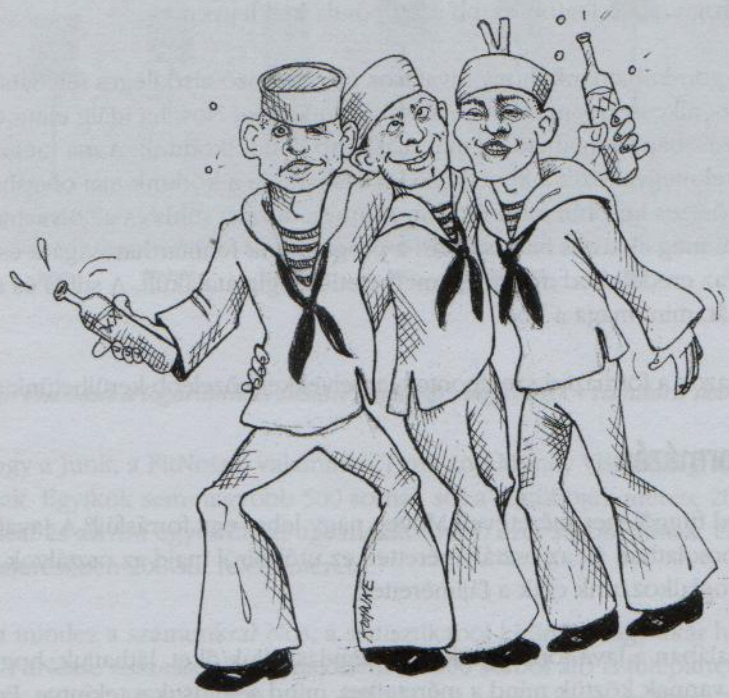
5

Formázás



Ha a kódunk olvasói a színfalak mögé néznek, szeretnénk, ha tiszta, egységes és részleteiben igényes munkát látnának. Szeretnénk, ha a rendezettség láttán leesne az álluk, ha a modulok olvasása közben az ajkuk elégedett mosolyra húzódna – röviden szólva, szeretnénk azt az érzetet kelteni, hogy valóban értjük a dolgunkat. Ha ugyanakkor keszekusza kódot látnak, ami úgy fest, mintha részeg tengerészek csapták volna össze egy görbe estén, hajlamosak arra következtetni, hogy ez az igénytelenség más szempontból is érezhető a programon.

Fontos tehát, hogy különös gondot fordítsunk a kódunk formázására. Néhány egyszerű szabályt kell kijelölnünk a kód formátumának meghatározására, és a későbbiekben szigorúan ragaszkodni a betartásukhoz. Ha csapatban dolgozunk, a teljes csapatnak meg kell állapodnia a szabályokban, majd mindenkinek be kell tartania ezeket. Megkönnyíti a dolgunkat, ha a formázási szabályok alkalmazására valamilyen automatikus segédeszközt állítunk munkába.

A formázás célja

Először is, jobb, ha tisztázzuk, hogy a kód formázása fontos dolog. Túlzottan fontos ahhoz, hogy figyelmen kívül hagyjuk, vagy hogy vallásos tisztelettel viszonyuljunk hozzá. A kód formázása nem más, mint kapcsolat az olvasóval, ennek alkalmas kialakítása pedig a hivatásos programozó legfontosabb szempontja kell legyen.

Micsoda? Azt gondolhatnánk, hogy hivatásos programozó elsődleges feladata az, hogy működő kódot alkosson – minden más csak ez után jön. Nos, ha idáig eljutottunk a könyv olvasásában, remélhetőleg már másképp gondolkodunk. A ma megírt kód működése jó eséllyel megváltozik a következő kiadásban, de a kódunk mai olvashatósága hatással van az összes későbbi módosításra. A programozási stílus és az olvashatóság példát mutatnak, ami még akkor is hatással van a programunk fenntarthatóságára és bővíthetőségére, amikor az eredeti kód már a felismerhetetlenségig átalakult. A stílus és a fegyelem tovább él tehát, mint maga a kód.

Melyek tehát azok a formázási szempontok, amelyekkel közelebb kerülhetünk az olvasóhoz?

Függőleges formázás

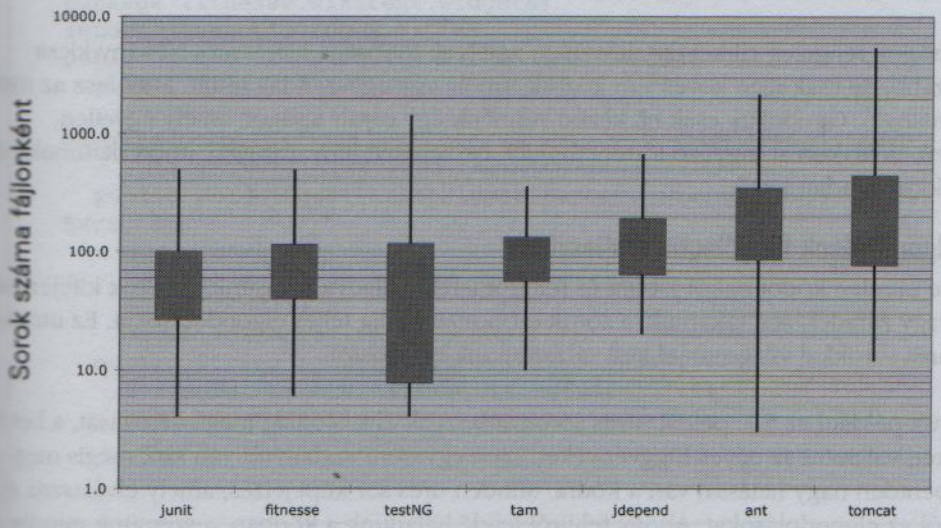
Kezdjük a kód függőleges méretével. Milyen nagy lehet egy forrásfájl? A Javában a fájlméret közeli kapcsolatban áll az osztálymérettel; ez utóbbiról majd az osztályok kapcsán szólunk – most foglalkozzunk csak a fájlmérettel.

Mekkorák általában a Java forrásfájlok? Ha megvizsgáljuk őket, láthatjuk, hogy komoly különbségek vannak köztük mind a méreteiket, mind a stílusukat tekintve. Erről árulkodik az 5.1. ábra is.

Hét projektet – Junit, FitNesse, testNGm, Time and Money, JDepend, Ant és Tomcat – vizsgáltunk. A téglalapokat metsző vonalak a legkisebb és a legnagyobb fájlméretet jelzik a projektben. Maguk a téglalapok durván a fájlok egyharmadát tartalmazzák (közelítőleg ennyi a standard deviáció[1] értéke). A téglalap közepe jelöli az átlagot. Így a fájlok átlagos

[1] A téglalap vízszintes oldalai az átlagértéktől szigma/2 távolságra helyezkednek el. Igen, jól tudom, hogy a fájlhosszak nem normál eloszlást követnek, így matematikailag megkérdőjelezhető, hogy a standard deviációval számolunk. Most azonban nem a pontosságot tartjuk szem előtt, mindössze általános képet szeretnénk kapni a helyzetről.

mérete a FitNesse-ben nagyjából 65 sor, és a fájlok durván egyharmada a 40–100+ soros tartományba esik. A FitNesse legnagyobb fájlja 400 sornál is hosszabb, a legkisebb pedig 6 soros. Vegyük észre, hogy a függvényértékeket logaritmikus skálán ábrázoltuk, így a függőleges helyzetben mutatkozó kis különbség is komoly méretbeli eltéréseket mutat.



5.1. ábra

A fájlhosszak eloszlása a logaritmikus skálán (téglalap magassága = standard deviáció)

Látható, hogy a Junit, a FitNesse, valamint a Time and Money viszonylag kis méretű fájlokból állnak. Egyikük sem nagyobb 500 sornál, sőt a legtöbbjük mérete 200 sor alatt marad. A Tomcat és az Ant egyes fájljai ugyanakkor több ezer sorból állnak, és a többi fájl is majdnem felerészben 200 sor feletti méretű.

Mit is jelent mindez a számunkra? Nos, a statisztikából kitűnik, hogy akár hatalmas rendszereket (a FitNesse összességében majdnem 50 000 sorból áll) is felépíthetünk jellemzően 200 sor körüli fájlokból, az 500 soros felső határt egyáltalán nem lépve túl. Jóllehet ezt nem tekinthetjük szigorú szabálynak, annyit azonban elmondhatunk, hogy érdemes ilyen rendszerek létrehozására törekednünk, hiszen a kisebb fájlok megértése könnyebb, mint nagyobb társaiké.

Az újság, mint hasonlat

Gondoljunk egy jól megírt újságcikkre. Nyilván függőleges irányban haladunk végig a tartalmán. A legtetején láthatjuk a címsort, amely alapján eldönthetjük, hogy egyáltalán belevágunk-e a cikk olvasásába. Az első bekezdés összefoglalót ad a cikkről, elrejtve a részleteket, hogy madártávlati képet kapjunk a történetről. Ezután, ahogy továbbhaladunk, úgy kerülnek felszínre a részletek: dátumok, nevek, idézetek, kijelentések és más hasonlók.

Azt szeretnénk, ha a forrásfájlokat is úgy olvashatnánk, mint az újságcikkeket. A fájlnév eszerint legyen egyszerű, de sokatmondó – ha rápillantunk, azonnal tudjuk, hogy a megfelelő modulban vagyunk-e. A forrásfájl tetején kell elhelyeznünk a legmagasabb szintek leírását – az alapvető elemeket és algoritmusokat. A részletek ez után egyre finomodnak, és a fájl végén találhatjuk meg a legalacsonyabb szintű függvényeket.

Az újságok rengeteg cikket tartalmaznak, amelyek többsége igen apró. Néhány kicsit hosszabb, de csak igen kevés van köztük, amely egy egész oldalt kitölt. Ettől lesz az újság használható. Gondoljuk csak el, kézbe vennénk egy olyan újságot, amely egyetlen, hosszú oldalakon át kígyózó történetből áll, összehányt tényanyaggal, itt-ott dátumokkal, nevekkel és adatokkal?

Az elgondolások függőleges elválasztása

Szinte minden kódot balról jobbra és felülről lefelé haladva olvasunk. A sorok kifejezéseket vagy záradékokat takarnak, a sorok csoportjai pedig teljes elgondolásokat. Ez utóbbiakat üres sorokkal világosan el kell választanunk egymástól.

Vegyük például az 5.1. példát. Üres sorok választják el a csomag meghatározását, a betöltéseket, valamint az egyes függvényeket. Igen egyszerű szabályról van szó, mégis megdöbbentően nagy hatással van a kódra. Minden üres sor képi jelzés, amely elválasztja egymástól az elgondolásokat. Ahogy felülről lefelé haladunk a kódban, a szemünk mindig az üres sorok alatti első soron akad meg.

5.1. példa

BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern =
Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );
  public BoldWidget(ParentWidget parent, String text) throws
                                                    Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Az 5.2. példában bemutatjuk, mennyivel átláthatatlanabb lesz a kód, ha kivesszük ezeket az üres sorokat.

5.2. példa

BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern =
Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws
Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

A hatás még erőteljesebb, ha nem a kódra fókuszálunk a szemünkkel. Az első példában a sorok csoportjai világosan különválnak, itt pedig egy igazi katyvaszt kapunk. A két kódpélda között mindössze a függőleges térközökben van különbség.

Függőleges sűrűség

Mivel a térközök elválasztják az elgondolásokat, a függőlegesen sűrű kódot összetartozónak érzékeljük. Azok a kódsorok tehát, amelyek szorosan összetartoznak, függőlegesen közel kerülnek egymáshoz. Figyeljük meg, miként törik meg a két példányváltozó szoros kapcsolatát a felesleges megjegyzések az 5.3. példában.

5.3. példa

```
public class ReporterConfig {
  /**
   * A jelentéskészítő figyelőosztály neve
   */
  private String m_className;
  /**
   * A jelentéskészítő figyelőosztály tulajdonságai
   */
  private List<Property> m_properties = new ArrayList<Property>();
  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Az 5.4. példa olvasása ugyanakkor sokkal könnyebb. Belefér az ember látóterébe – legalábbis az enyémbe mindenképpen. Első ránézésre világos, hogy itt egy osztályról van szó, amely két változót és egy függvényt tartalmaz – és ehhez nem igazán kell megmozdítani a fejemet vagy a szememet. Az előző kód megértéséhez ugyanakkor sokkal több szem- és fejmozgásra van szükség.

5.4. példa

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();
  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Függőleges távolság

Előfordult már, hogy kétségbeesetten ugráltunk egyik függvényről a másikra, keresztülkasul a forráskódon, hogy megpróbáljuk kideríteni, milyen kapcsolatban állnak egymással – az eredmény pedig csak még nagyobb tanácstalanság volt? Próbáltuk már nyomon követni egy változó vagy függvény meghatározásának öröklési láncát? Ha igen, érdemes észrevennünk, hogy valójában azért fáradtunk el, mert azt szerettük volna megtudni, hogy mit csinál a rendszer, miközben folyamatosan a „hol" kérdésre kellett válaszokat keresnünk a kódban és az emlékezetünkben egyaránt.

Az egymáshoz közel álló elgondolásoknak a kódban is közel kell elhelyezkedniük [G10]. Ez a szabály persze nem alkalmazható olyan elgondolásokra, amelyek különböző fájlokban kapnak helyet – az egymáshoz valóban közel állókat ugyanakkor nem is érdemes külön fájlban tárolnunk, hacsak valamilyen hathatós érv nem támasztja alá ezt a döntést. Ez az egyik oka annak, hogy miért érdemes kerülnünk a védett változók használatát.

Azoknak az elgondolásoknak az esetében, amelyek elég közel állnak egymáshoz ahhoz, hogy egy forrásfájlba kerüljenek, a függőleges távolságot annak megfelelően kell megválasztanunk, hogy mennyire lényegesek az adott kódrészletek egymás megértésében. Szeretnénk elejét venni annak, hogy az olvasóink össze-vissza ugráljanak a forrásfájlok és az osztályok között.

A változók bevezetése

A változókat a lehető legközelebb kell bevezetnünk a használatuk helyéhez. Mivel a függvényeink igen rövidek, a helyi változókat a tetejüknél kell bevezetnünk – ezt láthatjuk a Junit 4.3.1 alább bemutatott, a kelleténél talán kicsit hosszabb függvényében.

```
private static void readPreferences() {
  InputStream is= null;
  try {
```

```
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

A ciklusok vezérlőváltozóit jobbára a ciklusutasításban érdemes elhelyeznünk. Jól példázza ezt az alábbi aprócska függvény, amely szintén az előbb említett helyről származik:

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
    count += each.countTestCases();
  return count;
}
```

Ritkán előfordulhat, hogy egy változót egy blokk vagy egy ciklus előtt vezetünk be, amennyiben a függvény nem kifejezetten rövid. Egy ilyen helyzetet mutat az alábbi kódrészlet, amelyet a testNG egy igen hosszú függvényének közepéből ollóztunk ki:

```
...
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);
      invoker = tr.getInvoker();
      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }
      for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
        afterSuiteMethods.put(m.getMethod(), m);
      }
    }
...
```

Példányváltozók

Ezeknek a változóknak a szerepe indokolttá teszi, hogy az osztály kódjának tetején vezessük be őket. Ez nem növelheti e változók függőleges távolságát a felhasználásuk helyétől, ugyanis egy alkalmasan megtervezett osztályban számos, esetleg mindegyik tagfüggvény használja őket.

A példányváltozók helyéről rengeteg vita folyt. A C++-ban az úgynevezett *ollószabály* terjedt el széles körben, ami szerint a példányváltozók a kód aljára kell kerüljenek. A Javában ugyanakkor az osztály elején szokás elhelyezni őket, és nem látok különösebb okot arra, hogy másképpen tegyünk. A legfontosabb, hogy a példányváltozókat egy sokak által jól ismert helyen tároljuk, hiszen mindenkinek tudnia kell, hol nézhet utána a meghatározásuknak.

Vegyük például a Junit 4.3.1 `TestSuite` osztályának különös esetét. A lényeg kiemelése érdekében egyes részeket elhagytunk. Az így összetömörített kód közepe táján akadhatunk rá két példányváltozó meghatározására. Nos, igencsak össze kellene szednünk magunkat, ha ennél jobban el akarnánk rejteni őket! A meghatározásukra csak véletlenül, a kód átolvasása során bukkanhatunk rá – így jártam én is.

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
    ...
  }
  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
    ...
  }
  public static Test warning(final String message) {
    ...
  }
  private static String exceptionToString(Throwable t) {
    ...
  }
  private String fName;
  private Vector<Test> fTests= new Vector<Test>(10);
  public TestSuite() {
  }
   public TestSuite(final Class<? extends TestCase> theClass) {
    ...
  }
  public TestSuite(Class<? extends TestCase>  theClass, String name) {
    ...
  }
  ... ... ... ... ...
}
```

Függő függvények

Ha egy függvény meghív egy másikat, függőlegesen közel kell állniuk egymáshoz, méghozzá úgy, hogy amennyiben lehetséges, a hívó a hívott felett szerepeljen a kódban. Így biztosíthatjuk a program természetes folyását. Ha ezt a szabályt betartjuk, a kódunk olvasói biztosan tudhatják, hogy ha egy függvényt használunk valahol, kicsit tovább haladva

hamarosan megtalálhatják a meghatározását is. Vegyük például az 5.5. példában látható részletet, amely a FitNesse-ből származik. Figyeljük meg, miként hívja meg a legfelső függvény az alatta levőt, majd az szintén az alatta levő függvényt. Így tehát könnyen megtalálhatjuk a meghívott függvényeket, és jelentősen javul a modul olvashatósága.

5.5. példa

WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;
  public Response makeResponse(FitNesseContext context, Request
                                                        request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
     loadPage(pageName, context);
     if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }
  private String getPageNameOrDefault(Request request, String
                                                  defaultPageName)
  {
     String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;
     return pageName;
  }
  protected void loadPage(String resource, FitNesseContext context)
     throws Exception {
     WikiPagePath path = PathParser.parse(resource);
     crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
     if (page != null)
       pageData = page.getData();
  }
  private Response notFoundResponse(FitNesseContext context,
Request request)
     throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }
```

folytatódik

5.5. példa

WikiPageResponder.java – folytatás

```
    private SimpleResponse makePageResponse(FitNesseContext context)
        throws Exception {
      pageTitle = PathParser.render(crawler.getFullPath(page));
      String html = makeHtml(context);
      SimpleResponse response = new SimpleResponse();
      response.setMaxAge(0);
      response.setContent(html);
      return response;
    }
  ...
```

Ez a kódrészlet mellesleg arra is nagyszerű példát ad, hogy miként tarthatjuk az állandókat a megfelelő szinten [G35]. A `FrontPage` állandót eláshattuk volna a `getPageNameOrDefault` függvény mélyén, de így végeredményben egy jól ismert és elvárt állandót egy alacsonyszintű függvényben rejtettünk volna el. Több értelme van annak, ha az ilyen állandókat azon a helyen tüntetjük fel, ahol érdemes tudni, mire szolgálnak, és innen adjuk őket tovább oda, ahol fel is használjuk őket.

Fogalmi közelség

Bizonyos kódrészletek szeretnek egyes társaik mellé kerülni, hiszen fogalmilag közel állnak egymáshoz. Minél erősebb ez a kapcsolat, annál közelebb kell őket helyeznünk egymáshoz fizikailag is a kódban.

Láthattuk, hogy ez a közelség eredhet a közvetlen függésből, mint amikor egyik függvény meghívja a másikat, vagy a függvény egy változót használ. Ugyanakkor a kapcsolatnak más okai is lehetnek. Előfordulhat például, hogy a függvényeink azért tartoznak össze, mert hasonló műveletek elvégzésére alkalmasak. Vegyük például a Junit 4.3.1 alábbi részletét:

```
public class Assert {
  static public void assertTrue(String message, boolean condition) {
    if (!condition)
      fail(message);
  }
  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }
```

```
  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }
  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

Az itt látható függvények fogalmilag közel állnak egymáshoz, mert az elnevezésük ugyanazon az elven alapul, és egy alapművelet különböző módozatait valósítják meg. Az, hogy egymást hívják, csak hab a tortán – enélkül is összetartoznak.

Függőleges sorrend

Összességében arra törekszünk, hogy a függvényhívások függőségei lefelé mutassanak, vagyis a hívott függvény a hívó fél alatt kell legyen.[2] Így egy kellemesen átlátható kódfolyamot kapunk, ami a magas szintű kódtól az alacsony szintű felé áramlik.

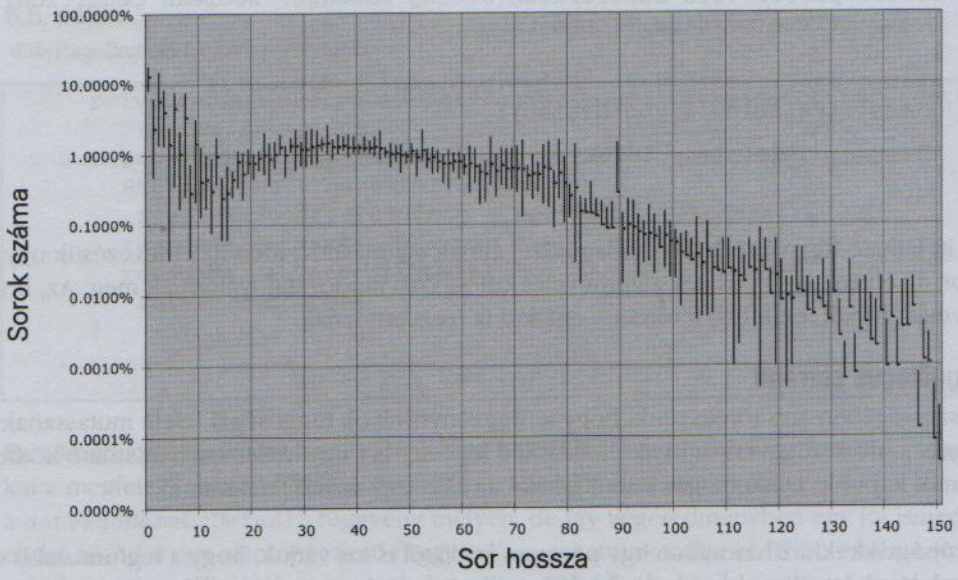
Az újságcikkekhez hasonlóan egy program kódjától is azt várjuk, hogy a legfontosabb elgondolások kerüljenek a szemünk elé legelőször, méghozzá mindenféle zavaró részletek nélkül. Az alacsony szintű apróságokra a kód végén számítunk. Így gyorsan áttekinthetjük a forrásfájlokat, átlátva a lényeget az első pár függvény alapján anélkül, hogy bele kellene bonyolódnunk a részletekbe. Az 5.5. példa nagyszerűen mutatja ezt az elvet, de a 15.5. és a 3.7. példa talán még jobban szemlélteti.

Vízszintes formázás

Milyen hosszú legyen egy sor? Ahhoz, hogy erre a kérdésre megkapjuk a választ, nézzük meg, hogy jellemzően milyen hosszúak a sorok más programokban! Térjünk vissza a korábban már megvizsgált hét projekthez – az eredményt, vagyis a sorok hosszának eloszlását az 5.2. ábra mutatja.

A szabályosság döbbenetes, különösképpen a 45 karakter körüli területen. Elmondhatjuk, hogy 20 és 60 karakter között nagyjából minden sorhossz egy százalékban jelenik meg az összes lehetséges hosszhoz képest. Ez 40 százalék! Talán még úgy 30 százalékot tesznek ki a 10 karakter alatti sorok. Ne feledjük, logaritmikus skálát használunk, azaz a 80 karakter feletti látszólag lineáris esés valójában jóval nagyobb ütemű. Vagyis: a programozók láthatóan a rövid sorokat kedvelik.

[2] Ez éppen ellentétes a Pascal, a C, a C++ és a hozzájuk hasonló nyelvek elvárásaival, miszerint a függvényeket még a használatuk előtt meg kell határozni, de legalábbis be kell vezetni.



5.2. ábra

Java-programok sorhosszainak eloszlása

Mindez azt sugallja a számunkra, hogy törekednünk kell a rövid sorok használatára. Hollerith 80 karakteres korlátja talán kicsit szigorú, tehát a 100, netán 120 karakteres sorok használata sem ördögtől való – ennél hosszabb sorok azonban már leginkább a programozó gondatlanságát mutatják.

Jómagam ahhoz a szabályhoz tartom magam, hogy soha ne kelljen jobbra görgetni a képernyő tartalmát. A monitorok azonban egyre szélesebbek, és a fiatal programozók olyannyira kis betűmérettel is beérik, hogy akár 200 karakter is elfér egy sorban. Idáig azonban ne menjünk el. Jómagam 120 karakterben húzom meg a határt.

Vízszintes térközök és sűrűség

A vízszintes térközök szerepe, hogy összekössék azokat a kódrészeket, amelyek összetartoznak, és szétválasszák azokat, amelyek nem. Vegyük a következő függvényt:

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Az értékadó műveleteket térközökkel emeltem ki. Az értékadások alapvetően két nagy egységből állnak: a bal és a jobb oldalból. A térközök ezeket a részeket teszik jobban elkülöníthetővé.

Másrészről viszont a függvénynevek és az utánuk álló nyitó zárójelek közé nem tettem térközöket. Ennek az az oka, hogy a függvény és paraméterei között szoros kapcsolat áll fenn. Ha elválasztanánk őket, az rontana ennek a kapcsolatnak a láthatóságán. A paraméterek között ugyanakkor ismét térközöket hagytam ki, amelyek jobban kiemelik a vesszőket, és egyúttal azt, hogy itt valóban különálló paraméterekről van szó.

A térközöket a műveleti sorrend kihangsúlyozására is használhatjuk:

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
     double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }
  public static double root2(int a, int b, int c) {
     double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }
  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Vegyük észre, milyen könnyen olvashatók az egyenletek. A tagok belsejében nincsenek térközök, hiszen ezek szoros egységeket képeznek. Az egyes tagok között viszont megjelennek a térközök, hiszen az összeadás és a kivonás alacsonyabb rangú a szorzásnál és az osztásnál.

Sajnos a kód újraformázását segítő eszközök többsége nem érzi a műveleti sorrendet, így mindenütt ugyanazokat a térközöket használják. Így a kód újraformázása során a fentihez hasonló térközkezelési finomságok lassacskán eltűnnek a kódból.

Vízszintes igazítás

Réges-régen, amikor gépi kódokat írtam[3] a vízszintes igazítással bizonyos kódszerkezeteket emeltem ki. Amikor később elkezdtem C, C++, majd Java nyelven dolgozni, továbbra is megpróbáltam a változók neveit egymás alá igazítani a meghatározásokban, vagy a jobb oldali értékeket az értékadásokban. A kód végül valahogy így nézhetett ki:

[3] Mit is beszélek? A lelkem mélyén ma is gépi kódban gondolkodom... Az ifjú titán soha sem hal meg bennem!

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket              socket;
  private   InputStream         input;
  private   OutputStream        output;
  private   Request             request;
  private   Response            response;
  private   FitNesseContext     context;
  protected long                requestParsingTimeLimit;
  private   long                requestProgress;
  private   long                requestParsingDeadline;
  private   boolean             hasError;
  public FitNesseExpediter(Socket           s,
                           FitNesseContext context) throws
Exception
  {
    this.context =              context;
    socket =                    s;
    input =                     s.getInputStream();
    output =                    s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Később azonban ráébredtem, hogy ennek az igazításnak nincs sok haszna. Nem a megfelelő dolgokat emeli ki, és eltereli a figyelmet a lényegről. A fenti meghatározásokat nézve az ember legszívesebben felülről lefelé végigolvasná a változók neveit, tudomást sem véve a típusaikról. Ehhez hasonlóan az értékadásoknál is leginkább a jobb oldali értékek oszlopával foglalkoznánk, nem figyelve az értékadó műveletekre. Ráadásul az automatikus újraformázási segédeszközök többnyire véget vetnek az ilyen típusú igazításnak.

Végeredményben tehát azt mondhatjuk, hogy az ilyesfajta igazításnak lejárt az ideje. Manapság jobban szeretem az alábbihoz hasonló, igazítás nélküli meghatározásokat és értékadásokat – annál is inkább, mert van egy hiányosság, amire rávilágíthatnak. Ha egy hosszú kóddal találkozunk, amelyre véleményünk szerint ráférne az igazítás, a *probléma magával a kóddal van*, nem pedig az igazítással. Az alábbi, `FitNesseExpediter` nevű osztály meghatározásainak listája olyan hosszú, hogy az osztály felbontásának szükségességét veti fel.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
private Response response;
private FitNesseContext context;
protected long requestParsingTimeLimit;
```

```
private long requestProgress;
private long requestParsingDeadline;
private boolean hasError;
public FitNesseExpediter(Socket s, FitNesseContext context) throws
Exception
{
  this.context = context;
  socket = s;
  input = s.getInputStream();
  output = s.getOutputStream();
  requestParsingTimeLimit = 10000;
}
```

Behúzás

A forrásfájl adatai vázlatszerű hierarchiába rendeződnek. Vannak adatok, amelyek a teljes fájlra vonatkoznak, mások a fájlban található osztályokra, az osztályok tagfüggvényeire, a tagfüggvények blokkjaira, a blokkokon belüli további blokkokra, és így tovább. A hierarchia minden szintje egy hatókört is jelent, amelyben azonosítókat határozhatunk meg, és amelyben a fordító a meghatározásokat és a végrehajtható utasításokat értelmezi.

A hierarchia szintjeinek láthatóvá tételére a forráskód sorait a szintjüknek megfelelő mértékben behúzzuk. A fájl szintjének utasításai – ilyen a legtöbb osztálymeghatározás – egyáltalán nem kapnak behúzást. Az osztályok tagfüggvényeit egy szinttel jobbra húzzuk be. Ennél egy szinttel beljebb következnek a tagfüggvények megvalósításai. A blokkok ismét egy szinttel beljebb kerülnek, a blokkokon belüli blokkok még egy szinttel, és így tovább.

A programozók komolyan támaszkodnak a behúzások rendszerére. A kód áttekintésekor megnézik, mely sorok bal széle esik egy vonalba, és ezeket helyezik gondolatban egy szintre. Így gyorsan átugorhatnak egyes hatóköröket, például `if` és a `while` blokkokat, amelyek pillanatnyilag nem érdeklik őket. Szintén a bal oldalt nézik, ha új tagfüggvények, új változók, sőt új osztályok meghatározásaira vadásznak. A behúzások nélkül a kód gyakorlatilag olvashatatlanná válna.

Jól mutatja mindezt a következő két program, amelyek a tartalmukat tekintve tökéletesen megegyeznek:

```
public class FitNesseServer implements SocketServer { private
FitNesseContext context; public FitNesseServer(FitNesseContext
context) { this.context = context; } public void serve(Socket s) {
serve(s, 10000); } public void serve(Socket s, long requestTimeout) {
try { FitNesseExpediter sender = new FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
-----
```

```
public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }
  public void serve(Socket s) {
    serve(s, 10000);
  }
  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

A szemünk pillanatok alatt befogadja a behúzott kód szerkezetét. Szinte azonnal kiszúrjuk a változókat, konstruktorokat, elérőket és tagfüggvényeket. Mindössze pár másodpercbe telik, amíg rájövünk, hogy egy csatoló egyszerű felületéről van szó, amely időkorlátot alkalmaz. A behúzások nélküli változat ugyanakkor tüzetes vizsgálat nélkül megfejthetetlen rejtvénynek tűnik.

A behúzási szabályok mellőzése

Olykor nagy a csábítás, hogy eltekintsünk a behúzásoktól rövid `if` vagy `while` utasításoknál, illetve apró függvényeknél. Jómagam, ahányszor csak engedtem ennek a csábításnak, később szinte mindig visszatértem a tett színhelyére, és visszaállítottam a behúzásokat. Vagyis inkább elkerülöm az alábbi tömörítéseket:

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";
  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Jobb szeretem a hatóköröket szélesre tárni, és behúzásokkal is jelezni:

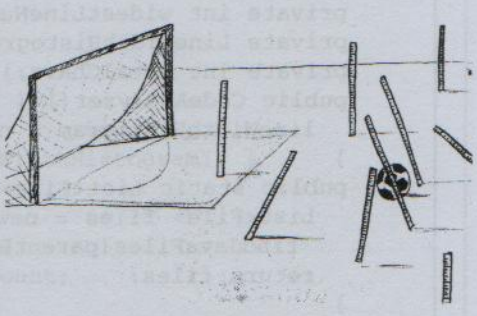
```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";
  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }
  public String render() throws Exception {
    return "";
  }
}
```

Üres hatókörök

Előfordul, hogy egy `while` vagy `for` utasítás hatóköre üres – erre látunk példát az alábbi kódban. Nem igazán kedvelem az ilyen kódszerkezeteket, így ha csak lehet, elkerülöm őket. Ha erre nincs lehetőség, mindenképpen ügyelek arra, hogy a hatókört jelezzem a kapcsos zárójelpárral és a megfelelő behúzásokkal. Nehéz lenne összeszámolni, hányszor csapott be a `while` utasítás sorában némán meglapuló pontosvessző. Ez az apró írásjel igencsak hajlamos elbújni, hacsak nem gondoskodunk arról, hogy láthatóvá tegyük:

```
while (dis.read(buf, 0, readBufferSize) != -1)
  ;
```

Csapatszabályok



Minden programozónak megvannak a maga formázási elvei, de ha csapatban dolgozunk, a csapat szabályai vonatkoznak ránk is. Fontos, hogy a fejlesztőcsapat megegyezzen a formázási szabályokban, és megkövetelje, hogy minden tagja betartsa azokat. Az a cél, hogy a program stílusán érződjön az összhang – ne tűnjön úgy, mintha a programozók nézeteltérési jelennének meg a kódban.

Amikor 2002-ben elindítottam a FitNesse projektet, leültem a csapattal, hogy egyeztessünk a kódolási stílusról. Nagyjából tíz percet beszélgettünk. Eldöntöttük, hogyan helyezzük el a kapcsos zárójeleket, mekkorák legyenek a behúzások, hogyan nevezzük el az osztályokat, a változókat, a tagfüggvényeket, és így tovább. Ez után rögzítettük a szabályokat a fejlesztőkörnyezet kódformázójában, és azóta is tartjuk magunkat hozzájuk. A szabályok nem feltétlenül az én ízlésem szerint alakultak, hanem a csapat közös akaratát tükrözték. A csapat tagjaként azonban én is betartottam őket, amikor csak a FitNesse projekt kódját írtam.

Ne feledjük, egy jó programrendszer valójában nem más, mint könnyen olvasható dokumentumok összessége. Fontos, hogy a stílusuk egységes és könnyedén befogadható legyen. Így az olvasó bízvást gondolhatja, hogy az egyik forrásfájlban alkalmazott formázási elvek visszaköszönnek a többiben is. Semmit sem szeretnénk jobban elkerülni, mint azt, hogy a forráskódot az egyéni stílusok tovább bonyolítsák.

Bob bácsi formázási szabályai

Az általam alkalmazott szabályok igen egyszerűek, amint azt az 5.6. példában is láthatjuk. Tekintsük ezt az osztályt olyan példának, ami jól mutatja, miként fektethetjük le egy kód segítségével a kódolási szabályainkat.

5.6. példa

CodeAnalyzer.java

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;
  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }
  public static List<File> findJavaFiles(File parentDirectory) {
    List<File> files = new ArrayList<File>();
     findJavaFiles(parentDirectory, files);
    return files;
  }
  private static void findJavaFiles(File parentDirectory,
List<File> files) {
    for (File file : parentDirectory.listFiles()) {
      if (file.getName().endsWith(".java"))
        files.add(file);
      else if (file.isDirectory())
        findJavaFiles(file, files);
    }
  }
  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new
FileReader(javaFile));
     String line;
     while ((line = br.readLine()) != null)
      measureLine(line);
  }
  private void measureLine(String line) {
     lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }
```

folytatódik

5.6. példa

CodeAnalyzer.java – folytatás

```
    private void recordWidestLine(int lineSize) {
        if (lineSize > maxLineWidth) {
         maxLineWidth = lineSize;
          widestLineNumber = lineCount;
        }
    }
    public int getLineCount() {
      return lineCount;
    }
    public int getMaxLineWidth() {
      return maxLineWidth;
    }
    public int getWidestLineNumber() {
      return widestLineNumber;
    }
    public LineWidthHistogram getLineWidthHistogram() {
        return lineWidthHistogram;
    }
    public double getMeanLineWidth() {
      return (double)totalChars/lineCount;
    }
    public int getMedianLineWidth() {
      Integer[] sortedWidths = getSortedWidths();
      int cumulativeLineCount = 0;
      for (int width : sortedWidths) {
        cumulativeLineCount += lineCountForWidth(width);
        if (cumulativeLineCount > lineCount/2)
          return width;
      }
      throw new Error("Cannot get here");
    }
    private int lineCountForWidth(int width) {
      return lineWidthHistogram.getLinesforWidth(width).size();
    }
    private Integer[] getSortedWidths() {
      Set<Integer> widths = lineWidthHistogram.getWidths();
      Integer[] sortedWidths = (widths.toArray(new Integer[0]));
       Arrays.sort(sortedWidths);
      return sortedWidths;
    }
}
```