9

Egységtesztek

A szakmánk nagyon sokat változott az utóbbi tíz évben. 1997-ben senki sem hallott még a tesztvezérelt fejlesztésről (TTD, Test Driven Development). Túlnyomó többségünk számára az egységtesztek rövid, egyszer használatos kódok voltak, amiket azért írtunk, hogy meggyőződjünk róla, hogy a programunk „működik". Az osztályainkat és tagfüggvényeinket aprólékosan kidolgoztuk, aztán gyorsan összeütöttünk valamit a tesztelésükre; ehhez jellemzően valamilyen egyszerű illesztőprogramra volt szükség, amely lehetővé tette, hogy kézi módszerrel kapcsolatba lépjünk az általunk írt programmal.

Emlékszem egy C++ nyelvű programra, amit egy beágyazott valósidejű rendszerhez írtam a 90-es évek közepén. A program egy egyszerű időzítő volt, az alábbi aláírással:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

A program egyszerű elven működött: a `Command` osztály `execute` tagfüggvénye végrehajtódott egy új szálban a megadott számú ezredmásodperc után. A gondot csak az jelentette, hogy ezt hogyan lehetne tesztelni.

Egy egyszerű illesztőprogramot kapartam össze, ami a billentyűzetet figyelte, és egy karakter beírásakor beütemezett egy parancsot, ami ugyanezt a karaktert öt másodperc múlva megismételte. Elzongoráztam egy ritmusos dallamot a billentyűzeten, és vártam, hogy a melódia öt másodperccel később megjelenjen a képernyőn:

„I...want-a-girl...just...like-the-girl-who-marr...ied...dear...old...dad."

A pontok beírása közben valójában dúdoltam is a dallamot, aztán újra, ahogy a pontok megjelentek a képernyőn. Ez volt a teszt. Miután láttam, hogy működik, és bemutattam a munkatársaimnak, a tesztkódot kidobtam.

Ahogy azonban említettem, a szakmánk alaposan megváltozott azóta. Manapság már olyan teszteket írok, amelyek a kód minden apró-cseprő elemét ellenőrzik, hogy a várt módon viselkednek-e. A programjaim kódját elszigetelem az operációs rendszertől, ahelyett, hogy egyszerűen meghívnám a szabványos időzítő függvényeket. Magam írom meg az időzítő függvényeket, hogy teljesen az irányításom alatt tarthassam az időt. Olyan parancsokat ütemezek be, amelyek logikai jelzőket állítanak be, majd előreléptetem az időt, és figyelem ezeket a jelzőket, hogy hamisról igazra váltanak-e, ahogy az időt a megfelelő értékre állítom.

Ha megvan a tesztek csomagja, amelyeken a programnak át kell mennie, meggyőződöm róla, hogy a teszteket más is kényelmesen futtathatja, akinek esetleg dolga akad a kóddal. Gondoskodom róla, hogy a tesztek a kóddal együtt ugyanabba a forráscsomagba kerüljenek.

Igen, sokat fejlődtünk – de még messzebb kell jutnunk. Az agilis és a tesztvezérelt fejlesztés mozgalma már sok programozót ösztönzött arra, hogy automatizált egységteszteket írjon, és még többen csatlakoznak hozzájuk nap mint nap. Az őrült rohanásban azonban, hogy a tesztelést felvegyék a mindennapi eszköztárukba, sok programozó megfeledkezett valamiről, ami még fontosabb: arról, hogy jó teszteket írjon.

A tesztvezérelt fejlesztés három törvénye

Ma már mindenki tudja, hogy a TDD arra kér minket, hogy először az egységteszteket írjuk meg, és csak az után a végleges („üzemi") kódot. Ez a szabály azonban csak a jéghegy csúcsa. Gondoljuk végig az alábbi három törvényt[1]:

> ***Első törvény:*** *Ne írjunk üzemi kódot, amíg nem írtunk meg egy kudarcot valló egységtesztet!*
> **Második törvény:** Az egységtesztből csak annyit írjunk meg, amennyi a kudarchoz elegendő, és a fordítás sikertelensége kudarcnak számít.

[1] Robert C. Martin: *Professionalism and Test-Driven Development* (Object Mentor, IEEE Software, 2007. május/június, Vol.24, No.3, 32–36. oldal), `http://doi.ieeecomputersociety.org/10.1109/MS.2007.85`.

Harmadik törvény: Az üzemi kódból csak annyit írjunk meg, amennyi a jelenleg kudarcot valló teszt sikeres teljesítéséhez szükséges!

Ez a három törvény egy olyan ciklusra kényszerít minket, ami körülbelül harminc másodpercig tart. A teszteket és az üzemi kódot *egyszerre* írjuk meg; a tesztek csupán néhány másodperccel előzik meg az üzemi kódot.

Ha így dolgozunk, minden nap tesztek tucatjait, havonta tesztek százait, évente pedig tesztek ezreit írhatjuk meg. Ha így dolgozunk, a tesztek lényegében a teljes üzemi kódot lefedik majd. A tesztcsomag puszta mérete – ami magának az üzemi kódnak a méretével vetekszik – önmagában súlyos kezelési problémákat vet fel.

A tesztek tisztán tartása

Néhány évvel ezelőtt egy olyan fejlesztőcsapat oktatására kértek fel, akik kifejezetten úgy döntöttek, hogy a tesztkódjaiknak *nem szabad* ugyanahhoz a minőséghez ragaszkodniuk, mint az üzemi kódnak. Engedélyt adtak tehát egymásnak, hogy az egységtesztekben megszegjék a szabályokat – a kulcsszó a „gyors és piszkos" volt. A változóknak nem kellett beszédes neveket adniuk, a tesztfüggvényeknek pedig nem kellett rövidnek és leíró jellegűnek lenniük. A tesztkódot nem kellett jól megtervezni és gondosan részekre bontani. Amíg a tesztkód működött, és lefedte az üzemi kódot, addig megfelelt.

Biztosak vannak az olvasók között olyanok, akiknek szimpatikus ez a döntés, talán mert réges-régen ők is olyan teszteket írtak, mint én ahhoz a bizonyos `Timer` osztályhoz. Az ilyen eldobható tesztektől hatalmas ugrás eljutni az automatizált egységtesztekig, ezért az általam oktatott csapathoz hasonlóan ők is úgy vélhetik, hogy a piszkos tesztek is jobbak a semminél.

*Amit azonban ez a csapat nem ismert fel, az az, hogy a piszkos tesztek egyenértékűek az-*zal – ha nem rosszabbak nála –, mint ha egyáltalán nem rendelkeznénk tesztekkel. A problémát az jelenti, hogy a teszteknek az üzemi kód fejlődésével párhuzamosan változniuk kell. Csakhogy minél piszkosabb egy teszt, annál nehezebb módosítani. Minél kuszább a tesztkód, annál valószínűbb, hogy több időt fogunk azzal tölteni, hogy az új teszteket belegyömöszöljük a csomagba, mint az új üzemi kód megírásával. Ahogy az üzemi kódot módosítjuk, a régi tesztek elkezdenek kudarcot vallani, és a tesztkód rendetlensége megnehezíti, hogy ismét működésre bírjuk őket – így aztán a tesztek egyre nagyobb teherré válnak.

A csapatom tesztcsomagjának fenntartási költségei kiadásról kiadásra nőttek, míg végül a fejlesztők legnagyobb problémáját a tesztek kezdték jelenteni. Amikor a vezetők megkérdezték őket, hogy miért ilyen nagyok a becsült költségek, a fejlesztők a teszteket okolták. Végül arra kényszerültek, hogy teljesen elvessék a teszteket.

A tesztcsomag nélkül azonban arra sem voltak képesek, hogy megbizonyosodjanak róla, hogy a kód az elvárt módon működik. Tesztcsomag nélkül nem ellenőrizhették, hogy a rendszer egyik részének módosítása nem teszi-e működésképtelenné a rendszer más részeit. Így aztán a hibaszázalék is nőni kezdett. Ahogy a hibák száma nőtt, egyre kevésbé mertek változtatni. Többé nem tisztították ki az üzemi kódot, mert attól féltek, hogy a módosítások több kárt okoznának, mint amennyi hasznot hajtanak. Az üzemi kód pedig elkezdett szétmállani. Végül tesztek nélkül maradtak, a nyakukon egy zavaros, hibáktól hemzsegő üzemi kóddal, bosszankodó megrendelőkkel, és azzal az érzéssel, hogy belebuktak a tesztelésre tett erőfeszítéseikbe.

Bizonyos szempontból igazuk volt. A tesztelésbe *tényleg* belebuktak. Csakhogy az ő döntésük volt, hogy megengedik a tesztekben az összevisszaságot, és ez a döntés hintette el a bukás magját. Ha a tesztjeiket tisztán tartották volna, a tesztelés nem vallott volna kudarcot. Ezt bizton állíthatom, egyrészt mert ott voltam, másrészt mert számos csapatot oktattam, akik *tiszta* egységtesztekkel sikert értek el.

A története tanulsága egyszerű: a *tesztkód ugyanolyan fontos, mint az üzemi kód.* A tesztkód nem másodosztályú állampolgár – megfontolást, tervezést és gondosságot igényel, és ugyanolyan tisztán kell tartani, mint az üzemi kódot.

A tesztek tesznek képessé

Ha nem tartjuk tisztán a tesztjeinket, elveszítjük őket, nélkülük pedig elveszítjük azt, ami az üzemi kódot rugalmassá teszi. Igen, jól olvastuk. Az *egységtesztek* teszik rugalmassá, karbantarthatóvá és újrahasznosíthatóvá a kódunkat. Ennek egyszerű oka van: ha vannak tesztjeink, nem félünk változtatni a kódon. Tesztek nélkül minden módosítás egy lehetséges hibát jelent. Nem számít, milyen rugalmas a programunk felépítése, vagy hogy milyen szépen osztottuk részekre, tesztek nélkül nem merünk változtatásokat eszközölni rajta, mert attól félünk, hogy véletlenül rejtett hibákat csempészünk a kódba.

Tesztekkel azonban ez a félelem lényegében semmivé foszlik. Minél nagyobb területet fednek le a tesztek, annál kevésbé félünk. Nyugodtan módosíthatunk egy kevésbé csodás felépítésű, kusza és átláthatatlan programot is – sőt félelem nélkül *javíthatunk* a szerkezetén.

Ahhoz, hogy képesek legyünk a programunk felépítését a lehető legtisztábban tartani, az üzemi kódot lefedő automatizált egységtesztek jelentik a kulcsot. A tesznek tesznek képessé minket mindenre, mert lehetővé teszik, hogy *változtassunk.*

Tehát ha a tesztjeink piszkosak, akkor a kód módosítására kevésbé leszünk képesek, és ezzel elveszítjük annak a lehetőségét is, hogy javítsunk a kód szerkezetén. Minél piszkosabbak a tesztjeink, annál piszkosabb lesz a kódunk is, végül pedig elveszítjük a teszteket, és a kód rothadásnak indul.

Tiszta tesztek

Mi tesz egy tesztet tisztává? Három dolog: az olvashatóság, az olvashatóság és az olvashatóság. Az egységtesztek esetében az olvashatóság talán még fontosabb, mint az üzemi kódban. Mi tesz egy tesztet jól olvashatóvá? Ugyanazok a dolgok, mint bármely más kódot: az egyértelműség, az egyszerűség és a kifejezés tömörsége. Egy tesztben a lehető legtöbbet kell kérdeznünk a lehető legkevesebb kifejezéssel.

Vegyük a 9.1. példában látható kódot, amely a FitNesse-től származik. Ez a három teszt nehezen érthető, és biztosan lehetne javítani rajtuk. Először is, rengeteg az ismétlődő kód [G5] az `addPage` és az `assertSubString` ismételt hívásaiban. Ami azonban még fontosabb: a kód tömve van olyan részletekkel, amelyek csökkentik a teszt kifejezőkészségét.

9.1. példa

SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
   crawler.addPage(root, PathParser.parse("PageOne"));
   crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
   crawler.addPage(root, PathParser.parse("PageTwo"));
   request.setResource("root");
   request.addInput("type", "pages");
   Responder responder = new SerializedPageResponder();
   SimpleResponse response =
     (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
   String xml = response.getContent();
   assertEquals("text/xml", response.getContentType());
   assertSubString("<name>PageOne</name>", xml);
   assertSubString("<name>PageTwo</name>", xml);
   assertSubString("<name>ChildOne</name>", xml);
}
public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
   WikiPage pageOne = crawler.addPage(root,
                                      PathParser.parse("PageOne"));
   crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
   crawler.addPage(root, PathParser.parse("PageTwo"));
   PageData data = pageOne.getData();
   WikiPageProperties properties = data.getProperties();
   WikiPageProperty symLinks =
                        properties.set(SymbolicPage.PROPERTY_NAME);
```

folytatódik

9.1. példa

SerializedPageResponderTest.java – folytatás

```
        symLinks.set("SymPage", "PageTwo");
        pageOne.commit(data);
        request.setResource("root");
        request.addInput("type", "pages");
        Responder responder = new SerializedPageResponder();
        SimpleResponse response =
          (SimpleResponse) responder.makeResponse(
             new FitNesseContext(root), request);
        String xml = response.getContent();
        assertEquals("text/xml", response.getContentType());
        assertSubString("<name>PageOne</name>", xml);
        assertSubString("<name>PageTwo</name>", xml);
        assertSubString("<name>ChildOne</name>", xml);
        assertNotSubString("SymPage", xml);
    }

    public void testGetDataAsHtml() throws Exception
    {
        crawler.addPage(root, PathParser.parse("TestPageOne"),
                                                    "test page");
        request.setResource("TestPageOne");
        request.addInput("type", "data");
      Responder responder = new SerializedPageResponder();
      SimpleResponse response =
         (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
      String xml = response.getContent();
      assertEquals("text/xml", response.getContentType());
      assertSubString("test page", xml);
      assertSubString("<Test", xml);
    }
```

Nézzük meg például a `PathParser` hívásokat, amelyek karakterláncokat alakítanak át `PagePath`-példányokká, amelyeket a keresőrobotok használnak. Ez az átalakítás teljesen érdektelen a jelen teszt számára, és csak arra jó, hogy elfedje a szándékunkat.

A `responder` létrehozásának, valamint a `response` begyűjtésének és átalakításának részletei ugyancsak nem többek zajnál. Aztán ott van még a kérelem URL-jének bumfordi felépítése egy `resource` objektumból és egy paraméterből. (Jómagam is segítettem a kód megírásában, ezért nyugodtan kritizálhatom.)

Összefoglalva, ezt a kódot nem olvasásra tervezték. A kód szerencsétlen olvasóját részletek tömege árasztja el, amelyeket meg kell értenie, mielőtt a tesztek értelme kibontakozna.

Most vegyük szemügyre a tesztek javított változatát, amelyet a 9.2. példa mutat be. Ezek a tesztek pontosan ugyanazt csinálják, de az újratervezésnek köszönhetően sokkal tisztább és érthetőbb formában.

9.2. példa

SerializedPageResponderTest.java (újratervezve)

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");
  submitRequest("root", "type:pages");
  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>",
                                    "<name>ChildOne</name>"
  );
}
public void testSymbolicLinksAreNotInXmlPageHierarchy()
                                                  throws Exception {
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");
  addLinkTo(page, "PageTwo", "SymPage");
  submitRequest("root", "type:pages");
  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>",
                                    "<name>ChildOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}
public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");
  submitRequest("TestPageOne", "type:data");
  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

A felépítés–művelet–ellenőrzés (BUILD-OPERATE-CHECK) minta[2] nyilvánvaló a fenti tesztek szerkezetében. Mindegyik teszt világosan három részre oszlik. Az első rész felépíti a tesztadatokat, a második valamilyen műveletet végez rajtuk, a harmadik pedig ellenőrzi, hogy a művelet a várt eredményt adta-e.

Vegyük észre, hogy a zavaró részletek túlnyomó többségétől megszabadultunk. A tesztek lényegre törőek, és csak azokat az adattípusokat és függvényeket használják, amelyekre valóban szükségük van. Bárki, aki ezeket a teszteket olvassa, nagyon gyorsan képes átlátni, hogy mit csinálnak, anélkül, hogy tévútra tévedne, vagy agyonnyomnák a részletek.

[2] http://fitnesse.org/FitNesse.AcceptanceTestPatterns

Tartományfüggő tesztnyelv

A 9.2. példában látott tesztek azt az eljárást szemléltetik, amelynek során egy tartományfüggő nyelvet építünk fel a tesztjeink számára. Ahelyett, hogy azokat az alkalmazásprogramozási felületeket használnánk, amelyekre a programozók a rendszeren végrehajtott műveletek során támaszkodnak, olyan függvényeket és segédeljárásokat építünk fel, amelyek ezeket az API-kat veszik igénybe, ami kényelmesebben megírhatóvá és könnyebben olvashatóvá teszi a teszteket. Egy olyan *tesztnyelvet* hozunk tehát létre, amely segít a programozónak a tesztek megírásában, illetve azoknak, akik később olvasniuk kell a teszteket.

A tesztelési felületet nem előre tervezzük meg, hanem az olyan tesztkódok folyamatos újratervezése során fejlődik ki, amelyek túl zavarossá váltak a céljukat elfedő részletek miatt. A fegyelmezett fejlesztők ugyanúgy hozzák tömörebb és kifejezőbb formára a tesztkódjaikat, ahogy újratervezéssel mi is előállítottuk a 9.2. példa tesztjeit a 9.1. példa tesztjeiből.

A kettős mérce

A fejezet elején említett fejlesztőcsapatnak egy dologban igaza volt. A tesztelési API kódjára *valóban* más tervezési szabályok vonatkoznak, mint az üzemi kódra. Az egyszerűség, a tömörség és a kifejezőerő a tesztek esetében ugyanúgy követelmény, de nem kell olyan hatékonynak lenniük, hiszen nem üzemi, hanem tesztkörnyezetben futnak, a két környezetnek pedig jelentősen eltérő igényei vannak.

Vegyük a 9.3. példában látható tesztet. Ezt a tesztet egy környezetszabályozó rendszer részeként írtam, prototípus-készítés közben. Anélkül, hogy belemennék a részletekbe, annyit mondhatok, hogy a teszt azt ellenőrzi, hogy az alacsony hőmérsékleti riasztás, a fűtőtest és a ventilátor egyaránt bekapcsol-e, amikor a hőmérséklet szerint „túl hideg van".

9.3. példa

EnvironmentControllerTest.java

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

Temészetesen itt nagyon sok a részlet. Például mit csinál a `tic` függvény? Inkább ne gondolkozzunk rajta, miközben a tesztet böngésszük. Foglalkozzunk inkább csak azzal, hogy egyetértünk-e vele, hogy a rendszer végállapota megegyezik a „túl hideg" hőmérsékletértékkel.

Figyeljük meg, hogy a teszt olvasása közben a szemünknek oda-vissza kell ugrálnia az éppen ellenőrzött állapot neve és az állapot *jelentése* között. Látjuk a `heaterState` állapotot, aztán a szemünk balra, az `assertTrue` felé fordul. Látjuk a `coolerState` elemet, és a szemünknek vissza kell térnie balra az `assertFalse`-hoz. Ez fárasztó és megbízhatatlan, a tesztet pedig nehezen olvashatóvá teszi.

A 9.4. példa azt mutatja, hogyan javítottam jelentősen a teszt olvashatóságán.

9.4. példa

EnvironmentControllerTest.java (újratervezve)

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Természetesen elrejtettem a `tic` függvényt a `wayTooCold` („túl hideg") függvény létrehozásával, de az igazán érdekes dolog a különös karakterlánc az `assertEquals` függvényben. A nagybetű azt jelenti, hogy „bekapcsolva", a kisbetű azt, hogy „kikapcsolva", a betűk pedig a következőket jelentik, mindig azonos sorrendben: `{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}` (fűtőtest, ventilátor, hűtő, magas hőmérsékleti riasztás, alacsony hőmérsékleti riasztás).

Bár ez a megoldás majdnem megsérti a fejtörők kerülésére vonatkozó szabályt[3], ebben az esetben helyénvalónak tűnik. Vegyük észre, hogy ha már ismerjük a jelentését, a szemünk átsiklik az említett karakterláncon, és gyorsan az eredmény értelmezésére térhetünk. A teszt olvasása szinte élvezetessé válik.

Vessünk egy pillantást a 9.5. példára, és figyeljük meg, milyen könnyen érthetőek ezek a tesztek.

[3] 2. fejezet, „Kerüljük a fejtörőket!".

9.5. példa

EnvironmentControllerTest.java (hosszabb részlet)

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
  tooHot();
  assertEquals("hBChl", hw.getState());
}
@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
  tooCold();
  assertEquals("HBchl", hw.getState());
}
@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
  wayTooHot();
  assertEquals("hBCHl", hw.getState());
}
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
  wayTooCold();
  assertEquals("HBchL", hw.getState());
}
```

A `getState` („állapot kiolvasása") függvényt a 9.6. példa mutatja. Megjegyzendő, hogy a kódja nem túl hatékony. Ahhoz, hogy hatékonyabb legyen, valószínűleg egy `StringBuffer`-t kellett volna használnom.

9.6. példa

MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

A `StringBuffer`-ek azonban egy kissé csúnyák, ezért még a végleges kódokban is igyekszem kerülni a használatukat, amennyiben nem kell nagy árat fizetni érte – márpedig a 9.6. példa kódjának költsége igen kicsi. Világos ugyanakkor, hogy az alkalmazás egy beágyazott valósidejű rendszer, így valószínű, hogy a számítógép- és memória-erőforrások igencsak korlátozottak. A *tesztkörnyezet* azonban egyáltalán nem muszáj, hogy az legyen.

Ebből adódik a kettős mérce: vannak dolgok, amiket soha nem tennénk egy üzemi környezetben, de a tesztkörnyezetben tökéletesen megfelelnek. Ez általában a memória- vagy a processzor-hatékonysággal kapcsolatos, de a tisztasághoz *soha* nincs köze.

Egy teszt – egy állítás

Létezik egy iskola, amely szerint egy JUnit tesztben minden tesztfüggvény csak egyetlen megerősítésre váró állítást (assert) fogalmazhat meg[4]. Ez a szabály drákóinak tűnhet, de az előnye kitűnik a 9.5. példából. Az ott szereplő tesztek egyetlen következtetésre jutnak, amelyet gyorsan és könnyen fel lehet fogni.

De mi a helyzet a 9.2. példával? Nem tűnik könnyen kivitelezhetőnek, hogy valahogy összeolvasszuk azt a feltételezést, hogy a kimenet XML, azzal, hogy megtalálhatók benne bizonyos rész-karakterláncok. A tesztet ugyanakkor két külön tesztre bonthatjuk, amelyek egyetlen saját állítást fogalmaznak meg (lásd a 9.7. példát).

9.7. példa

SerializedPageResponderTest.java (egyetlen állítás)

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldContain(
      "<name>PageOne</name>", "<name>PageTwo</name>",
                                "<name>ChildOne</name>"
    );
}
```

Figyeljük meg, hogy a függvények nevét megváltoztattuk, hogy a szokványos given-when-then[5] (adott–ha–akkor) jelölést kövessék, ami még könnyebben olvashatóvá teszi a teszteket. Sajnos a tesztek fenti felbontása sok ismétlődő kódot eredményez.

Az ismétlődést a *sablonfüggvény* mintával[6] küszöbölhetjük ki, a *given/when* részeket az alaposztályba, a *then* részeket pedig különböző származtatott osztályokba helyezve. Létrehozhatunk egy teljesen önálló tesztosztályt is, és a *given*, illetve *when* részeket

[4] Lásd Dave Astel webnapló-bejegyzését a `http://www.artima.com/weblogs/viewpost.jsp?thread=35578` címen.

[5] [RSpec]

[6] [GOF]

a `@Before` függvénybe tehetjük, a *then* részeket pedig az egyes `@Test` függvényekbe, de ez már túlzás lenne egy ilyen aprósághoz. Én végül a 9.2. példában szereplő több állítást részesítettem előnyben.

Úgy vélem, hogy az „egyetlen állítás" szabálya jó iránytű.[7] Én általában olyan tartomány-függő tesztnyelvet igyekszem létrehozni, ami támogatja (ahogy a 9.5. példában tettem), de nem félek attól sem, hogy egy tesztbe egy állításnál többet tegyek. Valószínűleg az a legtöbb, amit mondhatunk, hogy egy tesztben az állítások számát a lehető legkevesebbre célszerű csökkenteni.

Egy teszt – egy elem

Ennél talán jobb szabály, hogy minden tesztfüggvényben csak egyetlen elemet teszteljünk. Nincs értelme olyan hosszú tesztfüggvényeket írni, amelyek egymás után a legkülönfélébb dolgokat vizsgálják. A 9.8. példában egy ilyen tesztre láthatunk példát. Ezt a tesztet három önálló tesztre kellene felbontani, mert három különböző dolgot vizsgál. Ha ezeket egyetlen függvénybe olvasztjuk, arra kényszerítjük az olvasót, hogy találja ki, minek a vizsgálatára szolgálnak az egyes részek, és miért vannak ott.

9.8. példa

SerializedPageResponderTest.java (egyetlen állítás)

```
    /**
     * Különféle tesztek az addMonths() tagfüggvényhez
     */
    public void testAddMonths() {
        SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

        SerialDate d2 = SerialDate.addMonths(1, d1);
        assertEquals(30, d2.getDayOfMonth());
        assertEquals(6, d2.getMonth());
        assertEquals(2004, d2.getYYYY());
        SerialDate d3 = SerialDate.addMonths(2, d1);
        assertEquals(31, d3.getDayOfMonth());
        assertEquals(7, d3.getMonth());
        assertEquals(2004, d3.getYYYY());
        SerialDate d4 = SerialDate.addMonths(1,
                                    SerialDate.addMonths(1, d1));
        assertEquals(30, d4.getDayOfMonth());
        assertEquals(7, d4.getMonth());
        assertEquals(2004, d4.getYYYY());
    }
```

[7] „Tarts a kód felé!"

A három tesztfüggvénynek valószínűleg az alábbiakat kellene vizsgálnia:

- Ha egy 31 napos hónap (például a május) utolsó napja *adott*:
  1. Ha a dátumhoz egy hónapot adunk, és az új hónap utolsó napja a 30. (mint júniusban), *akkor* a dátum a hónap 30. és nem a 31. napja kell legyen.
  2. Ha a dátumhoz két hónapot adunk, és az új hónap 31 napos, *akkor* a dátum a hónap 31. napja kell legyen.
- Ha egy 30 napos hónap (például a június) utolsó napja *adott*:
  1. Ha a dátumhoz két hónapot adunk, és az új hónap 31 napos, *akkor* a dátum a hónap 30. és nem a 31. napja kell legyen.

Így megfogalmazva láthatjuk, hogy a különböző tesztek mögött egy általános szabály rejtőzik: amikor növeljük a hónapot, a dátum nem lehet nagyobb az eredeti hónap utolsó napjánál. Ez azt jelenti persze, hogy február 28-ról egy hónappal előre lépve március 28-at kell kapnunk – *ennek* a vizsgálata hiányzik, és a tesztet hasznos lehet megírni.

A 9.8. példa egyes szakaszaiban tehát nem a több állítás okozza a gondot, hanem az a tény, hogy egy elemnél többet vizsgálunk. A legjobb szabály tehát valószínűleg az, hogy csökkentsük a lehető legkevesebbre az elemenkénti állítások számát, és egy tesztfüggvényben csak egyetlen elemet teszteljünk.

F.I.R.S.T.[8]

Egy tiszta teszt öt további jellemzővel rendelkezik, amelyek a fenti betűszót adják ki:

Fast (Gyors): A teszteknek gyorsnak kell lenniük, és gyorsan kell lefutniuk. Ha a tesztek lassúak, nem szívesen futtatjuk őket gyakran, ha pedig nem futtatjuk a teszteket gyakran, nem fogjuk elég korán megtalálni a hibákat ahhoz, hogy könnyen kijavíthassuk őket, ezért nem merjük majd eléggé kitisztítani a kódot, és a kód végül rothadni kezd.

Independent (Független): A teszteknek nem szabad egymástól függniük, és nem támaszthatnak feltételeket a következő teszt számára. Képesnek kell lennünk rá, hogy minden tesztet önállóan futtassunk, mégpedig tetszőleges sorrendben. Ha a tesztek egymásra támaszkodnak, akkor az első kudarcot valló teszt kudarchullámot indít el, ami megnehezíti a hiba okának a felderítését, ráadásul elfedheti a sorban később következő hibákat.

Repeatable (Megismételhető): A teszteknek bármely környezetben megismételhetőnek kell lenniük: képesnek kell lennünk a tesztek futtatására mind az üzemi, mind a minőségbiztosítási környezetben, és a laptopunkon is, miközben a hálózattól elszakadva vonatozunk hazafelé. Ha a tesztjeink nem ismételhetők meg tetszőleges

[8] Object Mentor Training Materials (Object Mentor-oktatóanyagok).

környezetben, akkor mindig lesz rá mentség, hogy miért vallottak kudarcot, ráadásul a teszteket nem tudjuk majd futtatni, ha a szükséges környezet nem áll rendelkezésre.

Self-validating (Egyértelmű): A teszteknek logikai eredményt kell adniuk: sikert vagy kudarcot kell jelezniük. Nem szabad, hogy egy naplófájlt kelljen végigolvasnunk ahhoz, hogy megmondhassuk, hogy a tesztek sikerrel jártak-e. Nem szabad, hogy kézi módszerrel két különböző szövegfájlt kelljen összehasonlítanunk, hogy megállapítsuk a teszt eredményét. Ha a tesztek nem egyértelműek, akkor a kudarc viszonylagossá válik, és a tesztek futtatása hosszas értékelést vonhat maga után.

Timely (Időszerű): A teszteket a megfelelő időben kell megírni: az egységteszteket *közvetlenül az előtt* az üzemi kód előtt, amelynek teljesítenie kell őket. Ha a teszteket az üzemi kód elkészítése után írjuk meg, akkor előfordulhat, hogy a kódot nehezen lehet majd ellenőrizni, ezért úgy döntünk, hogy egyes kódrészek túl nehezen tesztelhetők, és esetleg nem is úgy tervezzük meg a kódot, hogy tesztelhető legyen.

Összefoglalás

Ennek a témakörnek éppen csak megkapargattuk a felszínét – a *tiszta tesztekről* valójában egy teljes könyvet lehetne írni. A tesztek ugyanolyan fontosak egy program egészsége szempontjából, mint a végleges („üzemi") kód, sőt talán még fontosabbak, mert a tesztek őrzik meg és növelik az üzemi kód rugalmasságát, karbantarthatóságát és újrahasznosíthatóságát. A tesztjeinket tehát tartsuk mindig tisztán, törekedjünk rá, hogy kifejezők és tömörek legyenek, és hozzunk létre tesztelési API-t, amely a tesztek megírását segítő tartományfüggő nyelvként szolgál. Ha hagyjuk szétmállani a tesztjeinket, a kód is erre a sorsra fog jutni. A tesztjeink legyenek tiszták!

Irodalomjegyzék

[RSpec]: Aslak Hellesry, David Chelimsky: *RSpec: Behavior Driven Development for Ruby Programmers*, Pragmatic Bookshelf, 2008.

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