TESTOVÁNÍ SOFTWARU NA PLATFORMĚ JAVA EE

Tomáš Pitner[a]

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

Abstrakt
Vytvořit funkčně úplný, spolehlivý a robustní programový systém je cílem každého vývojáře. Řada soudobých metodik vývoje (Test-driven Development, Extreme Programming) chápe testování jako jeden ze svých základních pilířů. Příspěvek se zaměří především na softwarové nástroje na podporu testování rozsáhlých aplikací na platformě Java Enterprise Edition (Java EE, dříve J2EE) a ukáže jejich praktickou použitelnost.

1. TESTOVÁNÍ SOFTWARU

1.1 Motivace

Cílem softwarového vývoje je produkovat korektně fungující programové systémy, které kromě toho splňují i další požadavky, mezi něž patří bezpečnost, robustnost, efektivita, rozšiřitelnost atd. Netřeba připomínat, že i u velkých softwarových projektů se to beze zbytku dodržet nepodaří.

K obecně známým případům selhání vývojářů – někdy s velmi vážnými důsledky – jistě patří problém roku 2000, na nějž bylo založeno už v sedmdesátých letech, tj. dávno předtím, než se jeho následky mohly projevit či vůbec předpovědět.

Začátkem devadesátých let se firmě Intel „podařilo“ do prvních Pentií vnést konstrukční chybu v počítání v pohyblivé řádové čárce. O chybě přitom vedení firmy vědělo – nepovažovalo však za vhodné kvůli tomu zdržet uvedení procesoru na trh – a následně rozsah závady bagatelizovalo. Pod tlakem uživatelů nakonec nabídlo výměnu postižených čipů za opravené.

Chyby mívají, bohužel, i tragické následky. Nehoda přistávacího modulu NASA na Marsu v roce 1999 vedla „jen“ k obrovským finančním a morálním ztrátám. Naproti tomu selhání naváděcího systému amerických obranných raket Patriot během první války v Perském zálivu stálo bezprostředně životy několika desítek amerických vojáků.

Moderní metodiky vývoje (zhruba od konce devadesátých let) se proto snaží pojímat testování v širokém slova smyslu jako integrální součást vývoje. Dospělo se k poznání, že až následné řešení následků chyb je dražší a bolestivější, protože náklady rostou geometrickou řadou s tím, v jak pokročilé fázi vývoje či nasazení jsou odhaleny.

1.2 Chyby softwaru

Základním východiskem návrhu softwarového celku je *specifikace*, tj. jistá „smlouva“ mezi budoucím uživatelem softwaru a jeho návrhářem. Specifikace, ať už formální či neformální, vznikající jednorázově nebo i průběžně při vývoji – čehož jsme svědky u agilních metodik – dává východisko k posuzování, co vlastně chyba je. Velkou skupinu chyb lze definovat jako nesoulad mezi specifikací a výsledným softwarem. Program nedělá, co by dělat měl (nebo to dělá nekorektně), dělá naopak věci, které by neměl (nebo nemusí dělat). Chybou ovšem může být i nedostatek ve specifikaci vedoucí k tomu, že software nedělá něco, co ve specifikaci sice není, ale uživatel to vyžaduje. Konečně za chybu lze považovat i stav, kdy systém sice

odpovídá specifikaci a v zásadě funguje, nicméně uživatel není spokojen s jeho obsluhou či rychlostí odezvy [16].

1.3 Testování softwaru

Ať už uvažujeme jakoukoli metodiku vývoje, vždy by v ní měl být prostor pro úlohu *testera*. Jeho cílem je *vyhledávat chyby, vyhledat je co nejdříve a zajistit jejich nápravu* [16]. Známé zásady formulované Scottem Amblerem [1] říkají, že bychom měli testovat během celého životního cyklu projektu, vyvíjet testy ještě před samotným kódem, (kontinuálně) testovat všechny artefakty programu, testováním odhalovat příčiny chyb a nepřekrývat je, při testování používat přitom jednoduché a efektivní nástroje a testování zahrnout po všech stránkách do vývoje.

2. TESTOVÁNÍ APLIKACÍ V PROSTŘEDÍ JAVA EE

2.1 Prostředí Java EE

Java Enterprise Edition představuje plnohodnotnou platformu pro budování rozsáhlých, často komponentně orientovaných programových systémů schopných běhu ve specializovaných prostředích označovaných jako aplikační servery. Prostředí poskytuje komponentám middlewarové zázemí pro jejich vývoj, nasazení, konfiguraci, běh, vzájemnou komunikaci a sledování. Komplexnost Java EE aplikací i běhového prostředí ale znamená mimořádné nároky na testování. Systémy zahrnují řadu aplikačních vrstev a jsou často distribuované mezi více fyzických počítačů – uzlů sítě.

2.2 Co a jak testovat?

Typická Java EE aplikace na bázi Enterprise JavaBeans (EJB) má klientské webové rozhraní, jenž komunikuje se servlety a stránkami JSP na aplikačním či webovém serveru. Tato část prezentační vrstvy je od vlastní aplikační logiky realizované Session Beans izolována fasádou z komponent, data jsou držena jako komponenty Entity Beans s perzistencí zajištěnou obvykle relační databází. Situace je navíc zpestřena tím, že tento plný aplikační model se v mnoha případech „očesává“ – například místo „těžkých“, na zdroje náročných, Entity Beans se používá objektově-relační mapování běžných (POJO) objektů a pro řízení na vyšších vrstvách se využívají webové rámce.

V dalším si budeme všímat jednotlivých vrstev typické Java EE aplikace a nastíníme, jaké testovací techniky a nástroje lze použít – zejména ty, s nimiž lze aplikaci testovat komplexně, na více vrstvách.

3. KONKRÉTNÍ TECHNIKY A NÁSTROJE

3.1 Obecně použitelné techniky

V javovém prostředí nelze nezačít známým nástrojem JUnit [9], určeným pro *unit testing*, tedy *testování jednotek programu*. Jednotkou se myslí třída nebo skupina (balík) tříd poskytující jako celek určitou funkcionalitu navenek. Pomocí JUnit vytvoříme *test*, v nejjednodušší podobě jako *testovací třídu* obsahující *testovací metody*. Testy pracují tak, že si nejprve vytvoří prostředí složené z *přípravků* (fixture) – objektů, které budou v testech figurovat, na nichž se budou spouštět testované metody a ověřovat jejich výsledky. Existují spouštěče, které testy spustí, vyhodnotí a sdělí jejich úspěšnost a případně generují testovací protokoly.

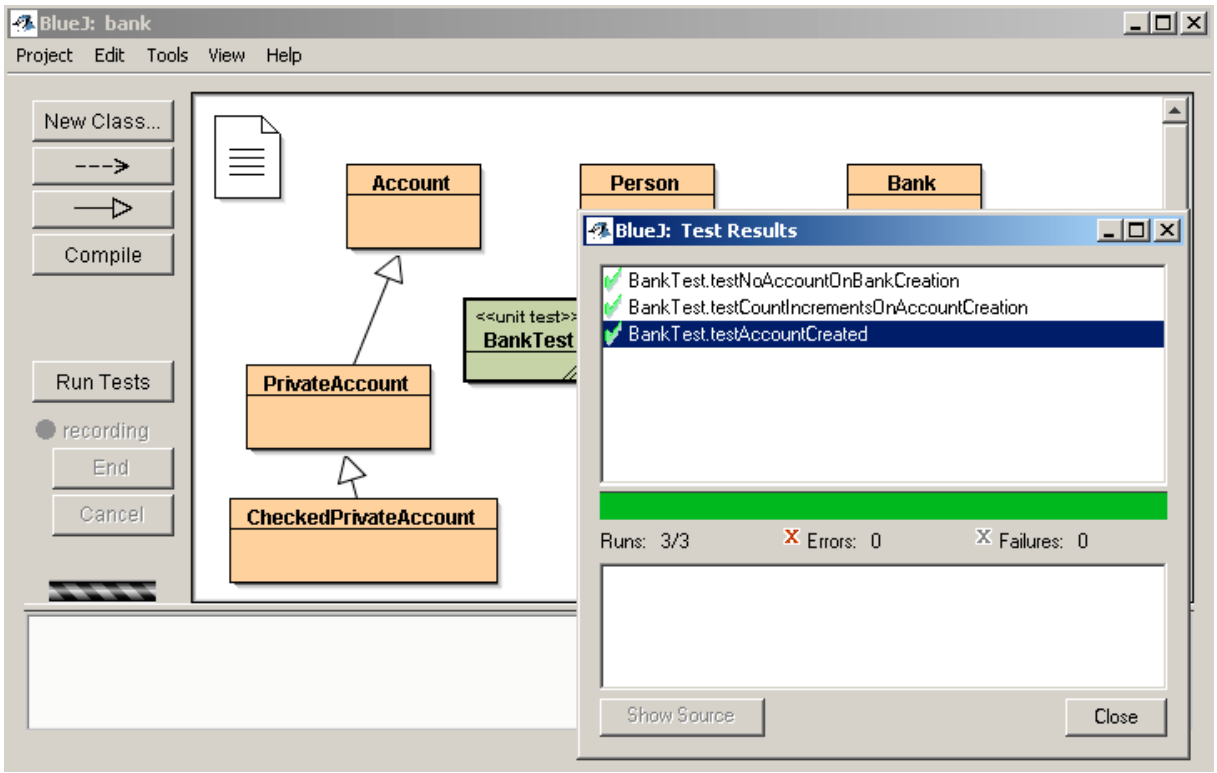
Ukázka fragmentu JUnit testovací třídy k prověření třídy Stack:

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

```
    private Stack st;

    public StackTest(String testCaseName) {
        super(testCaseName);
    }
    // vytvoří přípravek, nastaví prostředí každého testu
    public void setUp() {
        st = new Stack(10);
    }
    // testuje prázdnost právě vytvořeného zásobníku
    public void testEmptyAfterCreation() {
      assertTrue("Stack should be empty after creation.",
                 st.isEmpty());
    }
    // testuje neprázdnost zásobníku po vložení prvku
    public void testPushPopEquals() {
      st.push("something");
      assertEquals("What was pushed, must be popped...",
                 "something", st.pop());
    }
    ...
    // uklidí po testu, je-li třeba
    public void tearDown() {  }
}
```

Spuštění (jiného) JUnit testu v prostředí BlueJ:



3.2 Integrace testování do vývoje

S testováním úzce souvisí moderní prostředky vývoje. Nástroje na *správu vývoje* (sestavování, dokumentace, distribuce aplikací), jako je volně dostupný Maven [14] nebo starší Ant, ale i IDE (NetBeans, Eclipse, IDEA a dokonce i výuková - BlueJ) mají podporu JUnit testování integrovanou. Dnes už se ani nepředpokládá, že by k jakémukoli netriviálnímu

projektu testy neexistovaly – jasně je to vidět na open-source vývoji, kde projekt bez testů rovná se ostuda…

K modernímu týmovému vývoji nezbytně patří *systémy správy zdrojových kódů* zajišťující jejich centralizovanou evt. distribuovanou správu s podporou verzování. Známým a rozšířeným systémem řízení verzí je CVS, postupně nahrazované systémy modernějšími, jako je Subversion [19]. Při neúspěšnosti integračních testů se tak můžeme v repozitáři vrátit ke starší verzi určitého modulu a pokusit se chybu najít. Samozřejmě platí, že jednotky chybné samy o sobě by se do repozitáře ani neměly dostat.

Jakmile uživatelé nebo automatické testy zjistí chyby, je na místě použít systém jejich evidence. Známá je BugZilla [3], kde se chyba zaznamená spolu s následným postupem jejího řešení.

3.3 Testy datové vrstvy

DbUnit [6] je rozšířením JUnit pro testování databázových aplikací. Motivace je jasná – typickým problémem testování databázových aplikací je uvedení prostředí před testem do vhodného stavu a po testu je "uklidit". To znamená vytvořit databázi, tabulky, naplnit je daty, spustit testovaný kód (který může data totálně zničit) – a po skončení testu uvést opět do konzistentního stavu – např. pro spouštění testů dalších, které nesmějí na výsledku předchozího testu záviset. DbUnit dokáže podle definičního souboru vytvořit tabulky, z XML či jiných zdrojů do nich načíst data a spustit testy.

Příklad popisu testů pro DbUnit:

```
public class SampleTest extends DatabaseTestCase {
    ...
    // vrátí spojení na databázi
    protected IDatabaseConnection getConnection() throws Exception { ... }
    // vytvoří dataset z předchystaného datového souboru
    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSet(new FileInputStream("dataset.xml"));
    }
    // a nyní vlastní testování
    public void testMe() throws Exception {
        // zde se zavolá testovaný kód modifikující DB
        ...
        // získají se data z takto modifikované databáze
        IDataSet databaseDataSet = getConnection().createDataSet();
        ITable actualTable = databaseDataSet.getTable("TABLE_NAME");
        // zavedou se očekávaná data z externího souboru
        IDataSet eDS = new FlatXmlDataSet(new File("expectedDataSet.xml"));
        ITable expectedTable = eDS.getTable("TABLE_NAME");
        // skutečný stav se porovná s očekávaným
        Assertion.assertEquals(expectedTable, actualTable);
    }
}
```

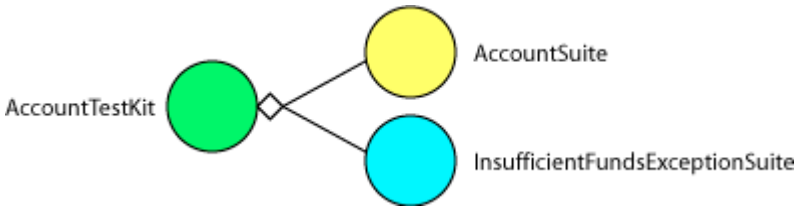
3.4 Testy aplikační vrstvy

Na aplikační vrstvě má své pevné místo JUnit, jímž můžeme testovat logiku většiny jednotek. Existuje řada nadstaveb, ať už k uživatelskému zpříjemnění, řízení, organizaci výsledků a protokolování testů. Například potřebujeme-li při testech vytvářet mnoho objektů s různými

daty, lze použít nástroj [7], který umožní načíst jednoduše, textově v souboru uložená, data a použít je při opakovaném vytváření a testování datově bohatých objektů.

Někteří výrobci místo nadstaveb JUnit nabízejí alternativní testovací rámce – např. *Artima SuiteRunner* [2], s pokročilejší koncepcí, poučenou z chyb JUnit. Jednou z jeho předností je lepší – např. stromová – organizace testů do skupin (suite), možnost vracet se k historii testů, atd.

Ukázka strukturovaného testu pro Artima SuiteRunner:



Komplexní integrační testy lze řídit například systémem *Cactus* [5]. U velkých systémů je mj. podstatné, jak spravují výsledky testů.

Příklad zdrojového tvaru protokolu o testu (Cactus):

```
<?xml version="1.0" encoding="UTF-8" ?>
- <testsuites>
  - <testsuite name="TestSampleServlet" tests="1" failures="0" errors="0" time="0.27">
      <testcase name="testSaveToSessionOK" time="0.2" />
    </testsuite>
  </testsuites>
```

3.5 Testy prezentační vrstvy – webové rozhraní

Kromě velkých systémů na integrační testování lze přímo pro webovou vrstvu použít jednoúčelové nástroje – např. JWebUnit [11] – a intuitivně v něm psát testy, které za nás realizují komunikaci se serverovou částí aplikace a testují její výstupy.

Příklad testu webové aplikace v JWebUnit, [11]:

```
public class JWebUnitSearchExample extends WebTestCase {
   ...
   public void setUp() {
      getTestContext().setBaseUrl("http://www.google.com");
   }
   public void testSearch() {
      beginAt("/"); // na kterém URL začít
      setFormElement("q", "httpunit"); // co na stránce vybrat
      submit("btnG"); // jaké tlačítko stisknout
      clickLinkWithText("HttpUnit"); // kam kliknout
      assertTitleEquals("HttpUnit"); // co má od serveru přijít
      assertLinkPresentWithText("User's Manual");
   }
}
```

3.6 Testy prezentační vrstvy – grafické rozhraní

Grafické uživatelské rozhraní se obecně testuje těžko, aplikaci je však i tak vhodné před testy použitelnosti předložit automatizovanému testeru, který se skrz ni virtuálně „prokliká“, zadá vstupy do dialogů, vybere ze seznamů,… a zkontroluje výsledky. Dá se to zařídit např. záznamy uživatelských akcí člověka a jejich „přehrání“ jako makro, nebo skriptovat akce nad GUI pomocí specializovaného nástroje, jako je např. Marathon [12].

Ukázka skriptování při testech GUI nástrojem Marathon, [12]:

```
username = 'cowboydan'
password = 'sixshooter'
click('Create User')
select('usernameField', username)
select('passwordField', password)
click('Confirm')
myUserTable.assertContains(username,password)
myMessages.assertNewUserMessage(username,password)
```

3.7 Mock-objekty

Při testování Java EE aplikací řešíme často problém konfigurace a vůbec zajištění prostředí pro běh testů. Ustavení tohoto prostředí je natolik náročné, že nás může brzdit a nutit testy odkládat, což je chyba. Proto se pro testování dílčích vrstev používají tzv. *mock-objects* (zástupné, nepravé objekty), které do jisté míry – pro účely testování – nahrazují reálné prostředí běhu komponenty. Mock-objekty si můžeme vytvářet sami, buď „na zelené louce“, s pomocí nástrojů, nebo použít hotová prostředí jako je Mockrunner [15].

Např. datová vrstva používá JDBC rozhraní a databázi. Jak jsme uvedli výše, můžeme ji plnohodnotně testovat např. přes dbunit. Ale k tomu, abychom pouze ověřili, že aplikační logika správně navazuje spojení na databázi a klade dotazy ve správném pořadí a ve správnou chvíli, postačí nahradit skutečnou databázi mock-objektem poskytujícím nepravé, zástupné spojení na databázi, a evidujícím pořadí a obsahy volání.

Rovněž tak webové (servletové) aplikace – jejich opakované sestavování a nasazování je velmi časově náročné, vynucuje si znovuspouštění webových kontejnerů. Pak je např. místo Cactus lepší použít mock-objekt simulující reálný kontejner a na něm servlet jednoduše testovat.

Příklad testu servletu v prostředí Mockrunner, [15]:

```
public class RedirectServletTest extends BasicServletTestCaseAdapter {
    protected void setUp() throws Exception {
        super.setUp();
        createServlet(RedirectServlet.class);
    }
    public void testServletOutputAsXML() throws Exception {
        addRequestParameter("redirecturl", "http://www.mockrunner.com");
        doPost();
        Element root = getOutputAsJDOMDocument().getRootElement();
        assertEquals("html", root.getName());
        Element head = root.getChild("head");
        Element meta = head.getChild("meta");
        assertEquals("refresh", meta.getAttributeValue("http-equiv"));
        assertEquals("0;URL=http://www.mockrunner.com",
                     meta.getAttributeValue("content"));
    }
}
```

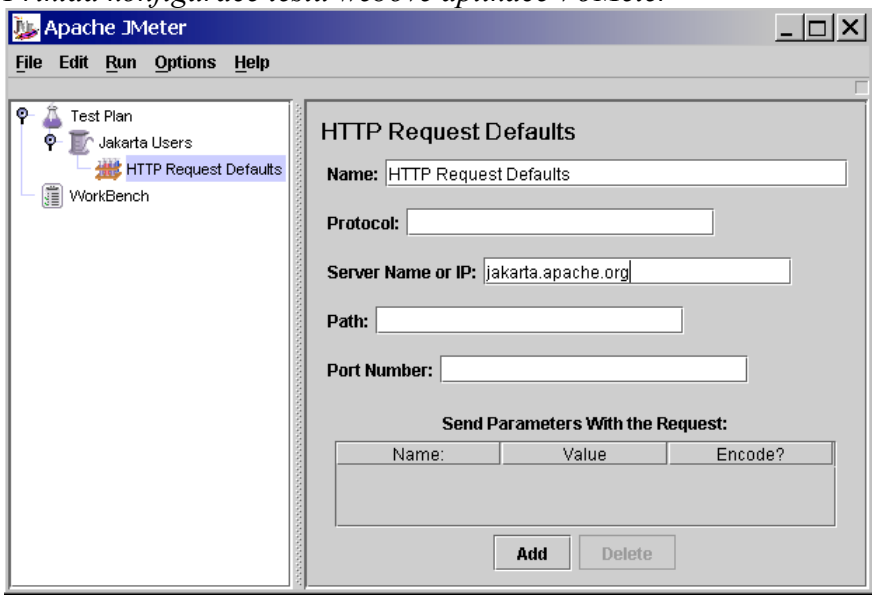
Obdobně oceníme mock-testování programů využívajících systémů řízení zpráv (na bázi Java Messaging Systems). Nepotřebujeme plné, ale pomalé a náročné implementace JMS, pro otestování komunikace přes JMS stačí Mockrunner. Ten, konečně, dokáže simulovat i prostředí pro hostování Enterprise JavaBeans.

Tak můžeme postupovat s testováním jednotek (vrstev) až do finální fáze, kdy musíme přikročit k testování v prostředí, kde aplikace v praxi poběží.

3.8 Zátěžové testy

K testování patří i zkoumání aplikace pod zátěží – jaká pak v reálném nasazení nastane. Nástroj JMeter [10] umožňuje spouštět testy funkčnosti a zátěžové testy Java EE aplikací, vč. testů grafického rozhraní. Nabízí přitom uživatelsky komfortní prostředí:

Příklad konfigurace testu webové aplikace v JMeter



3.9 Monitoring a záznam činnosti aplikace

Významnou skupinou jsou nástroje pro sledování (monitoring) a zaznamenávání (logging) běhu programu či celého programového systému.

Přes pět let je vývojářům v Javě k dispozici mocný, snadno použitelný a přitom bohatě konfigurovatelný, volně dostupný nástroj Log4J (http://logging.apache.org/log4j), jenž byl následně přenesen pod podobnými názvy i do dalších prostředí (C++, .NET, Python, PL-SQL, PHP, Perl). Do Javy 1.4 se jako reakce na popularitu Log4J dostalo podobné API (v balíku java.util.logging), takže dnes lze ve všech soudobých verzích JDK používat zaznamenávání bez potřeby knihoven třetích stran.

Schéma využití je následovné – programátor se rozhodne, do kterých částí výkonného kódu umístí volání sledovacích metod, jež za běhu programu provedou záznam o jeho činnosti – většinou s identifikací, kdy byly volány, místa, odkud byly volány (třída, metoda). Vývojář kromě toho může nechat zachytit i podrobnější popis stavu programu v daném místě – formou libovolné hlášky či objektu výjimky, která vznikla, apod.

Nástroje záznamu většinou disponují možností závažnost záznamových volání kategorizovat. Je tak možné odlišit záznam čistě informativní a ladicí od signalizace vážných chyb.

Při vývoji a provozu rozsáhlejších systémů uvítáme možnost nastavit záznamovou politiku různě pro různé části kódu – např. v odladěných komponentách potlačíme záznam ladicích událostí. Konfiguraci politiky děláme obvykle deklarativně, v souboru popisovače, a tak je možné ji pohodlně, operativně, někdy i dynamicky za běhu systému měnit.

Některá specifická Java EE API (např. JavaServlet) nabízejí přístup k záznamovým možnostech aplikačního prostředí, v tomto případě kontejneru webových aplikací. O vlastní zpracování tohoto volání se postará kontejner, jenž lze nakonfigurovat tak, aby záznam buď prostě vypsal na chybový výstup, připsal do souborů (které mohou „rolovat“, a tak po určitou dobu záznamy uchovávat), vložil do databáze nebo dokonce poslal e-mailem správci aplikace.

Aplikační rámce, viz [17], nabízejí často také vlastní řešení záznamů, takže softwarový architekt každé větší Java EE aplikace musí řešit, co a jak použít – i s ohledem na znovupoužitelnost vytvořeného kódu, kterou těsnější vazba na konkrétní aplikační rámec může ztížit.

Pokud jde o řízení a monitoring komponent Java EE systému – což už nespadá výlučně pod testování –, nabízí se použití speciálního API Java Management Extension (JMX). Běh lze sledovat prostřednictvím „agentů" nasazených do aplikace či systému a komunikujících s klientským rozhraním.

4. ZÁVĚR

Tento příspěvek volně navázal na článek *Návrh podle kontraktu* [18], který byl věnován postupům a nástrojům pomáhajícím při odhalování chyb programů „zevnitř", pomocí specifikací podmínek, které jsou následně za běhu testovány. Testování programů však představuje mnohem rozsáhlejší úkol, než je ověřování dílčích podmínek na mikroúrovni.

Přestože ani zde uvedené techniky testování nás nedovedou k formálním důkazům korektnosti programů ani nenahradí testování použitelnosti lidmi, jsou schopny detekovat celou řadu běžných vývojářských chyb.

LITERATURA

[1] AMBLER, S. J2EE Testing Primer. SD Magazine. http://www.sdmagazine.com, 2001

[2] Artima SuiteRunner, http://www.artima.com/suiterunner

[3] BugZilla. http://bugzilla.mozilla.org

[4] BUCHALCEVOVÁ, A. Jaký má být dnes vývoj softwaru - business driven, test driven, model driven, architecture driven nebo service oriented?. Konference EurOpen 2005

[5] Jakarta Cactus. http://jakarta.apache.org/cactus

[6] DbUnit. http://dbunit.sourceforge.net

[7] djunit. http://sourceforge.net/projects/tomp

[8] Volně dostupné testovací nástroje pro Javu, http://java-source.net/open-source/testing-tools

[9] JUnit, http://junit.org

[10] JMeter, http://jakarta.apache.org/jmeter

[11] JWebUnit. http://jwebunit.sourceforge.net

[12] Marathon. http://marathonman.sourceforge.net

[13] MATULÍK, P., PITNER, T. Podpora aplikační logiky v J2EE aplikacích. In Sborník konference Objekty 2005, VŠB-TU, Ostrava, 2005, ISBN 80-248-0595-2. http://www.cs.vsb.cz/objekty/2005/download/objekty2005.pdf

[14] Apache Maven. http://maven.apache.org

[15] Mockrunner. http://mockrunner.sourceforge.net

[16] PATTON, R. Testování softwaru. SAMS, Computer Press, Praha, 2002, ISBN 80-7226-636-5

[17] PITNER, T. Webové aplikační rámce. In Sborník konference Tvorba software, ČSSI, Ostrava, 2004, ISBN 80-85988-96-8

[18] PITNER, T. Návrh podle kontraktu. In Sborník konference Tvorba software, ČSSI, Ostrava, 2005, ISBN 80-86840-14-X

[19] Subversion. http://subversion.tigris.org

[20] XMLUnit. http://xmlunit.sourceforge.net