UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Miran Okorn

NABOR TESTNIH ORODIJ ZA RAZVIJALCA JAVANSKIH APLIKACIJ

DIPLOMSKO DELO NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

MENTOR: viš. pred. dr. Igor Rožanc

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika dela:

Dobro izbran nabor testnih orodij lahko bistveno pripomore k učinkovitemu delu programerja in kakovosti njegovih končnih izdelkov. V diplomski nalogi izpostavite nekaj ključnih področij testiranja, nato pa za vsako od teh področij predstavite večji nabor sodobnih odprtokodnih orodij za testiranje javanskih aplikacij v razvojnem okolju Eclipse. Na koncu na podlagi objektivnih kriterijev in možnosti sodelovanja med orodji predlagajte optimalen nabor testnih orodij za izkušenega javanskega razvijalca.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Miran Okorn, z vpisno številko 63010252, sem avtor diplomskega dela z naslovom:

Nabor testih orodij za razvijalca javanskih aplikacij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca,

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.), ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,

- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 21. september 2014 Podpis avtorja:

ZAHVALA

Rad bi se zahvalil mentorju viš. pred. dr. Igorju Rožancu za podeljeno zanimivo temo za diplomsko nalogo, za spodbujanje in bodrenje, za optimizem in prijaznost ter pomoč pri izdelavi diplomske naloge. Hvala Slavku Mahnetu Shyami in Zali Mahne za pomoč in ekstremno voljo za opravljanje izpitov ter za dokončanje študija. Hvala moji čudoviti punci Snežki za vso pomoč, za podporo, za spodbujanje, za razumevanje in potrpežljivost. Hvala mamici, očiju in mami za vso finančno in moralno podporo pri študiju, za skupno veselje pri opravljanju izpitov ter za spodbujanje. Hvala bratu Petru za polno nasvetov in tehnični pomoči v času študija. Hvala vsem mojim prijateljem, sodelavcem, sošolcem in znancem za pomoč, da smo skupaj opravili vse študijske obveznosti. Hvala.

Seznam uporabljenih kratic

HTML - Označevalni jezik za izdelavo spletnih strani, (*ang. HyperText Markup Language*)
XML - Razširljiv označevalni jezik, (*ang. Extensible Markup Language*)
JRE - Javansko izvajalno okolje, (*ang. Java Runtime Environment*)
TXT - Datotečna končnica za besedilne datoteke, (*ang Filename extension for text files*)
JAR - Javanska arhivska datoteka (*ang. Java Archive*)
JDK - Skupek javanskih razvojnih orodij, (*ang. Java Development Kit*)

Kazalo vsebine

Povzetek

Namen diplomske naloge je določiti kriterije in predlagati nabor različnih testnih orodij za različne vrste testiranja, ki pomagajo programerju pri učinkovitem programiranju javanskih aplikacij v razvojnem okolju Eclipse. Pri tem smo se omejili na naslednje ključne vrste testiranja: testiranje enot, testiranje in analiza sintakse, testiranje varnosti v kodi spletnih aplikacij in profiliranje aplikacij. Pri testiranju enot smo preučili najbolj razširjeni ogrodji JUnit in TestNG, pri testiranju in analizi sintakse pa odlična orodja FindBugs, PMD, CheckStyle in platformo SonarQube. Pri testiranju varnosti smo preverili najbolj obetavna vtičnika Find Security Bugs in OWASP (Excentia). Za profiliranje aplikacij sta obetavni orodji VisualVM in JVM Monitor. Omenjena orodja smo prikazali na kratkem zgledu in prikazali njihovo delovanje. V nadaljevanju smo določili naslednji izbor smiselnih kriterijev: dokumentiranost orodja, enostavnost namestitve, prijazen uporabniški vmesnik, učna krivulja in funkcionalno pokrivanje orodja. Na podlagi kriterijev smo predlagali končni nabor testnih orodij, ki javanskemu programerju omogočajo učinkovit razvoj kakovostne kode.

Ključne besede: testiranje, testna orodja, testiranje enot, odkrivanje napak, varnost, profiliranje.

Abstract

The purpose of the thesis is to determine the criteria and to propose a set of different test tools for several different types of testing that help the programmer at the efficient programming of java applications in the Eclipse development environment. In this we limited ourselves to the following key types of testing: testing the units, testing and the analysis of syntax, testing the security in the code of web applications and profiling the applications. When testing the units, we studied the most spread frameworks JUnit and TestNG; when testing and analyzing the syntax we studied outstanding tools FindBugs, PMD, CheckStyle and the platform SonarQube. When testing the security we checked the most promising plug-ins Find Security Bugs and OWASP (Excentia). For profiling applications the promising tools are VisualVM and JVM Monitor. All tools are presented by a short example and their operation is also presented. In continuation we determined the following selection of the logical criteria: documentation of the tools, simplicity of installation, user-friendly interface, learning curve and functional covering of the tools. On the basis of the criteria we proposed the final set of test tools, which enable a java programmer an efficient development of a qualitative code.

Keywords: testing, test tools, unit testing, detection of defects, security, profiling.

1. Uvod

Veliko razvijalcev aplikacij razvija svoje rešitve v programskem jeziku Java v razvojnem okolju Eclipse. Porodila se nam je želja, da bi imeli na voljo določen nabor testnih orodij, s katerimi bi lahko preverili svojo izvorno kodo še v času razvoja in sproti odpravljali morebitne napake, ki lahko v kodi obstajajo. Določene napake se v praksi lahko pojavijo šele kasneje, ko npr. našo aplikacijo uporabljajo že končni uporabniki in takrat določene napake odkrijejo. Čas za odpravo napake, ki jo odkrijejo končni uporabniki, je neprimerljivo večji od časa odprave napake, ki se jo odkrije še v času razvoja.

Radi bi torej zasnovali svoj nabor orodij za testiranje, da si z njimi pomagamo pri odkrivanju napak in pri tem izboljšamo kakovost ter varnost spisane izvorne javanske kode. Želja je, da bi bila orodja po možnosti brezplačna, enostavna za namestitev in uporabo, so združljiva z razvojnim okoljem Eclipse, so dokumentirana in se jih da hitro naučiti.

Orodja za testiranje, ki jih imamo v mislih za naš nabor, bomo predstavili v 2. poglavju. Najprej bomo predstavili orodja za testiranje enot (*ang. unit testing tools*). Zatem bomo predstavili orodja, ki v izvorni kodi odkrijejo napake, katerih sam prevajalnik (*ang. compiler*) ne uspe odkriti. Nato bomo opisali orodja, ki se uporabljajo za odkrivanje varnostnih tveganj v izvorni kodi spletnih javanskih aplikacij. V nadaljevanju pa bomo predstavili še orodja za profiliranje (*ang. profiling tools*), s katerimi lahko identificiramo tiste dele v naši izvorni kodi, ki se izvajajo najpočasneje in so glede porabe pomnilnika in procesorja najbolj požrešni.

V 3. poglavju bomo določili kriterije, po katerih bomo ovrednotili predstavljena orodja iz 2. poglavja. Izbrali bomo tista orodja, ki ustrezajo prav našim postavljenim kriterijem in tako predlagali naš končni nabor testnih orodij za razvijalca javanskih aplikacij.

V zaključku bomo prikazali, kaj smo se pri spoznavanju z orodji naučili in predstavili naše ugotovitve.

2. Orodja za različna testiranja

2.1 Testiranje enot

Testiranje enot (*ang. unit testing*) je najnižji nivo testiranja programske kode. Uporablja se na najmanjših enotah v programskem jeziku. Za razvijalca programske kode je običajno enota metoda v programskem razredu. Za drugega razvijalca je enota lahko kar cel razred. Teste enot običajno pripravi razvijalec sam v času razvoja kode in si največkrat pomaga z nekim ogrodjem, ki že vsebuje podporo za pisanje testov enot. Za teste enot se pričakuje, da so izredno hitri v primerjavi z ostalimi vrstami testov programske kode [1]. Enote lahko testiramo z vhodnimi podatki, ki so že predhodno pripravljeni in z njimi preverjamo dobljene rezultate. S testiranjem enot dosežemo, da enote delujejo pravilno in samostojno ne glede na ostale dele aplikacije.

Lokacija testov testnih enot

Priporočljivo je, da se testi enot tvorijo v posebnem projektu ali pa vsaj v drugi mapi kot se nahajajo razredi oz. metode, ki jih želimo testirati (slika 1). Na ta način dosežemo, da se programska koda in koda testov med seboj ne pomešata. Nekateri zagovorniki testiranja menijo, da je treba testirati vsako vrstico kode, vendar zadošča, da spišemo teste za tiste dele kode, ki so kritični.

```
package helperPackage;

public class StringHelperClass {

    // invert upper case to lower or vice versa
    public String invertCase(String sText){

        char[] chars = sText.toCharArray();
        for (int i = 0; i < chars.length; i++){
            char c = chars[i];
            if (Character.isUpperCase(c))
                chars[i] = Character.toLowerCase(c);
            else if (Character.isLowerCase(c))
                chars[i] = Character.toUpperCase(c);
        }
        return new String(chars);
    }
}
```

Slika 1: Primer enostavne metode za katero bomo spisali test enote

Tipična orodja za testiranje enot

Najbolj razširjeni in uporabljeni brezplačni ogrodji za testiranje enot sta JUnit [2] in TestNG [3], zato ju bomo v nadaljevanju predstavili.

2.1.1 JUnit

JUnit je preprosto brezplačno ogrodje za pisanje ponovljivih testov enot v programskem jeziku Java [2]. Izhaja iz arhitekture xUnit, ki je skupno ime za različna ogrodja za testiranje enot v programskih jezikih. JUnit sta ustvarila razvijalca Kent Beck in Erich Gamma leta 1997 [4]. Predstavlja standard za testiranje enot in je eno izmed najbolj uporabljenih ogrodij v programskem jeziku Java glede na raziskavo, ki je potekala na projektih GitHub-a v letu 2013 [5].

```
package helperPackageTest;

import static org.junit.Assert.*;

public class StringHelperClassTest {

    private StringHelperClass stringHelperClass;

    @Before
    public void setUp() throws Exception {
        stringHelperClass = new StringHelperClass();
    }

    @Test
    public void testInvertCase() {
        String sText = "Povsod Je Lepo, A Doma Je Najlepše!";
        String sResultText = stringHelperClass.invertCase(sText);
        assertEquals("pOVSOD jE lEPO, a dOMA jE nAJLEPŠE!", sResultText);
    }

    @Ignore
    @Test(expected=NullPointerException.class, timeout=1)
    public void testInvertCaseIfNull(){
        String sText = "primer";
        String sResultText = stringHelperClass.invertCase(null);
        assertEquals("PRIMER", sResultText);
    }
}
```

Slika 2: Primer JUnit testnega razreda

Javanske oznake

Testne metode, ki jih ogrodje poganja, so označene z javanskimi oznakami (*ang. annotation*) [6]. Gre za oznake metapodatkov, na podlagi katerih ogrodje ve, katere akcije mora ob pogonu izvesti. Osnovno testno metodo v JUnit označimo z oznako `@Test`, s čimer tudi ogrodju nakažemo, da gre za testno metodo (slika 2). Ostale oznake so opcijske npr. `@BeforeClass`, `@AfterClass` (koda metode se bo izvedla pred prvo oz. za zadnjo testno metodo), `@Before`, `@After` (koda metode se bo izvedla pred vsako oz. za vsako testno metodo), `@Ignore` (koda metode se ne bo izvedla). Osnovni javanski oznaki

`@Test` lahko v parametru podajamo različne možnosti. Na primer `@Test(timeout=1)` pomeni, da se mora določena metoda izvesti v določenem časovnem intervalu (v tem primeru najkasneje v času 1 milisekunde), sicer bo rezultat testa metode neuspešen [7].

Omejitve metod

Za JUnit testne metode velja, da v osnovi ne morejo imeti parametrov niti vračati vrednosti. Če želimo nastavljati podatke v testnih metodah, si pomagamo s konstruktorji razreda. Vsaka testna metoda uporablja svojo instanco testnega razreda in to omogoča izolacijo oziroma ohranitev neodvisnosti posameznih testnih metod [1].

Trditve

S trditvami (*ang. assertion methods*) preverjamo pričakovane rezultate v testnih metodah. V najbolj osnovnem primeru na ta način preverimo, če so pričakovani rezultati enaki rezultatom, ki nam jih testna enota vrne. Obstaja več trditev, ki so vse statične in definirane v razredu `org.junit.Assert` [8].

Najbolj pogosto uporabne trditve so:

- `assertEquals(a,b)` - primerja, če sta dva objekta enaka
- `assertNotNull(o)` / `assertNull(o)` - preveri, če je objekt `null`
- `assertFalse(b)` / `assertTrue(b)` - preveri, če je vrednost `false` ali `true`
- `assertSame(a,b)` - primerja, če dve objekta pripadata istemu objektu z uporabo operatorja "==" npr. `(a == b)`
- `assertArrayEquals(a, b)` - primerja, če sta dve tabeli enaki
- `fail()` - test se bo v vsakem primeru izvršil neuspešno

Podatkovno usmerjeno testiranje in izjeme

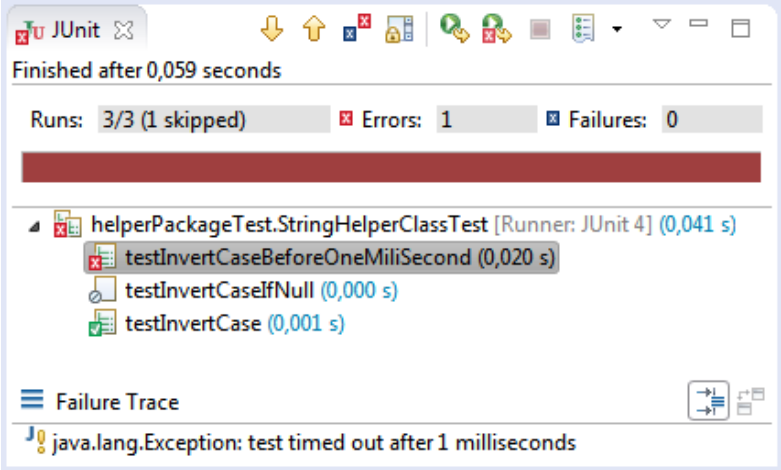
Pri podatkovno usmerjenem testiranju (*ang. data driven testing*) [9] uporabljamo parametrizirane teste (*ang. Parameterized test*), s katerimi lahko pripravimo vhodne podatke in ogrodje nam jih bo samodejno uporabilo nad želenimi testnimi enotami. Podatke za testiranje lahko pridobivamo tudi iz zunanjih virov. Pri podatkovno usmerjenem testiranju, s katerim izvajamo samodejno testiranje večje količine podatkov, moramo razrede označiti z oznako `@RunWith(Parameterized.class)` [10]. Namensko lahko testiramo klice metod v primeru, če se pripetijo pričakovane izjeme (*ang. Exceptions*). V tem primeru tako metodo označimo z oznako in parametrom `@Test(expected=NameOfTheException.class)` [11].

Uporaba ogrodja JUnit z razvojnimi okolji

JUnit je na voljo za uporabo kot vtičnik (*ang. plug in*) za najbolj uporabljena javanska razvojna okolja (npr. Eclipse, NetBeans, , IntelliJ IDEA, JDeveloper). V okolju Eclipse je že del standardnega namestitvenega paketa in ga je nadvse preprosto uporabljati [12].

Zaganjanje testov in prikaz rezultatov testiranja

Poženemo lahko več testov hkrati. Na primer, v razvojnem okolju Eclipse lahko z miško kliknemo na celoten paket in JUnit bo v njem izvedel vse testne metode v vseh testnih razredih. Nastavimo lahko tudi katere testne razrede bomo pognali skupaj in sicer na način, da jih združimo v tako imenovane testne zbirke (*ang. test suites*). Ko se izvedejo vsi testi, dobimo na koncu pregleden grafični prikaz uspešnosti testov, ki si jih lahko shranimo oz. izvozimo za kasnejše analize (slika 3).



Slika 3: Prikaz rezultatov JUnit testiranja v okolju Eclipse

2.1.2 TestNG

TestNG je brezplačno ogrodje za testiranje enot v programskem jeziku Java, ki je nastalo leta 2004 in dobilo navdih po ogrodju JUnit in NUnit [2]. Ustvaril ga je Cedric Beust. Ogrodje JUnit je pred verzijo 4 imelo kar nekaj omejitev in to je bil povod za nastanek ogrodja TestNG, ki je želelo takratne pomanjkljivosti odpraviti. Dodana je bila podpora za javanske oznake, podpora za uporabo parametrov pri testnih metodah, bolj fleksibilna nastavitev testiranja, različne oblike končnih rezultatov testiranja oz. poročil (*ang. reports*), podpora za testne metode, ki so lahko med seboj odvisne, podpora za večnitno testiranje … [3]. Poleg testiranja enot je v ogrodju možno testirati še ostale nivoje testiranja, kot so npr. funkcijsko, integracijsko in ostale.

```
package helperPackageTestNG;

import static org.testng.AssertJUnit.assertEquals;

public class StringHelperClassTestNG {

   private StringHelperClass stringHelperClass;

   @BeforeMethod
   public void beforeMethod() {
      stringHelperClass = new StringHelperClass();
   }

   @Test (timeOut = 1)
   public void invertCase() {
      String sText = "Povsod Je Lepo, A Doma Je Najlepše!";
      String sResultText = stringHelperClass.invertCase(sText);
      assertEquals("pOVSOD jE lEPO, a dOMA jE nAJLEPŠE!", sResultText);
   }

   @Test (expectedExceptions=NullPointerException.class)
   public void invertCaseIfNull() {
        String sText = "primer";
        String sResultText = stringHelperClass.invertCase(null);
        assertEquals("PRIMER", sResultText);
   }
}
```

Slika 4: Primer TestNG testnega razreda

Načini uporabe ogrodja

Ogrodje TestNG lahko uporabljamo na več načinov [13]:

- z uporabo datoteke `testng.xml`,
- z orodji Ant [14] ali Maven [15],
- preko ukazne vrstice,
- preko razvojnega okolja, kjer ogrodje namestimo preko vtičnika.

Struktura organizacije nivojev

Za lažjo predstavo strukture nivojev je na najvišjem nivoju testna zbirka, ki je lahko sestavljena iz več testov. Vsak test je lahko sestavljen iz več testnih razredov. Vsak testni razred ima svoje testne metode. Testne metode lahko združujemo v grupe in na podlagi tega lahko kasneje poganjamo teste za točno določena poimenovanja grup [16].

Konfiguracija preko javanskih oznak

Enako kot v JUnit tudi v TestNG uporabljamo javanske oznake, s katerimi ogrodju podamo metapodatke za pravilno izvajanje (slika 4). Osnovno metodo za testiranje označimo z `@Test`, poleg tega pa so med najbolj uporabljenimi oznakami naslednje [17]:

- `@BeforeSuite / @AfterSuite:` (metoda se izvede pred / za celotno suito testov)
- `@BeforeTest / @AfterTest:` (metoda se izvede pred / za celotnim testom, ki je označen z značko (*ang. tag*) `<test>`)
- `@BeforeClass / @AfterClass:` (metoda se izvede pred prvo / za zadnjo tesno metodo v trenutnem testnem razredu)
- `@BeforeGroups / @AfterGroups:` (metoda se izvede pred prvo / za zadnjo testno metodo v pred definirani grupi testnih metod)
- `@BeforeMethod / @AfterMethod:` (metoda se izvede pred / za vsako metodo v trenutnem testnem razredu)
- `@DataProvider:` metoda bo vsebovala podatke za podatkovno usmerjeno testiranje

Trditve

Trditve v TestNG npr. `assertEquals()` so praktično enake kot v ogrodju JUnit, le poimenovane so lahko malo drugače in vrstni red parametrov je lahko malo obrnjen. Testne metode lahko kličemo s parametri, zato ni potrebno polniti podatkov tesnih razredov (npr. pri podatkovno usmerjenem testiranju) preko konstruktorja razreda, kot je to potrebno pri ogrodju JUnit [16].

Nastavitvena datoteka

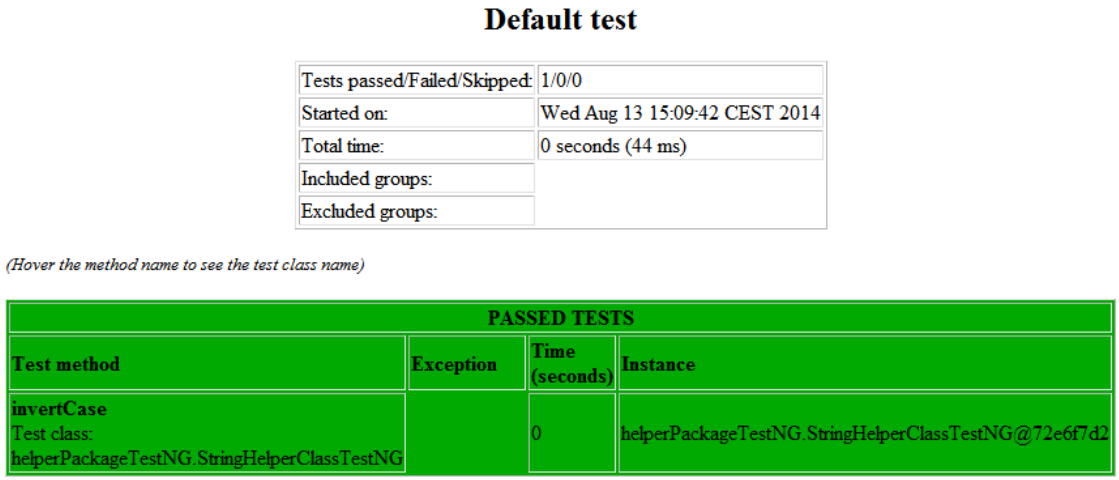
Za konfiguracijo testiranja se uporablja nastavitvena datoteka `testng.xml`, v kateri je zajeta celotna nastavitev testiranja. Na ta način enostavno definiramo testne grupe in ostale parametre, ki jih po potrebi nastavljamo [18].

Ponudnik podatkov

Ponudnik podatkov (*ang. data provider*) je metoda v testnem razredu, ki omogoča testiranje metode z večjim številom podatkov, ki so lahko tudi zunanjega izvora (datoteke, podatkovna baza…) [19]. Metodo ponudnika označimo z javansko oznako `@DataProvider(name = "imePonudnika")`, kjer za ime ponudnika izberemo ime, na katerega se bomo kasneje sklicevali pri metodah, ki bodo tega ponudnika uporabljale. Metodam, ki bodo ta ponudnik podatkov uporabljale, v javanski oznaki `@Test` preko parametra napovemo ime ponudnika npr. `@Test(dataProvider = "imePonudnika")`.

Uporaba z razvojnim okoljem Eclipse in prikaz rezultatov

TestNG ogrodje se vključi preprosto preko vtičnika v razvojno okolje Eclipse [30]. Uporablja se ga podobno kot ogrodje JUnit, le da imamo tu več možnosti. Po končanem testiranju lahko pregledujemo različne formate zaključnih poročil (npr. `html`, `xml`, tekstovna sporočila…), ki nam jih ogrodje naredi že v privzetem načinu (slika 5). Lahko pa definiramo tudi svoj lasten format končnega poročila.



Default test

Tests passed/Failed/Skipped:	1/0/0
Started on:	Wed Aug 13 15:09:42 CEST 2014
Total time:	0 seconds (44 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
invertCase Test class: helperPackageTestNG.StringHelperClassTestNG		0	helperPackageTestNG.StringHelperClassTestNG@72e6f7d2

Slika 5: Prikaz zaključnega poročila testiranja v ogrodju TestNG

2.2 Testiranje in analiza sintakse

V razvojnih okoljih (npr. Eclipse, NetBeans, IntelliJ IDEA, JDeveloper…) nam pri pisanju programov pomaga že sam prevajalnik (*ang. compiler*). Tako lahko sproti popravljamo napake, ki nastajajo med pisanjem kode. V naši kodi pa lahko naredimo napake takega tipa, ki so sicer sintaktično pravilne, vendar se bodo kljub temu lahko kasneje izrazile kot programske napake. Če želimo v kodi odkrivati napake takega tipa, lahko posežemo po specifičnih orodjih, ki so namenjena prav temu.

Statična in dinamična analiza kode

Statična analiza kode (*ang. static code analysis*) je programska verifikacija kode, ki analizira izvorno kodo glede na njeno kakovost in zanesljivost [20]. Gre za analizo izvorne programske kode, ki se jo izvede statično brez izvajanja izvorne kode. Zato se uporabljajo posebna orodja. Razvijalec na podlagi rezultatov analize oceni, kako resne so odkrite napake ter jih po potrebi odpravi.

Obstajajo tudi orodja, ki delujejo na principu poganjanja izvorne kode. Koda se v tem primeru izvede na pravem ali virtualnem procesorju. Orodja tega tipa uporabljajo tako imenovano dinamično analizo kode (*ang. dynamic code analysis*) [21].

Tipična orodja za testiranje in analizo sintakse

V nadaljevanju bomo opisali odprtokodna (*ang. open source*) orodja FindBugs [22], PMD [23] in CheckStyle [24], saj so izredno priljubljena med razvijalci zaradi preprostosti uporabe in integracije z različnimi drugimi orodji. Opisali bomo tudi platformo SonarQube [25], ki v zadnjem času postaja vse bolj priljubljena med razvijalci in še posebej pri tehničnih vodjih, saj vsebuje pregleden grafični vmesnik, ki omogoča visokonivojski pregled na izvorno kodo.

2.2.1 FindBugs

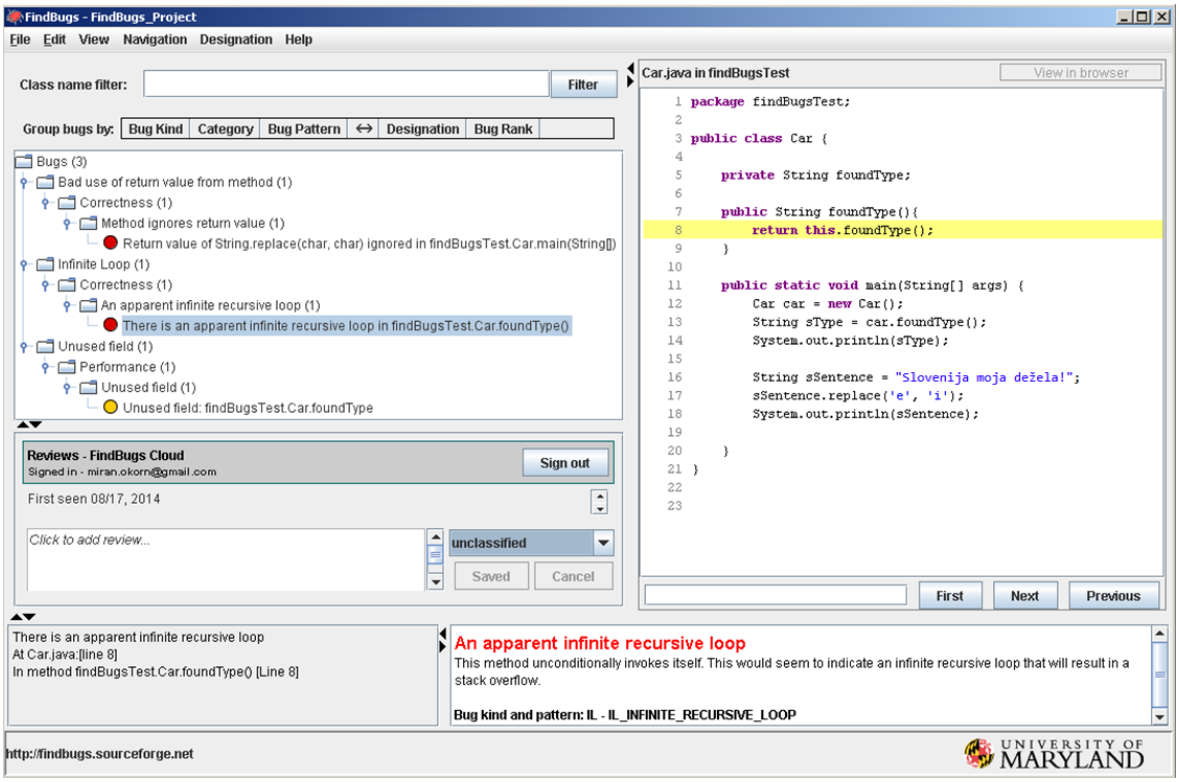
FindBugs je odprtokodno orodje, ki na podlagi statične analize poišče napake v javanski kodi [22]. Orodje ni namenjeno za preverjanje stila pisanja kode, pač pa išče prave in potencialne performančne napake. Ustvarila sta ga Bill Pugh in David Hovemeyer. Deluje na že prevedeni (*ang. compiled*) bitni javanski kodi (prevedenih razredih tipa `class` ali `jar` datotekah) na način, da v kodi poišče potencialne nevarnosti z iskanjem vzorcev znanih napak [26]. V prevedeni kodi orodje odkrije napake, ki so sicer sintaktično pravilne, zato jih tudi prevajalnik ne zazna, v resnici pa lahko kljub sintaktični pravilnosti povzročijo znane potencialne napake [27].

Štiri skupine odkritih napak

Odkrite potencialne napake nam orodje razvrsti v štiri skupine: najhujše (*ang. scariest*), hude (*ang. scary*), zaskrbljujoče (*ang. troubling*) in manj zaskrbljujoče (*ang. of concern*). Na podlagi razvrščenih napak razvijalec sam oceni oziroma analizira resnost odkritih napak in jih lahko tudi sam pravilneje razvrsti.

Načini uporabe orodja

FindBugs se lahko uporablja kot samostojna javanska aplikacija (slika 6) z uporabniškim vmesnikom, lahko se ga uvozi tudi kot vtičnik za najbolj pogosto uporabljena javanska razvojna okolja, kot so Eclipse, NetBeans, IntelliJ IDEA [22]... Poganja se ga lahko iz ukazne vrstice. Orodje lahko uporabljamo z orodji za samodejna grajenja projektov (*ang. automatic build*) kot sta npr. Ant in Maven [26]. Kot izhodni format obdelave za omenjena orodja si lahko nastavimo izvozni format `xml` in shranjene rezultate kasneje uvažamo ter analiziramo v samostojni aplikaciji FindBugs.



Slika 6: Primer uporabe orodja FindBugs kot samostojne aplikacije

Kot rezultat analize nam samostojna aplikacija FindBugs lahko naredi poročilo v obliki `xml` ali kakega drugega tekstovnega tipa. Poročila si lahko shranimo in primerjamo zgodovino odkritih napak.

Primeri odkritih napak z orodjem FindBugs

```
String sSentence = "Povsod je lepo, a doma je najlepše!";
sSentence.replace('e', 'i');
System.out.println(sSentence);
```

Slika 7: Primer prezrte vrnjene vrednosti

V zgornjem primeru (slika 7) je razvijalec želel spremeniti vse pojavitve znaka `'e'` v znak `'i'` nad `String` objektom `sSentence`, vendar se ta v resnici ni prav nič spremenil. Objektov tipa `String` namreč ni možno spreminjati, saj so nespremenljivi (*ang. immutable objects*) [28]. Metoda je sicer vrnila nov objekt, ki pa se ni priredil nikamor. Pravilen popravek kode bi bil (slika 8):

```
String sSentence = "Povsod je lepo, a doma je najlepše!";
sSentence = sSentence.replace('e', 'i');
System.out.println(sSentence);
```

Slika 8: Primer popravljene prezrte vrnjene vrednosti

S tem popravkom je objekt `sSentence` postal povsem nov objekt z novo referenco in vrednostjo.

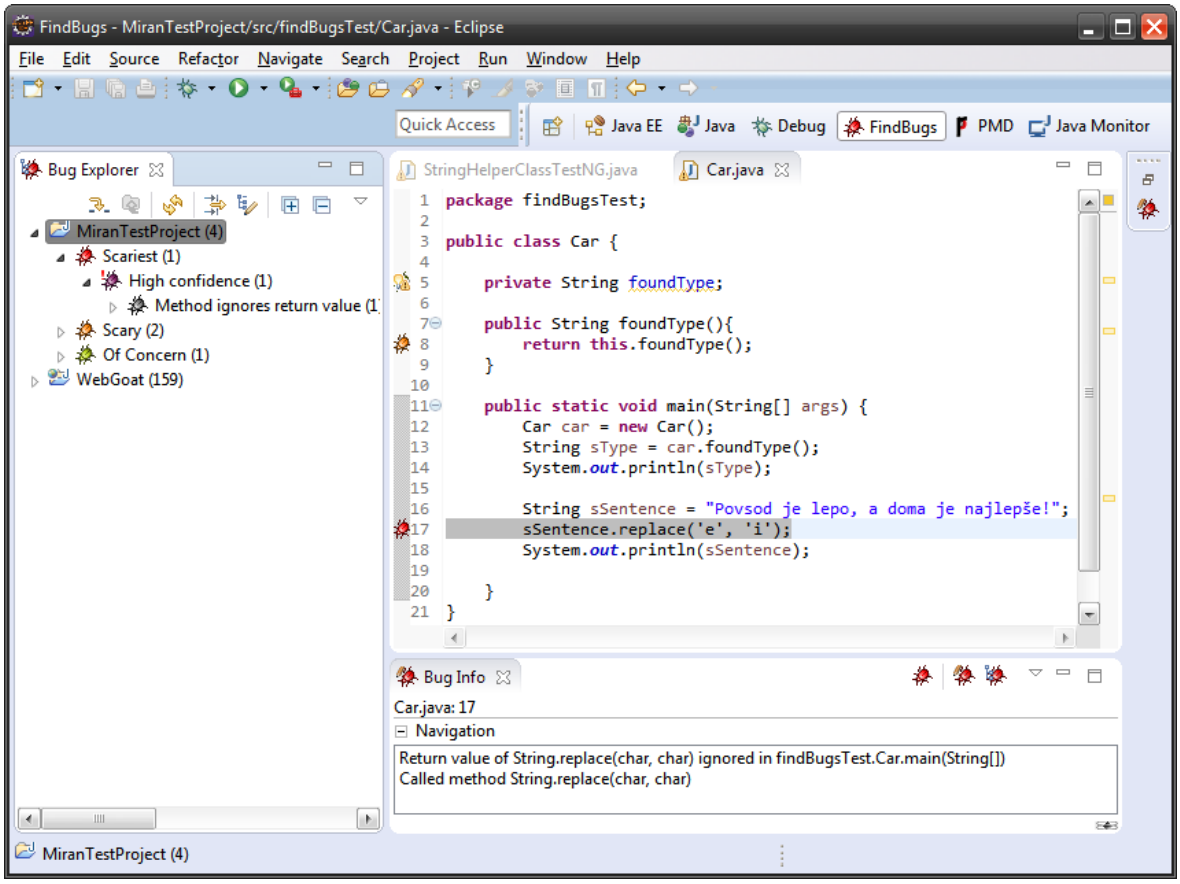
```
public class Car {

    private String foundType;

    public String foundType(){
        return this.foundType();
    }

    public static void main(String[] args) {
        Car car = new Car();
        String sType = car.foundType();
        System.out.println(sType);
    }
}
```

Slika 9: Primer najdene neskončne rekurzivne zanke v JDK

(Slika 9) prikazuje primer zanimive odkrite napake, ki jo je našel FindBugs, ko so ga pognali nad izvorno kodo `jdk`. Napako je po nesreči "zakuhal" Joshua Bloch, ki je znan po tem, da je zasnoval ogromno izvornih javanskih metod (npr. `Java Collections`) in je eden izmed vodilnih arhitektov pri podjetju Google [31]. V gornjem primeru zagon programa v razvojnem okolju vrne napako tipa "`StackOverflowError`". Metoda `foundType()` namreč v neskončnost kliče samo sebe. V resnici je želel avtor z metodo `foundType()` vrniti le vrednost spremenljivke `foundType` [29].

Uporaba Findbugs preko vtičnika z razvojnim okoljem Eclipse

Za uporabo FindBugs verzije 3.0 moramo predhodno namestiti Javo (`jre`) vsaj verzije 1.7 [22]. Če želimo orodje uporabljati v razvojnem okolju Eclipse (slika 10), namestitev opravimo preko uvoza vtičnika in FindBugs poženemo nad projektom, katerega želimo pregledati [32].



Slika 10: Uporaba orodja FindBugs v razvojnem okolju Eclipse

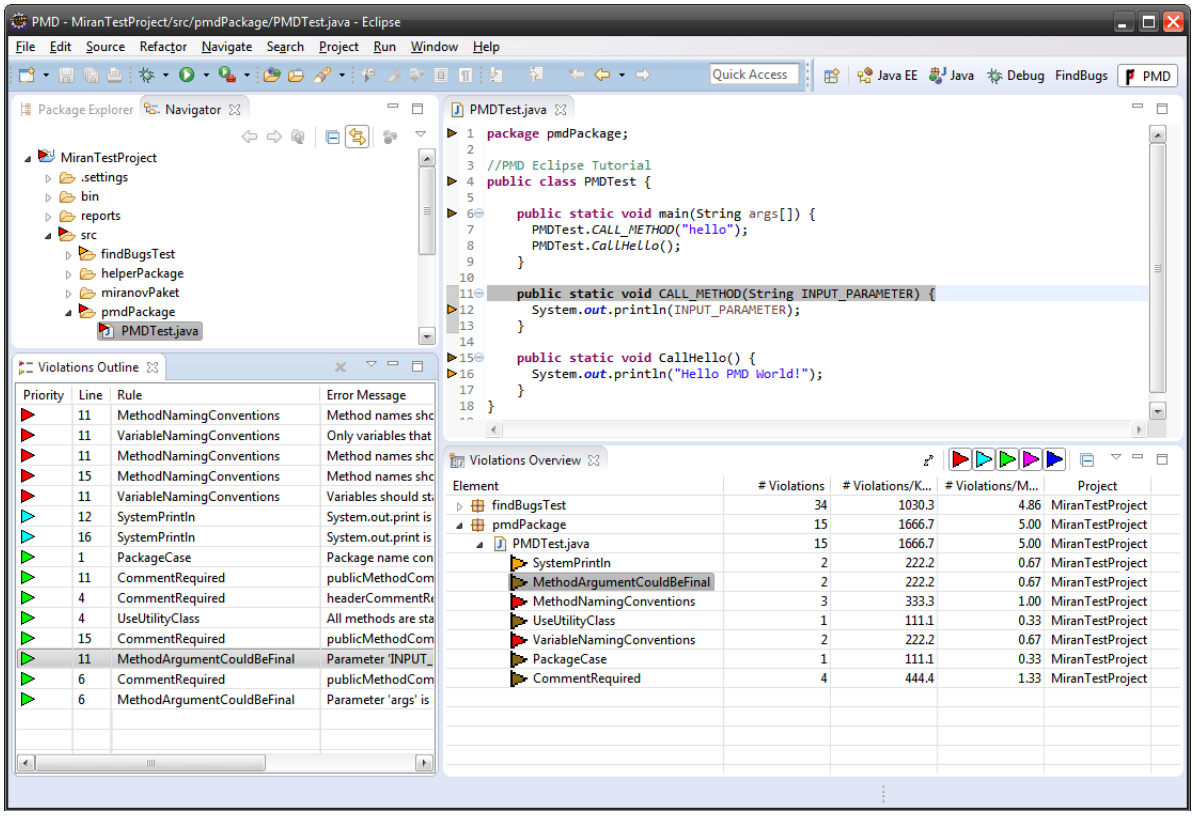
Poročila in analize

Poročila lahko shranjujemo lokalno v obliki datoteke `xml`. Orodje FindBugs nam ponuja tudi shranjevanje analiz v oblak (*ang. cloud*). Na podlagi rezultatov nato razvojni delavci orodja FindBugs izvajajo svoje analize za izboljšavo orodja (npr. razvrstitev resnosti odkritih napak in zmanjševanje napak tipa *false-positive*) [33]. Poleg tega si lahko v oblaku shranjujemo zgodovino odkritih napak.

2.2.2 PMD

PMD je brezplačno odprtokodno orodje, ki analizira izvorno kodo (in ne binarne kode) v kateri poišče običajne programske napake kot so npr. neuporabljene spremenljivke, prazne dele kode, mrtvo kodo, nepotrebne inicializacije objektov, nepotrebno zapletenost kode, podvojeno kodo, nepravilno pisanje določenih metod. [23][34] Uporablja se v programskem jeziku Java, JavaScript, `xml` in `xsl`.

Napake, ki jih orodje odkrije, običajno niso prave napake (včasih lahko tudi so), pač pa le pokazatelj, kje lahko razvijalec kodo izboljša in jo izpopolni.



Slika 11: Primer uporabe orodja PMD v razvojnem okolju Eclipse

Uporaba orodja

Orodje lahko uporabljamo preko ukazne vrstice, z orodji Ant in Maven in z uporabo preko vtičnika v razvojnih orodjih (npr. Eclipse).

Množice pravil

PMD vsebuje vgrajen seznam pravil, po katerih orodje išče napake [35]. Ta pravila lahko definiramo tudi sami in sicer z uporabo jezika `XPath` [36] ali z uporabo javanskih razredov.

Način delovanja

PMD ne uporablja neposredno izvorne kode programov, nad katerimi smo orodje pognali, pač pa uporablja razčlenjevalnik *(ang. parser)* `JavaCC` [38], s katerim razčleni izvorno kodo ter naredi abstraktno sintaksno drevo (AST) (*ang. abstract syntax tree*) [37]. Na podlagi dobljenega AST drevesa orodje najde in razvrsti vrste najdenih napak.

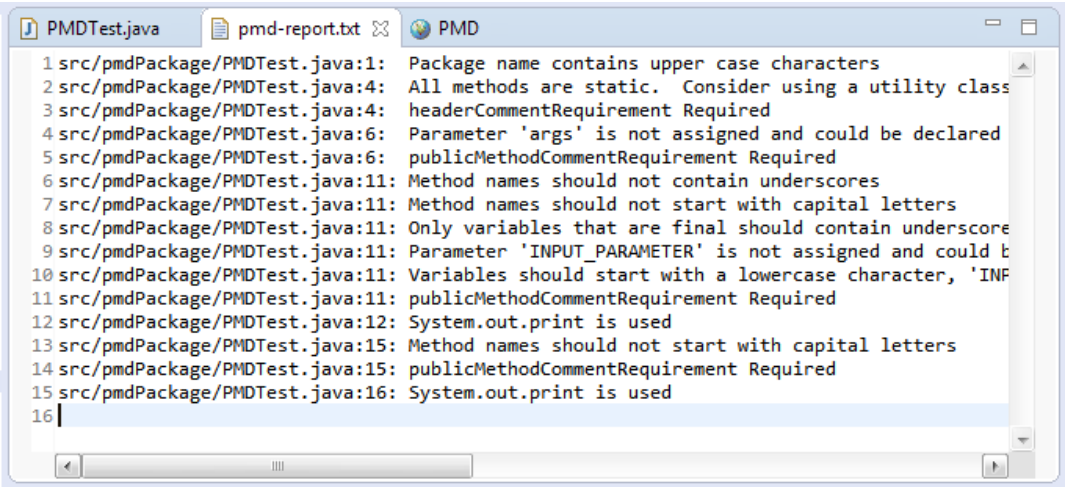
Uporaba z razvojnim okoljem Eclipse:

Orodje vključimo preko vtičnika in nad celotnim projektom, paketom ali posameznem razredom poženemo PMD (slika 11). Najdene napake lahko enostavno filtriramo glede na izbiro prikazanih tipov napak z gumbi:

.

Generiranja poročil

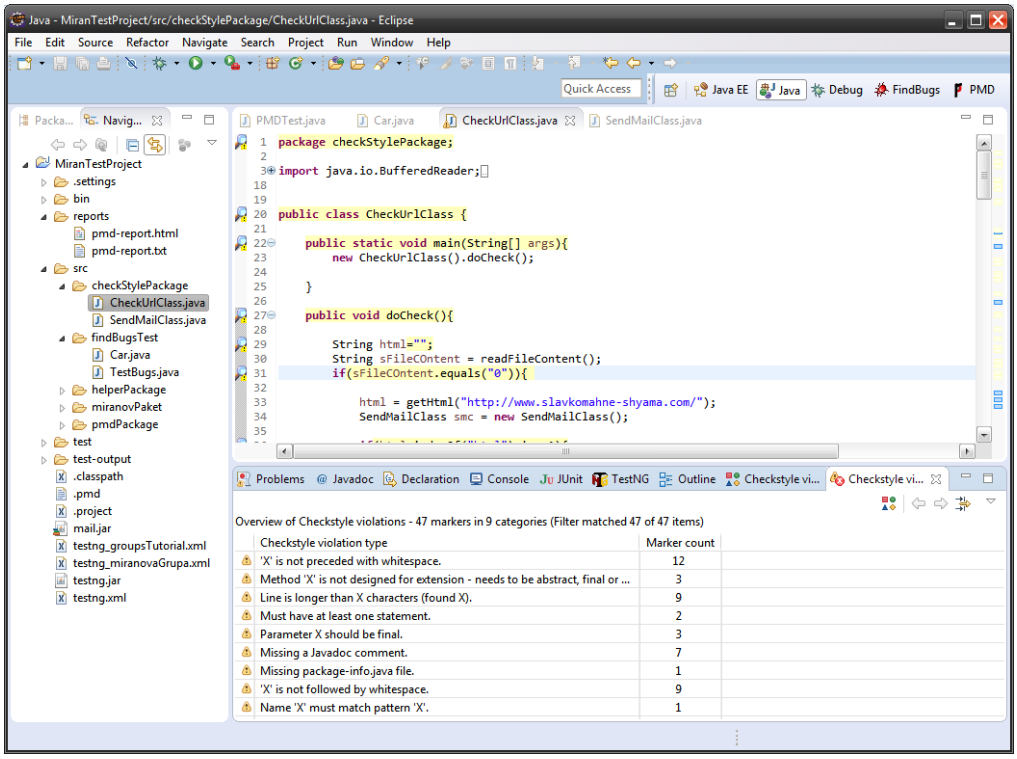
S PMD lahko ustvarimo poročila v različnih formatih npr. `xml, txt, html`... (slika 12). V okolju Eclipse formate poročil le označimo v nastavitvah vtičnika PMD.



Slika 12: Primer poročila napak z orodjem PMD v razvojnem okolju Eclipse

2.2.3 Checkstyle

Checkstyle je brezplačno odprtokodno razvojno orodje, ki uporablja statično analizo pri analizi izvorne kode v Javi [24]. Razvijalcem pomaga pri pisanju kode na način, da ta ustreza standardom oziroma pravilom pisanja kode. Razvil ga je Oliver Burn leta 2001, trenutno pa ga vzdržuje ter posodablja več razvijalcev po svetu. Orodje avtomatizira proces preverjanja stila javanske kode in s tem razbremeni razvijalca tega dolgočasnega opravila. Orodje se izkaže še posebej koristno pri projektih, kjer morajo razvijalci upoštevati standard kodiranja.



Slika 13: Uporaba orodja Checkstyle v razvojnem okolju Eclipse

Funkcionalna uporaba orodja

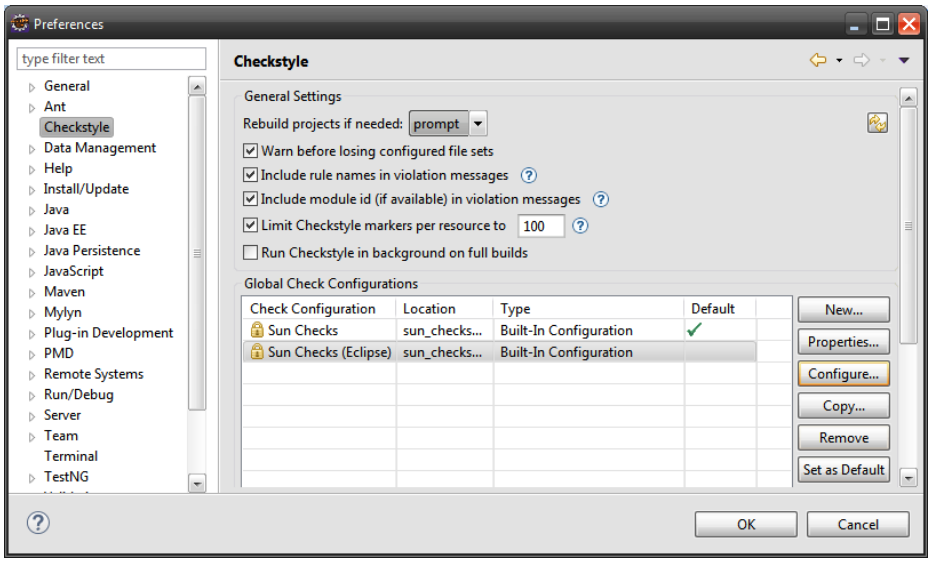
Checkstyle je zelo nastavljiv in ga lahko nastavimo, da podpira skoraj katerikoli standard kodiranja. Nastavljamo lahko tudi, kako občutljivo je orodje na zaznavanje odkritih napak ter po potrebi to nastavitev prilagajamo. V začetku se ga je funkcionalno uporabljalo le za pregledovanje stila kode. Od verzije 3 naprej pa so bile dodane dodatne možnosti pregledovanja kode kot so: dobra praksa pri zasnovi razredov, podvojena koda, podvojena zaklepanja v kodi. Oceni nam lahko komentarje tipa `javadoc`, poimenovanje atributov in metod, oceni število podanih parametrov v metodah, poišče nepotrebne presledke v kodi, večkratno merjenje kompleksnosti kode in podobno.

Načini uporabe orodja

Orodje je najbolj koristno ravno pri vključevanju v procese avtomatskih grajenj aplikacij kot npr. opravilo za orodje Ant. Checkstyle lahko poženemo v javanskem navideznem stroju (*ang. java virtual machine*) kot datoteko tipa `jar`. Lahko se ga uporablja tudi preko ukazne vrstice. Prav tako je na voljo kot vtičnik za razvojna okolja Eclipse (slika 13), Netbeans, IntelliJ IDEA [39]…

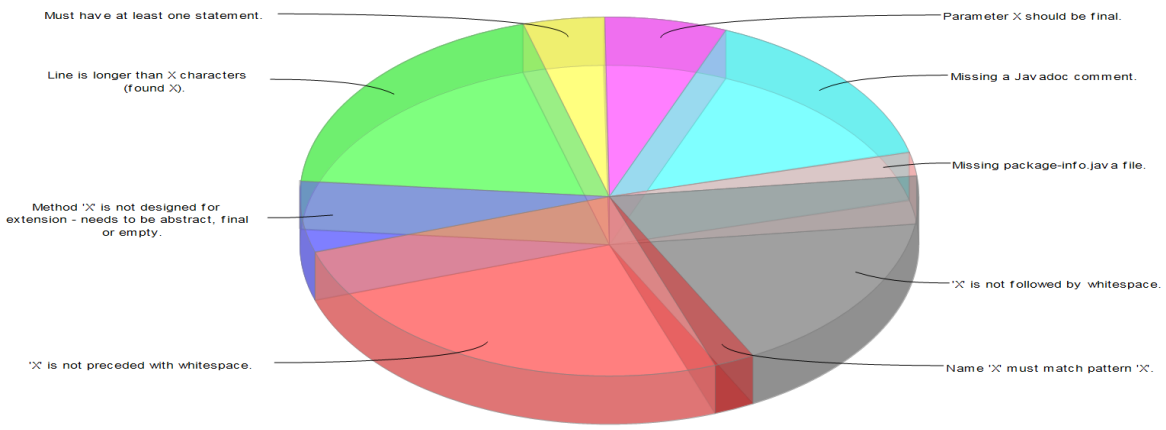
Uporaba orodja Checkstyle v razvojnem okolju Eclipse

Orodje Checkstyle namestimo preko vtičnika. Nastavimo ga na želeni nivo pregledovanja (slika 14) ter lahko dodajamo svoje standarde kodiranja [40].



Slika 14: Nastavitve orodja CheckStyle v razvojnem okolju Eclipse

Posamezna pravila pregledovanje kode lahko vklopimo in izklopimo. Pred tem je priporočljivo narediti kopijo obstoječega profila v nastavitvah orodja in ga po spremembi nastaviti tudi za privzetega. Checkstyle lahko vklopimo ali izklopimo za posamezne projekte, v katerih orodje pregleda izvorne javanske datoteke in označi vrstice, kjer je odkril napake. Vtičnik nam predstavi tudi grafično predstavitev odkritih napak (slika 15).



Slika 15: Grafična predstavitev najdenih napak z orodjem Checkstyle v okolju Eclipse

Uporaba Checkstyle pravil za razvojne ekipe

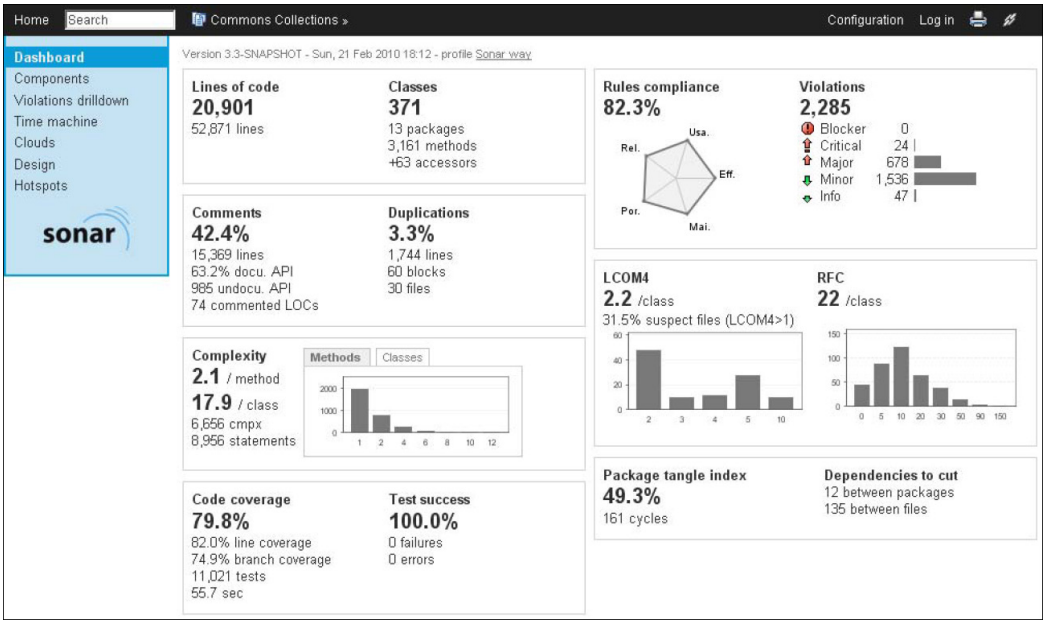
Priporočljivo je, da celotna razvojna ekipa, ki dela na nekem skupnem projektu, uporablja skupna pravila pisanja kode. Vtičnik CheckStyle za Eclipse omogoča, da datoteko s pravili odložimo na neko skupno spletno odložišče, s katerim se poveže celotna razvojna ekipa in tako uporablja pravila [40].

Način delovanja orodja Checkstyle in pisanje lastnih pravil

Checkstyle uporablja razčlenjevalnik `ANTLR` [41], ki poskrbi za to, da prebere izvorno javansko kodo ter jo pretvori v ustrezno obliko za generiranje AST drevesne strukture prebrane datoteke. Če želimo, lahko sami v Javi spišemo nova pravila in jih tako uporabljamo v svojih projektih [42].

2.2.4 SonarQube

SonarQube je odprtokodna visoko nivojska platforma napisana v Javi za upravljanje kakovosti izvorne kode [25]. Orodje je bilo zasnovano z namenom, da bi kar v najkrajšem času lahko odpravili napake, ki so v kodi prisotne. Platforma vsebuje orodja za analizo kode, orodja za kreiranje poročil, module za odkrivanje napak v kodiranju, orodja za pregledovanje in analiziranje zgodovine odkritih napak. Ponuja mehanizem za dodajanje vtičnikov, ki omogoča, da se funkcionalnost orodja še dodatno razširi. Pri tem velja poudariti, da se za platformo razvijajo tako brezplačni kot tudi plačljivi vtičniki.



Slika 16: Primer izgleda spletnega vmesnika za platformo SonarQube

Podprti jeziki

S SonarQube lahko analiziramo kodo v več kot 25 programskih jezikih: Java, JavaScript, C/C++, C#, PHP, Python… Za uporabo platforme za točno določen programski jezik si namestimo za to namenjen vtičnik (npr. vtičnik za Javo za SonarQube).

Način uporabe orodja

SonarQube lahko uporabljamo tudi preko ukazne vrstice. Omogoča povsem avtomatične analize in se integrira z različnimi orodji za avtomatsko grajenje projektov (npr. Maven, Ant, Gradle [43]), kot tudi z orodji za neprekinjeno integracijo (npr. Atlassian Bamboo [44]). Platforma se integrira tudi z zunanjimi orodji, kot je npr. Jira [45]. V razvojnem okolju Eclipse lahko platformo uporabljamo preko za to namenjenega vtičnika.

Podpora za sedem aspektov kakovosti kodiranja

- arhitektura in načrt,
- podvajanje kode,
- testiranje enot,
- zahtevnost kode,
- potencialne napake,
- pravila kodiranja,
- komentarji v kodi.

Arhitektura SonarQube

SonarQube arhitektura je zasnovana fleksibilno in vsebuje več komponent [46].

- Skupek različnih orodij za analizo kode, ki se lahko po želji vključujejo in nastavljajo.
- Podatkovna baza za shranjevanje rezultatov analiz, shranjevanje nastavitev projektov in globalnih nastavitev ter shranjevanje zgodovine analiz različnih orodij, ki so del platforme. Lahko se uporablja na različnih podatkovnih bazah (MySql, Oracle, PostgreSQL, Microsoft SQL Server…)
- Spletni uporabniški vmesnik (slika 16) v katerem lahko pregledujemo rezultate analiz projektov iz visokega nivoja in se pomikamo na nižje nivoje, pregledujemo grafični prikaz zgodovine odkritih napak, iščemo napake v kodi in v profilih nastavljamo konfiguracije globalnih in posameznih orodij, ki jih SonarQube vsebuje.

Skupek najboljših javanskih orodij za analizo kode

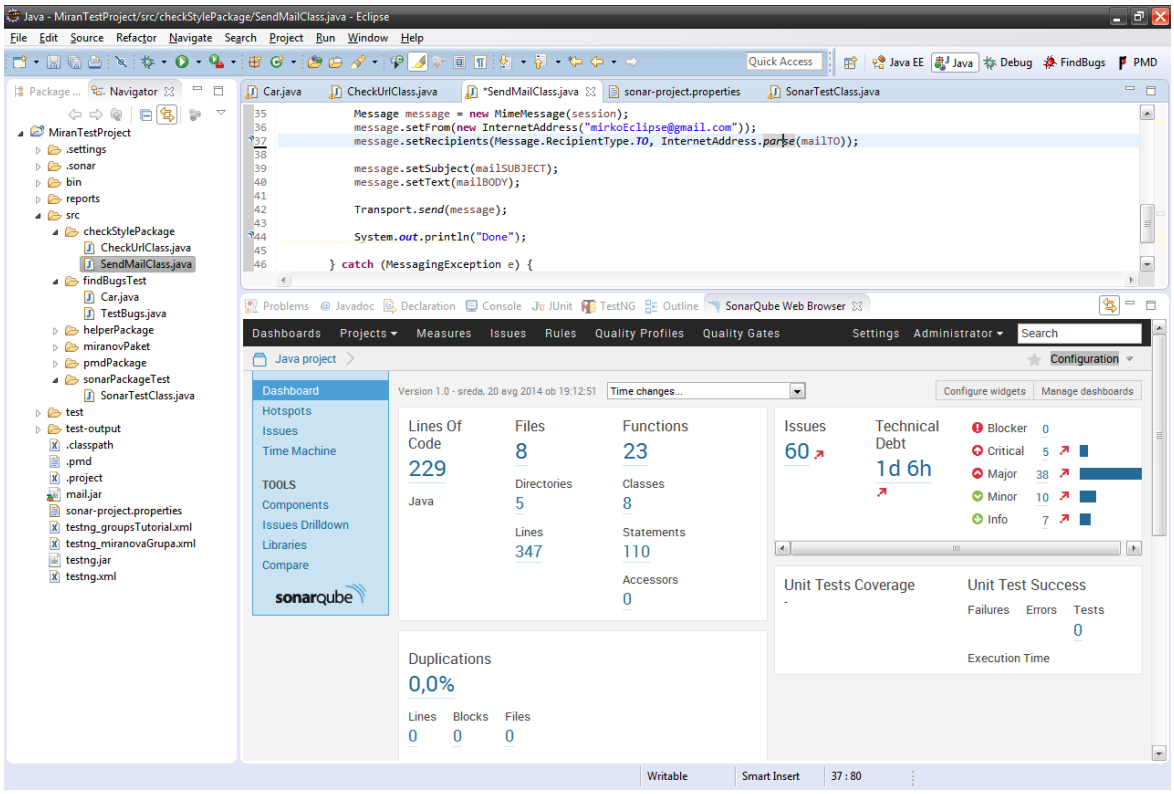
SonarQube vtičnik za programski jezik Java vsebuje (ali pa jih po potrebi namestimo) najboljša orodja za analizo kode kot so orodja za analizo pravilnega kodiranja (PMD, CheckStyle…), orodja za odkrivanje napak (FindBugs), orodja za merjenje pokritosti kode s testi za testiranje enot (Jacoco [47], Cobertura [48], Clover [49]…). SonarQube vtičnik za Javo vsebuje in se zanj razvijajo tudi lastna orodja za odkrivanje napak [46].

Umik orodij PMD in CheckStyle iz vtičnika za Javo za SonarQube

V starejših verzijah vtičnika za Javo je SonarQube med drugimi orodji vseboval tudi orodji PMD in CheckStyle. Od vtičnika verzije 2.0 naprej pa so omenjeni orodji razvijalci SonarQube umaknili [50], saj so pravila orodij prepisali in vključili v svoje interno javansko orodje `SSLR` (*ang. Sonar Source Language Recognizer*) [51]. S svojim internim orodjem naj bi lažje upravljali s pravili in jih tako tudi neodvisno od drugih ponudnikov sproti posodabljali. PMD in CheckStyle lahko še vedno dodatno namestimo [52].

Uporaba orodja z razvojnim okoljem Eclipse

V okolje Eclipse lahko namestimo vtičnik za platformo SonarQube [53]. Predhodno moramo komponente orodja namestiti tudi na svoj lokalni računalnik (podatkovno bazo, strežnik in zaganjalnik (*ang. runner*)), v kolikor želimo imeti komponente postavljene lokalno pri sebi [54]. Prvič moramo analizo nad svojim projektom pognati s SonarQube zaganjalnikom. Zatem pa lahko v okolju Eclipse nad svojim projektom poganjamo analize kar preko vtičnika za SonarQube (slika 17).



Slika 17: Uporaba SonarQube v okolju Eclipse

Zgodovina matrik

Platforma si shranjuje zgodovino matrik v podatkovni bazi, ki jih lahko pregledujemo preko grafičnega prikaza v spletnem uporabniškem vmesniku z notranjim orodjem *TimeMashine*. Sproti lahko spremljamo trend (ali se število napak v našem projektu povečuje ali zmanjšuje).

SourceMeter za SonarQube

Preko vtičnikov lahko za SonarQube vključujemo še dodatna orodja. Na primer brezplačno orodje SourceMeter [55] vsebuje še dodatna orodja za preverjanje kode. Eno izmed teh orodij je sicer tudi plačljivo orodje FaultHunter [56], ki naj bi odkrivalo napake še bolj pravilno in natančno. FaultHunter naj bi namesto abstraktnega sintaksnega drevesa (kar uporabljata za primer PMD in CheckStyle) uporabljal abstraktni semantični graf (*ang. abstract semantic graph*) [57], ki omogoča globljo analizo kode in naj bi na ta način še povečal število najdenih pravih napak ter zmanjšal prikaz zadetkov, ki v resnici niso napake.

2.3 Testiranje varnosti v kodi spletnih aplikacij

Če razvijamo spletne aplikacije, je varnostni vidik zelo pomemben. Veliko spletnih aplikacij lahko vsebuje varnostna tveganja, ki pogojujejo, da lahko neka tretja oseba naredi spletni aplikaciji ali uporabnikom škodo, bodisi s tem, da nekako dostopa do uporabniških podatkov, bodisi s tem, da uporabnikom prestreže vtipkana gesla in lahko na različne načine oškoduje uporabnike.

Za razvijalce spletnih strani je nujno potrebno spoznati in razumeti kritične ranljivosti, ki so lahko tarča za vdor in zlorabo spletnih aplikacij. Potrebno je spoznati znane varnostne napake v spletnih aplikacijah in se naučiti pisati varno kodo na način, da ta ustreza varnostnem dokumentu OWASP Top 10 [58] svetovne neprofitne dobrodelne organizacije OWASP (*ang. Open Web Application Security Project*). Organizacija se ukvarja s tem, da ozavešča svet, katere varnostna tveganja se pojavljajo v spletnih aplikacijah in daje napotke, kako jih odpraviti [59].

OWASP Top 10

Dokument OWASP Top 10 prikazuje deset najbolj kritičnih varnostnih tveganj, ki obstajajo v spletnih aplikacijah. Sestavili so ga različni varnostni strokovnjaki iz raznih delov sveta. Namen dokumenta je, da bi posamezniki ali podjetja, ki razvijajo spletne aplikacije, pisala varno kodo in na ta način zagotovila varno uporabo aplikacij.

Slika 18: OWASP deset najbolj kritičnih spletnih varnostnih tveganj v letu 2013

Za vsako tveganje dokument OWASP Top 10 prikaže:

- opis tveganja,
- primer kode, ki vsebuje omenjeno varnostno tveganje,
- primer napadov, ki se lahko izvršijo,
- smernice, kako se lahko tveganju izognemo,
- sklicevanja na druge OWASP vire, za obsežno pomoč in odpravo varnostnih tveganj.

V trenutno zadnjem OWASP Top 10 dokumentu 2013 najdemo opis desetih najbolj kritičnih znanih varnostnih tveganj (slika 18) [60][61].

Orodja za odkrivanje varnostnih tveganj v kodi

Obstajajo brezplačna orodja, ki lahko analizirajo kodo spletne aplikacije in v njej poiščejo, če obstajajo znana varnostna tveganja. Nekatera lahko uporabljamo kar v razvojnem okolju Eclipse (sploh če uporabljamo Eclipse IDE za J2EE) kot npr. Find Security Bugs [62], Google CodePro Analytix [63] ali LAPSE+ [64]. Ta orodja niso mišljena kot celovita rešitev pri analiziranju varnosti, pač pa so namenjena kot pripomoček, da v kodi identificiramo tiste dele, ki lahko predstavljajo varnostna tveganja. V kolikor iščemo celovite programske rešitve za iskanje in odpravo varnostnih tveganj v spletnih aplikacijah, lahko posežemo po sicer plačljivih orodjih, ki jih uporabljajo številna podjetja za odkrivanje napak v svojih aplikacijah. Nekaj primerov takih zelo dobrih orodij je: Coverity [65], Security AppScan (IBM) [66] in Fortify (HP) [67].

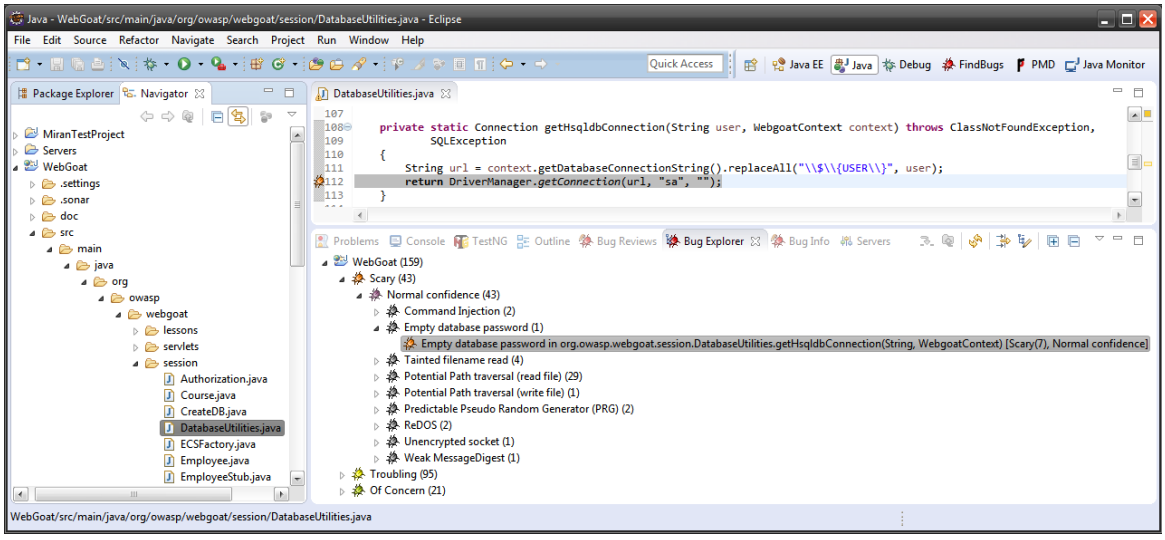
Tipična orodja za testiranje varnosti

V nadaljevanju bomo opisali brezplačno orodje oziroma vtičnik za orodje FindBugs, ki se imenuje Find Security Bugs [62]. Trenutno je to verjetno edino res obetavno brezplačno orodje, ki je sposobno odkriti varnostna tveganja v spletnih javanskih aplikacijah [68]. Za primerjavo bomo vzeli sicer plačljivi (oz. brezplačni le za določeno časovno obdobje) vtičnik OWASP (Excentia) [69] za platformo SonarQube. Vzeli smo ga zaradi že prej opisane platforme SonarQube, v katero ga je zelo enostavno vključiti ter glede na dejstvo, da drugih brezplačnih dobrih orodij enostavno ni bilo moč najti. Za SonarQube se sicer že razvija odprtokodni vtičnik OWASP SonarQube Project [70], ki bo namenjen prav odkrivanju napak, ki ga opisuje dokument OWASP Top 10, vendar v času pisanja diplomske naloge še ni bil dosegljiv za preizkušanje.

Orodja za odkrivanje napak lahko poženemo nad kodo svoje spletne aplikacije, lahko pa si v izobraževalne namene iz spletne strani OWASP prenesemo javanski spletni projekt WebGoat [71], ki je bil razvit prav z namenom, da vsebuje znana varnostna tveganja, na katerih se lahko učimo in orodja preizkušamo.

2.3.1 Find Security Bugs

Find Security Bugs je vtičnik za že znano orodje FindBugs, ki ga lahko prav namensko uporabimo za pregled varnostnih tveganj v kodi javanskih spletnih aplikacij [62]. Odkrije tudi varnostna tveganja, ki jih opisuje dokument OWASP Top 10.



Slika 19: Uporaba Find Security Bugs v razvojnem okolju Eclipse

Uporaba vtičnika

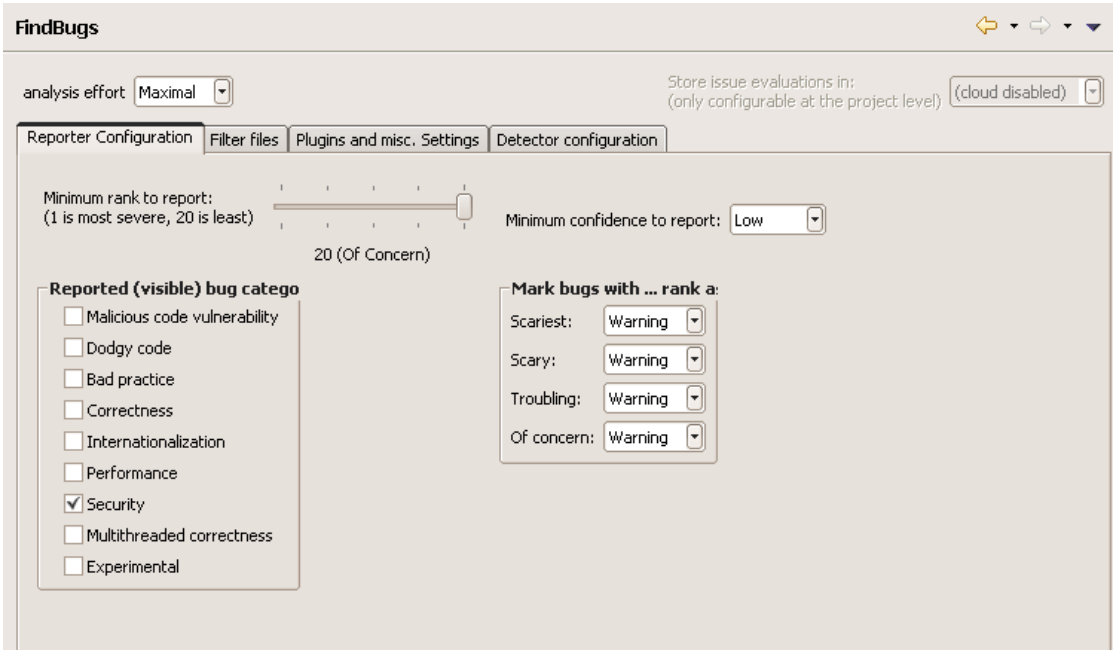
Vtičnik se lahko vključi v različna razvojna okolja. Lahko se ga uporablja v okolju Eclipse (slika 19), IntelliJ, Netbeans preko vtičnika za orodje FindBugs. Integrira se ga lahko tudi z orodji za neprekinjeno integracijo (*ang. continuous integration*) kot je npr. Jenkins in SonarQube, za katera obstaja prav v ta namen razviti vtičnik.

Uporaba orodja v okolju Eclipse

Če je FindBugs že vključen v razvojno okolje Eclipse, moramo presneti iz spleta le še vtičnik Find Security Bugs. Shraniti si ga moramo na poljubno lokacijo in vtičnik vključiti v nastavitvah orodja FindBugs ter nastaviti oziroma nastavitve omejiti, da želimo poiskati le varnostne napake (slika 20). Po ponovnem zagonu okolja Eclipse nad spletnim projektom poženemo analizo z orodjem FindBugs in tokrat bomo dobili zadetke o najdenih varnostnih tveganjih.

Najdene napake

Napake, ki jih vtičnik zazna, niso nujno prava varnostna tveganja. Predstavljajo lahko le smiselna mesta v kodi, katera bi bilo potrebno bližje analizirati. Ob odkriti napaki dobimo še opis napake ter reference, s katerimi si lahko pomagamo in po potrebi varnostno tveganje odpravimo. Find Security Bugs ima trenutno 38 detektorjev in 45 različnih vzorcev napak [72].



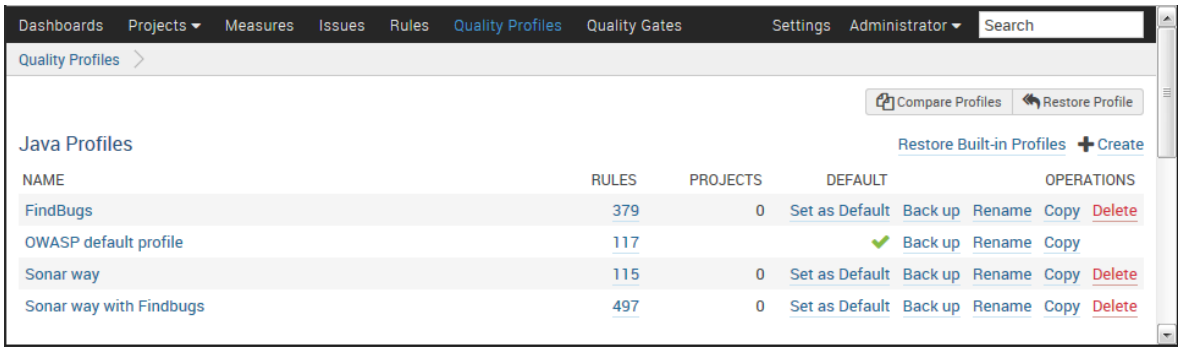
Slika 20: Nastavitve FindBugs za vtičnik Find Security Bugs v okolju Eclipse

2.3.2 Vtičnik OWASP (Excentia) za SonarQube

Za platformo SonarQube obstaja sicer plačljivi vtičnik OWASP razvit pri podjetju Excentia, ki je namenjen prav za analizo in revizijo varnostnih tveganj v kodi spletnih aplikacij [69]. Vsebuje pravila za odkrivanje tveganj po varnostnem dokumentu OWASP Top 10. Podprtih je več programskih jezikov: Java, PHP, JavaScript, Web, VB.NET, C#, VB6.

Uporaba vtičnika

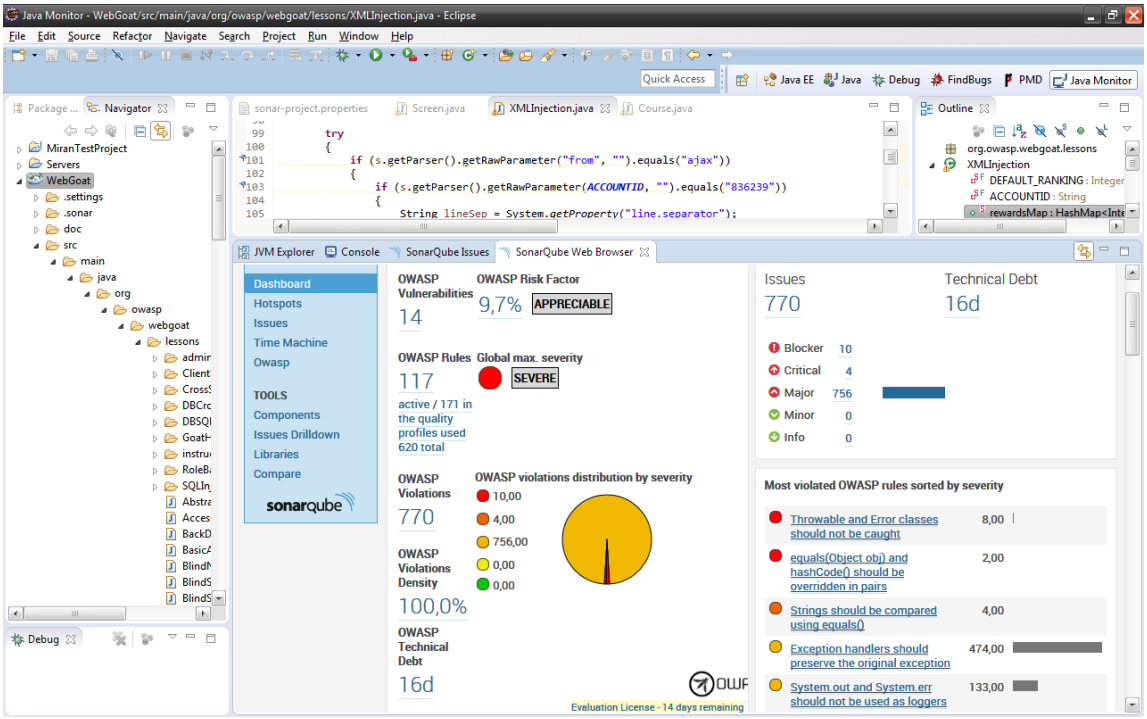
Vtičnik lahko brezplačno preizkušamo 14 dni. Prenesemo ga v mapo, kjer se nahajajo vtičniki za SonarQube in ponovno zaženemo SonarQube strežnik. V spletnem vmesniku za SonarQube v nastavitvah izberemo, naj bo privzet profil kakovosti (*ang. quality profiles*) OWASP profil (slika 21). Npr. v okolju Eclipse lahko nato z vtičnikom za SonarQube naredimo analizo nad svojim projektom.



Slika 21: Nastavitve profila za analizo projektov v spletnem vmesniku SonarQube

Značilnosti

SonarQube OWASP vtičnik nam omogoča vključitev dodatnih gradnikov, ki jih lahko postavimo na naš osnovni pregled nad našim celotnim projektom v spletnem vmesniku SonarQube (slika 22). Odkrite napake razvrsti v več različnih gradnikov, kjer dobimo informacije, koliko varnostnih tveganj naša aplikacija vsebuje. Prikaže nam varnostni nivo tveganja kršitev (*ang. violations debt*), kateremu je naša aplikacija izpostavljena. Faktor varnostnega tveganja (*ang. risk factor*) nam v odstotkih prikaže, kako ranljiv je naš projekt v relaciji števila kršitev ter velikosti celotnega projekta. Globalna resnost napak (*ang. global max. severity*) nam na vizualen način v barvnem odtenku prikaže, kako resne so najdene napake v projektu ne glede na to, da imamo morda nizek faktor tveganja. Gostota kršitev (*ang. violations density*) pa nam v odstotkih pokaže količino varnostnih kršitev v našem projektu [73].



Slika 22: Prikaz varnostnih tveganj z OWASP vtičnikom za SonarQube v Eclipse

Razširitev pravil in ustvarjanje svojega varnostnega modela

Znotraj vtičnika SonarQube OWASP se nahaja `xml` datoteka, v kateri so definirana pravila za zaznavanje napak, ki se uporabljajo v privzetem načinu nastavitev [74]. Ta pravila lahko spreminjamo, jih odstranjujemo ali dodajamo. Vse to lahko naredimo kar preko spletnega uporabniškega vmesnika za SonarQube [75].

2.3.3 Starejša odlična orodja za odkrivanje napak

Orodja Google CodePro Analytix [63] in OWASP LAPSE+ [64] so bila izjemno dobra brezplačna orodja za analizo varnostnih pomanjkljivosti v izvorni kodi. Žal pa za obe orodji že nekaj časa ni posodobljenega vtičnika, tako sta zastareli in tudi nista več podprti v trenutno zadnjem razvojnem okolju Eclipse (Luna). Omenjeni orodji se lahko preizkusi v starejših verzijah okolja Eclipse, za današnje potrebe pa po našem mnenju zaradi nepodprtosti in brez novih posodobitev nista več primerni. V kolikor bodo orodji v prihodnosti posodobili, se jih nedvomno splača preizkusiti.

2.4 Orodja za profiliranje

Z orodji za profiliranje (*ang. profiling tools*) lahko merimo, kako dolgo se posamezni deli izvorne kode izvajajo, koliko spomina in koliko procesorske moči pri tem zahtevajo. Ta orodja podobno kot orodja za razhroščevanje delujejo na principu dinamične analize kode in na ta način kodo analizirajo. To se izkaže za posebej učinkovito, če naša koda vsebuje dele, ki se izvajajo počasneje kot bi si želeli. Ravno z orodjem za profiliranje lahko odkrijemo tiste dele kode, ki nam povzročajo največ izgube pomnilnika in procesorske moči ter se lahko tako lažje usmerimo v izboljšavo teh delov. Analizo lahko izvajamo tako nad izvorno kodo kot tudi na že prevedeni bitni kodi. Veliko razvijalcev kode verjame, da človek ne bi smel ročno optimizirati kode, dokler ne bi uporabil orodja za profiliranje.

Upravljanje javanskega spomina

Java že sama upravlja s spominom, katerega pri svojem delovanju uporablja. Novi objekti, ki se med izvajanjem aplikacije tvorijo, se shranjujejo na za to rezerviran prostor imenovan kopica (*ang. heap*). Ko v javanski aplikaciji ni več reference na določen objekt (ki je na kopici shranjen), ga lahko javanski "sproščevalnik spomina" (*ang. garbage collector*) samodejno pobriše in na ta način sprosti prostor na kopici. Dokler obstajajo reference na določeni objekt, ga sproščevalnik spomina ne more zbrisati in ravno v takih primerih lahko prihaja do tako imenovanega "puščanja pomnilnika" (*ang. memory leak*) [76]. S programi za profiliranje lahko odkrivamo, kateri deli kode so tisti, ki povzročajo nenormalno veliko porabo pomnilnika ali procesorske moči in lahko pri tem povzročijo tudi puščanje pomnilnika.

Tipična orodja za profiliranje

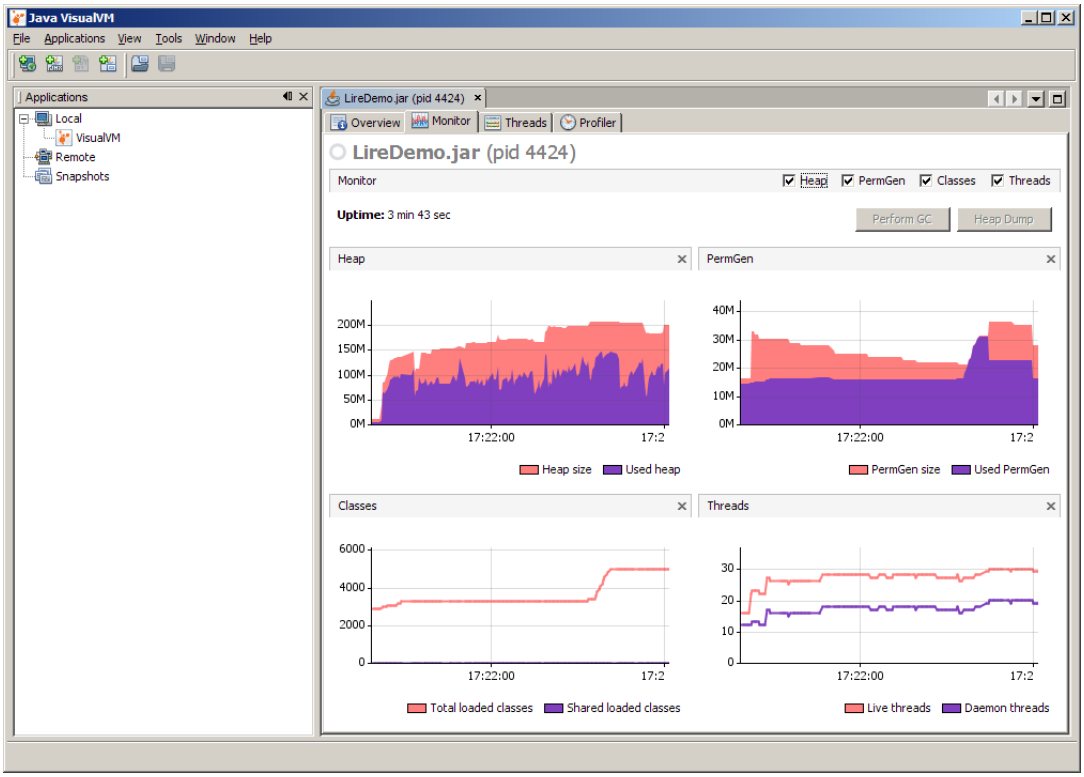
V nadaljevanju bomo opisali brezplačno orodje za profiliranje VisualVM [77], ki je po številnih forumih zelo priljubljeno in je že v osnovi del `jdk`. Za njim bomo predstavili še orodje JVM Monitor [88], ki se zelo dobro integrira z razvojnim okoljem Eclipse. Na trgu obstajajo tudi plačljiva napredna orodja za profiliranje, ki omogočajo še dodatne analize z atraktivnimi uporabniškimi vmesniki, med njimi morda najbolj odmevna JProfiler [78] in YourKit [79].

2.4.1 VisualVM

VisualVM je brezplačno vizualno orodje za profiliranje razvito pri podjetju Oracle, ki vsebuje množico `jdk` orodij, katera se sicer lahko poganja iz ukazne vrstice [77]. Omogoča nam, da preko grafičnega vmesnika dobimo vpogled, kako deluje javanska aplikacija, ko se koda naše aplikacije izvaja v javanskem navideznem stroju. Orodje združi podatke, ki jih dobi od `jdk` orodij in informacije prikaže na enem mestu. Našo aplikacijo lahko z orodjem analiziramo, profiliramo porabo procesorja in pomnilnika, pregledujemo obnašanje niti, dobimo podatke, kaj se dogaja na kopici (*ang. heap dump*)... Analiziramo lahko tudi oddaljene javanske aplikacije po celotni mreži. Delamo lahko posnetke kopice in posnetke niti (*ang. threads*) v določenem trenutku. Analiziramo lahko že shranjene pomnilniške izpise (*ang. core dumps*). Orodje je zgrajeno na platformi NetBeans in je narejeno tako za produkcijsko kot tudi za razvojno uporabo [80].

Uporaba orodja kot samostojne aplikacije

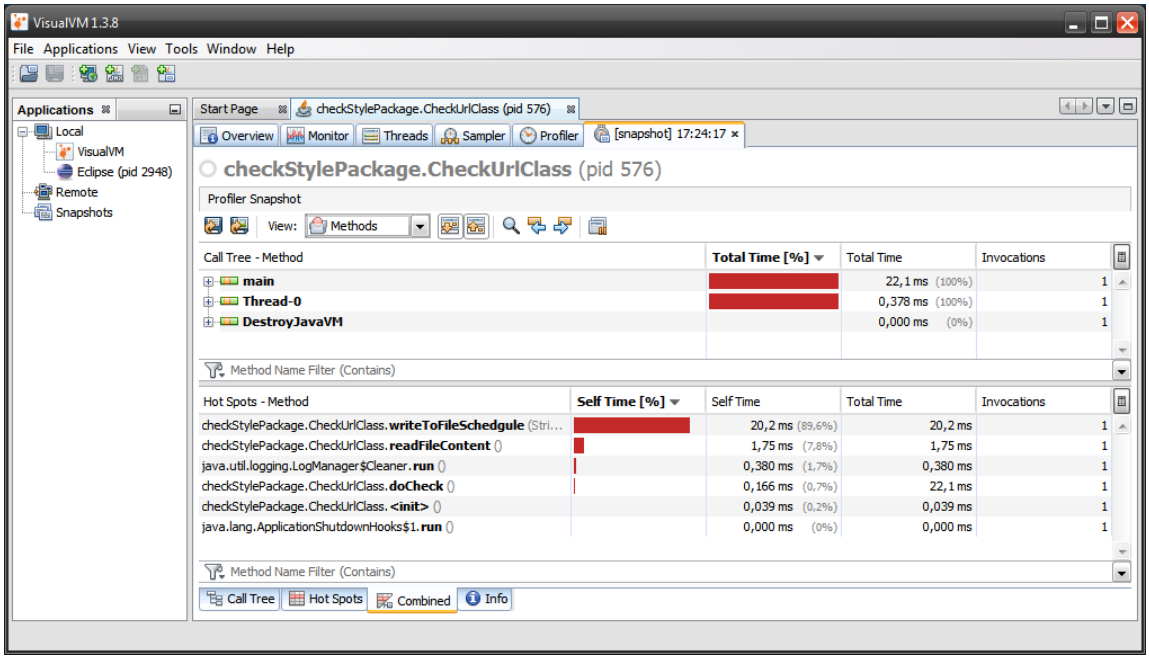
VisualVM se uporablja kot samostojna aplikacija (slika 23) [81]. Analizira nam vse trenutno aktivne javanske aplikacije, ki tečejo na našem lokalnem sistemu ali pa tečejo na oddaljenih sistemih [82].



Slika 23: Uporaba orodja VisualVM kot samostojne aplikacije

Uporaba orodja preko razvojnega okolja Eclipse

VisualVM samostojno aplikacijo lahko poženemo kar iz razvojnega okolja Eclipse, v kolikor namestimo za to ustrezen vtičnik [83]. Orodje se nam bo samodejno zagnalo iz okolja Eclipse. Nad javanskimi razredi lahko z orodjem poženemo analizo in hitro ugotovimo, katere metode v naših razredih porabijo največ časa, pomnilnika, procesorja in jih po potrebi tudi optimiziramo.



Slika 24: Uporaba orodja VisualVM pri analizi porabe procesorja

Analiza porabe procesorja ali spomina

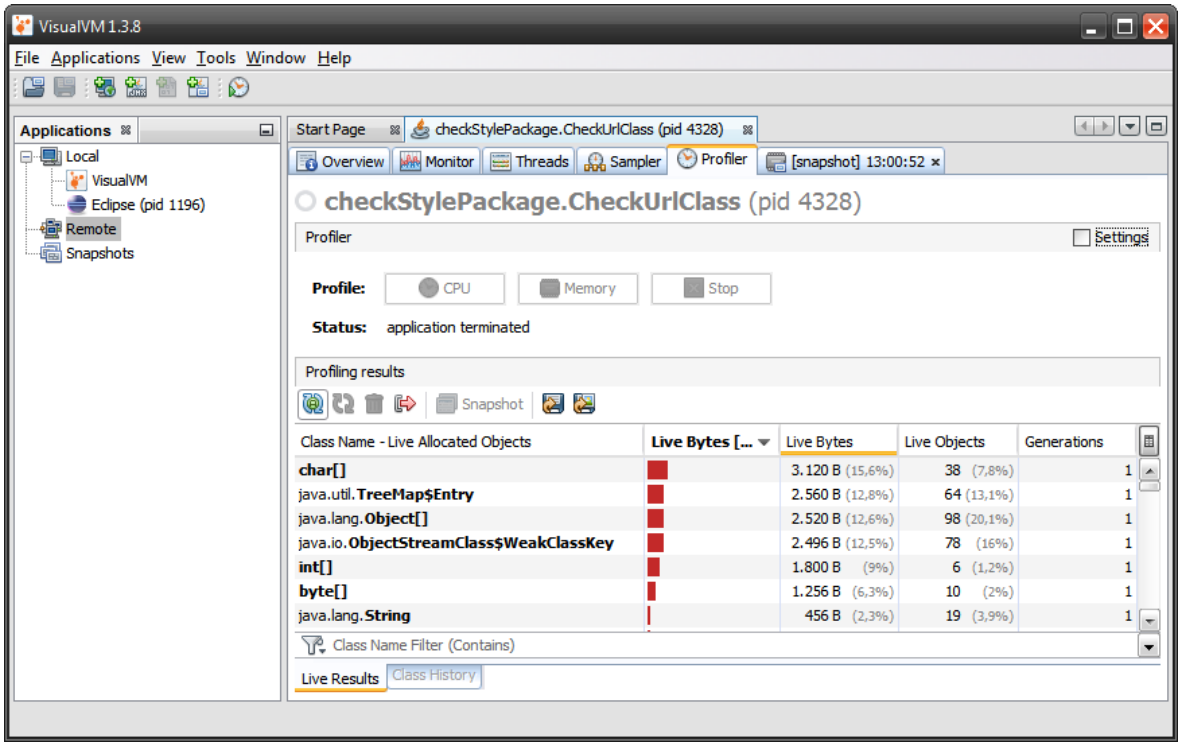
Analiziramo lahko tako uporabo procesorja kot tudi porabo pomnilnika. V osnovi orodje samo ne prične s pridobivanjem podatkov, v kolikor ga v ta namen ne nastavimo. Ko pričnemo s sejo analize, se VisualVM priključi na aplikacijo, katero želimo analizirati in prične pridobivati podatke. Ko so podatki dosegljivi, se v orodju samodejno prikažejo.

Pri analizi porabe procesorja orodje vrne podrobne informacije o skupnem času izvedbe celotne aplikacije in številu proženja posameznih metod (slika 24) [84].

Analiza porabe pomnilnika

Ko analiziramo uporabo pomnilnika, orodje prične z analizo trenutno izvajajočih javanskih razredov in prikaže skupno število objektov, ki so dodeljeni posameznemu razredu. Za vsak trenutno naloženi razred v javanskem navideznem stroju orodje prikaže porabo pomnilnika in število objektov, ki so bili dodeljeni, odkar se je analiza porabe pomnilnika

pričela. VisualVM prikaže število klicanih objektov kot absolutno številko in v odstotkih. Dodeljene količine so ravno tako prikazane tudi v grafični obliki (slika 25).



Slika 25: Uporaba orodja VisualVM pri analizi porabe pomnilnika

Shranjevanje posnetkov

VisualVM omogoča, da lahko med izvajanjem analize naredimo posnetek trenutnega stanja seje (*ang. snapshots*). Posnetke si lahko shranjujemo in jih med seboj primerjamo. Naredimo lahko posnetke profiliranja procesorja in spomina, lahko pa ustvarimo tudi posnetek trenutnega stanja celotne seje aplikacije, kamor se shranijo podatki o kopici, javanskih nitih in splošne informacije javanskega navideznega stroja v času zajema posnetka [85].

Profiliranje hitro izvršljivih aplikacij

Če se javanska aplikacija v razvojnem okolju Eclipse izvede zelo hitro, se lahko zgodi, da orodje VisualVM ne uspe pridobiti podatkov izvršene aplikacije. V takem primeru si pomagamo tako, da v naši aplikaciji postavimo prekinitveno točko, jo zaženemo v razhroščevalnem načinu (*ang. debug mode*), pričnemo z analizo in spustimo našo aplikacijo do konca. Tako nam orodje zlahka odkrije kritične dele v naši aplikaciji.

Dodajanje novih vtičnikov

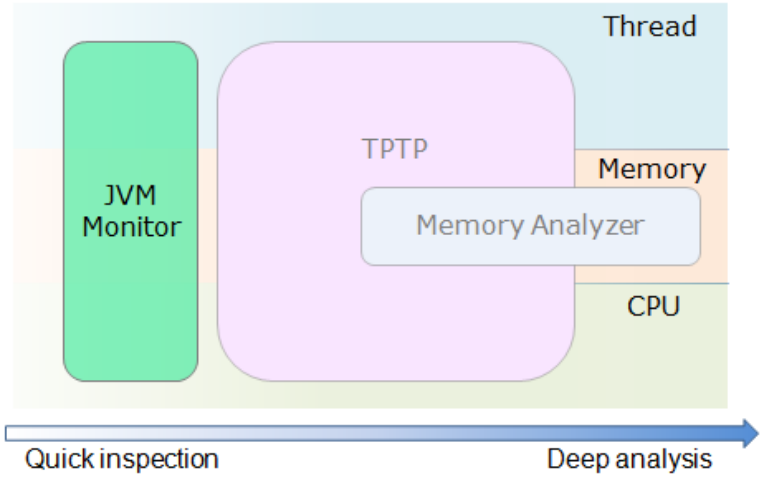
Uporabnost orodja VisualVM lahko razširimo še z dodatnimi uporabnimi vtičniki [86]. Kot primer uporabnega vtičnika velja omeniti vtičnik Startup Profiler, ki je namenjen prav za profiliranje kratkih hitro izvršljivih aplikacij [87].

2.4.2 JVM Monitor

JVM Monitor je brezplačno javansko orodje za profiliranje, ki se odlično integrira z razvojnim okoljem Eclipse [88]. Z njim lahko pregledujemo porabo procesorske moči, obnašanje javanskih niti ter porabo pomnilnika, ki ga aplikacije zasedejo. Še posebej uporaben je zato, ker lahko hitro analiziramo javanske aplikacije, brez predhodnih nastavitev zaganjanja. JVM Monitor samodejno zazna trenutno aktivne javanske navidezne stroje na lokalnem računalniku in jih lahko hitro analiziramo. Z orodjem je možno analizirati tudi oddaljene aplikacije.

Namen orodja

Orodje je namenjeno le za hitre analize javanskih aplikacij [89]. V kolikor bi potrebovali globlje analize, je priporočljivo uporabiti še kakšna druga orodja kot na primer TPTP [90] ali Memory Analyzer [91] (slika 26).

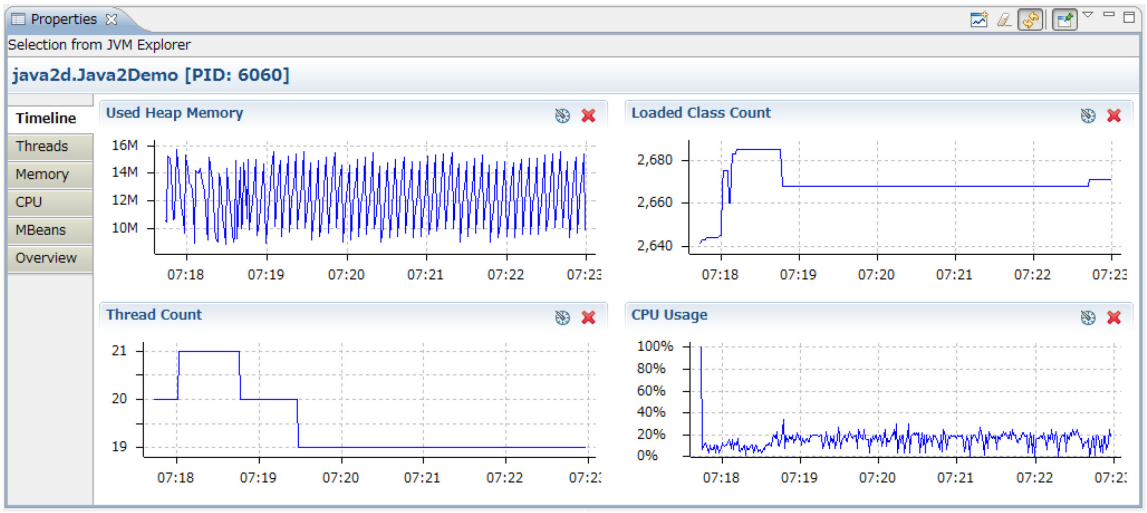


Slika 26: Prikaz funkcionalnega delovanja orodja JVM Monitor z drugimi orodji

Uporaba orodja v razvojnem okolju Eclipse

V razvojnem okolju Eclipse izberemo perspektivo Java Monitor in odpre se nam zavihek JVM Explorer. Tu vidimo vse trenutno aktivne javanske aplikacije, ki tečejo na lokalnem

računalniku. Javansko aplikacijo z desnim klikom miške označimo in že jo lahko pričnemo analizirati. V zavihku časovnice (*ang. timeline*) imamo prikazane osnovne podatke o analizi aplikacije, kjer lahko pregledujemo trend uporabe pomnilnika kopice, število naloženih javanskih razredov, prikaz javanskih niti in uporabo procesorske moči (slika 27) [92].

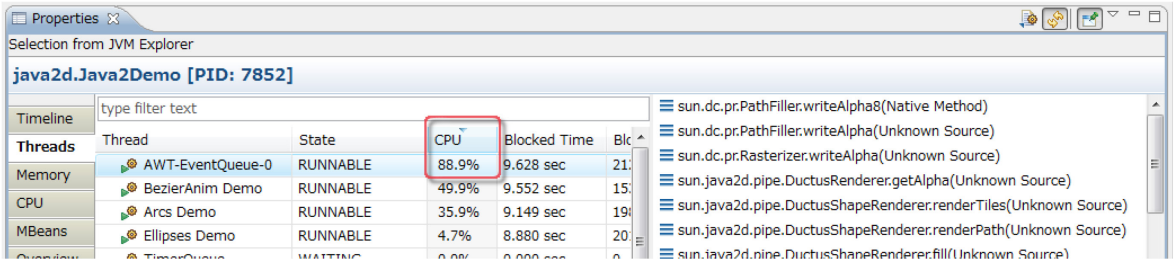


Slika 27: Grafični prikaz analize aplikacije z orodjem JVM Monitor

V prikazu časovnice je možno dodajati dodatne podatke, ki jih želimo analizirati, lahko pa definiramo tudi nove grafe (slika 27).

Analiza javanskih niti

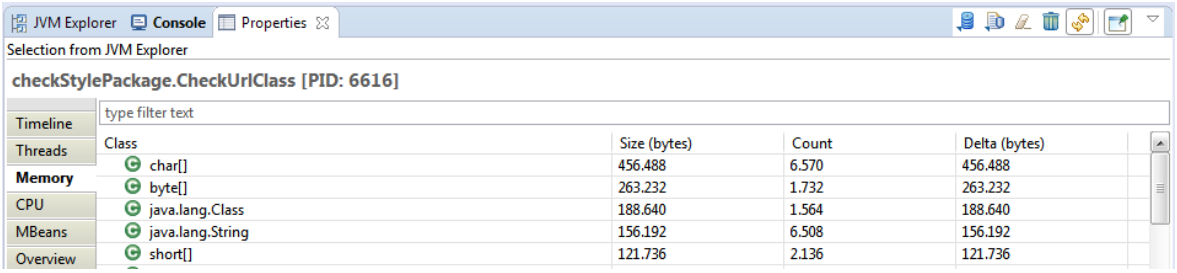
Pri podrobnem pregledovanju javanskih niti (slika 28) nam orodje prikaže stanje niti, npr. če pride kje do mrtvih zank (*ang. deadlock*). Če opazimo, da naša aplikacija zelo obremeni procesor, lahko analiziramo, kateri del v naši kodi povzroča največ obremenitev. Poiščemo tisto nit, ki porabi največ procesorske moči, jo kliknemo in prikaže se nam kritična metoda na javanskem skladu (*ang. stack trace*).



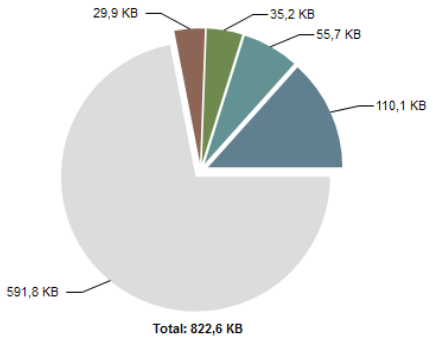
Slika 28: Prikaz analize javanskih niti z uporabo orodja JVM Monitor

Analiza porabe pomnilnika

Če opazimo, da je poraba spomina kopice na grafu nepričakovano velika, lahko identificiramo, kateri objekti to povzročajo (slika 29). Zajamemo lahko posnetek kopice v določenem trenutku, ki se shrani v datoteko s končnico `HPROF` [93]. Posnetek lahko po potrebi dalje analiziramo npr. z orodjem Memory Analyzer (MAT), katerega zadnjo verzijo si lahko namestimo preko vtičnika in z njim posnetke v razvojnem okolju Eclipse dalje samodejno analiziramo (slika 30) [94].



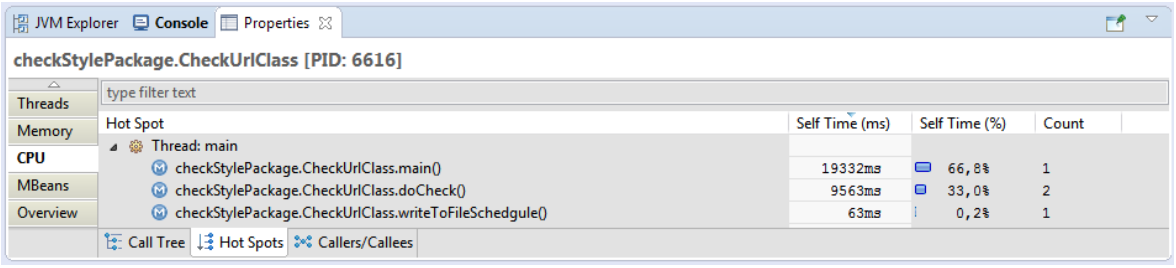
Slika 29: Prikaz analize porabe pomnilnika z uporabo orodja JVM Monitor



Slika 30: Nadaljnja analiza posnetka kopice z orodjem Memory Analyzer (MAT)

Analiza porabe procesorske moči

Pri analizi lahko hitro vidimo, katere metode zasedejo največ procesorske moči (slika 31). Pregledujemo lahko vroče točke (*ang. hot spots*). Z dvoklikom na posamezno metodo se nam označi tudi metoda javanskem razredu. Če opazimo, da je neka metoda klicana zelo pogosto, lahko naprej analiziramo, katere metode so to metodo klicale (*ang. callers/calles*).



Slika 31: Prikaz analize porabe procesorske moči z uporabo orodja JVM Monitor

Analiza aplikacij na oddaljenem računalniku

JVM Monitor nam omogoča analizo trenutno aktivnih javanskih aplikacij na oddaljenem računalniku na katerega se povežemo. Ko analiziramo oddaljeno aplikacijo, obstaja omejitev, da ne moremo videti histograma kopice, lahko pa naredimo posnetek kopice in ga po potrebi analiziramo bolj podrobno.

3. Določanje nabora testnih orodij

Pri določanju nabora testnih orodij je potrebno postaviti smiselne kriterije. Naše zahteve do vseh orodij so take, da nam je najbolj pomembna preprostost uporabe določenega orodja. To pomeni, da ga je enostavno namestiti, se ga hitro priučiti, ga učinkovito uporabljati in si z orodjem na ta način olajšati vsakodnevno delo v razvoju javanskih aplikacij.

Orodja bomo ocenili po naslednjih kriterijih:

- dokumentiranost ogrodja (ali obstajajo knjige, spletna stran, spletni tečaji...)
- enostavnost namestitve (ali se hitro namesti, ali se uporablja v okolju Eclipse...)
- prijazen uporabniški vmesnik (ali je prijeten za uporabo, ali ima grafične prikaze...)
- učna krivulja (ali se da orodje hitro naučiti, ali je enostavna uporaba orodja...)
- funkcionalno pokrivanje (katere funkcionalne dele orodje pokriva...)

3.1 Izbira ogrodja za testiranje enot

Pri pregledu ogrodij za testiranje enot v razvojnem okolju Eclipse smo uporabili trenutno zadnji verziji ogrodij:

- JUnit (verzije 4.11.0) in
- TestNG (verzije 6.8.6).

Vtisi

Ogrodji smo na ravni osnovne uporabe pričeli hitro uporabljati. Na koncu testiranja obe ogrodji prikažeta lepo poročilo, pri TestNG dobimo celo poročilo v prijetni `html` obliki. Pri obeh ogrodjih smo naleteli na občasne manjše programske hrošče, ki so morda posledica integracije s trenutno zadnjo verzijo razvojnega okolja Eclipse (Luna). Pri JUnit smo npr. pri podatkovno usmerjenem testiranju na lepem v konzolo (*ang. console*) pričeli dobivati napako tipa "*`unrooted test junit initializationerror`*". Po brskanju v forumih se je izkazalo, da bi ogrodje lahko motil določen stavek v naši kodi, ki izpisuje na standardni izhod (`System.out.println()`). Ko smo omenjeni stavek zakomentirali, se napaka res ni več pojavila, vendar tudi potem, ko smo stavek ponovno odkomentirali, napake ni bilo več zaznati. To se je zgodilo le enkrat in ni moteče vplivalo na nadaljnje testiranje. Tudi pri TestNG se je enkrat pojavila manjša napaka pri prevajanju, ki pa se ni več ponovila. V TestNG smo pri poganjanju testnih metod, ki smo jih grupirali, naleteli na težave po naši krivdi. Izkazalo se je, da smo pozabili v grupo dodati tudi metodo `@BeforeMethod`, v kateri smo imeli inicializacijo objekta, zato se seveda ostale metode sprva niso pravilno izvršile.

Dokumentiranost ogrodja

Tako JUnit kot TestNG imata ustrezno dokumentacijo in knjige. Za JUnit se toplo priporoča knjiga: "*JUnit in Action*" [94] za TestNG pa "*Next Generation Java Testing*" [95]. Dokumentacijo se najde tako na straneh ogrodij kot tudi na različnih pogosto uporabljenih spletnih straneh, ki vsebujejo spletne tečaje in enostavne primere uporabe za hitro ter učinkovito delo. Tako JUnit kot TestNG sta dobro dokumentirana.

Enostavnost namestitve

JUnit je že del distribucije okolja Eclipse, TestNG pa se enostavno namesti z dodajanjem novega vtičnika. Glede na to, da je JUnit že del distribucije, ima v tem kriteriju malo prednosti in je tudi na podlagi tega v svetu razvoja programske opreme bolj razširjen.

Prijazen uporabniški vmesnik

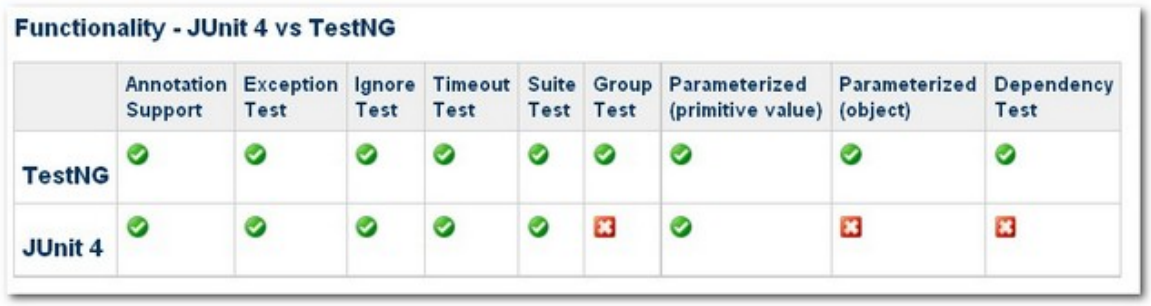
Obe ogrodji se lahko zaganjata tudi s klikom miške in tu ni velikih razlik. Obe imata enostaven prikaz rezultatov o testiranju, pri čemer pa velja poudariti, da ima ogrodje TestNG veliko več osnovnih možnosti. Že v privzetih nastavitvah nam naredi poročila v različnih razširjenih formatih (npr. `html`, `xml`) in ga je enostavno prilagoditi tudi za poročila po lastnem okusu. Pri JUnit lahko sicer tudi sami programsko razvijemo drugačne končne formate in prikaze rezultatov. Ker je večina najbolj uporabnih formatov že v privzetem načinu podprta v TestNG, je po tem kriteriju TestNG primernejši.

Učna krivulja

Obe ogrodji se da enostavno in hitro osvojiti, saj obstaja dobra dokumentiranost. Zlahka najdemo knjige in enostavne spletne tečaje za hitro osvojitev osnovnih konceptov uporabe. Še posebej hitro se da osnovno znanje za ogrodji osvojiti z ogledom interaktivnih spletnih tečajev za hitri pričetek dela, ki obstajajo tako za JUnit [12] kot za TestNG [97]. Po kriteriju učne krivulje nobeden od obeh ogrodij ne prednjači.

Funkcionalno pokrivanje

JUnit je namenjen izključno za testiranje enot. TestNG pa poleg tega nudi podporo še za druge vrste testiranja (npr. funkcijsko in integracijsko). Že na nivoju testiranja enot se pokažejo razlike med ogrodji (slika 32) [98].



Slika 32: Primerjava funkcij JUnit in TestNG (leta 2009)

(Slika 32) prikazuje stanje funkcionalnega pokrivanja JUnit 4 in TestNG, kot je bilo v letu 2009. Takrat je bil TestNG očitno boljši. JUnit še ni omogočal grupiranja testov, parametriziranega testiranja z objekti (*ang. parametrized test*) in pa uporabe med seboj odvisnih testov (*ang. dependency test*). V številnih spletnih forumih še vedno obstajajo povezave na primerjavo iz leta 2009 in novi uporabnik lahko zelo hitro naleti na omenjeno primerjavo in se na podlagi tega tudi odloči za ogrodje, ki ga bo pri svojem delu uporabljal. Tudi v različnih spletnih tečajih hitro pridemo do omejitev ogrodja JUnit, ki pa se je v zadnjih letih zelo izpopolnil. V današnjih zadnjih stabilnih verzijah JUnit (4.11.0) in TestNG (6.8.6) je v obeh ogrodjih možno izvajati teste za različne tehnike objektno usmerjenega pristopa. Tako TestNG kot tudi JUnit vztrajno dopolnjujeta in izboljšujeta svoj produkt in uporabo le-tega. Za primer JUnit je v zadnjih verzijah dodal tudi podporo za paralelno izvajanje testov, kar je imelo pred tem le ogrodje TestNG. Orodju JUnit je dodana podpora za grupiranje razredov ali metod z uporabo kategorij *(ang. categories*) [99]. Predstavljeni so novi koncepti kot npr. predpostavke (*ang. assumptions*), s katerimi je mogoče zaobiti pomanjkljivost testiranja med seboj odvisnih metod [100]. Uporaba javanske oznake `@Parameter` omogoča parametrizirano izvajanje podatkovno usmerjenega testiranja brez uporabe konstruktorjev [101].

Ugotovitve

TestNG ima v resnici na voljo večje število javanskih oznak, ki so po našem mnenju tudi malo bolj intuitivne kot pri ogrodju JUnit. Lažje ga je tudi uporabljati pri konfiguraciji velikega števila testov. JUnit v starejših verzijah ni omogočal testiranja med seboj odvisnih metod, TestNG je to že od vsega začetka omogočal z uporabo parametra "`dependsOnMethods`", ki nam lahko pride prav v teh primerih. Tudi v JUnit je sicer v najnovejši verziji možno zaobiti to pomanjkljivost, vendar na ne tako enostaven način kot pri TestNG. TestNG v resnici pokriva prav vse aspekte, kar jih pokriva JUnit, poleg tega pa omogoča še dodatne funkcionalnosti [98].

Za male projekte ali za učenje ogrodje JUnit povsem zadošča, saj lahko vsak zelo hitro prične z uporabo testov enot. Za projekte, kjer pa bo potrebno spisati veliko testov in bomo za njih imeli različne scenarije poteka, pa bi morda lahko bila bolj primerna uporaba ogrodja TestNG [102].

Kar se tiče testiranja enot sta sicer ogrodji bolj ali manj izenačeni, sploh glede na dejstvo, da je v obeh ogrodjih v najnovejših verzijah možno pokriti praktično vse potrebe razvijalcev. Glede na to, da pa ima ogrodje TestNG boljšo konfiguracijo testiranja (npr. s nastavitveno datoteko `testng.xml`) in da pokriva poleg testiranja enot še druge nivoje testiranja, se ga je po našem mnenju morda primerneje priučiti in ga uporabljati, saj nam bo lahko prišel prav tudi še na drugih nivojih testiranja. Odločitev za izbiro ogrodja gre na podlagi teh ugotovitev in kriterijev v prid ogrodja TestNG.

Izbira prvega ogrodja za naš nabor testnih orodij je tako **TestNG**.

3.2 Izbira orodja statične analize

Verzije vtičnikov za razvojno okolje Eclipse oziroma verzije orodij, ki smo jih analizirali so naslednje.

- Findbugs vtičnik za Eclipse (verzije 3.0.0),
- PMD vtičnik za Eclipse (verzije 4.0.3),
- CheckStyle vtičnik za Eclipse (verzije 5.7.0) in
- SonarQube vtičnik za Eclipse (verzije 3.4.0) z javanskim vtičnikom za platformo SonarQube (verzije 2.2.1).

Vtisi

FindBugs orodje se nam je zdelo izredno zanimivo, saj je res našlo napake, ki jih prevajalnik sam ni našel. Napake je orodje lepo združilo in s kliki na napake se je odprla tista vrstica v kodi, ki vsebuje najdeno napako. Izkazalo se je da FindBugs zaradi globoke analize porabi kar nekaj pomnilnika, vendar le v času, ko analizira vse prevedene datoteke v projektu. Zatem so napake označene in lepo razvrščene. Uporaba orodja je bila zelo enostavna, podobno velja tudi za PMD in za CheckStyle. SonarQube je bil malo trši oreh, saj nam ga ni uspelo takoj pravilno namestiti. Sicer dokumentacija je zelo dobra, si je pa za to potrebno vzeti malo več časa, da se vse pravilno nastavi. V spletnih forumih sicer izkušeni razvijalci pišejo, da uspejo platformo SonarQube nastaviti za delo v nekaj

minutah. Sami smo porabili malo več časa, da je vse delovalo tako kot je treba. Imeli smo tudi manjše težave pri povezavi projekta v okolju Eclipse z SonarQube strežnikom. Izkazalo se je, da je potrebno prvo analizo projekta narediti zunaj okolja Eclipse s SonarQube zaganjalnikom in je šele zatem orodje možno uporabljati neposredno v okolju Eclipse. Ko smo te težavice odpravili, se je pokazalo, da je SonarQube izjemno prijeten za uporabo. Ima čudovit spletni uporabniški vmesnik, do katerega lahko dostopamo kar v okolju Eclipse ali pa v navadnem spletnem brskalniku. Najdene napake se nam ves čas lepo izpisujejo. Vidi se, da so razvijalci ogrodja dali velik pomen prijetnemu uporabniškem vmesniku. Res pa je, da za delo SonarQube potrebuje malo močnejši računalnik, saj komunicira s strežnikom in podatkovno bazo.

Dokumentiranost orodja

Vsa orodja so dobro dokumentirana. Imajo prosto dostopne spletne informacije za namestitve ter za njihovo uporabo. Tudi za platformo SonarQube obstaja obširna dokumentacija. Na voljo je tudi knjiga tako za orodje PMD [103] kot tudi za platformo SonarQube [104].

Enostavnost namestitve

FindBugs, PMD in CheckStyle je v okolje Eclipse povsem enostavno namestiti. Vsa tri orodja namestimo preko vtičnika. Tudi FindBugs samostojna aplikacija zahteva le, da imamo nameščeno Javo 1.7 in že jo lahko poženemo. SonarQube zahteva več dela, saj je pri njem potrebno namestiti poleg vtičnika za okolje Eclipse tudi strežnik, podatkovno bazo in zaganjalnik, v kolikor želimo imeti sistem postavljen pri sebi. Pri tem je potrebno kar nekaj ročne konfiguracije. Glede enostavnosti namestitve so FindBugs, PMD in CheckStyle v prednosti.

Prijazen uporabniški vmesnik

FindBugs, PMD in CheckStyle je povsem enostavno uporabljati v okolju Eclipse. Nad projektom poženemo analizo posameznih orodij in že lahko pregledujemo in odpravljamo najdene napake, ki se pokažejo v obarvanih vrsticah kode ter opisom napak, ko se vrstici približamo s kazalčkom miške. V SonarQube vtičniku pa imamo nad projektom še visokonivojski pregled, saj imamo na voljo zelo prijeten spletni uporabniški vmesnik z grafičnimi prikazi, preko katerega pregledujemo odkrite napake. Vsa orodja lepo prikažejo odkrite napake, SonarQube pa nam nudi še spletni uporabniški vmesnik, ki vsebuje veliko grafičnih prikazov, pregleda zgodovine za nazaj, nastavljanja konfiguracij. Glede na to, da ima SonarQube tako napreden uporabniški vmesnik, ima po tem kriteriju prednost.

Učna krivulja

Za FindBugs, PMD in CheckStyle ne potrebujemo predhodnega znanja o uporabi orodij, saj so enostavna in nam analizirajo ter prikažejo odkrite napake v naši kodi. Orodja lahko pričnemo hitro uporabljati. Prebrati moramo malo več dokumentacije, v kolikor želimo nastavljati in po potrebi vključevati / izključevati določene tipe napak. Za bolj napredno uporabo lahko razvijamo tudi lastna pravila in se moramo v tem primeru poglobiti v delovanje internih razčlenjevalnikov kode. Pri platformi SonarQube je učna krivulja bolj počasna kot pri prejšnjih orodjih, saj gre tu za večji skupek orodij, oziroma za platformo, ki pokriva več stvari. Tudi možnosti v spletnem uporabniškem vmesniku je veliko. Za osnovne stvari je sicer uporaba orodja intuitivna. Da pa osvojimo napredne možnosti, ki jih orodje ponuja, potrebujemo nekaj več časa. Že pri namestitvi orodja se moramo malo bolj potruditi, saj tu ni dovolj le namestitev vtičnika. Potrebno je prebrati nekaj dokumentacije, da lahko platformo učinkovito uporabljamo in ji po potrebi spreminjamo nastavitve globalnih ali posameznih notranjih orodij, ustvarjamo lastne profile. Po kriteriju učne krivulje se najhitreje naučimo uporabljati orodja FindBugs, PMD in pa CheckStyle.

Funkcionalno pokrivanje

Vsa orodja izvajajo statično analizo kode in najdejo tudi napake, ki jih sam prevajalnik v izvorni kodi ne uspe zaznati. Pri tem velja poudariti, da si je primerno pravila orodij v praksi nastaviti na nivo, da ne zaznavajo preveč napak takega tipa, ki v resnici niti niso "prave" napake. Najbolj občutljivih in nekritičnih napak ne potrebujemo imeti prikazanih in se tako posvetimo raje pravim popravkom. Sicer lahko izgubimo preveč časa, da bi zadostili vsem zahtevam orodij. Vedno se moramo namreč zavedati, da so orodja v osnovi namenjena temu, da služijo nam, ne pa da mi služimo njim.

Tako PMD, CheckStyle kot FindBugs so trenutno najboljša izbira orodij za statično analizo kode v programskem jeziku Java. Vsa orodja iščejo različne tipe napak. PMD in CheckStyle analizirata izvorno javansko kodo. PMD se uporablja za odkrivanje napak slabe prakse kodiranja, preveč zapletene kode, podvojene kode... Odkrije tudi preveč zapleteno kodo npr. tipa (*ang. cyclomatic complexity*), ki označuje koliko neodvisnih razvejanj (možnih poti) imamo v določenem modulu. Orodje FindBugs npr. napak tega tipa ne zaznava. CheckStyle na drugi strani sicer ne najde pravih napak, odkrije pa stilske napake v kodi in z njim lahko preverjamo, če naš stil pisanja kode ustreza določenemu standardu kodiranja. Ravno v podjetjih, kjer obstaja zahteva po enotnem stilu kodiranja, se lahko izkaže za zelo uporabno orodje. Orodje FindBugs analizira že prevedene datoteke in je namenjeno za odkrivanje potencialnih napak, ki se v večini primerov izkažejo, da so prave napake in jih je potrebno nujno odpraviti. Orodje FindBugs je neprecenljivo, saj nam

lahko prihrani marsikatero težavo, ki bi se lahko sicer kasneje pojavila, v kolikor napak ne bi odkrili sami.

Vsa tri orodja se kot omenjeno uporabljajo za statično analizo kode in so namenjena odkrivanju in odpravi napak, pripravi poročil, vključevanju v samodejna grajenja projektov z orodji Ant in Maven. Nikakor ne bi bilo smiselno, da bi izbrali le eno orodje, saj ravno vsa tri orodja skupaj nudijo tisto pravo kombinacijo moči in funkcij odkrivanja napak, ki so nepogrešljive pri razvoju programske opreme. Zato bi bilo najbolj smiselno uporabljati kar vsa tri orodja skupaj.

Platforma SonarQube pa vsebuje vsa tri omenjena orodja, oziroma jih lahko po želji vključimo / izključimo. Vsebuje tudi svoje interno orodje `SSLR`, v katero so prepisana pravila orodij PMD in CheckStyle. SonarQube ponuja še bolj napredne možnosti npr. pogled na projekte iz visokega nivoja. Z njim lahko pregledujemo tudi pokritost projektov s testi enot. V spletnem uporabniškem vmesniku SonarQube pregledujemo zgodovino napak za nazaj, pregledujemo grafične prikaze napredka odpravljanja napak, spreminjamo globalne nastavitve in nastavitve pravil notranjih orodij.

Ugotovitve

Za razvijalca začetnika je po našem mnenju zelo priporočljivo in dobrodošlo, da se najprej spozna z orodji FindBugs, PMD in CheckStyle (v tem vrstnem redu), saj na ta način spozna njihovo veliko uporabnost. Tudi glede na to, da je namestitev omenjenih treh orodij izjemno enostavna in izvršena le v nekaj minutah, je to izredna prednost orodij. Kakovost kode se bo z uporabo teh orodij drastično izboljšala in razvijalec se bo naučil ogromno dobrih praks kodiranja. Zatem pa lahko razvijalec po želji preide na bolj napredno platformo (npr. SonarQube), ki nudi poleg zaznavanja napak še druge napredne možnosti (npr. združena poročila na enem mestu). Za izkušenega razvijalca bo SonarQube zelo dobrodošel, saj ga lahko vključi v samodejna grajenja projektov. Še posebej za orodje Maven platforma SonarQube ponuja zelo dobro podporo in integracijo. Platforma pride prav tudi tehničnim vodjem projektov, saj lahko iz visokega nivoja pregledujejo stanje kode svojih razvojnih ekip. Ker je platforma SonarQube namenjena za zelo veliko aspektov testiranja programske kode in imamo lahko nad vsem tem še nadzor iz visokonivojskega pogleda, je v letu 2010 prejela tudi nagrado [105].

Torej vsa orodja so odlična, zato je tu izbira orodja odvisna le od zahtev, ki jih imamo do orodij. Če bi potrebovali orodje z naprednimi možnostmi, bi se odločili za SonarQube. Zaradi enostavnosti namestitve, hitre uporabe v okolju Eclipse, manjše porabe pomnilnika

in procesorja ter glede na to, da nismo zahteven uporabnik, pa smo se v tem primeru odločili, da za naš nabor testnih orodij vzamemo kar tri orodja in sicer FindBugs, PMD in CheckStyle. Kombinacijo vseh treh orodij zelo hvalijo po številnih spletnih forumih in so to tista prava orodja, ki imajo že dolgoletno tradicijo ter dobro ime.

Kandidati pri izbiri orodja statične analize za naš nabor testnih orodij so torej **FindBugs, PMD in CheckStyle**.

3.3 Izbira orodja za odkrivanje varnostnih tveganj v kodi

Verzije vtičnikov za razvojno okolje Eclipse, ki smo jih analizirali, so naslednje:

- Find Security Bugs vtičnik za FindBugs (verzije 1.2.0) in
- OWASP (Excentia) vtičnik za SonarQube (verzije 2.0.3).

Vtisi

Z obema orodjema smo analizirali OWASP projekt WebGoat, ki vsebuje znana varnostna tveganja. Find Security Bugs vtičnik je bilo potrebno le vključiti v orodje FindBugs, ga nastaviti, kot piše v navodilih in normalno pognati analizo. Orodje je našlo napake v projektu, ki jih je lepo razvrstilo s standardno FindBugs razvrstitvijo. Še posebej priročno pri tem je bilo, da smo ob najdenih napakah dobi tudi neposredno referenco na napotke za odpravo le-teh. Tudi namestitev vtičnika OWASP (Excentia) v SonarQube je šla brez kakršnihkoli težav, kar je tudi za pričakovati glede na to, da gre tu za plačljivo orodje. Vtičnik OWASP prav tako najde napake v projektu WebGoat. Še posebej zanimiv je uporabniški vmesnik, saj nam vtičnik ponudi nove gradnike za prikaz najdenih varnostnih tveganj.

Dokumentiranost orodja

Obe orodji sta dokumentirani na njuni uradni spletni strani. Glede na to, da gre za specifični vtičnik za večje orodje FindBugs in platformo SonarQube, dokumentacija omenjenih vtičnikov ni preveč obsežna. Za Find Security Bugs imamo sicer na voljo tudi opis vseh varnostnih pravil, na katerih orodje deluje [106]. Za vtičnik OWASP lahko pregledujemo pravila kar v spletnem vmesniku SonarQube. Obe orodji sta primerno dokumentirani na njuni uradni spletni strani.

Enostavnost namestitve

Obe orodji se namesti zelo preprosto. Vse, kar moramo narediti, je to, da iz spleta prenesemo vtičnik in ga pri Find Securitiy Bugs shraniti na poljubno lokacijo ter ga vključiti v orodje FindBugs. Vtičnik OWASP pa le odložimo na mesto, kjer se vtičniki v SonarQube nahajajo ter ponovno zaženemo strežnik. Zatem le še nastavimo tako FindBugs kot SonarQube, da nova pravila upošteva in že lahko naredimo analizo nad svojim projektom.

Prijazen uporabniški vmesnik

FindBugs nam v okolju Eclipse na svoj privzet način pregledno prikaže najdene varnostne napake, označi vrstice v kodi in napake razvrsti v skupine. Tako jih lahko hitro pregledamo in pričnemo odpravljati. V platformi SonarQube pa lahko za naš projekt vidimo poleg odkritih varnostnih tveganj in označenih vrstic v kodi še grafične OWASP prikaze v spletnem vmesniku SonarQube. Tu imamo na voljo grafične gradnike, ki nam na atraktiven način prikažejo resnost odkritih napak. Zaradi grafičnih gradnikov prikaza odkritih napak ima OWASP vtičnik malo bolj prijazen uporabniški vmesnik kot pa Find Security Bugs. Na drugi strani pa Find Security Bugs izjemno lepo postreže z referencami za dodatna pojasnila in odpravo najdenih varnostnih tveganj, česar nismo zaznali v okolju Eclipse pri uporabi vtičnika OWASP. V spletnem uporabniškem SonarQube vmesniku so bile za vtičnik OWASP reference za odpravo napak prav tako lepo prikazane.

Učna krivulja

Oba vtičnika lahko pričnemo takoj uporabljati brez dodatnega učenja, v kolikor že poznamo orodje FindBugs in platformo SonarQube. Odkrijeta nam varnostna tveganja v naši spletni aplikaciji. Če želimo varnostna tveganja odpraviti, je zelo priporočljivo, da se poglobimo v reference za odpravo le-teh in se tudi poglobimo v dokument OWASP Top 10, kjer imamo primere uporabe pravilnega kodiranja in odprave varnostnih tveganj.

Funkcionalno pokrivanje

Oba vtičnika se uporabljata za analizo in odkrivanje varnostnih tveganj v kodi spletnih aplikacij. Na podlagi statične analize kode odkrijeta varnostna tveganja in označita vrstice v izvorni kodi, kjer se napake nahajajo. Find Security Bugs je vtičnik za orodje FindBugs in je namenjeno za odkrivanje napak v programskem jeziku Java. Vtičnik OWASP za SonarQube pa je namenjen za odkrivanje napak v več programskih jezikih: Java, PHP, JavaScript, Web, VB.NET, C#, VB6.

Ugotovitve

Obe orodji sta nedvomno dobra odločitev, če želimo v kodi spletnih javanskih aplikacij preveriti varnostna tveganja. Z njima bomo našli tudi varnostna tveganja kot jih tudi opisuje dokument OWASP Top 10. Pri tem pa je potrebno ponovno omeniti, da je namen orodij le odkrivanje varnostnih tveganj in pomoč pri analiziranju lastne kode. V kolikor želimo imeti prava celovita napredna orodja za zagotavljanje varnosti, je bolje da posežemo po zelo razširjenih in odmevnih plačljivih orodjih Coverity [65], Security AppScan (IBM) [66] in Fortify (HP) [67].

Uporaba vtičnika OWASP za Sonarqube zahteva, da imamo seveda zagnan strežnik SonarQube in postavljeno podatkovno bazo, zato moramo to upoštevati. Licenca za eno letno naročnino na vtičnik OWASP za SonarQube znaša malo manj kot 240 eur, kar za podjetja ki tržijo veliko spletnih strani, ne bi smelo biti ovira, sploh če že uporabljajo platformo SonarQube za analizo kode v svojih projektih. Na drugi strani pa je Find Security Bugs brezplačen, zato bo verjetno tudi več ljudi poseglo po njem.

V resnici Find Security Bugs ne potrebuje drugega kot le to, da ga vključimo v razvojno okolje Eclipse, kjer že imamo vtičnik FindBugs in to je vse kar potrebujemo. FindBugs orodje je enostavno pregledno in za razvijalce kode se ga priporoča že ves čas, odkar je nastalo. FindBugs je trenutno bolj priljubljeno orodje od SonarQube, saj obstaja že dalj časa in ima tudi več zadetkov, če obe orodji vpišemo v spletni brskalnik Google.

FindBugs je tudi že del distribucije SonarQube (vključimo ga lahko v profilih), kar seveda potrjuje, da je FindBugs izjemno dobro orodje. Še posebej vtičnik Find Security Bugs je tudi priporočila skupnost OWASP v svojem OWASP Top 10 dokumentu kot najbolj obetajoče brezplačno odprtokodno orodje [68].

Obe orodji sta dobri, glede na kriterije in vtise pa kot orodje za odkrivanje varnostnih tveganj v kodi spletnih aplikacij za naš nabor testnih orodij izberemo vtičnik **Find Security Bugs**.

3.4 Izbira orodja za profiliranje

Verzije vtičnikov za razvojno okolje Eclipse oziroma verzije orodij, ki smo jih analizirali so naslednje:

- VisualVM vtičnik za Eclipse (verzije 1.1.1) z VisualVM samostojno aplikacijo (verzije 1.3.8) in
- JVM Monitor vtičnik za Eclipse (verzije 3.8.1) z Eclipse vtičnikom Memory Analizer (verzije 1.4.0).

Vtisi

Pri obeh orodjih je bilo zanimivo videti, kako v realnem času prikazujeta stanje kopice, porabo pomnilnika in procesorja, število niti in grafično vse to izrisujeta. V začetku nam aplikacij nikakor ni uspelo profilirati. Razlog je bil v tem, da smo izvajali profiliranje na hitro izvršljivih programčkih, zato jih profiler ni uspel analizirati. Na forumih smo našli rešitev, da profiliranje prehitro izvršljivih programčkov v okolju Eclipse zahteva razhroščevalni način in postavitev prekinitvene točke. Zatem sta orodji delovali brez težav. Pri orodju JVM Monitor se je tudi dogajalo, da je občasno prišlo do tega, da so informacije o analizah kar izginile ter da se občasno grafi (ki prikazujejo kopico in niti v realnem času) niso pričeli prikazovati. Lahko da so se težave pojavljale zaradi uporabe vtičnika JVM Monitor v trenutno najnovejši verziji okolja Eclipse (Luna), na katerih smo izvajali analize in morda trenutno vse napakice v vtičniku še niso odpravljene. Pri orodju VisualVM je bilo tudi začutiti, da je močnejše orodje, saj je imelo tudi možnosti vključevanja dodatnih vtičnikov. Svojevrsten izziv je predstavljal tudi preizkus vtičnika "*Startup Profiler*" za VisualVM, ki je namenjen ravno temu, da lahko profiliramo hitro izvršljive aplikacije brez kakšnih trikov (npr. zaganjanja aplikacij v že omenjenem razhroščevalnem načinu). Ko smo po večkratnih neuspelih poizkušanjih ugotovili, da smo pozabili nastaviti prave VM argumente v okolju Eclipse in jih nastavili, je orodje pričelo delovati brez težav. Sicer pa je bilo obe orodji zanimivo spoznavati.

Dokumentiranost orodja

Tako VisualVM kot JVM Monitor imata ustrezno dokumentacijo na svojih uradnih spletnih straneh. Res pa je, da je dokumentacije za orodje VisualVM veliko več, saj je zanj napisano tudi veliko drugih spletnih tečajev, člankov in referenc. Že samo število zadetkov obeh orodij v iskalniku Google je po več tisočkrat večje za orodje VisualVM. Glede dokumentiranosti in primerov uporabe prednjači orodje VisualVM.

Enostavnost namestitve

Zadnjo verzijo VisualVM lahko prenesemo iz spleta in orodje uporabljamo kot samostojno aplikacijo brez potrebne namestitve. Sicer pa se orodje že nahaja v `bin` mapi, kjer imamo nameščeno Javo (`jdk`), saj je del distribucije le-te. Za še lažjo uporabo v razvojnem okolju Eclipse namestimo vtičnik, tako lahko aplikacije analiziramo neposredno iz razvojnega okolja in se na ta način tudi povežemo s samostojno aplikacijo VisualVM. Orodje JVM Monitor je še lažje namestiti, saj ga dobimo neposredno preko namestitve vtičnika. V tem pogledu zaradi popolne integracije z razvojnim okoljem malo prednjači orodje JVM Monitor.

Prijazen uporabniški vmesnik

Obe orodji imate prijazen uporabniški vmesnik, saj lahko pri obeh pridobljene podatke aplikacij pregledujemo grafično. Na voljo nam je dinamično prikazovanje grafov. VisualVM ima bolj napredno prikazovanje grafičnih izpisov ter vsebuje še orodja za analizo posnetkov, sicer pa je uporabniški vmesnik za osnovno uporabo pri obeh orodjih zadovoljiv.

Učna krivulja

Za učinkovito uporabo orodij za profiliranje je potrebno malo globlje spoznati delovanje Jave in njenih performans. Za osnovno uporabo se da hitro videti, kateri deli v kodi poberejo največ spomina ali obremenijo procesor. Glede na dobro dokumentiranost se da obe orodji hitro naučiti in ju pričeti uporabljati.

Funkcionalno pokrivanje

Tako VisualVM kot JVM monitor se uporabljata za profiliranje javanskih aplikacij, ki tečejo na lokalnem ali oddaljenem računalniku. Z obema orodjema lahko pregledujemo podatke o trenutno aktivnih javanskih nitih, naloženih razredih, porabi procesorja in porabi spomina. Orodje JVM Monitor je res narejeno z namenom enostavnih hitrih analiz posameznih aplikacij, za bolj podrobne analize pa je primerno poseči po dodatnih orodjih, kot sta. TPTP [90] in Memory Analizer [91]. VisualVM pa že vsebuje orodja za podrobne analize (npr. pregledovanje pomnilniških izpisov in analizo posnetkov stanja), poleg tega pa ga lahko še dodatno razširimo z vtičniki. Z orodjem VisualVM lahko funkcionalno pokrijemo več stvari kot pa z orodjem JVM Monitor.

Glede na podane kriterije, večjo dokumentiranost, številnejših omenjanjih na raznih forumih ter obsežnejše funkcionalnosti je izbira orodja za profiliranje v tem primeru orodje **VisualVM**.

3.5 Sodelovanje izbranih testnih orodij med sabo

Nabor testnih javanskih orodij, ki smo ga izbrali med analiziranimi orodji je naslednji:

ORODJE	KRATEK OPIS
TestNG	Ogrodje za pisanje testov enot
FindBugs	Orodje za odkrivanje pravih napak v že prevedeni kodi
PMD	Orodje za odkrivanje slabe prakse kodiranja, nepotrebne kode in preveč zapletene kode
CheckStyle	Orodje za odkrivanje napak v stilu kodiranja
Find Security Bugs	Namenski vtičnik za orodje FindBugs namenjen za odkrivanje varnostnih tveganj v kodi spletnih aplikacij.
VisualVM	Orodje za profiliranje aplikacij

Tabela 1: Končni nabor testnih orodij za razvijalca javanskih aplikacij

Vsa izbrana orodja lahko uporabljamo v okolju Eclipse in dobro sodelujejo med seboj. Vsako orodje pokriva različen del funkcionalnosti (tabela 1), zato je najbolje, da jih uporabljamo po eno naenkrat. Z ogrodjem TestNG bomo spisali teste enot, katerih kodo bomo zatem lahko preverili z ostalimi izbranimi orodji.

Pri statični analizi kode bomo uporabili FindBugs, PMD in CheckStyle. Najbolje je, da vsako od omenjenih orodij uporabljamo posamično; torej poženemo analizo z enim orodjem, pregledamo najdene napake, jih odpravimo in skrijemo napakice, za katere smo ugotovili, da jih ni potrebno odpraviti. Zatem poženemo analizo z naslednjim orodjem in tako naprej. Lahko bi sicer analizo pognali z vsemi tremi orodji, vendar se nam lahko zgodi, da se določene napake (ki jih orodja odkrijejo) pojavijo na istih vrsticah v kodi, pri čemer lahko pride do manjše čitljivosti. Primerno je tudi, da vsa tri omenjena orodja v nastavitvah "uglasimo" na za nas primeren nivo občutljivosti zaznavanja napak. Če bi bila orodja nastavljena na polno občutljivost, bi dobili ogromno odkritih napak in bi tako prišlo do pretiravanja. Veliko odkritih napak (ki so glede na kritičnost pri posameznih orodjih razvrščene nižje) je pogosto povsem nedolžnih in so stil kodiranja posameznika.

Pri statični analizi odkrivanja napak v spletnih aplikacijah bomo uporabili vtičnik Find Security Bugs, ki ga vključimo v orodje FindBugs. V tem primeru moramo v nastavitvah orodja FindBugs izključiti ostale nivoje pregledovanja in izbrati izključno varnostni nivo. Orodje FindBugs bomo tako za pregledovanje spletnih aplikacij lahko uporabljali na dva

načina: za pregledovanje običajnih napak in za pregledovanje varnostnih tveganj v kodi. Najprej na en način, zatem pa še na drug način. Po želji seveda lahko pregledamo kodo spletne aplikacije še z orodjem PMD ali s CheckStyle.

Za profiliranje uporabimo VisualVM. Orodje se iz okolja Eclipse z nastavitvami le požene in nobeno od prejšnjih omenjenih orodij nanj nima negativnega vpliva.

Vsa izbrana orodja torej lepo sodelujejo med sabo in nam bodo lahko v veliko pomoč.

4. Zaključek

Pri raziskovanju in pisanju diplomske naloge smo se naučili veliko stvari. Naredili smo pregled in spoznali različna zelo dobra orodja za različne vrste testiranja. Kot rezultat raziskave smo dobili nabor testnih orodij (tabela 1), ki so po našem mnenju zelo primerna za nekega programerja, ki razvija aplikacije v programskem jeziku Java in pri svojem delu uporablja razvojno okolje Eclipse ter ima podobne kriterije za izbiro testnih orodij, kot smo jih navedli mi.

Orodja iz izbranega nabora testnih orodij so v resnici izjemno dobra za zagotavljanje kakovosti pri pisanju kode. Številni programerji jih uporabljajo pri svojem delu po celem svetu. V večini forumov so ta orodja omenjena kot prva izbira, v kolikor želimo pri svojem delu uporabljati brezplačna odprtokodna orodja za nadzor kakovosti ali varnosti kodiranja. Z uporabo izbranih orodij si bomo lahko občutno olajšali odpravo napak in izboljšali kakovost ter varnostni vidik v svoji izvorni kodi. Če bomo kdaj pri svojem delu potrebovali naprednejša orodja, lahko posežemo tudi po drugih (že omenjenih) odličnih plačljivih komercialnih orodjih, ki jih že vrsto let uporabljajo številna razvojna podjetja po svetu in si s tem zagotovijo kakovostne in varne produkte.

V diplomski nalogi smo se omejili na posamezne dele testiranja v programskem jeziku Java: testiranje enot, testiranje kakovosti izvorne kode, testiranje varnosti v kodi v spletnih aplikacijah in na profiliranje aplikacij. Obstajajo še drugi vidiki testiranja, kar bi lahko raziskali. Že na nižjem nivoju bi lahko pregledali še orodja za analizo pokritosti kode s testi enot (npr. JaCoCo [47] in Cobertura [48]). Lahko bi pregledali orodja za testiranje modulov, integracijsko testiranje, sistemsko testiranje, sprejemno testiranje. Za testiranje grafičnih vmesnikov bi lahko pregledali orodja (kot so Ranorex [107], Selenium [108], WindMill [109]). V spletnih aplikacijah bi lahko raziskali še nivo varnosti z orodji, ki omogočajo prestrezanje zahtevkov (*ang. request*) in odgovorov (*ang. response*), ki jih posreduje spletni brskalnik (npr. WebScarab [110], OWASP's Zed Attack Proxy ("ZAP") Project [111]).

Seznam slik

Seznam tabel

Literatura

[1] (2014) Unit Test by Martin Fowler. Dostopno na: http://martinfowler.com/bliki/UnitTest.html

[2] (2014) Official home page of JUnit framework. Dostopno na: http://junit.org/

[3] (2014) Official TestNG home page. Dostopno na: http://testng.org/doc/

[4] (2014) XUnit by Martin Fowler. Dostopno na: http://www.martinfowler.com/bliki/Xunit.html

[5] (2014) Favorited Java libraries analysis of GitHub projects in 2013. Dostopno na: http://www.takipiblog.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[6] (2014) Annotations Tutorial by Oracle. Dostopno na: http://docs.oracle.com/javase/tutorial/java/annotations/index.html

[7] (2014) Unit Testing with JUnit 4.x – Tutorial (Lars Vogel). Dostopno na: http://www.vogella.com/tutorials/JUnit/article.html

[8] (2014) JUnit Class Assert API. Dostopno na: http://junit.sourceforge.net/javadoc/org/junit/Assert.html

[9] (2014) Data Driven Testing. Dostopno na: http://en.wikipedia.org/wiki/Data-driven_testing

[10] (2014) JUnit Parametrized Tests. Dostopno na https://github.com/junit-team/junit/wiki/Parameterized-tests

[11] (2014) JUnit Exceptions Testing. Dostopno na: https://github.com/junit-team/junit/wiki/Exception-testing

[12] (2014) JUnit Video Tutorial by Jirka Pinkas. Dostopno na: http://www.javavids.com/tutorial/junit.html

[13] (2014) TestNG - Quick Guide by tutorialspoint. Dostopno na: http://www.tutorialspoint.com/testng/testng_quick_guide.htm

[14] (2014) Official Apache Ant Home Page. Dostopno na: http://ant.apache.org

[15] (2014) Official Maven Home Page. Dostopno na: http://maven.apache.org

[16] (2014) TestNG Documentation and Tutorial. Dostopno na: http://testng.org/doc/documentation-main.html

[17] (2014) TestNG Annotations. Dostopno na: http://testng.org/doc/documentation-main.html#annotations

[18] (2014) TestNG Configuration file testng.xml. Dostopno na: http://testng.org/doc/documentation-main.html#testng-xml

[19] (2014) TestNG Parameterized Test. Dostopno na: http://www.tutorialspoint.com/testng/testng_parameterized_test.htm

[20] (2014) Static Code Analysis. Dostopno na: http://www.mathworks.com/discovery/static-code-analysis.html
[21] (2014) Industrial Perspective on Static Analysis. Dostopno na: https://web.archive.org/web/20110927010304/http://www.ida.liu.se/~TDDC90/papers/industrial95.pdf,
[22] (2014) FindBugs Official Site. Dostopno na: http://findbugs.sourceforge.net/
[23] (2014) PMD Official Page. Dostopno na: http://pmd.sourceforge.net/
[24] (2014) Checkstyle Official Page. Dostopno na: http://checkstyle.sourceforge.net/
[25] (2014) SonarQube Official Home Page. Dostopno na: http://www.sonarqube.org/
[26] (2014) FindBugs Part 1: Improve the quality of your code. Dostopno na: http://www.ibm.com/developerworks/java/library/j-findbug1/
[27] (2014) FindBugs Bug Descriptions. Dostopno na: http://findbugs.sourceforge.net/bugDescriptions.html
[28] (2014) Why String is Immutable in Java? Dostopno na: http://java.dzone.com/articles/why-string-immutable-java
[29] (2014) Using Static Analysis For Software Defect Detection. Dostopno na: https://www.youtube.com/watch?v=GgK20Yv9QRk
[30] (2014) TestNG use with Eclipse Tutorial. Dostopno na: http://testng.org/doc/eclipse.html
[31] (2014) Joshua Bloch. Dostopno na: http://en.wikipedia.org/wiki/Joshua_Bloch
[32] (2014) FindBugs for Eclipse. Dostopno na: http://www.vogella.com/tutorials/Findbugs/article.html
[33] (2014) Mistakes that matter, JavaOne, 2009. Dostopno na: http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf
[34] (2014) Introduction to PMD. Dostopno na: http://pmd.sourceforge.net/pmd-5.1.2/
[35] (2014) PMD List of Rulsets. Dostopno na: http://pmd.sourceforge.net/pmd-5.1.2/rules/index.html
[36] (2014) XPath (Wikipedia) http://en.wikipedia.org/wiki/XPath
[37] (2014) Abstract Syntax Tree. Dostopno na: http://en.wikipedia.org/wiki/Abstract_Syntax_Tree
[38] (2014) How to write PMD rule. Dostopno na: http://pmd.sourceforge.net/pmd-5.1.0/howtowritearule.html
[39] (2014) Eclipse Checkstyle Tutorial. Dostopno na: http://www.javatips.net/blog/2013/07/eclipse-checkstyle-tutorial

[40] (2014) Using the Checkstyle Eclipse plug-in - Tutorial. Dostopno na: http://www.vogella.com/tutorials/Checkstyle/article.html
[41] (2014) ANother Tool For Language Definition. Dostopno na: http://www.antlr.org/
[42] (2014) Writing Own Checks for CheckStyle. Dostopno na: http://checkstyle.sourceforge.net/writingchecks.html
[43] (2014) Gradle Official Home Page. Dostopno na: http://www.gradle.org/
[44] (2014) Atlassian Bamboo Official Home Page. Dostopno na: https://www.atlassian.com/software/bamboo
[45] (2014) Jira Official Home Page. Dostopno na: https://www.atlassian.com/software/jira
[46] (2014) Methods and Tools SonarQube. Dostopno na: http://www.methodsandtools.com/PDF/mt201001.pdf
[47] (2014) JaCoCo (Java Code Coverage Library) Official Home Page. Dostopno na: http://www.eclemma.org/jacoco/
[48] (2014) Cobertura Offical Home Page. Dostopno na: http://cobertura.github.io/cobertura/
[49] (2014) Clover Official Home Page. Dostopno na: https://www.atlassian.com/software/clover/overview
[50] (2014) SonarQube is deprecating PMD and CheckStyle Rules. Dostopno na: http://www.sonarqube.org/already-158-checkstyle-and-pmd-rules-deprecated-by-sonarqube-java-rules/
[51] (2014) SonarQube native Language Recognizer SSLR. Dostopno na: http://docs.codehaus.org/display/SONAR/SSLR
[52] (2014) Sonar Java Plugin Change Log. Dostopno na: http://docs.codehaus.org/display/SONAR/Java+Plugin
[53] (2014) SonarQube in Eclipse. Dostopno na: http://docs.codehaus.org/display/SONAR/SonarQube+in+Eclipse
[54] (2014) SonarQube Install. Dostopno na: http://docs.codehaus.org/display/SONAR/Installing
[55] (2014) SourceMeter Plugin for SonarQube. Dostopno na: https://frontendart.com/products/sourcemeter/
[56] (2014) FaultHunter plugin for SonaQube for better code analysis. Dostopno na: http://www.slideshare.net/FrontEndART/faulthunter-workshop
[57] (2014) Abstract semantic graph. Dostopno na: http://en.wikipedia.org/wiki/Abstract_semantic_graph
[58] (2014) OWASP Top 10. Dostopno na: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[59] (2014) OWASP (Open Web Application Security Project). Dostopno na: https://www.owasp.org/index.php/Main_Page

[60] (2014) OWASP Top 10 PDF Document. Dostopno na: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf
[61] (2014) Informacijska Varnost (Predavanje podjetja ASTEC). Dostopno na: http://lusy.fri.uni-lj.si/files/courses/fri-courses/kpov/Predavanja/ppt/Vabljena/Borut_Znidar-1314_InformacijskaVarnost.zip%20v0.4.pdf
[62] (2014) Find Security Bugs official home page. Dostopno na: http://h3xstream.github.io/find-sec-bugs/
[63] (2014) Google CodePro Analytix. Dostopno na: https://developers.google.com/java-dev-tools/codepro/doc/
[64] (2014) OWASP LAPSE Project. Dostopno na: https://www.owasp.org/index.php/OWASP_LAPSE_Project
[65] (2014) Coverity Official Home Page. Dostopno na: http://www.coverity.com/
[66] (2014) Security AppScan (IBM) Official Home Page. Dostopno na: http://www-03.ibm.com/software/products/sl/appscan
[67] (2014) Fortify (HP) Official Home Page. Dostopno na: http://www8.hp.com/us/en/software-solutions/application-security/index.html
[68] (2014) Find Security Bugs OWASP Recomendation. Dostopno na: https://www.owasp.org/index.php/Top_10_2013-What%27s_Next_for_Verifiers
[69] (2014) OWASP (excentia) Plugin for SonarQube Official Page. Dostopno na: http://www.excentia.es/plugins/owasp/descargar_en.html
[70] (2014) OWASP SonarQube Project Official Page. Dostopno na: https://www.owasp.org/index.php/OWASP_SonarQube_Project
[71] (2014) OWASP WebGoat Project. Dostopno na: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
[72] (2014) Find Security Bugs patterns. Dostopno na: http://h3xstream.github.io/find-sec-bugs/bugs.htm
[73] (2014) OWASP (Excentia) Plugin for SonarQube Features. Dostopno na: http://www.excentia.es/plugins/owasp/caracteristicas_en.html
[74] (2014) XML file for defined rules for OWASP (Excentia) Plugin. Dostopno na: http://www.excentia.es/plugins/owasp/OwaspTop.xml
[75] (2014) Usage for OWASP (Excentia) Plugin. Dostopno na: http://www.excentia.es/plugins/owasp/utilizacion_en.html
[76] (2014) Memory in Java. Dostopno na: http://www.vogella.com/tutorials/EclipseMemoryAnalyzer/article.html
[77] (2014) VisualVM Official Home Page. Dostopno na: http://visualvm.java.net/
[78] (2014) JProfiler Official Home Page. Dostopno na: https://www.ej-technologies.com/products/jprofiler/overview.html
[79] (2014) YourKit Official Home Page. Dostopno na: http://www.yourkit.com/

[80] (2014) Introduction to VisualVM. Dostopno na: http://visualvm.java.net/intro.html
[81] (2014) VisualVM included in JDK. Dostopno na: http://www.semanticmetadata.net/2008/07/14/visual-vm-is-part-of-java-16-update-7/
[82] (2014) VisualVM Profiler. Dostopno na: http://visualvm.java.net/profiler.html
[83] (2014) Using Eclipse launcher for VisualVM. Dostopno na: http://visualvm.java.net/eclipse-launcher.html
[84] (2014) Getting started with VisualVM. Dostopno na: http://visualvm.java.net/gettingstarted.html
[85] (2014) Working with Snapshots With VisualVM. Dostopno na: http://visualvm.java.net/snapshots.html
[86] (2014) Get useful plugins for VisalVM. Dostopno na: http://visualvm.java.net/plugins.html
[87] (2014) Startup Profiler plugin for VisualVM. Dostopno na: http://visualvm.java.net/startupprofiler.html
[88] (2014) JVM Monitor Official Home page. Dostopno na: http://www.jvmmonitor.org/
[89] (2014) JVM Monitor Documentation. Dostopno na: http://www.jvmmonitor.org/doc/index.html
[90] (2014) TPTP. Dostopno na: http://www.eclipse.org/tptp/
[91] (2014) Memory Analyzer (MAT) http://www.eclipse.org/mat/
[92] (2014) JVM Monitor Documentation. Dostopno na: http://www.jvmmonitor.org/doc/index.html
[93] (2014) HPROF File. Dostopno na: http://docwiki.cisco.com/wiki/Java_HProf_Files
[94] (2014) Memory Analyzer. Dostopno na: http://www.eclipse.org/mat/
[95] *JUnit in Action, Second Edition* (PDF Book), https://docs.google.com/file/d/0Bz8xHhAV9yYbNTE4NTIzOTQtMjIxOC00NTdmLWFhNmEtYTQ1YzllN2UxM2Jm/edit
[96] Next Generation Java Testing (PDF Book) http://www.wangyuxiong.com/wp-content/uploads/downloads/2013/02/Next-Generation-Java-Testing.-TestNG-and-Advanced-Concepts.pdf
[97] (2014) TestNG video tutorial by Learn With Video Tutorials. Dostopno na: http://www.learn-with-video-tutorials.com/testng-tutorial-video
[98] (2014) Feature comparison between JUnit and TestNG (Mkyong) . Dostopno na: http://www.mkyong.com/unittest/junit-4-vs-testng-comparison
[99] (2014) JUnit Categories. Dostopno na: https://github.com/junit-team/junit/wiki/Categories

[100] (2014) JUnit Assumptions. Dostopno na: https://github.com/junit-team/junit/wiki/Assumptions-with-assume
[101] (2014) JUnit Parametrized Tests. Dostopno na: https://github.com/junit-team/junit/wiki/Parameterized-tests
[102] (2014) Usage comparison between JUnit and TestNG (scratch pad). Dostopno na: http://scratchpad101.com/2011/08/07/testng-or-junit/
[103] PMD Book on Amazon http://www.amazon.com/PMD-Applied/dp/0976221411
[104] SonarQube Book on Amazon http://www.amazon.com/SonarQube-Action-G-Ann-Campbell/dp/1617290955
[105] (2014) Award for SonarQube. Dostopno na: http://www.drdobbs.com/tools/jolt-productivity-award-2-testing-and-de/228400216
[106] (2014) Find Security Bugs (Bugs descriptions). Dostopno na: http://h3xstream.github.io/find-sec-bugs/bugs.htm
[107] (2014) Ranorex Official Home Page. Dostopno na: http://www.ranorex.com/
[108] (2014) Selenium Official Home Page. Dostopno na: http://www.seleniumhq.org/
[109] (2014) Windmill. Dostopno na: http://www.getwindmill.com/
[110] (2014) OWASP WebScarab Project. Dostopno na: https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[111] (2014) OWASP Zed Attack Proxy Project. Dostopno na: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project