Kusper Gábor
Eszterházy Károly Főiskola
gkusper@aries.ektf.hu

Márien Szabolcs
Wit-Sys ZRt.
szabolcs.marien@wit-sys.hu

AZ OBJEKTUM-ORIENTÁLT TERVEZÉSI ALAPELVEK KRITIKAI VIZSGÁLATA

Absztrakt

A szakirodalom jelenleg a következő objektum-orientált tervezési alapelveket fogadja el:

- GOF1: Programozz felületre implementáció helyett.
- GOF2: Használj objektum összetételt öröklés helyett, ha csak lehet.
- SRP (Single Responsibility Principle): Az osztálynak csak egy oka legyen a változásra.
- OCP (Open-Closed Principle): Az osztályok legyenek nyitottak a bővítésre, de zártak a módosításra.
- LSP (Liskov Substitutional Principle): Az alosztály legyen egyben altípus is.
- DIP (Dependency Inversion Principle): Absztrakciótól függj, ne függj konkrét osztályoktól.
- ISP (Interface Segregation Principle): Egy kliens se legyen rászorítva, hogy olyan metódusoktól függjön, amiket nem is használ.
- HP (Hollywood Principle): Ne hívj, majd mi hívunk.

– Ezek közül ebben a rövid cikkben csak az első öttel foglalkozunk.

A tervezési alapelvek közti összefüggéseket a szakirodalom nem tárgyalja, nem lehet tudni, hogy az egyes elvek közül melyik az erősebb. Ennek a hiánynak a betöltése a célunk jelen cikkben.

1. Bevezetés

Az objektum orientált tervezés alapelvei (object-oriented design principles) a tervezési mintáknál [GOF95] magasabb absztrakciós szinten írják le, milyen a „jó” program. A tervezési minták ezeket az alapelveket valósítják meg szintén még egy elég magas absztrakciós szinten. Végül a tervezési mintákat realizáló programok az alapelvek megvalósulásai. Az alapelveket természetesen úgy is fel lehet használni, hogy nem követjük a tervezési mintákat.

A tervezési alapelvek abban segítenek, hogy több, általában egyenértékű programozói eszköz (pl. öröklődés és objektum összetétel) közül kiválasszuk azt, amely jobb kódot eredményez. Általában jó a kód, ha rugalmasan bővíthető, újrafelhasználható komponensekből áll és könnyen érthető más programozók számára is.

A tervezési alapelvek segítenek, hogy ne essünk például abba a hibába, hogy egy osztályba kódolunk mindent, hogy élvezzük a mezők, mint globális változók programozást gyorsító hatását. A tapasztalat az, hogy lehet programozni az alapelvek ismerete nélkül, vagy akár tudatos megszegésével, csak nem érdemes. Gondoljunk a programozási technológiák alapelvére: „A program kódja állandóan változik!" [KusRad11]. Azaz, ha rugalmatlan programot írunk, akkor a jövőben keserítjük meg az életünket, amikor egy, akár előre sejthető, változást kell beleilleszteni a programunkba. Inkább érdemes a jelenben több időt rászánni a fejlesztésre és biztosítani, hogy a jövőben könnyű legyen a változások kezelése. Ezt biztosítja számunkra az alapelvek betartása.

A szakirodalom jelenleg az absztraktban felsorolt objektum-orientált tervezési alapelveket fogadja el [GOF95, Martin02]. Ezek: GOF1, GOF2, SRP, OCP, LSP, DIP, ISP, HP. Ezek közül ebben a rövid cikkben csak az első öttel foglalkozunk.

A tervezési alapelvek közti összefüggéseket a szakirodalom nem tárgyalja, nem lehet tudni, hogy az egyes elvek közül melyik az erősebb. Ennek a hiánynak a betöltése a célunk jelen cikkben.

1.1. GOF1

A GOF1 alapelv a „Design Patterns: Elements of Reusable Object-Oriented Software" című könyvben [GOF95] jelent meg 1995-ben. A könyv magyar címe: „Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz." Az alapelv eredeti angol megfogalmazása: „Program to an interface, not an implementation", azaz „Programozz felületre implementáció helyett".

Mit jelent ez a gyakorlatban? Egyáltalán, hogy lehet implementációra programozni? Miért rossz implementációra programozni? Miért jó felületre?

Akkor programozunk implementációra, ha kihasználjuk, hogy egy osztály hogyan lett implementálva. Az InstrumentedHashSet [Tarr04, 15. dia] osztály jó példa arra, amikor tudnunk kell, hogyan lett az ős implementálva. Egy másik példa NagySzám osztály a [KusRad2011] jegyzetből.

Ha implementációra programozunk, és ha megváltozik az osztály, akkor a vele kapcsolatban álló osztályoknak is változniuk kell. Ezzel szemben, ha felületre programozunk, és megváltozik az implementáció, de a felület nem, akkor nem kell megváltoztatni a többi osztályt.

1.2. GOF2

A GOF2 alapelv a [GOF95] könyvben jelent meg 1995-ben. Az alapelv eredeti angol megfogalmazása: „Favor object composition over class inheritance", azaz „Használj objektum összetételt öröklés helyett, ha csak lehet".

Mit jelent ez a gyakorlatban? Egyáltalán mit jelent az objektum összetétel? Miért jobb az öröklődésnél? Mi a baj az öröklődéssel? Ha jobb az objektum összetétel, akkor miért nem mindig azt használjuk?

Tudjuk, hogy objektum összetétellel mindig ki lehet váltani az öröklődést [KusRad011]. Az öröklés azért jó, mert megörököljük az ős összes szolgáltatását (metódusait), amit használni tudunk. Objektum összetételnél ezen osztály egy példányára szerzek egy referenciát és azon keresztül használjuk a szolgáltatásait. Ez utóbbi futási

időben dinamikusan változhat, hiszen az, hogy melyik objektumra mutat a referencia, futási időben változtatható.

Csatoltság szempontjából az öröklődés a legerősebb, éppen ez az oka, hogy a GOF2 kimondja, hogy használjunk inkább objektum összetételt öröklődés helyett, hiszen az kisebb csatoltságot eredményez és így rugalmasabb kódot kapunk. Ugyanakkor ki kell emelni, hogy az ilyen kód nehezebben átlátható, ezért nem szabad túlzásba vinni az objektum összetételt.

1.3. SRP

Az egy felelősség egy osztály alapelve (angolul: Single Responsibility Principle – SRP) azt mondja ki, hogy minden osztálynak egyetlen felelősséget kell lefednie, de azt teljes egészében. Eredeti angol megfogalmazása: „A class should have only one reason to change"[Martin02], azaz „Az osztálynak csak egy oka legyen a változásra".

Már a GOF1 elvnél is láttuk, hogy ha egy osztály nem fedi le teljesen a saját felelősségi körét, akkor muszáj implementációra programozni, hogy egy másik osztály megvalósítsa azokat a szolgáltatásokat, amik kimaradtak az osztályból.

Ha egy osztály több felelősségi kört is ellát, például a MacsKuty eszik, alszik, ugat, egerészik, akkor sokkal jobban ki van téve a változásoknak, mintha csak egy felelősséget látna el. A MacsKuty osztályt meg kell változtatni, ha kiderül, hogy a kutyák nem csak a postást ugatják meg, hanem a bicikliseket is, illetve akkor is, ha a macskák viselkedése változik vagy bővül.

Tudjuk, hogy minden módosítás magában hordozza a veszélyt, hogy egy forráskód szörnyet kapjunk, amihez már senki se mer hozzányúlni [Martin08]. Az ilyen kód fejlesztése nagyon drága.

1.4. OCP

Az Open-Closed Principle (OCP), magyarul a nyitva zárt elv, kimondja, hogy a program forráskódja legyen nyitott a bővítésre, de zárt a módosításra. Eredeti angol megfogalmazása: „Software entities (classes, modules, functions, etc.) should be open for extension butclosed for modification." [Meyer97, Martin02].

Az OCP elvet meg lehet fogalmazni a szintaxis szintjén is C# nyelv esetén: Ne használjuk az override kulcsszót, kivéve ha absztrakt vagy horog (angolul: hook) metódust írunk felül.

Ugyebár az absztrakt metódusokat muszáj felülírni, de ez nem az OCP megszegése, hiszen az absztrakt metódusnak nincs törzse, így lényegében a törzzsel bővítem a kódot, nem módosítok semmit. A másik eset, amikor használhatok felülírást, a horog metódusok felülírása. Akkor beszélek horog metódusokról, ha a metódusnak ugyan van törzse, de az teljesen üres. Ezek felülírása nem kötelező, csak opcionális, így arra használják őket, hogy a gyermek osztályok opcionálisan bővíthessék viselkedésüket. Ezek felülírásával lényegében megint csak bővítem a kódot, nem módosítom, azaz nem szegem meg az OCP elvet.

Ha egy kódban if – else if szerkezetet látunk, akkor az valószínűleg azt mutatja, hogy nem tartottuk be az OCP elvet. Nem tartottuk be, hiszen, ha új lehetőséget akarunk hoz-

záadni a kódhoz, akkor az if – else if szerkezetet tovább kell bővítenünk. Egy ilyen példát, illetve ennek javítását megtaláljuk a [KusRad11] jegyzetben.

1.5. LSP

A Liskov féle behelyettesítési elv, angolul Liskov Substitutional Principle (LSP), kimondja, hogy a program viselkedése nem változhat meg attól, hogy az ős osztály egy példánya helyett a jövőben valamelyik gyermek osztályának példányát használom. Azaz a program által visszaadott érték nem függ attól, hogy egy Kutya vagy egy Vizsla vagy egy Komondor példány lábainak számát adom vissza. Eredeti angol megfogalmazása: „If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T" [Liskov88].

Nézzünk egy példát, amely nem felel meg az LSP elvnek. A klasszikus ellenpélda az ellipszis – kör illetve a téglalap – négyzet példa. A kör olyan speciális ellipszis, ahol a két sugár egyenlő. A négyzet olyan speciális téglalap, ahol az oldalak egyenlő hosszúak. Szinte adja magát, hogy a kör az ellipszis alosztálya, illetve a négyzet a téglalap alosztálya legyen. A Téglalap – Négyzet példát lásd részletesen a [KusRad11] jegyzetben.

Az LSP egyik következménye a szerződés alapú tervezés (Design by Contract) [ManMey91], ami megengedi a nem absztrakt és nem horog metódusok felülírását is, de csak úgy, hogy a gyermek betartja az ős szerződéseit.

2. Összefüggések

Az alábbi összefüggéseket vettük észre az elemzés során:

- GOF1 ~ GOF2: Ha objektum összetételt akarok használni öröklődés helyett, azaz betartani a GOF2 elvet, akkor a megoldás gyakran egy absztrakt ős bevezetése. Ha az összetételt megvalósító referencia ilyen absztrakt ős típusú mindenhol, akkor ezzel betartom a GOF1 elvet is, hiszen az absztrakt osztálynak még nincs implementációja, amitől függhetnék.
- GOF2 ~ OCP: Ha objektum összetételt akarok használni öröklődés helyett, azaz betartani a GOF2 elvet, akkor a megoldás gyakran egy absztrakt ős bevezetése. Ha a gyermek osztályok csak az ősben megfogalmazott absztrakt és horog metódusokat írják felül, akkor betartjuk az OCP elvet is.
- GOF1 ~ SRP: Általában akkor kényszerülünk implementációra programozni, azaz megszegni a GOF1 elvet, ha az osztály felelősségi körét rosszul határoztuk meg és egy osztály a felelősséget nem teljesen fedi le, tehát megszegjük a SRP elvet. Erre látunk példát a [KusRad2011] jegyzetben, a NagySzám példában.
- GOF2 ~ SRP: Ha úgy sértjük meg az SRP elvet, hogy egy osztály több felelősségi kört is lefed, akkor megszegjük a GOF2 elvet is, mert a két felelősséget szét lehetne szedni két osztályra, amit egy harmadik foghatna össze.
- OCP ~ LSP: Ha a Téglalap – Négyzet példában, lásd [KusRad11] jegyzet, betartottuk volna az OCP elvet, akkor az LSP elvet se sértettük volna meg. Ezt úgy lehet elérni, hogy egyáltalán nem készítünk setA és setB metódust, mert akkor azokat mindenképpen felül kell írni. Csak konstruktort készítünk és a terület metódust.

3. Összefüggések

Az első fejezetben elemeztünk öt objektum-orientált tervezési alapelvet (GOF1, GOF2, SRP, OCP, LSP). A második fejezetben néhány általunk érvényesnek vélt összefüggésre világítottunk rá rövid magyarázattal. Az a kérdés adódik, hogy melyik a legerősebb elv, melyiket érdemes betartani?

Az irodalomban egyetértés van, hogy a két legfontosabb elv: SRP, OCP [Martin02]. Ugyanakkor az irodalom nem használja a GOF1 és GOF2 elveket, csak mint technikát írják le, név megnevezése nélkül. Mi, akik elismerjük ezt a két elvet is, az alábbi erősségi sorrendet fogadjuk el (a legerősebbel kezdve): GOF2, SRP, OCP, GOF1, LSP.

A legerősebb a GOF2, mert ennek használata nélkül a többi elv nem érhető el. Második az SRP, mert egy nagyon fontos heurisztikát ad a GOF2 alkalmazására. A harmadik az OCP, azzal a megjegyzéssel, hogy akár a második helyre is jogosult lenne. Az OCP olyan fontos megszorítás, ami a teljesen szabad programozót a tiszta kód [Martin08] irányába tereli, és általánosan elfogadott vélekedés szerint ez vezet a fő ellenség, a „program kódja állandóan változik" elv [KusRad11], megszelídítéséhez. Az utolsó elv, GOF1, az előbbiek betartásából már adódik.

Az LSP és az OCP közti viszony még nem teljesen tisztázott. Az LSP egyik következménye a szerződés alapú tervezés (design by contract), ami megengedi a nem absztrakt és nem horog metódusok felülírását is, de csak úgy, hogy a gyermek betartja az ős szerződéseit. Ebből a szemszögből az LSP az OCP egy finomítása. Ugyanakkor az OCP nem követeli meg a gyermektől az ős szerződéseinek betartását. Ez azért fájdalmas, mert egy absztrakt metódusnak is lehet szerződése. Ha az LSP jobban ki lenne dolgozva az irodalomban, akkor alkalmas lenne az OCP leváltására, így inkább annak kiegészítésé.

Munkánkat a Decision Lifting Principle (DLP) elv vizsgálatával szeretnénk folytatni, amely kimondja, hogy „Öröklést csak döntés kiemelésre használjunk!". Ez azért nagyon fontos, mert képes megmagyarázni a GOF2 elvben lévő „hacsak lehet" kitételt, lásd: „Használj objektum összetételt öröklés helyett, ha csak lehet!".

Felhasznált irodalom

[GOF95] The "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides); Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[KusRad11] Kusper Gábor, Radványi Tibor; Jegyzet a projekt labor című tárgyhoz, http://aries.ektf.hu/~gkusper/ProgTechJegyzet.v.1.1.docx, 2011.

[Liskov88] Barbara Liskov; Data Abstraction and Hierarchy, SIGPLAN Notices, 23(5), 1988.

[ManMey91] D. Mandrioli and B. Meyer; Design by Contract, Advances in Object-Oriented Software Engineering, Prentice Hall, 1–50, 1991.

[Martin02] Robert C. Martin; Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.

[Martin08] Robert C. Martin; Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.

[Meyer97] Bertrand Meyer; Object Oriented Software Construction, 2d. ed., Prentice Hall, 1997.

[Tarr04] Bob Tarr; Some Object-Oriented Design Principles, Design Patterns In Java, http://userpages.umbc.edu/~tarr/dp/lectures/OOPrinciples.pdf, 2004.