KAPITOLA 7

Zpracování chyb

Michael Feathers

V této kapitole najdete:

Může se vám zdát divné mít sekci o zpracování chyb v knize, která pojednává o čistém kódu. Zpracování chyb je jen jednou z těch věcí, kterým se musíme během programování věnovat všichni. Vstupní hodnoty mohou být abnormální a zařízení se může porouchat. Stručně řečeno, věci se mohou pokazit, a když k tomu dojde, jako programátoři jsme zodpovědní za to, že náš kód bude dělat to, co má.

Avšak vztah k čistému kódu by měl být zřejmý. V mnoha kódech aplikace zpracování chyb naprosto převládá. Když říkám, že převládá, nemám na mysli, že zpracování chyb je to jediné, co dělají. Myslím tím to, že je téměř nemožné pochopit, co kód dělá, kvůli zpracování chyb, které je všudypřítomné. Zpracování chyb je důležité, *ale když zatemňuje logiku, je špatné.*

V této kapitole popíšu několik technik a úvah, které můžete použít pro psaní kódu, jenž je jak čistý, tak i robustní – kód, který zpracuje chyby s elegancí a šarmem.

Používejte výjimky raději než návratové kódy

V dávné minulosti bylo mnoho jazyků, které neměly výjimky. V těchto jazycích byly techniky zpracování a informování o chybách omezené. Kód mohl buď nastavit příznak chyby, nebo vrátit číslo chyby, které mohl volající objekt prověřit. Tento pohled kód ilustruje ve výpisu 7.1.

Výpis 7.1. DeviceController.java

```
public class DeviceController {
   ...
   public void sendShutDown() {
      DeviceHandle handle = getHandle(DEV1);
      // Zkontrolujte stav zařízení
      if (handle != DeviceHandle.INVALID) {
         // Uložte stav zařízení do pole záznamu
         retrieveDeviceRecord(handle);
         // pokud není pozastaveno, vypnout
         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());
      }
   }
   ...
}
```

Problém tohoto pohledu spočívá v tom, že zaneřáďuje volající objekt. Ten musí prověřit chyby bezprostředně po zavolání. Bohužel se na to dá jednoduše zapomenout. Z tohoto důvodu je lepší, když se při výskytu chyby vyvolá výjimka. Volající kód je čistší. Jeho logika není zamlžována zpracováním chyb.

Výpis 7.2 ukazuje kód poté, co jsme si zvolili možnost vyvolávat výjimky v metodách, které mohou detekovat chyby.

Výpis 7.2. DeviceController.java (s výjimkami)

```
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());
      ...
   }
   ...
}
```

Všimněte si, oč je kód čistší. Není to pouze záležitost estetiky. Kód je lepší, protože dvě předtím propletené záležitosti, algoritmus pro odstavení zařízení a pro zpracování chyb, jsou od sebe nyní odděleny. Můžete se na obě věci podívat a chápat je nezávisle.

Pište nejdříve příkazy Try-Catch-Finally

Jednou z nejzajímavějších věcí u výjimek je, že uvnitř vašeho programu *definují rozsah platnosti identifikátorů*. Když provádíte kód v části `try` příkazu `try-catch-finally`, určujete tím, že provádění programu může být přerušeno ve kterémkoliv bodě a může pokračovat ve větvi `catch`.

Z tohoto pohledu se bloky `try` podobají transakcím. Část `catch` musí zanechat váš program v konzistentním stavu bez ohledu na to, co se stane v části `try`. Z tohoto důvodu je dobré začít s příkazem `try-catch-finally`, pokud píšete kód, který může generovat výjimky. To vám pomůže definovat, co by měl uživatel tohoto kódu očekávat, bez ohledu na to, jaká chyba nastane v kódu, který se provádí v části `try`.

Podívejme se na jeden příklad. Potřebujeme napsat kód, který přistupuje k souboru a čte nějaké serializované objekty.

Začneme u jednotkového testu, který ukazuje, že obdržíme výjimku, pokud soubor neexistuje:

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

Test nás přivede k tomu, abychom vytvořili tento prázdný kód:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
   // prázdná vrácená hodnota, dokud nebudeme mít skutečnou implementaci
   return new ArrayList<RecordedGrip>();
}
```

Náš test neprojde, protože nevyvolá výjimku. V dalším kroku změníme jeho implementaci tak, aby se pokusila přistoupit k neplatnému souboru. Tato operace vygeneruje výjimku:

```
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>();
}
```

Test nyní prochází, protože jsme výjimku zachytili. V tomto okamžiku můžeme refaktorovat. Můžeme zúžit typ výjimky, kterou zachytáváme, aby odpovídala typu, jenž se ve skutečnosti generuje v konstruktoru `FileInputStream`: `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>();
}
```

Když jsme nyní definovali obor ve struktuře příkazu `try-catch`, můžeme použít TDD (test-driven development), abychom napsali zbytek logiky, kterou potřebujeme. Tato logika bude zařazena mezi vytvoření `FileInputStream` a `close` a může předstírat, že je vše v pořádku.

Zkuste napsat testy, které si vynucují výjimky, a poté přidejte do obslužné rutiny funkčnost tak, aby testy prošly. To způsobí, že nejdříve vytvoříte transakční obor bloku `try` a pomůže vám to udržovat transakční povahu této části.

Používejte nekontrolované výjimky

Tato debata je již za námi. Již léta debatovali programátoři v Javě o výhodách a nevýhodách kontrolovaných výjimek. Když byly kontrolované výjimky poprvé představeny v první verzi Javy, zdálo se, že jde o výbornou myšlenku. Všechny typy výjimek by byly uvedeny v signatuře každé metody, které může předávat volajícímu. Dále byly tyto výjimky součástí typu metody. Váš kód by se vůbec nepřeložil, pokud by signatura nevyhovovala tomu, co může kód dělat.

Tehdy jsme mysleli, že kontrolované výjimky jsou dobrým nápadem, a skutečně mohou *nějaké* výhody mít. Avšak nyní je zřejmé, že pro tvorbu robustního softwaru nejsou nezbytné. Jazyk C# kontro-

lované výjimky nemá a navzdory chrabrým pokusům je neobsahuje ani jazyk C++. Python a Ruby také ne. Přesto je možné psát ve všech těchto jazycích robustní software. Protože právě o tohle jde, musíme se rozhodnout – skutečně – zda se používání kontrolovaných výjimek vyplatí.

Jaká je jejich cena? Cenou za používání kontrolovaných výjimek je porušení principu otevřenosti a uzavřenosti[1]. Pokud v kódu některé metody generujete kontrolovanou výjimku, a příkaz `catch` je tři o úrovně výše, *musíte tuto výjimku deklarovat v signatuře všech metod mezi místem jejího vzniku a klauzulí catch.* To znamená, že změna na nízké úrovni softwaru může vyvolat změny v signaturách na více vyšších úrovních. Změněné moduly je nutné znovu sestavit a nainstalovat, i když se jich změny vůbec netýkají.

Podívejte se na hierarchii volání ve velkém systému. Funkce, které jsou nahoře, volají funkce, jež jsou níže. Ty volají další nižší funkce a tak to jde do nekonečna. Řekněme, že jedna z nejnižších funkcí se změní takovým způsobem, že musí generovat výjimku. Jestliže jde o kontrolovanou výjimku, pak je nutné přidat do signatury funkce klauzuli `throws`. Ale to znamená, že každá funkce, která volá naši změněnou funkci, se také musí změnit. Buď musí zachytávat novou výjimku, nebo musíme do její signatury přidat příslušnou klauzuli `throws`. Do nekonečna. Čistým výsledkem je kaskáda změn, které si razí cestu od nejnižších úrovní softwaru do těch nejvyšších! Zapouzdření se poruší, protože všechny funkce v cestě od klauzule `throw` musí znát detaily o výjimce nízké úrovně. Za předpokladu, že smyslem výjimek je umožnit zpracování chyb „na dálku", je ostudou, že kontrolované výjimky tímto způsobem dokážou zapouzdření porušit.

Kontrolované výjimky mohou být někdy užitečné, pokud píšete nějakou důležitou knihovnu: Musíte je zachytit. Ale během vývoje obecné aplikace je cena za závislost vyšší, než je jejich přínos.

Poskytujte kontext s výjimkami

Každá výjimka, kterou vygenerujete, by měla poskytnout dostatek souvisejících informací, aby bylo možné určit zdroj a místo chyby. V Javě můžete mít k dispozici ke každé chybě trasování zásobníku. Ale trasování zásobníku vám nemůže poskytnout záměr operace, která se nezdařila.

Vytvářejte informativní chybová hlášení a předávejte je spolu s vašimi výjimkami. Uveďte operaci, která se nezdařila, a druh selhání. Pokud běh vaší aplikace protokolujete, předávejte dostatečné informace, abyste byli schopni chybu zaprotokolovat v klauzuli `catch`.

Definujte třídy výjimek z hlediska potřeb volajícího

Existuje mnoho způsobů klasifikace chyb. Můžeme je klasifikovat podle jejich zdroje: přicházejí z té, nebo oné komponenty? Nebo podle jejich typu: jde o selhání zařízení, sítě, nebo jde o programovou chybu? Jestliže však v aplikaci definujeme třídy výjimek, měla by naším nejdůležitějším úkolem být znalost, *jak byly zachyceny.*

Podívejme se na příklad špatné klasifikace chyb. Máme zde příkaz `try-catch-finally`, která volá knihovnu třetí strany. Zahrnuje všechny výjimky, které může volání způsobit:

1. [Martin].

```
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 {
  …
}
```

Tento příkaz obsahuje mnoho zdvojení a neměli bychom tím být překvapeni. Ve většině případů zpracování výjimek je prováděná činnost relativně standardní bez ohledu na skutečný důvod. Musíme zaznamenat chybu a ověřit si, že můžeme pokračovat.

V tomto případě, protože víme, že prováděná činnost je pro všechny výjimky zhruba stejná, můžeme svůj kód značně zjednodušit, když ho zabalíme do API, které voláme, a zajistíme, aby návratovou hodnotou byl obecný typ výjimky:

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

Třída `LocalPort` je pouze jednoduchou obálkou, která zachytává a překládá výjimky generované třídou `ACMEPort`:

```
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);
    }
  }
  …
}
```

Obálka, kterou jsme definovali pro třídu `ACMEPort`, může být velmi užitečná. Obalování API třetí strany je skutečně tím nejlepším postupem. Když vytváříte obal pro API třetí strany, minimalizujete tím svou závislost na něm: v budoucnu si můžete bez nějakých větších důsledků zvolit jinou knihovnu. Obalování rovněž zjednodušuje napodobování volání objektů třetí strany při testování vašeho vlastního kódu.

Jednou z posledních výhod obalování je, že nejste při návrhu vázáni na volbu rozhraní API nějakého konkrétního dodavatele. Můžete definovat API, se kterým se vám bude pohodlně pracovat. V předchozím příkladu jsme definovali jediný typ výjimky pro selhání zařízení `port` a zjistili jsme, že bychom mohli psát mnohem čistší kód.

Samostatná třída pro výjimky je často dobrá pro určitou oblast kódu. Informace, která se posílá spolu s výjimkou, může chyby dále charakterizovat. Používejte jiné třídy jen v těch případech, kdy potřebujete zachytit jednu výjimku a ostatní nechat být.

Definujte normální tok

Pokud se budete řídit radami z předchozích sekcí, bude výsledkem dobré oddělení kódu obchodní logiky od kódu pro zpracování chyb. Značná část kódu bude vypadat jako čistý a nevyumělkovaný algoritmus. Avšak proces jeho vytváření vytlačuje detekci chyb na okraj vašeho programu. Zabalíte externí API tak, aby mohlo generovat vaše vlastní výjimky, a nad vaším kódem definujete obslužnou rutinu tak, abyste mohli zpracovat jakýkoliv předčasně ukončený chod programu. Většinou je tento přístup velmi dobrý, ale existují případy, kdy běh předčasně ukončit nechcete.

Podívejme se na jeden příklad. Máme zde jakýsi nešikovný kód, který sečítá náklady v nějaké účetní aplikaci:

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

V tomto obchodě platí, že pokud jsou jídla účtována do nákladů, stávají se součástí celkové sumy. Pokud ne, dostává v ten den zaměstnanec částku za jídlo *per diem*. Výjimka vnáší do logiky zmatek. Nebylo by lepší, kdybychom nemuseli zpracovávat jeden speciální případ? Pokud bychom nemuseli, vypadal by náš kód mnohem jednodušeji. Vypadal by takto:

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

Můžeme kód takto zjednodušit? Zdá se, že ano. Můžeme změnit metodu `ExpenseReportDAO` tak, aby vždy vracela objekt `MealExpense`. Pokud žádné náklady na jídlo neexistují, vrací objekt `MealExpense`, který vrací hodnotu *per diem* jako celkovou sumu:

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // vrácení implicitní hodnoty per diem
  }
}
```

Tohle se nazývá objekt pro zvláštní případ (Special Case Pattern – [Fowler]). Vytvoříte třídu nebo konfigurujete objekt tak, aby pro vás ošetřil zvláštní případy. Když to uděláte, klientský kód se nemusí zabývat funkčností pro výjimky. Tato funkčnost je zapouzdřena v objektu určeném pro zvláštní případy.

Nevracejte hodnotu null

Jsem toho názoru, že jakákoliv diskuse o ošetřování chyb by měla obsahovat zmínku o tom, jak chyby vyvoláváme. Na prvním místě je na seznamu vracení hodnoty `null`. Viděl jsem bezpočet aplikací, ve kterých téměř každý druhý řádek kontroloval hodnotu `null`. Zde máme příklad takového kódu:

```
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);
      }
    }
  }
}
```

Pokud pracujete s podobným kódem aplikace, nemusí se vám to zdát tak špatné, ale špatné to je! Pokud vracíme hodnotu `null`, v podstatě si přidáváme další práci a vytváříme problémy volajícím objektům. Přitom stačí jedna chybějící kontrola hodnoty `null` a celá aplikace se vymkne kontrole.

Všimli jste si, že ve druhém řádku vnořeného příkazu `if` chybí kontrola na `null`? V době běhu programu bychom obdrželi výjimku typu `NullPointerException`, a ať už tuto výjimku na vysoké úrovni někdo zachytí nebo ne, obojí je špatně. Jak byste měli přesně reagovat na výjimku typu `NullPointerException`, která vznikla někde v hlubinách vaší aplikace?

Je jednoduché říci, že problém nadřazeného kódu je v tom, že chybí kontrola hodnoty `null`, ale ve skutečnosti je problém v tom, že jich je příliš mnoho. Pokud jste v pokušení vracet z metody `null`, zvažte místo toho vyvolání výjimky nebo vrácení objektu pro zvláštní případ (SPECIAL CASE OBJECT). Pokud voláte metodu API třetí strany, která vrací hodnotu `null`, zvažte obalení této metody tak, aby buď vyvolala výjimku, nebo vrátila objekt pro zvláštní případ.

V mnoha případech představují objekty pro zvláštní případ jednoduché řešení. Představte si, že máte kód, jako je tento:

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

Právě zde může metoda vrátit hodnotu `null`, ale musí? Pokud změníme metodu `getEmployee` tak, aby vracela prázdný seznam, můžeme kód vyčistit takto:

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

Naštěstí obsahuje Java metodu `Collections.emptyList()`, jež vrací předdefinovaný neměnný seznam, který můžeme použít pro náš účel:

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

Budete-li psát kód tímto způsobem, minimalizujete možnost výjimek typu `NullPointerExceptions` a váš kód bude čistší.

Nepředávejte hodnotu null

Vracení hodnoty `null` je špatné, ale předávání této hodnoty je ještě horší. Pracujete-li s API, které předpokládá, že budete předávat `null`, měli byste se tomu ve vašem kódu vyhnout všude, kde je to možné.

Podívejme se na tento příklad, který ukazuje proč. Máme zde jednoduchou metodu, která počítá metriku dvou bodů:

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

Co se stane, když někdo předá jako argument hodnotu `null`?

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

Obdržíme samozřejmě výjimku typu `NullPointerException`.

Jak to můžeme opravit? Mohli bychom vytvořit nový typ výjimky a vygenerovat ji:

```
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;
  }
}
```

Je to lepší? Mohlo by to být o něco lepší, než je výjimka způsobená hodnotou `null`, ale mějte na paměti, že potřebujeme definovat obslužnou rutinu pro výjimku typu `InvalidArgumentException`. Co by měla tato rutina dělat? Je na tomto postupu vůbec něco dobrého?

Existuje i jiná alternativa. Mohli bychom použít několik asercí:

```
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;
  }
}
```

Je to dobré k dokumentaci, ale problém to neřeší. Pokud někdo předá hodnotu `null`, stále to bude znamenat chybu za běhu programu.

Ve většině programovacích jazyků neexistuje dobrý způsob, jak se vypořádat s hodnotou `null`, kterou volající objekt předává náhodně. Protože se jedná o náš případ, je racionálním přístupem jako standard zakázat předávání `null`. Když to provedete, můžete psát kód s tím, že hodnota `null` v seznamu argumentů indikuje problém a výsledkem bude mnohem menší množství chyb vzniklých nedbalostí.

Závěr

Čistý kód je čitelný, ale musí být zároveň robustní. Tohle nejsou protikladné cíle. Můžeme psát robustní a čistý kód, pokud spatřujeme ve zpracování chyb samostatnou záležitost, něco, co lze vnímat nezávisle na hlavní logice kódu. Když se nám to podaří, můžeme o tom argumentovat nezávisle a udělat v udržovatelnosti našeho kódu velký pokrok.

Použitá literatura

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