6

Objektumok és adatszerkezetek

Oka van annak, hogy privát változókat használunk. Nem akarjuk ugyanis, hogy bárki is függjön tőlük. Szeretnénk megőrizni a szabadságunkat, hogy bármikor megváltoztathassuk a típusukat vagy a megvalósításukat, amikor csak kedvünk szottyan rá. Nos, ha ez ilyen világos, miért látunk mégis annyi beállító és kiolvasó tagfüggvénnyel elért privát változót úton-útfélen, mintha csak nyilvánosak lennének?

Elvont adatábrázolás

Figyeljük meg a 6.1. és a 6.2. példa közötti különbséget! Mindkettő egy Descartes-koordinátarendszerbeli pont adatait tárolja, az egyik mégis teljesen felfedi a megvalósítást, a másik pedig tökéletesen elrejti.

6.1. példa

A Point konkrét megvalósítása

```
public class Point {
  public double x;
  public double y;
}
```

6.2. példa

A Point elvont megvalósítása

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

A 6.2. példa valódi szépsége abban rejlik, hogy nem tudjuk eldönteni, hogy derékszögű vagy polárkoordinátákat alkalmaz – vagy egyiket sem. Ugyanakkor a felület emellett egyértelműen megjeleníti az adatszerkezetet.

A kód azonban ez esetben több egyszerű adatszerkezetnél. A tagfüggvények ugyanis egy elérési eljárást kényszerítenek ki. Az egyes koordinátákat külön kiolvashatjuk, a beállításukra azonban csak egyszerre, egy elemi műveletben van lehetőségünk.

A 6.1. példában ugyanakkor egyértelműen derékszögű koordinátákat használunk, amelyeket külön-külön kell kezelnünk. Mindez felfedi az adatszerkezet megvalósítását – még akkor is ezt tenné, ha privát változókat és a hozzájuk tartozó beállító és kiolvasó tagfüggvényeket használnánk.

A megvalósítás elrejtése nem egyszerűen abban áll, hogy egy függvényréteget helyezünk el a változók és a felhasználó között – sokkal inkább az elvont ábrázolásról van szó. Egy lépést sem jutunk előbbre, ha az osztályaink változóit kiolvasó és beállító tagfüggvényekkel tesszük elérhetővé. Ehelyett inkább elvont felületeket kell biztosítanunk, amelyekkel az adatok *lényegéhez* férhetünk hozzá, anélkül, hogy tudnunk kellene a megvalósításról.

Nézzük meg most a 6.3. és 6.4. példát. Az első konkrét egységekben fejezi ki egy jármű üzemanyagszintjét, míg a másik elvont százalékos értékekkel él. Az első esetben biztosak lehetünk benne, hogy itt mindössze a változók elérőit kapjuk, a második esetben azonban fogalmunk sincs róla, hogy az osztály pontosan milyen formában tárolja az adatokat.

6.3. példa

A Vehicle konkrét megvalósítása

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

6.4. példa

A Vehicle elvont megvalósítása

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

Mindkét esetben a második megoldást érdemes választanunk, nem szeretnénk ugyanis az adataink részleteit nyilvánosságra hozni – sokkal jobban tesszük, ha az adatokat elvont formában jelenítjük meg. Ezt pedig nem lehet egyszerűen annyival elintézni, hogy beállító és kiolvasó tagfüggvényeket hozunk létre. Komolyan el kell gondolkodnunk azon, hogy milyen módon jelenítsük meg az objektumban tárolt adatokat. A szemellenzősen alkalmazott elérőfüggvények a lehető legrosszabb megoldást jelentik.

Adatszerkezetek és objektumok – az erő két oldala

A fenti két példa nagyszerűen mutatja az objektumok és az adatszerkezetek közötti különbséget. Az objektumok egy elvont alak mögé rejtik az adataikat, és csak függvényeket tesznek nyilvánossá, amelyek ezeket az adatokat elérik. Az adatszerkezetek ugyanakkor teljes egészében felfedik az adataikat, és nincsenek adatkezelő függvényeik. Olvassuk csak el még egyszer az előző mondatokat! Figyeljük meg, hogy a két meghatározás szinte pontos ellentéte egymásnak. A különbség persze nyilvánvalónak tűnhet, de távolba mutató következményekkel jár.

Vegyük az alakzatok kezelésének a 6.5. példában látható eljárásközpontú megvalósítását. A `Geometry` osztály három alakzatosztályt képes kezelni. Ez utóbbiak egyszerű adatosztályok, amelyek nélkülöznek mindenféle viselkedést – az összes művelet a `Geometry` osztály hatáskörébe tartozik.

Az objektumközpontú programozók joggal húzzák fel a szemöldöküket, és jegyzik meg, hogy „De hiszen ez a kód eljárásközpontú!". Nos, igazuk lehet – bár némi mentségünk azért van... Képzeljük el, mi történne, ha a `Geometry` osztályt egy `perimeter()` (kerület) nevű függvénnyel bővítenénk. Az alakzatosztályokat ez egyáltalán nem érintené, más részről viszont, ha új alakzatot hozunk létre, a `Geometry` osztály összes függvényét át kellene írnunk ahhoz, hogy képesek legyenek kezelni. Olvassuk csak el ismét a leírtakat – megint két, szinte pontosan ellentétes állítást kaptunk.

6.5. példa

Alakzatok eljárásközpontú kezelése

```
public class Square {
  public Point topLeft;
  public double side;
}
public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}
public class Circle {
  public Point center;
  public double radius;
}
public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Figyeljük meg a 6.6. példában bemutatott objektumközpontú megoldást. Az új, `area()` (terület) nevű tagfüggvény többalakú. Itt nincs szükség `Geometry` osztályra, ha tehát új alakzatot illesztünk be, egyik már meglevő *függvényt* sem kell módosítanunk. Ugyanakkor új függvény bevezetése esetén minden *alakzat* kódját át kell írnunk.[1]

[1] Léteznek módszerek ennek az elkerülésére, amelyeket a tapasztaltabb objektumközpontú programozók jól ismernek – ilyen például a Látogató minta (kettős kézbesítés) használata. Ezeknek azonban egytől-egyig megvannak a maguk költségei, és végeredményben visszavezetnek az eljárásközpontú programok szerkezetéhez.

6.6. példa

Alakzatok többalakú megvalósítása

```
public class Square implements Shape {
  private Point topLeft;
  private double side;
  public double area() {
    return side*side;
  }
}
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;
  public double area() {
    return height * width;
  }
}
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;
  public double area() {
    return PI * radius * radius;
  }
}
```

Ismét láthatjuk, hogy a két meghatározás szinte egymás ellentettje. Mindez rávilágít az objektumok és az adatszerkezetek közötti alapvető ellentétre:

> *Egy eljárásközpontú kódban (ahol adatszerkezeteket használunk) könnyen elhelyezhetünk új függvényeket anélkül, hogy a már létező adatszerkezetekhez hozzá kellene nyúlnunk. Az objektumközpontú kódokban ugyanakkor az új osztályok beillesztése egyszerű, anélkül, hogy a függvényeket kellene módosítanunk.*

Ez visszafelé is igaz:

> *Egy eljárásközpontú kódban nehéz új adatszerkezeteket használatba venni, mivel ehhez minden függvényt meg kell változtatnunk. Az objektumközpontú kódokban ugyanakkor a függvények beillesztése nehéz, mivel ehhez minden osztályt módosítanunk kell.*

Tehát ami az objektumok számára könnyű, az az eljárásoknak nehéz, ami pedig az eljárásoknak könnyű, az az objektumoknak esik nehezére.

Egy összetett rendszerben mindenképpen előfordul, hogy inkább új adattípusokat vezetnénk be új függvények helyett – ilyenkor az objektumközpontú programozás adja az alkalmasabb megoldást. Másrészről, találkozunk majd olyan helyzettel is, amikor az új függvények felé hajlanánk ahelyett, hogy új adattípusokat találnánk ki. Ilyen esetben az eljárásközpontú kód és az adatszerkezetek lesznek a segítségünkre.

Az érettebb gondolkodású programozók jól tudják, hogy egyáltalán nem „minden dolog objektum". Esetenként igenis egyszerű adatszerkezetekre és az azokat kezelő eljárásokra van szükség.

Demeter törvénye

Az objektumközpontú programozás világában ismeretes egy szabály, ami azt mondja ki, hogy egy modulnak nem szabad tudomást vennie az általa kezelt *objektum* belső szerkezetéről. Ez *Demeter törvénye*.[2] Amint az előzőekben láthattuk, az objektumok elrejtik az adataikat, és nyilvánosságra hozzák a műveleteiket. Ez azt jelenti, hogy nem ajánlatos elérőkkel kiteregetnünk ezt a belső szerkezetet, hiszen ez ellentétes lenne az objektumok alaptermészetével.

Demeter törvénye egészen pontosan azt mondja ki, hogy *C* osztály *f* tagfüggvénye csak a következőkhöz tartozó tagfüggvényeket hívhat meg:

- *C*
- Az *f* által létrehozott objektumok
- Az *f*-nek paraméterként átadott objektumok
- A *C* példányváltozóiban tárolt objektumok

A tagfüggvény ugyanakkor *nem* hívhatja meg olyan objektumok tagfüggvényeit, amelyeket a meghívható függvények adtak vissza. Vagyis: csak a barátokkal álljunk szóba, idegenekkel soha!

Az alábbi kód[3] Demeter törvényébe ütközik, mivel a `getOptions()` visszatérési értékének `getScratchDir()` tagfüggvényét, valamint a `getScratchDir()`-től visszakapott objektum `getAbsolutePath()` tagfüggvényét hívja meg.

```
final String outputDir =
                ctxt.getOptions().getScratchDir().getAbsolutePath();
```

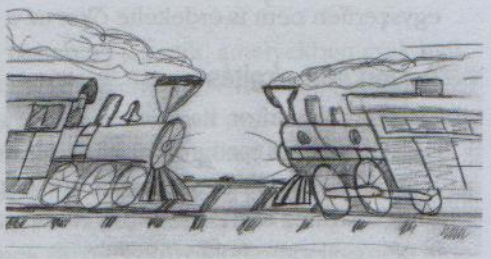
[2] `http://en.wikipedia.org/wiki/Law_of_Demeter` (magyarul `http://hu.wikipedia.org/wiki/Demeter_t%C3%B6rv%C3%A9nye`).

[3] *A kódot valahol az Apache keretrendszerében találtam.*

Vonatroncsok

Az ilyesfajta kódot csak *vonatroncsnak* szoktam nevezni, mivel úgy fest, mint egy sornyi egymás után kötött vasúti kocsi. A hívási láncok használata általában stílusbeli hiányosságnak minősül, ezért kerülendő [G36]. A legjobb, ha szétvágjuk a láncot:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```



De vajon ez a két kódrészlet megsérti-e Demeter törvényét? Nos, a kódot befogadó modul nyilvánvalóan tudja, hogy a `ctxt` objektum beállításokat (options) tárol, amelyek ideiglenes könyvtárakat (scratch directory) adnak meg, amelyekhez viszont tartozik egy abszolút elérési út (absolute path). Nos, egy függvény számára ez talán túlzottan is nagy tudás. A hívó függvény ezek szerint tudatában van annak, hogy miként haladjon végig az objektumok során.

Arról, hogy megsértettük-e Demeter törvényét, csak akkor nyilatkozhatunk, ha tudjuk, hogy a `ctxt`, az `Options` és a `ScratchDir` adatszerkezetek vagy objektumok. Utóbbi esetben a belső szerkezetüket el kell rejtenünk, így ez a tudás egyértelműen ellentmond Demeter törvényének. Ha azonban mindhárom esetben viselkedéssel nem rendelkező adatszerkezetekről van szó, akkor természetes, hogy felfedik a belső felépítésüket, így nem sértik meg a törvényt.

Az elérőfüggvények használata kissé bonyolítja a helyzetet. Ha a kódot az alábbi formában írtuk volna, jó eséllyel el sem gondolkodunk Demeter törvényének sérüléséről:

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

A gondjainkat jelentősen egyszerűsítené, ha az adatszerkezetekben egyszerűen nyilvános változók állnának, függvényeik pedig egyáltalán nem lennének, az objektumok ugyanakkor csak privát változókat és nyilvános függvényeket tartalmaznának. Mindazonáltal, egyes környezetek és szabványok (mint a „babszemek") megkövetelik, hogy a legegyszerűbb adatszerkezeteknél is alkalmazzunk elérőket és módosítókat.

Kevert adatszerkezetek

Ez a zavaros helyzet olykor furcsa, kevert (hibrid) adatszerkezetekhez vezet, amelyek egyaránt magukon viselik az objektumok és az adatszerkezetek jellemzőit. Egyrészről rendelkeznek komoly műveletek elvégzésére alkalmas függvényekkel, másrészről helyet ad-

nak a nyilvános változóknak vagy nyilvános elérőknek és módosítóknak, amelyek végeredményben nyilvánossá teszik a privát változókat, arra csábítva ezzel más külső függvényeket, hogy használják is ezeket – úgy, ahogy azt egy eljárásközpontú program tenné egy adatszerkezettel.[4]

Az ilyen hibridek egyaránt megnehezítik az új függvények és az új adatszerkezetek bevezetését, tehát a két világ legrosszabb tulajdonságait egyesítik – kerüljük el őket messzire! Megjelenésük az átgondolatlan tervezés jele, ahol a szerzők nem voltak biztosak benne, hol kell megvédeniük a függvényeket és az adattípusokat, vagy – ami még rosszabb – egyszerűen nem is érdekelte őket ez a kérdés.

A szerkezet elrejtése

Mi a helyzet akkor, ha a `ctxt`, az `options` és a `scratchDir` objektumok, valódi viselkedéssel? Nos, ez esetben, mivel az objektumok jellemzője a belső szerkezetük elrejtése, nem engedhetjük meg, hogy a kód végighaladjon rajtuk. Rendben, de akkor hogyan jutunk hozzá az ideiglenes könyvtár abszolút elérési útjához? Így?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

Vagy így?

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

Az első lehetőség választása esetén igencsak megszaporodhatnak a `ctxt` objektum tagfüggvényei. A második alak viszont azt feltételezi, hogy a `getScratchDirectoryOption()` egy adatszerkezetet ad vissza, nem pedig egy objektumot. Egyik eset sem igazán kedves a szívünknek.

Ha a `ctxt` egy objektum, akkor arra kell kérnünk, hogy tegyen valamit, nem pedig a belső felépítése felől érdeklődnünk. Miért is volt szükségünk az ideiglenes könyvtár abszolút elérési útjára? Mit szerettünk volna vele kezdeni? Nézzük meg az alábbi kódot ugyanebben a modulban, jó pár sorral lejjebb:

```
String outFile = outputDir + "/" + className.replace('.', '/') +
".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

A különböző szintek keverése [G34][G6] némiképp zavaró. Pontok, perjelek, fájlkiterjesztések és `File` objektumok zagyvaléka a köré szövődő kóddal. Ha azonban erről egy pillanatra megfeledkezünk, láthatjuk, hogy azért van szükségünk az ideiglenes könyvtár abszolút elérési útjára, hogy létrehozzunk egy megadott nevű ideiglenes fájlt.

[4] Ezt a jelenséget olykor a „szolgáltatások elirigylésének" (Feature Envy) is nevezik, lásd [Refactoring].

Mi lenne, ha erre közvetlenül a `ctxt` objektumot kérnénk meg?

```
BufferedOutputStream bos =
                    ctxt.createScratchFileStream(classFileName);
```

Nos, valami ilyesmit kell tennie egy objektumnak! Így a `ctxt` elrejtheti a belső felépítését, és a függvényünk nem kényszerül arra, hogy megsértse Demeter törvényét olyan objektumok használatával, amelyeknek a létezéséről sem kellene tudnia.

Adatátviteli objektumok

Az adatszerkezetek legtisztább formáját az olyan osztályok jelentik, amelyekben csak nyilvános változók vannak, függvények pedig egyáltalán nincsenek. Ezeknek a példányait gyakran adatátviteli objektumoknak nevezik. Nagy szerepet kapnak az adatbázisokkal folytatott adatcserében, a csatolók üzeneteinek értelmezésében és rengeteg hasonló helyzetben. Gyakori, hogy első átmeneti állapotként tűnnek fel az adatbázisok nyers adataitól a kódbeli objektumokig vezető úton. Némiképp gyakoribb a „babszem" (bean) alak, amelyre a 6.7. kódpélda mutat példát. A babszemek privát változókkal rendelkeznek, amelyeket beállító és kiolvasó tagfüggvényekkel érhetünk el. Ez az egységbe zárási álca pár objektumközpontú megszállottat talán kielégít, de nem sokkal viszi előbbre az ügyünket.

6.7. példa

address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;
  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }
  public String getStreet() {
    return street;
  }
  public String getStreetExtra() {
    return streetExtra;
  }
  public String getCity() {
    return city;
  }
```

folytatódik

6.7. példa

address.java – folytatás

```
    public String getState() {
      return state;
    }
    public String getZip() {
      return zip;
    }
  }
```

Aktív rekordok

Az aktív rekordok az adatátviteli objektumok különleges alakjának tekinthetők. Adatszerkezetek nyilvános (vagy elérőfüggvényekkel kezelhető) változókkal, amelyek jellemzően tartalmaznak bizonyos navigációs függvényeket, mint a `save` vagy a `find`. Az aktív rekordok többnyire adatbázisok táblái vagy egyéb adatforrások közvetlen átiratai.

Sajnos túl gyakran fordul elő, hogy a fejlesztők objektumként kezelik ezeket az adatszerkezeteket, a működési szabályokat érvényesítő tagfüggvényeket helyezve el bennük. Ez azonban nem helyénvaló, hiszen így az adatszerkezet és az objektum közötti keveréket kapunk. A megoldás természetesen az, hogy az aktív rekordokat adatszerkezetként kezeljük, és külön objektumokat hozunk létre a működési szabályok tárolására, amelyek elrejtik a belső adataikat (ezek jobbára az aktív rekordok példányai lesznek).

Összefoglalás

Az objektumok büszkén mutatják a viselkedésüket, de elrejtik az adataikat. Ennek köszönhető, hogy könnyen bevezethetünk új objektumokat anélkül, hogy a már létező műveleteket megváltoztatnánk. Az adatszerkezetek ugyanakkor megmutatják az adataikat, és nem rendelkeznek lényeges viselkedéssel. Így a meglevő adatszerkezetekhez könnyen rendelhetünk új műveleteket, de a már létező függvényeinkkel nehezebb elfogadtatni az új adatszerkezeteket.

A rendszerünk olykor megköveteli, hogy készen álljunk új adattípusok bevezetésére – így itt az objektumok használatára érdemes törekednünk. Máskor viszont új viselkedésre lehet szükségünk, így a rendszer egyes részein jobb szolgálatot tehetnek az adattípusok és az eljárások. A jó programozó képes előítélet nélkül felmérni a helyzetet, és mindig a körülményeknek megfelelő eszközt választja.

Irodalomjegyzék

[Refactoring]: Martin Fowler és mások: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999. (magyarul: *Refactoring – Kódjavítás újratervezéssel*, Kiskapu, 2006).