Revival der Mutationstests

Dehla Sokenou
WPS – Workplace Solutions
dehla.sokenou[at]wps.de

1 Motivation

Einige Techniken brauchen länger, um sich durchzusetzen. Dies gilt insbesondere in der IT, wo einige Konzepte bereits sehr früh beschrieben wurden, es aber eine Reihe von Jahren gebraucht hat, bis sie sich auch in der Praxis etabliert haben. In manchen Fällen scheiterte der praktische Einsatz an mangelnder Rechenleistung und Werkzeugunterstützung, so auch im Fall der Mutationstests.

Ein kurzer Abriss zur Historie sowie eine Erklärung für die auch Anfang der 2000er Jahre noch vorhandene Zurückhaltung beim Einsatz von Mutationstests findet sich in [1]. Darin schreiben die Autoren, dass die Idee der Mutationstests bereits im Jahr 1971 von Richard Lipton vorgestellt und erst im Jahr 1978 zusammen mit anderen Autoren in [2] veröffentlicht wurde. Eine der wichtigsten Thesen in diesem Papier ist die Aussage: "Programmers have one great advantage that is almost never exploited: they create programs that are *close* to being correct!" [2]. Diese auch als *competent programmer hypothesis* bekannte Aussage impliziert, dass Programmierer nicht absichtlich falsche Programme implementieren, sich aber oft trotzdem kleine Fehler in Programmcode finden. Zudem stellen die Autoren die These auf, dass komplexe Fehler durch die Kopplung von kleineren Fehlern entstehen [2], auch bekannt als *coupling effect hypothesis*. Beide Thesen bilden die Grundlage für die Anwendung von Mutationstests. Denn wenn sich kleine Fehler im Programm durch Tests finden lassen, so haben diese auch das Potential, größere Probleme im Quellcode aufzudecken. Mutationstests stellen somit ein Maß zur Messung der Qualität der Tests zur Verfügung. Erste Werkzeuge wurden ebenfalls Ende der 1970er, Anfang der 1980er Jahre vorgeschlagen und implementiert (z.B. PIMS und Mothra für Fortran) [1].

Nachdem es lange ruhig um die Mutationstests war, nehmen die Veröffentlichungen zum Thema Mutationstest seit den 2020er Jahre deutlich zu. Dies liegt aus unserer Sicht auch daran, dass es inzwischen Werkzeuge gibt, die moderne Programmiersprachen unterstützen, bspw. PIT für JVM-Sprachen wie Java und Kotlin (erste Versionen in 2014) [3] und Stryker für Javascript, .Net und Scala (erste Versionen in 2016) [4]. Diese erleichtern das Ausführen der Mutationstests durch die einfache Integration in eine Build-Pipeline. Ein Nachteil bleibt jedoch: Mutationstests kosten auch heute noch aufgrund ihrer Arbeitsweise Zeit und Ressourcen und verlangsamen in der Regel die Laufzeit von Build-Pipelines signifikant. Es gilt also die Vor- und Nachteile des Einsatzes abzuwägen.

Wir stellen im Folgenden einige Beispiele aus realen Projekten vor, um zu zeigen, warum für uns die Mutationstests einen Mehrwert bedeuten. Zuvor werfen wir jedoch einen kurzen Blick auf die Funktionsweise von Mutationstests.

2 Funktionsweise

Beim Mutationstest wird zunächst für ein zu testendes Programm (Original) die erstellte Testsuite ausgeführt und die Ergebnisse protokolliert (siehe Abb. 1). Moderne Mutationstestwerkzeuge setzen dabei voraus, dass der Test des Originals fehlerlos durchläuft.

Im Anschluss werden eine Reihe von Mutanten erzeugt. Jeder der Mutanten unterscheidet sich vom Original durch kleine Fehler, wie diese einem kompetenten Programmierer typischerweise unterlaufen können. Typische Fehler sind dabei u.a. eins-daneben-Fehler, also die Verwendung von minimal falschen Grenzen, oder die Vertauschung von Variablen oder falsche Operatoren, z.B. wird ein Plus im Original zu einem Minus im Mutanten. Danach wird die Testsuite gegen jeden Mutanten ausgeführt und das jeweilige Testergebnis mit dem des Originals verglichen. Liefert der Test gegen den Mutanten ein anderes Ergebnis als der Test gegen das Original, so spricht man von einem "gekillten" Mutanten. D.h. für moderne Mutationstestwerkzeuge, dass mindestens ein Test der Testsuite gegen den Mutanten fehlschlagen muss.

Die Qualität der Testsuite ergibt sich dann, indem die Rate der "gekillten" Mutanten zu der aller Mutanten ins Verhältnis gesetzt wird. Je höher die Anzahl der "gekillten" Mutanten, desto höher ist die Eignung der Tests, Fehler zu finden. Die Zahl wird auch als *Mutation Kill Rate* oder *Mutation Coverage* bezeichnet.

Vergleicht man die Mutation Coverage mit der beim Testen gemessenen Code Coverage, so zeigt letztere nur die Qualität der ACTs (*Habe ich alle Teile meines Source-Codes mit den Tests "getroffen"?*), während erstere auch die Qualität der ASSERTs einbezieht (*Sind meine Tests gut genug?*).

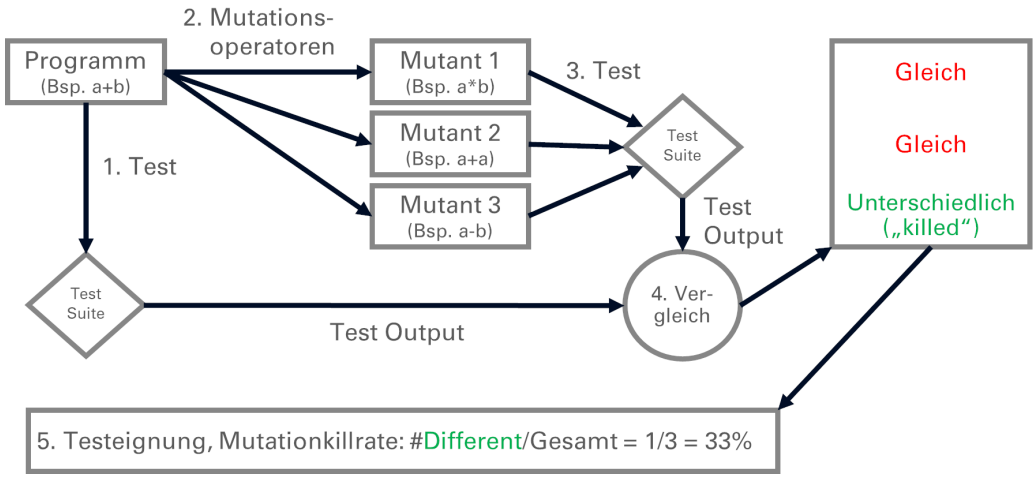


Abb. 1: Funktionsweise von Mutationstests

3 Einige Beispiele aus der Praxis

Wir berichten aus zwei verschiedenen Projekten, in denen Mutation Coverage zum Einsatz kommt. Das zweite Projekt ist eines, in dem die Entwickler entschieden hatten, testgetrieben zu entwickeln. In diesem Projekt war die Untersuchung, ob Mutationstests einen Mehrwert bringen, besonders interessant, da durch die Vorgehensweise die Qualität der Tests bereits sehr hoch ist und Testlücken unwahrscheinlicher sind.

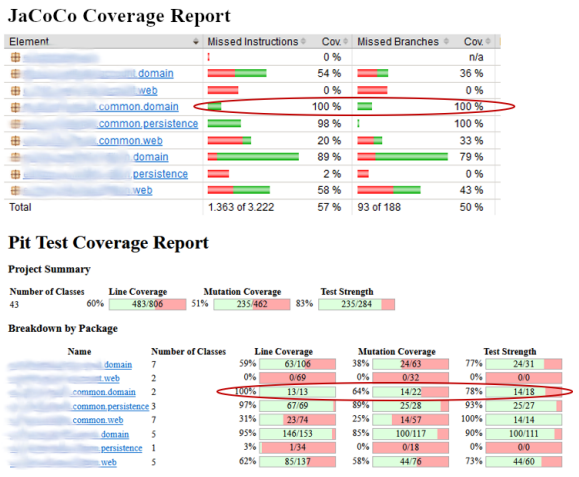


Abb. 2: Vergleich von Code Coverage (JaCoCo, oben) und Mutation Coverage (PIT, unten)

Sehen wir uns zunächst ein Beispiel aus dem ersten Projekt an, das nicht testgetrieben entwickelt wird. Abb. 2 zeigt die Reports zu Code Coverage (erstellt mit JaCoCo [5]) und Mutation Coverage (erstellt mit PIT [3]) auf derselben Code-Basis und mit derselben Testsuite. Zunächst einmal weist das Mutationstestwerkzeug nicht nur die Mutation Coverage aus, sondern auch die Code Coverage, hier auf Anweisungsebene. Die sollte als Hinweis betrachtet werden, dass vor der Analyse der Mutationstestergebnisse zunächst einmal die Tests eine hohe Code-Überdeckung erreichen sollten. Auf Basis einer sowieso schon schlechten Testsuite lassen sich nur sehr begrenzt zusätzliche Aussagen durch eine Mutationsanalyse ableiten.

Schauen wir aber einmal auf die Stellen im Code, die gut durch Tests abgedeckt sind. Was auffällt ist, dass das Package *...common.domain* zwar eine sehr gute Code Coverage von 100% Line und Branch Coverage aufweist, jedoch nur eine Mutation Coverage von 64% und damit nur eine durchschnittliche Güte der Tests.

Wobei auch hier ein genauer Blick angebracht ist, denn nicht aus jedem überlebenden Mutant ergibt sich Handlungsbedarf. Neben den Mutanten, die auf echte Lücken im Test hinweisen, gibt es auch Fälle, wo die Mutation das Programm in einen invaliden, nicht mehr lauffähigen Zustand versetzt oder die Mutation des Mutanten äquivalent zum Verhalten des Originals ist. Diese Fälle lassen sich nur durch eine genaue Analyse des Mutationstestsberichts identifizieren.

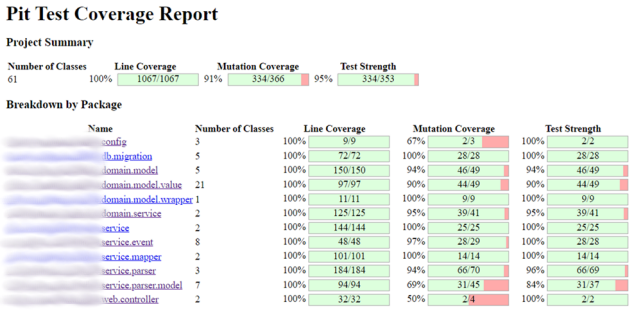


Abb. 3: Ursprüngliche Mutation Coverage

Alle validen, nicht äquivalenten Mutanten, die überlebt haben, sollten dazu genutzt werden, die Testsuite zu verbessern, in der Regel, in dem weitere Tests oder zurätzliche ASSERTs hinzugefügt werden. Wie dies im konkreten Fall aussehen kann, zeigen einige Beispiele aus dem zweiten Projekt, in dem testgetrieben gearbeitet wird. Ausgangspunkt war ein PIT-

Report, der eine Mutation Coverage von 91% auswies (siehe Abb. 3).

```
enum class BookingItemType(val value: String) {

    PRIVATE("privat"),
    BUSINESS("geschäftlich"),
    BUSINESS_ABROAD("geschäftlich (auswärtig)")

}

replaced return value with ""
```

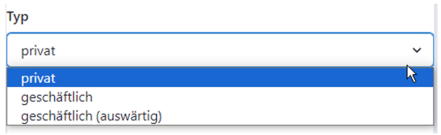


Abb. 4: Das leere UI

Bei der Analyse haben wir uns alle Stellen im Code angesehen, bei denen es keine Tests gab, die die entsprechenden Mutanten gekillt haben. Eine der Stellen, die wir analysiert haben, zeigt Abb. 4. Es werden Enums definiert, die zusätzlich einen String als Wert enthalten. Der überlebende Mutant ersetzt diesen Wert durch einen leeren String. Da diese Strings im UI angezeigt werden, führen leere Werte zu einer Auswahl, die eine Reihe von scheinbar leeren Einträgen anzeigt und so für den Nutzer unbedienbar ist. Es ist sinnvoll, hier einen Test zu ergänzen, der prüft, ob zu einem Enum der passende Wert definiert ist, und der bei versehentlichen Änderungen anschlägt.

```
fun saveProject(projectValues: ProjectValues): Project {
    var project = repository.loadProject(project)
    project.updateWith(projectValues)

    project = repository.save(project)
    return project
}


removed call to …
```

Abb. 5: Nichts in der Datenbank

Abb. 5 zeigt eine weitere Fundstelle, an der Handlungsbedarf bestand. Da testgetrieben entwickelt wird, werden viele Funktionen mit Hilfe von Mocks getestet. In diesem Fall wurde vergessen zu prüfen, ob der Datenbankaufruf, der im Test gemockt wurde, auch tatsächlich vom umgebenden Code aufgerufen wird. Es gab keinen Test, der garantierte, dass die Daten wirklich in der Datenbank gespeichert werden. Ein versehentliches Löschen der betroffenen Code-Zeile hätte zu keinem Fehlschlag der Testsuite geführt, aber immense Auswirkungen auf das Verhalten des Programms gehabt.

Bei der Analyse stießen wir auf eine weitere Stelle im Code, wo das Streichen einer Anweisung von keinem Test aufgedeckt wurde. Trotz Bemühungen gelang es uns nicht, einen Test zu erzeugen, der diese Stelle abgedecken könnte. Bei genauerem Hinsehen

```
fun checkProject(project: Project) {
    checkProjectDates(project)
    checkProjectTimes(project)
}

fun checkProjectDates(project: Project) {
    require(project.date.start <= project.date.end) {
       "Der Starttag muss vor dem Endtag liegen oder gleich sein."
    }
    checkProjectTimes(project)
}

removed call to …
```

Abb. 6: Doppelter Aufruf

wurde festgestellt, dass durch ein Refactoring die nicht abzudeckende Anweisung (*checkProjectTimes*) bereits in der Funktion in der darüberliegenden Anweisung (*checkProjectDates*) aufgerufen wurde und somit an dieser Stelle keinen Mehrwert mehr hatte. Die Analyse wurde dadurch erschwert, dass beide Funktionen in einer separaten Bibliothek implementiert sind.

```
class BookingItem(val start: LocalDate, val end: LocalDate) {

    fun checkDateInFuture() {
        require(start > LocalDate.now()) {
            "Der Start muss in der Zukunft liegen."
        }
    }

}


changed conditional boundary
```

Abb. 7: Grenzen beachten

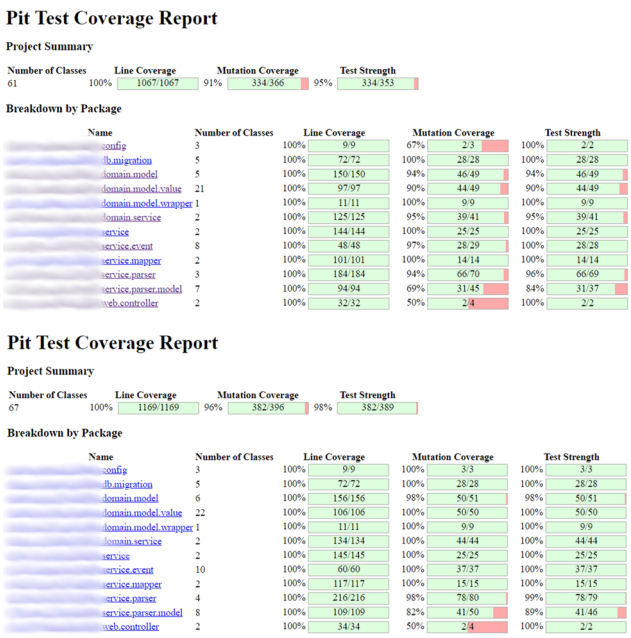
Das nächste Beispiel zeigt ein klassisches Problem von mangelndem Testentwurf auf (siehe Abb. 7). Der Mutationstest manipuliert die Grenze einer Bedingung, also ersetzt bspw. ein größer durch ein größergleich, und kein Test killt diesen Mutanten. In diesem Fall wurden die Tests nicht mit Hilfe der Grenzwertanalyse entworfen. Das Einbeziehen von Testdaten auf und direkt neben der Grenze war hier zielführend.

```
fun generateRandomValidPassword(length: Int): String {
    var password = ""
    do {
        password = generateRandomPassword(length)
    } while (!isValidPassword(password))
    return password
}

private fun isValidPassword(password: String): Boolean {
    // some implementation
}


replaced return value with true
```

Abb. 8: Unerreichbarer Code

Als schwierigster Fall erwies sich von unseren Beispiel das in Abb. 7 dargestellte. Hier ist Code scheinbar unerreichbar. Es gelang uns auch mit viel Aufwand nicht, einen Test zu erzeugen, der mindestens einmal ein invalides Passwort erzeugte. Gleichzeitig birgt der dargestellte Code auch das Risiko einer End-

losschleife. Die Frage blieb offen, ob es sich um eine falsche Annahme handelte oder nicht. Da relativ zeitnah auf ein anderes Verfahren zur Passwortgenerierung umgestellt wurde, blieb die Frage und eine abschließende Klärung unbeantwortet. Der Mutationstest hat aber in jedem Fall das Entwicklungsteam sensibilisiert, zukünftig auf solch risikobehafteten Implementierungen stärker auch im Reviewprozess zu achten.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
61	100%	1067/1067	91%	334/366	95%	334/353

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
[redacted].config	3	100%	9/9	67%	2/3	100%	2/2
[redacted].db.migration	5	100%	72/72	100%	28/28	100%	28/28
[redacted].domain.model	5	100%	150/150	94%	46/49	94%	46/49
[redacted].domain.model.value	21	100%	97/97	90%	44/49	90%	44/49
[redacted].domain.model.wrapper	1	100%	11/11	100%	9/9	100%	9/9
[redacted].domain.service	2	100%	125/125	95%	39/41	95%	39/41
[redacted].service	2	100%	144/144	100%	25/25	100%	25/25
[redacted].service.event	8	100%	48/48	97%	28/29	100%	28/28
[redacted].service.mapper	2	100%	101/101	100%	14/14	100%	14/14
[redacted].service.parser	3	100%	184/184	94%	66/70	96%	66/69
[redacted].service.parser.model	7	100%	94/94	69%	31/45	84%	31/37
[redacted].web.controller	2	100%	32/32	50%	2/4	100%	2/2

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
67	100%	1169/1169	96%	382/396	98%	382/389

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
[redacted].config	3	100%	9/9	100%	3/3	100%	3/3
[redacted].db.migration	5	100%	72/72	100%	28/28	100%	28/28
[redacted].domain.model	6	100%	156/156	98%	50/51	98%	50/51
[redacted].domain.model.value	22	100%	106/106	100%	50/50	100%	50/50
[redacted].domain.model.wrapper	1	100%	11/11	100%	9/9	100%	9/9
[redacted].domain.service	2	100%	134/134	100%	44/44	100%	44/44
[redacted].service	2	100%	145/145	100%	25/25	100%	25/25
[redacted].service.event	10	100%	60/60	100%	37/37	100%	37/37
[redacted].service.mapper	2	100%	117/117	100%	15/15	100%	15/15
[redacted].service.parser	4	100%	216/216	98%	78/80	99%	78/79
[redacted].service.parser.model	8	100%	109/109	82%	41/50	89%	41/46
[redacted].web.controller	2	100%	34/34	50%	2/4	100%	2/2

Abb. 9: Mutation Coverage nach Verbesserung der Tests (vorher oben, nachher unten)

Abb. 9 zeigt den Erfolg der Maßnahmen zur Verbesserung der Tests. Die Mutation Coverage ist von 91% auf 96% gestiegen. In den meisten Packages konnte die Mutation Coverage auf 100% gesteigert werden. Zu beachten ist, dass die Code-Basis nicht identisch ist, da das Projekt weiterentwickelt wurde, während die Schwächen in den Tests behoben wurden. Bei den verbleibenden, nicht gekillten Mutanten handelt es sich um äquivalente Mutanten.

Neben den ganz konkreten Verbesserungen einzelner Tests hat das Entwicklungsteam die eigenen Testskills verbessert. Mutationstests weisen durch ihre Funktionsweise auf Schwächen im Testdesign, bei der Auswahl von Testdaten und an manchen Stellen auch auf Probleme im Produktivcode hin.

4 Fazit

Wie die Beispiele zeigen, lohnt sich der Einsatz von Mutationstests selbst in Projekten, in denen testgetrieben entwickelt wird. Mutationstests geben Hinweise auf Lücken im Tests, fordern auf der anderen Seite aber auch Best Practices wie die Anwendung der Grenzwertanalyse ein.

Da sie dabei ressourcenhungrig sind und in der Build-Pipeline relativ viel Zeit brauchen, sind sie typische Kandidaten für einen Nightly- oder Weekly-Build. Wir lassen die Mutationstests bspw. einmal wöchentlich auf allen Repositories des Projekts laufen und einen gemeinsamen Report erzeugen. Wichtig ist, dass solch ein Bericht auch gelesen und Handlungen abgeleitet werden.

Das Einbeziehen von Tests höherer Teststufen wie Integrations- und End2End-Tests verlangsamt die Ausführung der Mutationstests noch einmal deutlich stärker und bringt in manchen Fällen keinen Mehrwert. Zudem ist hier zu beachten, dass eine noch stärkere Parallelisierung der Tests auf den verschiedenen Mutanten durch das Mutationstestwerkzeug vorgenommen werden kann und somit die einzelnen Tests noch stärker isoliert werden müssen, als es bei der Ausführung der Testsuite auf nur einem zu testenden System der Fall ist.

Mutationstests haben durch die bessere technische Unterstützung und ein Umfeld, in dem Testautomatisierung die Regel und nicht mehr die Ausnahme ist, an Bedeutung gewonnen. Da die Ergebnisse interpretationsbedürftig sind, werden sie jedoch oft einfach nur ausgeführt, ohne dass eine Verschlechterung der Mutation Coverage aktiv beobachtet und Tests daraufhin aktiv verbessert werden. Mutationtests lohnen sich aus unserer Sicht, aber nur, wenn sie als Teil des Entwicklungsprozesses verstanden und einbezogen werden.

Referenzen

[1] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 34–44. Springer US, 2001.

[2] Richard Demillo, R.J. Lipton, and Fred Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34 – 41, 05 1978.

[3] Pitest (PIT). https://pitest.org/.

[4] Stryker Mutator. https://stryker-mutator.io/.

[5] Java Code Coverage (JaCoCo). https://www.jacoco.org/.