NÁVRH PODLE KONTRAKTU – KLASICKÁ METODIKA A MODERNÍ NÁSTROJE

Tomáš Pitner[a]

a) Masarykova univerzita v Brně, Fakulta informatiky, Botanická 68a, 602 00 Brno, ČR, e-mail: tomp@fi.muni.cz

Abstrakt

Návrh podle kontraktu (Design by Contract, DbC) je relativně dlouho známým, stále však nedoceněným přístupem k tvorbě spolehlivého programového vybavení. Principem DbC je podrobná specifikace požadovaného chování vytvářené softwarové entity (třídy, balíku, modulu) a následné použití takové vývojové metodiky a nástrojů, aby specifikace byla vytvořeným kódem splněna. Jelikož dnes většina relevantních metodik staví na objektovém (či následně komponentním) paradigmatu, musí být i specifikace softwaru popsána jako souhrn požadavků na chování objektů – tvoří ji specifikace vstupních a výstupních podmínek metod, případných výjimek jimi vyvolaných, vedlejších efektů metod, invariantů na objektech atd. To vše lze definovat i na úrovni rozhraní a pak pomocí nástrojů zajišťovat, že každá implementace rozhraní specifikaci splní.

V současnosti, zejména v souvislosti s rostoucí popularitou agilních metodik vývoje software, jakou je extrémní programování, se DBC znovu dostává do popředí zájmu. DbC se velmi dobře doplňuje s metodikami testování softwarových jednotek (unit testing). V moderních, „živých“, profesionálně používaných jazycích (typicky Java, ale i dalších), se objevují vyspělé volně dostupné i komerční nástroje, které vývoj podle zásad DbC prakticky umožňují. Tento příspěvek představí hlavní charakteristiky DbC, jeho přínosy a vhodné nástroje k praktické aplikaci, jakož i žádoucí kombinaci s nástroji pro testování jednotek v prostředí Java.

1. NÁVRH PODLE KONTRAKTU

1.1 Motivace

Cílem softwarového vývoje je produkovat korektně fungující a robustní programy. *Korektnost* je v tomto smyslu chápána jako soulad výsledného programu se specifikací. *Robustnost* pak představuje míru odolnosti proti nesplnění vstupních podmínek daných specifikací, tj. program se musí přiměřeně „slušně“ chovat i v případě, že jeho klient (obsluha, jiný programový kód) nesplní vstupní podmínku.

Jedním z možných postupů, jak korektnost formálně dokázat, je využít Hoareho kalkul, [1]. Tony Hoare poprvé v roce 1969 ve svém článku v Communications of the ACM popsal sémantiku programu pomocí vstupních a výstupních podmínek. Známá expertka na objektovou metodiku Barbara Liskow navrhla v sedmdesátých letech programovací jazyk CLU, v němž byly tyto podmínky nativní součástí.

Zcela systematicky se tímto směrem ubíral Bertrand Meyer při návrhu nové softwarové metodiky zvané *Design by Contract*[1] – návrh podle kontraktu. Meyer tuto metodiku použil v programovacím jazyce Eiffel. Jedná se tedy o formálně podložený přístup, který je však i prakticky použitelný v mnoha programovacích prostředích a v současnosti podpořený řadou softwarových nástrojů.

[1] „Design by Contract“ je registrovaná obchodní značka B. Meyera, [2], [3].

1.2 Základní princip

Základní zásadou návrhu podle kontraktu je jistá „smlouva“ mezi budoucím uživatelem softwaru a jeho návrhářem. Uživatel se zaváže, že dodrží vstupní podmínky dané specifikací. Návrhář se bude snažit zaručit, že v případě respektování vstupní podmínky volající stranou bude splněna i podmínka výstupní, tzn. program vrátí správný výsledek.

Podpůrné softwarové nástroje mohou pomoci v několika ohledech. Nejprve se věnujme možnostem prověřování dodržení kontraktu ze strany uživatele. Nástroje dovolí v tvořeném kódu naspecifikovat vstupní podmínky a přeložit program tak, aby (většinou za pomocí běhových knihoven) detekoval a signalizoval pokusy o porušení vstupních podmínek.

Podobně je hlídáno i dodržení výstupních podmínek, které jsou typicky specifikovány s využitím výrazů dávajících do vztahu původní stav výpočetního systému (před započetím požadované akce) a stav po provedení dané akce.

Jelikož se v rámci příspěvku omezujeme – v souladu s obecnými trendy – především na vývoj podle objektového (komponentního) paradigmatu, lze pro tuto třídu programů spektrum sledovaných podmínek zjemnit a upřesnit. Nyní popíšeme, co všechno je žádoucí v objektovém programu specifikovat a sledovat. Budeme vycházet z obvyklé terminologie a uvedeme rovněž klíčová slova konkrétního podpůrného prostředí pro DbC – systému *jass*, [4], [5].

1.3 Podmínky ke hlídání

Vstupní podmínka (precondition) určité metody objektu – běhový systém bude hlídat, zda je kontrakt plněn ze strany volajícího, tzn. zda tuto metodu volá se správnými parametry a případně také, je-li objekt v tu chvíli ve vhodném stavu. (*jass* označuje klíčovým slovem `require`)

Výstupní podmínka (postcondition) metody – běhový systém bude hlídat, zda je kontrakt plněn ze strany metody, tzn. jestli metoda vrací, co má – zda je výsledek ve správném vztahu se vstupními parametry. (*jass* označuje klíčovým slovem `ensure`)

Invariant objektu – podmínka, která musí zůstat zachována v každém „klidovém“ stavu objektu – tedy mezi voláními jeho veřejných metod. Invariant může být dočasně porušen mezi voláními metod soukromých. (*jass* označuje klíčovým slovem `invariant`)

Invariant cyklu– podmínka, která musí zůstat zachována v každém průchodu cyklem. Má smysl ho definovat v místě, které je v cyklu opakovaně navštíveno, např. ve chvíli testování vstupní (a současně pokračovací) podmínky u `while`. (též `invariant`)

Variant cyklu – je naopak výrazem, který se s každým průchodem cyklem změní (sníží) a tak zaručí ukončení cyklu po konečném počtu kroků. (*jass* označuje klíčovým slovem `variant`)

V objektovém návrhu je přirozené, že vstupní a výstupní podmínky a invarianty objektů definujeme pro *rozhraní* – pak očekáváme, že každý objekt třídy, která ho implementuje, se bude předepsaným způsobem chovat. Obrovskou výhodou je mít systém, který automaticky pohlídá plnění těchto podmínek v každé implementaci tohoto rozhraní, tj. v každé jeho implementující třídě – aniž bychom do kódu třídy podmínky znovu explicitně psali. Nástroj *jass* toto, bohužel, neumí.

2. NÁSTROJE PRO NÁVRH PODLE KONTRAKTU

2.1 Principy fungování

Podívejme se na možné principy fungování takových nástrojů v prostředí Java:

- Systémy založené na předzpracování zdrojového kódu a vygenerování upraveného zdrojového kódu s kontrolami podmínek. Po překladu tohoto kódu program běží s použitím speciální knihovny.
- Bez předzpracování, ale s nutností použít speciálně upravenou JVM, která podmínky pohlídá na úrovni bajtkódu. Programátor obvykle napíše podmínky ve formě metod s určitými vyhrazenými signaturami. Zavaděč tříd (classloader) musí být upraven tak, aby dokázal při zavedení třídy její bajtkód dynamicky upravit.

2.2 Dostupné nástroje

Obsáhlý článek [6] uvádí přehled nástrojů pro podporu DbC v Javě. Kromě zmiňovaného *jass* jsou volně dostupnými systémy např. *iContract*, *JContract*, *jContractor,* novinka *Contract4J* určená pro Javu 1.5 a další. Vedle toho jsou úspěšné i komerční nástroje jako *JMSAssert*™. Vzhledem k volné dostupnosti a snadnosti použití budeme pro demonstrační účely nadále využívat balík *jass*, [4], [5].

2.3 Příklad použití

Každý budoucí vývojář procházející formálním vzděláváním v návrhu algoritmů pravděpodobně implementoval některý z jednoduchých, snadno pochopitelných, abstraktních datových typů (ADT) – např. zásobník. Tento datový typ lze zavést jako algebraickou strukturu – množinu posloupností nad bázovým typem a množinu operací nad těmito posloupnostmi. Následná implementace v Javě s podporou systému *jass* je realizací takové specifikace. Zdrojové kódy včetně skriptů pro překlad jsou k dispozici na [7].

```
public class Stack implements Cloneable {

    protected int top = -1;

    protected Object[] storage;

    public Stack(int capacity) {
      /** require capacity > 0; **/
      storage = new Object[capacity];
      /** ensure [empty_after_creation] isEmpty(); **/
    }

    public boolean isEmpty() {
      return top < 0;
      /** ensure changeonly{}; **/
    }

    public boolean isFull() {
      return top == storage.length - 1;
      /** ensure changeonly{}; **/
    }

    public Stack push(Object o) {
      /** require [valid_object] o != null;
                  [stack_not_full] !isFull(); **/
      top++;
      storage[top] = o;
      return this;
      /** ensure changeonly{top,storage};
```

```
        [nonempty_after_push] !isEmpty();
        [push_increments_top] Old.top == top - 1;
    **/
  }

  public Object pop() {
    /** require [stack_not_empty] !isEmpty(); **/
    Object o = storage[top];
    top--;
    return o;
    /** ensure changeonly{top}; [valid_object] Result != null; **/
  }

  public boolean contains(Object o) {
    /** require o != null; **/
    int i = 0;
    while (i <= top && !storage[i].equals(o))
        /** invariant 0 <= i && i <= top + 1; **/
        /** variant top + 1 - i **/
    {
        i++;
    }
    return i <= top;
    /** ensure changeonly{}; **/
  }

  protected Object clone() {
    /* Metoda clone() je nutná při použití Old v podmínkách */
    Object b = null;
    try {
        b = super.clone();
    }
    catch (CloneNotSupportedException e){}
    return b;
  }

  /** invariant
     [range] -1 <= top && top < storage.length;
     [valid_content] storage.length == 0
                  || (forall i : {0 .. top} #
                     storage[i] != null);
     [LIFO] isFull() || push("neco").pop().equals("neco"); **/
}
```

Systém *jass* umožnil do implementace ADT zásobník dodat kontroly hlavních podmínek daných axiomy. Kontroly jsou vkládány jako speciální konstrukty v dokumentačních komentářích (`require`, `ensure`, `invariant`...):

- po inicializaci je zásobník prázdný – platí `isEmpty()`
  `ensure [empty_after_creation] isEmpty()`
- po uložení libovolné hodnoty (bez ohledu na předchozí stav) bude zásobník neprázdný
  `ensure [nonempty_after_push] !isEmpty()`

- naposledy uložená hodnota je jako první odebraná: `push(x).pop() = x`
  ```
  invariant [LIFO] isFull() ||
  push("neco").pop().equals("neco")
  ```
- určitý prvek je v (neprázdném) zásobníku právě když je buďto na jeho vrcholu, nebo je obsažen v zásobníku vzniklém odebráním prvku na vrcholu

Kromě axiomů pohlídáme i technicky korektní manipulaci se zásobníkem z hlediska omezenosti jeho kapacity, pohyb s vrcholem zásobníku atd.:

- hodnotu lze vložit, právě když `!isFull()`:
  ```
  require [stack_not_full] !isFull()
  ```
- lze vložit jen neprázdnou hodnotu, tj. o `!= null` (důsledek: v zásobníku jsou na platných pozicích, tj. mezi indexy `0..top` včetně vždy jen nenulové odkazy):
  ```
  require [valid_object] o != null
  invariant [nonnull_content] storage.length == 0
                            || (forall i : {0 .. top} #
                               storage[i] != null)
  ```
- po uložení libovolné hodnoty (bez ohledu na předchozí stav) se index vrcholu zásobníku o jedničku zvětší: `Old.top + 1 = top` (důsledek tohoto a omezené kapacity při vkládání: index `top` je stále v rozsahu `-1` až `storage.length - 1`)
  ```
  ensure [push_increments_top] Old.top == top - 1
  invariant [range] -1 <= top && top < storage.length
  ```
- `isFull()` právě když index vrcholu zásobníku `top` je na poslední pozici v poli, tj. na hodnotě `storage.length - 1`
  (Zde je specifikace tak blízko implementaci, že je těžké chování jakkoli dále nezávisle „zevnitř“ hlídat.)
- Kromě toho je v řadě metod testováno, že mění skutečně jen určité části výpočetního stavu daného objektu `Stack`:
  ```
  ensure changeonly{ metodou_měněné_objekty }
  ```

Při překladu a spouštění takto popsaného programu postupujeme v těchto krocích:

1. předkompilace nástrojem *jass*:
   ```
   java -cp jass.jar;. jass.Jass Stack.jass
   ```
2. překlad výsledku a demo-třídy `StackDemo` běžným překladačem:
   ```
   javac -classpath jass.jar;. Stack.java StackDemo.java
   ```
3. spuštění demo-třídy `StackDemo` s běhovou knihovnou:
   ```
   java -cp jass.jar;. StackDemo
   ```

Výhoda je, že není-li dostupný preprocesor *jass*, lze stejný zdrojový kód bez problémů (ale také bez funkcionality *jass*) přeložit přímo běžným překladačem *javac*.

2.4 Co ještě chtít?

Vítanou praktickou výhodou je, když nástroj není příliš pervazivní a dokáže na požádání (typicky ve finální, distribuční verzi programu) veškeré kontroly potlačit tak, aby po nich, pokud možno, „nezbylo ani stopy“ a rychlost běhu nebyla nijak ovlivněna.

U většiny produktů založených na předzpracování zdrojového kódu obsahujícího specifikace podmínek uvnitř komentářů to nebývá problém – překlad „normálním“ překladačem totiž instrukce pro hlídání podmínek ignoruje.

3. SROVNÁNÍ S NÁSTROJI NA TESTOVÁNÍ JEDNOTEK

Zatímco výše uvedené techniky dovolují hlídat chování „zevnitř“, v mnoha případech je to buďto úplně nemožné (např. nemáme-li zdrojový kód k dispozici pro modifikace) nebo nabízené konstrukty, jak psát hlídané podmínky, jsou příliš těžkopádné a zejména technicky motivované podmínky specifikace se v nich těžko sestavují.

Účinnou kombinací tak může být specifikace podmínek kontraktu např. s využitím *jass* a současné testování jednotek pomocí *junit*. Metodika použití *junit* je dnes již dostatečně známá, zvládnutá a podporovaná většinou užívaných vývojových nástrojů. Na příkladu si ukažme, jak by se některé vlastnosti ADT zásobník otestovaly pomocí *junit* testovací třídy. Zdrojové kódy lze opět najít na [7].

```
public class StackTest extends junit.framework.TestCase {

    private Stack st;

    public StackTest(String testCaseName) {
        super(testCaseName);
    }

    public void setUp() {
        st = new Stack(10);
    }

    public void testEmptyAfterCreation() {
      assertTrue("Stack should be empty after creation.",
                 st.isEmpty());
    }

    public void testNonEmptyAfterPush() {
      st.push("something");
      assertTrue("Stack should not be empty after push.",
                 !st.isEmpty());
    }

    public void testFullAfter10Pushes() {
        for(int i = 0; i < 10; i++) {
            st.push("something");
        }
        assertTrue("Stack should be full after 10x push.",
                   st.isFull());
    }

    public void testLIFO() {
      st.push("something");
      assertEquals("The last pushed value should be popped first.",
                   "something", st.pop());
    }

    public void tearDown() {
    }
}
```

U nejjednoduššího testování jednotek s použitím *junit* tedy píšeme testovací třídy s metodami `setUp()`, `tearDown()` a sadou testovacích metod `testXXX()`. Na provedení testů použijeme spouštěč v textovém nebo grafickém režimu, velmi často přístupný přímo z vývojového prostředí (např. v Eclipse, Netbeans). Překlad a spuštění testu v okénkovém prostředí vyvoláme příkazy:

```
java -cp jass.jar;. jass.Jass Stack.jass
javac -classpath junit.jar;jass.jar;. Stack.java StackTest.java
java -cp junit.jar;jass.jar;. junit.swingui.TestRunner StackTest
```

3.1 Co je vhodnější?

Kdy použít nástroje na DbC s běhovým ověřováním podmínek (typu *jass*) a kdy testování jednotek (s nástroji jako je *junit*)? Shrňme přednosti a nedostatky obou přístupů:

- Testování jednotek je spíše testování zvenčí, „blackbox testing“, použitelné i když nemáme zdrojový kód testovaných tříd.
- Testování jednotek je podporováno většinou vývojových nástrojů.
- Při testování jednotek nejsme odkázáni na speciální jazyk výrazů použitelných v podmínkách – testy programujeme přímo v Javě, jak jsme zvyklí, pouze s použitím speciálních metod `assertTrue()`, `assertEquals()` a podobných.
- Nástroje typu *jass* umožní naproti tomu testovat chování i uvnitř metod, cyklů,... („whitebox testing“).
- Rovněž na rozdíl od *junit* dovolují hlídat dodržení kontraktu ze strany volajícího/uživatele dané třídy.
- Rozdíl je též v autorství kontrolních a testovacích podmínek: zatímco v *jass* tvůrci programu obvykle sami vpisují i kontrolní podmínky – i když toto je spíše dáno praktickými důvody – vše je v jednom zdrojovém souboru, u testování jednotek je psaní testů záležitostí nezávislého testera. Vývojové nástroje většinou dovolují separátní údržbu testovacích tříd.

Celkově lze říci, že dosavadní profesionální praxe jednoznačně více přijala nástroje na testování jednotek než systémy běhového ověřování specifikace. Testování jednotek se obejde bez předzpracování testovaných zdrojových kódů, je podporováno vývojovými nástroji a není třeba se učit nový jazyk na specifikaci podmínek – stačí umět použití API testovacího rámce.

To ovšem neznamená zavržení systémů na běhové ověřování specifikace – např. chceme-li hlídat chování uvnitř metod nebo dodržení kontraktu ze strany volajícího, jinou možnost nemáme. Jistou alternativou komplexních nástrojů typu *jass* je nový příkaz `assert` dostupný od verze Java 1.4, viz [8]. Výhodou `assert` je, že je součástí jazyka, nejsou tedy potřeba žádné knihovny třetích stran.

4. ZÁVĚR

Přestože uvedené techniky nevedou k formálním důkazům korektnosti programů, jsou schopny detekovat celou řadu běžných programátorských chyb. V kombinaci s testováním jednotek tak napomáhají psaní korektních a robustních programů.

V současnosti neexistuje vážný racionální důvod, proč uvedené technologie v profesionálním vývoji nepoužívat. Přinejmenším ve zde diskutovaném javovém prostředí nabízí několik slušně vybavených komerčních i volně dostupných nástrojů (*JMSAssert, jass*) integrovatelných např. pomocí systémů pro sestavování programů *Ant* nebo *Maven* do libovolného projektu. Širšímu využití brání tedy spíše setrvačnost, neochota se učit používat

nové prostředky a často také neznalost principů, na nichž je DbC postaven. K odstranění takových překážek se snažil přispět i tento článek.

LITERATURA

[1] GUMM, H. P. The Hoare Calculus, http://www.csi.uottawa.ca/ordal/papers/peter/node13.html.

[2] MEYER, B. Object-Oriented Software Construction, Second Edition. Prentice Hall Professional Technical Reference, 1997, ISBN 0-13-629155-4

[3] HÖLZL, M. Design by Contract. http://www.gauss.muc.de/tools/dbc/dbc-intro.html

[4] BARTEZKO D., FISCHER C., MÖLLER M., WEHRHEIM H.: Jass – Java with Assertions, Electronic Notes in Theoretical Compuer Science, Proceedings of RV 01, Paris, France, Volume 55, Issue 2, July 2001

[5] BARTETZKO, D. et al. Java with ASSertion (jass). http://csd.informatik.uni-oldenburg.de/~jass/, University of Oldenburg, Germany

[6] PLÖSCH, R. Evaluation of Assertion Support for the Java Programming Language. http://www.jot.fm/issues/issue_2002_08/article1

[7] PITNER, T. Design By Contract s nástrojem jass. http://www.fi.muni.cz/~tomp/dbc

[8] SUN MICROSYSTEMS. Programming With Assertions. http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html