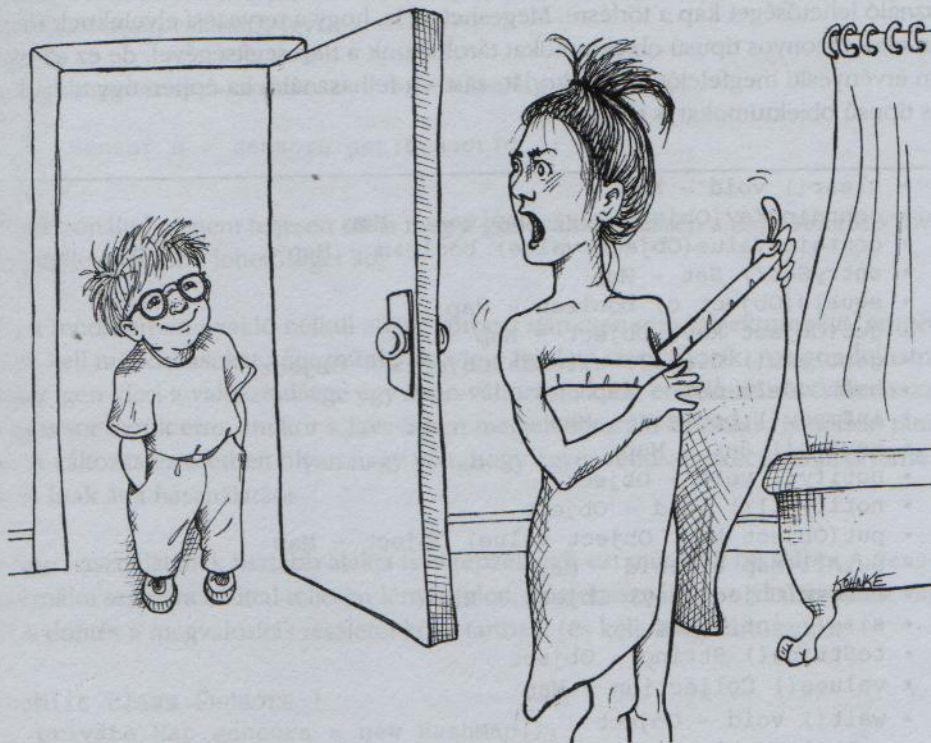


Határok

James Grenning




Ritkán fordul elő, hogy teljes egészében az ellenőrzésünk alatt tartjuk a rendszerünk összes programját. Olykor szükségünk van külső programokra vagy nyílt forrású alkalmazásokra, de a saját cégünkön belül is külön csapatokra kell támaszkodnunk, amelyek elkészítik a megfelelő összetevőket vagy alrendszereket. A cél tehát az, hogy ezt az idegen kódot valahogy beolvasszuk a rendszerünkbe. Ebben a fejezetben olyan módszereket és eljárásokat ismerünk meg, amelyekkel a programjaink határait tisztán tarthatjuk.

Külső kódok használata

A felületek készítői és felhasználói között létezik egyfajta természetes ellentét. A külső programcsomagok és környezetek gyártói a széles körű felhasználhatóságot tartják szem előtt, hogy minél több rendszerben működhessenek, és minél szélesebb közönséget nyerjenek meg maguknak. A felhasználók ugyanakkor olyan felületet szeretnének, amelyet kifejezetten az ő igényeikhez szabtak. Ez az ellentét gondokat okozhat a rendszereink határainál.

Példaképpen nézzük meg a `java.util.Map` osztályt. A 8.1. ábrára tekintve láthatjuk, hogy a `Map` objektumok igen széles felületet biztosítanak, rengeteg lehetőséggel. Ez a komoly tudás és rugalmasság persze hasznos, de egyúttal felelősséggel is jár. Az alkalmazásunk megteheti például, hogy felépít egy `Map` objektumot, és körbeadja. Lehet, hogy a szándékaink szerint a fogadó felek egyikének sem lenne szabad törölnie az objektum tartalmát, de a tagfüggvények listájának elején ott virít a `clear()`, amellyel bármely felhasználó lehetőséget kap a törlésre. Megeshet az is, hogy a tervezési elveinknek megfelelően csak bizonyos típusú objektumokat tárolhatunk a `Map` segítségével, de ez az osztály nem érvényesíti megfelelően ezt a korlátozást – a felhasználó, ha éppen úgy tartja kedve, más típusú objektumokat is tárolhat itt.



- `clear()` void - Map
- `containsKey(Object key)` boolean - Map
- `containsValue(Object value)` boolean - Map
- `entrySet()` Set - Map
- `equals(Object o)` boolean - Map
- `get(Object key)` Object - Map
- `getClass()` Class<? extends Object> - Object
- `hashCode()` int - Map
- `isEmpty()` boolean - Map
- `keySet()` Set - Map
- `notify()` void - Object
- `notifyAll()` void - Object
- `put(Object key, Object value)` Object - Map
- `putAll(Map t)` void - Map
- `remove(Object key)` Object - Map
- `size()` int - Map
- `toString()` String - Object
- `values()` Collection - Map
- `wait()` void - Object
- `wait(long timeout)` void - Object
- `wait(long timeout, int nanos)` void - Object

8.1. ábra

A `Map` osztály tagfüggvényei

Ha az alkalmazásunknak szüksége van egy Sensor objektumokat tartalmazó Map-re, elképzelhető, hogy így hozza létre azokat:

```
Map sensors = new HashMap();
```

Ezután, a kód egy másik részében, ahol e szenzorok valamelyikét el szeretnénk érni, ezt a kódot találjuk:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

Ez nem kivételes eset – ilyen kóddal számos helyen találkozhatunk. A kód felhasználójának felelőssége, hogy egy Object típusú objektumot szerezzon a Map-től, és ezt a megfelelő típusúvá alakítsa. Persze ez működő megoldás, de nem a legszebb, ráadásul a kód tisztább módon is elmesélhetné a „történetét”. Általánosítással jelentősen javíthatnánk az olvashatóságát, valahogy így:

```
Map<Sensor> sensors = new HashMap<Sensor>();
```

```
...
Sensor s = sensors.get(sensorId );
```

Mindazonáltal ez nem teljesen oldja meg a gondjainkat, hiszen a Map<Sensor> továbbra is a kelleténél több lehetőséget ad.

Ha a rendszerben nyaklő nélkül adjuk körbe a Map<Sensor> objektumokat, rengeteg helyen kell módosításokat végeznünk, ha a Map felülete megváltozik. Azt gondolhatnánk, hogy igen kicsi a valószínűsége egy ilyen változásnak, de emlékezzünk csak vissza, hogy mégis sor került erre, amikor a Java 5-ben megjelent az általánosítás (generics) támogatása. A változás ez esetben olyan nagy volt, hogy egyes rendszereket az átállás terhe miatt nem írtak át a használatára.

A Map használatának tisztább alakja is elképzelhető; ezt mutatjuk be alább. A Sensors felhasználói számára ezúttal teljesen lényegtelen, hogy használunk-e általánosítást vagy sem. Ez a döntés a megvalósítás részletei közé tartozik (és kell, hogy tartozzon):

```
public class Sensors {
    private Map sensors = new HashMap();
    public Sensor getById(String id) {
        return (Sensor) sensors.get(id);
    }
    // stb.
}
```


A `határ` (`Map`) felülete rejtve marad, így a fejlődése igen kis hatást gyakorol a teljes alkalmazás kódjára. Az általánosítás használata többé nem okoz komolyabb gondot, hiszen a típusok kezelése és átalakítása már a `Sensors` osztályon belül történik.

A felületet így testreszabhatjuk, és az alkalmazás igényei szerint korlátozhatjuk. Az eredményként kapott kód könnyebben megérthető, és nehezebb „visszaélni” vele. A `Sensors` osztály kikényszeríti a tervezési és a működési szabályokat.

Mindezzel nem azt mondjuk, hogy a `Map` minden felhasználását ilyen burkolóval kell támogatnunk, inkább csak annyit tanácsolunk, hogy ne adjunk körbe felelőtlenül `Map` objektumokat (illetve más, a rendszer határaitól érkezőket) a programjainkban. Ha ilyen, a `Map`-hez hasonló határfelületet alkalmazunk, tartjuk abban az osztályban, illetve osztályok szoros családjában, ahol valóban használjuk. Ha tehetjük, ne adjunk át, illetve ne fogadjunk ilyen objektumokat nyilvános API-któl.

A határok felfedezése

A külső kódok segítenek abban, hogy új lehetőségeket építsünk a programjainkba, mégpedig rövidebb idő alatt, mintha magunk írnánk meg az ehhez szükséges kódrészeket. De hogyan fogunk a munkához, ha külső kódot szeretnénk alkalmazni? A működésének az ellenőrzése nem a mi feladatunk, de a megfelelő tesztek elvégzése mégiscsak a mi érdekünket szolgálja.

Tegyük fel, hogy nem egészen tiszta, hogyan használjuk a frissen beszerezett külső könyvtárat. Eltöltünk egy-két (vagy több) napot a leírás tanulmányozásával, és eldöntjük, hogy is szeretnénk használni. Ezután írunk némi kódot, és megnézzük, hogy valóban azt teszi-e a külső kód, amit vártunk tőle. Egyáltalán nem lennék meglepődve, ha igen hamar a hiba-keresés mocsarának közepén találánánk magunkat, kétségbeesetten kutatva a választ arra a kérdésre, hogy a hibák a saját programunkban vannak vagy a külső kódban.

A külső kódok használatának elsajátítása nem könnyű feladat. A beágyazásuk a programunkba szintén nem sétalogopp. Ha pedig egyszerre végezzük a kettőt, a nehézségeink megkétszereződnek. Mi lenne, ha másik megközelítést választanánk? Ahelyett, hogy rögtön élesben próbálnánk ki a kapott kódot, miért nem készítünk pár tesztet, amelyek segíthetnek a kód megértésében? Jim Newkirk ezeket nevezi *tanulóteszteknek*.¹

A tanulótesztekben éppen úgy hívjuk meg a külső API függvényeit, mintha a programunkban tennénk, vagyis valójában szabályozott kísérleteket végzünk, amelyekkel ellenőrizhetjük az API-ról alkotott képünk helyességét. Arra összpontosítunk tehát, amit valóban fel szeretnénk használni az API lehetőségei közül.

¹ [BeckTDD], 136-137. oldal.

A log4j felfedezése

Tegyük fel, hogy az apache log4j csomagját szeretnénk használni ahelyett, hogy saját kóddal valósítanánk meg a naplózást. Letöltjük hát, és megnyitjuk a bevezető, leíró oldalt. Nem olvasgatunk sokat – el is készítjük az első tesztünket, amely a reményeink szerint a "hello" szöveget írja a konzolra.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("hello");
}
```

A teszt futtatásakor a naplózó hibát jelez, és felhívja a figyelmünket, hogy szüksége lenne egy Appender-re. Némi olvasás után rájövünk, hogy nekünk egy ConsoleAppender objektumra van szükségünk. Elkészítjük hát, és lélegzetvisszafojtva várjuk, hogy a kívánt üzenet megjelenjen a konzolon.

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("hello");
}
```

Nos, azt várhatjuk, ugyanis a rendszer most meg az Appender kimeneti folyamat hiányolja. Furcsa, hiszen logikusnak tűnik, hogy eleve rendelkezzen ilyennel. A Google segítségét igénybe véve a következő megoldással állunk elő:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("hello");
}
```

No, ez végre működik: a "hello" szöveg megjelenik a konzolon. Mindazonáltal továbbra is igen furcsa, hogy a ConsoleAppender-nek nekünk kellett elárulnunk, hogy a konzolra fog írni.

Érdekes módon, ha eltávolítjuk a `ConsoleAppender.SystemOut` paramétert, a kód továbbra is kiírja a "hello" üzenetet. Ha azonban a `PatternLayout` paramétert vesszük el, ismét a kimeneti folyam hiányára panaszkodik. Mi tagadás, furcsa viselkedés...

Ha kicsit tüzetesebben átolvassuk a dokumentációt, azt találjuk, hogy a `ConsoleAppender` alapértelmezett konstruktora „beállítatlan” (unconfigured), ami világosnak nem világos, hasznosnak pedig egyáltalán nem mondható. Nos, ez úgy tűnik, hiba vagy legalábbis következetlenség a `log4j` modulban.

Némi internetes keresgélés, olvasás és tesztelés után végül a 8.1. példához jutunk. Sok mindent megtudtunk a `log4j` működéséről, és a tudásunkat néhány egyszerű egységtesztben foglaltuk össze.

8.1. példa

LogTest.java

```
public class LogTest {
    private Logger logger;
    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }
    @Test
    public void addAppenderWithStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }
    @Test
    public void addAppenderWithoutStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}
```

Tudjuk tehát, hogyan kell elvégezni egy egyszerű konzolnapló kezdeti beállításait, ezt a tudást pedig felhasználhatjuk egy saját osztályban, amellyel elszigetelhetjük a `log4j` határfelületét az alkalmazásunk többi részétől.

A tanulótesztek használata kifizetődő

Ha összeszámoljuk a tanulótesztek költségeit, arra jutunk, hogy voltaképpen semmilyen áldozatot nem kellett hoznunk. Valahogy meg kellett tanulnunk az API használatát, a tesztek elkészítése pedig egyszerű és az alkalmazás többi részétől függetlenül használható módszert adott erre.

A tesztek nemhogy nem kerültek egy fityingünkbe sem, de még hoztak is a konyhára, hiszen ha a külső programcsomag újabb kiadása lát napvilágot, a tanulótesztekkel megvizsgálhatjuk, hogy a viselkedés terén észlelhető-e változások.

A tanulótesztekkel meggyőződhetünk arról, hogy az általunk használt külső programcsomagok valóban úgy működnek, ahogy azt mi elvárjuk. A külső kód a beépítése után nem feltétlenül marad véglegesen összhangban az igényeinkkel, hiszen a szerző a rá nehezedő nyomás alatt más célok elérése felé viheti el a megvalósítást. Sor kerül a hibák kijavítására, és új lehetőségek jelennek meg. Minden újabb kiadás újabb kockázatokat jelent. Ha azonban a változás ütközik a tesztejünkkel, azt azonnal észleljük.

Akár tanulóteszteken keresztül fedezzük fel az új kódot, akár máshogy, a határ tisztán tartásához mindenképpen szükség van kifelé kutakodó tesztekre, amelyek ugyanonnan vizsgálják a külső kódot, ahonnan majd a valódi program hozzáfér. Ha nincsenek ilyen, az átmenetet segítő *határteszteink*, nagy lehet a csábítás, hogy a kelleténél tovább kitartsunk a régi változat mellett.

Jelenleg még nem létező kód használata

Létezik egy másik fajta határ is – ami az ismert az ismeretlentől választja el. Gyakorta találkozhatunk olyan helyekkel a kódban, ahol a tudásunk, úgy fest, nem elég a továbblépéshez. Van, hogy a határon túli világ valóban (legalábbis pillanatnyilag) nem ismerhető meg, de olykor a saját döntésünk, hogy nem lépünk tovább.

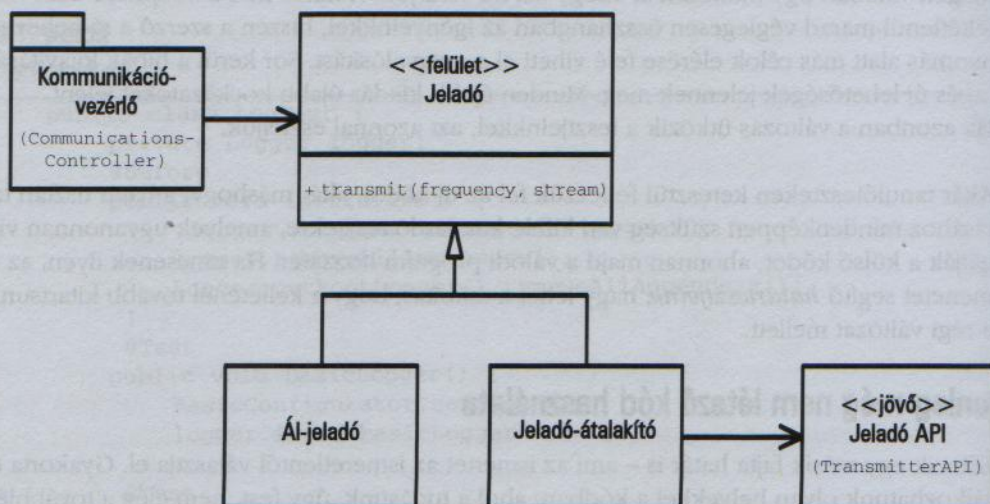
Jó pár évvel ezelőtt részt vettem egy rádiós adatátviteli rendszer programozásában. Volt a programnak egy „Transmitter” (jeladó) nevű része, amelyről igen keveset tudtunk, és a részrendszer felelősei még nem jutottak el addig a pontig, hogy meghatározták volna a felületét. Nem szerettük volna, ha ezért le kell állnunk a munkával, így hát továbbléptünk, gondosan elkerülve az ismeretlen kódrészt. Meglehetősen tiszta képünk volt arról, hogy hol végződik a mi világunk, és hol kezdődik az ismeretlen. Munkánk során számos alkalommal beleütköztünk ebbe a határba. A tudatlanság és a homály nehezítette, hogy túlzottan messzire lássunk a határon túl, de a saját munkánk nyomán világossá vált, milyen felületre van szükségünk. A jeladónak valami ilyesmit szerettünk volna mondani:

Állítsd be a jeladót a megadott frekvenciára, és sugározd a megadott adatfolyam tartalmát analóg jelek alakjában.

Fogalmunk sem volt arról, hogyan fogjuk ezt megcsinálni, hiszen az API még nem készült el, így hát a részletek kidolgozását későbbre hagytuk.

A továbbhaladás érdekében létrehoztunk egy felületet a saját elvárásaink mentén. Adtunk neki egy jól csengő nevet, így lett Transmitter. Elhelyeztünk benne egy `transmit` nevű tagfüggvényt, amely egy frekvenciát és egy adatfolyamot fogadott. Ez volt az a felület, amelyet szerettünk volna használni.

Az ilyen képzeletbeli felületek nagy előnye, hogy itt mindenről magunk rendelkezünk. Így az ügyfélkód is olvashatóbb marad, és arra összpontosíthatunk, amit mi magunk szeretnénk elérni.



8.2. ábra

A jeladó kitalálása

A 8.2. ábrán látható, hogy a `CommunicationsController` osztályokat elszigeteltük a jeladó API-jától (amelyre nincs befolyásunk, és még meg sincs határozva). Saját alkalmazásunkra jellemző felületünket használva a `CommunicationsController` kódját úgy valósíthatuk meg, hogy tiszta és kifejező maradjon. Mihelyt megszületett a jeladó API, megírtuk a `TransmitterAdapter` osztályt, amely áthidalta a közte és a mi kódunk közt tátongó szakadékot. Az átalakító² így magában foglalja az API elérését, és ez az egyetlen hely, ahol módosítanunk kell a kódot, ha az API változik.

² Lásd: [GOF], Átalakító minta.

Ez a szerkezet egyúttal egy igen kényelmes varratot (seam)³ ad a tesztkódban. Megfelelő FakeTransmitter választásával ugyanis tesztelhetjük a CommunicationsController osztályokat. Ha pedig már rendelkezésünkre áll a TransmitterAPI, így határteszteket is készíthetünk a helyes működésének az ellenőrzésére.

Tiszta határok

A határoknál érdekes dolgok történnek – a változás csak az egyik közülük. A jó program-szerkezet képes anélkül befogadni a változásokat, hogy ehhez hatalmas befektetésre vagy átszerkesztésre lenne szükség. Ha olyan kódot használunk, amelyre nincs befolyásunk, különös figyelmet kell fordítanunk az addigi befektetéseink védelmére, no meg annak biztosítására, hogy a jövőbeni módosítások se váljanak túlzottan költségesekké.

A határokon található kódot tisztán el kell választanunk a program többi részétől, és teszttekkel rögzíteni az elvárásainkat. Mindenképpen törekednünk kell arra, hogy a saját kódunk minél kevesebbet tudjon a külső kód részleteiről. Jobb, ha olyasmitől függünk, ami-re befolyással vagyunk, mint ha olyasmitől, amire nem – egyébként még az fog befolyásolni minket.

A külső határok kezelésének legfontosabb alapelve, hogy minél kevesebb helyen hivatkozunk rájuk, Beburkolhatjuk őket, ahogy a Map esetében tettük, de alkalmazhatunk egy átalakítót is, amely megteremti a kapcsolatot a külső kód és az általunk készített „tökéletes” felület között. Akármelyik eljárást választjuk is, a saját kódunk többet mond nekünk, biztosítja a következetes használatot, és sokkal kevesebb helyen kell módosítanunk, ha a külső kódban változások következnek be.

Irodalomjegyzék

[BeckTDD]: Kent Beck: *Test Driven Development*, Addison-Wesley, 2003.

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

[WELC]: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

³ A varratokról bővebben lásd: [WELC].