7

Hibakezelés

Michael Feathers

Furcsának tűnhet, hogy egy tiszta kódról szóló könyvben egy egész fejezetet szentelünk a hibakezelésnek. A hibakezelés azonban minden programozó elkerülhetetlen feladatai közé tartozik. A bemenet lehet érvénytelen, az eszközök meghibásodhatnak – röviden: a baj bármikor bekövetkezhet. Ilyenkor pedig mi, programozók felelünk azért, hogy a kódunk úrrá legyen a helyzeten.

Egyvalami nem vitás: tiszta kódra e téren is nagy szükség van. Rengeteg kódalappal találkoztam, amelyet valósággal eluralt a hibakezelés. Itt nem egyszerűen arra gondolok, hogy a kód csak hibakezeléssel foglalkozik – inkább arra, hogy szinte lehetetlen megérteni, hogy valójában mit is végez a kód, annyira összegubancolódik a hibakezeléssel. Fontos leszögeznünk, hogy a hibakezelés fontos dolog, *de semmiképpen nem fedheti el a program alapszerkezetét.*

Ebben a fejezetben bemutatunk néhány módszert és irányelvet, amelyeket követve tiszta és szilárd kódhoz jutunk, amely a működése közben elegánsan és stílusosan kezeli a hibákat.

Kivételek használata hibakódok helyett

A régmúltban számos programozási nyelv létezett, amelyekben nem volt ismeretes a kivétel fogalma. A hibakezelés és -bejelentés módjai itt meglehetősen korlátozottak voltak – kimerültek egy hibajelző beállításában vagy egy hibakód visszaadásában. Ezt a megközelítést mutatja a 7.1. példa.

7.1. példa

DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
     // Nézzük meg az eszköz állapotát
    if (handle != DeviceHandle.INVALID) {
       // Mentsük az eszköz állapotát a rekordmezőbe
      retrieveDeviceRecord(handle);
       // Ha nincs felfüggesztve, álljunk le
       if (record.getStatus() != DEVICE_SUSPENDED) {
         pauseDevice(handle);
         clearDeviceWorkQueue(handle);
         closeDevice(handle);
       } else {
         logger.log("Device suspended.  Unable to shut down");
       }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

A gondot az okozza, hogy ezek a megoldások zavarossá teszik a hívó kódját. A hívónak ugyanis azonnal ellenőriznie kell a hibákat a hívás után, amiről bizony igen könnyű megfeledkezni. Ezért ha hibával találkozunk, jobb, ha kivételt váltunk ki. A hívó kód így sokkal tisztább marad, hiszen a szerkezetét nem torzítja el a hibakezelés.

A 7.2. példa a kód újabb változatát mutatja azt követően, hogy a tagfüggvényeinket alkalmassá tettük a kivételek kezelésére.

7.2. példa

DeviceController.java (kivételekkel)

```
public class DeviceController {
  ...
  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);
     pauseDevice(handle);
    clearDeviceWorkQueue(handle);
     closeDevice(handle);
  }
  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " +
id.toString());
    ...
  }
  ...
}
```

Vegyük észre, hogy mennyivel tisztább eredményt kaptunk – és ez nem egyszerűen esztétikai kérdés. A kód minőségileg is jobbá vált, hiszen két műveletet – az eszköz leállítását és a hibakezelést –, amelyek korábban összefonódtak, most különválasztottunk. Így mindkettőt áttekinthetjük a maga tiszta valójában.

Első a try-catch-finally utasítás!

A kivételek egyik legérdekesebb tulajdonsága, hogy egy *hatókört határoznak meg* a programunkon belül. Ha a `try-catch-finally` utasítás `try` részében hajtunk végre egy kódrészletet, azzal azt mondjuk, hogy a kód futása bármely ponton megszakadhat, és a `catch` blokkban folytatódhat.

A `try` blokkok bizonyos értelemben olyanok, mint a tranzakciók. Az elsődleges szempont az, hogy a programunk állapota egységes legyen a `catch` blokk elhagyása után, függetlenül attól, hogy mi történt a `try` blokkban. Ezért jó, ha először a `try-catch-finally` utasítást vázoljuk fel, amennyiben olyan kódot szeretnénk készíteni, ami kivételeket válthat ki. Így könnyebben átláthatjuk, mit várhat a felhasználó a kódtól, akár történik valami a `try` blokkban, akár nem.

Lássunk most egy példát! Szükségünk van némi kódra, amely elér egy fájlt, és beolvas néhány sorosított objektumot. Egy egységteszttel kezdjük, ami azt mutatja, hogy kivételt kapunk, ha a fájl nem létezik:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

A teszt hatására ezt a csonkot készítjük el:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // üres return, amíg valódi megvalósítást nem készítünk
  return new ArrayList<RecordedGrip>();
}
```

A tesztünk kudarcának oka, hogy a kód nem vált ki kivételt. A megvalósításunkat most úgy módosítjuk, hogy megpróbáljuk elérni az érvénytelen fájlt. A művelet kivételhez vezet:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

A tesztünk most továbbhalad, mivel elfogtuk a kivételt. Mi azonban álljunk meg, hogy némi átalakítást végezzünk: szűkítsük le a kivétel típusát arra, amit a `FileInputStream` konstruktortól várhatunk – ez a `FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Miután a `try-catch` szerkezettel meghatároztuk a hatókört, a tesztvezérelt fejlesztés elvei mentén megszerkeszthetjük a kód további részét. Ez a `FileInputStream` objektum létre-

hozása és a `close` meghívása közé kerül, és itt úgy tehetünk, mintha minden a legnagyobb rendben lenne.

Próbáljunk olyan teszteket készíteni, amelyek kikényszerítik a kivételeket, majd írjuk meg úgy a kivételkezelőket, hogy kielégítsék a tesztek elvárásait. Mindez segít abban, hogy elsőként kiépítsük a `try` blokk tranzakciós hatókörét, és abban is, hogy megőrizzük a hatókör tranzakciójellegét.

Alkalmazzunk ellenőrizetlen kivételeket!

A Java-programozók éveken át vitatkoztak az ellenőrzött kivételek előnyeiről és hátrányairól – a vitának azonban mostanra vége. A Java első változatában megjelent ellenőrzött kivételek akkor jó ötletnek tűntek. Egy adott tagfüggvény aláírásában fel kellett sorolni mindenféle kivételt, amelyet továbbadhatott a hívójának, ráadásul ezek a kivételek beépültek a tagfüggvény típusába. A fordítóprogram visszadobta a kódot, ha az aláírás nem felelt meg a kódban történteknek.

Akkor úgy gondoltuk, hogy az ellenőrzött kivételek bevezetése nagyszerű ötlet – és nem tagadhatjuk, hogy valóban járt némi előnnyel. Azóta viszont megtanultuk, hogy a jelenlétükre nincs feltétlenül szükség szilárd programok fejlesztéséhez. A C#-ból hiányoznak az ellenőrzött kivételek, és a C++-ba sem kerültek be, minden erőfeszítés ellenére. A Python és a Ruby is ellenállt. Mégis léteznek szilárd programok, amelyeket ezeken a nyelveken írtak. Mindennek ismeretében komolyan el kell gondolkodnunk azon, hogy megéri-e az ellenőrzött kivételek használata azt az árat, amelyet cserébe meg kell fizetnünk?

De mi ez az ár? Nos, az ellenőrzött kivételek használatának ára a Zárt nyíltság elvének (Open/Closed Principle)[1] megsértése. Ha a kód egyik tagfüggvényében kiváltunk egy ellenőrzött kivételt, és a hozzá tartozó `catch` három szinttel feljebb található, *akkor ezt a kivételt el kell helyeznünk a közöttünk és a* `catch` *között álló összes tagfüggvény aláírásában*. Ez pedig azt jelenti, hogy egy alacsony szintű változtatás számos magasabb szintű módosítást von maga után. A megváltoztatott modulokat pedig újra kell építenünk és telepítenünk, annak ellenére, hogy a tevékenységük egy fikarcnyit sem változott.

Vegyük példaként egy nagy rendszer hívási hierarchiáját. A legfelső függvények meghívják az alattuk levőket, azok a még lejjebb találhatókat, és így tovább. Most pedig tegyük fel, hogy egy igen alacsony szintű függvényt úgy módosítunk, hogy kivételt válthat ki a későbbiekben. Ha ez a kivétel ellenőrzött, a függvény aláírásában fel kell tüntetnünk egy `throws` részt is. Ez azonban azt jelenti, hogy minden olyan függvényt, amely a mi függvényünket hívja, módosítanunk kell úgy, hogy kezelje a kivételt, vagy feltüntesse a `throws` tagot a saját aláírásában. És ez így gyűrűzik tovább felfelé, egészen a legfelső

[1] *[Martin]*

szintekig. Az egységbe zárás sérül, hiszen az útba eső függvényeknek mind-mind tudniuk kell, hogy mit is jelent ez az alacsony szintű kivétel. Márpedig a kivételekkel éppen azt próbáltuk elérni, hogy a hibákat a keletkezésük helyétől távol kezelhessük – ezzel a kapcsolódással éppen ennek az ellenkezőjét érjük el.

Az ellenőrzött kivételek nagy szolgálatot tehetnek egy létfontosságú könyvtárnál, hiszen ezeket mindenképpen el kell fognunk. Általános alkalmazásfejlesztés során azonban a hátrányaik jóval felülmúlják az előnyeiket.

Határozzuk meg a hiba környezetét a kivételekkel!

Olyan kivételeket váltsunk ki, amelyek kellőképpen behatárolják a környezetüket a hiba forrásának és helyének megállapításához. A Javában minden kivétellel hozzájuthatunk a veremlenyomathoz, de ez nem feltétlenül tájékoztat a sikertelen művelet mögött álló programozói szándékról.

Írjunk kellően beszédes hibaüzeneteket, és adjuk át őket a kivételeinkkel. Említsük meg a sikertelen műveletet, valamint a kudarc típusát. Ha egy alkalmazás működését naplózzuk, adjunk át elég adatot ahhoz, hogy a `catch` blokkban el tudjuk készíteni a megfelelő naplóbejegyzést.

A hívó igényei szerint határozzuk meg a kivételosztályokat!

A kivételek osztályozásának számos módja ismeretes. Besorolhatjuk őket például a származási helyük szerint: a program melyik összetevőjéből érkeztek? Vagy a típusuk szerint: eszközök meghibásodásáról, hálózati problémákról vagy programozási hibákról van szó? Akárhogy is, ha egy alkalmazásban kivételosztályokat határozunk meg, a legfontosabb szempont az legyen, hogy *miként fogjuk el őket.*

Lássunk most egy példát a rossz osztályozásra. Íme egy külső könyvtári híváshoz tartozó `try-catch-finally` blokk, amely lefed minden kivételt, ami csak a hívásból származhat:

```
ACMEPort port = new ACMEPort(12);
try {
  port.open();
} catch (DeviceResponseException e) {
   reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
   reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
   reportPortError(e);
  logger.log("Device response exception");
```

```
    } finally {
      ...
    }
```

Az utasítás rengeteg ismétlődést tartalmaz, ami nem is meglepő. A legtöbb kivételkezelési helyzetben jobbára meglehetősen egyhangú munkát kell végeznünk – rögzítenünk kell a hibát, és gondoskodnunk kell arról, hogy a program végrehajtása továbbhaladhasson.

Esetünkben a helyzet egyszerű, mivel tudjuk, hogy nagyjából ugyanazt tesszük, akármi legyen is a kivétel, így jelentősen egyszerűsíthetjük a kódot, ha a hívott API-t beburkoljuk, és gondoskodunk róla, hogy egyetlen, közös típusú kivételt adjon vissza.

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  ...
}
```

A `LocalPort` osztály nem más, mint egy egyszerű burkoló, amely elfogja és átalakítja az `ACMEPort` osztály kivételeit:

```
public class LocalPort {
  private ACMEPort innerPort;
  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }
  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  ...
}
```

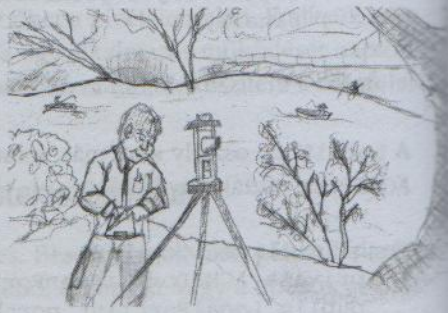
Az ilyen burkolók igen hasznosak lehetnek – egy külső API beburkolásával megszabadulhatunk a hozzá kapcsolódó függőségektől, vagyis később komolyabb költségek nélkül

másik könyvtárra válthatunk. A burkolással könnyebb a külső hívások kizárása is, amikor a saját kódunkat szeretnénk tesztelni.

Végezetül, a burkolás további előnye, hogy így nem kell kötődnünk egy adott gyártó API-jának szerkezeti megoldásaihoz – az API-t saját kedvünk szerint formálhatjuk. A fenti példában egyetlen kivételtípust határoztunk meg a `port` eszköz meghibásodásához, és láttuk, mennyivel tisztább kódot tudunk előállítani.

Gyakori jelenség, hogy a kód adott területén belül egyetlen kivételosztály tökéletesen megfelel a céljainknak, és a kivétellel átadott adatok segítségével különbséget tehetünk a hibák között. Eltérő kivételosztályokat tehát csak akkor használjunk, ha bizonyos típusú kivételeket el szeretnénk fogni, a többit pedig inkább továbbadjuk.

Határozzuk meg a program normál menetét!



Ha követjük az előzőekben szereplő útmutatásokat, meglehetősen hatékonyan szétválaszthatjuk a lényegi kódot és a hibakezelést. A kódunk magja egyre inkább hasonlít egy tiszta, háborítatlan algoritmusra. Miközben azonban efelé törekszünk, a hibakezelés egyre kijjebb szorul a program „széleire". Beburkoljuk a külső API-kat, hogy saját kivételeket válthassunk ki, és hibakezelőt határozunk meg a kódunk felett, amely képes kezelni a félbeszakadt számításokat. Az esetek többségében ez jó megoldást jelent, de előfordulhat, hogy nem szeretnénk, ha a kód végrehajtása megszakadna.

Nézzünk egy példát! Az alábbiakban egy sutácska kódrészletet láthatunk, amely egy számlázó alkalmazásban elkészíti a kiadások összegét:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

A kód gondoskodik arról, hogy ha egy dolgozó képes költségként leírni az ebéd (meal) árát, akkor ezt hozzáadja a kiadásokhoz (expenses). Ha azonban ez nem működik, napi

(per diem) ebédköltséget számol el. A kivétel összezavarja a kód szerkezetét. Ugye jobb lenne, ha nem kellene törődnünk ezzel a különleges esettel? Hát persze, hiszen a kód valahogy így festene:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Meg tudjuk oldani mindezt ilyen egyszerűen is? Nos, úgy fest, hogy igen. Módosíthatjuk az `expenseReportDAO` tagfüggvényt úgy, hogy minden esetben visszaadjon egy `MealExpense` objektumot. Ha a dolgozó nem tud elszámolni az ebéddel, a tagfüggvény egy olyan `MealExpense` objektumot ad, amelynek az összege megegyezik a napi ebédköltséggel:

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // visszaadjuk az alapértelmezett napi ebédköltséget
  }
}
```

Ez a Különleges eset minta [Fowler], ami szerint úgy hozunk létre egy osztályt, vagy állítunk be egy objektumot, hogy az kezelje a különleges esetet az ügyfélkód helyett. A viselkedést ezzel a különleges eset objektumába zárjuk.

Ne adjunk vissza null értéket!

Úgy vélem, hogy ha a hibakezelésről beszélünk, feltétlenül említést kell tennünk pár dologról, amelyek valósággal vonzzák a hibákat. Az első ilyen a `null` érték visszaadása. Megszámlálhatatlan olyan alkalmazást láttam már az életem során, amelynek szinte minden második sora a `null` érték vizsgálatával foglalkozott. Íme egy példa a sok közül:

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Ha ilyen kódalappal dolgozunk, nem feltétlenül botránkozunk meg rajta, de jobb, ha tudjuk, hogy nagy a baj. A `null` érték visszaadásával csak a saját vállunkra pakolunk többletterhet, és áthárítjuk a munkát a hívóra. Elég egyetlen helyen megfeledkezni a `null` érték vizsgálatáról, és a program máris elszáll a végtelenbe…

Észrevettük, hogy a kódrészlet nem vizsgálja a `null` értéket a beágyazott `if` utasítás második sorában? Mi történne, ha a program futása közben a `persistentStore` a `null` értéket adná vissza? Nos, ez bizony egy `NullPointerException` megjelenésével járna, amelyet vagy elfog valaki a legfelső szinten, vagy nem. Egyik eset sem kecsegtet semmi jóval. Mit kezdhetünk egy `NullPointerException` kivétellel, ami valahogy hozzánk került az alkalmazás mélyéről?

Könnyű azt mondani, hogy a problémát a `null` vizsgálatának hiánya jelenti, de valójában inkább az a baj, hogy *túl sok* ilyen vizsgálatra kerül sor. Ha erős késztetést érzünk arra, hogy `null` értéket adjunk vissza, inkább válasszunk helyette egy „különleges eset" objektumot. Ha pedig egy külső API tagfüggvényét hívjuk, amely `null` értéket is visszaadhat, gondolkodjunk el valamilyen burkolófüggvény használatán, amely inkább kivételt vált ki vagy „különleges eset" objektumot ad vissza.

A „különleges eset" objektumok számos esetben jó megoldást jelenthetnek. Tegyük fel, hogy a kódunk így fest:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

A `getEmployees` pillanatnyilag `null` értéket is visszaadhat, de valóban szükség van erre? Ha úgy módosítjuk a függvényt, hogy ehelyett inkább egy üres lista mellett döntsön, máris kitisztul a kód:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Szerencsére a Java rendelkezésünkre bocsátja a `Collections.emptyList()` nevű tagfüggvényt, amely egy előre meghatározott, nem módosuló listát ad vissza – ez éppen megfelel a céljainknak:

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

Ha ilyen elvek mentén programozunk, a `NullPointerException` megjelenésének esélyét jelentősen csökkenthetjük, és a kódunk is tisztábbá válik.

Ne adjunk át null értéket!

`Null` értéket kapni rossz dolog, átadni azonban egyenesen szörnyű! Ilyesmire kizárólag akkor vetemedjünk, ha egy API kifejezetten elvárja a `null` érték átadását.

De hogy ne csak a levegőbe beszéljünk, lássunk egy példát, ami bemutatja, miért is ezek a kemény szavak. Íme egy egyszerű tagfüggvény, amely két pont távolságát számítja ki:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  ...
}
```

Mi történik, ha valaki paraméterként a `null` értéket adja át?

```
calculator.xProjection(null, new Point(12, 13));
```

Természetesen `NullPointerException` kivételhez jutunk. Hogyan találhatunk erre megoldást? Meghatározhatunk egy új kivételtípust, és kiválthatjuk ezt:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
  }
}
```

Jobban jártunk? Egy kissé jobb a helyzet, mintha egy `NullPointerException`-nal állnánk szemben, de ne feledjük, hogy az `InvalidArgumentException` kivételt kezelnünk kell valahogy. De hogyan? Eszünkbe jut valamilyen értelmes művelet?

Létezik egy másik út is – alkalmazhatunk feltételezéseket (állításokat, assertion):

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
```

```
        return (p2.x - p1.x) * 1.5;
    }
}
```

Dokumentációnak nagyszerű, de nem oldja meg a problémát, hiszen ha valaki `null` értéket ad át, továbbra is futásidejű hibát kapunk.

A legtöbb programozási nyelvben sajnos nincs megfelelő módszer azoknak a `null` értékeknek a kezelésére, amelyeket a hívók véletlenül adnak át a függvényeknek. Ennek ismeretében a legokosabb, ha alapértelmezés szerint megtiltjuk a `null` érték átadását. Így ha a `null` érték mégis megjelenik a paraméterlistában, biztosan tudhatjuk, hogy gond van, így a véletlenül elkövetett hibák zömét elkerülhetjük.

Összefoglalás

A tiszta kód jól olvasható, de egyúttal fontos az is, hogy szilárd legyen. Ezek a célok látszólag ellentétesek, pedig valójában nem azok. Lehetséges szilárd *és* tiszta kódot készíteni, ha a program magját világosan elválasztjuk a hibakezelési feladatoktól. Minél hatékonyabb az elválasztás, annál önállóbban hozhatunk döntéseket a különböző területeken, ami viszont jelentősen hozzájárul a kód fenntarthatóságához.

Irodalomjegyzék

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