Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito

Robert Garmann
Hochschule Hannover
Fakultät IV - Wirtschaft und Informatik
Ricklinger Stadtweg 120
30459 Hannover
robert.garmann@hs-hannover.de

Zusammenfassung

Wir demonstrieren, dass der Einsatz von Test Doubles bei der automatisierten Bewertung studentischer Java-Programme besondere Sicherheitsfragen aufwirft. Wie skizzieren kurz eine mögliche Antwort auf diese Fragen.

Im ersten Teil demonstrieren wir die Erstellung eines sog. AssignmentGraders, der in der Lage ist, interne Programmschnittstellen eines maschinell zu bewertenden Java-Programms zu beobachten. Beispielhaft nutzen wir den Autobewerter Graja in Verbindung mit mockito Test Doubles. Wir zeigen, wie man Leistungsaspekte einer studentischen Lösung einzeln und gezielt automatisiert bewertet. Bei einer Beschränkung auf die Beobachtung externer Programmschnittstellen würden sich diese Aspekte der gezielten Bewertung entziehen. Im zweiten Teil beleuchten wir Bedrohungen des das zu bewertende Programm ausführenden Systems und gewichten ausgewählte Ursachen. Wir konzentrieren uns auf Bedrohungen, die mit Mitteln der Java-2-Sicherheitsarchitektur beherrschbar sind und liefern einige Beispiele zur Abgrenzung. Die Eigenheiten der Java-2-Sicherheitsarchitektur werfen im Kontext des Test Double Einsatzes Fragen auf, die wir im dritten Teil erörtern. Test Doubles sollten in einem anderen Schutzbereich der Java-2-Sicherheitsarchitektur als der studentische Code liegen. Um eine privilegierte Ausführung der Test Doubles auch dann zu forcieren, wenn der Aufruf aus studentischem Code erfolgt, schlagen wir den Einsatz von Proxy-Objekten vor. Um auch Nicht-Spezialisten der beschriebenen Sicherheitsbelange die Autorenschaft von AssignmentGradern zu ermöglichen, entwickeln wir eine Wrapper-Bibliothek für mockito, deren zentrale Entwurfsideen wir kurz skizzieren.

1. Einleitung

Automatisierte Bewertung studentischer Programme wird in der Informatik-Lehre zurzeit überwiegend im formativen Assessment eingesetzt. Zeit- und ortsunabhängiges, maschinell erstelltes, quantitatives und qualitatives Feedback soll Studierende in die Lage versetzen, ihre (Teil-)lösungen zu Programmierübungen schrittweise zu verbessern.

Studentische Programme müssen funktionale und nichtfunktionale Anforderungen erfüllen, die mit statischen oder dynamischen Analyseverfahren untersucht werden können. Für die Analyse der „inneren Qualität“ des Bewertungsobjekts spielt die dynamische Analyse an internen Programmschnittstellen (Methodenaufruf) eine wichtige Rolle. Unittest-Frameworks wie JUnit wurden dazu entwickelt, kleinste Einheiten eines Testobjekts durch Methodenaufruf und Verifizierung des Ergebnisses dynamisch zu untersuchen. Um die zu untersuchende Einheit für die Dauer des Tests vom Restsystem zu isolieren werden Test Doubles eingesetzt, die z. B. mittels der Bibliothek mockito [Ka13] erstellt werden. An einer Beispielaufgabe verdeutlichen wir in Abschnitt 2, wie der JUnit-basierte Autobewerter Graja und mockito zur Beobachtung interner Programmschnittstellen genutzt werden können.

Das studentische Programm wird bei der dynamischen Analyse ausgeführt. Sicherheitsmaßnahmen auf dem Bewertungsrechner sind notwendig, um ungewolltes oder böswilliges Fehlverhalten des studentischen Programms zu unterbinden. In Abschnitt 3 stellen wir das betrachtete Modell eines Bewertungssystems vor und diskutieren relevante Sicherheitsaspekte.

Bibliotheken wie mockito wurden nicht für die automatisierte Programmbewertung entworfen. Die Sicherheit der Testumgebung spielt beim (Regressions-)Test großer Softwaresysteme meist eine untergeordnete Rolle. Im Bewertungsszenario jedoch wirft der Einsatz einer solchen Bibliothek essentielle Sicherheitsfragen auf, für die wir in Abschnitt 4 eine mögliche Lösung vorstellen.

2. Beobachtung programminterner Schnittstellen

Die Frage, wann ein studentisches Programm gut ist, ist nicht leicht zu beantworten. Neben dem Kriterium der funktionalen Korrektheit muss das Programm nichtfunktionale Kriterien erfüllen (Effizienz, Änderbarkeit, …). Einige dieser Kriterien lassen sich recht gut automatisiert untersuchen. Grundsätzlich gibt es zwei Herangehensweisen bei der maschinellen Untersuchung eines Bewertungsobjektes [SL05]: die statische Code-Analyse (bspw. in [SBG10]) und die dynamische Code-Analyse (bspw. in [KSZ02], [Ed03]). In diesem Aufsatz befassen wir uns mit der letztgenannten Herangehensweise, der Beobachtung des *Verhaltens* eines studentischen Java-Programms durch Tests.

Bei der dynamischen Code-Analyse beobachtet man das Laufzeitverhalten des Bewertungsobjekts an beobachtbaren Schnittstellen. Hervorragend geeignet für die Überprüfung der funktionalen Korrektheit des Bewertungsobjekts sind Ein- und Ausgabeschnittstellen des Benutzers, die auch leicht einer maschinellen Bedienung bzw. Abfrage zugänglich sind. Interessante Schnittstellen für die Bewertung des studentischen Programmverhaltens sind u. a.[1] Tastatureingabe, Consolenausgabe, Mauseingabe, GUI-Ausgabe [TE08], Kommandozeilenargumente beim Programmstart, Dateiein-/-ausgabe. Allen genannten Schnittstellen gemein ist, dass sie prinzipiell auch vom

[1] Über diese nicht abschließende Aufzählung hinaus sind selbstverständlich viele weitere Schnittstellen denkbar (Umgebungsvariable, Audio/Video, Netzwerk, etc.).

Benutzer des studentischen Programms bedient werden können. Um ein Bewertungsobjekt nun automatisiert mit Eingaben über die genannten Schnittstellen zu versorgen und die Ausgaben maschinell auszuwerten, werden Autobewerter wie der hier eingesetzte Graja [St13] verwendet.

Ein automatisiertes Bewertungsverfahren nimmt bei Nutzung der oben genannten Schnittstellen das Bewertungsobjekt als „black box" wahr und kann wenig über die Qualität der inneren Struktur des Bewertungsobjekts erfahren. Gut überprüfbar ist die funktionale Korrektheit und die Überprüfung einiger nichtfunktionaler Eigenschaften (z. B. Ressourcenverbrauch). Um mehr über die „innere Qualität" des Bewertungsobjekts zu erfahren, ist es hilfreich, das studentische Programm an internen Programmschnittstellen zu beobachten. Eine geeignete Schnittstelle ist der Methodenaufruf mit Methodenparametern als Eingabe und Methodenrückgabe als Ausgabe.

Im Folgenden geben wir einen kurzen Überblick über die Funktionen von Graja und stellen eine Beispielaufgabe vor. Danach zeigen wir, wie studentische Lösungsversuche für diese Aufgabe mit Graja unter Einsatz von Test Doubles bewertet werden können.

2.1 Graja

Graja ("**Gra**ding **Ja**va programs") ist i. w. ein Überbau über dem JUnit[2]-Testframework und hat seine Charakterzüge von seinem Vorbild WebCAT [Ed03] geerbt. Graja bietet u. a. folgende Funktionen:

- Extrahieren und Übersetzen von Code aus einem ZIP-Archiv,
- Umleitung von Standard-Ein/Ausgabeströmen zur Vereinfachung der maschinellen Steuerung des Bewertungsobjekts,
- intelligente Vergleiche von erwarteter und beobachteter Ausgabe eines Bewertungsobjekts,
- bequeme „reflection"-basierte Analyse studentischen Codes,
- Generierung formatierter, zielgruppenorientierter, und nach Kritikalität abgestufter Bewertungskommentare,
- abgestufte Punktevergabe für verschiedene vom Dozenten festzulegende Bewertungsaspekte,
- Übersetzung und Ausführung von „on the fly" generiertem Java-Code während der Bewertung,
- Begrenzung der genutzten Systemresourcen (Laufzeit, persistenter Speicherplatz, Hauptspeicher),
- Ausführung des studentischen Codes und des Bewertungscodes in verschiedenen, separat abgesicherten Schutzbereichen.

2.2 Beispielaufgabe

Wir betrachten eine für Zwecke der konzentrierten Darstellung stark vereinfachte Programmieraufgabe (s. Abbildung 1).

Diese Aufgabe verlangt vom Studierenden nicht nur die Programmierung einer einfachen mathematischen Formel, sondern die Implementierung einer vorgegebenen Schnittstelle. Eine korrekte Lösung kann nicht nur Hypotenusen berechnen (2); eine korrekte Lösung kann darüber hinaus Quadrate berechnen (1) und sie setzt Wiederverwendung durch Delegation ein (3). Für jeden der im vorstehenden Satz markierten Bewertungsaspekte möchten wir Teilpunkte vergeben.

[2] http://junit.org/

Berechnen Sie die Länge der Hypotenuse eines rechtwinkligen Dreiecks. Teilen Sie den Programmcode auf zwei Klassen auf: `QuadratImpl` implementiere das folgende vorgegebene Interface:

```
package de.hsh.prog;
public interface Quadrat {
  double quadrat(double v); // berechnet v²
}
```

Schreiben Sie eine weitere Klasse `Hypo` mit einer statischen Methode `hypo`, die als ersten Parameter ein `Quadrat`-Objekt und als zwei weitere Parameter die Längen zweier Katheten erhält. Gewünschter Rückgabewert ist die Länge der Hypotenuse. Quadrierungen soll `hypo` an das `Quadrat`-Objekt delegieren.

Abbildung 1: Stark vereinfachte Beispielaufgabe

2.3 Bewertung mit Graja

Den Bewertungsaspekt (1) überprüft der in Abbildung 2 dargestellte, mit Graja kompatible Code.

Der Aufruf der internen Programmschnittstelle des studentischen Codes ist in Programmzeile 06 zu finden. Davor wird eine Instanz der studentischen Klasse erzeugt und danach das beobachtete Ergebnis überprüft. Diese Art von Bewertungscode ist dem in der professionellen Softwareentwicklung verbreiteten Testcode für Modultests sehr ähnlich. Spezialitäten der hier betrachteten Programmbewertung sind die in Zeile 01 vergebenen Teilpunkte und der in Zeile 08f zu findende ausführliche Fehlerhinweis. Weiterhin besonders ist die von Graja angebotenen Reflection-Bibliothek (Zeile 04f), die etwaige Exceptions in verständliche Fehlermeldungen verwandelt.

```
01 @Test @ScoringWeight(35.0)
02 public void pruefeQuadratMethode() {
03   double a= 3, expected= a*a;
04   Quadrat studentImpl= ReflectionSupport.create(
05     getSubclassForName("QuadratImpl",Quadrat.class));
06   double observed= studentImpl.quadrat(a);
07   org.junit.Assert.assertEquals(
08     "quadrat("+a+")="+observed+" (erwartet :"+expected
09     +")", expected, observed, 1E-5);
10 }
```

Abbildung 2: Bewertung des Aspekts (1)

```
11 @Test @ScoringWeight(25.0)
12 public void pruefeGesamtfunktion() {
13   double a=3, b=4, cExpected=5;
14   Quadrat stub= Mockito.mock(Quadrat.class);
15   Mockito.when(stub.quadrat(a)).thenReturn(a*a);
16   Mockito.when(stub.quadrat(b)).thenReturn(b*b);
17   double c= ReflectionSupport.invokeStatic(
18    getClassForName("Hypo"),double.class,"hypo",stub,a,b);
19   org.junit.Assert.assertEquals("hypo(q,"+a+","+b+")="+c+
20    " (erwartet: "+cExpected+")", cExpected, c, 1E-5);
21 }
```

Abbildung 3: Bewertung des Aspekts (2)

```
22 @Test @ScoringWeight(40.0)
23 public void pruefeAufrufVonQuadrat() {
24   double a=3, b=4;
25   Quadrat mock= Mockito.mock(Quadrat.class);
26   ReflectionSupport.invokeStatic(
27    getClassForName("Hypo"),double.class,"hypo",mock,a,b);
28   try {
29    Mockito.verify(mock, Mockito.times(1)).quadrat(a);
30    Mockito.verify(mock, Mockito.times(1)).quadrat(b);
31   } catch (WantedButNotInvoked e) {
32    org.junit.Assert.fail("hypo muss die Quadrierung "+
33      "an den Quadrat-Parameter delegieren");
34   }
35 }
```

Abbildung 4: Bewertung des Aspekts (3)

2.4 Test Doubles: Stubs und Mocks

Ähnlich ließe sich nun der Bewertungsaspekt (2) untersuchen, indem eine `QuadratImpl`-Musterlösung als Parameter an die studentische `hypo`-Methode übergeben wird. Die erneute Verwendung einer u. U. fehlerhaften studentischen `QuadratImpl`-Klasse würde die Bewertung des Aspekts (2) negativ verzerren. Der Fehler würde in zwei Bewertungsaspekten „gezählt".

Als Alternative zu einer eigenen Musterlösung kann man ein *Test Double* [Me07] einsetzen – ein Platzhalter-Objekt, mit dem ein untersuchter Programmteil (das sog. *system under test* - kurz *SUT*) während des Tests interagiert. *Stub*-Objekte sind Test Doubles, die vorgefertigte Antworten an das SUT liefern.

Der Code in Abbildung 3 verwendet in Zeile 14 die Bibliothek mockito[3] zur Erzeugung eines Stubs, welches das Interface `Quadrat` vollständig implementiert. Die beiden folgenden Zeilen „bestücken" das Stub mit vorbereiteten Antworten auf erwartete Anfragen des SUT[4]. In Zeile 17f kommt das SUT (hier die studentische Methode `hypo`) zur Ausführung. Als Übergabeparameter erhält `hypo` das Stub-Objekt.

Der Bewertungsaspekt (3) lässt sich schließlich durch eine weitere Test Double Variante, die sog. *Mock*-Objekte realisieren (vgl. Abbildung 4). Ein Mock-Objekt wird in Zeile 25 erstellt, welches die Quadrat-Schnittstelle implementiert. Da uns bei diesem Bewertungsaspekt das Berechnungsergebnis nicht interessiert, programmieren wir keine vorbereiteten Antworten. Stattdessen befragen wir das Mock-Objekt nach Ausführung des SUT in Zeile 29f, ob es Aufrufe mit den angegebenen Parametern erhalten hat.

3. Sicherheit des Bewertungssystems

Der Ansatz, studentische Lösungen dynamisch zu analysieren, erfordert die Ausführung des studentischen Codes. Im Unterschied zu herkömmlichen Software-Tests, wo Tester und Autor einander vertrauen (bei Unit-Tests oft sogar dieselbe Person sind), wo also die Testumgebung vom zu testenden System kein böswilliges Verhalten zu befürchten hat, ist dies beim Test von studentischen Einreichungen anders. Hier muss mit Angriffen des getesteten Codes auf die Testumgebung gerechnet werden. Einige Angriffsbeispiele stellt Abschnitt 3.4 vor. Dazu kommen unabsichtliche Programmierfehler in weitaus größerem Umfang als bei Tests professioneller Software. Schutzmaßnahmen gegen solche Angriffe sind daher unabdingbar. In diesem Abschnitt wollen wir konkrete Bedrohungen und Gegenmaßnahmen beleuchten.

3.1 Schutzbereiche

Als Bewertungssystem betrachten wir einen separaten Betriebssystemprozess. Aus konzeptioneller Sicht lädt der Prozess drei verschiedene Komponenten und führt sie aus (vgl. Abbildung 5). Wir gehen davon aus, dass alle Komponenten aus Java Bytecode bestehen, die von derselben JVM-Instanz geladen werden:

- *AutograderCore*: Der Kern des Autobewerters, welcher eine Startroutine und eine Bibliothek mit bewertungsunterstützenden Funktionen enthält. Ein Beispiel einer AutograderCore-Komponente ist Graja.
- *AssignmentGrader*: vom Dozenten erstellter, auf eine spezielle Programmieraufgabe zugeschnittener Code (kann Fremdsoftware, bspw. die mockito-Bibliothek beinhalten). Die Codeschnipsel der Abschnitte 2.3 und 2.4 entstammen dieser Komponente.
- *Submission*: vom Studenten erstellter Lösungsversuch.

Jede Komponente[5] stellt einen eigenen Schutzbereich (*protection domain*) mit jeweils eigenen Rechten (*permissions*) dar.

Abbildung 5: Drei geladene Komponenten

3.2 Benötigte Rechte

Der AutograderCore benötigt für seine Funktion weitgehende Rechte (Zugriffe auf das lokale Dateisystem, Ausführen des Compilers, Verwendung eigener Klassenlader, Zugriff auf Reflection-Funktionen, etc.). Die Submission benötigt in der Regel wenige Rechte, wobei dies sicher von der Art der zu lösenden Programmieraufgabe abhängt. Dazwischen im Sinne der Rechtemenge steht der AssignmentGrader, der häufig grundlegende Rechte für Dateizugriffe, manchmal auch weitergehende Rechte besitzen muss. Eine Teilmengenbeziehung zwischen den benötigten Rechtemengen der drei genannten Schutzbereiche besteht nicht notwendigerweise.

3.3 Bedrohungspotential

Um das Bedrohungspotential[6] der einzelnen Komponenten einschätzen zu können, betrachten wir zunächst die in Tabelle 1 aufgeführten Merkmale der Komponenten-Autoren. Die beiden ersten Merkmale begünstigen bei hoher Ausprägung unabsichtlich herbeigeführte Bedrohungen, das dritte Merkmal begünstigt absichtlich herbeigeführte Bedrohungen, das vierte Merkmal begünstigt jede Art von Bedrohung.

Tabelle 1: Betrachtete Merkmale der Komponenten-Autoren

Nr.	Merkmal	Ausprägungsgrad[7] für …	Autograder-Core	Assignment-Grader	Submission
1	Geringe Programmiererfahrung		-	-	+
2	Geringe Bereitschaft, Zeit für die korrekte Programmierung der Komponente aufzuwenden		-	0	+
3	„Kriminelle Energie", d. h. Bereitschaft, Bedrohungen absichtlich herbei zu führen.		-	-	+
4	Größe der Autorengruppe		-	0	+

Vor diesem Hintergrund erwarten wir von der Submission-Komponente das größte Bedrohungspotential. Die AssignmentGrader-

[3] http://code.google.com/p/mockito/

[4] Zur Funktionsweise der auf der ersten Blick verwirrenden Aufrufsyntax folgt eine kurze Erläuterung: `Mockito.mock` erzeugt ein Objekt einer zur Laufzeit mit cglib (http://cglib.sourceforge.net/) erzeugten Subklasse von Quadrat. Das wie ein normaler Methodenaufruf aussehende `stub.quadrat(a)` führt mockito-intern dazu, dass eine ID des aktuell ausgeführten Threads in einer Datenstruktur zusammen mit dem `stub`-Objekt und dem Wert `a` abgelegt wird. Der Rückgabewert von `stub.quadrat(a)` wird zwar an `Mockito.when` weiter gereicht, dort jedoch nicht ausgewertet. Stattdessen weist die ID des aktuell ausführenden Threads den Weg zum soeben abgelegten `stub`-Objekt.

[5] In der Praxis entspricht jede Komponente einem jar-Archiv oder jeweils einer Menge von jar-Archiven. Jedem jar-Archiv wird ein Satz von Rechten zugeordnet.

[6] Wir meinen hiermit das, was in etablierten Risikoschätzmethoden wie [oV13] „likelihood" genannt wird.

[7] Legende: -=gering, 0=mittel, +=hoch

Komponente, die in der Regel von Lehrkräften oder Hilfspersonal erstellt wird, ist weniger kritisch, den AutograderCode erachten wir als Komponente mit dem geringsten Bedrohungspotential.

3.4 Bedrohungen im Einflussbereich des Java Securitymanagers

Uns interessieren ganz allgemein alle Bedrohungen der Vertraulichkeit und Integrität der Daten auf dem die JVM beherbergenden System (Dateien, Nutzerdaten, Musterlösungen, …), sowie alle Bedrohungen der Verfügbarkeit des Bewertungsdienstes. Wir wollen mit der Bedrohungsanalyse dieses Kapitels keine konkret auszumerzenden Schwachstellen identifizieren, sondern es soll hier darum gehen, einige mögliche Angriffe mit ihren Auswirkungen („impacts“) zu benennen, die genügend Motivation liefern, um den Einsatz des Java Securitymanagers für sinnvoll zu erachten. In der folgenden Darstellung nehmen wir die zu motivierende Maßnahme vorweg: den Einsatz des Securitymanagers. Die Motivation der Maßnahme durch Angriffsbeispiele folgt in Abschnitt 3.4.2.

3.4.1 Einsatz der Java Sicherheitsarchitektur

Konkret wollen wir dem Submission-Schutzbereich Rechte erteilen oder entziehen, die der Menge aller durch die Java Sicherheitsarchitektur definierten Rechte entnommen werden. Ein paar Beispiele seien hier genannt[8]:

- PropertyPermission: Auslesen oder Setzen von Umgebungsinformationen (bspw. mag eine Submission berechtigterweise das auf dem System gebräuchliche Zeichen für den Zeilenumbruch abfragen wollen - `line.separator`)
- FilePermission: Dateizugriffsrechte (bspw. Einlesen und Schreiben von Dateien im Arbeitsbereich der Submission)
- SocketPermission: Netzzugriffsrechte (bspw. soll die Submission auf eine im Netz vorgehaltene Datenbank zugreifen)
- ReflectPermission: potentes Recht zur Überwindung der private/protected-Kapselung (wird der Submission i. d. R. entzogen; Begründung am Beispiel in Abschnitt 3.4.2.5)
- RuntimePermission: diverse potente Rechte vom Nachladen weiterer Klassen bis zur Umleitung der Ein-/Ausgabe (für Submission i. d. R. nicht erforderlich und damit entziehbar)

Viele Bedrohungen können durch die Java Sicherheitsarchitektur kontrolliert werden. Insbesondere gegen unabsichtlich herbeigeführte Bedrohungen ist das Sandbox-Prinzip der Java Sicherheitsarchitektur ein wirksamer Mechanismus. Absichtlich herbeigeführte Bedrohungen können nicht völlig[9] durch die Java Sicherheitsarchitektur unterbunden werden, wie die Beispiele des folgenden Abschnitts zeigen sollen.

3.4.2 Beispiele absichtlich herbeigeführter Bedrohungen

Zur Ein- und Abgrenzung seien einige ganz konkrete Beispiele für absichtlich herbeigeführte Bedrohungen genannt. Diese Liste ist verständlicherweise nicht erschöpfend. Sie soll über die Liste des Abschnitts 3.4.1 hinaus einen Eindruck davon vermitteln, welche Bedrohungen allein mit Mitteln der Java Sicherheitsarchitektur abwendbar sind und welche zusätzliche Maßnahmen erfordern. Die Beispiele geben außerdem einen Eindruck des möglichen „impact“ [oV13] erfolgreicher Angriffe.

3.4.2.1 „rm –rf /“

Ein bösartiges Beispiel für Submission-Code, welcher versucht, alle Dateien im Dateisystem zu löschen:

```
Runtime.getRuntime().exec("rm –rf /");
```

Durch Gewährung selektiver Dateizugriffsrechte mittels der Java Sicherheitsarchitektur oder auch mittels der vom Betriebssystem verwalteten Dateirechte kann dieser Angriff gut abgewehrt werden.

3.4.2.2 Denial of service

Ein anderes Beispiel, das versucht Schaden anzurichten, in dem es die Festplatte füllt[10]:

```
FileOutputStream f=new FileOutputStream("x");
while (true) f.write(42);
```

Wenn wir davon ausgehen, dass Submission-Code in wenigstens einem Verzeichnis Schreibrechte eingeräumt werden müssen, ist dieser denial of service Angriff mit Mitteln der Java Sicherheitsarchitektur unseres Wissens nicht abwehrbar. Hier müssen dem die JVM beherbergenden Betriebssystemprozess Ressourcen in begrenztem Maße zugeteilt werden. Beispielsweise in Form einer separaten Festplattenpartition oder in Form einer Datei begrenzter Größe, die als loopback device verwendet wird.

Ähnlich müssen die Ressourcen Hauptspeicher und CPU-Zeit durch die Umgebung der JVM begrenzt werden, etwa über einen Parameter zum maximalen Speicherverbrauch beim JVM-Start bzw. mit Hilfe des Linuxkommandos timeout.

3.4.2.3 „/etc/passwd“

Es folgt ein Beispiel, das versucht, die Liste der Benutzernamen des Systems auszuspähen:

```
System.out.println(Files.readAllLines(new File("/etc/passwd").
toPath(), Charset.forName("UTF-8")));
```

Da die Datei `/etc/passwd` i. d. R. für jeden Systemnutzer lesbar ist, können vom Betriebssystem verwaltete Dateirechte nicht greifen. Dies ist eine Bedrohung, die durch selektive Rechte gemäß der Java Sicherheitsarchitektur abwehrbar ist.

3.4.2.4 Ausspähen der Musterlösung

Ist etwa der Klassenname einer Musterlösung, die häufig Teil der AssignmentGrader-Komponente ist, bekannt oder erraten worden, könnte man wie folgt versuchen, an den Bytecode der Musterlösung zu kommen:

```
InputStream is= Class.forName("org.domain.sample.Solution").
getResourceAsStream("Solution.class");
int b; while ((b= is.read()) >= 0) System.out.print(b+" ");
```

Die Standardausgabe wird dem Autor der Submission i. d. R. zur Erläuterung der Bewertung angezeigt.

Dieser Angriff ist leicht mit Mitteln der Java Sicherheitsarchitektur abwehrbar, indem das Recht zum Einlesen des AssignmentGrader-jar-Archivs nicht gewährt wird.

3.4.2.5 Ausführen der Musterlösung

Die Submission könnte versuchen, die Musterlösung einfach auszuführen und dadurch die generierten Ausgaben als die eigenen zu „verkaufen“. Dieser Angriff lässt sich abwehren, indem die Musterlösungsklasse als package private deklariert wird und indem Submission-Code das Recht zur Überwindung der Package-Kapselung nicht erhält.

---

[8] Weitere s. z. B. http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html

[9] Noch dazu muss immer wieder mit neu entdeckten Sicherheitslücken in der Java Sandbox gerechnet werden.

[10] Bei diesem Beispiel kann es sich auch um eine unabsichtlich herbeigeführte Bedrohung handeln.

4. Test Doubles bei aktivem Securitymanager

Das vorangegangene Kapitel hat motiviert, dass der Einsatz der Java Securitymanagers sinnvoll ist, um reale Bedrohungen abzuwenden. Die durch den Einsatz des Securitymanagers resultierenden Auswirkungen auf den Einsatz von Test Doubles sind Gegenstand des nun folgenden Kapitels.

4.1 Wirkung der Schutzbereiche zur Laufzeit

Die drei in Abschnitt 3.1 eingeführten Schutzbereiche werden üblicherweise der Klasse der Anwendungsbereiche (*application domains*) zugeordnet. Ein vierter Schutzbereich mit i. d. R. uneingeschränkten Rechten ist der Systembereich (*system domain*), dem u. a. die Klassen der Java-Standardbibliothek angehören.

Bei eingeschaltetem Securitymanager prüft die JVM zur Laufzeit alle an der aktuell aufzurufenden Operation beteiligten Komponenten auf dem Aufrufstapel. Programmcode eines eingeschränkten Anwendungsbereichs darf durch Aufruf einer Operation des Systembereichs selbstverständlich keine zusätzlichen Rechte erlangen. Die JVM ermittelt daher die Schnittmenge der Rechte aller am Aufrufstapel beteiligten Schutzbereiche, um eine potentiell bedrohliche Operation zu überprüfen [Go13]. Die Aufrufreihenfolge zur Laufzeit ist in der Regel[11] AutograderCore → AssignmentGrader → Submission (vgl. Abbildung 6). Mit zunehmender Aufruftiefe stehen geringere Rechte zur Verfügung.

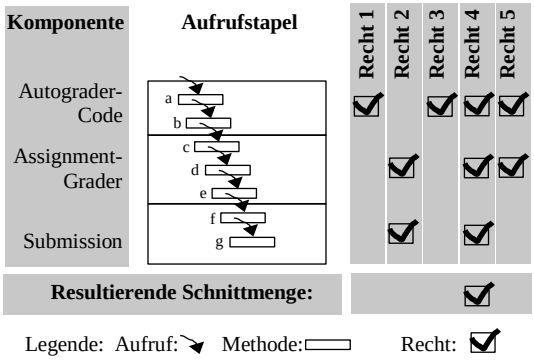

Abbildung 6: Aufrufstapel und resultierende Rechte (Beispiel)

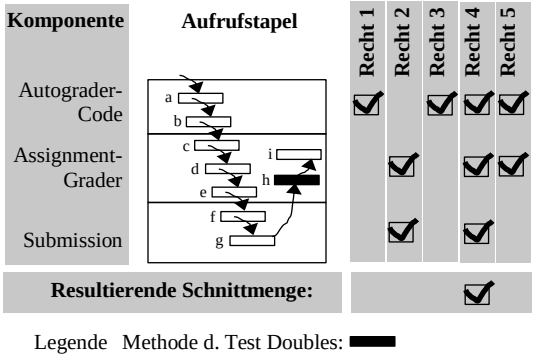

Abbildung 7: Privilegierter Callback eines Test Doubles

Damit Code des Systembereichs Rechte (bspw. Einlesen von Font-Dateien zwecks Erzeugung einer Ausgabe) erhalten kann, die aufrufender Anwendungsbereichs-Code nicht besitzt, gibt es in Java den `doPrivileged`-Aufruf. Ein solcher Aufruf wirkt als Abbruchskriterium bei der Bildung der o. g. Rechte-Schnittmenge. Oberhalb des `doPrivileged`-Aufrufs am Aufrufstapel beteiligte Schutzbereiche werden nicht berücksichtigt.

Wäre etwa im Beispiel der Abbildung 6 der Aufruf der Methode d ein privilegierter Aufruf, würde die resultierende Schnittmenge die Rechte {2, 4} umfassen.

4.2 Sicherheitsimplikationen durch Test Doubles

Beim Einsatz von Test Doubles ruft Submission-Code Test Doubles des AssignmentGrader-Schutzbereichs auf (vgl. Aufruf der Methode h in Abbildung 7). Ohne besondere Vorkehrungen würde der Code im Test Double mit den Rechten der Schnittmenge (AutograderCore ∩ AssignmentGrader ∩ Submission) ausgeführt, also der Rechtemenge {4}. Der von mockito erzeugte Bytecode eines Test Doubles ruft jedoch Methoden der mockito Bibliothek auf (Methode i), die weitgehende Rechte erfordern, u. a. das sehr potente Recht `java.lang.reflect.ReflectPermission "suppressAccessChecks"`, welches den Zugriff auf private und protected member einer Klasse erlaubt (in Abbildung 7 in der Spalte „Recht 5“ vorstellbar).

Eine Lösung kann nun nicht sein, dem Submission-Schutzbereich alle Rechte zu gewähren, die das Test Double benötigt. Mindestens das Beispiel in Abschnitt 3.4.2.5 spricht dagegen. Wäre der Aufruf der Methode h privilegiert, würden die Methoden h und i mit der Rechtemenge {2, 4, 5} ausgeführt. Leider hat der Autor des AssignmentGraders keine Möglichkeit, den Test Double-Aufruf privilegiert zu gestalten. Die Programmierschnittstelle von mockito, wie wir sie in Abschnitt 2.4 kennen gelernt haben, ermöglicht keine privilegierten Test Doubles. Der eigentliche Methodenaufruf findet im Submission-Code statt, der nicht im Einflussbereich des AssignmentGrader-Autors steht.

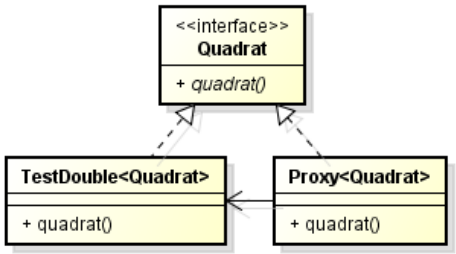

Abbildung 8: Proxy-Klasse für ein Test Double von Quadrat

Die vorgeschlagene Lösung besteht nun darin, eine Proxy[12]-Klasse in der AssignmentGrader-Komponente zu erstellen, die ein Test Double umhüllt. Beispielhaft ist in Abbildung 8 das `Quadrat`-Interface und das zugehörige, mit `Mockito.mock` erzeugte Test Double[13] zu sehen.

Die Proxy-Klasse besitzt genau die gleiche Schnittstelle wie das Original (`Quadrat`) und eine dauerhafte Referenz zum Test Double Objekt. Die Implementierung der `quadrat`-Methode der Proxy-Klasse besteht nun aus einem privilegierten Aufruf von `TestDouble<Quadrat>.quadrat`.

[11] Rückwärts gewandte Aufrufe von Methoden des Autograder-Core aus dem AssignmentGrader heraus sind ebenfalls denkbar, sind jedoch hier nicht Gegenstand der Betrachtung.

[12] http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

[13] Der von mockito erzeugten Klasse haben wir hier für Darstellungszwecke einen Namen gegeben. In Wirklichkeit ist die erstellte Klasse aus Benutzersicht anonym.

Den wenigsten Autoren von AssignmentGradern wird die Sicherheit des Autobewerters so wichtig sein, dass sie die Mühe auf sich nehmen und eine Klasse `Proxy<T>` für jeden zu doublenden Typ `T` programmieren wollen. Es handelt sich zudem um höchst spezialisierten Code, der ein tiefes Verständnis von Java Reflection und Generics erfordert.

Die naheliegende Idee ist nun, die Erzeugung der Proxy-Objekte in eine Bibliothek auszulagern, die dem AssignmentGrader-Schutzbereich angehört. Wir stellen im folgenden Abschnitt kurz die Entwurfsideen einer neuen Bibliothek *MockitoWrap* als Hülle um mockito vor, die sich um die Erzeugung von Proxy-Objekten kümmert, welche Aufrufe an Test Doubles delegiert, wobei jeder delegierte Aufruf in ein `doPrivileged`-Konstrukt „verpackt" ist.

4.3 Zentrale Entwurfsideen von Mockito-Wrap

Die zentrale Operation (vgl. Abschnitt 2.4) bei der Nutzung von mockito ist die generische Methode `Mockito.mock(Class<T> c)`[14]. Sie liefert ein Test Double vom Basistyp `T`, für das alle Methoden der Klasse `T` aufrufbar sind. Das Test Double wird im weiteren Verlauf von Methoden wie `Mockito.when` oder `Mockito.verify` verwendet. An dieser Art des Aufrufs von mockito soll sich durch die beschriebene Sicherheitsanforderung nichts ändern. Einzig der Name der Einstiegsklasse `Mockito` soll nun `MockitoWrap` heißen.

Die vollständige Beschreibung von MockitoWrap würde den Rahmen dieses Aufsatzes sprengen. Wir nennen nur einige grundlegende Entwurfsideen, die vermutlich in ähnlicher Form auf andere Mock-Frameworks übertragen werden können. MockitoWrap wird im Zuge der Erstellung neuer Programmieraufgaben sukzessive an der Hochschule Hannover weiter entwickelt.

Entwurfsidee 1: Sei `Class<T> c` der Typ, für den ein Test Double erstellt werden soll. Statt direkt `Mockito.mock(c)` aufzurufen, soll der Wrapper genutzt werden: `MockitoWrap.mock(c)`. Der Wrapper delegiert den Aufruf an das Original (`T td=Mockito.mock(c)`) und erstellt dann ein Proxy-Objekt für das Test Double `td`. Das Proxy-Objekt erhält als Konstruktorparameter ein Objekt einer neuen generischen Klasse `MockProxyInvocationHandler<T>`, die von `InvocationHandler`[15] erbt. `MockProxyInvocationHandler<T>` besitzt als Instanzattribute Referenzen auf das Test Double `td` und dessen Klasse `c`[16], so dass stets vom Proxy-Objekt zum Test Double navigiert werden kann.

Entwurfsidee 2: Die Methode `MockProxyInvocationHandler<T>.invoke` delegiert den Methodenaufruf innerhalb einer `PrivilegedAction`[17] an das Test Double `td`.

Entwurfsidee 3: Für weitere Klassen der mockito-Bibliothek werden nun (einfachere) Wrapper geschrieben werden. Bspw. besitzt `OngoingStubbingWrap<T>` ein Instanzattribut vom Originaltyp `OngoingStubbing<T>`. Methodenaufrufe wie `thenReturn` werden an das Instanzattribut delegiert. Rückgabewerte werden wieder in Objekte der `…Wrap`-Klasse eingehüllt.

[14] http://docs.mockito.googlecode.com/hg/latest/index.html?org/mockito/Mockito.html

[15] http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/InvocationHandler.html

[16] Type erasure bei Java Generics macht den Einsatz dieses zweitgenannten Instanzattributs notwendig.

[17] http://docs.oracle.com/javase/7/docs/api/java/security/PrivilegedAction.html

5. Fazit

Wählt man wie in Graja JUnit-Tests als Werkzeug zur automatisierten Programmbewertung, ist der Einsatz einer Bibliothek zur Erzeugung von Test Doubles naheliegend. Studentischer Code und Test Doubles gehören verschiedenen Schutzbereichen an. Test Doubles sind im Aufrufbaum Kinder des studentischen Codes, so dass tendenziell restriktive Rechte des studentischen Codes an Test Doubles vererbt werden. Mock-Bibliotheken benötigen für ihre Test Doubles jedoch Rechte von erheblichem Bedrohungspotential. Um den von der Lehrkraft geschriebenen Bewertungscode nicht mit sicherheitsspezifischen Details (Proxy-Objekte mit in `doPrivileged`-Aufrufen gekapselten Delegationen) zu „verschmutzen", bietet sich die Verlagerung dieser Details in die Mock-Bibliothek oder in einen hierfür zu schreibenden Adapter an. Die Erstellung eines solchen Adapters kann durch Nutzung von Java Generics und der Java-Reflection-Bibliothek kompakt gelingen.

6. Literaturverzeichnis

[Ed03] Edwards, S.: Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In Proc. Int'l Conf. Education and Information Systems: Technologies and Applications (EISTA '03), Aug. 2003.

[Go13] Gong, L.: Java™ SE Platform Security Architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc.html, Abruf 13.09.2013.

[Ka13] Kaczanowski, T.: Practical Unit Testing with JUnit and Mockito. Verlag Tomasz Kaczanowski, 2013.

[KSZ02] Krinke J, Störzer M, Zeller A, *Web-basierte Programmierpraktika mit Praktomat*, Softwaretechnik-Trends, Vol. 22, (3), October 2002.

[Me07] Meszaros, G.: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Longman, Amsterdam, 2007.

[oV13] o. V.: OWASP Risk Rating Methodology. https://www.owasp.org/index.php?title=OWASP_Risk_Rating_Methodology&oldid=158022, Abruf 13.09.2013.

[SBG10] Striewe, M.; Balz, M.; Goedicke, M.: Enabling Graph Transformations on Program Code, In *Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, Enschede, Netherlands, 2010.

[SL05] Spillner, A.; Linz, T.: Basiswissen Softwaretest (3. Auflage). dpunkt, Heidelberg, Germany, 2005

[St13] Stöcker, A., Becker, S., Garmann, R., Heine, F., Kleiner, C., Bott, O. J.: Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern „aSQLg" und „Graja" aus studentischer Perspektive. In Proceedings DeLFI, 2013.

[TE08] Thornton, M., Edwards, S. H., Tan, R. P., Pérez-Quiñones, M. A.: Supporting student-written tests of GUI programs. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education. ACM Press, New York, NY, 2008, pp. 537-541.