Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Evaluation der Leistungsfähigkeit ausgewählter Mutationstestwerkzeuge

Lea Kristin Gerling[1]

Abstract: Mutationstests dienen zur Qualitätsbestimmung von Softwaretests. Um die Ausführung dieser Tests zu automatisieren, gibt es eine Reihe von Mutationstestwerkzeugen. Diese Arbeit beschäftigt sich mit den öffentlich verfügbaren Werkzeugen für die Programmiersprache Java und dem Testframework JUnit. Sie vergleicht die Werkzeuge MuJava, Jumble, Pitest, Judy und Major hinsichtlich ihres Funktionsumfangs, ihrer Leistungsfähigkeit und ihrer Praxistauglichkeit miteinander. Die Ergebnisse dieser Evaluation dienen als Entscheidungshilfe bei der Auswahl eines Werkzeugs zur Testverbesserung.

Keywords: Mutationstest, Mutationsanalyse, Mutationstestwerkzeug, Evaluation.

1 Einleitung

Softwaretests sind ein wichtiger Bestandteil der Softwareentwicklung. So spart frühzeitiges Testen nicht nur Geld, da ein Fehler umso mehr Kosten verursacht, je später er gefunden wird [Bo79]. Es sorgt auch für eine Sicherung der Softwarequalität. Testen hat jedoch nur einen geringen Nutzen, wenn die Tests selbst eine schlechte Qualität haben und das Programm auch nach ausführlichem Testen noch viele Fehler aufweist. Um herauszufinden, wie gut ein Test ist, muss auch seine Qualität analysiert werden. Dazu eignet sich die in den 1970er Jahren entwickelte Technik des *Mutationstestens* [DLS78]. Hierbei werden Fehler in ein Programm eingebaut, welche die Tests entdecken müssen. Je mehr Fehler entdeckt werden, umso besser ist die Qualität der Tests.

Es existiert eine Menge unterschiedlicher Werkzeuge, mit denen Mutationstests automatisiert durchgeführt werden können [JH11]. Allerdings variiert die Leistungsfähigkeit dieser Werkzeuge. Dadurch erfordert die Entscheidung für ein geeignetes Werkzeug eine gründliche Abwägung der Vor- und Nachteile. Diese Arbeit beschäftigt sich mit dem Vergleich von ausgewählten Mutationstestwerkzeugen, kurz *MTW*, für die Programmiersprache Java und dem Testframework JUnit. Sie geht dabei der Frage nach, welches dieser MTW sich am besten für einen Einsatz in der Praxis eignet. Dazu erfolgt in Kapitel 2 zunächst eine kurze Einführung in den Bereich des Mutationstestens. In Kapitel 3 erfolgt die Vorstellung der untersuchten MTW und ihrer genutzten Ansätze. Kapitel 4 befasst sich mit der praktischen Umsetzung der Evaluation. Kapitel 5 zieht schließlich ein Fazit über die gewonnen Erkenntnisse.

[1]Universität Hildesheim, Abteilung Software Systems Engineering, Universitätsplatz 1, 31141 Hildesheim, gerlingl@uni-hildesheim.de

2 Mutationstests

Mutationstesten ist ein dynamisches, fehlerbasiertes Whitebox-Testverfahren, welches Ende der 1970er Jahre erstmals von DeMillo, Lipton und Sayward [DLS78] sowie Hamlet [Ha77] in wissenschaftlichen Publikationen vorgestellt wurde. Das Besondere an diesem Verfahren ist, dass es sich um eine Methode zur Qualitätsbestimmung einer Testmenge handelt. Getestet wird demnach nicht die Software selbst, sondern die dazugehörigen Tests. Ursprünglich war der Mutationstestansatz zur Überprüfung von Tests auf Komponentenebene gedacht [DLS78], mit der Zeit wurden zudem Anwendungs-möglichkeiten auf Integrations-, System- und Entwurfsebene geschaffen [JH11]. Auch die Anzahl der unterstützten Programmiersprachen hat sich seit den Anfängen deutlich erhöht. Zunächst gab es nur Mutationen für FORTRAN Programme, später kamen dann Sprachen wie Java, C oder SQL hinzu [JH11]. Aufgrund dieser Erweiterungen lassen sich Mutationstests heutzutage nach Offutt [Of11] folgendermaßen definieren: *„The use of well defined rules defined on syntactic descriptions to make systematic changes to the syntax or to objects developed from the syntax."*

Wie solch ein Mutationstest abläuft, sei an nachfolgendem Beispiel aus Tab. 1 erklärt. Zu Beginn steht die Eingabe eines Programms sowie der dazugehörigen Tests. Das Beispielprogramm erwartet die Eingabe einer Zahl. Ist die Zahl kleiner als 10, wird der Wert 0 zurückgegeben. Ist die Zahl größer oder gleich 10, wird der Wert der Zahl zurückgegeben. Der dazugehörige Test in der rechten Spalte von Tab. 1 überprüft dieses verhalten. Dazu wird in Zeile 1 überprüft, ob bei der Eingabe des Wertes 5 der Wert 0 zurückgegeben wird. In Zeile 2 wird geprüft, ob bei einer Eingabe von 15 auch der Wert 15 zurückgegeben wird. Beide Tests sind erfolgreich. Die erfolgreiche Durchführung der Tests ist eine notwendige Bedingung für fehlerbasiertes Testen [Mo90].

Programm	**Test**

1. int comp(int a){
2. if (a < 10){
3. return 0;
4. } else {
5. return a;
6. }
7. }
	1.assertEquals(0,comp(5));
	2.assertEquals(15,comp(15));

Tab. 1: Beispielprogramm und Test

Als nächstes erfolgt die *Mutation* des Programms. Eine Mutation verändert immer nur einen kleinen Teil der Software, um realistische Fehler zu simulieren [DLS78]. Dies ist eine der Kernhypothesen, auf die sich Mutationstests stützen. Sie wird als *Competent Programmer Hypothesis* bezeichnet. Die Hypothese wurde folgendermaßen definiert: *„Programmers have one great advantage that is almost never exploited: they create programs that are close to being correct!"* [DLS78]. Sie besagt, dass kompetente Programmierer nur kleine Fehler machen, im Prinzip also nahezu korrekte Programme schreiben. Sie programmieren nicht zufällig, sondern versuchen ihr Programm so zu schreiben, dass es der Soll-Version möglichst nahe kommt.

Welche Arten von Veränderungen eingebaut werden, ist abhängig von den gewählten *Mutations-Operatoren* [JH11]. Allgemein definiert ein Mutations-Operator zwei Aspekte der Mutation: Was sind die Zielobjekte und welcher Art ist die Veränderung. Als Zielobjekte dienen beispielsweise Methodenaufrufe, mathematische Operatoren, Vergleichsoperatoren oder `return` Statements. Die Veränderung kann beispielsweise darin bestehen, Zielobjekte zu löschen, umzukehren oder ihren Wahrheitswert auf `true` zu setzen. Eine mögliche Mutation des Programms aus Tab. 1 wäre also beispielsweise, den Vergleichsoperator aus Zeile 2 umzukehren in `(a > 10)`. Neben diesen einfachen Mutationen, auch Mutationen auf Methodenebene genannt, gibt es noch weiterführende Konzepte wie beispielsweise objektorientierte Mutationen. Hier werden dann beispielsweise Sichtbarkeiten, Konstruktoren oder Schlüsselwörter wie `static`, `super` und `this` verändert. Außerdem gibt es noch die Möglichkeit, *Mutationen höherer Ordnung* durchzuführen. Das bedeutet, dass auch komplexere Mutationen erlaubt sind, um die Anzahl der insgesamt erzeugten Mutationen zu reduzieren [JH08].

Nach Erzeugung der mutierten Programmversionen, auch *Mutanten* genannt, werden die Testfälle für jeden Mutanten erneut ausgeführt. Daher ist es notwendig, dass die erzeugten Mutationen syntaktisch korrekt sind und die Software weiterhin kompilierbar ist [OU01]. Schlägt ein Testfall fehl, gilt der Mutant als *eliminiert*. Dies passiert auch bei der zuvor erwähnten Umkehrung des Vergleichsoperators des Beispielprogramms. Je mehr Mutanten eliminiert werden, desto besser sind die Tests. Ändert man den Vergleichsoperator des Beispielprogramms aus Tab. 1 beispielsweise in `(a <= 10)`, würden die Tests trotzdem noch erfolgreich durchlaufen. Diese Tests können also noch verbessert werden.

Allerdings gibt es auch Mutationen, die nicht automatisch entdeckt werden können, da sie semantisch äquivalent sind [BA82]. Hier spricht man dann von *äquivalenten Mutanten.* Diese Mutanten stellen eine Schwachstelle der Mutationstests dar, da sie die Bewertung der Tests verzerren können, wenn sie nicht als solche gekennzeichnet werden. Deswegen fließen die äquivalenten Mutanten bei der Berechnung des *Mutation Scores* mit ein. Dieser Wert steht am Ende des Mutationstestprozesses und wurde zuerst von Adrion, Branstad und Cherniavsky im Jahre 1982 eingeführt [ABC82]:

$$\text{Mutation Score} = \frac{|\text{eliminierte Mutanten}|}{|\text{erzeugte nicht-äquivalente Mutanten}|}$$

Der aus dieser Formel errechnete Wert gibt an, wie gut die getesteten Testfälle sind. Die Testfälle haben eine besonders hohe Qualität, wenn der Mutation Score einen Wert von 1 beziehungsweise 100% erreicht. Er berechnet sich aus der Anzahl an eliminierten Mutanten geteilt durch die Anzahl an nicht-äquivalenten Mutanten.

3 Vorstellung der Mutationstestwerkzeuge

MTW sind für die effiziente Anwendung von Mutationstests nahezu unersetzbar, da eine

manuelle Ausführung einen hohen Aufwand erfordert. Doch die Entscheidung für eines dieser MTW erfordert die Einbeziehung verschiedener Faktoren. Als besonders wichtig wurden dabei die Faktoren Verfügbarkeit, unterstützte Java-Version, Plattformunabhängigkeit, sowie Status der Entwicklung gewertet. Daher werden in dieser Arbeit nur die MTW betrachtet, die frei verfügbar zum Download stehen, mindestens Java Version 5 unterstützen, plattformunabhängig funktionieren und innerhalb der letzten 3 Jahre weiterentwickelt wurden. Dieses Kapitel stellt die ausgewählten MTW und ihre Eigenschaften vor.

3.1 Übersicht der Werkzeuge

MuJava ist das älteste MTW für Java, welches noch aktiv weiterentwickelt wird. Seine Ursprünge hat es in dem 2002 von Chevalley und Thévenod-Fosse entwickeltem Werkzeug JavaMut [CT03]. Die erste Version von MuJava wurde 2003 noch unter dem Namen JMutation von Offutt, Ma, und Kwon veröffentlicht [OMK04]. Diese Arbeit untersucht Version 4, welche im Juni 2013 veröffentlicht wurde. Beispiele und genauere Erläuterungen der verwendeten Mutationen von MuJava finden sich in [MO11] und [MO14].

Das MTW *Jumble* wurde erstmals 2007 von Irvine et al. veröffentlicht [Ir07]. Zuvor wurde es von 2003-2006 als Werkzeug der Firma Reel Two entwickelt. Es wurde dazu entworfen, in einer industriellen Umgebung mit großen Programmen zu arbeiten. In dieser Arbeit wird Version 1.2.0 vom 22.04.2013 untersucht. Eine ausführliche Erklärung der Mutations-Operatoren findet sich auf [Ju07].

Das von Henry Coles entwickelte MTW *Pitest* wurde zum ersten Mal im Dezember 2011 veröffentlicht [Co14a]. Es basierte ursprünglich auf Jumble, benutzt aber mittlerweile eigene Ansätze zur Mutanten-Erzeugung. Die betrachtete Version ist Version 1.0.0 vom 18. Mai 2014. Eine Erklärung zu Pitests Mutations-Operatoren findet sich auf [Co14b].

Judy ist ein von Lech Madeyski und Norbert Radyk im Jahr 2011 veröffentlichtes MTW [MR10]. In dieser Arbeit wird Version 2.1.0 vom 26. Januar 2014 untersucht. Erläuterungen und Beispiele zu den Mutations-Operatoren von Judy finden sich in [MR10].

Die erste Version des MTW *Major* wurde im Januar 2014 von René Just veröffentlicht [Ju14]. Somit ist Major das jüngste der betrachteten MTW. Diese Arbeit befasst sich mit der aktuellen Version 1.1.2, welche am 27. Juni 2014 zum Download bereitgestellt wurde. Eine genaue Beschreibung von Majors Mutationen findet sich in [Ma14].

3.2 Gegenüberstellung der Eigenschaften

Jedes der MTW bietet dem Benutzer die Möglichkeit, die angewendeten Mutations-Operatoren auszuwählen, siehe Tab. 2. Dieses Vorgehen nennt sich selektive Mutation.

Es dient auch dazu, die Erzeugung äquivalenter Mutanten zu verringern. Dies geschieht, indem nur die Operatoren ausgewählt werden, die statistisch gesehen sehr wenig äquivalente Mutanten erzeugen. Allerdings lässt sich dadurch die Erzeugung nur indirekt und bis zu einem gewissen Maße verringern, mehr dazu findet sich auch in [MB99]. Methoden zur Identifizierung äquivalenter Mutanten werden nicht angewendet. Weiterhin fällt der Umfang der Selektionsmöglichkeiten der einzelnen MTW recht unterschiedlich aus. Während MuJava 41 Mutations-Operatoren zur Auswahl stellt, sind es bei Jumble nur Sieben.

Name	Selektive Mutation	Mutationen Höherer Ordnung	Objektorientierte Mutationen
MuJava	Ja	Nein	Ja
Jumble	Ja	Nein	Nein
Pitest	Ja	Nein	Nein
Judy	Ja	Ja	Ja
Major	Ja	Nein	Nein

Tab. 2: Vergleich der Mutationsarten

MuJava und Judy nutzten im Gegensatz zu den anderen MTW sowohl Mutations-Operatoren auf Methodenebene als auch objektorientierte Operatoren, wie in Tab. 2 zu sehen ist. Major bietet als einziges MTW die Möglichkeit, Operatoren nicht nur ein- und auszuschalten, sondern auch ihre Funktionsweise zu konfigurieren. Dafür nutzt Judy als einziges MTW noch zusätzlich die Technik der Mutationen höherer Ordnung.

Um die Testausführung zu optimieren, nutzt MuJava als einziges MTW keine Techniken, siehe Tab. 3. Jumble, Pitest und Major hingegen kombinieren drei verschiedene Methoden zur Priorisierung. Auch wenn sich diese Methoden innerhalb der MTW unterscheiden, so geht es doch im Kern darum, nur die Tests mit der kürzesten Dauer auszuführen, die zu der aktuell betrachteten Mutation gehören und die im Vergleich zum letzten Testlauf ein anderes Ergebnis erwarten lassen. Weiterhin nutzen Jumble, Judy und Major noch jeweils eine andere Technik zur Verbesserung der Ausführungszeiten. Zudem nutzten alle MTW eine Begrenzung der Ausführungsdauer der Tests, um Endlosschleifen zu vermeiden.

Name	Testausführungsoptimierung
MuJava	Keine
Jumble	Priorisierung nach Dauer, Änderung und Zugehörigkeit, Parallele JVM
Pitest	Priorisierung nach Dauer, Änderung und Zugehörigkeit
Judy	Pointcut and Advice
Major	Priorisierung nach Dauer, Änderung und Zugehörigkeit, Weak Mutation

Tab. 3: Testausführungsoptimierung der MTW

Die nachfolgende Tab. 4 enthält die Eigenschaften der MTW, welche für die technische Integration relevant sind. MuJava, Jumble und Judy unterstützen Java Version 6, Major

unterstützt Version 7 und Pitest Version 8. Alle MTW unterstützen JUnit 4. Sämtliche MTW außer MuJava können über Ant betrieben werden. Für Jumble und Pitest gibt es zusätzlich noch ein Eclipse Plugin, welches aber nicht alle Konfigurationsmöglichkeiten des Kommandozeilen-Betriebs bietet. Weiterhin unterstützen beide noch einige Mock-Versionen und Pitest bietet zusätzlich noch Unterstützung für Maven, Gradle, TestNG, Sonar und Intellij. Da es sich um MTW für Java handelt, sind sie prinzipiell nicht an ein Betriebssystem gebunden. Allerdings funktioniert die Installation von Judy über Shell bzw. Bat Skripte, wodurch eine einfache Installation nur unter Windows und Linux möglich ist.

Name	Java Version	JUnit Version	Toolunterstützung	Betriebssystem
MuJava	6	4	-	Plattformunabhängig
Jumble	6	4	Ant, Eclipse, Mock	Plattformunabhängig
Pitest	8	4	Ant, Maven, Gradle, TestNG, Eclipse, Mock, Sonar, Intellij	Plattformunabhängig
Judy	6	4	Ant	Windows, Linux
Major	7	4	Ant	Plattformunabhängig

Tab. 4: Technische Eigenschaften der MTW

4 Leistungsvergleich der Mutationstestwerkzeuge

Mutationstests finden nur selten eine Anwendung in der Praxis [JH11]. Dies hängt unter anderem mit ihrem relativ hohen Ressourcenverbrauch zusammen. Daher wird in diesem Kapitel die Frage beantwortet, wie gut die getesteten MTW mit den Performanzproblemen umgehen. Weiterhin stellt sich auch die Frage, ob die MTW überhaupt für einen Praxiseinsatz in Wirtschaft oder Forschung geeignet sind. Hohe Kosten sind kein Argument, wenn sich der Aufwand lohnt. Daher liegt die Vermutung nahe, dass es noch weitere Probleme bei der Anwendung gibt. Deswegen wird in diesem Kapitel zusätzlich auch die Bedienbarkeit und Aussagekraft der MTW überprüft.

4.1 Beschreibung der Testumgebung

Inhalt dieser Arbeit ist der Vergleich von MTW für die Programmiersprache Java. Um die Aktualität der Ergebnisse zu gewährleisten, wurde daher mit der Java Laufzeitumgebung Version 7.65 getestet. Als Systemumgebung wurde eine virtuelle Maschine mit Ubuntu 14.04 LTS verwendet. Das System wurde mit 2 GB RAM und einem Dualcore Prozessor mit 3,3 GHz aufgesetzt. Zur Ausführung der Evaluation wurden das Linux Terminal, Eclipse Luna Version 4.4.0 und JUnit Version 4.11 verwendet. Leistungsdaten wurden mit der Java VisualVM Version 1.3.3 erhoben.

Nicht jedes MTW ist eigenständig dazu in der Lage, in einem Testlauf mehrere Klassen zu mutieren. Um die Vergleichbarkeit der Ergebnisse zu gewährleisten, wurde daher als

Testobjekt ein sehr simples Programm verwendet. Dieses Programm dient zur Klassifizierung von Dreiecken und wurde ursprünglich von Kevin Jalbert als Testbeispiel für das MTW Javalanche geschrieben [Ja12]. Es besteht aus einer Klasse und zwölf JUnit-Tests.

4.2 Vorstellung der Testkriterien

Die hier vorgestellten Kriterien finden in Abschnitt 4.4 ihre Anwendung. Dabei enthält Tab. 5 die nachfolgend erläuterten Leistungskriterien. Das erste Kriterium ist der *Mutation Score*. Er gibt Auskunft über die Qualität der getesteten Testmenge. Außerdem dient er als Effektivitätskriterium zur Bewertung der MTW, da die Berechnung dieses Wertes ein Hauptanliegen der Mutationstests ist. Ein MTW funktioniert nur effektiv, wenn es einen sinnvollen Mutation Score berechnen kann. Damit verbunden ist die *Anzahl der erzeugten Mutanten*. Ist sie zu gering, hat der erzeugte Mutation Score keine Aussagekraft. Ein weiteres Effektivitätskriterium ist die *Anzahl der ausgeführten JUnit Tests* zur Eliminierung der Mutanten. Auch hier bedeutet ein zu geringer Wert einen Verlust an Effektivität. Ein zu hoher Wert hingegen kann ein Anzeichen für unnötig ausgeführte Tests sein.

Als Effizienzkriterium dient unter anderem der *maximale Speicherverbrauch* während eines Durchlaufs. Der Verbrauch war bei jedem MTW während des Testlaufs konstant. Weiterhin wurde die durchschnittliche *CPU-Auslastung* pro Testlauf gemessen. In Klammern steht die durchschnittliche Abweichung von diesem Wert. Diese Abweichung zeigt an, ob es starke Schwankungen während eines Testlaufs gibt. Die letzten beiden Effizienzkriterien sind die *Anzahl der erzeugten Mutanten pro Sekunde* und der *ausgeführten Tests pro Sekunde*. Hierbei wird noch einmal besonderes Augenmerk auf die effiziente Erzeugung der Mutanten beziehungsweise effiziente Durchführung der Tests zur Eliminierung der Mutanten gelegt.

Neben den objektiven Effektivitäts- und Effizienzbewertungen gibt es auch subjektive Bewertungen hinsichtlich der Bedienbarkeit der MTW. Diese werden in Tab. 6 angewendet und im nachfolgenden Absatz erläutert. Da sich die Bedienbarkeit eines MTW nur schwer objektiv messen lässt, wird jedes Kriterium stattdessen mit einer Note von 1 (*sehr gut)* bis 5 (*mangelhaft)* bewertet. Diese Note fasst die Erfüllung der Teilaspekte in einem leicht zu interpretierenden Wert zusammen.

Zum einen wurden *Installation*, *Mutanten-Erzeugung* und *Testausführung* hinsichtlich der Schwierigkeit der Ausführung dieser Schritte bewertet. Hier haben die MTW besonders gut abgeschnitten, bei denen sich diese Aktionen mit wenig Aufwand zügig erledigen lassen. *Dokumentation* und *Ergebnisdarstellung* wurden hinsichtlich des enthaltenen Umfangs bewertet und der in diesem Umfang enthaltenen Informationen. Hier steht Qualität über Quantität. Ähnlich verhält es sich auch mit der *Konfigurierbarkeit*. Die Anzahl der Möglichkeiten spielt eine Rolle, allerdings ist auch die Umsetzung nicht zu vernachlässigen. Falls die Konfiguration sehr umständlich ist, oder angegebene Möglichkeiten nicht umsetzbar sind, führt dies zu einer Abwertung.

4.3 Vorgehensbeschreibung

Zu Beginn der Evaluation erfolgte einmalig das Aufsetzen einer neuen virtuellen Maschine mit der in Abschnitt 4.1 beschriebenen Konfiguration. Innerhalb dieser Testumgebung wurde dann jeweils die neueste Version des MTW heruntergeladen. Auf Grundlage der verfügbaren Dokumentation wurde jedes MTW genau nach Anweisung installiert. Damit keine Seiteneffekte auftreten, wurden sämtliche Änderungen am System nach Erhebung der Testdaten rückgängig gemacht. Die Ausführung der Tests wurde 20-mal wiederholt. Die letztendlichen Ergebnisdaten wurden mithilfe des Median ermittelt, um Ausreißern keine zu große Gewichtung zu geben. Die MTW wurden mit den empfohlenen Standardeinstellungen getestet.

4.4 Darstellung der Ergebnisse

In Tab. 5 findet sich eine Übersicht über die einzelnen Bewertungen. Ein Fragezeichen bei der Bewertung bedeutet, dass dieses Kriterium nicht gemessen werden konnte beziehungsweise dazu keine Angaben gemacht wurden.

Betrachtet man den ausgegebenen Mutation Score, so fällt auf, dass dieser Wert starken Schwankungen unterworfen ist. Da immer die gleiche Testmenge bewertet wurde, liegt die Ursache dieser Schwankungen in der Methodik der MTW. Da nur MuJava und Major genaue Angaben über die erzeugten Mutationen machen, lässt sich nicht eindeutig feststellen, warum der Mutation Score so unterschiedlich ausfällt. Weiterhin scheint der berechnete Wert von MuJava nicht korrekt zu sein, da die erzeugten Mutanten zumindest teilweise von den Tests hätten entdeckt werden müssen. Die Gründe dafür sind nicht nachvollziehbar. Bei den anderen MTW hängen die unterschiedlichen Ergebnisse vermutlich mit der Anzahl der äquivalenten Mutanten und den genutzten Mutations-Operatoren zusammen. Dieselbe Argumentation betrifft auch die Anzahl der eliminierten Mutanten.

	MuJava	**Jumble**	**Pitest**	**Judy**	**Major**
Mutation Score	0%	97%	82%	100%	67%
Erzeugte Mutanten	432	39	44	121	79
Eliminierte Mutanten	0	38	36	121	53
Ausführungsdauer	36s	1,7 s	4,8 s	3,1 s	3,5 s
Ausgeführte Tests	?	?	116	662	641
Mutanten / Sekunde	12	22,9	9,2	39	22,6
Tests / Sekunde	?	?	24,2	213,5	183,1
Speicherverbrauch	232MB	5 MB	19 MB	95 MB	?
CPU-Auslastung	66% (± 27%)	2,6% (± 0,2%)	27% (± 3,1%)	0,01% (± 0%)	?

Tab. 5: Ergebnisse der Leistungstests

Die Anzahl der erzeugten Mutanten hängt mit den anwendbaren Mutations-Operatoren

zusammen. Je mehr Operatoren aktiv sind, desto mehr Mutanten können auch erzeugt werden. Eine höhere Anzahl erzeugter Mutanten ist einerseits positiv, da somit die Aussagekraft des Mutation Scores steigt. Andererseits bedeuten mehr Mutanten aber auch einen höheren Ressourcenbedarf. Deswegen ist es sinnvoll, auch die erzeugten Mutanten pro Sekunde zu betrachten. Hier fällt auf, dass Judy die Mutanten mit am effizientesten erzeugt. Klarer Verlierer hingegen ist das MTW Pitest, welches nicht nur vergleichsweise wenig Mutanten erzeugt, sondern dies auch am ineffizientesten tut.

Bei der Anzahl der durchgeführten Tests zur Eliminierung der Mutanten liegen Judy und Major mit weitem Abstand vorne, wobei hier auch nicht jedes MTW Daten dazu ausgegeben hat. Die unterschiedlichen Ergebnisse lassen sich durch die Anwendung verschiedener Optimierungsmethoden bei der Testausführung erklären, siehe Tab. 3.

Judy und Major nutzen beide Techniken zur Verbesserung der Effizienz, welches sich auch in der großen Anzahl der ausgeführten Tests pro Sekunde erkennen lässt. Pitest hingegen arbeitet mit Selektion der durchzuführenden Tests, wodurch die vergleichsweise geringe Anzahl an durchgeführten Tests zustande kommt. Allerdings ist auch Pitests Anzahl der ausgeführten Tests pro Sekunde im Vergleich sehr gering.

Bei den Daten zum Leistungsverbrauch konnte Major nicht gemessen werden, da es mit einer eigenen VM arbeitet. Judy überzeugt durch eine extrem niedrige CPU-Auslastung, hat dafür aber einen hohen Verbrauch an Arbeitsspeicher. Bei der Ausführungsdauer liegen Judy, Major und Pitest im Mittelfeld. Am schlechtesten schneidet MuJava ab, da es mit Abstand am längsten braucht und die höchste CPU-Auslastung hervorruft sowie am meisten Arbeitsspeicher benötigt. Jumble hingegen hat die kürzeste Ausführungsdauer, verbraucht sehr wenig RAM und benötigt auch nur eine geringe CPU-Auslastung.

Nachfolgend werden die Ergebnisse der Bedienbarkeitsbewertung diskutiert. Die einzelnen Bewertungen sind dabei in Tab. 6 zusammengestellt. Es sei anzumerken, dass die Benotung subjektiver Wahrnehmung entspricht und somit von Betrachter zu Betrachter abweichen kann.

	MuJava	**Jumble**	**Pitest**	**Judy**	**Major**
Installation	3	1	1	1	1
Dokumentation	3	2	3	3	5
Mutanten-Erzeugung	2	1	1	2	2
Testausführung	2	1	1	2	2
Ergebnisdarstellung	2	3	2	2	2
Konfigurierbarkeit	5	2	2	3	1
Automatisierbarkeit	4	1	1	1	1

Tab. 6: Ergebnisse der Bedienbarkeitstests

Bei der Bewertung der Installation gibt es nur zwei verschiedene Ausprägungen. Zum einen haben die MTW Jumble, Pitest, Judy und Major alle eine sehr gute Bewertung bekommen, da die Installation nur aus dem Herunterladen und eventuellen Entpacken

der MTW besteht. Einzig MuJava hat schlechter abgeschnitten, da hier eine Anpassung der Umgebungsvariablen und das Ändern einer Konfigurationsdatei notwendig waren.

Die Dokumentation zu den MTW war bestenfalls gut, im Falle von Major sogar nur mangelhaft. Letzteres kommt daher, dass Major zwar eine sehr umfangreiche Dokumentation besitzt, die wirklich wichtigen Informationen aber nicht enthalten sind. Die beste Dokumentation gehört zum MTW Jumble. Die vorhandenen Beschreibungen sind zwar kurz gehalten, dafür sind trotzdem alle wünschenswerten Informationen enthalten und übersichtlich dargestellt. MuJava, Pitest und Judy besitzen nur befriedigende Dokumentationen, da teilweise Lücken vorhanden oder fehlerhafte Informationen angegeben sind. Die Auswirkungen dieser Fehler sind allerdings nur minimal und führen daher nur zu einer geringen Abwertung.

Was Mutanten-Erzeugung und Testausführung angeht, haben alle MTW gut bis sehr gut abgeschnitten. Jumble und Pitest haben den Vorteil, dass durch ihr Eclipse Plugin die Ausführung dieser Aktionen sehr simpel ist. Auf Knopfdruck lassen sich Mutanten-Erzeugung und Testausführung in einem Schritt starten. Auch bei MuJava, Judy und Major lassen sich diese Schritte gut und einfach durchführen. MuJava verwendet dabei eine grafische Oberfläche, die über die Kommandozeile aufgerufen wird. Judy und Major benötigen nur die Eingabe über die Kommandozeile mit der Übergabe einiger Parameter. Das ist zwar nicht ganz so intuitiv wie die Ausführung direkt in Eclipse, aber dennoch ohne großen Aufwand zu bewerkstelligen.

Bei der Darstellung der Ergebnisse hat Jumble die schlechteste Note bekommen. Es werden zwar solch wesentliche Angaben wie Mutation Score und überlebende Mutanten ausgegeben, Angaben über die erzeugten Mutanten oder nützliche Statusinformationen fehlen hingegen. MuJava, Pitest, Judy und Major haben eine gute Darstellung der Ergebnisse. Hier fehlten nur die Angaben über die erzeugten Mutationen beziehungsweise bei Major und MuJava eine übersichtliche Darstellung. Bei Major werden zwar die Mutationen genannt, allerdings benötigt man drei verschiedene Logdateien, um mit den Informationen auch etwas anfangen zu können. Bei MuJava werden die Logausgaben und die grafische Oberfläche zum Nachvollziehen der Detailinformationen benötigt.

Was die Konfigurierbarkeit der MTW angeht, haben Jumble, Pitest und Judy alle ähnliche Möglichkeiten zur Verfügung wie beispielsweise Anpassung der Mutations-Operatoren oder der Testklassen. Judy hat allerdings eine Abwertung bekommen, da einige der Konfigurationsmöglichkeiten nicht funktioniert haben, wie beispielsweise die Ausgabe von Debug-Informationen oder der erzeugten Mutationen. Klarer Sieger ist Major, da durch seine eigene domänenspezifische Sprache eine Vielzahl an Konfigurationsmöglichkeiten der Mutationen gegeben ist. Der Verlierer in Sachen Bedienbarkeit ist MuJava, da hier außer der Selektion der Mutations-Operatoren keine Konfigurationsmöglichkeiten gegeben sind.

Das letzte Bedienbarkeitskriterium ist die Automatisierbarkeit des Testprozesses. Auf diesem Punkt liegt noch einmal besonderes Augenmerk, da die benötigte menschliche

Beteiligung ein großer Kritikpunkt der Mutationstests ist. Hier ist erfreulicherweise festzustellen, dass alle MTW außer MuJava komplett automatisierbar sind. MuJava hat eine ausreichende Bewertung bekommen, da es Ant nicht unterstützt und die Bedienung über eine grafische Oberfläche erfolgt.

5 Fazit

Die Ergebnisse der Evaluation haben gezeigt, dass MTW durchaus eine Existenzberechtigung in der modernen Softwareentwicklung haben. Ihre Anwendung ist voll automatisierbar und auch die Performanzprobleme sind mittlerweile kein großes Problem mehr. Allerdings lässt die Bedienbarkeit teilweise noch stark zu wünschen übrig und auch die Nachvollziehbarkeit der ausgegebenen Ergebnisse ist nicht immer gewährleistet. Daher können von den getesteten MTW nur Jumble und Pitest für einen praktischen Einsatz empfohlen werden. Major eignet sich nur, falls man schon Erfahrung mit Mutationstests gesammelt hat und ein hoch-konfigurierbares MTW benötigt. Von MuJava und Judy ist wegen der mangelnden Nachvollziehbarkeit des berechneten Mutation Scores abzuraten.

Obwohl es ständig neue Entwicklungen im Bereich der Mutationstests gibt und die getesteten MTW ihre Tauglichkeit bewiesen haben, ist die praktische Anwendung immer noch wenig verbreitet. Dies ändert sich hoffentlich in den nächsten Jahren, wenn die Vorurteile gegenüber den MTW beseitigt werden können. Dazu wäre es natürlich auch hilfreich, wenn die Bedienbarkeit weiter verbessert würde. Vielleicht wäre es hier sinnvoll, bei der Entwicklung neuer MTW einen größeren Fokus auf ihre Praxistauglichkeit zu legen.

Literaturverzeichnis

[ABC82] Adrion, W. R.; Branstad, M. A.; Cherniavsky, J. C.: Validation, verification, and testing of computer software. ACM Computing Surveys (CSUR), Volume 14, Nr. 2, S. 159–192, 1982.

[BA82] Budd, T. A.; Angluin, D.: Two Notions of Correctness and Their Relation to Testing. Acta Informatica, Volume 18, Nr. 1, S. 31–45, 1982.

[Bo79] Boehm, B. W.: Guidelines for verifying and validating software requirements and design specifications. In Proc. EURO IFIP'79, S. 711-719, London, Sep. 1979.

[Co14a] Coles, H.: Pitest., http://pitest.org, Stand: August 2014.

[Co14b] Coles, H.: Pitest Mutators., http://pitest.org/quickstart/mutators/, Stand: August 2014.

[CT03] Chevalley, P.; Thévenod-Fosse, P.: A mutation analysis tool for Java programs. Internal Journal on Software Tools for Technology Transfer, Volume 5, Nr. 1, S. 90-103, 2003.

[DLS78] DeMillo, R. A.; Lipton, R. J.; Sayward, F. G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11 (4), S. 34–41, 1978.

[Ha77] Hamlet, R. G.: Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering, Volume 3, Nr. 4, S. 279-290, 1977.

[Ir07] Irvine, S. A.; Pavlinic, T.; Trigg, L.; Cleary, J. G.; Inglis, S.; Utting, M.: Jumble Java byte code to measure the effectiveness of unit tests. In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, S. 169–175, 2007.

[Ja12] Jalbert, K.: Triangle Example., https://github.com/david-schuler/javalanche/tree/master /examples/triangleJunit4, Stand: 30. Januar 2012.

[JH08] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, S. 249–258, 2008.

[JH11] Jia, Y.; Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, Volume 37, Nr. 5, S. 649–678, Sep. 2011.

[Ju07] Jumble Mutations., http://jumble.source-forge.net/mutations.html, Stand: 6. Juni 2007.

[Ju14] Just, R.: The Major mutation framework: Efficient and scalable mutation analysis for Java. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014.

[Ma14] Major Mutation Framework., http://mutation-testing.org/doc/, Stand: Juni 2014.

[MB99] Mresa, E. S.; Bottaci, L.: Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing Verification and Reliability, Volume 9, Nr. 4, S. 205–232, 1999.

[MO11] Ma, Y.-S.; Offutt, J.: Description of class mutation operators for java. 2011.

[MO14] Ma, Y.-S.; Offutt, J.: Description of method-level mutation operators for java. 2014.

[Mo90] Morell, L. J.: A theory of fault-based testing. IEEE Transactions on Software Engineering, Volume 16, Nr. 8, S. 844–857, 1990.

[MR10] Madeyski, L.; Radyk, N.: Judy – a mutation testing tool for Java. IET Software, Volume 4, Nr. 1, S. 32, 2010.

[Of11] Offutt, J.: A mutation carol: Past, present and future. Information and Software Technology, Volume 53, Nr. 10, S. 1098–1107, 2011.

[OMK04] Offutt, J.; Ma, Y.-S.; Kwon, Y.-R.: An experimental mutation system for Java. ACM SIGSOFT Software Engineering Notes, Volume 29, Nr. 5, S. 1–4, 2004.

[OU01] Offutt, J.; Untch, R. H.: Mutation 2000: Uniting the orthogonal. Mutation testing for the new century, Springer US, S. 34-44, 2001.