Mutationstestning av Java-mjukvara

Emil Aura

33417

Kandidatavhandling

Handledare: Dragos Truscan & Jerker Björkqvist

Datateknik

Institutionen för informationsteknik

Åbo Akademi

1 april 2012

Sammandrag

Dagens samhälle har höga krav på fungerande mjukvara, vilket medför att testning utgör en mycket viktig del av utvecklingsprocessen. För att säkerställa att denna mjukvara fungerar enligt givna direktiv är tester högkvalitativa tester ett måste. Denna avhandling kommer att behandla mutationstestning, vilket är ett sätt att utvärdera testernas kvalitet. Första delen koncentrerar sig på teorin bakom mutationstestning samt allmänna principer, medan den andra delen tar upp mutationstestning av Java-mjukvara. Slutligen görs en jämförelse av diverse verktyg som finns tillgängliga för att underlätta mutationstestning av Java-mjukvara

Den teoretiska delen inleds med en presentation av grundläggande begrepp inom ramen för mutationstestning, så som mutationsoperatorer, ekvivalenta mutanter och mutationsvärde, för att sedan gå vidare till själva mutationsprocessen samt Javaspecifik teori. Teorin beskrivs av kodexempel skrivna i Java.

Nyckelord: *Mutationstestning, mutationsoperatorer, Java, MuJava, Javalance, Judy*

Innehåll

Sammandrag **II**

Innehåll **III**

Figurer **IV**

Tabeller **V**

1 Introduktion **1**

2 Bakgrund **2**

2.1 Testning 2

2.1.1 White-box-testning 2

2.1.2 Testningsnivåer 3

2.2 Java 5

2.2.1 JUnit 5

3 Mutationstestning **7**

3.1 Mutationsoperatorer 7

3.2 Ekvivalenta mutanter 8

3.3 Mutationsvärde 9

3.4 Processen 9

4 Mutationstestning i Java **11**

4.1 Mutationsoperatorer 12

4.2 Verktyg för mutationstestning 12

4.2.1 MuJava och MuClipse 13

4.2.2 Javalanche 15

4.2.3 Judy 16

5 Avslutning **18**

Litteratur **19**

A Bilaga 1 **21**

Figurer

2.1 V modellen för testning 4

3.1 Mutationsprocessen 10

4.1 Indelning av publikationer angående mutationstestning utgående från programmeringspåk de behandlat. 11

A.1 MuClipse GUI för generering av mutanter 21

A.2 Lista över genererade mutanter i MuClipse GUI 22

A.3 Jämförelse av mutant och original i MuClipse 22

A.4 MuClipse GUI exekvering av mutanterna mot testfallen 23

Tabeller

2.1 Enhetstest i JUnit 6

3.1 Mutationsoperatorer på metodnivå 8

3.2 Sex olika mutanter 9

4.1 Mutationsoperatorer på klassnivå 13

1 Introduktion

På Software QA and Testing Resource Centers hemsida [1] listas några färska exempel på hur en liten mjukvarubugg kunnat få stora mjukvarusystem att fungera inkorrekt. På listan hittas bl.a betygssystem som gett felaktiga vitsord åt två miljoner studerande och massor av system som gett ut hemlig information om privatpersoner. Några av dessa buggar skulle antagligen ha varit möjliga att upptäcka i ett tidigare skede och på så viss undvika katastrofer, om man testat dem bättre.

Utvecklingen går visserligen mot det bättre. I takt med att mjukvarusystemen blir allt större och mer komplexa, tvingas även mjukvaruprocesserna att satsa mer på testning och kvalitetssäkring. Ett av de stora problemen är att bestämma hur mycket testning som krävs. Vanligtvis mäts mängden testning i täckningsgrad, det vill säga hur stor del av koden som körs av testerna. Utvecklarna bestämmer sedan ett tröskelvärde och slutar testa när detta värde är uppnått. Mutationstestning bidrar med ett koncept som skiljer sig helt från det vanliga sättet att mäta täckningsgraden. Genom att införa fel i programkoden av programmet som skall testas kan testernas förmåga att märka av felaktigt beteende testas. Tanken är att testerna skall försöka upptäcka så många av dessa *mutanter* av originalkoden som möjligt och utgående från detta antal beräkna en täckningsgrad.

Målet med denna avhandling är inte att utveckla något system för mutationstestning utan att ge läsaren en inblick i vad mutationstestning är och hur det kan tillämpas för att framställa högkvalitativa tester samt att presentera några av de verktyg som redan finns till förfogande för att utföra mutationstestning.

2 Bakgrund

2.1 Testning

Dagens samhälle är beroende av programvara, allt ifrån små system i konsumentelektronik till stora system som bygger upp samhället så som finanssystem, bokningssystem och trafiknät. Ju större ansvar dessa system har desto viktigare är det att de fungerar felfritt. I och med att dessa system blir mer avancerade ökar betydelsen av programvarutestning och testning är därför en mycket viktig del i utvecklingen av programvara. För att underlätta testning finns olika metoder för hur det utförs. Testen kan delas upp på olika nivåer och ha olika ansvarsområden.[2]

Det är omöjligt att genom testning försäkra sig om att ett program är helt felfritt. Antalet möjliga indata samt utdata är mycket stort, och så är även antalet möjliga vägar genom programmet. Att försöka testa dem alla vore extrem resurskrävande och priset för testning ökar därav exponentiellt mot mängden testning, samtidigt som antalet fixade buggar minskar. En gyllene medelväg bör därför hittas.[3]

2.1.1 White-box-testning

White-box-testning har som mål att testa programmets interna struktur genom att t.ex försöka få varje påstående i programmet att köras minst en gång. Detta är känt som uttömmande testning av vägar och kan liknas vid uttömmande testning av indata vid black-box-testning, som försöker gå igenom all potentiell indata och verifierar utdata.

Att gå igenom alla möjliga vägar genom programmet kan dessutom inte ses som ett tillräckligt krav för att programmet skall vara fullständigt testat. För det första kan antalet möjliga vägar vara nästintill oändligt många och för det andra kan ett program som alla vägar testats i fortfarande vara fullt av fel. Till detta finns tre förklaringar. Programmet måste inte möta sina specifikationer, det kan med andra ord vara ett fullkomligt felfritt program, men som dessvärre utför fel uppgift. Vidare kan vägar genom programmet helt saknas eftersom white-box-testning endast testar befintliga vägar men söker inte efter obefintliga. Slutligen så kan programmet ha datakänsliga fel som uttömmande testning av vägar inte upptäcker, exempelvis följande java program.[4]

```
1 if (a-b < c)
2 System.out.println("a-b < c");
```

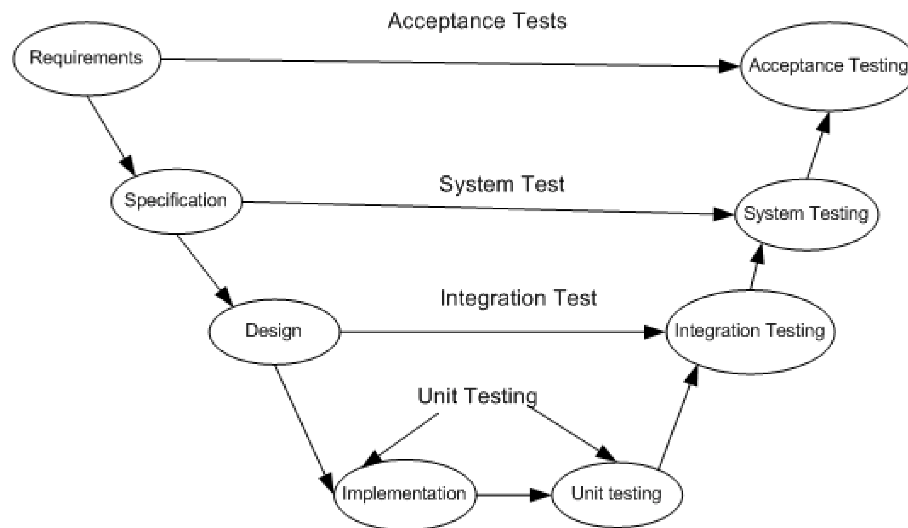
Programmet skall jämföra två värden och se ifall skillnaden mellan dessa är mindre än ett förut bestämt värde. För att detta skall göras korrekt borde givetvis det absoluta värdet av de två talens differens jämföras. Detta är ett problem som uttömmande testning av vägar kanske inte observerar.[4]

2.1.2 Testningsnivåer

Testningsnivåer kan även delas upp enligt mjukvaruutvecklings olika aktiviteter: krav och specifikationer, design eller själva källkoden. Denna uppdelning kan ses ur figur 2.1. Det rekommenderas att skriva tester för respektive aktivitet i aktivitetens utvecklingsskede, även om programmet inte kommer att vara exekverbart förrän i implementationsskedet. Detta görs för att fel som annars skulle gå obemärkta kan uppmärksammas under planeringen av testerna. [2]

Enhets- och modultestning

Enhetstestning är den lägsta nivån av testning och den har som uppgift att testa de minsta enheterna av implementeringen, i Java betyder detta klasserna. Användaren verifierar funktionaliteten av dessa enheterna utan att beakta resten av systemet.[6, 2] Med hjälp av verktyg såsom *JUnit* kan testerna packas med med



Figur 2.1: V modellen visar vilka testningsnivåer som motsvarar vilken aktivitet. [5]

källkoden och enhetstestning sköts därför oftast av samma person som skrivit koden.[2]

Integrationstestning

Integrationstestning baserar sig på enhetstestning och därför antas att enhetstestning är gjord när integrationstestningen tar vid. Den fokuserar på att testa interaktioner mellan enheterna testade av enhetstestningen och görs oftast i samband med implementering av koden i programmerarens utvecklingsmiljö.[7]

Systemtestning

Systemtestning utförs efter att integrationstestningen slutförts. Testerna skrivs dock oftast redan i ett tidigt skede av utvecklingsprocessen, eftersom de baserar sig på systemets kravspecifikationer, vilka oftast är tillgängliga i ett tidigt skede. Systemtestning skiljer sig från integrationstestning i det att det ser på funktionaliteten av hela systemet, till skillnad från integrationstestning som testar strukturen.[7]

Acceptanstestning

Acceptanstestning undersöker ifall systemet inte motsvarar klientens krav och planeringen av acceptanstest utförs helst av eller i samarbete med slutanvändaren. Testerna är mycket användarfokuserade och utförs gärna i en miljö liknande distributionsmiljön. [7]

2.2 Java

Javas designmål att vara: enkelt, objektorienterat, distribuerat, robust, dynamiskt, portabelt, effektivt, säkert, stöda multitrådning och vara arkitektur neutralt [8] har gjort det till ett av de mest använda programmeringsspråken nuförtiden. Därför kommer denna avhandling att koncentrera sig på Java, även om testning och mutationstestning likaväl kan tillämpas på andra programmeringsspråk.

2.2.1 JUnit

Ett av de mest använda ramverken för testning av Javaapplikationer är JUnit som är ett ramverk med öppen källkod gjort för att automatisera upprepad testning. JUnit är skrivet av Erich Gamma and Kent Beck, vilka även var med i Gang of Four som skrev boken “Design Patterns”. Junit används i stor utsträckning inom industrin och kan användas direkt från kommandotolken eller via en integrerad utvecklingsmiljö så som Eclipse.[9]

JUnit fungerar med hjälp av testmetoder vars uppgift är att evaluera villkor (assertions) vars resultat sedan presenteras för användaren. Tabell 2.1 visar ett exempel på ett enhetstest för koden i tabell 3.2. Raden “*assertEquals (2, **new** Calc().Min(2,3));*” fungerar som testmetod och dess uppgift är att verifiera utdata från funktionen *Min()* mot det väntade resultatet, i detta fall 2. Ifall dessa två är lika är testet godkänt och programmet fungerar enligt specifikationen. [9]

Test	Klass

<pre>1 import org.junit.Test
2 import static org.junit.Assert.*;
3 public class CalcTest{
4 @Test
5 public void testMin(){
6   assertEquals (2,
7   new Calc().Min(2,3));
8   }
9 }</pre>
	<pre>1 public class Calc{
	2 static public int Min(int a, int b){
	3 int minVal;
	4 minVal=a;
	5 if(b<a){
	6 minVal=b;
	7 }
	8 return minVal
	9 }
	10 }</pre>

Tabell 2.1: Enhetstest i JUnit.

3 Mutationstestning

Principen för mutationstestning är att genom att införa fel i programkoden producera mutanter av denna. Dessa mutanter körs sedan mot testsviten, vars förmåga att urskilja förändringarna i koden då testas. Principen nämndes för första gången år 1971 i Richard Liptons' vetenskapliga publikation "Fault Diagnosis of Computer Programs".Det dröjde dock ända till slutet av 1970 talet förrän det började forskas mer inom området och DeMillo, Lipton och Saywards rapport[10] brukar ses som den första inflytelserika publikationen. Mutationstestning har sedan dess tillämpats på både källkod (Programmutation) samt mjukvaruspecifikationer (Specifikationsmutation). Programmutation hör till white-box-testning medan specifikationsmutation hör till black-box-testning.[11] Denna avhandling kommer att koncentrera sig på programmutation.

3.1 Mutationsoperatorer

Mutationsoperatorer är regler som beskriver syntaktiska förändringar i koden och som har som avsikt att imitera ofta förekommande fel och missar från programmeraren. Dessa förändringar åstadkoms genom insättnings-, borttagnings- och bytes-operatorer. Ett effektivt test skall märka av dessa små förändringar eftersom de kan innebär radikala förändringar i programmets funktionalitet. Figur 3.2 visar ett java program med sex muterade rader där varje muterad rad motsvarar ett nytt program. Mutanterna 1, 3 och 5 byter ut en variabelreferens mot en annan medan mutant 2 byter ut en relationsoperator. Mutanterna 4 och 6 är specialoperatorer som inte kommer att beaktas i denna avhandling.[12, 11]

Mutationsoperatorerna måste designas noga för att vara effektiva. Väldefinierade mutationsoperatorer kan därmed resultera i mycket effektiv testning, medan dåliga endast resulterar i ineffektiva tester. De derivat av det ursprungliga programmet som uppstår efter att mutationsoperatorerna har tillämpats kallas för mutanter av det ursprungliga programmet. Två problem som måste beaktas vid design av mutationsoperatorer är: ifall fler än en operator kan tillämpas per mutant och ifall varje möjlighet att tillämpa operatorn skall tas? Alltså skall en mutant innehålla endast en förändring eller skall flera tillåtas? Forskning har visat att det effektivaste sättet är att använda endast en operator per mutant[2]

Tabell 3.1 listar några av de mutationsoperatorer som mutationssystemet muJava använder. Dessa är grundläggande mutationsoperatorerna som används både inom objektorienterad och procedurell mutationstestning.

Mutationsoperator	**Beskrivning**
AOR	Ersättning av aritmetisk operator
AOD	Radering av aritmetisk operator
AOI	Insättning av aritmetisk operator
ROR	Ersättning av relationsoperator
COR	Ersättning av villkorsoperator
COD	Radering av villkorsoperator
COI	Insättning av villkorsoperator
SOR	Ersättning av skiftoperator
LOR	Ersättning av logisk operator
LOD	Radering av logisk operator
LOI	Insättning av logisk operator
ASR	Ersättning av tilldelningsoperator

Tabell 3.1: Metodnivå-specifika mutationsoperatorer. [12]

3.2 Ekvivalenta mutanter

Ett av de största problemen med mutationstestning är mutanter som är funktionellt ekvivalenta med ursprungsprogrammet. Detta innebär att mutanten i alla scenarion kommer att ge samma utdata som det ursprungliga programmet. Mutant nummer 3 i Figur 3.2 är ett exempel på en ekvivalent mutant. Intuitivt har **A** och **minVal** samma värde på mutationsplatsen så ett byte gör ingen skillnad. Att försöka bevisa att en mutant är ekvivalent är oftast ett oavgörbart problem även om vissa ekvivalenta mutanter kan bevisas med hjälp av mjukvara.[2]

Ursprunglig metod	**Metoden med mutanter**
int Min (int A, int B)	int Min (int A, int B)
{	{
int minVal;	int minVal;
minVal = A:	minVal = A;
if(B<A)	Δ 1 minVal = **B**;
{	if(B<A)
minVal = B;	Δ 2 if (B>A)
}	Δ 3 if (B < **minVal**)
return (minBal);	{
} //end Min	minVal = B;
	Δ 4 **Bomb();**
	Δ 5 minVal =**A**;
	Δ 6 minVal = **failOnZero (B)**;
	}
	return (minVal);
	} // end Min

Tabell 3.2: Sex olika mutanter [2]

3.3 Mutationsvärde

För att kunna ge ett mätbart resultat över kvaliteten på testerna räknas ett *mutationsvärde* ut. Detta värde baserar sig på antalet döda mutanter dividerat med alla mutanter förutom de som visar sig vara *ekvivalenta*, formell3.1.[13] Med *döda mutanter* avses mutanter vars utdata skiljer sig från ursprungsprogrammet.[2, 14, 12]

$$\text{mutationsvärde} = \frac{|\text{döda mutanter}|}{|\text{totala antalet mutanter}| - |\text{ekvivalenta mutanter}|} \tag{3.1}$$

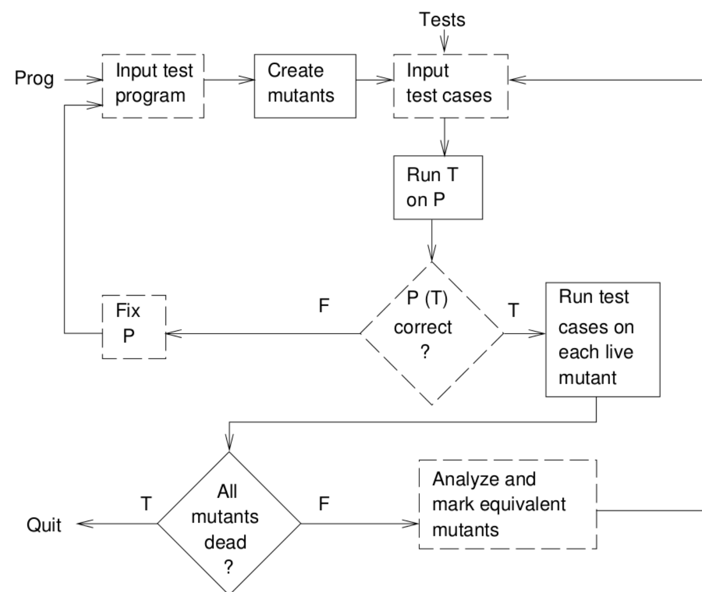
Ett mutationsvärde på 1,0 innebär att testerna lyckats döda alla mutanter och uppsättningen av tester anses vara adekvat. Att sträva efter ett mutationsvärde på 1,0 är dock inte rimligt, eftersom testerna mycket sällan lyckas döda alla mutanter. Därför bör ett tröskelvärde, som fungerar som det minsta godkända värdet, definieras.[2]

3.4 Processen

Mutationsprocessen mäter kvaliteten på testerna, själva testningen är en sidoeffekt. I praktiken kommer programmet ändå att testas grundligt. Ifall programmet innehåller ett fel, finns det oftast några mutanter som endast kan dödas av samma test som uppmärksammar detta fel. [2, 14]

Figur 3.1 illustrerar mutationsprocessen. De heldragna rektanglarna represente-

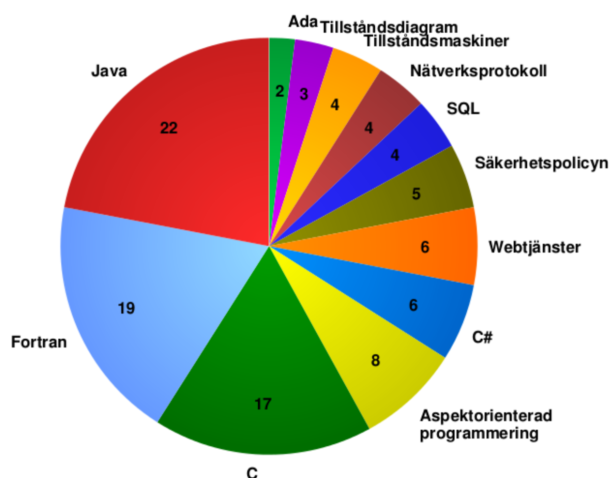
rar steg som sköts av mutationssystemet och är därmed automatiserade, medan de streckade rektanglarna fodrar mänsklig aktivitet och är därmed manuella. Systemet tar som indata koden för ursprungsprogrammet och genererar en mutant per mutationsoperator. Till följande tas testsviten som indata och varenda testfall körs mot originalkoden varefter utvecklaren verifierar att utdata är korrekt. Ifall utdatan visar sig vara felaktig har en bugg hittats. Utvecklaren måste då förbättra koden och sedan köra om testfallen tills utdata är korrekt. När så är fallet kan processen fortgå genom att exekvera varenda testfall mot mutanterna. De mutanter vars utdata avviker från ursprungsprogrammet markeras som *döda*. De döda mutanterna kommer inte att exekveras mot senare testfall. Vi detta skede räknas mutationsvärdet ut och ifall det av utvecklaren definierade tröskelvärdet uppfylles kan processen avslutas. I annat fall måste de fortfarande levande mutanterna analyseras och eventuella ekvivalenta mutanter avlägsnas. Utvecklaren kan nu även förbättra testerna varefter processen börjar om igen med att de nya testerna körs mot originalprogrammet och dess utdata verifieras. [2, 14]



Figur 3.1: Processen för mutationstestning. Heldragna boxar representerar automatiserade steg och streckade manuella. [14]

4 Mutationstestning i Java

Ur figur 4.1 ses att största delen av den forskning som gjorts angående mutationstestning har gjorts i området för programmutation och cirka 50 procent i Java, C och Fortran. Java har numera störst andel eftersom Fortrans popularitet avtagit avsevärt. Eftersom effektiviteten hos mutationstestning baserar sig på de



Figur 4.1: Indelning av publikationer angående mutationstestning utgående från de programmeringspåk de behandlar. [11]

fel som mutationsoperatorerna representerar, är mutationsoperatorernas kvalitet mycket viktigt vid mutationstestningen. Vanligtvis tillämpas mutationstestning på procedurella program, vilka skiljer sig mycket från de objektorienterade. Metodernas kroppar är oftast betydligt kortare vid objektorienterade program vilket leder till ökad interaktion sinsemellan. Objektorienterad programmering använder sig dessutom, till skillnad från procedurella program, av inkapsling, arv, och polymorfism. [12, 11]

4.1 Mutationsoperatorer

På grund av Javas annorlunda struktur räcker inte de mutationsoperatorer till, som designats för procedurella språk. Därför har nya operatorer skapats för att kunna utnyttja Javas funktioner till fullo.[12] Dessa operatorer kan läsas ur tabell 4.1. Från tabellen ses även att dessa delas in i fyra olika grupper: inkapsing, arv, polymorfism, och Java-specifika beroende på vilken språklig funktion de behandlar.

Åtkomstnivåerna för variabler och metoder ignoreras ofta vid designtillfället och kan därmed leda till problem i implementationsskedet. Denna ignorans kan förklaras med dålig kunskap om semantiken för olika åtkomstnivåer. Felaktig användning av inkapsling behöver inte direkt leda till problem, men kan göra så när klassen integreras med andra klasser. MuJava testar detta genom att ändra åtkomsttypen för variabler, för att styra utvecklaren att skriva korrekt kod.[15]
Arv är en mycket kraftfull abstraktionsmekanism, men felaktig användning kan leda till många fel. MuJava behandlar flera olika aspekter av arv: variabelskuggning, metod-överskuggning, användningen an nyckelordet *super* och definition av konstruktorer. Detta görs bland annat genom att lägga till eller radera *super*.[15]
Polymorfism medför att ett objekts typ kan vara variera. Dess typ kan vara alla de typer som dess subklasser är. Därför måste alla möjliga typer testas och detta görs genom att ändra på typkonverteringar mellan över och underordnade klasser.[15]
De Javaspecifika mutationsoperatorerna försöker härma fel som programmerare ofta gör och som är typiska för java. Exempelvis fel användning av nyckelorden *static* och *this*.[15]

4.2 Verktyg för mutationstestning

Det finns ett flertal verktyg som förenklar mutationsverktyg i java, varav de flesta är små universitetsprojekt. Denna avhandling har behandlat tre av dessa: MuJava, Javalanche och Judy.

Funktion	Mutationsoperator	Beskrivning
Inkapsling	AMC	Ändring av åtkomsttyp
Arv	IHD	Radering av gömd variabel
	IHI	Insättning av gömd variabel
	IOD	Radering av överskuggande metod
	IOP	Flytning av överskuggande metodanrop
	IOR	Namnbyte på överskuggande metod
	ISI	Insättning av nyckelordet *super*
	ISD	Radering av nyckelordet *super*
	IPC	Radering av explicit anrop av överordnad konstruktor
Polymorfism	PNC	*new* metodanrop med underordnad klasstyp
	PMD	Underordnad variabeldeklaration med överordnad klasstyp
	PPD	Parameter-variabeldeklaration med underordnad klasstyp
	PCI	Insättning av typkonvertering
	PCC	Ändring av typkonvertering
	PCD	Radering av typkonvertering
	PRV	Referenstilldelning med annan kompatibel typ
	OMR	Ändring av innehåll i överladdad metod
	OMD	Radering av överladdad metod
	OAC	Ändring av ordningen på argument
Java-specifika	JTI	Insättning av nyckelordet *this*
	JTD	Radering av nyckelordet *this*
	JSI	Insättning av *static*
	JSD	Radering av *static*
	JID	Radering av initiering av medlemsvariabel
	JDC	Skapande av javas standardkonstruktor
	EOA	Ändring av referens och tilldelningsinnehåll
	EOC	Byte av referens och innehålls jämförelse
	EAM	Ändring av accessmetod
	EMM	Ändring av modifieringsmetod

Tabell 4.1: Mutationsoperatorer på klassnivå som används av MuJava

De två viktigaste aspekterna att beakta vid jämförelse av mutationsverktyg är vilka mutationsoperatorer då stöder, samt vilken metod de använder vid mutation. Operatorerna är viktiga dels för att de avgör hur omsorgsfullt testerna utvärderas men även för att deras antal kan ha en avgörande inverkan på exekveringstiden. [16]

4.2.1 MuJava och MuClipse

MuJava

MuJava, (**Mu**tation System for **Java**) stöder hela processen för mutationstestning, vilket innefattar generering av mutanter, analys av mutanter och exekvering av mutanter mot testfall skrivna av utvecklaren. Projektet är ett samarbete mellan två universitet: *Korea Advanced Institute of Science and Technology* (KAIST)

i Sydkorea och *George Mason University* i USA. Den första versionen av MuJava släpptes är 2003 och den senaste versionen 2008. Programmet kan laddas ner från respektive universitets hemsida. [17]

Mujava kan delas upp i tre komponenter: mutantgeneratorn, mutantpresenteraren samt mutantexekveraren. Generatorn ger användaren möjlighet att själv välja vilka filer som skall muteras samt vilka mutationsoperatorer som skall användas. Mujava stöder både klass-specifika samt metod-specifika operatorer. Mutantpresenteraren visar sedan hur många mutanter av respektive typ som har genererats. Det är också möjligt för användaren att se vilka ändringar som gjorts i koden, vilket kan underlätta vid analys av ekvivalenta mutanter samt vid utveckling av tester för att döda svår-dödade mutanter. Mutantexekveraren ansvarar för att köra alla de av generatorn genererade mutanterna mot de av användaren specificerade testfallen. Resultaten av körningen, det vill säga mutationsvärdet, skrivs sedan ut. Mujava gör muteringarna direkt i java-bytekoden för att undvika dyr omkompilering. [12]

Nackdelarna med MuJava är de facto att det inte kan köras via kommandotolken, vilket förhindrar satsvis bearbetning av en stor mängd data. Dessutom saknar det integration med JUnit samt Ant eller Maven, vilket visserligen delvis kan åtgärdas tack vare MuClipse.[16]

MuClipse

MuClipse är ett insticksprogram för Eclipse som fungerar som en bro mellan utvecklingsmiljön Eclipse och MuJava. Det har utvecklats av Benjamin Hatfield Smith, men den aktiva utvecklingen lades dessvärre ned 29.9.2011. Utvecklingen av MuClipse baserar sig på MuJava och dess uppgift är att förse användaren med ett grafiskt användargränssnitt via Eclipse samt möjliggöra användning am Mujava med JUnit. MuClipse har samma funktioner som MuJava och kan därför långt delas upp i samma tre komponenter. Figur A.1 visar hur användaren, i körningsalternativen i Eclipse, kan välja vilken fil som skall muteras och vilka mutationsoperatorer som skall användas. Med “Traditional mutants” menas här

det som tidigare i avhandlingen gått under namnet “mutanter på metodnivå”. Resultaten av körningen sparas sedan i en mapp “result” och användaren kan se de genererade mutanterna som en lista i en skild vy “Mutants and Results” i Eclipse figur A.2. Användaren kan klicka på mutanterna i denna lista för att öppna vyn “Compare Mutants” figur A.3 var man kan se skillnaderna mellan originalkoden och mutanten. De genererade mutanterna exekveras mot testfallen med hjälp av ytterligare ett nytt körningsalternativ i Eclipse figure A.4. Här kan användaren välja vilken testfil som skall köra och vilken fils mutanter den skall testa. Det finns även möjlighet att välja en över tidsgräns för hur länge en mutant skall köras innan den anses ha fastnat i en oändlig loop. Resultaten av körningen skrivs sedan till en fil i mappen “result” och listan i figur A.2 uppdateras med status “killed” eller “alive”.[18]

Även om MuClipse underlättare användandet av MuJava genom integration med Eclipse, uppstår fortfarande problem vid användning på större projekt eftersom MuClipse kräver en viss namngivningskonvention som inte alltid behöver vara den samma som använts vid utvecklingen av systemet. Vidare kan endast en klass i gången testas och verktyget saknar integration med ett byggverktyg så som Ant eller Maven.[16]

Muclipse finns att ladda ned gratis från http://muclipse.sourceforge.net/ eller direkt från Eclipse genom att lägga till deras sida till Eclipse uppdateringssidor.

4.2.2 Javalanche

Javalance är ett mutationsverktyg för projekt som använder sig av Junit 3 eller 4. Programmet är terminalbaserat, dvs. det har inget grafiskt användargränssnitt.[19] Ett tilläggsprogram till Eclipse har enligt [20] existerat men är för tillfället inte under utveckling. Javalanche presenterades första gången 24-28 Augusti 2009 under *European Software Engineering Conference* (ESEC) och *ACM SIGSOFT Symposium on the Foundations of Software Engineering* (FSE)[21].

Tyngdpunkten vid utvecklingen av Javalanche är effektivitet och att försöka möj-

liggöra mutationstestning av stora projekt med mycket källkod. Detta åstadkoms m.h.a. att värdera inverkan av enskilda mutanter. Med inverkan avses här skillnaden mellan körningen av en mutant och ursprungskoden. På så sätt kan man undvika ekvivalent mutanter och använda färre mutationsoperatorer, vilket leder till lägre exekveringstider tack vare färre mutanter att exekvera. Vidare kommer varje testfall inte att exekvera varje mutant, eftersom programmet samlar täckningsinformation för varje test och exekverar endast de tester som täcks av just det testfallet. Javalanche muterar dessutom liksom MuJava, Java-bytekoden direkt. Slutligen stöder det parallellexekvering och kan således användas på parallella och distribuerade system. Javalanche är fullt automatiserat och kräver endast namnet på testsviten, namnet på projektets paket och klasserna som behövs för att exekverar testsviten.[20]

Javalanche körs via kommandotolken och alla inställningar görs i en XML-fil vid namn *javalanche.xml*. Resultaten sparas i en HSQL-databas och kan avläsas från en HTML-fil.[19]

4.2.3 Judy

Judy är ett relativt nytt mutationsverktyg. Dess första version släpptes 7.7.2011 och det utvecklas av professor Lech Madeyski vid Wroclaw University of Technology.[22]

FAMTA Light

FAMTA Light (Fast aspect-oriented mutation testing algorithm) baserar sig på den från aspektorienterad programmering (AOP) bekanta mekanismen "aspekt och punktsnitt". Denna mekanism försöker eliminera upprepad kod med samma funktionalitet som vid t.ex loggning eller tidtagning av metoder. För att åstadkomma detta definieras kodpunkter i programmet, där den upprepade koden skall köras. Punktsnitten samlar sedan ihop dessa punkter och ger dem namn. Genom att sedan ange kod-direktiv som skall köras vid dessa punkter kan alla dessa

samlas i en enda aspekt. Detta kan lättare ses ur 4.1[23]

$$\text{aspekt} = \text{punktsnitt}(\text{kodpunkter}) + \text{direktiv} \tag{4.1}$$

Denna mekanism tillämpas genom FAMTA Light på mutationstestning. Varje direktiv ansvarar för exekveringen av en metod i programmet. I direktivet görs sedan valet ifall originalet eller en mutant skall exekveras. Mutanterna sparas som mutantmetoder i samma klass som originalmetoden. Detta medför att alla mutanter kan kompileras på samma gång så kompileringen behöver inte upprepas för varje mutant, vilket medför en märkbart mindre exekveringstid. Dessvärre resulterar detta tillvägagångssätt i mycket långa klasser, med uppemot 10000 rader kod, vilket leder till bl.a *OutOfMemoryError* undantag. Detta löstes genom att göra processen iterativ och köra mindre iterationer av: genererings-, kompilerings- och testnings-faser för att minska antalet mutanter som kan tillämpas på samma gång. [16]

Judy

Även Judy koncentrerar sig på att göra mutationstestning av stora system möjligt genom minska antalet mutanter och onödiga kompileringar samt körningar. Med hjälp av FAMTA Light, fullt automatiserad mutationstestnings-process och stöd för de senaste Java-versionerna vill Judy överkomma de problem som gjort att mutationstestning fortfarande inte används i stor utsträckning inom industrin för programvaruproduktion. I en jämförelse mellan MuJava och Judy visar sig Judy, med ett medeltal av 52,05 mutanter/min mot MuJava 4,15 mutanter/min, vara överlägset snabbare än MuJava.

Eftersom Judy muterar på källkodsnivå och FAMTA Light endast gör mutanter av metoder, klara Judy för tillfället endast av metodspecifika-mutationsoperatorer. Judy är dock under aktiv utveckling och målet är att snart även kunna stöda klasspecifika-mutationsoperatorer, genom att kombinera FAMTA Light med exemplevis bytekodsmutering.[16]

5 Avslutning

Mutationstestning är som tidigare nämndes ett av de mest kraftfulla verktygen för att producera högkvalitativa test och därmed även högkvalitativ mjukvara. Dessvärre har dess användning länge varit begränsad till mindre system p.g.a den enorma beräkningskapacitet som skulle behövas för tillämpning på större system[14]. Lyckligtvis går utvecklingen framåt och tack vare verktyg som Javalanche och Judy som prioriterar användbarhet och skalbarhet har möjligheten till mutationstestning av större system blivit betydligt större. Forskningen har dessutom gått från att koncentrera sig på simpla operatorer till mer detaljerad mutation. Detta innebär att fokus gått från syntaktiska förändringar till de semantiska effekterna av mutation. Denna förändring har medfört ökat intresse för mutation på högre nivå som ger upphov till detaljerad mutation , vilket förhoppningsvis ger resulterar i mer realistiska mutanter.[11]

Framtiden ser lovande ut för mutationstestning. Antalet verktyg har ökat under de senaste åren och även storleken av de program som testas har vuxit till avsevärt.[11] Forskare tror att inom en snar framtid kan det vara möjligt att endast ge en mjukvarumodul som indata till testningsverktyget och efter en rimlig tid få färdigt skrivna test, som garanterat ger effektiv testning, som utdata. [14]

Litteratur

[1] Software QA and Testing Resource Center, "Software QA and Testing Frequently-Asked-Questions, Part 1." http://www.softwareqatest.com/qatfaq1.html. **Hämtad 30.3.2012**.

[2] P. Ammann och J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 1 ed., 2008.

[3] R. Patton, *Software Testing*. Indianapolis, IN, USA: Sams, 2 ed., 2005.

[4] G. J. Myers och C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[5] I. Zahid, "What is v model in testing?." http://www.blogs.zahidiqbal.info/post/What-is-V-model-in-testing.aspx, Augusti 2011. **Hämtad: 16.2.2012**.

[6] J. A. Whittaker, "What is software testing? and why is it so hard?," *IEEE Softw.*, vol. 17, sid. 70–79, Januari 2000.

[7] P. Oladimeji, "Levels of testing," December 2007.

[8] J. Gosling och H. McGilton, *The Java language environment: a white paper*. Sun Microsystems Computer Co., 1995.

[9] P. Ammann och J. Offut, "Junit, automated software testing framework." http://cs.gmu.edu/~offutt/classes/637/slides/Ch1-junit.pdf. **Hämtad 24.3.2012**.

[10] R. A. DeMillo, R. J. Lipton, och F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, nr. 4, sid. 34–41, 1978.

[11] Y. Jia och M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, sid. 649–678, 2011.

[12] Y.-S. Ma, J. Offutt, och Y.-R. Kwon, "Mujava: a mutation system for java," i *Proceedings of the 28th international conference on Software engineering*, ICSE '06, (New York, NY, USA), sid. 827–830, ACM, 2006.

[13] J. Z. Gao, J. Tsao, Y. Wu, och T. H.-S. Jacob, *Testing and Quality Assurance for Component-Based Software*. Norwood, MA, USA: Artech House, Inc., 2003.

[14] A. J. Offutt och R. H. Untch, "Mutation 2000: Uniting the orthogonal," i *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, (San Jose, Kalifornien), sid. 34–44, 6-7 Oktober 2001.

[15] Y.-S. Ma, Y.-R. Kwon, och J. Offutt, "Inter-class mutation operators for java," i *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, (Washington, DC, USA), sid. 352–, IEEE Computer Society, 2002.

[16] L. Madeyski och N. Radyk, "Judy - a mutation testing tool for java.," *IET Software*, vol. 4, nr. 1, sid. 32–42, 2010.

[17] Y.-S. Ma, J. Offutt, och Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, sid. 97–133, Juni 2005.

[18] B. H. Smith och L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Softw. Engg.*, vol. 14, sid. 341–369, Juni 2009.

[19] "Javalanche wiki." `https://github.com/david-schuler/javalanche/wiki/Documentation`. **Hämtad: 17.3.2012**.

[20] D. Schuler och A. Zeller, "Javalanche: Efficient mutation testing for java," i *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, sid. 297–298, Augusti 2009.

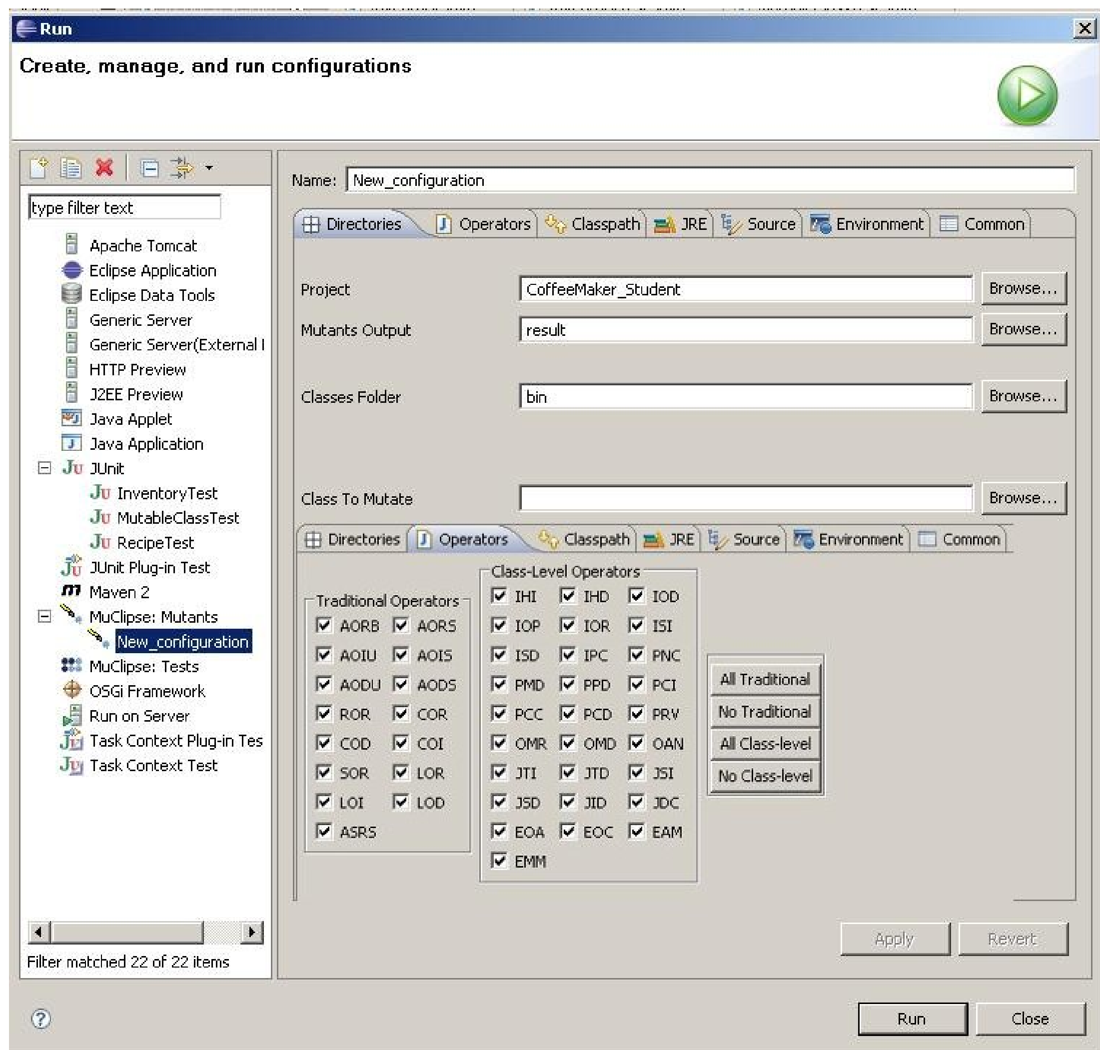
[21] "Javalanche." `https://www.javalanche.org`. **Hämtad: 17.3.2012**.

[22] "Judy documentation." `http://java.mu/`. **Hämtad: 18.3.2012**.

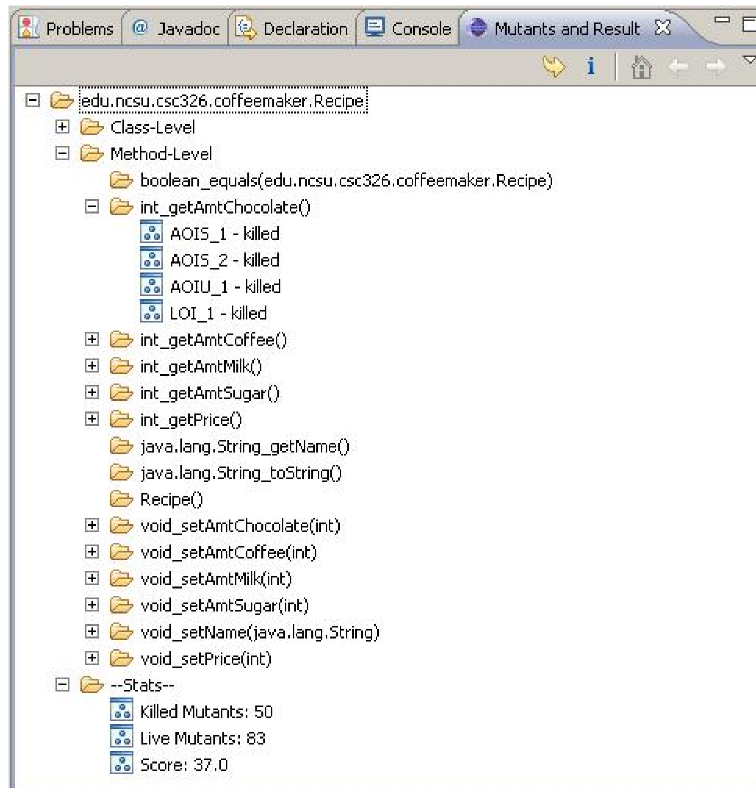
[23] B. Granvik, "Ytterligare en aspekt på din kod." `http://blog.jayway.com/2004/01/20/ytterligare-en-aspekt-pa-din-kod/`. **Hämtad: 18.3.2012**.

A Bilaga 1

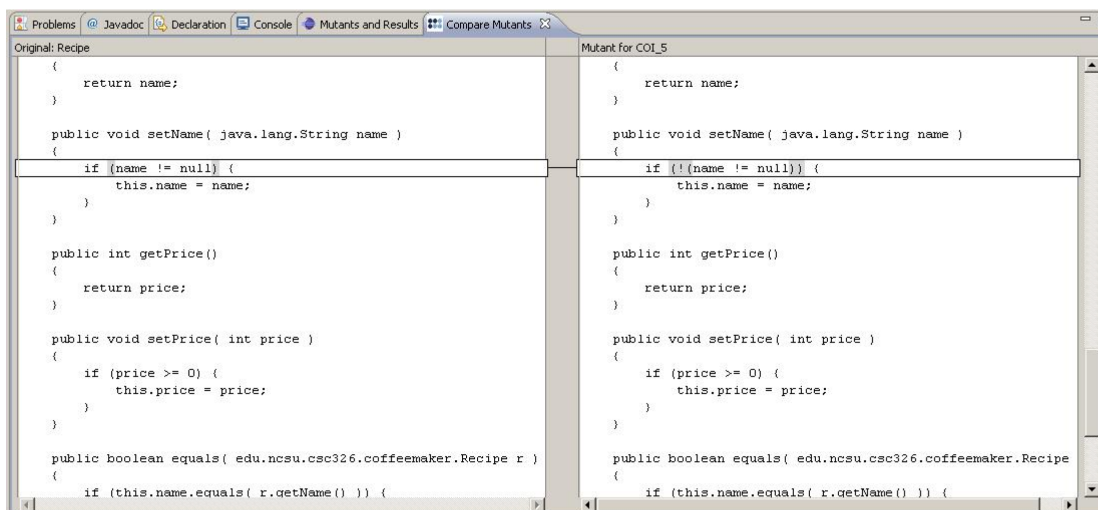
Skärmdumpar från Muclipse.



Figur A.1: MuClipse GUI för generering av mutanter

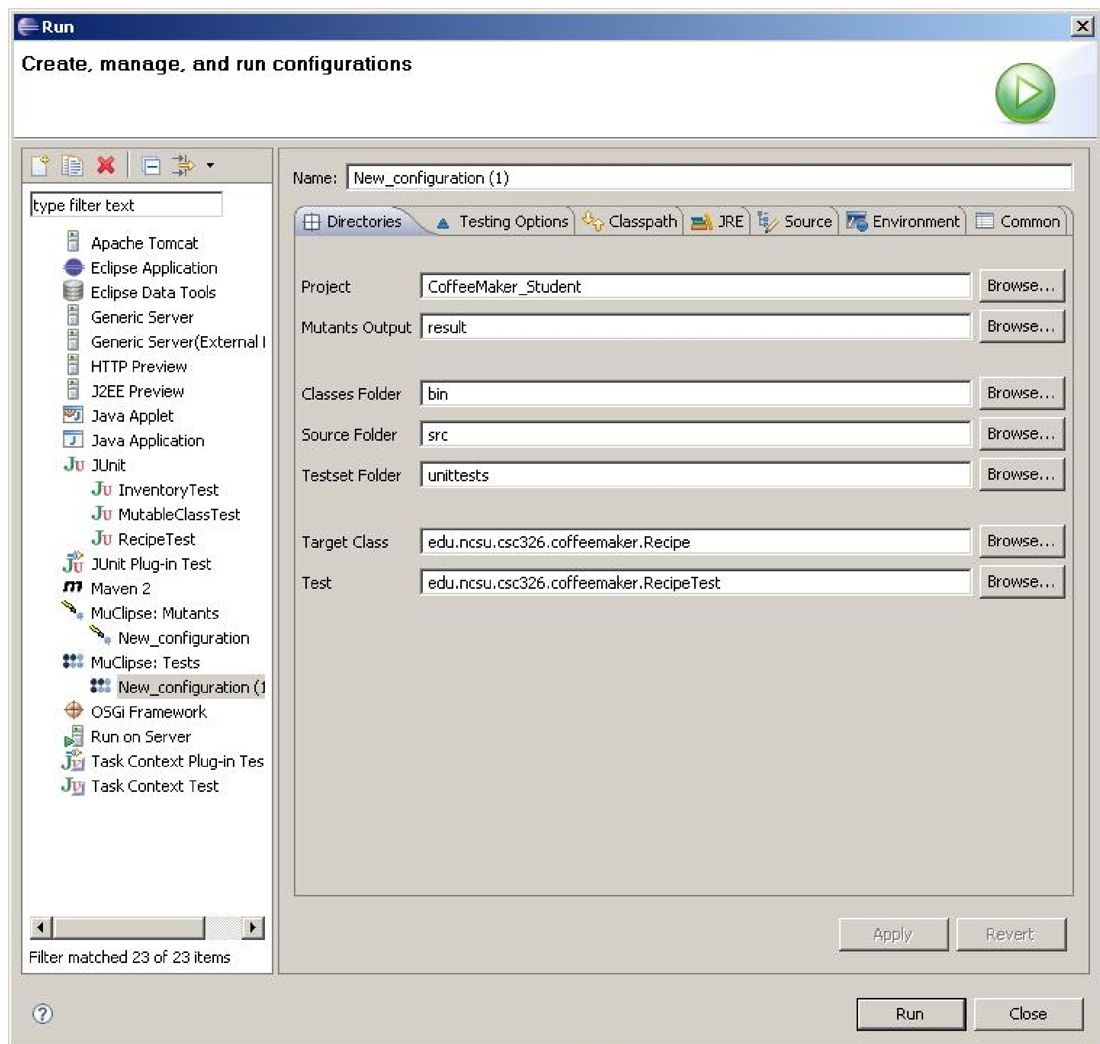


Figur A.2: Lista över genererade mutanter i MuClipse GUI




```
{                                             {
    return name;                                  return name;
}                                             }

public void setName( java.lang.String name )  public void setName( java.lang.String name )
{                                             {
    if (name != null) {                           if (!(name != null)) {
        this.name = name;                             this.name = name;
    }                                             }
}                                             }

public int getPrice()                         public int getPrice()
{                                             {
    return price;                                 return price;
}                                             }

public void setPrice( int price )             public void setPrice( int price )
{                                             {
    if (price >= 0) {                             if (price >= 0) {
        this.price = price;                           this.price = price;
    }                                             }
}                                             }

public boolean equals( edu.ncsu.csc326.coffeemaker.Recipe r )   public boolean equals( edu.ncsu.csc326.coffeemaker.Recipe
{                                             {
    if (this.name.equals( r.getName() )) {        if (this.name.equals( r.getName() )) {
```

Figur A.3: Jämförelse av mutant och original i MuClipse



Figur A.4: MuClipse GUI exekvering av mutanterna mot testfallen