését anélkül, hogy láthatóvá tenné a mögöttes reprezentációját.

- Közvetítő: Több egymással kapcsolatot tartó objektum kommunikációjának egységbe (objektumba) zárása. Definiál egy interfészt, amin keresztül objektumokkal kommunikálhat.

- Memento: Az egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát. Így az objektum állapota később visszaállítható.

- Üres objektum: Null referenciák elkerülése érdekében null helyett egy alapértelmezett objektummal (null object) térnek vissza a metódusok.

- Megfigyelő: Meghatároz ez egy-a-többhöz függőséget objektumok között. Egy adott objektum módosulásáról automatikus értesítő információt küld a tőle függő objektumoknak, amik ezek alapján frissülnek.

- Szolga: Osztályok csoportjának biztosít funkciókat, melyeket elvégez az osztályok helyett. Az adott funkciók csak a szolga osztályban vannak megvalósítva.

- Specifikáció: Egyesíti az alkalmazások üzleti logikáját Boole-algebra-szerűen.

- Állapot: Az objektum viselkedése megváltoztatható a belső állapottól függően.

- Stratégia: Egységbe zárt, azonos interfésszel rendelkező algoritmusok dinamikus cserélgetése. Lehetővé teszi, hogy az algoritmus az őt használó kliensektől függetlenül változzon.

- Sablonfüggvény: Az algoritmus vázának megvalósítása oly módon, hogy bizonyos közbenső végrehajtási lépések specializálhatók legyenek az alosztályokban.

- Látogató: Lehetőséget ad új funkciók hozzáadására anélkül, hogy megváltoztatnánk az osztályok szerkezetét.