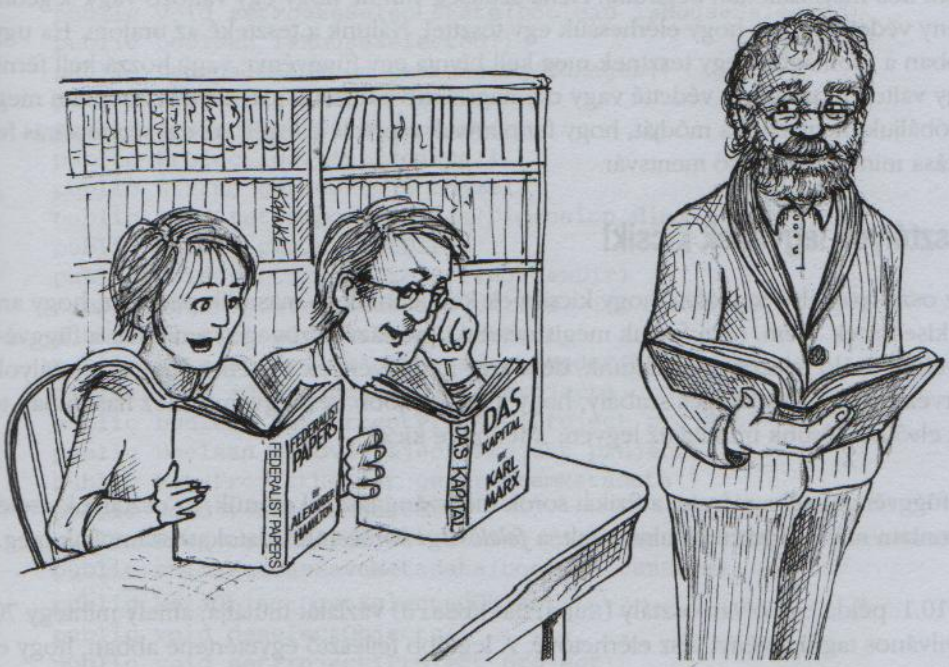
10

Osztályok

Jeff Langr



A könyvben eddig arra összpontosítottunk, hogy miként írhatunk helyesen kódsorokat és kódblokkokat, valamint megmutattuk a függvények helyes felépítését és a közöttük levő kapcsolatokat. A kódutasításokra és az azokból összeállított függvények kifejezőkészségére fordított figyelem azonban még nem elég ahhoz, hogy tiszta kódunk legyen – a kód szervezésének magasabb szintjeire is oda kell figyelnünk. Beszéljünk hát a tiszta osztályokról!

Az osztályok szervezése

A Javában szokásos megegyezést követve egy osztálynak a változók listájával kell kezdődnie. Ezek között is első helyen a nyilvános statikus állandók kell álljanak (ha vannak ilyenek), majd a privát statikus változók, végül pedig a privát példányváltozók. A nyilvános változók használata ritkán indokolt.

A változók listáját a nyilvános függvények felsorolásának kell követnie. A nyilvános függvények által meghívott privát segédfüggvényeket közvetlenül maga a hívó nyilvános függvény után célszerű elhelyezni. Mindez a Leszálló szabályt követi, és segít, hogy a programot úgy lehessen olvasni, mint egy újságcikket.

Egységbe zárás

A változóinkat és segédfüggvényeinket szeretjük priváttá tenni, de ezt a szabályt azért nem kell megszállottan betartani. Néha szükség van rá, hogy egy változó vagy segédfüggvény védett legyen, hogy elérhessük egy teszttel. Nálunk a teszteké az uralom. Ha ugyanabban a csomagban egy tesztnek meg kell hívnia egy függvényt, vagy hozzá kell férnie egy változóhoz, akkor védetté vagy csomagszintűvé tesszük azt, először azonban megpróbáljuk megtalálni a módját, hogy fenntartsuk a privát jellegét. Az egységbe zárás fellazítása mindig az utolsó mentsvár.

Az osztályok legyenek kicsik!

Az osztályok első szabálya, hogy kicsiknek kell lenniük. A második pedig az, hogy annál is kisebbnek. Nem, nem fogjuk megismételni ugyanazt a szöveget, amit már a függvényekről szóló fejezetben is láttunk, de ahogy a függvények esetében, úgy az osztályok tervezésekor is elsődleges szabály, hogy a kisebb jobb. A függvényekhez hasonlóan tehát az első kérdésünk mindig ez legyen: „Mennyire kicsi?".

A függvényeknél a méretet a fizikai sorok megszámolásával mértük, az osztályok esetében azonban más módszert alkalmazunk: a *felelősségi köröket* (feladatokat) számoljuk meg.[1]

A 10.1. példa egy olyan osztály (`SuperDashboard`) vázlatát mutatja, amely mintegy 70 nyilvános tagfüggvényt tesz elérhetővé. A legtöbb fejlesztő egyetértene abban, hogy ez túl sok – egyesek talán még „istenosztálynak" is neveznék a `SuperDashboard`-ot.

[1] [RDD]

10.1. példa

Túl sok felelősségi kör

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMouseSelected)
    public boolean isMouseSelected()
    public LanguageManager getLanguageManager()
    public Project getProject()
    public Project getFirstProject()
    public Project getLastProject()
    public String getNewProjectName()
    public void setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public void setCurrentDir(String newDir)
    public void updateStatus(int dotPos, int markPos)
    public Class[] getDataBaseClasses()
    public MetadataFeeder getMetadataFeeder()
    public void addProject(Project project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
    public MetaProjectHeader getProgramMetadata()
    public void resetDashboard()
    public Project loadProject(String fileName, String projectName)
    public void setCanSaveMetadata(boolean canSave)
    public MetaObject getSelectedObject()
    public void deselectObjects()
    public void setProject(Project project)
    public void editorAction(String actionName, ActionEvent event)
    public void setMode(int mode)
    public FileManager getFileManager()
    public void setFileManager(FileManager fileManager)
    public ConfigManager getConfigManager()
    public void setConfigManager(ConfigManager configManager)
    public ClassLoader getClassLoader()
```

folytatódik

10.1. példa

Túl sok felelősségi kör – folytatás

```
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
    public MetaObject pasting(
        MetaObject target, MetaObject pasted, MetaProject project)
    public void processMenuItems(MetaObject metaObject)
    public void processMenuSeparators(MetaObject metaObject)
    public void processTabPages(MetaObject metaObject)
    public void processPlacement(MetaObject object)
    public void processCreateLayout(MetaObject object)
    public void updateDisplayLayer(MetaObject object, int layerIndex)
    public void propertyEditedRepaint(MetaObject object)
    public void processDeleteObject(MetaObject object)
    public boolean getAttachedToDesigner()
    public void processProjectChangedState(boolean hasProjectChanged)
    public void processObjectNameChanged(MetaObject object)
    public void runProject()
    public void setAçowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    //... itt különböző nem nyilvános tagfüggvények következnek ...
}
```

De mi lenne, ha a `SuperDashboard` osztály csak a 10.2. példában látható tagfüggvényeket tartalmazná?

10.2. példa

Elég kicsi?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
     public int getMajorVersionNumber()
     public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Öt tagfüggvény nem olyan sok, nem igaz? Ebben az esetben azonban mégis túl sok, mert a tagfüggvények kis száma ellenére a `SuperDashboard` osztály túl sok *felelősségi körrel* rendelkezik.

Az osztály nevének le kell írnia, hogy az osztály milyen feladatokat lát el – valójában már az elnevezés is segít, hogy meghatározzuk az osztály méretét. Ha nem tudunk tömör nevet adni egy osztálynak, akkor az osztály valószínűleg túl nagy. Minél több értelmű az osztály neve, annál nagyobb az esély rá, hogy az osztály túl sok felelősségi körrel rendelkezik. Az olyan töltelékszavakat, mint a `Processor`, a `Manager` vagy a `Super` tartalmazó osztálynevek gyakran a feladatok szerencsétlen összevonására utalnak.

Arra is képesnek kell lennünk, hogy röviden (körülbelül 25 szóban) leírjuk az osztályt az „if" (ha), „and" (és), „or" (vagy), illetve „but" (de) kötőszavak használata nélkül. Hogyan írnánk körül a `SuperDashboard` osztályt? „A `SuperDashboard` osztály a fókuszt utoljára birtokló összetevőhöz nyújt hozzáférést, és lehetővé teszi a változat- és felépítésszámok nyomon követését is." Nos, az első „és" szócska világosan jelzi, hogy a `SuperDashboard` osztálynak túl sok a feladata.

Az egyetlen felelősségi kör elve

Az egyetlen felelősségi kör elve (Single Responsibility Principle, SRP)[2] azt mondja ki, hogy egy osztálynak vagy modulnak kizárólag egy oka lehet a változásra. Ez az elv megadja mind a felelősségi kör meghatározását, mind az osztályok méretére vonatkozó útmutatást. Az osztályok csak egy felelősségi körrel vagy feladattal szabad, hogy rendelkezzenek – csak ez adhat okot arra, hogy módosuljanak.

A látszólag kicsi `SuperDashboard` osztály a 10.2. példában két okból változhat meg. Először is, nyomon követi a változatinformációt, amelyet valószínűleg minden alkalommal frissíteni kell, amikor a szoftvert leszállítják. Másodszor, az osztály Java Swing összetevőket is kezel (a `JFrame` leszármazottja, amely a legfelső szintű grafikus ablakok Swingbeli ábrázolása). Semmi kétség, a változatszámot frissíteni kell, ha bármit változtatunk a Swing-kódon, de ennek a fordítottja már nem feltétlenül igaz: a változatinformáció módosítására akkor is szükség lehet, ha csak a rendszer más részeinek kódja változik meg.

A felelősségi körök (a változásra okot adó események) azonosítása gyakran segít, hogy pontosabban felismerjük, milyen elvont ábrázolásra van szükség a kódban. A `SuperDashboard` osztály mindhárom tagfüggvényét, amely a változatinformációt kezeli, könnyen kiemelhetjük egy önálló osztályba, mondjuk `Version` néven (lásd a 10.3. példát). A `Version` osztály olyan szerkezet, amelyet nagy eséllyel újrahasznosíthatunk más alkalmazásokban is.

[2] Erről az elvről bővebben lásd: [PPP].

10.3. példa

Egyetlen felelősségi körrel rendelkező osztály

```
public class Version {
      public int getMajorVersionNumber()
      public int getMinorVersionNumber()
     public int getBuildNumber()
}
```

Az SRP az objektumközpontú tervezés egyik legfontosabb elve, ráadásul az egyik legkönnyebben felfogható és betartható. Különös módon azonban éppen az egyetlen felelősségi kör elve az, amelyet az osztálytervezési elvek közül a leggyakrabban hagynak figyelmen kívül. Rendszeresen találkozunk olyan osztályokkal, amelyek túl sok feladatot látnak el. Mi lehet ennek az oka?

Működőképes kódot alkotni és tiszta szoftvert írni két igencsak különböző dolog. A legtöbbünk fejében korlátozott a hely, ezért a szervezés és a tisztaság helyett inkább a kód működőképességére összpontosítunk. Ez teljesen rendben is van, hiszen a feladatok rangsorolása a *programozásban* éppen olyan fontos, mint magukban a programokban.

A gond csak az, hogy túl sokan gondolják úgy, hogy ha a program működik, akkor a munkájuk befejeződött, és nem váltanak át a *másik* feladatra: a szervezésre és a kitisztításra. Hajlamosak vagyunk a következő problémára fordítani a figyelmünket, ahelyett, hogy visszatérnénk, és a túlzsúfolt osztályokat egyetlen feladatot ellátó, önálló egységekre bontanánk.

Ugyanakkor sok fejlesztő attól fél, hogy a nagy számú kicsi, egyetlen felelősségi körrel rendelkező osztály megnehezíti a nagyobb egész megértését. Attól tartanak, hogy osztályról osztályra kell ugrálniuk, hogy kitalálják, hogyan végezhető el egy nagyobb feladat.

Egy sok kis osztályból álló rendszernek azonban nincs több mozgó része, mint egy néhány nagy osztályból állónak. Egy nagy osztályokból felépülő rendszerben ugyanolyan sok a megtanulnivalónk. A kérdés tehát az, hogy a szerszámainkat kis fiókokba szeretnénk rendezni, amelyek mindegyikében jól meghatározott és felcímkézett szerszámok vannak, vagy csak beleömlesztenénk mindent néhány nagy fiókba?

Minden komolyabb méretű rendszer bonyolult, és jelentős mennyiségű programlogikát tartalmaz. Ennek a bonyolultságnak a kezelése elsősorban *szervezést* igényel, hogy a fejlesztő tudja, hol keresse az elemeket, és csak annyit kelljen megértenie a rendszerből, amennyire az adott pillanatban szüksége van. Ezzel szemben egy nagy és többcélú osztályokból álló rendszer mindig akadályt gördít elénk, mert egy csomó olyan dolgon is át kell gázolnunk, amiről jelen pillanatban nem is kellene tudnunk.

A nyomaték kedvéért megismételnénk: a rendszereink épüljenek fel sok kis osztályból, nem néhány nagyból. Minden kis osztály egyetlen feladatot lásson el, egyetlen oka legyen csak a változásra, és kevés más osztállyal működjön együtt, hogy elérje a kívánt rendszerviselkedést.

Összetartás

Az osztályok kis számú példányváltozóval kell rendelkezzenek, és az osztály minden tagfüggvénye fel kell, hogy használjon egyet vagy többet ezek közül a változók közül. Általában véve, minél több változót használ fel egy tagfüggvény, annál jobban összetart az adott tagfüggvény az osztályával. Azt az osztályt nevezzük maximálisan összetartónak, amelyben minden tagfüggvény minden változót felhasznál.

Általában nem tanácsos és nem is lehetséges ilyen maximálisan összetartó osztályt létrehozni – másrészről viszont azt szeretnénk, ha az összetartó erő nagy lenne. Ha az összetartás nagy, az azt jelenti, hogy az osztály tagfüggvényei és változói egymástól függnek, és logikus egészet alkotnak.

Vegyük a 10.4. példában látható veremmegvalósítást: ez egy igen összetartó osztály, mert a három tagfüggvénye közül csak a `size()` nem használja mindkét változót.

10.4. példa

Egy összetartó osztály: Stack.java

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();
  public int size() {
    return topOfStack;
  }
  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }
  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

A rövid függvények és paraméterlisták írására való törekvés néha a példányváltozók elburjánzásához vezet, amelyeket csak a tagfüggvények egy része használ. Amikor ez törté-

nik, az szinte mindig azt jelenti, hogy van legalább egy másik osztály, amelyet kiemelhetnénk a nagyobb osztályból. A változókat és tagfüggvényeket meg kell próbálnunk két vagy több osztályba szétválasztani, hogy az új osztályok összetartása nagyobb legyen.

Az összetartás sok kicsi osztályt eredményez

A nagy függvények kisebbekre történő felbontása önmagában az osztályok elszaporodását eredményezi. Vegyünk egy nagy függvényt, amelyben sok-sok változót vezetünk be. Tegyük fel, hogy ennek a függvénynek csak egy kis részét szeretnénk egy önálló függvénybe kiemelni. A kiemelni kívánt kód azonban négy olyan változót is használ, amelyet a függvényben vezettünk be. Mind a négy változót át kell adnunk paraméterként az új függvénynek?

Egyáltalán nem! Ha az említett négy változót az osztály példányváltozóivá léptetjük elő, akkor a kódot *egyetlen* változó átadása nélkül kiemelhetjük, és a függvényt *könnyen* felbonthatjuk kisebb darabokra.

Sajnos azonban ez azt is jelenti, hogy az osztályaink elvesztik az összetartó erejüket, mert egyre több példányváltozót halmoznak fel, amelyek létezésének kizárólag az a célja, hogy lehetővé tegyék néhány függvénynek, hogy osztozzanak rajtuk. De várjunk csak! Ha bizonyos változókat közösen kíván használni néhány függvény, akkor nem kellene önálló osztályt létrehozni belőlük? Dehogynem – ha az osztályok összetartása megszűnik, bontsuk fel őket!

Egy nagy függvény sok kisebb függvényre való felbontása gyakran ad alkalmat arra is, hogy felbontással több kisebb osztályt hozzunk létre. Ez jobban szervezetté és átláthatóbb szerkezetűvé teszi a programunkat.

Szemléltetésképpen vizsgáljunk meg egy réges-régi példát Knuth csodálatos könyvéből, a *Literate Programming*-ból.[3] A 10.5. példa Knuth `PrintPrimes` programjának Java nyelvre fordított változatát mutatja. Hogy Knuth tisztességén ne essen csorba, el kell mondanunk, hogy nem ő írta meg így a programot, hanem a WEB nevű eszköze állította elő. Azért ezt a programot használjuk, mert jó kiindulópontot jelent a nagy függvények sok kisebb függvényre és osztályra való felbontásához.

[3] [Knuth92]

10.5. példa

PrintPrimes.java

```
package literatePrimes;
public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];
    J = 1;
    K = 1;
    P[1] = 2;
     ORD = 2;
     SQUARE = 9;
     while (K < M) {
       do {
        J = J + 2;
        if (J == SQUARE) {
           ORD = ORD + 1;
           SQUARE = P[ORD] * P[ORD];
          MULT[ORD - 1] = J;
         }
        N = 2;
        JPRIME = true;
         while (N < ORD && JPRIME) {
          while (MULT[N] < J)
            MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
          N = N + 1;
         }
       } while (!JPRIME);
```

folytatódik

10.5. példa

PrintPrimes.java – folytatás

```
        K = K + 1;
        P[K] = J;
      }
      {
        PAGENUMBER = 1;
        PAGEOFFSET = 1;
        while (PAGEOFFSET <= M) {
          System.out.println("The First " + M +
                             " Prime Numbers --- Page " + PAGENUMBER);
          System.out.println("");
          for (ROWOFFSET = PAGEOFFSET; ROWOFFSET <
                                       PAGEOFFSET + RR; ROWOFFSET++){
            for (C = 0; C < CC;C++)
              if (ROWOFFSET + C * RR <= M)
                System.out.format("%10d", P[ROWOFFSET + C * RR]);
            System.out.println("");
          }
          System.out.println("\f");
          PAGENUMBER = PAGENUMBER + 1;
          PAGEOFFSET = PAGEOFFSET + RR * CC;
        }
      }
    }
  }
```

Egyetlen függvényként megírva ez a program egy nagy katyvasz: sok-sok behúzási szintet és furcsa változót tartalmaz, a szerkezete pedig szoros csatolású. Az a minimum, hogy az egyetlen nagy függvényt több kisebbre bontsuk fel.

A 10.6.–10.8. példák a 10.5. példa kódjának felosztását mutatják olyan kisebb osztályokra és függvényekre, amelyeknek a változóikkal együtt beszédes nevet adtunk.

10.6. példa

PrimePrinter.java (újratervezve)

```
package literatePrimes;
public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);
    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                          "The First " + NUMBER_OF_PRIMES +
                                 " Prime Numbers");
    tablePrinter.print(primes);
  }
}
```

10.7. példa

RowColumnPagePrinter.java

```
package literatePrimes;
import java.io.PrintStream;
public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;
  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
     numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }
  public void print(int data[]) {
    int pageNumber = 1;
     for (int firstIndexOnPage = 0;
          firstIndexOnPage < data.length;
          firstIndexOnPage += numbersPerPage) {
       int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
                 data.length - 1);
```

folytatódik

10.7. példa

RowColumnPagePrinter.java – folytatás

```
        printPageHeader(pageHeader, pageNumber);
        printPage(firstIndexOnPage, lastIndexOnPage, data);
        printStream.println("\f");
        pageNumber++;
      }
    }
    private void printPage(int firstIndexOnPage,
                           int lastIndexOnPage,
                           int[] data) {
      int firstIndexOfLastRowOnPage =
        firstIndexOnPage + rowsPerPage - 1;
      for (int firstIndexInRow = firstIndexOnPage;
           firstIndexInRow <= firstIndexOfLastRowOnPage;
           firstIndexInRow++) {
        printRow(firstIndexInRow, lastIndexOnPage, data);
        printStream.println("");
      }
    }
    private void printRow(int firstIndexInRow,
                          int lastIndexOnPage,
                          int[] data) {
      for (int column = 0; column < columnsPerPage; column++) {
        int index = firstIndexInRow + column * rowsPerPage;
        if (index <= lastIndexOnPage)
          printStream.format("%10d", data[index]);
      }
    }
    private void printPageHeader(String pageHeader,
                                 int pageNumber) {
      printStream.println(pageHeader + " --- Page " + pageNumber);
      printStream.println("");
    }
    public void setOutput(PrintStream printStream) {
      this.printStream = printStream;
    }
  }
```

10.8. példa

PrimeGenerator.java

```
package literatePrimes;
import java.util.ArrayList;
public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;
  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
     checkOddNumbersForSubsequentPrimes();
     return primes;
  }
  private static void set2AsFirstPrime() {
     primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }
  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
      if (isPrime(candidate))
         primes[primeIndex++] = candidate;
     }
  }
  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate))
{
      multiplesOfPrimeFactors.add(candidate);
      return false;
     }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }
  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor =
primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor *
nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }
  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
```

folytatódik

10.7. példa

RowColumnPagePrinter.java – folytatás

```
        printPageHeader(pageHeader, pageNumber);
        printPage(firstIndexOnPage, lastIndexOnPage, data);
        printStream.println("\f");
        pageNumber++;
      }
    }
    private void printPage(int firstIndexOnPage,
                           int lastIndexOnPage,
                           int[] data) {
      int firstIndexOfLastRowOnPage =
        firstIndexOnPage + rowsPerPage - 1;
      for (int firstIndexInRow = firstIndexOnPage;
           firstIndexInRow <= firstIndexOfLastRowOnPage;
           firstIndexInRow++) {
        printRow(firstIndexInRow, lastIndexOnPage, data);
        printStream.println("");
      }
    }
    private void printRow(int firstIndexInRow,
                          int lastIndexOnPage,
                          int[] data) {
      for (int column = 0; column < columnsPerPage; column++) {
        int index = firstIndexInRow + column * rowsPerPage;
        if (index <= lastIndexOnPage)
          printStream.format("%10d", data[index]);
      }
    }
    private void printPageHeader(String pageHeader,
                                 int pageNumber) {
      printStream.println(pageHeader + " --- Page " + pageNumber);
      printStream.println("");
    }
    public void setOutput(PrintStream printStream) {
      this.printStream = printStream;
    }
  }
```

10.8. példa

PrimeGenerator.java

```
package literatePrimes;
import java.util.ArrayList;
public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;
  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
     checkOddNumbersForSubsequentPrimes();
     return primes;
  }
  private static void set2AsFirstPrime() {
     primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }
  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
          primeIndex < primes.length;
          candidate += 2) {
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }
  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate))
{
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }
  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor =
primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor *
nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }
  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
```

folytatódik

10.8. példa

PrimeGenerator.java – folytatás

```
        if (isMultipleOfNthPrimeFactor(candidate, n))
          return false;
      }
      return true;
    }
    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
      return
        candidate ==
  smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }
    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n)
  {
      int multiple = multiplesOfPrimeFactors.get(n);
       while (multiple < candidate)
         multiple += 2 * primes[n];
      multiplesOfPrimeFactors.set(n, multiple);
      return multiple;
    }
  }
```

Az első dolog, amit észrevehetünk, hogy a program jelentősen hosszabb lett – a hossza valamivel több, mint egy oldalról majdnem három oldalra nőtt. Ennek a növekedésnek több oka is van. Először is, az újratervezett program hosszabb, leíróbb jellegű változóneveket használ. Másodszor, az újratervezett program a függvények és osztályok bevezetésében fűz magyarázatokat a kódhoz. Harmadszor, a program olvashatóságát üreshely karakterekkel és formázási eljárásokkal biztosítottuk.

Figyeljük meg, hogy a programot három fő felelősségi körre osztottuk fel. A főprogramot teljes egészében a `PrimePrinter` osztály tartalmazza, amelynek a feladata a végrehajtási környezet kezelése. Az osztály akkor változik meg, ha a meghívás módja módosul. Ha a programot például SOAP-szolgáltatássá alakítanánk, az átalakítás ezt az osztályt érintené.

A `PrimeGenerator` osztály azt tudja, hogyan állítsa elő prímszámok listáját. Vegyük észre, hogy ezt az osztályt nem arra szánták, hogy objektumként példányosítsuk. Az osztály csupán egy hasznos hatókör, amelyben bevezethetjük és elrejthetjük a változóit. Ez az osztály akkor változik meg, ha a prímszámok kiszámítását végző algoritmust módosítják.

Vegyük észre, hogy nem átírás történt. Nem kezdtük elölről és írtuk újra a programot: ha közelebbről szemügyre vesszük a két programváltozatot, láthatjuk, hogy ugyanazt az algoritmust és ugyanazokat a műveleteket használják a feladatuk elvégzésére.

A változtatást úgy hajtottuk végre, hogy írtunk egy tesztcsomagot, amely *pontosan* meghatározta az első program viselkedését, majd rengeteg apró módosítást végeztünk, egyenként haladva. A programot minden módosítás után lefuttattuk, hogy meggyőződjünk róla, hogy a viselkedés nem változott. Egyik apró lépést a másik után elvégezve kitisztítottuk az eredeti programot, és előállítottuk belőle az új változatot.

Szervezés a változást szem előtt tartva

A legtöbb rendszer folyamatosan változik, és minden változás azzal a kockázattal jár, hogy a rendszer fennmaradó része többé nem a várt módon fog működni. Egy tiszta rendszerben az osztályainkat úgy szervezzük, hogy csökkentsük a változással járó kockázatot.

A 10.9. példában látható `Sql` osztályt arra használjuk, hogy helyes alakú SQL-karakterláncokat állítsunk elő a megfelelő metaadatok birtokában. Az osztály fejlesztése még nem fejeződött be, ezért még nem támogatja az olyan SQL-szolgáltatásokat, mint az `update` utasítások. Ha ennek eljön az ideje, „fel kell nyitnunk" az osztályt, hogy módosítsuk. Egy osztály felnyitásával viszont az a gond, hogy kockázattal jár. Minden módosítás, amelyet egy osztályon végrehajtunk, esélyt ad arra, hogy az osztály más kódrészei működésképtelenné váljanak, ezért az osztályt újra teljes tesztelés alá kell vonnunk.

10.9. példa

Osztály, amelyet fel kell nyitni a módosításhoz

```
public class Sql {
    public Sql(String table, Column[] columns)
    public String create()
    public String insert(Object[] fields)
    public String selectAll()
    public String findByKey(String keyColumn, String keyValue)
    public String select(Column column, String pattern)
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

Az `Sql` osztályt módosítani kell, amikor felveszünk egy új típusú utasítást, de akkor is, amikor változtatunk egy adott típusú utasítás részletein – például ha úgy kell módosítanunk a `select`-et, hogy a részkijelölést is támogassa. Ez már két ok a változásra, ami azt jelenti, hogy az `Sql` osztály megsérti az egyetlen felelősségi kör elvét.

Ezt a szabálysértést könnyen kiszúrhatjuk, ha a szervezés szempontjából vizsgáljuk az osztályt. Az `Sql` osztály tagfüggvény-szerkezete azt mutatja, hogy vannak olyan privát tagfüggvények – például a `selectWithCriteria` –, amelyek láthatólag csak a `select` utasításokkal állnak kapcsolatban.

A privát tagfüggvények olyan műveletei, amelyek egy osztálynak csak egy kis részhalmazára vonatkoznak, árulkodó jelei lehetnek annak, hogy mely területek szorulnak javításra. A változtatásra azonban elsősorban magának a rendszernek a megváltozása adhat okot. Ha az `Sql` osztály logikailag teljes, akkor nem kell aggódnunk a felelősségi körök szétválasztása miatt. Ha az `update` műveletre belátható időn belül nem lesz szükségünk, akkor jobb békén hagyni az `Sql` osztályt, de amint azon kapjuk magunkat, hogy felnyitunk egy osztályt, célszerű lehet javítani a program szerkezetén.

Mi lenne, ha egy olyan megoldást alkalmaznánk, mint a 10.10. példában? Itt a 10.9. példa `Sql` osztályában meghatározott minden nyilvános felületi tagfüggvényt a saját, az `Sql` osztályból származtatott osztályába emeltünk ki. Figyeljük meg, hogy a privát tagfüggvények, például a `valuesList`, közvetlenül oda kerültek, ahol szükség van rájuk. A közös privát műveleteket két segédosztályban (`Where` és `ColumnList`) szigeteltük el.

10.10. példa

Zárt osztályok halmaza

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}
public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
   @Override public String generate()
}
public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
   @Override public String generate()
}
public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[]
fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[]
columns)
}
public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
      String table, Column[] columns, Criteria criteria)
```

folytatódik

10.10. példa

Zárt osztályok halmaza – folytatás

```
    @Override public String generate()
}
public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String
pattern)
    @Override public String generate()
}
public class FindByKeySql extends Sql
    public FindByKeySql(
      String table, Column[] columns, String keyColumn, String
keyValue)
    @Override public String generate()
}
public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate() {
     private String placeholderList(Column[] columns)
}

public class Where {
     public Where(String criteria)
    public String generate()
}
public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}
```

Az egyes osztályok kódja hihetetlenül egyszerűvé vált, így az osztályok megértése szinte semennyi időt nem igényel. A kockázat, hogy egy függvény tönkretehet egy másikat, köddé válik. A tesztelés szempontjából ebben a megoldásban a logika minden pontját sokkal könnyebb ellenőrizni, mivel az osztályok elszigetelődnek egymástól.

Ugyanilyen fontos, hogy amikor fel kell vennünk az `update` utasításokat, a meglevő osztályok egyikét sem kell módosítanunk. Az `update` utasításokat felépítő logikát az `Sql` egy új alosztályába (`UpdateSql`) kódoljuk. Ez a változtatás a rendszer kódjának egyetlen más részét sem teszi működésképtelenné.

Az `Sql` osztály logikájának átszervezett változata mindenből a legjobbat nyújtja. Betartja az egyetlen felelősségi kör elvét, valamint az objektumközpontú programozás egy másik kulcsfontosságú szabályát, a zárt nyíltság elvét (Open/Closed Principle, OCP[4]) is, amely azt

[4] [PPP]

mondja ki, hogy az osztályoknak nyitottnak kell lenniük a bővítés, de zártnak a módosítás irányába. Az újraszervezett `Sql` osztályunk alosztály-származtatás révén lehetővé teszi a szolgáltatások bővítését, de ezt a változtatást úgy hajthatjuk végre, hogy közben minden más osztályt zárva tartunk. Az `UpdateSql` osztályt tehát csak le kell ejtenünk a helyére.

A rendszereinket úgy célszerű felépítenünk, hogy amikor új vagy módosított szolgáltatásokat adunk hozzájuk, a lehető legkevesebb helyen kelljen beléjük nyúlnunk. Egy ideális rendszerben az új szolgáltatásokat a rendszer bővítésével, és nem a meglevő kód módosításával építjük be.

Elszigetelés a változástól

A szükségletek folyamatosan változnak, ezért a kód is. Az objektumközpontú programozás egyszeregye megtanított minket arra, hogy vannak konkrét osztályok, amelyek a megvalósítás részleteit (a kódot) tartalmazzák, és vannak elvont osztályok, amelyek csak fogalmakat ábrázolnak. Ha egy ügyfélosztály konkrét részletekre támaszkodik, veszélybe kerül, ha ezek a részletek megváltoznak. A részletek megváltozásának hatását úgy szigetelhetjük el, ha felületeket és elvont osztályokat vezetünk be.

A konkrét részletektől való függés a rendszer tesztelése szempontjából is kihívást jelent. Ha felépítünk egy `Portfolio` nevű osztályt, amely egy külső, `TokyoStockExchange` nevű API-ra támaszkodik a befektetési csomag értékének kiszámításához, az említett API változékonysága befolyásolja a teszteseteinket. Nehéz tesztet írni, ha öt percenként másmás eredményt kapunk.

Ahelyett tehát, hogy a `Portfolio` osztályt úgy terveznénk meg, hogy közvetlenül függjön a `TokyoStockExchange` API-tól, létrehozunk egy `StockExchange` nevű felületet, amely egyetlen tagfüggvényt vezet be:

```
public interface StockExchange {
    Money currentPrice(String symbol);
}
```

A `TokyoStockExchange` API-t úgy szerkesztjük meg, hogy ezt a felületet valósítsa meg. Emellett arról is gondoskodunk, hogy a `Portfolio` konstruktora paraméterként egy `StockExchange` hivatkozást kapjon:

```
public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}
```

A tesztünk most létrehozhatja a `StockExchange` felület egy olyan tesztelhető megvalósítását, amely a `TokyoStockExchange` API-t utánozza. Ez a tesztmegvalósítás a teszteléshez használt minden szimbólum aktuális értékét rögzíti. Ha a tesztünk azt szemlélteti, hogy mi történik, ha a portfoliónkhoz öt Microsoft-részvényt vásárolunk, a tesztmegvalósítást úgy kódolhatjuk, hogy minden Microsoft-részvényt 100 dolláros értékkel számoljon.
A `StockExchange` felület tesztmegvalósítása így egy egyszerű táblakeresésre szűkül, ezt követően pedig megírhatunk egy tesztet, amely a portfolió teljes értékéül 500 dollárt vár:

```
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;
  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
  }
  @Test
  public void GivenFiveMSFTTotalShouldBe500() throws Exception {
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
  }
}
```

Ha egy rendszer elég laza csatolású ahhoz, hogy így tesztelhessük, akkor rugalmasabb és könnyebben újrahasznosítható is lesz. A csatolás hiánya azt jelenti, hogy a rendszerünk elemei jobban elszigetelődnek egymástól, illetve a változástól. Ez az elszigetelés könnyebbé teszi a rendszer egyes elemeinek megértését.

Ha a csatolást ilyen módon a minimumra csökkentjük, az osztályaink egy másik osztálytervezési elvhez, a függőségek megfordításának elvéhez (Dependency Inversion Principle, DIP[5]) is tartják magukat. A függőségek megfordításának elve lényegében azt mondja ki, hogy az osztályainknak elvont ábrázolásokra és nem konkrét részletekre kell támaszkodniuk.

`Portfolio` osztályunk ahelyett, hogy a `TokyoStockExchange` osztály megvalósítási részleteitől függne, a `StockExchange` felületre támaszkodik. A `StockExchange` felület egy szimbólum aktuális árának lekérdezését ábrázolja elvont formában. Ez az elvonatkoztatás az ár lekérdezésének minden konkrét részletét elszigeteli, beleértve azt is, hogy honnan származik az ár.

[5] [PPP]

Irodalomjegyzék

[RDD]: Rebecca Wirfsbrock és mások: *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2002.

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

[Knuth92]: Donald E. Knuth: *Literate Programming*, Center for the Study of language and Information, Leland Stanford Junior University, 1992.