Doporučené postupy v programování

Testování

Unit testing · Testovatelný kód

Lubomír Bulej

KDSS MFF UK

Cíle testování

Hlavní cíl

- včasná detekce chyb v programu
- ve výsledku snížení počtu chyb v programu

Vedlejší cíle (často důsledek testování)

- explicitní dokumentace chování/kontraktů
- zlepšení kvality kódu (a designu)

Testovat ručně nebo automaticky?

Všechny testy by měly být automatické

- spustitelné jedním příkazem
- vrací jednoznačný výsledek
- různé metody automatizace:
  - skripty
  - testovací frameworky
  - ...

Poznámky:
Jednoznačný výsledek je buď "test prošel", nebo "test neprošel".

Vyplatí se psát testy?

Záleží na ceně chyb a jejich nápravy

- Příklad 1: Webová aplikace
- Příklad 2: Komponenta/knihovna/framework
- Příklad 3: Software kardiostimulátoru

Poznámky:

V případě obyčejné webové aplikace se formalizace testování pravděpodobně nevyplatí – data v aplikaci obyčejně nebývají důležitá, chyby nebývají fatální, uživatelé je poměrně rychle odhalí a jejich oprava je snadná. Samozřejmě, situaci může ovlivnit např. fakt, že aplikace je placená (uživatelé ze sebe nenechají dělat pokusné králíky a utečou ke konkurenci) a nebo se v ní manipuluje s důležitými údaji (často peníze).

U jakékoliv komponenty, která je používána na více místech už je cena chyb větší, protože je automaticky budou obsahovat všechny programy, které komponentu používají, a po opravě nalezených chyb se všechny tyto programy budou muset opravit také. Důkladné testování se zde už pravděpodobně vyplatí.

V případě software kardiostimulátoru, nebo obecně jakéhokoliv jiného software, kde jsou v sázce zdraví či životy lidí a nebo velké finanční částky, mnohdy investice do testování přesahují investice do vývoje samotného programu. Často se používají metody z kategorie formálních specifikací a model checkingu.

Rozdělení testů

Rozsah

- Unit testy
- Testy komponent
- Integrační testy
- Systémové testy

Náplň

- Regresní testy
- Výkonnostní testy
- Zátěžové testy
- Testy použitelnosti
- ...

Poznámky:

Unit testy testují malé jednotky programu, typicky jednotlivé třídy a metody v nich. Testy komponent testují větší funkční celky, integrační testy pak jejich vzájemnou interakci. Systémové testy testují celý vyvíjený produkt jako celek v jeho finální konfiguraci a cílovém prostředí.

Regresní testy často vznikají implicitně, každý test spouštěný automaticky se totiž stává testem regresním.

Rozdělení testů

Black box testing

- bez přístupu k testovanému kódu
- testy psány často před napsáním kódu
- typicky oproti specifikaci

White box testing

- s přístupem k testovanému kódu
- testy psány nutně po napsání kódu
- typicky podle struktury kódu
  - snaha o co největší *pokrytí kódu*
  - jde lépe psát testy, co opravdu najdou chyby

Problémy testování

Opačný cíl než psaní programu

- místo fungujícího programu ho chceme rozbít

Nedokazuje správnost/bezchybnost programu

- špatné testy, nedostatečné pokrytí kódu

Nezlepšuje kvalitu, jen indikátor

- *Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often.* – Steve McConnell

Není až tak efektivní, jak si lidé myslí

- typicky odhalí jen 50–60 % chyb

Poznámky:
Opačný cíl než psaní programu je hlavně problém psychologický. Protože programátor chce, aby program fungoval, může mít (zejména je-li pod tlakem) tendenci testy vynechávat nebo odbývat, případně se v nich záměrně vyhnout některým situacím, o nichž ví, že v programu způsobí problémy.

Unit testing

Poznámky:

Pokud si budete chtít unit testy opravdu vyzkoušet v praxi, a budete při tom využívat nástroj JUnit, projděte si nejdříve následující odkazy (nejlépe v uvedeném pořadí).

- http://junit.sourceforge.net/doc/testinfected/testing.htm (úvod)
- http://junit.sourceforge.net/doc/cookstour/cookstour.htm (asi nejzajímavější z odkazů)
- http://junit.sourceforge.net/doc/faq/faq.htm (integrace s Antem)
- JUnit Anti-patterns (co v testech nedělat)

Unit testing

Testování funkčnosti malých jednotek kódu

- typicky na úrovni tříd a jejich metod
- ověření kontraktů tříd/metod

Větší celky primárně složeny z menších

- chceme se ujistit, že "součástky" fungují správně
- důraz na testování odspodu (bottom-up)

Proč je důležité testovat?

Řada pozitivních důsledků

- zjednodušuje integraci
- testy slouží jako dokumentace
- pomáhá oddělit rozhraní od implementace
- umožňuje/usnadňuje změny kódu/designu
  - změna požadavků je jedna z věcí, o které víme, že určitě nastane
  - bez testů je těžké přizpůsobit design novým požadavkům a nic nerozbít
  - s testy se nemusíme bát refaktorovat

Co testování brání?

- zpomaluje psaní a úpravy kódu
- vyžaduje disciplínu

Poznámky:

K podpoře změn: Nejčastější důvod ke snaze neměnit, co funguje, je riziko zanesení chyb do programu. Pokud používáme unit testy, vytvářejí pod programem jakousi záchrannou síť a nově zanesené chyby většinou odhalí. "Většinou" samozřejmě není "vždy", ale pokud jsou testy dobře napsané, je pravděpodobnost zanesení obvykle dostatečně malá, aby změnám nebránila. Celkovým výsledkem používání unit testů jsou obvykle častější refaktorizace a tedy i čistější kód.

Jak vypadá unit test?

Testovaný kód: SimpleStack.java

```
public final class SimpleStack <T> {
  private static final String STACK_EMPTY_MESSAGE = "Stack is empty.";

  private List <T> items = new LinkedList <T> ();

  public boolean isEmpty () {
    return items.isEmpty ();
  }

  public T top() {
    if (items.isEmpty ()) {
      throw new IllegalStateException (STACK_EMPTY_MESSAGE);
    }

    return items.get (0);
  }

  public T pop() {
    if (items.isEmpty ()) {
      throw new IllegalStateException (STACK_EMPTY_MESSAGE);
    }

    return items.remove (0);
  }

  public void push (T item) {
    items.add (0, item);
  }
}
```

Jak vypadá unit test?

Testovací kód: SimpleStackTest.java

```
public final class SimpleStackTest {

  private SimpleStack <String> emptyStack;
  private SimpleStack <String> fullStack;

  @Before
  public void setUp () {
    emptyStack = new SimpleStack <String> ();

    fullStack = new SimpleStack <String> ();
    fullStack.push ("A");
    fullStack.push ("B");
    fullStack.push ("C");
  }

  @Test
  public void emptyStackIsEmpty() {
    assertTrue(emptyStack.isEmpty());
  }

  @Test
  public void fullStackNotEmpty() {
    assertFalse(fullStack.isEmpty());
  }

  @Test(expected=IllegalStateException.class)
  public void topThrowsExceptionOnEmptyStack() {
    emptyStack.top();
  }

  @Test
  public void topReturnsTopItem() {
    assertEquals("C", fullStack.top());
  }

  ...
}
```

Co bylo dřív? Vejce nebo slepice?

Testovaný kód nebo test?

- záleží na tom, koho se zeptáte…

Jedna z možností: Test-driven development

1. Napsat test
2. Spustit test – ověřit nesplnění
3. Napsat kód – minimum pro splnění testu
4. Spustit test – ověřit splnění
5. Refaktorovat kód

Výhody TDD (pokud jej dokážete aplikovat)

- rychlá zpětná vazba: nutí k neustálému přemýšlení nad designem
- vede k lepším rozhraním (test = uživatel)
- zaručuje 100% pokrytí kódu

Poznámky:

Několik zajímavých odkazů k tématu:

- Test-driven development – článek na Wikipedii
- Red-Green-Refactor
- The Roots of TDD
- James Carr: TDD Anti-patterns

Doslovná interpretace TDD může být kontraproduktivní, pokud chceme vyzkoušet více variant řešení nějakého problému, zhodnotit je podle nějakého kritéria a následně jednu z variant použít a jiné zahodit. TDD může omezit experimentování s kódem v tom smyslu, že pokud máme k různým variantám kódu rovnou psát i testy, zvětší se náklady na každou variantu, což může vést až k tomu, že se experimentování nevyplatí. Někdy může být tedy výhodnější metodiku porušit, ozkoušet víc variant a teprve dodatečně k nim dopsat testy.

Pro inteligentního vývojáře může být také nepřirozené přemýšlet v tak malých úsecích návrhu/kódu, jak často demonstruje/vyžaduje TDD; lépe by se mu pracovalo s většími celky a možná i na abstraktnější úrovni, než je kód. Pro takového vývojáře může být TDD ztráta času a tedy i produktivity.

Oba předchozí odstavce lze shrnout pod obecnou poučku, že žádná metodika není dokonalá a vhodná pro všechny situace.

Pokrytí

Definice pokrytí

- Řádky? Příkazy? A co stavy? Komplikované…
- Většina nástrojů pragmaticky: řádky

Usilovat o 100% pokrytí?

- teorie rozbitých oken
  - unit testy musí pokrývat 100 % kódu
- zákon klesajících výnosů (law of diminishing returns)
  - zvýšit pokrytí z 95% na 100% vyžaduje typicky N × 5% úsilí
- kompromis: snaha o 100 % je užitečná, není nutné dosáhnout za každou cenu

Poznámky:

Odkazy k tématu:

- Advocating the use of code coverage
- Lessons learned on the road to 100% code coverage

Co patří do unit testů

Hlavní náplní jsou ...

- testy kontraktů tříd/metod
- testy všech větví zpracování
- testy speciálních případů
- ...

Navíc převeďte na unit test...

- cokoliv si kdykoliv zkusíte otestovat ručně
- všechny testovací a ladící výpisy
- testy použité k verifikaci chyb a jejich oprav

Co nepatří do unit testů

Test není unit test, pokud...

- pracuje s databází
- komunikuje po síti
- sahá na souborový systém
- není schopen běžet současně s ostatními testy
- je potřeba kvůli němu nastavovat běhové prostředí (např. editovat konfigurační soubory)

Testy, které toto dělají jsou také potřeba, a můžou být napsány s pomocí unit-testing frameworku, ale nejsou to unit testy.

Poznámky:
Zdroj: Unit Test Rulz

Jak se nepatřičným věcem vyhnout?

Stubs/Mock objects

- simulují objekty v okolí testovaného objektu
- implementují stejný interface jako plnohodnotný objekt
- nepřímo zlepšují kvalitu kódu
  - vynucují programování proti interfacům
  - vynucují možnost substituovat komponenty (modularita)
- existují frameworky na vytváření mock objektů
  - jMock, EasyMock

Poznámky:

Odkazy:

- Alexander Chaffee, William Pietri: Unit testing with mock objects
- Gary Pollice: Using mock objects for complex unit tests
- Martin Fowler: Mocks Aren't Stubs

Zásady pro psaní testů

Testy musí být kompletně automatizované

- minimalizuje "transakční náklady" na běh testů
- umožňuje integraci do CI/CD procesů

Nevynalézejte kolo: použijte existující frameworky

- JUnit (Java)
- NUnit (.NET)
- Test::Unit (Ruby)
- PyUnit (Python)
- ...

Zásady pro psaní testů

Obecná struktura testu (AAA)

1. příprava situace (arrange)
2. provedení testované operace a získání výsledků (act)
3. test shody očekávaných a skutečných výsledků (assert)

Obecné vlastnosti testů

- testy musí být viditelně správné
  - kód testu musí být jednoduchý, ne složitejší než testovaná funkce
  - složitejší přípravu tlačit do infrastruktury → dá se testovat

- testy musí testovat kontrakt, nikoliv implementaci
- testy musí pokrýt i okrajové případy

Zásady pro psaní testů

Snažte se kód co nejvíc "provětrat"

- chybná vstupní data, okrajové případy
- vyhazované výjimky, komplikované pasáže

Používejte správně asserty

- použijte assert, který je nejvíce specifický (JUnit):
  - `assertEquals`, `assertNotEquals`, `assertSame`, `assertNotSame`, `assertNull`, `assertNotNull`, `assertFalse`, `assertTrue`
- každý test by měl obsahovat assert
- pro různé situace by mělo být více testů

Neodchytávejte výjimky

- nechte je probublat ven, ať shodí test

Poznámky:

K používání správných assertů: Nebojte se si sadu assertů rozšířit, pokud zjistíte, že používáte nějaký assert spolu s pomocným kódem okolo na více místech.

Zásady pro psaní testů

1 třída kódu ~ 1 třída testů

- není striktní

1 metoda kódu ~ sada testovacích metod

- neplatí: 1 metoda kódu == 1 metoda testů

Často opomíjené: kód testů je taky kód!

- čistota, struktura, názvy, eliminace duplicit…

Zdrojový kód testů držte blízko testovaných tříd

- testovací třídy by měly být ve stejném package (Java) jako testované
  - umožňuje využít package-private mechanizmus pro přístup k detailům
- stejný zdrojový adresář: testy jsou "hned vedle" testované třídy
- ale: nemíchejte přeložený produkční a testovací kód

Poznámky:

K třídám kódu a testů: Toto pravidlo nemusí platit vždy, můžeme mít víc tříd testů na jednu třídu kódu. Pozor ale, je to často signál, že třída má víc různých zodpovědností. Podobné pravidlo o metodách neplatí – metody mají často víc různých aspektů, které je vhodné testovat zvlášť.

Testovatelný kód

Poznámky:

Ne všechen kód je dobře testovatelný. Aby tomu tak bylo, je potřeba tomu uzpůsobit design, aby neobsahoval velké "slitky", do kterých není vidět a není možné jejich chování testovat po malých částech. Základní pravidla hezky sepsal Miško Hevery ve svém blogu.

Jak psát testovatelný kód

V kódu musí zůstat "švy"

- hranice, podél kterých je možné kód "rozebrat" a složit jinak
- umožňuje testování tříd v izolovaně od zbytku systému

Časté chyby

- konstruktor dělá opravdovou práci
- spolupracující objekty je nutno "dolovat" z jiných objektů
- globální stav & nadužívání singletonů
- třída dělá příliš mnoho

Možno aplikovat i na metody

- metoda dělá příliš mnoho věcí — prostor pro funkční dekompozici
  - nahraďte odstavce kódu voláním funkcí/metod

Poznámky:

- Guide: Writing Testable Code

Konstruktor dělá opravdovou práci

Jakou práci?

- vytváření/inicializace spolupracujících objektů
- komunikace s jinými službami
- podmíněná logika pro inicializaci stavu

Práce v konstruktoru odstraňuje "švy" potřebné pro testování.

- odvozené třídy/mock objekty dědí nepotřebné/nechtěné chování
- znemožňuje instanciaci nebo změnu spolupracujících objektů v testu

Konstruktor dělá opravdovou práci

Varovná znamení (code smells)

- klíčová slova `new` v konstruktoru nebo deklaraci atributů
- volání statických metod v konstruktoru nebo deklaraci atributů
- cokoliv jiného než přiřazení do atributů v konstruktoru
- neúplná inicializace objektu po skončení konstruktoru (pozor na inicializační metody)
- řídící příkazy v kódu konstruktoru (cykly, podmínky)
- konstrukce grafu objektů v konstruktoru místo ve factory metodě
- přidání nebo použití inicializačního bloku

Konstruktor dělá opravdovou práci

Proč to vadí?

- porušuje princip "single responsibility"
  - vytváření grafu spolupracujících objektů je zodpovědnost sama o sobě
- testování konstruktoru je obtížné
  - práci je nutné vykonat vždy při vytváření objektu
  - změny v konstruktoru je obtížné pokrýt kód testy
- dědičnost a předefinování (inicializační) metody je pořád špatné
  - předefinované metody se špatně testují
- spolupracující objekty jsou hardcoded, tedy vnucovány testům
- více konstruktorů (některé jen pro testy) jsou stále špatné
  - ostatní třídy budou špatné konstruktory stejně používat

Příklad: používání `new` v konstruktoru

Obtížně testovatelný kód

```
/*
 * Basic new operators called directly in the class' constructor.
 * (Forever preventing a seam to create different kitchen and
 * bedroom collaborators).
 */
class House {
  Kitchen kitchen = new Kitchen();
  Bedroom bedroom;

  House() {
    bedroom = new Bedroom();
  }

  // ...
}


/*
 * An attempted test that becomes pretty hard.
 */
class HouseTest extends TestCase {
```

```
  public void testThisIsReallyHard() {
    House house = new House();
    // Darn! I'm stuck with those Kitchen and Bedroom
    // objects created in the  constructor.

    // ...
  }
}
```

Příklad: používání `new` v konstruktoru

Testovatelný a flexibilní design

```
class House {
  Kitchen kitchen;
  Bedroom bedroom;

  // Have Guice create the objects and pass them in
  @Inject
  House (Kitchen k, Bedroom b) {
    kitchen = k;
    bedroom = b;
  }
  // ...
}


/*
 * New and Improved is trivially testable, with any
 * test-double objects as collaborators.
 */
class HouseTest extends TestCase {
  public void testThisIsEasyAndFlexible() {
    Kitchen dummyKitchen = new DummyKitchen();
    Bedroom dummyBedroom = new DummyBedroom();

    House house =
        new House(dummyKitchen, dummyBedroom);

    // Awesome, I can use test doubles that
    //   are lighter weight.

    // ...
  }
}
```

Spolupracující objekty je nutné "dolovat"

Odkud?

- z holderů, kontextů, a jiných "kitchen sink" objektů
- obecně z jiných objektů předaných jako parametr
- ze složitých struktur, které je nutno procházet

Varovná znamení (code smells)

- předávané objekty nejsou používané (slouží pouze k získání jiných objektů)
- porušení "Law of Demeter" – volání prochází strukturu objektů
- podezřelá jména: context, environment, principal, container, manager, ...

Spolupracující objekty je nutné "dolovat"

Proč to vadí?

- falešné API
  - zavislosti by měly být explicitní
- křehký kód v důsledku velkého počtu vazebních objektů
  - nutno znát objekty na cestě k požadovanému objektu
- testování je obtížné
  - nutno vyrobit kontext – hledá se minimální nutný kontext

Globální stav a nadužívání singletonů

Skrývá závislosti

- umožňuje "spooky actions at distance"

Varovná znamení (code smells)

- přidávání/používání singletonů
- přidávání/používání statických atributů a metod
- přidávání/používání statických inicializátorů
- přidávání/používání všelijakých "registries" a lokátorů služeb