2

Beszédes nevek

Tim Ottinger

Bevezetés

Ahol program van, ott nevek is vannak. Neveket adunk a változóinknak, a függvényeinknek, a paramétereinknek, az osztályainknak és a csomagjainknak. El kell neveznünk a forrásfájlokat és az azokat tartalmazó könyvtárakat. A `jar` fájlokat, a `war` fájlokat, az `ear` fájlokat… Nevek, nevek, nevek, és újra: nevek. Annyiszor adunk nevet, hogy nem árt, ha beleviszünk némi szakértelmet ebbe a műveletbe. Ebben segíthet az alábbiakban bemutatott néhány egyszerű szabály.

Használjunk olyan neveket, amelyek felfedik a szándékainkat!

Mondani könnyű, de valamiképpen meg szeretnénk mutatni, hogy valóban komolyan így gondoljuk. A jó nevek megválasztása bizony időt vesz el, de később ennél jóval több időt nyerünk vele. Így hát járjunk el a lehető legnagyobb gondossággal a névválasztásban, és ha találunk egy névnél jobbat, ne habozzunk megváltoztatni! A kódunk minden olvasója (beleértve saját magunkat is) hálás lesz ezért a figyelmességért.

Legyen szó változó, függvény vagy osztály nevéről, önmagában meg kell válaszolnia a fontosabb kérdéseket – mi az oka a létezésének, milyen feladatot végez, és hogyan használják. Következésképpen, ha egy név megadásakor úgy érezzük, hogy megjegyzést kell írnunk hozzá, bizonyára rosszul választottuk meg.

```
int d; // az eltelt idő napban
```

A `d` név semmit sem árul el. Nem utal arra, hogy eltelt időről van szó, és arra sem, hogy ezt az időt napban mérjük. Olyan nevet kellene választanunk, ami meghatározza, hogy mit is mérünk, és milyen egységben:

```
int elteltIdoNapban
int LetrehozasOtaElteltNapok
int ModositasOtaElteltNapok
int FajlKoraNapban
```

A szándékainkat felfedő nevek révén a kód megértése és módosítása sokkal egyszerűbbé válik. Mi a rendeltetése például az alábbi kódnak?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
     if (x[0] == 4)
       list1.add(x);
  return list1;
}
```

Miért olyan nehéz kitalálni, hogy mit is csinál ez a kódrészlet? Nincsenek összetett kifejezések, a térközök és a behúzások jónak tűnnek, ráadásul mindössze három változó és két állandó szerepel a kódban. Még csak puccos osztályok vagy többalakú tagfüggvények sem jelennek meg – csak egyszerű tömbök listájáról van szó (legalábbis elsőre úgy tűnik).

A gond nem a kód egyszerűségével van, hanem azzal, amit elhallgat előlünk – burkoltan megköveteli ugyanis, hogy tudjuk a választ a következő kérdésekre:

1. Milyen elemek szerepelnek a `theList` listában?
2. Mi a jelentősége a `theList` nulladik elemének?
3. Miért éppen a 4 értékkel hasonlítjuk össze?
4. Hogyan használjuk a visszakapott listát?

A kérdésekre várt válaszok nincsenek ott a példakódban, de ott lehetnének. Tegyük fel, hogy egy aknakereső játékon dolgozunk. A játék táblája voltaképpen cellák listája – ez a `theList` („a lista"). Adjuk hát neki inkább a `gameBoard` (játéktábla) nevet.

A tábla celláit egy-egy tömb jeleníti meg. Ráébredünk arra is, hogy a nulladik elem egy állapotjelzőt ad meg, amelynek 4 értéke azt jelenti, hogy a játékos a mezőt megjelölte. Pusztán azzal jelentősen javíthatjuk a kód olvashatóságát, ha ezeket a jelentéseket a nevekben is jelöljük:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Figyeljük meg, hogy a kód egyszerűsége ezzel nem változik. Pontosan ugyanannyi művelet és állandó szerepel benne, mint korábban, és ugyanannyi beágyazási szintet is alkalmazunk. Az egyetlen különbség, hogy az új kód sokkal kifejezőbb.

Tovább is léphetünk, ha a cellák számára létrehozunk egy egyszerű osztályt ahelyett, hogy egészek tömbjeivel dolgoznánk. Bevezethetünk egy `isFlagged` (jelölve) nevű függvényt, ami jelzi a szándékainkat, és elfedi a „bűvös" számokat. Az eredeti függvényünk új változata ezzel így fest:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Néhány egyszerű névváltoztatás után többé nem okoz gondot megérteni, hogy mi is folyik a kódban. Ez a nevek ereje!

Kerüljük a félrevezetést!

Programozóként törekednünk kell arra, hogy ne hagyjunk félrevezető jelzéseket, amelyek elhomályosíthatják a kód jelentését. Mindenképpen kerüljük azokat a szavakat, amelyeknek a közkeletű jelentése eltér az általunk használttól. Így például a `hp`, az `aix` vagy az `sco` rosszul megválasztott nevek, hiszen ezek Unix-rendszerek, illetve -változatok nevei. Sőt, hiába tűnik nagyszerű elnevezésnek a `hp` egy derékszögű háromszög átfogójának (hypotenuse) számításakor, a használata így is félrevezető lehet.

Ha számlák csoportjára hivatkozunk, nem célszerű az `accountList` (számlalista) azonosítót használnunk, kivéve ha a csoport ténylegesen `List` típusú. A „list" szó többletjelentéssel bír a programozók számára, így ha a számlákat tároló egység nem `List` típusú, a név téves következtetésre adhat alapot.[1] Így hát az `accountGroup` (számlacsoport), a `bunchOfAccounts` (egy csomó számla) vagy az egyszerű `accounts` (számlák) név alkalmasabb lehet erre a szerepre.

Óvakodjunk az olyan nevektől, amelyek csak apróságokban térnek el egymástól. Próbáljuk csak ki, mennyi ideig tart, amíg észrevesszük a különbséget az egyik modul `XYZControllerForEfficientHandlingOfStrings` és egy másutt található `XYZControllerForEfficientStorageOfStrings` változó neve között. A hasonlóság túlságosan is meggyőző.

Ha hasonló dolgokat hasonlóképpen írunk, az tájékoztatás. Ha viszont nem vagyunk következetesek a nevek írásmódjában, az félrevezetés. Napjaink Java-fejlesztőkörnyezeteiben részesülhetünk az automatikus kódkiegészítés áldásaiból. Elég beírnunk egy név pár karakterét, leütnünk valamilyen billentyűkombinációt (vagy még azt sem), és máris hozzájutunk a név lehetséges alakjaihoz. Nagy segítség lehet, ha a hasonló dolgok nevei ábécésorrendben követik egymást, és a különbségek nyilvánvalóak közöttük, hiszen a fejlesztők előszeretettel választanak objektumokat pusztán a nevük alapján anélkül, hogy elolvasnák a hozzáfűzött megjegyzéseket, sőt gyakran anélkül, hogy áttekintenék a tagfüggvények listáját.

A legszörnyűbb félrevezető nevek talán a kis `l` és a nagy `O` használatából, illetve ezek kombinációiból erednek. A gondokat természetesen az okozza, hogy a képernyőn ezek a betűk úgy festenek, mint az `1` és a `0` számértékek:

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

[1] A későbbiekben megtanuljuk, hogy a `List` szócskát még abban az esetben sem érdemes rögzítenünk egy változó nevében, ha valóban `List` típusú tárolóról van szó.

Naivan azt gondolhatnánk, hogy ez nem más, mint merő kitalálmány, de a saját szememmel láttam olyan kódokat, ahol az ilyen nevek használata bevett szokás volt. Egyik esetben a szerző megoldásként javaslatot tett egy másik betűtípus használatára, ahol a különbségek szembetűnőbbek – ezt a megoldást azonban írásos formában vagy szájhagyomány útján minduntalan tovább kellene adnunk a fejlesztők későbbi generációinak. A megnyugtató és mellékhatások nélküli megoldást végül az egyszerű átnevezés adta.

Tegyünk valódi megkülönböztetéseket!

A programozók a saját életüket keserítik meg, amikor kizárólag azért döntenek egy kódolási forma mellett, hogy kielégítsék a fordítóprogram vagy az értelmező igényeit. Így például, ha észrevesszük, hogy azonos hatókörben nem használhatjuk ugyanazt a nevet két különböző dologra, kísértésbe eshetünk, hogy az egyiknek a nevét hasraütésszerűen változtassuk meg – például szándékosan vétett helyesírási hibákkal. Később igencsak meglepő lehet, hogy a nyilvánvaló helyesírási hibák kijavítása után a fordító nem képes elvégezni a munkáját.[2]

Nem elég, ha számokat vagy üres szavakat biggyesztünk az eredeti névhez, még ha ez a fordítót ki is elégíti. Ha a neveknek különbözniük kell, a jelentésük is legyen eltérő.

A számsorok használata (`a1`, `a2`, ... `aN`) teljesen szembemegy a szándékaink felfedését célzó elnevezési szabályunkkal. Ezek a nevek nem félretájékoztatnak – egyszerűen nem tájékoztatnak semmiről, vagyis semmilyen módon nem jelzik a szerző szándékait. Vegyük a következő példát:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

Ennek a függvénynek a kódja sokkal olvashatóbb lenne, ha a paramétereknek a `source` (forrás) és a `destination` (cél) neveket adnánk.

[2] Képzeljük el – ó, micsoda szörnyűség –, hogy valaki a `klass` nevet használja a `class` helyett, mert az utóbbi már foglalt.

A töltelékszavak hasonlóképpen alkalmatlanok a nevek valódi megkülönböztetésére. Képzeljük el, hogy van egy `Product` (termék) nevű osztályunk – ha emellett létrehoznánk egy `ProductInfo` vagy egy `ProductData` nevűt, egy centivel sem jutnánk előrébb. Az `Info` és a `Data` éppolyan semmitmondó töltelékszavak, mint a névelők.

Fontos megjegyeznünk, hogy semmi gond nincs azzal, ha előtagokat – mint az `a` vagy a `the` – használunk, amennyiben a szerepük világos. Így megtehetjük, hogy az `a` előtagot használjuk a helyi változókhoz, a `the` előtagot pedig a függvények paramétereinek tartjuk fenn.[3] Gond akkor van, ha azért használjuk a `theZork` nevet, mert már van egy `zork` nevű változónk.

A töltelékszavak feleslegesek. A `variable` (változó) szót soha ne használjuk változók nevében, a `table` szót pedig a tábláknál. Mennyivel jobb a `NameString` a `Name` névnél? Előfordulhat, hogy a `Name` értéke valaha is lebegőpontos lesz? Ha igen, megsérti a korábbi szabályunkat a félrevezetésről. Képzeljük csak el, hogy találunk egy `Customer` és egy `CustomerObject` osztályt. Vajon miben különböznek? Melyikkel érdemes hozzáférnünk az ügyfél fizetési előzményeihez?

Létezik egy alkalmazás, amelynek a kódja tökéletesen példázza a leírtakat. A neveket persze szemérmesen megváltoztattuk – legyen elég annyi, hogy ilyen alakú függvények szerepeltek benne:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Honnan kellene tudnia a programozónak, hogy melyik függvényt hívja meg a három közül?

Ha nincsenek konkrét elnevezési szabályok, a `moneyAmount` (pénzösszeg) nem mond többet a `money` (pénz) névnél, a `customerInfo` (ügyfélinfó) a `customer`-nél (ügyfél), az `accountData` (számlaadat) az `account`-nál (számla), vagy a `theMessage` (az üzenet) a `message` (üzenet) névnél. A tanulság az, hogy úgy kell megkülönböztetnünk meg az általunk alkalmazott neveket, hogy a különbségek jelentsenek is valamit a kód olvasója számára.

Alkalmazzunk könnyen kimondható neveket!

Az emberi agy nagyszerűen alkalmas a szavak kezelésére, ami nem meglepő, hiszen jelentős területe kifejezetten ezt a célt szolgálja. A szavak pedig kimondhatók. Így hát nagy hiba lenne, ha agyunknak ezt az evolúciós csúcsteljesítményét pihenni hagynánk, vagyis: használjunk könnyen kimondható neveket.

[3] Bob bácsi is így tett a C++-ban, de az új fejlesztőkörnyezetekben erre már nincs szükség.

Amit nem tudunk kimondani, arról csak igen esetlenül tudunk beszélni. „Nézd meg, ott a bé cé er három cé en tén van egy pé esz zé kú int, látod?" Ez igenis fontos szempont, a programozás ugyanis társas tevékenység.

Ismerek egy céget, ahol használatban van egy `genymdhms` (generation date, year, month, day, hour, minute, second – készítés dátuma, év, hónap, óra, perc és másodperc) nevű változó, amelyet egymás között betűzve emlegettek: „gen-í-em-dé-há-em-es". Én jobb szeretem közvetlenül a leírtakat kiejteni, így a „gen-ím-dö-höms" alakot használtam. Később a tervezők és elemzők egész serege kapott rá az én szójárásomra, de még így is igen furán hangzott. Persze, viccnek vettük, ugyanakkor „vétkeztünk", mivel eltűrtük egy alkalmatlan név használatát. Az új fejlesztőknek is el kell magyaráznunk, miért beszélünk úgy, mint valami idióták gyülekezete, ahelyett, hogy érthető szavakat használnánk, és kényszerűségből ők is átveszik a rossz szokásainkat. Hasonlítsuk össze az alábbi két kódrészletet:

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

és

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Így már lehetővé válik az emberi eszmecsere: „Hé, Miki, nézd ezt a bejegyzést! A készítés időbélyege (generation timestamp) a holnapi dátumot tartalmazza! Hogy lehet ez?"

Alkalmazzunk könnyen kereshető neveket!

Az egybetűs nevek és a számállandók komoly hátránya, hogy igen nehéz rájuk akadni nagyobb mennyiségű szöveg között.

A `MAX_CLASSES_PER_STUDENT` megkeresése nem jelenthet különösebb gondot, de próbáljuk ugyanezt megtenni a 7 értékkel! Egy számjegy megjelenhet fájlnevek részeként, más állandók meghatározásában, továbbá egyéb kifejezésekben, ahol az érték egészen más jelentésben szerepel. A gondok csak súlyosbodnak, ha hosszabb számról van szó, és valaki véletlenül felcserélt két számjegyet – így amellett, hogy a programba bekerült egy hiba, még a megtalálása is nehezebbé vált.

Hasonlóképpen, az e betű változónévként szerepeltetése szintén rossz ötlet, hiszen ez az angol nyelv leggyakoribb betűje, így a programjaink legkisebb részeiben is találunk belőle bőven. Ezen a téren a hosszabb nevek jobbak a rövidebbeknél, és a kereshető nevek ütik az állandókat.

Jómagam egybetűs neveket KIZÁRÓLAG rövid tagfüggvények helyi változóiként alkalmazok. *Fontos, hogy az azonosítóink nevének hossza arányban legyen a hatókörükkel* [N5]. Ha úgy véljük, hogy egy változó vagy állandó jó eséllyel felbukkan egy nagyobb kód több helyén is, feltétlenül adjunk neki jól kereshető nevet. Ismét érdemes összehasonlítanunk az alábbi két kódrészletet:

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

és

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Vegyük észre, hogy a `sum` név nem különösebben beszédes, de legalább kereshető. A szándékaink kimutatására alkalmas nevek használata növeli a függvény méretét, de vegyük észre, mennyivel könnyebben megtalálhatjuk most a `WORK_DAYS_PER_WEEK` állandót – így nem kell utánanéznünk az 5-ös szám összes előfordulásának, kiszűrve azokat, amelyek érdektelenek a számunkra.

Tartózkodjunk a típuskódolástól!

Éppen elég kódolással kell megbirkóznunk ahhoz, hogy magunk ne akarjunk bemenni ebbe az utcába. A típus, illetve a hatókör adatait kódolni a nevekben annyit jelent, hogy a visszafejtést is el kell végeznünk. Az pedig nehezen indokolható, hogy miért is kellene minden új dolgozónak egy további kódolási „nyelvet" is elsajátítania, amikor amúgy is van elég gondja a (többnyire terjedelmes) programkód áttekintésével. Ez a módszer a feladatok megoldásánál újabb szellemi akadályt jelent. Ráadásul a kódolt nevek kiejtése ritkán egyszerű, és könnyen elgépelhetjük őket.

A magyar jelölésről

A régi szép időkben, amikor olyan nyelveket használtunk, amelyekben számított a változók nevének hossza, a fenti szabályt fogcsikorgatva bár, de szükségből megszegtük. A Fortranban a változók első betűje mindenképpen a típus kódját hordozta, a BASIC korai változataiban pedig mindössze egyetlen betűt és egy számjegyet használhattunk változónévként. A magyar jelölésmód ebben a helyzetben komoly előrelépést jelentett.

Ezt a jelölésmódot igen fontosnak tartották a Windows C API-ban, ahol mindössze egész értékű leírókkal, `long` és `void` mutatókkal, valamint a `string` típus különféle megvalósításaival dolgozhattunk (amelyeknek a felhasználási módjai és jellemzői eltérőek voltak). Abban az időben a fordítók nem végeztek típusvizsgálatot, így a programozóknak szükségük volt valamilyen mankóra, hogy visszaemlékezzenek a típusokra.

A modern programozási nyelvekben a típusrendszerek jelentősen gazdagabbak, a fordítók pedig megjegyzik és kikényszerítik a típusokat. Mindemellett a világ a kisebb osztályok és rövidebb függvények használata felé halad, így jó eséllyel a változók használati helyéhez közel megtaláljuk a meghatározásukat is.

A Java-programozóknak nincs szükségük típuskódolásra. Az objektumok erősen típusosak, a szerkesztőkörnyezetek pedig odáig fejlődtek, hogy jóval a fordítás megkezdése előtt észreveszik a típushibákat. Napjainkban tehát a magyar jelölésmód és a típuskódolás más formái csak hátráltatják a munkánkat, és megnehezítik a változók, függvények és osztályok nevének megváltoztatását, illetve a kód olvasását. Mindemellett lehetőséget adnak arra, hogy a típuskódolási rendszer félrevezesse az olvasót.

```
PhoneNumber phoneString;
// A típus változását a név nem követte!
```

A tagváltozók előtagjairól

Többé nincs szükség arra sem, hogy a tagváltozókat az `m_` előtaggal (ami a „member", vagyis „tag" rövidítése) jelöljük. A függvényeink és osztályaink ahhoz már elég kicsik lesznek, hogy ne legyen szükségünk erre a megkülönböztetésre. A tagok jelzésére alkalmazzunk olyan szerkesztőkörnyezetet, amely színekkel emeli ki őket.

```
public class Part {
  private String m_dsc; // szöveges leírás
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Mindemellett a kód olvasói hamar megszokják az előtagok (illetve utótagok) jelenlétét, és csak a nevek lényeges részével foglalkoznak. Minél többet olvassuk a kódot, annál kevésbé vesszük észre az előtagokat, amelyek végül csak zavart okoznak, és arról árulkodnak, hogy a kód régebben készült.

Felületek és megvalósítások

Itt a típuskódolás egy különleges esetével találkozhatunk. Tegyük fel, hogy egy elvont gyárat építünk fel, amely alakzatok létrehozására lesz alkalmas. Maga a gyár egy felület, amelyet majd valódi osztályok valósítanak meg. Hogyan nevezzük hát el őket? Legyen a felület `IShapeFactory`, az osztály pedig `ShapeFactory`? Nos, jómagam inkább a felület nevét hagynám díszítetlenül. Az `I` előtag, amely oly gyakran feltűnik örökölt kódokban, legalábbis félrevezető, rosszabb esetben pedig túlzottan sokatmondó. Nem szeretném ugyanis, ha a felhasználóim tudnák, hogy egy felületet kapnak – elég legyen nekik annyi, hogy ez egy `ShapeFactory`. Ha tehát választani kell, hogy a felületet vagy a megvalósítást kódoljam a névben, az utóbbi mellett döntenék. A `ShapeFactoryImp` vagy akár a szörnyen kinéző `CShapeFactory` alak még mindig jobb szolgálatot tesz, mintha hozzányúlnánk a felület nevéhez.

Kerüljük a fejtörőket!

Nem szerencsés, ha az olvasóknak a programban használt neveket fejben kell megfeleltetniük olyan neveknek, amelyeket már eleve ismernek. Ezek a gondok többnyire akkor jelentkeznek, amikor a kívánt nevet sem a feladat, sem a megoldás tartományához nem tudjuk kötni.

A legtöbbször az egybetűs változónevek okoznak gondot. Természetesen egy ciklusszámláló neve lehet `i`, `j` vagy `k` (de sosem `l`!), amennyiben a hatóköre kicsi, és más név nem ütközik vele. Azért vagyunk ilyen megengedőek, mert az egybetűs ciklusszámlálók a programozói hagyomány részét képezik. Mindazonáltal, legtöbbször az egybetűs ciklusváltozók nem megfelelők – a betűjelük voltaképpen helykitöltő, amihez az olvasónak kell hozzágondolnia az igazi jelentését. Nincs szánalmasabb érv a `c` név használatára, mint az, hogy az `a` és a `b` már foglalt.

A programozók általában okos fickók, és mint ilyenek, gyakran szeretik fejtörők feladásával csillogtatni a képességeiket. Hiszen hatalmas agyunkkal végül is miért ne emlékezhet-

nénk arra, hogy az r nem más, mint egy URL kisbetűs változata, amelyből eltávolítottuk a gépnevet és a protokollt?

Az okos és a professzionális programozó közötti lényegi különbség abban áll, hogy az utóbbi megérti: a tisztaság szempontja mindenek felett áll. A profik ennek szellemében a „legfőbb jó" érdekében használják a képességeiket, és olyan kódot írnak, amit mások is képesek elolvasni.

Osztálynevek

Az osztályok és objektumok neveit főnevekből, illetve főnévi kifejezésekből érdemes összeállítanunk – ilyen a `Customer`, a `WikiPage`, az `Account` vagy az `AddressParser`. Kerüljük az osztálynevekben a `Manager`, a `Processor`, a `Data`, az `Info` és a hozzájuk hasonló szavakat. Az osztályok neve nem lehet ige.

Tagfüggvénynevek

A tagfüggvények neve igékből, illetve igei kifejezésekből állhat. Az elérőfüggvények, módosítófüggvények és lekérdezőfüggvények nevét az értékük alapján kell megadnunk, a `set`, a `get`, illetve az `is` előtaggal, a javabean-szabvány szerint.[4]

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

A túlterhelt konstruktorok esetében alkalmazzunk statikus gyártófüggvényeket a paramétereket leíró nevekkel. Így az alábbi kódsor...

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

...általában jobb, mint ez:

```
Complex fulcrumPoint = new Complex(23.0);
```

Az ilyen alakok használatát kikényszeríthetjük, ha privát konstruktorokat használunk.

[4] http://java.sun.com/products/javabeans/docs/spec.html

Félre a jófejkedéssel!

Ha a nevek túlzottan „jópofák", csak azok jegyzik majd meg őket, akik vevők a szerző humorára, és ők is csak addig, amíg emlékeznek rá, hogy mi volt a vicc. Vajon világos lesz mindenkinek, hogy mire is szántuk a `HolyHandGrenade` (szent kézigránát) nevű függvényt? Persze, vicces név, de a `DeleteItems` (elemek törlése) talán kicsit alkalmasabb választás lenne. Ha választanunk kell a tisztaság és a jópofaság között, mindig maradjunk az előbbinél.

A jópofaság legtöbbször bizalmas vagy szlengkifejezésekben bújik meg. Ne használjuk például a `kill()` helyett a `whack()` elnevezést, de olyan kultúrafüggő jófejségekre se ragadtassuk magunkat, mint az `abort()` helyett alkalmazott `eatMyShorts()` függvénynév.

Mondjuk ki, amit gondolunk – és amit kimondunk, azt gondoljuk komolyan!

Egy fogalom – egy szó

Egy elvont fogalom leírására egyetlen szót használjunk. Igen zavaró lehet például, ha különböző osztályok azonos célú tagfüggvényeinek a `fetch`, a `retrieve` és a `get` nevet adjuk. Hogy emlékszünk majd vissza, hogy melyik név melyik osztályhoz tartozik? Szomorú, de tény, hogy gyakorta azt is észben kell tartanunk, hogy melyik cég, csoport vagy munkatárs írta egy adott könyvtár vagy osztály kódját – csak így tisztázhatjuk, hogy pontosan melyik elnevezés van használatban. Egyébként pedig nem marad más lehetőségünk, mint órákat tölteni a fejlécek és korábbi kódrészletek tanulmányozásával.

Napjaink szerkesztőkörnyezetei, mint az Eclipse vagy az IntelliJ, környezetérzékeny segítséget adnak, így például megmutatják, hogy mely tagfüggvényeket hívhatjuk meg egy adott objektumra. Ez a lista azonban jobbára nem tartalmazza azokat a megjegyzéseket, amelyeket a függvénynevekhez és paraméterlistákhoz fűztünk. Egyáltalán, igen szerencsések vagyunk, ha hozzájutunk a meghatározásokban szereplő paraméterek nevéhez. A függvénynevek legyenek önállóan értelmezhetők, így további kutatómunka nélkül is kiválaszthatjuk a megfelelő tagfüggvényt.

Hasonlóképpen, igencsak zavaró, ha ugyanabban a kódalapban egyszerre jelennek meg `controller` (vezérlő), `manager` (kezelő) és `driver` (meghajtó) nevű elemek. Mi lehet a lényegi különbség a `DeviceManager` (eszközkezelő) és a `ProtocolController` (pro-

tokollvezérlő) között? Miért nem `controller` vagy `manager` mindkettő? És valóban a Driver-ek közé tartoznak? A nevek alapján arra következtetnénk, hogy a két objektum típusa igencsak különböző, és két teljesen eltérő osztályhoz tartoznak.

Egy következetes „szótár" alkalmazása hihetetlen segítséget jelenthet mindenkinek, aki a kódunkat olvassa.

Kerüljük a „szóvicceket"!

Ne használjuk ugyanazt a szót különböző célokra. Ha két eltérő fogalmat ugyanazzal a szóval azonosítunk, valójában egy szóviccet erőltetünk az olvasóra.

Ha követjük az „egy fogalom – egy szó" elvet, végül számos olyan osztályhoz jutunk, amely rendelkezik például `add` (hozzáad) nevű tagfüggvénnyel. Addig persze semmi gond, amíg a különböző `add` tagfüggvények paraméterlistái és visszatérési értékei a jelentésükben megegyeznek.

Előfordulhat azonban, hogy a „következetesség" érdekében valaki olyan helyzetben is alkalmazza az `add` nevet, amikor a korábbi értelemben nem hozzáadásról van szó. Tegyük fel, hogy rendelkezünk néhány osztállyal, amelyekben az `add` tagfüggvény új értéket hoz létre két már meglevő érték összeadásával vagy összefűzésével. Most képzeljük el, hogy létrehozunk egy új osztályt, amelynek az egyik tagfüggvénye egy új elemet helyez el egy gyűjteményben. Adjuk ennek is az `add` nevet? Ez a döntés következetesnek tűnhet, hiszen annyi más `add` tagfüggvényünk van már, ez esetben azonban a jelentés eltérő, így jobban tesszük, ha az `insert` (beszúr) vagy az `append` (hozzáfűz) nevet alkalmazzuk. Az `add` használata kimerítené a „szóvicc" fogalmát.

Szerzőként az a célunk, hogy a kódunkat minél érthetőbbé tegyük mások számára. Azt szeretnénk, ha elég lenne gyorsan átfutni, és nem lenne szükség hosszadalmas tanulmányozásra. A ponyvaregények világát szeretnénk a magunkévá tenni, ahol az író maga felel a műve érthetőségéért, nem pedig azokét a tudományos művekét, ahol az olvasónak kell beleásnia magát a témába, hogy megfejtse a cikk jelentését.

Alkalmazzuk a megoldástartomány neveit!

Ne feledjük, hogy kódunkat programozók olvassák majd. Vagyis nyugodtan használhatjuk a számítástudomány szakkifejezéseit, az algoritmusok és minták neveit, valamint matematikai szakkifejezéseket. Már csak azért sem érdemes a feladattartományból választani a neveket, mert nem szeretnénk, ha a munkatársainknak folyamatosan tisztázniuk kellene az ügyféllel olyan dolgok neveit, amelyeknek a jelentése egyébként világos a számunkra.

Az `AccountVisitor` név sokat mond egy olyan programozónak, akinek ismerős a Látogató minta. Melyik programozó ne tudná, mi is állhat a `JobQueue` (munkasor) név mögött? A programozóknak rengeteg technikai feladatot kell elvégezniük – ezek leírására pedig nincs jobb a technikai neveknél.

Használjuk a feladattartomány neveit!

Ha ugyanakkor az elgondolásunknak nincs programozástechnikai megfelelője, bátran alkalmazzuk a leírására a feladattartomány neveit. Így legalább a kódunkat kezelő programozó bizalommal fordulhat a feladat szakértőjéhez tanácsért, ha valahol elakadna.

A feladat és a megoldás fogalmainak elválasztása a jó programozók és tervezők munkájának szerves része. Azon kódok esetében, amelyek közelebb állnak a feladat tartományának fogalomrendszeréhez, nyilván a neveket is innen kell vennünk.

Hozzunk létre jelentésgazdag környezetet!

Kevés olyan név ismeretes, ami önmagában hordozza a jelentése magyarázatát – az igazság az, hogy a nevek többsége nem ilyen. Ahhoz, hogy az olvasóink értsék a jelentésüket, megfelelően elnevezett osztályokban, függvényekben, illetve névterekben kell elhelyeznünk őket. Csak ha minden más módszer kudarcot vall, akkor érdemes az előtagok bevezetését megfontolnunk.

Képzeljük el, hogy a következő változókat használjuk: `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` és `zipcode` (keresztnév, vezetéknév, utca, házszám, város, állam és irányítószám). Elég világos, hogy ezek összességében egy postacímet adnak meg. De mi a helyzet, ha a `state` változót önmagában használjuk egy tagfüggvényben? Ott is rögtön arra gondolunk, hogy ez egy postacím része (vagy inkább arra, hogy itt „állapot" jelentésben szerepel)?

A környezetet jelölhetjük előtagokkal, mint `addrFirstName`, `addrLastName`, `addrState`, és így tovább. Így az olvasók legalább tisztában lesznek azzal, hogy ezek a változók egy nagyobb szerkezet részei. Természetesen sokkal jobb megoldás, ha létrehozunk egy `Address` (cím) nevű osztályt. Így még a fordító is tudni fogja, hogy a változók együttesen valamilyen nagyobb célt szolgálnak.

Vegyük a 2.1. példában látható tagfüggvényt. Vajon szükség van arra, hogy érthetőbbé tegyük a változók környezetét? A függvény neve ugyanis csak részben árulja el, hogy miről van szó – a teljes kép az algoritmus láttán bontakozik ki. Csak a függvény átolvasása után válik világossá, hogy a `number`, a `verb` és a `pluralModifier` változók egy képernyőre írt üzenetben kapnak szerepet. Sajnos a környezetet magunknak kell kikövetkeztetnünk – a változók jelentését első ránézésre teljes homály fedi.

2.1. példa

Változók homályos környezetben

```
private void printGuessStatistics(char candidate, int count) {
      String number;
     String verb;
      String pluralModifier;
     if (count == 0) {
       number = "no";
        verb = "are";
        pluralModifier = "s";
     } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
     } else {
       number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
     }
     String guessMessage = String.format(
       "There %s %s %s%s", verb, number, candidate, pluralModifier
     );
     print(guessMessage);
   }
```

A függvény kissé hosszú, és mindenütt szerepelnek benne a kérdéses változók. Ahhoz, hogy a kódját több részre bontsuk, létre kell hoznunk egy `GuessStatisticsMessage` osztályt – ennek lesznek tagjai a változók. Ez már tökéletesen tiszta környezetet biztosít a számukra, hiszen egyértelműen az osztály részeivé váltak. A környezet tisztázása mellett az algoritmust is világosabbá tettük, ugyanis kisebb függvényekre tördeltük (lásd a 2.2. példát).

2.2. példa

Változók, ezúttal jól meghatározott környezetben

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;
  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
     return String.format(
       "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }
```

folytatódik

2.2. példa

Változók, ezúttal jól meghatározott környezetben – folytatás

```
    private void createPluralDependentMessageParts(int count) {
      if (count == 0) {
         thereAreNoLetters();
      } else if (count == 1) {
         thereIsOneLetter();
      } else {
        thereAreManyLetters(count);
      }
    }
    private void thereAreManyLetters(int count) {
      number = Integer.toString(count);
       verb = "are";
       pluralModifier = "s";
    }
    private void thereIsOneLetter() {
       number = "1";
      verb = "is";
      pluralModifier = "";
    }
    private void thereAreNoLetters() {
      number = "no";
       verb = "are";
       pluralModifier = "s";
    }
  }
```

Környezetből is megárt a sok!

Képzeletbeli „Gas Station Deluxe" (luxusbenzinkút) nevű alkalmazásunkban nem igazán jó ötlet minden osztály nevét a `GSD` rövidítéssel kezdeni. Ezzel csak a saját életünket nehezítjük meg. Képzeljük csak el: beírjuk a `G` betűt, lenyomjuk a kiegészítés billentyűjét, és egy mérföldes listát kapunk a program összes osztályáról. Ezt szeretnénk? Miért nehezítenénk meg a fejlesztőkörnyezetnek, hogy segítsen?

Hasonló a helyzet, ha kitaláltuk, hogy szükség lenne egy `MailingAddress` (levelezési cím) osztályra a `GSD` számlázási moduljában, és a `GSDAccountAddress` nevet adjuk neki. Mi történik később, ha egy ügyfélkapcsolati alkalmazásban szükségünk lesz a levelezési címre? Vajon egyből eszünkbe jut a `GSDAccountAddress`? Tényleg ez a megfelelő név? Nos, úgy tűnik, 17-ből 10 karakter felesleges vagy lényegtelen.

A rövidebb nevek általában jobbak, mint a hosszúak, mindaddig, amíg a jelentésük világos. Ne jelezzük tehát a szükségesnél beszédesebben a környezetet.

Az `accountAddress` és a `customerAddress` nagyszerűen alkalmazhatók az `Address` osztály példányainak neveként, de osztálynévként már nem állnák meg a helyüket. Erre a célra ugyanis az `Address` tökéletes választás. Ha különbséget kell tennünk MAC-címek, postacímek és webcímek között, a `PostalAddress`, a `MAC` és az `URI` nevek alkalmasabbak, hiszen pontosabban meghatározzák az osztály szerepét, és valójában ez lenne a névadás célja.

Zárszó

Ahhoz, hogy jó neveket válasszunk, szükség van némi leírókészségre, valamint a kollégáinkkal közös kulturális alapokra. Mindez leginkább az oktatói képességeinket érinti – semmi köze a technikai, az üzleti vagy a vezetői képességekhez. Éppen ezért sokan igen esetlenek ezen a téren.

Rengetegen tartanak az átnevezéstől, mert úgy gondolják, hogy a fejlesztőtársaik tiltakozni fognak. Mi nem osztjuk ezt a félelmet, sőt azt gondoljuk – amennyiben az új nevek valóban jobbak –, hogy kifejezetten hálásak lesznek ezért. Legtöbbször ugyanis egyáltalán nem jegyezzük meg az osztályok és a tagfüggvények nevét. Korszerű segédeszközeink gondoskodnak az ilyesfajta részletekről, így inkább arra figyelünk, hogy tudjuk-e úgy olvasni a kódot, mint bekezdéseket és mondatokat, vagy legalábbis mint táblákat és adatszerkezeteket (a mondatok nem mindig alkalmasak az adatok megjelenítésére). A névváltoztatással pontosan annyira lepjük meg a munkatársainkat, mint a kód bármely más javításával. Emiatt semmiképpen se torpanjunk meg!

Próbáljuk legalább részben követni a leírtakat, és ítéljük meg magunk, hogy valóban javul-e a kód olvashatósága. Ha mások által írt kódot kell a gondjainkba vennünk, alkalmazzunk újratervezési segédeszközöket a céljaink elérésére. Higgyük el, az energiabefektetés már rövid távon is kifizetődik – hosszú távon pedig jelentős hasznot hoz.