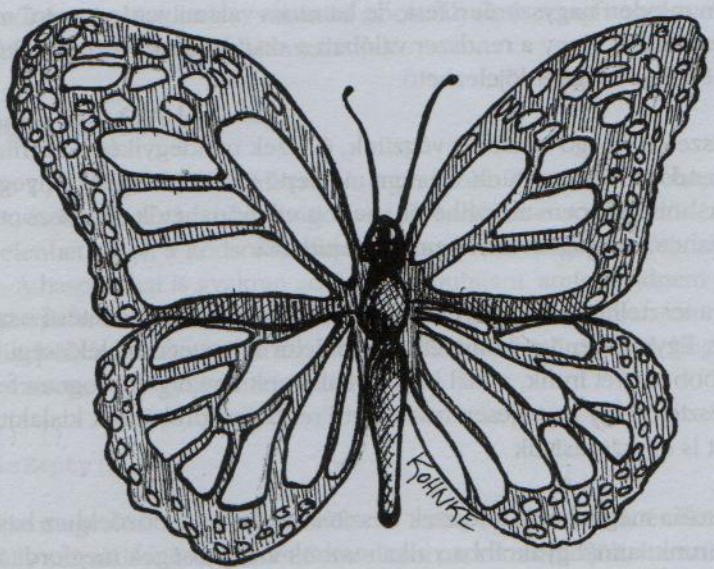
12

Láthatóság

Jeff Langr



Látható szerkezet – világos rendszer

Mi lenne, ha fel tudnánk mutatni négy egyszerű szabályt, amelyeknek a betartása biztosítaná a programjaink helyes szerkezetét? Mi lenne, ha e szabályok követésével világossá válna számunkra a kódunk kis- és nagyléptékű szerkezete, ami jelentősen megkönnyítené bizonyos elvek, mint az SRP vagy a DIP alkalmazását? Négy szabály, amely meghatározná, hogyan tegyük *jól láthatóvá* egy rendszer szerkezetét.

Nos, Kent Becknek az *egyszerű felépítésre* (Simple Design[1]) vonatkozó szabályait sokan tartjuk nagy segítségnek a jóltervezett programok készítéséhez. Kent szerint egy rendszer akkor elégíti ki az „egyszerűség" feltételeit, ha eleget tesz az alábbiaknak:

- Magában foglalja és kielégíti a tesztjeit.
- Nem tartalmaz felesleges ismétlődést.
- Kifejezi a programozó szándékait.
- A lehető legkisebbre csökkenti az osztályok és tagfüggvények számát.

A szabályokat a fontosságuk sorrendjében ismertetjük.

Az egyszerű felépítés 1. szabálya: a tesztek létfontosságúak

A rendszerünknek mindenekelőtt úgy kell működnie, ahogy azt mi szeretnénk. Lehet, hogy papíron minden nagyszerűen fest, de ha nincs valamilyen egyszerű módszerünk arra, hogy ellenőrizzük, hogy a rendszer valóban a szándékaink szerint működik-e, az erőfeszítéseink értelme megkérdőjelezhető.

Ha egy rendszeren átfogó teszteket végzünk, és ezek mindegyikének bármikor megfelel, tesztelhető rendszerről beszélünk. Ez nem meglepő állítás, de annál lényegesebb. Azok a rendszerek, amelyek nem tesztelhetők, nem is ellenőrizhetők. Ha viszont egy rendszer nem ellenőrizhető, természetesen nem is telepíthető.

Szerencsére a tesztelhetőség igénye az apró és egyértelmű rendeltetésű osztályok kialakítása felé visz. Egyszerűen így könnyebb megfelelni az egyetlen felelősségi kör (SRP) elvének. Minél több tesztet írunk, annál inkább haladunk egy olyan program felé, amelyet könnyebb tesztelni. Így ha teljesen tesztelhető rendszert próbálunk kialakítani, egyúttal a szerkezetét is egyszerűsítjük.

A szoros csatolás megnehezíti a tesztek készítését, ezért, az előzőekhez hasonlóan, minél több tesztet írunk, annál gyakrabban alkalmazzuk a függőségek megfordításához (DIP) hasonló elveket, illetve az olyan eszközöket, mint a függőség-befecskendezés, a felületek és az elvont ábrázolás. Ezzel a lehető legkisebb mértékűre csökkentjük a csatolást, és a rendszerünk szerkezete tovább javul.

Figyelemre méltó, hogy egyetlen egyszerű szabály követésével – miszerint tesztekre van szükségünk, és ezeket folyamatosan futtatnunk kell – milyen erővel mozdul el a rendszer az objektumközpontú programozás olyan alapvető céljai felé, mint a csatolás mértékének csökkentése és az erős összetartás. A tesztek írása tehát jobb programszerkezetet eredményez.

[1] [XPE].

Az egyszerű tervezés 2–4. szabálya: újratervezés

Ha vannak tesztjeink, ez biztosítja a motivációt a kód és az osztályok tisztán tartására. Mindezt egymásra épülő újratervezési lépésekkel valósítjuk meg. Beillesztünk néhány sornyi kódot, majd megállunk, és áttekintjük az új szerkezetet. Rontottunk a helyzeten? Ha igen, rendet rakunk, és futtatjuk a teszteket, hogy meggyőződjünk róla, hogy semmit nem tettünk tönkre. *Annak tudatában, hogy rendelkezünk tesztekkel, a legkevésbé sem kell félnünk attól, hogy a megtisztítása működésképtelenné teszi a kódot.*

Az újratervezés során a helyes programtervezést segítő fegyvertárunk minden elemét bevethetjük. Növelhetjük az összetartást, csökkenthetjük a csatolást, szétválaszthatjuk a felelősségi köröket, modulárissá tehetjük a rendszerműveleteket, zsugoríthatjuk a függvényeinket és osztályainkat, választhatunk jobb neveket, és így tovább. Most jön el az a pillanat is, amikor az egyszerű tervezés három alapszabályát – a felesleges ismétlések kiküszöbölése, a kifejezőkészség biztosítása, valamint az osztályok és a tagfüggvények számának lehető legkisebbre csökkentése – alkalmazhatjuk.

Nincs felesleges ismétlődés

Az ismétlődés a jól megtervezett rendszerek legnagyobb ellensége. Többletmunkát, többletkockázatot, és felesleges bonyolultságot visz a rendszerbe. Az ismétlődés számtalan formában megjelenhet. Azok a kódsorok, amelyek pontosan megegyeznek, nyilvánvalóan ismétlődőek. A hasonlókat is gyakran addig lehet puhítani, amíg majdnem egyformává válnak, így az újratervezésük is könnyebb. Az ismétlődésnek azonban vannak alattomosabb formái is, például az ismételt megvalósítás. Elképzelhető például, hogy az alábbi két tagfüggvény szerepel egy gyűjteményosztályban:

```
int size() {}
boolean isEmpty() {}
```

Írhatunk külön megvalósítást mindkettőhöz: az `isEmpty` egy logikai értéket vizsgálhat, míg a `size` egy számlálót. A kódismétlődést azonban elkerülhetjük, ha az `isEmpty` meghatározását a `size` segítségével valósítjuk meg:

```
boolean isEmpty() {
   return 0 == size();
}
```

A tiszta rendszer eléréséhez mindenféle ismétlődést ki kell küszöbölnünk, még ha csak pár sornyi kódról is van szó. Vegyük például a következőt:

```
public void scaleToOneDimension(
        float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) <
                                            errorThreshold)
```

```
            return;
        float scalingFactor = desiredDimension / imageDimension;
         scalingFactor =
                        (float)(Math.floor(scalingFactor * 100) * 0.01f);
        RenderedOp newImage = ImageUtilities.getScaledImage(
            image, scalingFactor, scalingFactor);
         image.dispose();
         System.gc();
         image = newImage;
      }
    public synchronized void rotate(int degrees) {
        RenderedOp newImage = ImageUtilities.getRotatedImage(
            image, degrees);
         image.dispose();
         System.gc();
         image = newImage;
      }
```

Ahhoz, hogy ezt a rendszert tisztán tartsuk, el kell távolítanunk a `scaleToDimension` és a `rotate` tagfüggvények közötti leheletnyi ismétlődést:

```
    public void scaleToOneDimension(
            float desiredDimension, float imageDimension) {
        if (Math.abs(desiredDimension - imageDimension) <
                                                    errorThreshold)
            return;
        float scalingFactor = desiredDimension / imageDimension;
        scalingFactor =
                        (float)(Math.floor(scalingFactor * 100) * 0.01f);
        replaceImage(ImageUtilities.getScaledImage(
            image, scalingFactor, scalingFactor));
    }
    public synchronized void rotate(int degrees) {
        replaceImage(ImageUtilities.getRotatedImage(image, degrees));
    }
    private void replaceImage(RenderedOp newImage) {
        image.dispose();
        System.gc();
        image = newImage;
    }
```

Ahogy kiemeljük a közös részeket ezen az igen alacsony szinten, kezdjük felfedezni, hol sértettük meg az SRP elvét. Az újonnan kiemelt tagfüggvényt tehát áthelyezhetjük egy másik osztályba, ami láthatóbbá teszi. A fejlesztőcsapatunk egy másik tagja ezt követően megláthatja benne a további elvonatkoztatás lehetőségét, és felhasználhatja más környezetben is. Ez az „újrahasznosítás kicsiben" jelentősen csökkentheti a rendszer bonyolultságát. Ahhoz, hogy az újrahasznosítást „nagyban" sikerrel űzzük, meg kell tanulnunk, hogyan végezhetjük el „kicsiben" is.

A *sablonfüggvény használata*[2] minta alkalmazása gyakori módszer a magasabb szintű ismétlődések eltávolítására. Vegyük például az alábbi kódot:

```
public class VacationPolicy {
    public void accrueUSDivisionVacation() {
      // a szabadság idejét a ledolgozott órák alapján kiszámító kód
      // ...
      // kód, amely biztosítja, hogy a szabadság ideje elérje
      // az amerikai minimumot
      // ...
      // kód, amellyel a szabadságot felvehetjük a fizetési adatok közé
      // ...
    }
    public void accrueEUDivisionVacation() {
      // a szabadság idejét a ledolgozott órák alapján kiszámító kód
      // ...
      // kód, amely biztosítja, hogy a szabadság ideje elérje
      // az európai minimumot
      // ...
      // kód, amellyel a szabadságot felvehetjük a fizetési adatok közé
      // ...
    }
}
```

Az `accrueUSDivisionVacation` és az `accrueEuropeanDivisionVacation` kódja nagyjából megegyezik, eltekintve a törvényi minimum megállapításától – ez a dolgozó típusától függően változik.

A nyilvánvaló ismétlődést a *sablonfüggvény használata* mintával küszöböljük ki:

```
abstract public class VacationPolicy {
    public void accrueVacation() {
      calculateBaseVacationHours();
      alterForLegalMinimums();
      applyToPayroll();
    }
    private void calculateBaseVacationHours() { /* ... */ };
    abstract protected void alterForLegalMinimums();
    private void applyToPayroll() { /* ... */ };
}
public class USVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
      // amerikai dolgozókra vonatkozó kód
    }
}
```

[2] [GOF].

```
public class EUVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // európai dolgozókra vonatkozó kód
    }
}
```

Az alosztályok az `accrueVacation` algoritmusában tátongó „lyukat" töltik be, csak azokat az adatokat átadva, amelyek nem ismétlődnek.

Kifejezőkészség

A legtöbbünk találkozott már kusza kóddal, sőt magunk is készítettünk már ilyet. Olyan kódot könnyű írni, amit *mi* megértünk, hiszen a megírásakor mélyen beleástuk magunkat a megoldandó feladatba. Akikre azonban később a kód karbantartásának feladata hárul, sokkal távolabbról tekintenek a kódra.

Egy program fejlesztési költségeinek túlnyomó részét a hosszú távú fenntartás teszi ki. Ahhoz, hogy a változások nyomán megjelenő hibalehetőségeket a lehető legnagyobb mértékben kizárjuk, fontos, hogy pontosan megértsük, hogyan is működik a rendszer. Ahogy egyre összetettebb lesz, a megértése egyre hosszabb időt vesz igénybe, és ezzel párhuzamosan a félreértés esélye is nő. Fontos hát, hogy a kód megfelelően kifejezze a szerzője szándékait. Minél tisztábbá tesszük a kódot, a többieknek annál kevesebb erőfeszítésükbe kerül megérteni, így kisebb az esély a hibák bekövetkezésére, és a karbantartás költségei is csökkennek.

A kifejezőkészséghez nagyban hozzájárul, ha megfelelően választjuk meg a programban alkalmazott neveket. Azt szeretnénk, hogy ha meghalljuk egy osztály vagy függvény nevét, ne lepődjünk meg, amikor kiderül, hogy milyen feladatot valósít meg. A jó kifejezőkészséghez tartozik az is, ha a függvények és osztályok méretét a lehető legkisebbre csökkentjük. A rövid függvényeket és osztályokat ugyanis rendszerint könnyű elnevezni, megírni és megérteni.

Egy szabványos elnevezési rendszer alkalmazásával is közelebb kerülhetünk a céljainkhoz. A tervezési minták például nagyrészt a szándékaink közlését és a kifejezőkészséget szolgálják. Ha az osztályok nevében a bennük alkalmazott szabványos minták neveit használjuk – amilyen a *parancs* (command) vagy a *látogató* (visitor) –, azzal tömören leírhatjuk a program szerkezeti elemeit más fejlesztők számára.

A jól megírt egységtesztek szintén meglehetősen kifejezőek. Elsődleges feladatuk a program leírása példák segítségével. Aki megtekinti a tesztjeinket, hamar átláthatja, milyen célokat is szolgál az adott osztály.

A kifejezőkészség legfontosabb előremozdítója azonban mindenekfelett a *kitartó próbálkozás*. Túl gyakori, hogy egy feladat tűrhető megoldásának birtokában máris továbbszáguldunk a következő feladathoz, anélkül, hogy végiggondolnánk a kódot az olvashatóság szempontjából is. Ne feledjük: a kódunk következő olvasója is valószínűleg mi leszünk.

Próbáljunk úgy dolgozni, hogy büszkék lehessünk az eredményre. Szenteljünk némi időt a függvényeinknek és osztályainknak. Válasszunk jobb neveket, osszuk fel a nagyobb függvényeket kisebbekre, röviden: programozzunk minél nagyobb műgonddal. Ez az odafigyelés a legfontosabb erőforrásunk.

Kevés osztály és tagfüggvény

Még az olyan alapvető szabályok alkalmazását is túlzásba lehet vinni, mint az ismétlődések megszüntetése, a kifejezőkészség biztosítása, vagy az egyetlen felelősségi kör elve. Miközben kicsi osztályokat és tagfüggvényeket próbálunk írni, lassan eljuthatunk odáig, hogy egyszer csak túl sok lesz belőlük. Ezt a parttalan burjánzást hivatott megállítani ez a szabály.

Az osztályok és tagfüggvények nagy száma gyakran az értelmetlen szabálykövetés eredménye. Gondoljunk például egy olyan kódolási szabályra, ami szerint minden osztályhoz felületet kell létrehoznunk, vagy azokra a fejlesztőkre, akik úgy vélik, hogy a mezőket és a viselkedést minden körülmények között külön adat- és viselkedésosztályokra kell szétválasztani. Az ilyen merev szabályokat jobb, ha elutasítjuk, és sokkal gyakorlatiasabban állunk a kérdéshez.

A cél az, hogy a teljes rendszer mérete kicsi maradjon, ugyanakkor a függvények és osztályok mérete se növekedjen. Ne feledjük azonban, hogy ez a szabály leghátul áll az *egyszerű felépítés* négyesének rangsorában. Vagyis, jóllehet a függvények és osztályok számának alacsonyan tartása fontos, a tesztek készítése, az ismétlődések megszüntetése és a kifejezőkészség biztosítása előrébb valók.

Összefoglalás

Vajon néhány egyszerű gyakorlati módszer elsajátítása helyettesítheti évek tapasztalatát? Semmiképpen sem. Másrészről azonban az itt és a könyv többi fejezetében ismertetett eljárások a szerzők több évtizednyi tapasztalatán alapulnak. Ha követjük az egyszerű felépítés szabályait, könnyebben és bátrabban alkalmazzuk azokat a helyes megoldásokat és mintákat, amelyeknek az elsajátítása egyébként éveket venne igénybe.

Irodalomjegyzék

[XPE]: Kent Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[GOF]: Gamma és mások: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1996. (magyarul: *Programtervezési minták – Újrahasznosítható elemek objektumközpontú programokhoz*, Kiskapu, 2004).