Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Softwarová podpora Duck-testů

Plzeň 2013 Kateřina Štollová

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracovala samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 6. května 2013

. . . . . .

Kateřina Štollová

Abstract

Software support of Duck-tests. Currently it is possible to test the functionality of program using JUnit tests and the quality of source code using PMD. This thesis examines the possibility of testing the implementation by duck-tests. Duck-tests do not test the functionality of the program but the way of implementation. While using these tests you can determine if the student actually used the language structures which was specified in the assignment. My task is to create two programs – the first to automatically generate duck-tests according to the teacher's pattern and the other to automatically launch and evaluate the tests.

Abstrakt

Softwarová podpora Duck-testů. V současné době lze testovat funkčnost programu pomocí JUnit testů a kvalitu zdrojového kódu pomocí nástroje PMD. Tato bakalářská práce zkoumá možnosti testování implementace pomocí duck-testů. Duck-testy netestují funčnost programu, ale to, zda je naimplementován správně. Pomocí těchto testů lze zjistit, zda student skutečně použil jazykové konstrukce, které měl určené v zadání. Mým úkolem je vytvořit dva programy – jeden pro automatické generování duck-testů podle učitelova vzorového příkladu a druhý na jejich automatické spouštění a vyhodnocení.

Obsah

1 Úvod

Práce se věnuje testování studentských úloh, zejména v počátcích výuky programování, kdy je třeba u těchto úloh kontrolovat dodržení zadání. Práce se zaměřuje pouze na programovací jazyk Java, protože se jedná o první jazyk, ve kterém se studenti studijního programu Inženýrská informatika učí programovat.

Studentské úlohy se běžně testují na funkčnost programu, např. pomocí jednotkových testů (tzv. JUnit testy). Druhým standardně používaným testem je test kvality zdrojového kódu, např. pomocí nástroje PMD. V současné době ale také existují nástroje pro další úroveň testování, kterou je testování dodržení zadání.

Smysl testování dodržení zadání je zřejmý. Při výuce programování se studenti učí používat různé jazykové konstrukce. Pomocí funkčních testů a testů kvality zdrojového kódu nelze ověřit, zda student opravdu výsledku dosáhl pomocí postupu požadovaného v zadání úlohy, nebo použil jakýkoliv (často velmi nevhodný) způsob, pouze aby zajistil správnou funkčnost programu. Typickým příkladem nevhodných postupů je používání `public` globálních proměnných pro předávání parametrů do podprogramů místo využití skutečných a formálních parametrů.

Pokud student použije pro zajištění funkčnosti programu podobný nevhodný způsob, je důvod, proč byla tato konkrétní úloha studentovi zadána, prakticky ztracen. Testování dodržení zadání dokáže do značné míry zjistit, zda student opravdu použil v zadání požadované jazykové konstrukce.

Je zřejmé, že tento způsob testování nemůže být použit pro širokou škálu předmětů a studentských programů, ovšem je mimořádně vhodný pro jednoduché úlohy zadávané v úvodních předmětech výuky programování.

Základním požadavkem pro sestavení odpovídajícího testu je existující vzorové zadání, které připravuje vyučující. Tento požadavek je ale možné velmi snadno splnit, takže není nijak limitující.

2 Teoretický úvod

2.1 Použité technologie

Pro vypracování bakalářské práce byly použity následující technologie a programy. Pro testy implementace je používán framework Duckapter [1], jehož základem jsou anotace a návrhový vzor adaptér. Tento framework bude popsán dále.

Již ze zadání byl určen programovací jazyk, kterým je Java. Použita byla Java 1.7.0_15, která má několik nových funkcí, hlavně pro práci se soubory. Použitým vývojovým prostředím byl Eclipse Juno. Grafické uživatelské rozhraní bylo vytvořeno pomocí Swingu. Pro testování se využívá frameworku JUnit testů. Jejich princip bude také objasněn dále.

2.2 Duck-testy a duck-typing

Duck-testy jsou testy implementace, které využívají duck-typing. Duck-typing popisuje Oraný [1] následovně:

> V programování pomocí objektově orientovaných programovacích jazyků je kachní typování způsobem dynamického typování, ve kterém je správná sémantika určována nikoliv díky dědění od určité třídy či implementací určitého rozhraní, ale pomocí aktuálního souboru metod a vlastností. Název konceptu se odvolává na termín kachní test, který je připisován Jamesi Whitcombovi Rileyovi a který je možné formulovat: "Pokud vidím opeřence, který chodí jako kachna, plave jako kachna a kváká jako kachna, nazývám tohoto opeřence kachnou."
>
> Při kachním typování se zabýváme pouze vlastnostmi objektu, který využíváme, nikoliv druhem objektu samotného. Například v jazycích, které nepodporují kachní typování, můžeme vytvořit funkci, která přebírá objekt druhu `Kachna` a zavolá metody `jdi()` a `kvákni()`. V jazycích podporujících kachní typování by podobná funkce přebírala objekt jakéhokoliv druhu a volala na něm metody `jdi()` a `kvákni()`. Pokud by objekt dané metody neměl, nastala by výjimka za běhu programu.

2.2.1 Návrhový vzor adaptér

Ve frameworku duckapter se využívá návrhový vzor adaptér. Jeho princip je podrobně popsán v [2]. Jedná se o jeden z návrhových vzorů skupiny GoF – Gang of Four [3].

Existuje-li třída, která nevyhovuje danému rozhraní, je řešením použití adaptéru, jehož úkolem je konvertovat rozhraní třídy na požadované rozhraní. Adaptér se vkládá mezi uživatelskou třídu a třídu, která má splňovat požadované rozhraní. V Javě se jedná o všechny obalové třídy – například kontejnery z balíku `java.util` přijímající pouze objekty. Do kontejnerů lze vkládat primitivní typy tím, že se konvertují na příslušný obalový typ.

Adaptér je tedy prostředník mezi třídou, která nesplňuje určité rozhraní, a uživatelskou třídou, která po třídě dané rozhraní požaduje. Dovoluje komunikovat těmto jinak nekompatibilním třídám. Adaptér se využívá v následujících případech:

- chcete použít třídu, jejíž rozhraní neodpovídá rozhraní, které potřebujete,
- chcete vytvořit třídu, která bude spolupracovat s neznámými třídami – rozhraní mohou být nekompatibilní,
- chcete použít objekt v prostředí, které očekává jiné rozhraní objektu.

2.2.2 Framework Duckapter

Pro testy implementace se využívá framework duckapter, který je podrobněji popsán v [1]. Jeho princip popíšu proto jen stručně. Požadované vlastnosti třídy se definují v rozhraní. Obvykle není v rozhraní možné popsat požadované atributy, konstruktory a nebo statické metody, to je řešeno pomocí anotací, které umí framework zpracovávat. Místo daných modifikátorů se použijí právě anotace. Všechny používané anotace jsou popsány dále.

Při testu framework ověřuje, zda třída obsahuje všechny prvky odpovídající rozhraní. K tomu slouží metoda `wrap()`, která přijímá testovanou třídu a požadované rozhraní. Pokud všechny prvky třídy odpovídají danému rozhraní, projde třída testem. V opačném případě je vyhozena výjimka `WrappingException`.

2.3 Anotace

Anotace, nazývané také metadata, se v Javě vyskytují od verze 5.0 viz [4]. Jedná se o modifikátory, které se, stejně jako ostatní modifikátory, vkládají do zdrojového kódu před anotované položky. Označují se znakem @, který jim předchází. Lze anotovat všechny prvky, které mohou být označeny modifikátory, tj. třídy, metody, parametry metod a podobně. Dokonce lze anotovat i balíky. Anotace se prostě přidají před ostatní modifikátory objektu. Podle konvencí by měly být úplně první.

Před použitím je nutné anotaci vytvořit. Podrobný postup je popsán v [5]. Při vytváření anotací se definují vlastnosti dané anotace. K této definici se používají tzv. metaanotace z balíčku `java.lang.annotation`. Při definici anotací se její vlastnosti určí následujícími metaanotacemi:

- metaanotace `@Documented` určuje, zda se anotace objeví v dokumentaci,
- metaanotace `@Retention` určuje, jak je anotace uchovávána. Jestli je uložena pouze ve zdrojovém kódu nebo se zapíše i do `class` souboru, aby mohla být zpracovávána za běhu programu,
- metaanotace `@Target` říká jaké prvky lze danou anotací označit

Jako další se musí určit parametry dané anotace, má-li nějaké. U parametrů se určuje jejich typ a může se nastavit výchozí hodnota.

2.3.1 Princip anotací

Anotace neovlivňují vykonávání samotného kódu, ale jsou většinou určené pro speciální nástroje, které s nimi pracují. Metaanotace `@Retention` určuje, zda je anotace pouze součástí zdrojového kódu nebo se zapíše i do `class` soubrou. Anotace lze tedy zpracovávat několika způsoby – buď pomocí nástrojů pracujícími se zdrojovým kódem (např. javadoc), nebo pomocí nástrojů pracující s přeloženým `class` souborem, ať už při instalaci nebo za běhu programu.

2.3.2 Získávání anotací za běhu programu

Právě zpracovávání anotací při běhu programu využívá framework Duckapter. Pomocí reflexe (`java.lang.reflect`) je možné získat z přeloženého `class` souboru množství informací o původním zdrojovém kódu třídy. Kromě atributů, metod a konstruktorů, lze získat i jejich modifikátory, typy parametrů a typy návratových hodnot. A samozřejmě i anotace dané třídy.

2.4 JUnit testy

Pro otestování správné funkčnosti studenstkých úloh lze použít tzv. jednotkové testy, v případě Javy JUnit testy (viz [6]). Tyto testy se používají k otestování funkčnosti malých úseků kódu. Měly by být co nejjednodušší, typicky je třeba na otestování jedné třídy několik desítek JUnit testů.

2.4.1 Princip JUnit testů

JUnit testy mohou testovat pouze funkčnost metod, ke kterým mají přístup, tj. `public` a `protected`. Všechny testy týkající se jedné třídy se typicky sdružují do jednoho souboru jako metody. Jak již bylo zmíněno výše, každý test testuje pouze určitý úsek kódu. JUnit metody musí být `public void` a jsou označené anotací `@Test`, aby byly zařazeny mezi testovací případy.

V JUnit testech se porovnává očekávaná hodnota se skutečnou získanou z instance testované třídy. Musíme tedy mít k dispozici instanci této třídy. Testovací metoda typicky obsahuje jednu metodu ze třídy `org.junit.Assert`. Těchto metod je několik, v ducktestech jsem využila dvou – `assertNull` a `assertTrue`. Jak obě metody vypadají ukazuje kód 2.1.

```
assertNull(String message, Object object);
assertTrue(String message, boolean condition);
```

Kód 2.1: Hlavička dvou používaných metod JUnit frameworku

Pokud je výsledek testu negativní, metody `assert` generují výjimku a vypíše se chybová zpráva `message`. U `assertNull` je to v případě nenulového objektu `object`, u `assertTrue`

při nesplnění podmínky `condition`.

Konkrétní použití JUnit testů je ukázáno na příkladě pro třídu `Dum`, jejíž zdrojový kód je v příloze A.1. Nejprve se metodě `setUpBeforeClass()` připraví data, která se budou testovat, tj. vytvoří se instance třídy `Dum` s požadovanými vlastnostmi. Tato metoda je označena anotací `@BeforeClass`, která znamená, že se provede pouze jednou, a to před spuštěním testů.

V testu se konstroluje, zda vytvořený objekt doopravdy odpovídá objektu, který měl být vytvořen. Jsou k tomu použity dva testy – jeden kontrolující výšku a jeden kontrolující šířku objektu. Kontrola se provádí metodou `Assert.assertEquals()`, jejíž parametry jsou: chybová zpráva, očekávaná hodnota a skutečná hodnota. V případě neshody očekávané a skutečné hodnoty se vypíše chybová zpráva. Samotný test pak zobrazuje kód 2.2.

```
    private static Dum instance;
    private static Rozmer r;

    @BeforeClass public static void setUpBeforeClass() {
        r = new Rozmer(50, 30);
        instance = new Dum(r);
    }

    @Test public void testVyska() {
        assertEquals("Dům nemá správnou výšku!", 50, instance.getVyska()
            );
    }

    @Test public void testSirka() {
        assertEquals("Dům nemá správnou šířku!", 30, instance.getSirka()
            );
    }
```

Kód 2.2: Příklad jednoduchého JUnit testu

2.5 Využití testů pro testování studentských úloh

Pro testování studentských úloh se v předmětu KIV/OOP využívají JUnit testy, které ale testují pouze správnou funkčnost programu. Použitím frameworku Duckapter [1] lze připravit další sadu testů pro testování implementace, tj. zda student splnil přesně zadání.

Testy pro každou třídu se skládají ze dvou částí. První částí testů je rozhraní obsahující testované atributy, metody a konstruktory označené anotacemi. Každé jedno rozhraní popisuje prvky třídy pro jeden test. Tento popis je realizován právě pomocí anotací. Všechny dostupné anotace jsou popsané v dokumentaci k frameworku Duckapter, v této práci budou zmíněny pouze ty nejčastěji používané.

Druhou částí jsou již samotné testy. V mnou popsaných případech jsou realizovány pomocí JUnit testů, ale to není podmínkou. Tento způsob je použit z důvodu jednotného testovacího prostředí používaného v předmětu KIV/OOP.

2.5.1 Testování atributů

Při testování atributů musí být všechny požadované atributy popsané v rozhraní. Musí být označené anotací `@Field`, aby bylo poznat, že se jedná o atribut. Další anotace využívané při testování atributů jsou popsány v tabulce 2.1.

`@Field`	značí, že se jedná o atribut
`@Declared`	atribut je deklarovaný v testované třídě, nedědí se
`@Private @Public @Protected`	přístupová práva
`@StaticField @NonStatic`	statický nebo instanční atribut
`@Final`	konstanta

Tabulka 2.1: Anotace používané u atributů

Položky rozhraní vypadají následovně — všechny anotace, které se vztahují k atributu následované typem a jménem atributu. Jméno atributu se uvádí s předponou `get` a zakončuje závorkami jako funkce. Je třeba dávat pozor na malá a velká písmena, protože Java je case-sensitive a špatně zapsaný atribut by nebyl nalezen a byl by nahlášen jako chyba.

Podle tabulky 2.1 s anotacemi lze určit, že v příkladu zobrazeném v kódu 2.3 jsou testované

```
@Field @Declared @Public @StaticField @Final int getIMPL_SIRKA();
@Field @Declared @Private @StaticField int getpocet();
@Field @Declared @Private @NonStatic @Final int getcisloDomu();
@Field @Declared @Private @NonStatic int getvyska();
```

Kód 2.3: Příklad rozhraní pro atributy

následující atributy: veřejná statická konstanta `IMPL_SIRKA`, privátní statická proměnná `pocet`, privátní instanční konstanta `cisloDomu` a privátní instanční proměnná `vyska`. Všechny jsou typu `int`.

2.5.2 Testování metod

Rozhraní pro testování metod je velmi podobné deklaraci metody, jen se před ni vloží odpovídající anotace. Nejčastěji využívané anotace jsou vyobrazeny v tabulce 2.2.

`@Declared`	metoda je deklarovaná v testované třídě, nedědí se
`@Private @Public @Protected`	přístupová práva
`@Static @NonStatic`	statická nebo instanční metoda

Tabulka 2.2: Anotace používané u metod

V případě zobrazeném kódem 2.4 je ukázka rozhraní pro tři metody – instanční veřejná metoda `getVyska()` s návratovým typem `int`, statická veřejná metoda `setPocet(int pocet)` s návratovým typem `void` a veřejná instanční metoda `toString()` s návratovým typem `String`.

```
@Declared @Public @NonStatic int getVyska();
@Declared @Public @Static void setPocet(int pocet);
@Declared @Public @NonStatic String toString();
```

Kód 2.4: Příklad rozhraní pro metody

2.5.3 Testování konstruktorů

Testy konstruktorů jsou odlišné od testů atributů a metod. Konstruktory se označují pouze anotací `@Construktor`, v případě soukromého konstruktoru se vkládá navíc anotace `@Private`. Pokud je konstruktor veřejný, žádná speciální anotace se nevkládá. Rozhraní

pro test konstruktoru je zobrazeno v kódu 2.5. Jedná se o konstruktor třídy `Dum` se dvěma parametry a to `vyska` a `sirka` typu `int`.

```
@Constructor Dum newInstance(int vyska, int sirka);
```

Kód 2.5: Příklad rozhraní pro konstruktor

3 Příprava testů

3.1 Ruční příprava testů

Před existencí automatického generátoru duck-testů, bylo nutné testy vytvořit manuálně. Jako testovací případy bylo vybráno osm sad úloh z předmětu KIV/OOP z roku 2011/12. Výsledky průběhu testů jsou popsány v kapitole 6. Při ruční přípravě jsem zjistila, že se jedná o poměrně mechanickou činnost a nebude problém ji zautomatizovat.

Rozhraní se vytváří podle vzorové úlohy připravené vyučujícím a podle nich pak soubor s testy. Během ruční přípravy testů jsem vytvořila knihovní třídu `DuckUtility`, která upravuje chybová hlášení frameworku Duckapter do čitelnější podoby pro uživatele a poskytuje další funkce pro testování pomocí duck-testů.

3.1.1 Popis knihovní třídy `DuckUtility`

Nejpoužívanější metodou `DuckUtility` je nepochybně `upravZpravu()`, která z výpisu chyby vytvořené frameworkem Duckapter vytvoří uživatelsky příjemné hlášení obsahující popis chyb. Hlavička metody je zobrazena v kódu 3.1.

```
String upravZpravu(char typ, String zprava, String od, String kam);
```

Kód 3.1: Hlavička metody upravZpravu

Parametr `typ` určuje přímo, o jaký typ testu se jedná. Může nabývat následujících hodnot:

- 'K' pro statické konstanty,
- 'k' pro instanční konstanty,
- 'P' pro statické proměnné,
- 'p' pro instanční proměnné,
- 'G' pro getry,
- 'S' pro setry,

- 'M' pro obecné metody a
- 'O' pro zděděné metody, které musí být překryty.

Podle typu se tvoří zpráva obsahující nápovědu, jak detekovat chybu. Parametr `zprava` je chybová zpráva, vytvořená frameworkem při vytvoření výjimky, obsahující nadbytečné informace. Obsahuje název výjimky a seznam chybných parametrů tak, jak jsou definované v rozhraní. Příklad zobrazuje toto hůře čitelné hlášení:

```
java.lang.AssertionError: [public abstract int  IStatickeKonstantyDum.
getIMPL_VYSKA(), public abstract int IStatickeKonstantyDum.getIMPL_SIRKA()]
```

Další parametry slouží jako značky k oddělení špatně deklarovaných nebo chybějících atributů/metod/konstruktorů. Parametr `od` značí řetězec, kterým začíná každý prvek zprávy, naopak parametr `kam` značí řetězec, kterým každý prvek končí. Pro statické konstanty tyto parametry nabývají typicky hodnot `od = ".get"` a `kam = "()"`.

Metoda využívá další dvě privátní metody pro získání jmen špatně deklarovaných prvků z chybové zprávy. Jsou to metody `String deleni(String message, String od, String kam)` a `String preparace(String nazev, String od, String kam)`. Metoda `deleni()` rozděluje zprávu na názvy jednotlivých chyb – tyto chyby ale stále obsahují celou cestu i se jménem balíčku a rozhraní. Pro úplné rozdělení se využívá metoda `preparace()`, která podle zadaných parametrů `od` a `kam` 'vypreparuje' pouze názvy atributů/konstruktorů/metod.

Další funkcí knihovní třídy DuckUtility je zjišťování, zda nebyly použity nějaké pomocné atributy navíc. Toto se zjišťuje porovnáním testované třídy s předem vytvořeným rozhraním, které obsahuje pouze povolené atributy. Rozhraní se nijak neliší od ostatních rozhraních. Obsahuje ale všechny atributy, které chceme označit jako jediné povolené (obvykle všechny). Může se proto zdát, že se jedná o duplicitní testování, ale není tomu tak. Pro příklad třídy `Dum` zobrazeném v příloze A.1 by rozhraní vypadalo jak je znázorněno v kódu 3.2.

Pro zjišťování nadbytečných atributů slouží metoda `zjistiNadbytecne(String jmenoTridy)`, která vrací počet atributů použitých ve třídě navíc. Metoda porovnává atributy použité ve třídě s atributy definovanými v rozhraní. Kvůli identifikaci se rozhraní musí jmenovat přesně podle vzoru: `IExistenceField` + jméno třídy, v tomto případě `IExistenceFieldDum`. Ještě před vrácením počtu nadbytečných atributů se jejich jména zapíší do statické proměnné.

Další metodou pro zjišťování nadbytečných atributů je `getJmenaNadbytecnych()`, která vypíše jména již dříve zjištěných nadbytečných atributů. Volat tuto metodu má smysl až po volání

```
interface IExistenceFieldDum {
    @Field @Declared @Public @StaticField @Final int getIMPL_VYSKA();
    @Field @Declared @Public @StaticField @Final int getIMPL_SIRKA();
    @Field @Declared @Private @StaticField int getpocet();
    @Field @Declared @Private @NonStatic @Final int getcisloDomu();
    @Field @Declared @Private @NonStatic int getvyska();
    @Field @Declared @Private @NonStatic int getsirka();
}
```

Kód 3.2: Rozhraní pro test nadbytečných parametrů

metody `zjistiNadbytecne(String jmenoTridy)`. Pro test nadbytečných atributů lze připravit další test, kterým se doplní testovací třída `TestDuckDum` viz kód 3.3.

```
    @Test
    public void testNadbytecneFieldsDum() {
        int povoleno = 1;
        int pocet = DuckUtility.zjistiNadbytecne("Dum");
        String message = "Použito více (" + pocet + ") pomocných
           proměnných než je  povoleno (" + povoleno + "):" +
           DuckUtility.getJmenaNadbytecnych();
        assertTrue(message, pocet <= povoleno);
    }
```

Kód 3.3: Test nadbytečných atributů s tolerancí 1 atributu

Atribut `povoleno` obsahuje počet atributů, které mohou být ve třídě použity maximálně navíc oproti rozhraní, aby testovaná třída testem prošla. V předchozím testu je tolerován jeden atribut navíc. V případě nulové tolerance se využívá speciální metoda `testNadbytecneFieldsDumPresne()`, která je o něco jednodušší (viz kód 3.4).

```
    @Test
    public void testNadbytecneFieldsDumPresne() {
        int pocet = DuckUtility.zjistiNadbytecne("Dum");
        String message = "Použity nadbytečné (pomocné) proměnné:" +
        DuckUtility.getJmenaNadbytecnych();
        assertTrue(message, pocet == 0);
    }
```

Kód 3.4: Test nadbytečných atributů bez tolerance

3.1.2 Příklad testů pro třídu `Dum`

Pro lepší pochopení přípravy duck-testů v praxi je použit následující příklad. Jedná se o jednoduchou třídu `Dum`, která obsahuje několik atributů, tři konstruktory a pět jednoduchých metod. Ukázku třídy lze najít mezi přílohami jako kód A.1.

Pro testování je třeba vytvořit dva soubory — soubor s testy pojmenovaný podle testované třídy `TestDuckDum` a soubor pro rozhraní `ITestDuckDum`. Toto rozhraní je pouze značkovací, aby bylo zajištěno vhodné jméno zdrojového souboru, a soubor obsahuje několik dalších rozhraní – pro každý test jedno. Protože jsou všechna rozhraní uložena v jednom souboru, nesmí mít přístupové právo `public`. Je velmi vhodné zakončit název každého rozhraní jménem testované třídy (v tomto případě `Dum`). Zejména při testování více tříd je tímto zaručen pořádek mezi testy a příslušnými rozhraními.

Do jednotlivých rozhraní lze sdružovat prvky se stejnými anotacemi — typicky vznikají skupiny statické konstanty, instanční konstanty a podobně. Zdrojový kód rozhraní je přiložen jako příloha A.2. Konstruktory je vhodné testovat zvlášť, tj. pro každý konstruktor vytvořit speciální rozhraní.

Veškeré anotace jsou popsány v kapitole 2, neměl by tedy být problém v takto definovaném rozhraní vyčíst, co se kde testuje. Rozhraní `IStatickeKonstantyDum` testuje statické konstanty `IMPL_VYSKA` a `IMPL_SIRKA`, rozhraní `IStatickePromenneDum` testuje privátní statickou proměnnou `pocet`, rozhrani `IInstancniPromenneDum` testuje privátní instanční proměnné `vyska` a `sirka` a tak dál.

Dalším krokem je vytvoření testů, které rozhraní využívají. Soubor s testy se jmenuje také podle testované třídy, pro tento případ `TestDuckDum`. Nejprve se před spuštěním testů vytvoří instance testované třídy, podle které se bude testovat, v metodě `setUpBeforeClass()`.

Testy mají téměř všechny stejnou konstrukci. Vytvořená instance se pomocí metody `wrap()` z frameworku Duckapter porovná s požadovaným rozhraním. V případě neshody vznikne výjimka `WrappingException`. Při ošetřování této výjimky se vypíše hlášení o chybě pro uživatele pomocí knihovní třídy `DuckUtility`. Celá testovací třída `TestDuckDum` je znázorněna v příloze A.3.

3.1.3 Ruční spouštění testů pod Windows

Pro spouštění připravených testů v `jar` souboru se dříve využívalo příkazové řádky Windows a dávkových souborů. Celý test se spouští pomocí dávkového souboru `test.bat`:

```
@echo off
set cisloDU=02
set nazvy=Dum
chcp 1250 > nul
for %%i in (%nazvy%) do del %%i.class 2> nul
javac -cp ducktest%cisloDU%.jar;. -encoding UTF-8 *.java
for %%i in (%nazvy%) do if not EXIST %%i.class goto konec
for %%i in (%nazvy%) do jar uf ducktest%cisloDU%.jar %%i.class
for %%i in (%nazvy%) do del %%i.class
for %%i in (%nazvy%) do call ducktest.bat %%i
:konec
set cisloDU=
set nazvy=
```

V testu se nastaví číslo úlohy – kvůli jménu `jar` souboru s testy – a názvy všech testovaných tříd. Pro každou položku v názvech, tj. každou testovanou třídu, se vymaže případný přeložený `class` soubor a třída se spolu s `jar` souborem znovu přeloží. Pokud nějaká třída, která má být testovaná, chybí, test skončí. Jinak se přeložené třídy přidají k `jar` souboru a pro každou testovanou třídu se volá dávkový soubor `ducktest.bat`:

```
jar xf ducktest%cisloDU%.jar TestDuck%1.class
if not EXIST TestDuck%1.class goto konec
del TestDuck%1.class
echo test %1
javaw -cp ducktest%cisloDU%.jar;. -jar ducktest%cisloDU%.jar TestDuck%1 > out.txt
type out.txt | find "OK ("
type out.txt | find "AssertionError:"
type out.txt | find "run:"
del out.txt
:konec
```

V tomto souboru se nejprve ověřuje, zda se v archivu nachází testovací třída pro potřebnou třídu. Pokud v něm není, test končí. V opačném případě se třída s testy spustí. Do konzole se vypisuje pouze filtrovaná zpráva o běhu testů.

Spouštění testů tímto způsobem je zbytečně složité a není přenositelné na jiné platformy. Právě

kvůli zjednodušení spouštění pro studenty a přenositelnosti programu bylo uvažováno o vytvoření automatického spouštěče duck-testů.

3.2 Automatizovaná příprava a spouštění testů

3.2.1 Generátor

Hlavní myšlenkou generátoru je programové získání všech atributů, metod a konstruktorů včetně jejich modifikátorů ze vzorové třídy. Vše lze získat pomocí reflexe, ale je k tomu třeba instance dané třídy. Při vytváření testů automaticky jsem narazila na problém jak získat danou instanci. Ne všechny třídy měly konstruktor bez parametrů a svůj implicitní bezparametrický přepisovaly. Metoda, kterou používám a která bude vysvětlena v kapitole 4, dokáže vytvořit instanci třídy i ze soukromého konstruktoru. Z toho důvodu jsem se rozhodla doplnit požadavek na soukromý konstruktor ve vzorových i testovaných třídách.

Další změnou oproti ručnímu generování je zrušení skupin atributů a metod se stejnými vlastnostmi. Při ručním generování to mělo smysl kvůli úspoře práce – nebylo nutné psát tolik rozhraní a testů, když všechny prvky ve skupině měly podobné vlastnosti a tedy hlášení mohlo být pro ně společné. V automatickém generování není problém vygenerovat více testů, pro každý prvek třídy jeden. Tímto se sice nevyužije kompletní knihovní třída `DuckUtility`, ale chybová zpráva může být napsána podrobně, protože se týká pouze jednoho prvku, a není třeba řešit výjimky.

3.2.2 Samostatný spouštěč testů

Pro výuku programování v předmětu KIV/OOP se používá programu BlueJ. Protože studenti veškerou práci s úlohami provádějí pod BlueJ, byl původní záměr vytvořit duck-testy jako externí doplněk do tohoto prostředí a spouštět testy přímo v BlueJ. Druhou uvažovanou možností bylo vytvořit samostatný GUI spouštěč.

Externí doplňky pro BlueJ musí dodržovat předem danou strukturu – `jar` soubor obsahuje hlavní třídu a ta se spouští při spuštění doplňku. Pro každou sadu testů (jednu studentskou úlohu) by se musel instalovat nový doplněk. Oproti tomu samostatný program na spuštění testů se dá přizpůsobit všem požadavkům a není vázán pouze na použití v BlueJ. Pokud bude třeba, lze ho později snadno modifikovat. Samostatný GUI spouštěč je tedy lépe využitelný a modifikovatelný než využití doplňků BlueJ, z toho důvodu jsem zvolila tuto možnost.

Myšlenka spouštěče je založená na původním spouštění pod konzolí Windows. Je nutné mít jar soubor s testy a testované úlohy. `Jar` soubor se rozbalí, přidají se k němu testované soubory a vše se přeloží. Poté se již spouští jednotlivé testy a filtrované chybové hlášení se vypisuje do okna.

4 Generátor testů

Generátor se skládá ze dvou souborů – třídy uživatelského grafického rozhraní pro interakci s uživatelem `GeneratorGUI.java` a knihovní třídy `TestMaker.java`, která obsahuje metodu `main` a metody pro generování.

4.1 Popis generátoru

Ke každé testované třídě je třeba vygenerovat dva soubory – soubor s rozhraním, který obsahuje popis testovaných prvků, a soubor s testy. Jediné, co je k tomu třeba, je seznam všech atributů, metod a konstruktorů včetně jejich modifikátorů, které obsahuje vzorová třída. Vše lze programově získat využitím reflexe ze vzorové úlohy.

Pro získání všech prvků a jejich vlastností jsou třeba `java` soubory vzorové úlohy. Po zvolení složky se vzorovou úlohou se tato úloha překopíruje do dočasně vytvořené složky pro generování. Se soubory se pak pracuje právě v této složce a nikde jinde. Zamezí se tím nechtěnému smazání nebo přepsání souborů. Po skopírování se všechny třídy přeloží do `class`, protože nad nimi funguje reflexe. Teprve pokud překlad proběhne v pořádku, nabídne se uživateli seznam tříd ze složky a on může vybírat, ke kterým se mají vytvořit testy.

4.2 Programové získání prvků třídy

Při zvolení třídy se zjistí její prvky. Třídy z balíku `java.lang.reflect` dokáží z `class` souboru získat všechny informace, které jsou pro generování testů třeba. Třída se předává pomocí jména. To ale platí pouze v případě, že její `class` soubor je ve stejné složce jako spuštěný generátor. Pokud tomu tak není, není třída nalezena. Proto se musí nastavit kořenový adresář, ve kterém se bude hledat požadovaný `class` soubor. Tělo metody, která získává atributy, metody a konstruktory zadané vzorové třídy, je znázorněno v kódu 4.1.

Takto se hledá `class` soubor z daného umístění na disku – zde v adresáři `tmpDir`. Poté se již pomocí metod `getDeclaredFields()`, `getDeclaredConstructors()` a `getDeclaredMethods()` získají atributy, konstruktory a metody třídy se všemi informacemi. Tyto atributy, konstruktory a metody jsou uchovávány v polích. Pokud v zadaném adresáři nebyla nalezena `class` dané třídy, je vyhozena výjimka `ClassNotFoundException`. Ta způsobí výpis varovné zprávy v GUI.

```
    try {
        File root = new File(tmpDir.toString());
        URLClassLoader classLoader;
        classLoader = URLClassLoader.newInstance(new URL[] { root.toURI
            ().toURL() });
        Class<?> cl = Class.forName(jmenoTridy, false, classLoader);
        atributy = cl.getDeclaredFields();
        kontruktory = cl.getDeclaredConstructors();
        metody = cl.getDeclaredMethods();
    } catch (ClassNotFoundException e) {
        gui.setInfo(jmenoTridy + "\nTřída nenalezena!\n");
    }
```

Kód 4.1: Získání instance třídy a informací o všech jejích prvcích

Následně se zjištěné prvky vloží do tabulek. Pro každou třídu se vytváří celkem tři tabulky – jedna s atributy, jedna s metodami a jedna s konstruktory. V daných tabulkách pro každou třídu si uživatel může vybrat/odškrtnout, které prvky třídy chce a které nechce testovat. Po spuštění generace testů se postupuje abecedně podle vybraných tříd. Data se vybírají rovnou z tabulek. Vytvářejí se současně soubor s testy a soubor s rozhraními.

4.2.1 Vytváření testů

Jak vypadají testy bylo zmíněno již v kapitole 3.1.2, kde byla popsána příprava testů pro příklad `Dum`. Při automatické přípravě testy vypadají stejně jako ve zmíněném případě, jen se nevolá knihovní třída `DuckUtility` pro úpravu zprávy. Protože je každý test pouze pro jeden prvek, je zpráva připravena automaticky podle jména a modifikátorů již při vytváření testu.

Navíc je pro testy potřeba dokázat vytvořit instanci testované třídy. Po několika pokusech jsem se rozhodla požadovat existenci bezparametrického konstruktoru, případně i soukromého, v každé testované třídě. Jedná se o administrativní řešení problému, což prakticky znamená, že požadavek na soukromý bezparametrický konstruktor musí být uveden v zadání studentské úlohy. Toto je pouze nouzové řešení a v další verzi spouštěče to bude vyřešeno elegantněji.

Do metody volané před puštěním všech testů jsem doplnila kód 4.2, který dokáže vytvořit instanci i ze soukromého bezparametrického konstruktoru.

```
try {
    File root = new File(".");
    URLClassLoader classLoader;
    classLoader = URLClassLoader.newInstance(new URL[] { root.toURI().
       toURL() });
    Constructor[] konstruktory = Class.forName("Dum", false, classLoader
       ).getDeclaredConstructors();
    Constructor ct = null;
    for (int j = 0; j < konstruktory.length; j++) {
        ct = konstruktory[j];
        if (ct.getGenericParameterTypes().length == 0) {
            break;
        }
        else if (j + 1 == konstruktory.length) {
            throw new Exception("Nenalezen konstruktor bez parametru.");
        }
    }
    ct.setAccessible(true);
    instance = (Dum) ct.newInstance();
} catch (Exception e) {
    System.exit(1);
}
```

Kód 4.2: Získání instance testované třídy

4.2.2 Vytváření rozhraní konstruktorů

Jako první se generují testy a rozhraní pro konstruktory dané třídy. Slouží k tomu metoda `TestMaker.vypisKontruktory()`, která přijímá jméno třídy a pole `Object[][]` dat z tabulky. Jedná se o dvojrozměrné pole obsahující vždy `Constructor` a `boolean` hodnotu, zda má být zařazen do testování nebo ne.

Pro rozhraní je třeba zjistit, zda je konstruktor soukromý pomocí `Modifier.isPrivate(Constructor.getModifiers())`. Jako další je nutné znát parametry konstruktoru – ty se zjistí použitím metody `Constructor.getParameterTypes()`. Metoda vrací pole objektů `Class`, ze kterého lze zjistit typ parametrů.

4.2.3 Vytváření rozhraní atributů

Testy a rozhraní pro atributy se vytvářejí pomocí metody `TestMaker.vypisAtributy()`. Metoda přijímá jméno třídy a pole `Object[][]` dat z tabulky. Tentokrát pole obsahuje `Field` a `boolean` hodnotu, zda má být atribut zařazen do testování nebo ne.

Pro rozhraní se musí zjistit modifikátory daného atributu a jeho typ. Metoda na zjišťování modifikátorů byla už uvedena výše, ale v tomto případě je její použití o něco složitější, protože se zjišťují všechny použité modifikátory. Jak se tímto způsobem tvoří rozhraní je zobrazeno v kódu 4.3.

```
int m = ((Field) data[i][0]).getModifiers();
String pomStr = "  @Field @Declared ";
if (Modifier.isPrivate(m))  {
    pomStr += "@Private ";
} else if (Modifier.isPublic(m)) {
    pomStr += "@Public ";
} else if (Modifier.isProtected(m)) {
    pomStr += "@Protected ";
}
if (Modifier.isStatic(m)) {
    pomStr += "@StaticField ";
} else {
    pomStr += "@NonStatic ";
}
if (Modifier.isFinal(m)) {
    pomStr += "@Final ";
}
```

Kód 4.3: Generování rozhraní pro atributy

V proměnné `pomStr` se postupně tvoří anotace, které danému atributu přísluší. Jeho typ se získá následně pomocí metody `Field.getType().getSimpleName()` a jeho jméno pomocí metody `Field.getName()`.

Pro atributy se navíc tvoří test, zda třída neobsahuje nějaké atributy navíc. Toto má na starost metoda `TestMaker.vypisExistenciAtributu()`, která přijímá stejně jako ostatní metody na generování jméno třídy, pole `Object` obsahující vždy `Field` a `boolean` a navíc počet tolerovaných proměnných navíc. Stejně jako ostatní metody i tato vytváří současně soubory rozhraní i testů.

4.2.4 Vytváření rozhraní pro metody

Testy a rozhraní pro metody vytváří metoda `TestMaker.vypisMetody()`. Tato metoda přijímá jméno třídy a pole `Object[][]` dat z tabulky. Toto pole obsahuje `Method` a `boolean` hodnotu, zda má být atribut zařazen do testování nebo ne.

Modifikátory metod se získají podobně jako u atributů viz kód 4.4. Oproti atributům je třeba navíc získat návratový typ pomocí metody `Method.getReturnType().getSimpleName()` a parametry pomocí `Method.getParameterTypes()`. Jméno metody se pak získá použitím metody `Method.getName()`.

```
int m = ((Method) data[i][0]).getModifiers();
String pomStr = "  @Declared ";
if (Modifier.isPrivate(m))  {
    pomStr += "@Private ";
} else if (Modifier.isPublic(m)) {
    pomStr += "@Public ";
} else if (Modifier.isProtected(m)) {
    pomStr += "@Protected ";
}
if (Modifier.isStatic(m)) {
    pomStr += "@Static ";
} else {
    pomStr += "@NonStatic ";
}
if (Modifier.isFinal(m)) {
    pomStr += "@Final ";
}
```

Kód 4.4: Generování rozhraní pro metody

4.3 Vytvoření jar souboru

Po vygenerování všech souborů s rozhraními a s testy je třeba vytvořit `jar` soubor. Část metody, která to má na starost, je zobrazena v kódu 4.5. Jedná se o část, kde se již pouze vytváří `jar`.

Před tímto je nutné nejdřív přesunout do pracovní složky všechny potřebné soubory. Jedná se o složky `org` a `junit`, které jsou třeba pro správné fungování JUnit testů a duck-testů, soubor `man.txt` a knihovní třídu `DuckUtility.java`. Poté se přesunou vygenerované soubory s testy a rozhraními a vše se přeloží. `Jar` nesmí obsahovat `class` soubor testovaných tříd (jsou připraveny vyučujícím), proto se všechny předem vymažou. Poté se stejně jako v příkazové řádce vytvoří

```
String prikaz = "jar cfm " + tmpDir + FileSystems.getDefault().
    getSeparator() + "duckTest" + jmeno + ".jar " + tmpDir +
    FileSystems.getDefault().getSeparator()
             + "man.txt " + tmpDir + FileSystems.getDefault().
                getSeparator() + "*.class";

if (pridatSoubory) {
     for (String s : tridy) {
         prikaz += " -C " + tmpDir + " TestDuck" + s + " -C " +
            tmpDir + " ITestDuck" + s;
     }
}
prikaz += " -C " + tmpDir + " junit  -C " + tmpDir + " org";
proc = r.exec(prikaz);
proc.waitFor();
from = FileSystems.getDefault().getPath(tmpDir.toString(), "duckTest
    " + jmeno + ".jar");
to = FileSystems.getDefault().getPath(System.getProperty("user.dir")
    , "duckTest" + jmeno + ".jar");
Files.copy(from, to, (CopyOption) StandardCopyOption.
    REPLACE_EXISTING);
```

Kód 4.5: Metoda na vyvoření `jar` souboru

`jar` soubor. Pokud uživatel zvolil, že chce i zdrojové soubory vygenerovaných tříd, ty se přidají. Tato možnost se používá zejména pro ladění.

4.4 Použité jazykové konstrukce

Pro vytvoření generátoru byla použita Java 7, která obsahuje několik nových funkcí, hlavně pro práci se soubory, které by nemusely být známé. Více o nových funkcích Javy 7 viz [7]. V práci je často využívaný nový datový typ `Path`.

Ke kopírování a přesouvání jednotlivých souborů se využívá metod `Files.copy(Path from, Path to, CopyOption copyOption)` a `Files.move(Path from, Path to, CopyOption copyOption)`. Novinkou Javy 7 je možnost pracovat s celým adresářovým stromem pomocí metody `Files.walkFileTree()`, která vyžaduje dva parametry: `Path` cestu k adresáři a instanci třídy `SimpleFileVisitor`, která určuje, jak se adresářovým stromem prochází.

`SimpleFileVisitor` má čtyři metody, které uživatel přepíše podle svých požadavků.

- Metodu `preVisitDirectory()`, která se provádí před navštívením adresáře,
- metodu `visitFile()`, která se volá při navštívení souboru,
- metodu `postVisitDirectory()`, která se volá po navštívení celého adresáře včetně všech jeho podadresářů a
- metodu `visitFileFailed()` prováděnou v případě, že soubor nemůže být navštíven.

Předešlé metody vždy vrací jednu z následujících hodnot:

- `FileVistitResult.CONTINUE` – pokračuje,
- `FileVistitResult.SKIP_SIBLINGS` – pokračuje bez navštívení souborů ve stejném adresáři
- `FileVistitResult.SKIP_SUBTREE` – pokračuje bez navštívení podadresáře a
- `FileVistitResult.TERMINATE` – ukončí procházení adresáře

Všechny metody vrací prvotně `FileVistitResult.CONTINUE` a nic nedělají. Uživatel může některé z nich překrýt, pokud je chce využít pro své účely. Kód 4.6 zobrazuje příklad použité třídy pro kopírování adresáře. Tento `SimpleFileVisitor` je použit v generátoru pro kopírování celé složky se vzorovou úlohou a ke kopírování potřebných souborů pro vytvoření `jar` souboru.

```
class CopyDirVisitor extends SimpleFileVisitor<Path> {
    private Path fromPath;
    private Path toPath;
    private StandardCopyOption copyOption = StandardCopyOption.
       REPLACE_EXISTING;

    public CopyDirVisitor(Path from, Path to) {
        this.fromPath = from;
        this.toPath = to;
    }

    @Override public FileVisitResult preVisitDirectory(Path dir,
       BasicFileAttributes attrs) throws IOException {
        Path targetPath = toPath.resolve(fromPath.relativize(dir));
        if (!Files.exists(targetPath)) {
            Files.createDirectory(targetPath);
        }
        return FileVisitResult.CONTINUE;
    }
```

```
    @Override public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOException {
         Files.copy(file, toPath.resolve(fromPath.relativize(file)),
            copyOption);
         return FileVisitResult.CONTINUE;
    }
}
```

Kód 4.6: Třída pro kopírování adresářového stromu

4.4.1 Procesy

Pro spouštění nástrojů `javac` pro překlad a `jar` pro vytváření archivu jsou využity samostatné procesy. Proces se spustí stejným příkazem jako by se spouštěl z příkazové řádky. Poté je třeba odebrat případný výstup procesu a počkat na jeho dokončení (pokud to program vyžaduje).

```
    String line, vystup = "";
    Process proc = r.exec("javac -cp " + tmpDir.toString() + ";. -
        encoding UTF-8 " + tmpDir.toString() + FileSystems.getDefault().
        getSeparator() + "*.java");
    BufferedReader error = new BufferedReader(new InputStreamReader(proc
        .getErrorStream()));
    while ((line = error.readLine()) != null) {
         vystup += line + "\n";
    }
    proc.waitFor();
    if (proc.exitValue() != 0) {
         System.out.println("Chyba při překladu!");
    }
```

Kód 4.7: Příklad použití procesů v Javě

V kódu 4.7 je zobrazen překlad všech `java` souborů ve složce `tmpDir` s případným chybovým výpisem do konzole. V příkladu se čeká na dokončení procesu. Kódování `UTF-8` je standardně používáno ve všech úlohách. Proto je v kódu uvedeno jako literál.

4.4.2 Inicializační soubor

Kvůli vyššímu komfortu pro uživatele je využíván inicializační soubor. Je použito standardní schéma `.ini` souboru, tj. klíčové slovo následované znakem „=“ a svou hodnotou. Inicializační soubor pro generátor má několik klíčových slov:

- `UlohaPath` – cestu ke vzorové úloze,
- `X` – x-ovou souřadnici okna uživatelského rozhraní,
- `Y` – y-ovou souřadnici okna uživatelského rozhraní,
- `Tolerance` – počet tolerovaných atributů navíc a
- `JSoubory` – zaškrtnutí, resp. nezaškrtnutí, možnosti přidání vygenerování zdrojových souborů.

Soubor se zpracovává jako tzv. `Properties` – hodnoty se načítají právě pomocí klíčových slov. Způsob načítání naposledy použitého nastavení ze souboru zobrazuje kód 4.8.

```
Properties p = new Properties();
p.load(new FileInputStream(new File("duckGenerator.ini")));
x = Integer.valueOf(p.getProperty("X"));
y = Integer.valueOf(p.getProperty("Y"));
ulohaPath = URLDecoder.decode(p.getProperty("UlohaPath"), Charset.
    defaultCharset().toString());
tolerance = Integer.valueOf(p.getProperty("Tolerance"));
java = (p.getProperty("JSoubory").contains("true")) ? true : false;
```

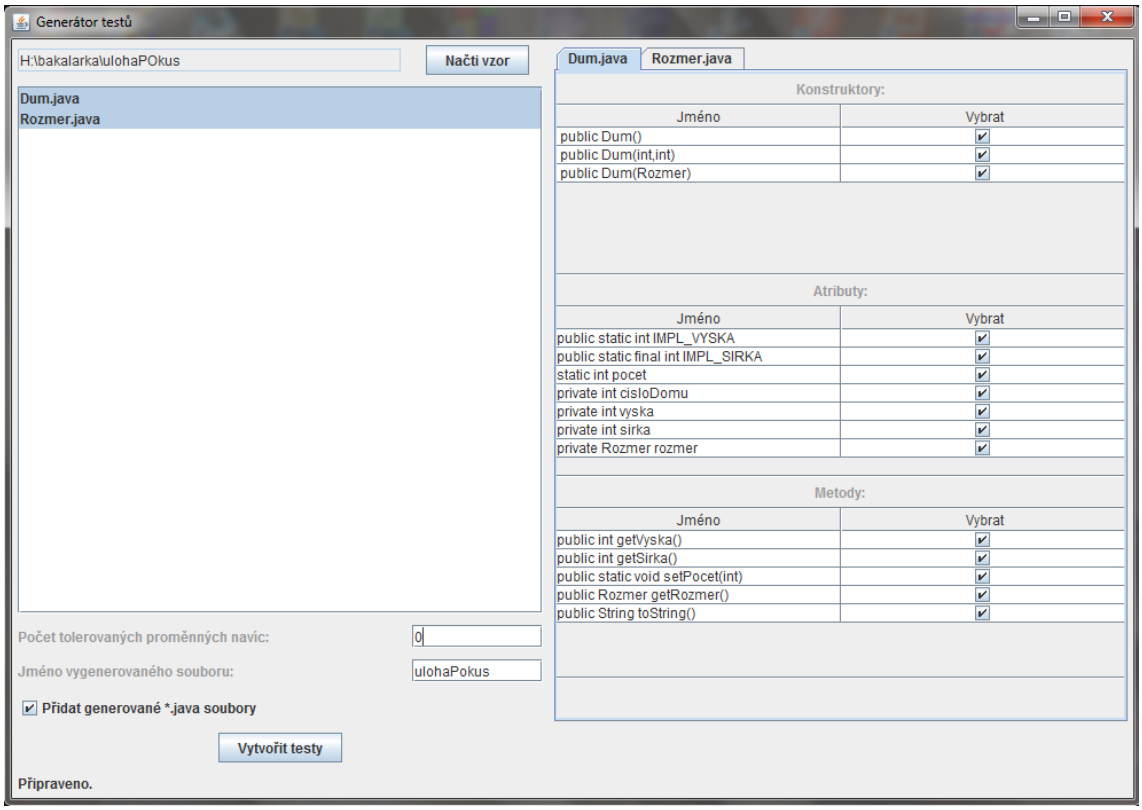
Kód 4.8: Načtení hodnot z `.ini` souboru

4.5 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní (dále GUI) generátoru je vytvořeno v javě pomocí Swingu [8]. GUI je zobrazeno na obrázku 4.1. Je rozděleno na dvě části – vlevo nastavení a spouštění generace, vpravo panel s vybranými vzorovými třídami a tabulkami jejich atributů, metod a konstruktorů.

Klasicky je vlevo nahoře tlačítko pro načtení složky se vzorovou úlohou. Využívá se `JFileChooser`. Cesta ke zvolené složce se zobrazí do přidruženého `JTextFieldu`. Pod těmito prvky se po načtení zobrazí nabídka všech `java` souborů ve zvolené složce. Pro zobrazení seznamu je použit `JScrollPane` a `ListModel`. Právě při výběru třídy z tohoto seznamu se v pravé části zobrazí panel se třídou a tabulkami se seznamem prvků třídy.

Pod seznamem tříd se nachází několik voleb nastavení. `JTextField` pro nastavení počtu tolerovaných proměnných v testu nadbytečných atributů. `JTextField` pro nastavení jména výstupního `jar` souboru, který se automaticky nastavuje na jméno složky. Poslední volbou je `JCheckBox` pro přidání vygenerovaných `java` souborů do výstupního `jar` souboru. Úplně dole v levé části GUI se nachází `JButton` pro spuštění generace testů. Pod ním je informační `JLabel`, která uživatele



Obrázek 4.1: Obrázek GUI generátoru

informuje o aktuálním průběhu generace.

Pravá část okna je dynamická. Jedná se o `JTabbedPane`. Podle počtu aktuálně vybraných tříd v levém okně se přidávají nové záložky. Každá záložka obsahuje tři tabulky – tabulku s atributy, konstruktory a metodami dané třídy. Uživatel kliknutím na záložku si může zobrazit tabulky dané třídy a nastavit, pro které prvky třídy budou generovány testy a pro které ne. Standardně jsou zvoleny všechny.

5 Spouštěč testů

Spouštěč se skládá celkem ze tří souborů – třídy uživatelského grafického rozhraní pro interakci s uživatelem `SpoustecGUI.java`, třídy `Okenko.java` na zobrazení čekacího dialogu a knihovní třídy `SpoustecDuckTestu.java` obstarávající funkčnost.

5.1 Popis spouštěče

Pro práci spouštěče testů jsou třeba dvě věci – `jar` soubor s testy, nejlépe vytvořený generátorem testů nebo ručně podle stejného vzoru, a složka se všemi testovanými třídami studentské úlohy. Před spuštěním testů je tedy nutné nastavit cestu k `jar` souboru a do složky s testovanou úlohou. Testy se nespustí, dokud nejsou obě podmínky splněné.

Při prvním spuštění testů je nejprve nutné rozbalit `jar`, protože se bude doplňovat o testované třídy, což provede spouštěč automaticky. Podobně jako u generátoru se využívá i pro testování pomocná dočasná složka, do které se rozbalí `jar` soubor s testy a kopírují se tam testované třídy.

5.1.1 Programové rozbalení jar souboru

Při prvním spuštění a nebo při vybrání nového souboru s testy se při spuštění nejprve vytvoří nová dočasná složka a do ní se rozbalí `jar` soubor. Zamezí se tím přepsání případných souborů, které by mohly být v použité složce. Kód 5.1 zobrazuje postup rozbalení `jar` souboru. Tento postup je převzat z [9]. Pro získání obsahu archivu je použita metoda `JarFile.entries()`.

5.1.2 Spuštění testů

Po rozbalení `jar` souboru při prvním spuštění a nebo ihned při druhém spuštění se postupuje podle následujícího schématu:

- metoda `SpoustecDuckTestu.getJmenaTestovanych()` zjistí z `jar` souboru podle jmen duck-testů jména všech testovaných tříd a uloží je do seznamu,
- kontroluje se, zda seznam testovaných tříd není prázdný, pokud ano, test se ukončí,

```
Enumeration<JarEntry> en = jar.entries();
while (en.hasMoreElements()) {
    JarEntry jarFile = (JarEntry) en.nextElement();
    final String name = jarFile.getName();
    File file = new File(tmpDir + FileSystems.getDefault().
       getSeparator() + name);
    if (jarFile.isDirectory())  {
        file.mkdir();
    }
    else {
        InputStream is = jar.getInputStream(jarFile);
        FileOutputStream fos = new java.io.FileOutputStream(file);
        while (is.available() > 0) {
            fos.write(is.read());
        }
        fos.close();
        is.close();
    }
}
```

Kód 5.1: Programové rozbalení `jar` souboru

- metoda `SpoustecDuckTestu.isTestovaneTridy()` kontroluje, zda se všechny zdrojové `java` soubory testovaných tříd v seznamu nachází ve složce s úlohou, pokud ne, test se ukončí,
- skopírují se všechny třídy ze složky s úlohou do dočasné složky a
- pro každou třídu se spustí test.

Test se spouští pro každou třídu podobně jako při ručním spouštění z příkazové řádky. Vytvoří se proces, který spustí pomocí hlavní třídy pro spouštění JUnit testů `org.junit.runner.JUnitCore` soubor s vytvořenými testy, tj. třídu začínající `TestDuck` a končící jménem testované třídy. Výpis tohoto procesu se filtruje a uživateli se zobrazuje pouze případné chybové hlášení nebo informace, kolik testů proběhlo.

5.2 Použité jazykové konstrukce

U spouštěče jsou využity podobné jazykové konstrukce jako u generátoru. Využívá se nových funkcí Javy 7 pro práci se soubory, hlavně kopírování a přesouvání. Stejně tak se používají procesy pro spouštění nástroje `javac` pro překlad.

A také jako v případě generátoru je použit `SimpleFileVisitor` pro práci s adresářem. U spouštěče je využíván pouze pro mazání dočasné složky. Po ukončení programu a nebo při otevření nového souboru s testy, se musí dočasná složka smazat. To provádí kód 5.2.

```
Files.walkFileTree(tmpDir, new SimpleFileVisitor<Path>() {
    @Override public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOException {
         Files.delete(file);
         return CONTINUE;
    }

    @Override public FileVisitResult postVisitDirectory(Path dir,
        IOException exc) throws IOException {
         if (exc == null) {
             Files.delete(dir);
             return CONTINUE;
         } else {
             throw exc;
         }
    }
});
```

Kód 5.2: Online použití `SimpleFileVisitor`u pro mazání celého adresáře

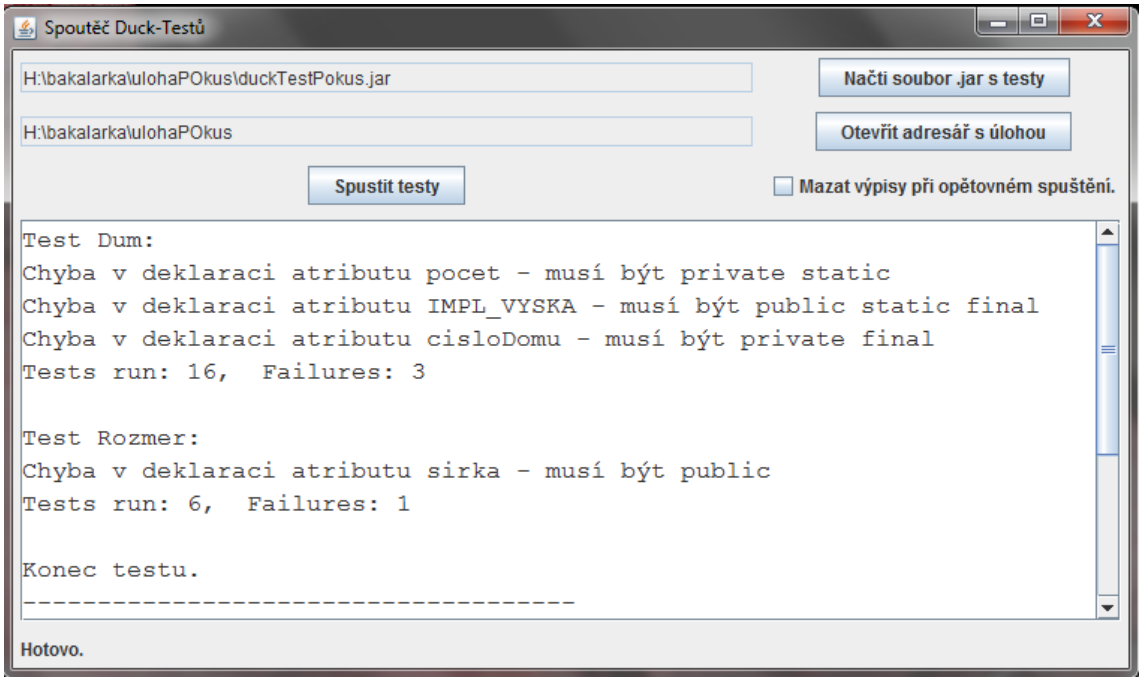
Stejně jako u generátoru, i spouštěč využívá inicializační soubor pro zapamatování pozice okna a cesty k naposledy otevřené složce s úlohou a k `jar` souboru s testy.

5.3 Uživatelské rozhraní

Uživatelské rozhraní spouštěče je vytvořeno pomocí balíčku `javax.swing` [10]. Jak je vidět na obrázku 5.1, je GUI jednodušší než u generátoru.

V horní části okna se nachází dvě tlačítka – jedno pro načtení souboru s testy, druhé pro načtení složky s úlohou. Pro načtení je použit `JFileChooser`. Celá cesta k načtenému `jar` souboru a složce se zobrazí vlevo v příslušném `JTextFieldu`. Jediná volba, která se nachází v okně, je zaškrtávací možnost přepisování zpráv v případě vícenásobného spuštění. Vedle něj je už jen tlačítko pro spuštění testů. Dolní část okna a jeho valnou většinu zabírá textové pole, kam se vypisují hlášení o průběhu testů a případná chybová hlášení.

V případě prvního spuštění testů, kdy dochází k rozbalení `jar` souboru s testy, se uprostřed okna zobrazí čekací dialog, který zobrazuje průběh rozbalování. Tento dialog nelze zavřít, uzavře se sám po rozbalení `jar` souboru.



Obrázek 5.1: Obrázek GUI spouštěče

6 Testy a zhodnocení

6.1 Popis testovaných úloh

Pro přípravu vzorových duck-testů bylo použito osm, resp. sedm, úloh předmětu KIV/OOP, které studenty učí základům objektového programování. První úloha studenty seznamuje s prostředím BlueJ a způsobem testování. Na této úloze není nic možné testovat. V sedmi zbývajících úlohách se testovaly třídy, které měli studenti za úkol vytvořit. Kolik tříd se testuje v jednotlivých úlohách a jejich jména jsou znázorněny v tabulce 6.1. Studenti vytvářejí ještě třídu `Pohlavi`, ale jedná se o jednoduchý `enum`, na kterém není co testovat.

úloha	jména testovaných tříd
2	`Osoba`
3	`Osoba`
4	`Osoba, Rozmer`
5	`Osoba, Rozmer, Zvyraznovac`
6	`Osoba, Par, Rande, Rozmer, Zvyraznovac`
7	`Osoba, Par, Rande, Rozmer, Superman, Zvyraznovac`
8	`Osoba, Par, Rande, Rozmer, Seznamka, Superman, Superwoman, Zvyraznovac`

Tabulka 6.1: Testované třídy pro každou úlohu

6.2 Postup přípravy duck-testů

Nejprve jsem připravila duck-testy ručně pro výše zmíněné sady úloh. Měla jsem k dispozici správné vzorové řešení, podle nějž jsem vytvářela duck-testy. Poznatky získané během ruční přípravy duck-testů jsem využila pro naprogramování automatického generátoru. Duck-testy pro stejnou sadu úloh jsem vytvářela poté i automaticky za použitím generátoru. Samozřejmě podle stejného vzorového řešení.

6.2.1 Ruční příprava duck-testů

Při ruční přípravě se musí všechny rozhraní a testy vytvořit manuálně. Stejně jako u automatické generace, je třeba vědět, jak má třída vypadat. Testy jsou si navzájem dost podobé. Rozhraní se tvoří právě podle vzorové třídy. Ruční příprava testů byla velmi časově náročná.

Také z důvodu monotónosti práce, je autor testů náchylnější k chybám, které se špatně odhalují.

Kvůli zminimalizování počtu testů a rozhraní, které je třeba psát, jsem prvky třídy při ruční přípravě rozdělila do skupin se stejnými modifikátory – například skupina statické konstanty zahrnovala atributy s modifikátory `private static final`. Skupina má vždy stejné modifikátory a stejné chybové hlášení.

Je zřejmé, že ne vždy všechny prvky třídy náleží do určité skupiny – například pro předchozí případ atribut s modifikátory `public static final` tvoří výjimku a nemůže být zařazen do skupiny statických konstant. Výjimky se musí řešit zvlášť. Většinou se pro ně vytvoří samostatný test a rozhraní. Počty testů (a zároveň počty rozhraní) pro jednotlivé třídy a úlohy znázorňuje tabulka 6.2. Celkový počet ručně připravených testů pro celou sadu úloh je 243. Jednotlivé třídy se v úlohách postupně rozšiřují. To znamená, že nebylo možné mechanicky převzít testy připravené pro dřívější úlohu.

číslo úlohy	2	3	4	5	6	7	8
počet testů pro třídu `Osoba`	12	13	20	20	20	22	22
počet testů pro třídu `Rozmer`	-	-	5	5	5	5	5
počet testů pro třídu `Zvyraznovac`	-	-	-	6	6	6	6
počet testů pro třídu `Par`	-	-	-	-	5	4	6
počet testů pro třídu `Rande`	-	-	-	-	7	7	7
počet testů pro třídu `Superman`	-	-	-	-	-	8	8
počet testů pro třídu `Seznamka`	-	-	-	-	-	-	7
počet testů pro třídu `Superwoman`	-	-	-	-	-	-	6

Tabulka 6.2: Počet testů pro jednotlivé třídy (ruční přírava)

Ruční příprava testů zabrala přibližně dva měsíce práce. Připravené testy byly několikrát upravovány a využívány pro rutinní testování úloh asi 100 studentů v zimním semestru 2012/13 v předmětu KIV/OOP. Testy byly studentům k dispozici jako samostatně spustitelný konzolový skript (jeho spouštění je podrobně pospáno v kapitole 3.1.3). Dále byly tytéž testy součástí validační domény na validačním serveru, kde byly spouštěny automaticky.

Aby bylo možné zjistit, jakých prohřešků se studenti nejčastěji dopouštějí (a poté na ně případně zaměřit větší pozornost při přípravě testů), byly ručně připravované testy použity i pro vyhodnocení předchozího ročníku (tj. 2011/12) studentských úloh. Tyto úlohy nebyly v průběhu výuky testovány, tzn. obsahovaly neodhalené prohřešky proti zadání. Výsledky z tohoto testování jsou zpracovány v části 6.3.

Zkušenosti získané jak z ručních příprav testů, tak i ze samotného testování byly podrobně

vyhodnocovány a staly se jedním z důležitých podkladů pro implementaci automatizovaného generátoru testů.

6.2.2 Automatické generování

Při automatickém generování jsem oproti ručnímu upustila od vytváření skupin prvků třídy podobných vlastností. Není problém vygenerovat automaticky více testů a rozhraní, pro každý prvek třídy zvlášť. Chybové hlášení může být pak pro každý prvek třídy vytvořeno „na tělo“. Navíc, když neexistují výše zmíněné skupiny, neexistují ani výjimky, které je potřeba ošetřovat zvláštními testy.

Tabulka 6.3 zobrazuje počty vytvořených testů pro jednotlivé třídy všech úloh při automatickém generování. Při porovnání s ručně generovanými testy je vidět poměrně velký rozdíl v počtu generovaných testů. Z toho vyplývá i větší počet rozhraní a tím pádem více přeložených `class` souborů u automatické generace. Celkový počet automaticky generovaných testů pro celou sadu úloh je 445.

číslo úlohy	2	3	4	5	6	7	8
počet testů pro třídu `Osoba`	25	23	34	36	36	36	36
počet testů pro třídu `Rozmer`	-	-	8	8	8	8	8
počet testů pro třídu `Zvyraznovac`	-	-	-	9	9	9	9
počet testů pro třídu `Par`	-	-	-	-	9	11	14
počet testů pro třídu `Rande`	-	-	-	-	15	18	18
počet testů pro třídu `Superman`	-	-	-	-	-	10	10
počet testů pro třídu `Seznamka`	-	-	-	-	-	-	22
počet testů pro třídu `Superwoman`	-	-	-	-	-	-	16

Tabulka 6.3: Počet testů pro jednotlivé třídy (automatické generování)

6.3 Výsledky testování

K otestování funkčnosti ručně vytvořených duck-testů jsem zvolila náhodně dvaceti procentní vzorek z úloh odevzdaných studenty předmětu KIV/OOP v roce 2011/12. U těchto náhodně vybraných studentů jsem testovala pomocí vytvořených duck-testů všech sedm úloh.

Zjistila jsem, že studenti nejčastěji chybují ohledně přístupových práv – místo `private` používají `public` nebo přístupové právo úplně vynechají a nechají atribut přístupný pro celý balíček. Další

objevující se chybou je vynechání modifikátoru `final` u atributů, které by měly být konstantní. Poměrně často se také chybuje ohledně modifikátoru `static`, ať už jde o jeho vynechání nebo naopak jeho vložení na místo, kam nepatří.

typ chyby	počet výskytů
vynechání přístupových práv	182
použití `public` místo `private`	168
vynechání modifikátoru `final`	149
vynechání modifikátoru `static`	15
použití modifikátoru `static` u atributů, které měly být instanční	10
použité pomocné proměnné	8
jinak pojmenované proměnné (malá/velká písmena, překlepy)	7
použití `protected` na místě, kde mělo být `private`	6
použití `public` místo `protected`	5
chybějící atribut	2
použití jiného než předepsaného datového typu	2

Tabulka 6.4: Nejčastější chyby ve studentských úlohách

Nejčastěji vyskytující se druhy chyb a jejich počty v kontrolovaném vzorku zobrazuje tabulka 6.4. Chyby jsou seřazené sestupně podle počtu výskytů v testovaném vzorku. Je vidět, že několik chyb se opakuje poměrně často, zbytek je oproti nim zanedbatelný.

Podle těchto výsledků usuzuji, že použití duck-testů při výuce programování má smysl. Studenti jinak poměrně často nerespektují implementační pravidla, která jim vyučující předepsal v zadání.

Duck-testy by měly zajistit, aby takto nevyhovující úlohy, které sice správně fungují, ale jsou naimplementovány špatně, byly odhaleny a nepřijaty jako správné. To by mělo studenty přinutit úlohu naprogramovat správně a skutečně použít jazykovou konstrukci, kterou mají předepsanou v zadání a mají se s ní naučit pracovat.

Automaticky generované testy zajišťují, že všechny výše uvedené „prohřešky“ jsou pokryty testovacími případy. Porovnáním nejčastějších chyb a existujících způsobů jejich odhalení jsem získala jistotu, že žádný z často se vyskytujících problémů nezůstane neodhalen.

7 Závěr

Celá práce měla dlouhodobý charakter – zahrnovala období dvou akademických let, resp. studentské úlohy připravené v těchto obdobích. Část práce byla řešena v rámci rozvojového projektu MŠMT *Racionalizace a objektivizace výuky* řešeného na KIV. Díky soustavné práci na dané problematice a postupnému vylepšování testů je v předkládané práci dosaženo stavu, kdy je možné její výsledky přímo nasadit do používání ve výuce. Podařilo se připravit ucelený systém, který bude dále rutinně používán ve výuce základů programování. Části bakalářské práce již byly ověřeny v praktické výuce.

V první části práce byl prozkoumán framework Duckapter a na základě jeho možností byla ručně připravena sada 243 testů pro sedm různých studentských úloh. Tato sada testů byla nasazena v praxi při výuce a na základě získaných informací postupně upravována a rozšiřována.

Zkušenosti z ruční přípravy testů a získané analýzou nejčastějších studentských „prohřešků“ se staly základem pro přípravu aplikace, která umožňuje generovat testy automaticky. Tato aplikace byla naprogramována v programovacím jazyce Java. Výsledky automaticky generovaných testů byly důkladně porovnány s výsledky testování pomocí ručně připravených testů. Použití aplikace významně zjednodušuje proces přípravy testů a zpřesňuje jej. Není tak možné, aby nějaký prvek zůstal díky přehlédnutí neotestován. Negativem tohoto přístupu je, že vygenerovaných testů (tj. i přeložených `.class` souborů) je o poznání větší množství (445 testů). To však nezpůsobuje žádné zvláštní potíže, protože jednotlivé `.class` soubory jsou sbaleny do jednoho `.jar` souboru, který je studentům poskytován.

Třetí část bakalářské práce spočívala ve vytvoření desktopové aplikace, která nad studentskou úlohou spouští a vyhodnocuje připravené testy. Dříve se testy spouštěly pomocí skriptu, což činilo problémy s přenositelností a vyhodnocováním výsledků. Připravená aplikace (opět napsaná v Javě) maximálně zjednodušuje proces spouštění a činí jej přenositelným. Tato aplikace nebyla jako jediná zatím testována na větším vzorku studentů. Plošné nasazení se předpokládá v zimním semestru 2013/14.

Možná další rozšíření práce jsou bezpochyby lokalizace aplikací do jiných jazyků, zejména do angličtiny, protože zadavatel hodlá proces testování publikovat. Další vylepšení, které by zvýšilo přehlednost generovaných souborů, by bylo oddělení zdrojových souborů vygenerovaných testů od testovaných úloh, tj. testy by byly generovány do jiného balíku (adresáře). Výzvou pro další, již netriviální, rozšíření, je možnost, kdy při spouštění testů bude v případě nutnosti automaticky vygenerován implicitní konstruktor testované třídy. Přípravu tohoto konstruktoru je zatím nutné zajistit administrativně v zadání studentské úlohy.

Literatura

[1] Vladimír ORANÝ. *Automatické vyhodnocování studentských úloh.* Diplomová práce, VŠE, Fakulta informatiky a statistiky, Praha, 2010.

[2] Rudolf PECINOVSKÝ. *Návrhové vzory.* Computer Press, a.s., Brno, 2007.

[3] Gof patterns. [online], [cit. 04/2013]. Dostupné z: http://www.gofpatterns.com/.

[4] Rudolf PECINOVSKÝ. *Java 5.0: Novinky jazyka a upgrade aplikací.* CP Books, a.s., Brno, 2005.

[5] Pavel HEROUT. *Přednášky KIV/OOP.* Západočeská univerzita, Plzeň, 2013.

[6] Paul HAMILL. *Unit Test Framework: Tools for High-Quality Software Development.* O'Reilly Media, 2004.

[7] What's new in java 7: Copy and move files and directories. [online], [cit. 04/2013]. Dostupné z: http://codingjunkie.net/java-7-copy-move/.

[8] Pavel HEROUT. *Java – grafické uživatelské prostředí a čeština.* nakladatelství KOPP, České Budějovice, 2007.

[9] Extract the contents of zip/jar files programmatically. [online], 1[cit. 01/2013]. Dostupné z: http://www.devx.com/tips/Tip/22124.

[10] Creating a gui with jfc/swing. [online], [cit. 01/2013]. Dostupné z: http://docs.oracle.com/javase/tutorial/uiswing/.

A Přílohy

A.1 Zdrojový kód příkladu `Dum`

```
public class Dum {
    public static final int IMPL_VYSKA = 100;
    public static final int IMPL_SIRKA = 30;

    private static int pocet = 0;

    private final int cisloDomu;

    private int vyska;
    private int sirka;
    private Rozmer rozmer;

    public Dum() {
        this(IMPL_VYSKA, IMPL_SIRKA);
    }

    public Dum(Rozmer r) {
        this(r.vyska, r.sirka);
        rozmer = r;
    }

    public Dum(int pVyska, int pSirka) {
        this.vyska = pVyska;
        this.sirka = pSirka;
        cisloDomu = ++pocet;
        rozmer = new Rozmer(pVyska, pSirka);
    }

    public static void setPocet(int novyPocet) {
        pocet = novyPocet;
    }

    public int getVyska() {
        return vyska;
    }

    public int getSirka() {
        return sirka;
    }

    public Rozmer getRozmer() {
        return rozmer;
```

```
    }

    @Override
    public String toString() {
        return "Dum č." + cisloDomu;
    }
}
```

Kód A.1: Třída `Dum`

```
import org.duckapter.annotation.*;

public interface ITestDuckDum {
}

interface IStatickeKonstantyDum {
    @Field @Declared @Public @StaticField @Final int getIMPL_VYSKA();
    @Field @Declared @Public @StaticField @Final int getIMPL_SIRKA();
}

interface IStatickePromenneDum {
    @Field @Declared @Private @StaticField int getpocet();
}

interface IInstancniKonstantyDum {
    @Field @Declared @Private @NonStatic @Final int getcisloDomu();
}

interface IInstancniPromenneDum {
    @Field @Declared @Private @NonStatic int getvyska();
    @Field @Declared @Private @NonStatic int getsirka();
}

interface IKonstruktorBezParamDum {
    @Constructor Dum newInstance();
}

interface IKonstruktorVyskaSirkaDum {
    @Constructor Dum newInstance(int vyska, int sirka);
}

interface IGetryDum {
    @Declared @Public @NonStatic int getVyska();
    @Declared @Public @NonStatic int getSirka();
}

interface ISetPocetDum {
    @Declared @Public @Static void setPocet(int novyPocet);
}

interface IToStringDum {
    @Declared @Public @NonStatic String toString();
}
```

Kód A.2: Zdrojový kód rozhraní pro třídu `Dum`

```
import static org.junit.Assert.assertNull;
import org.duckapter.*;
import org.junit.*;

public class TestDuckDum {

    private static Dum dum;

    @BeforeClass
    public static void setUpBeforeClass() {
        dum = new Dum();
    }

    @Test
    public void testStatickeKonstanty() {
        try {
            Duck.wrap(dum, IStatickeKonstantyDum.class);
        } catch (WrappingException e) {
            String message = "" + e.getAdapted().getClassWrapper().
                getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('X', message,
                ".get", "()")  + " - musí být public static final." ;
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testStatickePromenne() {
        try {
            Duck.wrap(dum, IStatickePromenneDum.class);
        } catch (WrappingException e) {
            String message = "" +                e.getAdapted().
                getClassWrapper().getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('P', message,
                ".get", "()");
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testInstancniKonstanty() {
        try {
            Duck.wrap(dum, IInstancniKonstantyDum.class);
        } catch (WrappingException e) {
            String message = "" +   e.getAdapted().getClassWrapper().
                getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('k', message,
                ".get", "()");
            assertNull(chybovaZprava, e.getMessage());
```

```
        }
    }

    @Test
    public void testInstancniPromenne() {
        try {
            Duck.wrap(dum, IInstancniPromenneDum.class);
        } catch (WrappingException e) {
            String message = "" +   e.getAdapted().getClassWrapper().
                getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('p', message,
                ".get", "()");
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testKonstruktorBezParam() {
        try {
            Duck.wrap(dum, IKonstruktorBezParamDum.class);
        } catch (WrappingException e) {
            String chybovaZprava = "Špatně vytvořen kontruktor bez
                parametrů Dum();";
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testKonstruktorVyskaSirka() {
        try {
            Duck.wrap(dum, IKonstruktorVyskaSirkaDum.class);
        } catch (WrappingException e) {
            String chybovaZprava = "Špatně vytvořen kontruktor Dum(int
                vyska, int   sirka);";
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testGetry() {
        try {
            Duck.wrap(dum, IGetryDum.class);
        } catch (WrappingException e) {
            String message = "" +   e.getAdapted().getClassWrapper().
                getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('G', message,
                "Dum.", "()");
            assertNull(chybovaZprava, e.getMessage());
        }
```

```
    }

    @Test
    public void testPocet() {
        try {
            Duck.wrap(dum, ISetPocetDum.class);
        } catch (WrappingException e) {
            String message = "" +   e.getAdapted().getClassWrapper().
               getUnimplementedForInstance();
            String chybovaZprava = DuckUtility.upravZpravu('X', message,
                "Dum.", "(") +     "- musí být public static";
            assertNull(chybovaZprava, e.getMessage());
        }
    }

    @Test
    public void testToString() {
        try {
            Duck.wrap(dum, IToStringDum.class);
        } catch (WrappingException e) {
            String chybovaZprava = "Chybně vytvořena metoda toString()";
            assertNull(chybovaZprava, e.getMessage());
        }
    }
}
```

Kód A.3: Zdrojový kód duck-testů pro třídu `Dum`

B Uživatelská příručka pro generátor testů

B.1 Účel programu

Generátor duck-testů je program umožňující vytvářet testy implementace podle vzorového zadání. Oproti jiným testům, tyto testy neověřují funkčnost studentských programů, ale to, zda jsou napsány přesně podle zadání. Pro spuštění programu je nutné mít nainstalovanou Javu 7 nebo vyšší.

B.2 Popis instalace

B.2.1 Instalace pro OS Windows

Před spuštěním programu je třeba rozbalit archiv `duckTestsGenerator.zip`. Klikněte na něj pravým tlačítkem myši, zvolte `Extrahovat...` a vyberte cílovou složku, kam bude program umístěn. Nejlépe ho umístěte do samostatné složky, např. `C:\Program Files\duckTestGenerator`.

Po rozbalení se ve vybrané složce objeví složka `testjar` a soubor `duckTestsGenerator.jar`. Tento `jar` soubor slouží ke spuštění programu. Složku `testjar` doporučuji nemazat ani nijak upravovat, vygenerované duck-testy po úpravě jejího obsahu by nefungovaly.

B.2.2 Instalace pro OS Linux

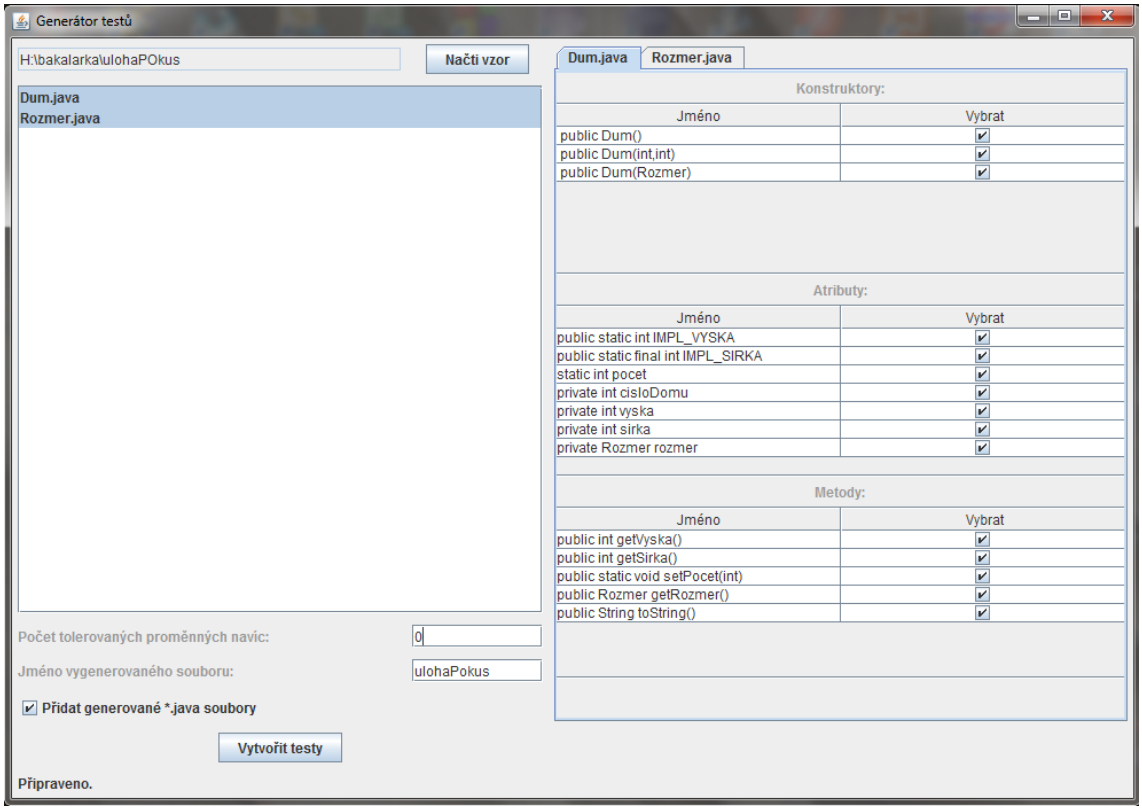
Pro rozbalení souboru `duckTestsGenerator.zip` je třeba nainstalovat balíček `unzip`, pokud již nebyl nainstalován. To se provede příkazem `apt-get install unzip` v distribucích na principech `Debianu`, nebo příkazem `yum install unzip` v případě distribucí `Red Hat` nebo `Fedora`.

Samotné rozbalení se pak provede příkazem `unzip duckTestsGenerator.zip -d /dir`, kde `/dir` představuje adresář, kam se program rozbalí. Program se spustí příkazem `/usr/lib/jvm/java -jar duckTestsGenerator.jar`. Je nutné mít nainstalovanou Javu 7 nebo vyšší v adresáři `/usr/lib/jvm`.

B.3 Práce s programem

B.3.1 Popis uživatelského rozhraní

Uživatelské rozhraní generátoru testů zobrazuje obrázek B.1. Okno je rozděleno na dvě části – vlevo nastavení a spouštění generace, vpravo panel s vybranými vzorovými třídami a tabulkami jejich atributů, metod a konstruktorů. V dolní části se nachází informační lišta, která uživatele informuje o průběhu během generace.



Obrázek B.1: GUI generátoru

Úplně nahoře v levé části okna se nachází tlačítko `Načti vzor`, které slouží k načtení složky se vzorovou úlohou. Cesta ke zvolené složce se zobrazí vedle tlačítka. Po načtení tříd ze složky se jejich jména zobrazí v seznamu, který je těsně pod tlačítkem.

Pod seznamem tříd se nachází několik voleb nastavení. V prvním textovém poli se nastavuje počet tolerovaných proměnných pro test nadbytečných atributů. Druhé pole je určeno pro změnu názvu vygenerovaného souboru s testy. Toto textové pole se vyplňuje po načtení vzorové složky jejím jménem. Poslední volbou je zaškrtávací možnost `Přidat generované .java soubory`,

která při zaškrtnutí přidá do `jar` souboru vygenerované soubory s testy a rozhraními. Pod všemi nastaveními se nachází tlačítko `Vytvořit testy`, které spustí generaci.

Pravá část se mění podle vybraných tříd. Pro každou vybranou třídu se v pravé části okna zobrazí panel obsahující tabulky s jednotlivými prvky třídy. V tabulkách lze zvolit, které prvky se mají zařadit do testů a které ne. V dolní části se nachází informační panel, kde se zobrazují informace o průběhu generování.

B.3.2 Postup při generování testů

Začněte kliknutím na tlačítko `Načti vzor`. Vyberte složku, ve které se nachází vzorová úloha a potvrďte stisknutím `Open`. V seznamu pod tlačítkem by se měly objevit názvy všech `java` souborů, pokud se tak nestalo, pravděpodobně došlo k chybě při jejich načítání a informace o chybě se zobrazí v dolní části okna v informační liště.

V případě bezproblémového průběhu můžete ze seznamu vybírat třídy, pro které chcete vytvořit duck-testy. Vícenásobný výběr se provádí pomocí tlačítka `Ctrl`. Zvolené třídy se zobrazí v pravé části okna – jejich atributy, metody a konstruktory jsou vypsány v tabulkách. Standardně jsou zvolené všechny prvky třídy pro testování, pokud nějaké nechcete testovat, odškrtněte je. Výběr prvků, které se nebudou testovat, je vhodné provádět až po zvolení všech tříd. Při jakkékoliv změně zvolených tříd se nastavení resetuje.

Před spuštěním testů můžete nastavit několik parametrů. V prvním textovém poli lze nastavit počet tolerovaných proměnných navíc pro test nadbytečných proměnných. Další textové pole specifikuje konec názvu generovaného souboru – soubor vždy začíná `duckTest` a končí jménem specifikovaném v tomto textovém poli. Standardně je nastaveno na jméno složky se vzorovou úlohou. Poslední možností nastavení je přidání vygenerovaných zdrojových souborů testů do `jar` souboru. Testy se pustí stisknutím tlačítka `Vytvořit testy`. Průběh generace, nebo případná chybová hlášení, se zobrazuje v informační liště.

C Uživatelská příručka pro spouštěč testů

C.1 Účel programu

Spouštěč duck-testů slouží jako uživatelské rozhraní ke spouštění duck-testů vygenerovaných pomocí výše zmíněného generátoru. Program dokáže pomocí zadaného `jar` souboru s testy zjistit jména testovaných tříd, zkontrolovat, zda se nacházejí ve vzorové složce, a pak pro každou třídu spustit příslušný test. Výsledek testu se zobrazí uživateli v okně.

C.2 Popis instalace

C.2.1 Spuštění na OS Windows

Progam není nutné nijak speciálně instalovat. Ke spuštění je potřeba pouze `duckTestSpoustec.jar`. Tento soubor uložte do samostatné složky a dvakrát na něj poklikejte. Tímto se spustí program. Program vyžaduje pro správné fungování Javu 7 nebo vyšší.

C.2.2 Spuštění na OS Linux

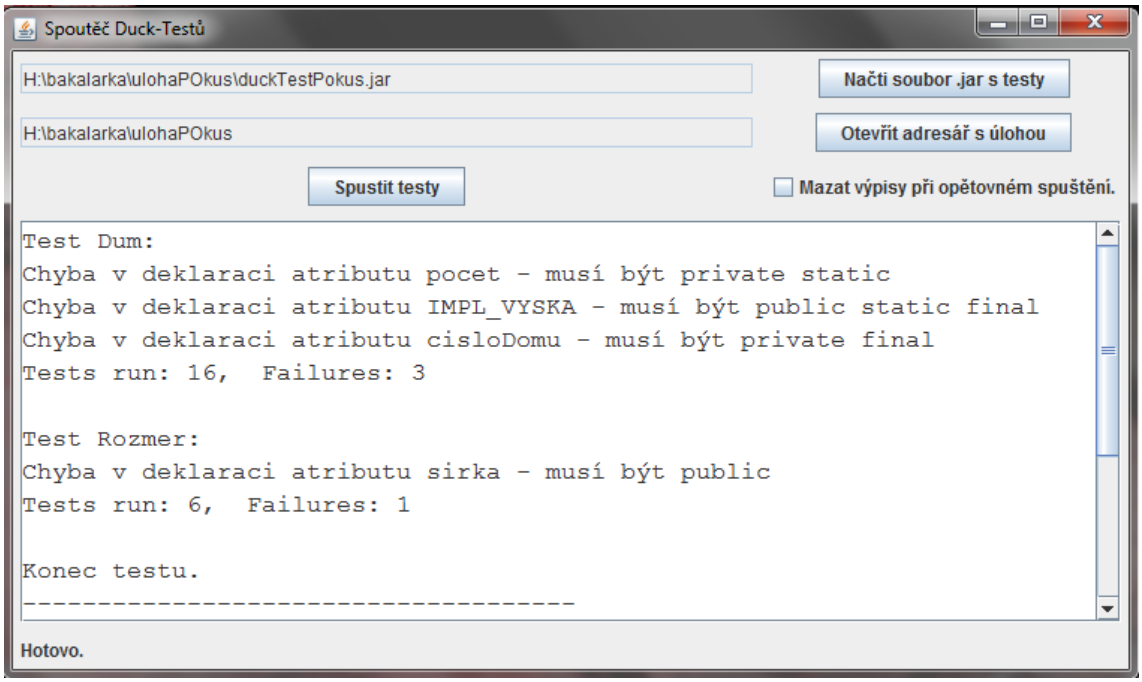
Spustitelný soubor `duckTestSpoustec.jar` slouží ke spuštění programu. Tento soubor je vhodné uložit do samostatné složky. Pomocí příkazu `/usr/lib/jvm/java -jar duckTestSpoustec.jar` se spustí program. Je třeba mít naistalovanou Javu 7 nebo vyšší.

C.3 Práce s programem

C.3.1 Popis uživatelského rozhraní

Uživatelské rozhraní spouštěče je zobrazeno na obrázku C.1. V horní části okna jsou dvě tlačítka. Tlačítko `Načti .jar soubor s testy` slouží pro načtení `jar` souboru s duck-testy. Druhé tlačítko `Otevřít adresář s úlohou` je určeno pro načtení testované úlohy. Textová pole vedle obou tlačítek zobrazují cestu k aktuálně otevřeným souborům.

Pod tlačítky se nachází zaškrtávací volba `Mazat výpisy při opětovném spuštění`, při jejímž



Obrázek C.1: GUI spouštěče

zaškrtnutí se budou mazat staré výpisy. V opačném případě se aktuální výpis testů zapíše za předešlé. Poslední tlačítko je tlačítko `Spustit testy`, které v případě, že jsou zvolené testy i testovaná úloha, spustí testy.

Dolní část okna je vyplněna textovým polem, ve kterém se zobrazují výsledky spuštěných testů, případně zprávy o chybách, které během testů nastanou. Pod textovým polem se nachází ještě informační lišta, na které se zobrazuje aktuální průběh při testování – jméno právě testované třídy a podobně.

C.3.2 Postup při spouštění testů

Před spuštěním testů je potřeba vybrat vytvořený soubor s testy a úlohu, která se bude testovat. Klikněte na tlačítko `Načti soubor .jar s testy` a vyberte soubor s testy. Potvrďte stisknutím `Open`. Vlevo od tlačítka se objeví cesta k souboru s testy a jeho název.

Nyní je třeba vybrat úlohu k otestování. Tentokrát klikněte na tlačítko `Otevřít adresář s úlohou`. Vyberte složku s úlohou. Protože se vybírá složka, nebudou `.java` soubory nebudou vidět, nelekněte se. Potvrďte vybranou složku stisknutím `Open`. Cesta ke složce se opět

objeví vlevo od tlačítka.

Teď je vše připraveno a je možné spustit testy. Stiskněte tlačítko `Spustit testy`. Pokud jste postupovali podle návodu, je vše v pořádku a testy se spustí. Pokud jste nezvolili úlohu nebo soubor s testy, zobrazí se informační zpráva `Musíte nejdřív načíst úlohu a testy`, a testy se nepustí.

První spuštění testů vyžaduje přípravu, a proto bude trvat o něco déle. V tomto případě se uprostřed uživatelského okna objeví informační okénko, které zobrazuje průběh přípravy. Po ukončení přípravy se toto okénko samo zavře a spustí se testy. Některé z nich mohou trvat o něco déle, to závisí zcela na charakteru testovaných tříd a je s tím třeba počítat. Do informační lišty se vypisuje průběh aktuálního testování. Po ukončení všech testů se zobrazí informační zpráva a do textové části okna se vypíše výsledek.

Při testu je možné narazit na několik chybových hlášení. Pokud došlo k chybě během přípravy, je vypsána zpráva `Chyba při načítání testů`. V případě, že `jar` soubor s testy neobsahuje testovací soubory nebo má špatný formát, je vypsána zpráva `Jar soubor neobsahuje testy`. Naopak, pokud složka s testovanou úlohou neobsahuje třídy, které se mají podle souboru `jar` otestovat, je vypsána zpráva `Složka s úlohou neobsahuje všechny testované třídy`. Ve všech těchto případech se nespouštějí testy.

Při spuštění testů může také dojít k několika chybám. Pokud se nepovedlo nějakou třídu přeložit, neprovedou se testy a vypíše se zpráva `Chyba při překladu - zkontrolujte, zda vaše třídy jdou přeložit a jsou v kódování UTF-8`. Při jiné chybě během spouštění testů se vypíše zpráva `Chyba při spouštění testů.`