Komentáře a jednotkové testy

8

1 Komentáře

Je samozřejmé, že od funkčního programu (příp. třídy, metody, funkce apod.) očekáváme, že se bude dát spustit a bude vracet námi očekávané hodnoty nebo provádět očekávanou posloupnost operací. To je ale pouze část práce programátora. Důležité je, aby program byl srozumitelný nikoliv pro počítač, ale zejména pro lidi, kteří budou s kódem programu dále pracovat. V praxi se málokdy stane, že vytvoříte program a už se k němu nikdy nebudete vracet.[1] Většinou narazíte na to, že program je potřeba rozšířit o nové funkce, upravit podle aktuálních potřeb nebo v něm opravit chyby. Pokud je kód nesrozumitelný, jsou pak jakékoliv úpravy časově (a tím pádem i finančně) náročné a v extrémním případě je lepší původní kód zahodit a napsat od začátku.

Abychom se takovým situacím vyhnuli, je důležité psát programy s maximálním ohledem na čitelnost a srozumitelnost pro člověka.[2] Důležité je proto volit správně jména tříd, metod a proměnných. Aby bylo jasné jaký účel má daná třídy, co provede nebo jako hodnotu metoda vrátí nebo jakou hodnotu můžeme nalézt v atributu nebo proměnné. V ideálním případě bychom z kódu samotného měli být schopni určit, co bude dělat. V případě, že se nám nepodaří kód napsat tak, aby z něj byly patrné všechny důležité informace, jsou na řadě komentáře. Ty jsou následujících typů a již jsme je v úvodních seminářích zmiňovali.

```
1  // jednořádkový komentář, kde je vše až do konce řádku ignorováno
2
3  /* víceřádkový komentář,
4     vhdnodný pro delší texty
5     například text licence v úvodu zdrojového kódu. */
6
7  /**
8   * Java-doc komentáře, slouží k doplnění informací, ke třídám, metodám, atributům.
9   */
10
```

[1] Pokud taková situace nastane, je buď program opravdu extrémně dobrý, nebo opravdu extrémně špatný.

[2] *„...a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."* — Abelson H., Sussman G., Sussman J. Structure and Interpretation of Computer Programs. `https://web.mit.edu/6.001/6.037/sicp.pdf`

```
/// Od verze Java 23.
/// Lze psát java-doc komentáře i v jazyce *Markdown*.
/// Použijí se k tomu _tři lomítka_.
```

Prominentní místo mají tzv. Java-doc komentáře, které jsou speciálním případem víceřádkových komentářů, kde jsou na začátku místo jedné hvězdičky uvedeny hvezdičky dvě. Název těmto komentářům dal nástroj, který umožňuje automaticky vygenerovat dokumentaci pro celý projekt. S těmito komentáři pak umí pracovat vývojová prostředí (IDE) a dokáží je zobrazovat v průběhu psaní programu.

Součástí Java-doc komentářů mohou být tzv. tagy, které označují specifické typy informací, které mohou být dále zpracovávány. Ukážeme si to na konkrétním příkladu.

```
/**
 * Metoda převede řetězec skládající se z textové reprezentace čísla
 * ve dvojkové soustavě na hodnotu typu int.
 *
 * @param number řetězec skládající se ze znaků '0' a '1' (např. "110011")
 * @return celé číslo, které odpovídá zápisu vstupního řetezce ve dvojkové soustavě
 * @throws IntegerOverflowException v případě, že není možné reprezentovat
 * výslednou hodnotu typem int
 * @throws InvalidDigitException pokud vstupní řetězec obsahuje neplatný znak
 * @see Integer#parseInt(String, int)
 * @author Jim Hacker
 * @since 0.7 (lecture 07)
 */
public static int binaryToInt(String number) throws IntegerOverflowException,
 ↪  InvalidDigitException {
 /* kód pro přehlednost vypuštěn */
}
```

V úvodu máme popis, co metoda dělá, a následují tagy, které upřesňují informace o metodě:

- `@param` upřesňuje informaci o parametru (v našem případě `number`), pokud bychom měli parametrů více, uvedeme tento tag vícekrát,
- `@return` upřesňuje, co je návratovou hodnotou metody,
- `@throws` poskytuje informace jaké výjimky mohou být vyvolány a za jakých podmínek,
- `@see` směruje na dodatečné informace, všimněme si, že je zde uveden odkaz na metodu `parseInt(String, int)` třídy `Integer` (podobnou funkcionalitu nabízí tag `@link`),
- `@author` udává, kdo je autorem metody (případně třídy),

- `@since` poskytuje informaci, od které verze je metoda k dispozici.

Poznámka 1 *Tento výčet není úplný a zahrnuje jen ty nejčastěji používáné tagy, laskavý čtenář může dohledat související informace v oficiální dokumentaci.*[3]

Poznámka 2 *V podobném duchu píšeme komentáře i k atributům. U nich si většinou vystačíme s prostým popisem bez použití tagů.*

Protože nástroj Java-doc je primárně určen k tomu, aby vygeneroval dokmentaci v jazyce HTML, je možné používat pro formátovaní HTML elementy, jako `<p>`, `<code>`, `<pre>`, `<b>` apod.

```
/**
 * Metoda převede řetězec skládající se z textové reprezentace čísla
 * ve <b>dvojkové</b> soustavě na hodnotu typu <code>int</code>.
 */
```

Několik doporučení pro psaní komentářů:

- Komentář by měl vysvětlovat, co daná část kódu dělá, nikoliv jak. Protože to, jak kód funguje by mělo být patrné přímo z kódu.
- Komentář by měl obsahovat informace, které na první pohled nejsou zjevné.

```
public class Car {
  /** registrační značka */
  private String regNo;

  /** registrační značka dle zákona 361/2000 Sb., o provozu na pozemních komunikacích */
  private String regNo;
}
```

  V prvním případě je komentář do značné míry zbytečný, protože informaci, že v atributu `regNo` je registrační značka, která musí splňovat nějaké konkretní podmínky. V druhém případě komentář poskytuje podrobnější informaci o tom, jaké hodnoty v atributu hledat.

- Komentáře by měly být použity jen na důležité informace, ať není kód plný informační „vaty", která nemá žádnou informační hodnotu.
- Žádný komentář je lepší než neaktuální nebo chybný komentář.
- Potřebujeme-li psát dlouhý vysvětlující komentář, je to příznak toho, že bychom měli kód přepsat a zjednodušit. (Nejspíš z důvodu, že je porušen princip jedné zodpovědnosti a naše třída nebo metoda dělá více věcí současně.)

[3] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html`

- Je-li těžké v komentáři vysvětlit, co program dělá, je to opět příznak toho, že bychom měli kód zjednodušit.
- Je-li potřeba psát dovnitř metod vysvětlující komentáře, je na místě opět kód přepracovat, aby byl srozumitelnější. Například v následujícím kódu

```
// pokud ma uzivatel nejakou slevu
if (((user.getAge() < 26) && user.isStudent()) || (user.getAge() > 60)) { }
```

je vhodnější podmínku přesunout do samostatné (pomocné) metody.

```
if (hasDiscount(user)) { }
```

2 Testování

S nějakou formou testování programů se nevyhnutelně musel setkat každý, kdo zkousil vytvořit nějaký program. Například v předchozích seminářích bylo potřeba ověřit, že námi vytvořené programy fungují správně. Tento typ testování byl postaven na tom, že jsme si pro nějaké vybrané hodnoty ověřili, že vrácený výsledek odpovídá nebo neodpovídá našim požadavkům. Je otázka, jestli takto provedené testování bylo správné a efektivní. To znamená, zda-li jsme vyzkoušeli všechny možné varianty (nejspíš ne) a zda-li čas strávený zadáváním zkušebních hodnot a opakovaným spouštením programu byl využit efektivně (nejspíš ne).

Jazyk Java (a podobně další jazyky) nám umožňuje přistoupit k testování zodpovědněji, systematičtěji a hlavně efektivněji. Naším cílem je dosáhnout toho, že v každý okamžik můžeme otestovat, že se náš program chová podle očekávání, tak jako bychom testování prováděli sami.[4]

Úplným základem testování jsou tzv. *jednotkové testy* (unit testy), které testují, zda ty nejmenší jednotky programu (metody, případně třídy) se chovají správně.[5] Testování metod probíhá tak, že:

- zavoláme metodu se zadanými parametry
- a testujeme, zda návratová hodnota odpovídá očekávané hodnotě.

Při testování tříd:

- nastavíme atributy třídy,
- zavoláme testovanou metodu,
- a zjišťujeme, zda se stav atributů změnil na očekávané hodnoty.

Protože tyto kroky potřebujeme provádět pro různé hodnoty a to opakovaně, ideálně při každé změně programu, je na místě tuto činnost automatizovat. A protože chceme testovat části kódu naprogramované v Javě,

[4] Bobužel výsledky z oblasti teoretické informatiky nám nedovolující obecně dokázat správnost programu.

[5] Tento předpoklad je přirozený, protože nebudou-li v pořádku ty nejmenší jednotky, nemůže být v pořádku ani celý program.

dává smysl si napsat si taktéž v Javě jednoduché malé programy, které testování budou provádět automaticky, kdykoliv bude potřeba.

Aby se tvorba takových testu usnadnila, byl navržen, mimo jiné, framework JUnit, který řeší rutinní záležitosti spojené s testováním a je integrován do vývojových prostředí.

2.1 Framework JUnit

V pojetí frameworku JUnit jsou testy běžné třídy jazyka Java, ve kterých jsou metody představující jednotlivé testy. V těchto metodách můžeme vytvářet instance tříd, volat metody dle potřeby a ověřit, jestli jsou splněny naše předpoklady, například, jestli odpovídá návratová hodnota zadaným argumentům.

Jak by test mohl vypadat pro metodu `binaryToInt` ukazuje následující kód:

```
1  import static org.junit.jupiter.api.Assertions.*;
2
3  import org.junit.jupiter.api.Test;
4
5  import cz.upol.jj1.lecture07.Binary;
6  import cz.upol.jj1.lecture07.IntegerOverflowException;
7  import cz.upol.jj1.lecture07.InvalidDigitException;
8
9  class BinaryTest {
10
11   @Test
12   void binaryToIntTest() throws IntegerOverflowException, InvalidDigitException {
13     assertEquals(5, Binary.binaryToInt("101"));
14     assertEquals(10, Binary.binaryToInt("1010"));
15     assertEquals(15, Binary.binaryToInt("1111"));
16   }
17 }
```

Na řádcích 11 až 16 máme metodu, která otestuje naši metodu pro tři vybrané hodnoty. Že se jedná o testovací metodu je určeno anotací `@Test`, která je uvedena před samotnou deklarací metody.

Poznámka 3 *Anotace jsou dodatečné informace (metadata), která můžeme připojit k jednotlivým elementům jazyka (třídám, metodám, atributům, argumentům apod.) a můžeme pak s nimi dál pracovat. Podrobněji se anotacím a práci s nimi budeme věnovat v kurzu Jazyk Java 2. Pro základní úvod do problematiky můžete nahlédnout do Magazínu Katedry informatiky č. 15 (září 2021).*[6]

Metoda `assertEquals` ověřuje, zda návratová hodnota (druhý argument) odpovídá očekávané hodnotě (první argument). Pokud si hodnoty nejsou rovny, test selže.

[6] `http://www.inf.upol.cz/magazin/magazin-15.pdf`

Podobně bychom mohli testovat, jestli je hodnota `true` nebo `false`, pomocí metod `assertTrue` a `assertFalse`. Pro testování polí je potřeba použít metodu `assertArrayEquals`.

Všimněme si, že naše třída `BinaryTest` má jako rodiče třídu `Object`. Odkud se metody `assertEquals` a další vzaly? Může za to první řádek testu začínající `import static`. Deklarace `import static` nám zpřístupní zadané statické metody tak, jako by byly deklarované v aktuální třídě. V tomto případě je použita `*` a jsou zpřístupněny všechny metody ze třídy `org.junit.jupiter.api.Assertions`. Mohli bychom být ale konkrétní a zpřístupnit jen některé metody například:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

Testování výjimek je komplikovanější a používá pokročilejší konstrukty jazyka.

```
assertThrows(InvalidDigitException.class, () -> Binary.binaryToInt("1102"));
```

Typ očekáváné výjimky určíme pomocí názvu třídy a speciálního atributu `class`, který obsahuje objekt popisující danou výjimku, a kód, který má testovat výjimku umístíme do lambda výrazu, který nemá žádný argument, viz `() ->` . Podrobněji se lambda výrazům budeme věnovat v některém z dalších seminářů.

2.2 Složitější testy

Často se stává, že potřebujeme nastavit objekt do nějakého stavu nebo připravit vzájemně provázené objekty. To s frameworkem JUnit není nijak problematické, protože testovaný kód i testy jsou objekty v Javě. V čem nám může JUnit usnadnit práci, je opakovaná inicilizace objektů. Ukažme si to na příkladu testů třídy `Car` z druhého semináře, do které si doplníme metodu pro zjištění aktuální rychlosti.

```
public class Car {
  private final String plateNo;
  private final String color;
  private int speed;

  public Car(String plateNo, String color) {
    this.plateNo = plateNo;
    this.color = color;
    this.speed = 0;
  }
  public void accelarate(int speed) {
    this.speed += speed;
  }
  public void brake(int speed) {
    this.speed -= speed;
    if (this.speed < 0) this.speed = 0;
```

```
  }
  public int getSpeed() {
    return speed;
  }
}
```

Tento test by mohl vypadat následovně:

```
class CarTest {

  private Car car;

  @BeforeEach
  public void setUp() {
    car = new Car("AAA 1234", "blue");
    car.accelarate(10);
  }
  @Test
  void accelerateTest01() {
    car.accelarate(10);
    car.accelarate(15);
    assertEquals(25, car.getSpeed());
  }
  @Test
  void brakeTest01() {
    car.accelarate(10);
    car.brake(6);
    assertEquals(4, car.getSpeed());
  }
  @Test
  void brakeTest02() {
    car.accelarate(15);
    car.brake(20);
    assertEquals(0, car.getSpeed());
  }
}
```

Narozdíl od předchozího testu, zde máme atribut `car`, který obsahuje objekt na němž budeme testovat chování metod třídy `Car`. Tento objekt je nastaven v metodě `setUp`, která je označena anotací `@BeforeEach`, která zajistí, že je metoda `setUp` bude zavolána před každým testem.[7] V metodě `setUp` vytvoříme před

[7] Pro inicializaci hodnot atributů nemůžeme použít konstruktor, protože instance třídy `CarTest`, bude nejspíše vytvořena jen

každým testem novou instanci a následně nastavíme počáteční stav, čím si usnadníme psaní dalších testů.

2.3 Testování vstupů a výstupů

Při testování vstupů a výstupů bychom měli být obzvlášť na pozoru. Například při testování zápisu nebo čtení souboru, není dobré pracovat přímo se skutečnými soubory, protože takové operace mohou selhat, například protože nemáme právo zapisovat nebo číst data z disku, a výsledek takového testu by mohl být falešně negativní. V takových případech se nám hodí možnost nepracovat přímo se soubory, ale načítat nebo ukládat data z/do paměti.

Připomeňme si metodu `writeEmployees(Writer writer, Employee[] employees)` z minulého semináře, která nepracuje přímo se souborem, ale abstraktní třídou `Writer`, která může zapsat data kamkoliv. V našem případě se hodí třída `StringWriter`, kdy následně můžeme srovnat výstup metody `writeEmployees` s námi očekávanou hodnotou.

```
StringWriter buf = new StringWriter();
writeEmployees(buf, new Employee[] {
                new Employee("Alice", 20, 1234.56),
                new Employee("Bob", 42, 7890.23) });
assertEquals("""
Alice,20,1234.56
Bob,42,7890.23
""", buf.toString();
```

2.4 Doporučení pro psaní testů

- Testy měly být součástí každého projektu. Obvykle jsou umístěny v samostatném adresáři `test`, který je na úrovni adresáře `src`.
- Testy by měly být co nejjednodušší a neměly by obsahovat nějakou logiku (ve smyslu výpočtu). Jinak hrozí, že v testu bude chyba a budeme získávat falešně pozitivní nebo falešně negativní výsledky.
- Testy fungují i jako forma dokumentace, protože se můžeme podívat, jak se třída nebo metoda používá.
- Nemá cenu testovat jednoduché metody jako jsou gettery a settery, které pouze nastavují nějakou hodnotu. Podobně nemá smysl testovat správnost tříd typu `Record` nebo `Enum`, pokud neobsahují nějakou logiku.
- Testy by měly testovat veřejné rozhraní třídy. (Ale tak, aby otestovaly i všechny soukromé metody.)
- Pokud je psaní jednotkového testu pro náš kód příliš komplikované a vyžaduje složitou inicializaci, je to příznak toho, že bychom měli zvážit přepracování třídy nebo metody, protože má příliš mnoho závislostí a jeji používání nebo ladění bude komplikované.

jednou, pro potřeby frameworku JUnit.

- Je důležité testovat i mezní případy a nejlepé všechy řádky kódu. Proto je dobré testování spojit s nějakým nástrojem, který sleduje *pokrytí kódu* (code coverage), např. EclEmma, JaCoCo, apod.
- Psaní testů nás může upozornit na některé mezní případy, které nás při psaní programu nenapadly. Například, jak by se měla metoda `Binary.binaryToInt` chovat, když bude zadán prázdný řetězec?