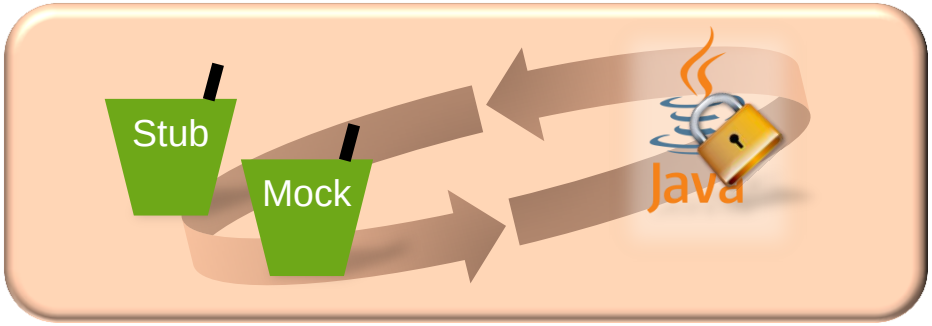
HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

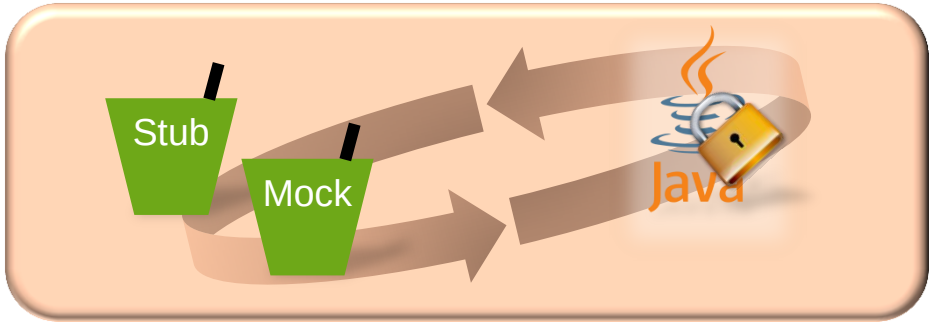
Workshop „Automatische Bewertung von Programmieraufgaben“, Hannover

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



Robert Garmann, 28.10.2013

Inhaltsverzeichnis



Programmieraufgabe (Beispiel)

mit Bewertungsaspekten

Eine richtige Lösung …

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.

… kann Quadrate berechnen

… kann Hypotenusen berechnen

… setzt Wiederverwendung durch Delegation ein

Musterlösung

mit Bewertungsplanung

1. pruefeQuadratMethode: Es existiert eine Klasse `QuadratImpl`, die `Quadrat` funktional korrekt implementiert. Gewicht: 35%.

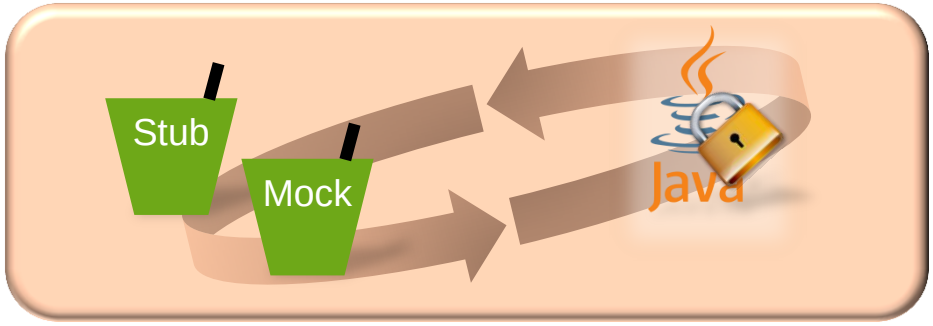
```
import de.hsh.prog.Quadrat;
public class QuadratImpl implements Quadrat {
    public double quadrat(double v) { return v*v; }
}
```

2. pruefeGesamtfunktion: Es existiert eine Klasse `Hypo`, die die Gesamtfunktion korrekt umsetzt. Gewicht: 25%.

```
import de.hsh.prog.Quadrat;
public class Hypo {
    public static double hypo(Quadrat q, double a, double b) {
        return Math.sqrt(q.quadrat(a)+q.quadrat(b));
    }
}
```

3. pruefeAufrufVonQuadrat: `hypo` ruft `q.quadrat` zwei Mal mit den Übergabewerten `a` und `b` auf. Gewicht: 40%.

Inhaltsverzeichnis



Graja

Grader for Java programs

- JUnit-Überbau – Bewertung mit JUnit-Testmethoden
- Inspiriert von Web-CAT (2003, Stephen H. Edwards)
- Entstanden 2012 (R. Garmann)
- Funktionen:
  - Studentisches ZIP-Archiv extrahieren, Code übersetzen
  - Standard-Ein/Ausgabe maschinell beschicken / auswerten
  - Erwartete und beobachtete Ausgabe intelligent vergleichen
  - Studentischen Code „reflection“-basiert analysieren
  - Bewertungskommentare: formatiert, zielgruppenorientiert, nach Kritikalität abgestuft
  - Punkte für mehrere Bewertungsaspekte vergeben
  - Systemresourcen begrenzen (Laufzeit, persistenter Speicherplatz, Hauptspeicher)
  - Studentischen Code und Bewertungscode in separat gesicherten Schutzbereichen ausführen

Bewertung mit Graja

Erster Bewertungsaspekt

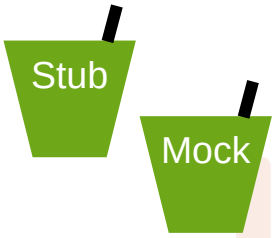
1. pruefeQuadratMethode: Es existiert eine Klasse `QuadratImpl`, die `Quadrat` funktional korrekt implementiert. Gewicht: 35%.

```
@Test
@ScoringWeight(35.0) // Teilpunkte vergeben
public void pruefeQuadratMethode() {
  // Studentische Klasse via Reflection instanziieren:
  Quadrat studentImpl=
          ReflectionSupport.create(getSubclassForName("QuadratImpl", Quadrat.class));

  // studentischen Code ausführen:
  double a=3;
  double observed= studentImpl.quadrat(a);
  double expected= a*a;

  // Ausführlicher Fehlerhinweis:
  String msg= "Überprüfen Sie Ihre Implementierung von QuadratImpl.quadrat. "
              +"quadrat("+a+") liefert "+observed+" (erwartet :"+expected+")";
  org.junit.Assert.assertEquals(msg, expected, observed, 1E-5);
}
```

mockito

simpler & better mocking

- Zweck: ein *SUT (system under test)* während des Tests von seiner Umgebung isolieren
- Inspiriert von EasyMock (2002, Tammo Freese)
- Entstanden 2007 (Szczepan Faber)
- Funktionen:
  - Erzeugung von Platzhaltern (*test doubles*) in der Umgebung des SUT
  - „Write beautiful, readable tests“
  - „Clean & simple API“
  - Unterstützt *stubbing* (Platzhalter mit vorgefertigten Antworten bestücken)
  - Unterstützt *mocking* (Interaktionen zwischen SUT und Platzhaltern prüfen)
- `http://code.google.com/p/mockito/`



Bewertung mit Graja / mockito

Zweiter Bewertungsaspekt

2. pruefeGesamtfunktion: Es existiert eine Klasse `Hypo`, die die Gesamtfunktion korrekt umsetzt. Gewicht: 25%.

Musterlösung zur Erinnerung:

```
public class Hypo {
  public static double hypo(Quadrat q, double a, double b) {
    return Math.sqrt(q.quadrat(a)+q.quadrat(b));
  }
}
```

- Plan: `Hypo.hypo` aufrufen und Ergebnis prüfen
- Was übergeben wir für Parameter `q`?
  - Nicht: Objekt der studentischen Klasse (Fehler nicht doppelt zählen)
  - Möglich: Objekt der eigenen Musterlösung (falls vorhanden)
  - Möglich: Test Double mit vorgefertigten Antworten (Stub)

Bewertung mit Graja / mockito

Zweiter Bewertungsaspekt mit Test Double (Stub)

2. pruefeGesamtfunktion:
Es existiert eine Klasse Hypo, die die Gesamtfunktion korrekt umsetzt. Gewicht: 25%.

```
@Test
@ScoringWeight(25.0)
public void pruefeGesamtfunktion() {
  double a=3, b=4; // zwei Katheten

  // Test Double mit vorgefertigten Antworten (Stub):
  Quadrat stub= Mockito.mock(Quadrat.class);
  Mockito.when(stub.quadrat(a)).thenReturn(a*a);
  Mockito.when(stub.quadrat(b)).thenReturn(b*b);

  // studentischen Code ausführen:
  double c= ReflectionSupport.invokeStatic(
                          getClassForName("Hypo"), double.class, "hypo", stub, a, b);

  // prüfe Ergebnis (black box):
  double cExpected= 5;
  String msg= "hypo(q,"+a+","+b+")="+c+" (erwartet: "+cExpected+")";
  org.junit.Assert.assertEquals(msg, cExpected, c, 1E-5);
}
```

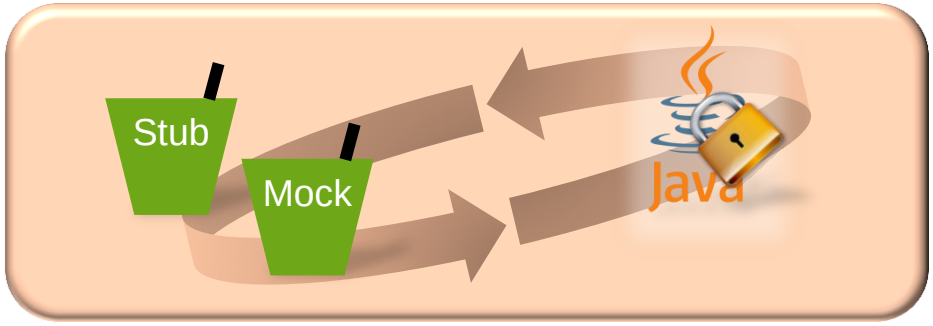
Bewertung mit Graja / mockito

Dritter Bewertungsaspekt mit Test Double (Mock)

3. pruefeAufrufVonQuadrat: hypo ruft `q.quadrat` zwei Mal mit den Übergabewerten `a` und `b` auf. Gewicht: 40%.

```
@Test
@ScoringWeight(40.0)
public void pruefeAufrufVonQuadrat() {
  double a=3, b=4; // zwei Katheten

  // Test Double zum Aufsammeln von Aufrufen aus studentischem Code (Mock):
  Quadrat mock= Mockito.mock(Quadrat.class);

  // studentischen Code ausführen, Ergebnis ignorieren:
  ReflectionSupport.invokeStatic(getClassForName("Hypo"), double.class, "hypo", mock, a, b);

  // prüfe Verhalten (white box) des studentischen Codes:
  try {
    Mockito.verify(mock, Mockito.times(1)).quadrat(a);
    Mockito.verify(mock, Mockito.times(1)).quadrat(b);
  } catch (WantedButNotInvoked e) {
    org.junit.Assert.fail("hypo muss die Quadrierung an den Quadrat-Parameter delegieren");
  }
}
```

Inhaltsverzeichnis

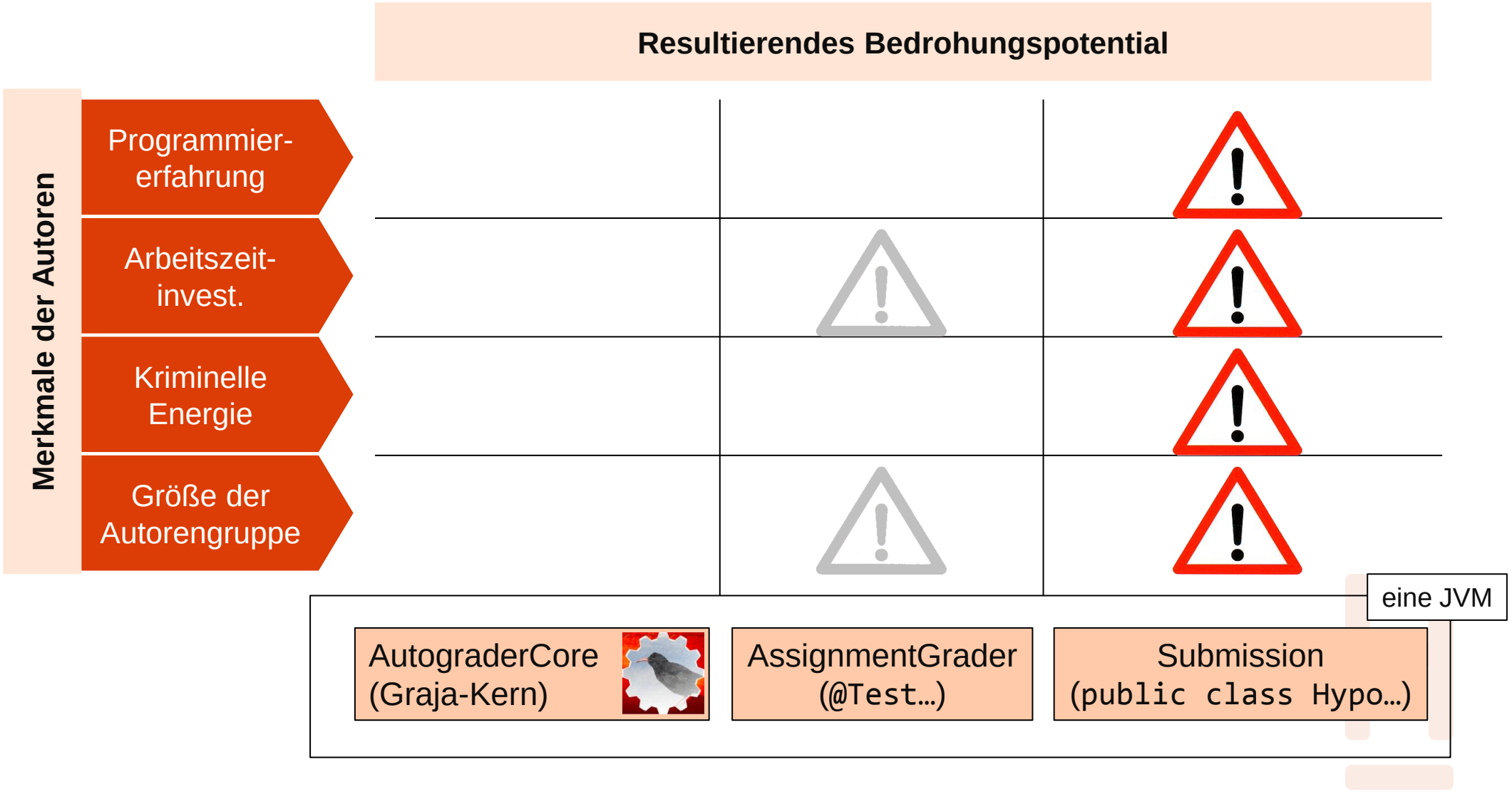


Sicherheit des Bewertungssystems

im Vergleich zu herkömmlichen Softwaretests

Herkömmliche Softwaretests	Bewertung von Programmieraufgaben
Tester und Autor vertrauen einander (häufig sogar dieselbe Person)	Bewertungssystem muss mit Angriffen des getesteten Codes rechnen
Programmierfehler mit sicherheitsrelevanten Folgen sind selten	Unabsichtliche Bedrohungen durch Programmierfehler sind häufig
→ Schutzmaßnahmen häufig nicht erforderlich	→ Schutzmaßnahmen unabdingbar

Schutzbereiche und ihr Bedrohungspotential



Beispielbedrohung

„rm –rf /“

- Bösartiger Submission-Code:

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

- Abwenden:
  - durch Java SecurityManager und selektive Gewährung des Rechts `java.io.FilePermission`

Beispielbedrohung

Denial of service

- Bösartiger (oder versehentlich bedrohlicher) Submission-Code:

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

- Nicht abwendbar durch Java SecurityManager

Beispielbedrohung

Ausspähen der Musterlösung

- Bösartiger Submission-Code:

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

- Abwenden:
  - durch Java SecurityManager und selektive Gewährung des Rechts `java.io.FilePermission`

Beispielbedrohung

Ausführen der Musterlösung

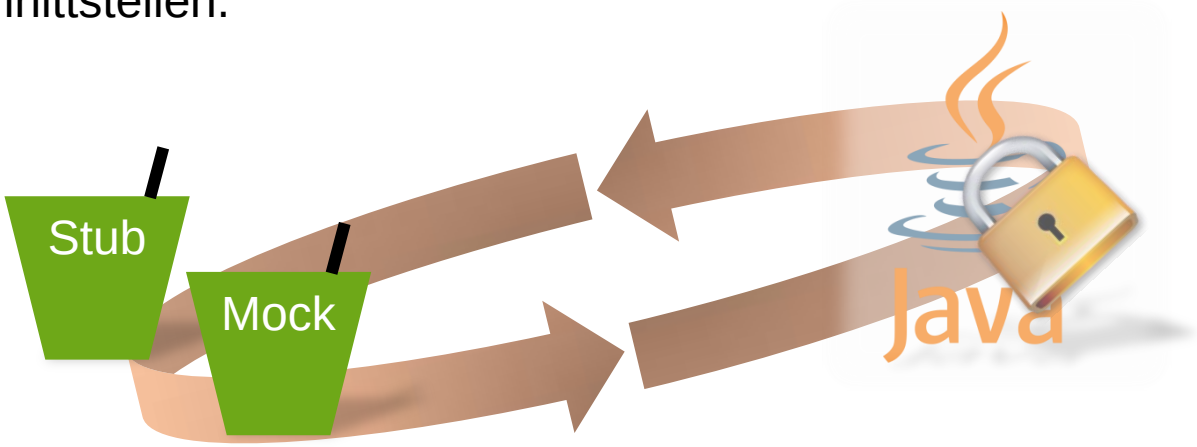
- Bösartiger Submission-Code:
  - Ruft Methoden der Musterlösungsklasse auf.
  - “Verkauft” die Ausgabe als die eigene.
- Abwenden:
  - Musterlösungsklasse package private deklarieren

  zusammen mit
  - Java SecurityManager: Selektive Gewährung des Rechts `java.lang.reflect.ReflectPermission.suppressAccessChecks`

Zwischenfazit

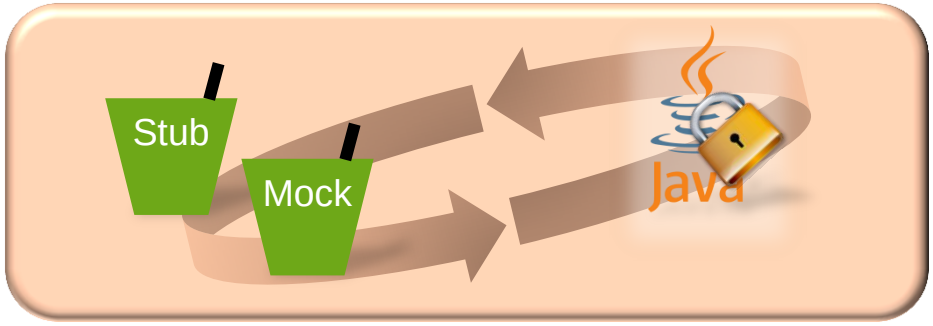
Test Doubles eignen sich zur Bewertung einer studentischen Lösung an internen Programmschnittstellen.

Java SecurityManager kann viele Bedrohungen abwenden.



Betrachte nun: Auswirkungen des Java SecurityManagers auf den Einsatz von Test Doubles.

Inhaltsverzeichnis



Schutzbereiche mit benötigten Rechten

- Zugriffe auf das lokale Dateisystem
- Ausführen des Compilers
- Verwendung eigener Klassenlader
- Zugriff auf Reflection-Funktionen
- etc.

AutograderCore (Graja-Kern)

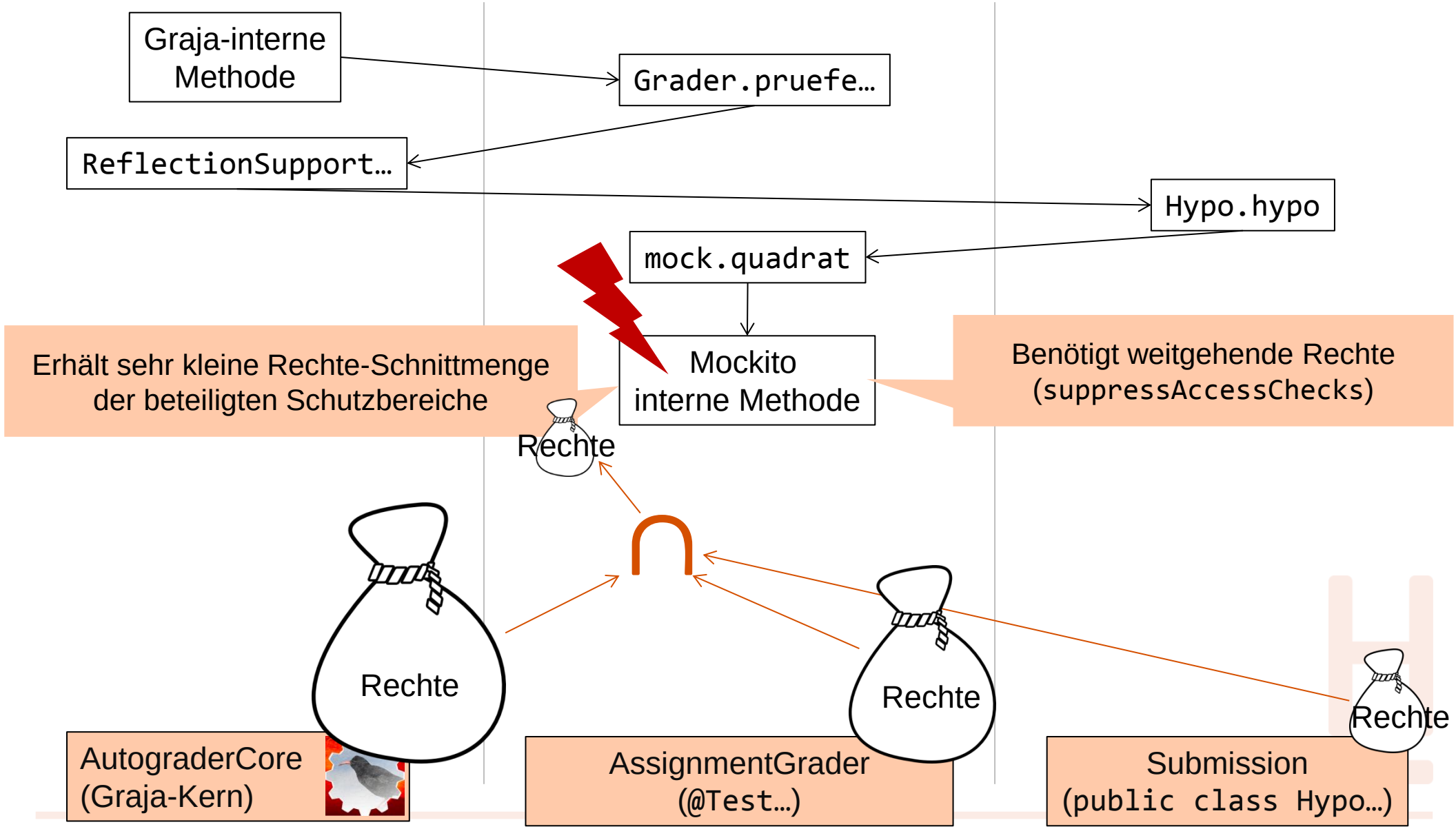
- mittlere Rechte (z. B. Dateizugriffe)
- weitergehende Rechte nach Bedarf, etwa von mockito benötigte Rechte (`suppressAccessChecks`)

AssignmentGrader (`@Test…`)

- i. d. R. stark beschränkte Rechte (abhängig von der Aufgabe)

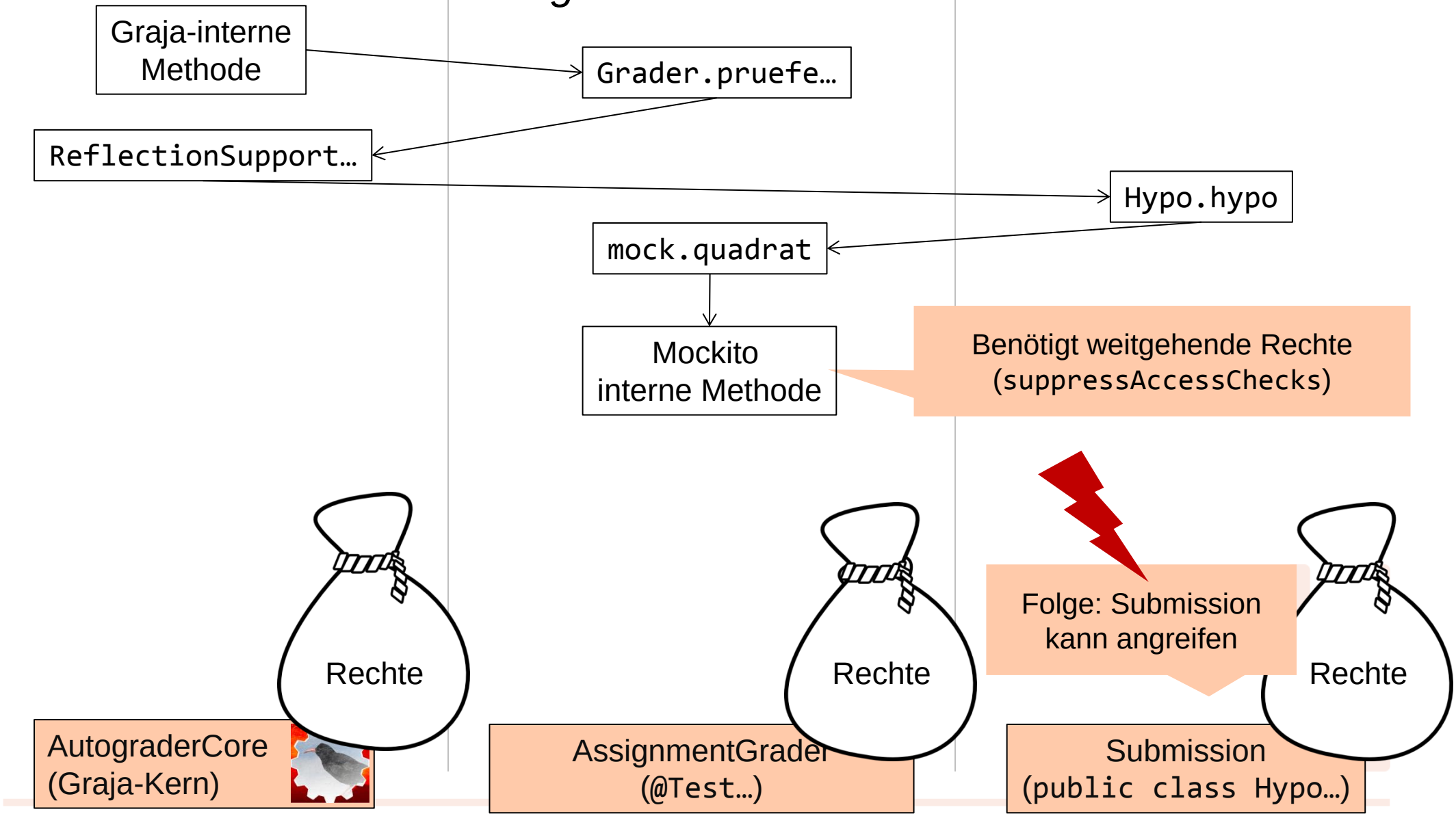
Submission (`public class Hypo…`)

Beispiel-Aufrufsequenz

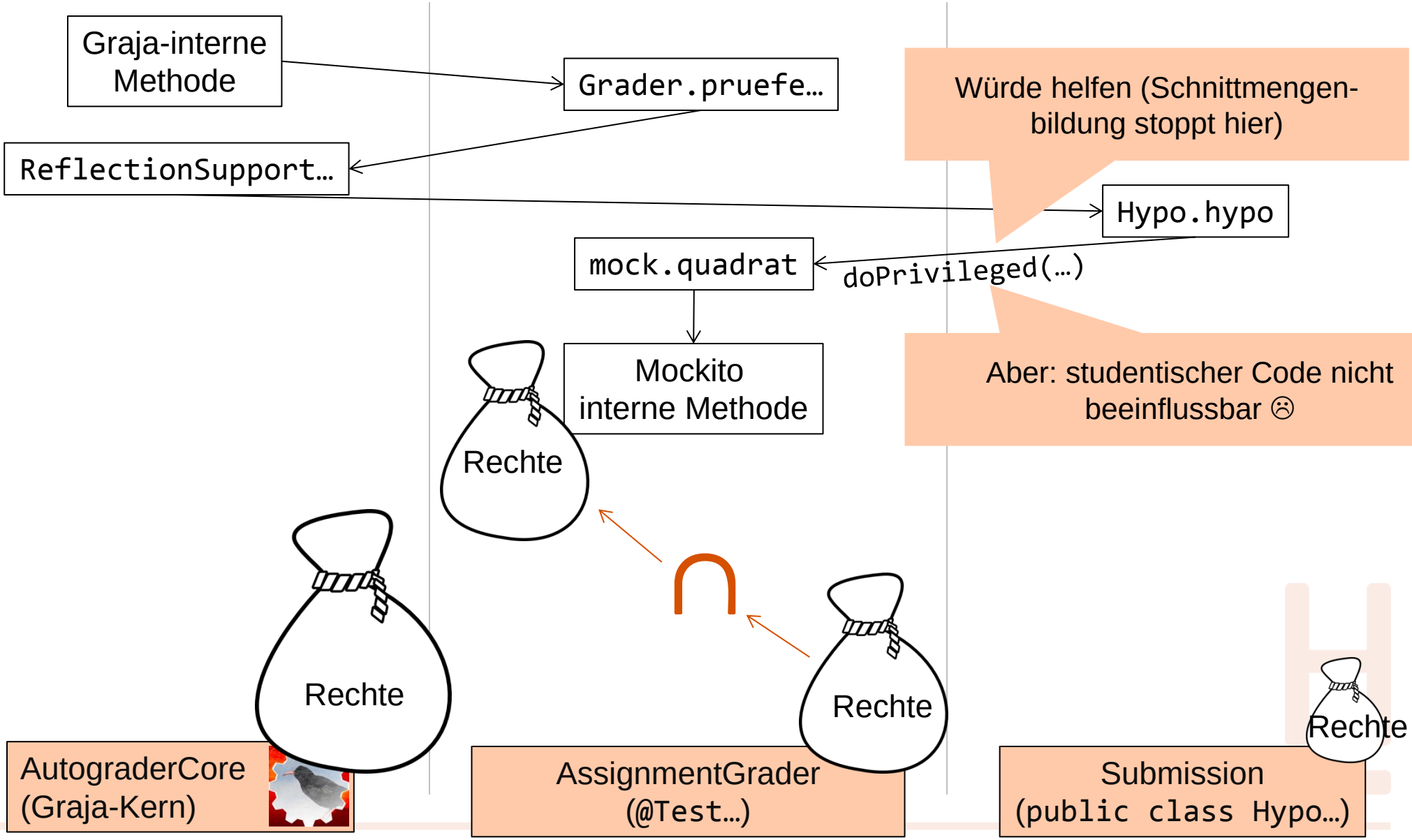


Keine Lösung

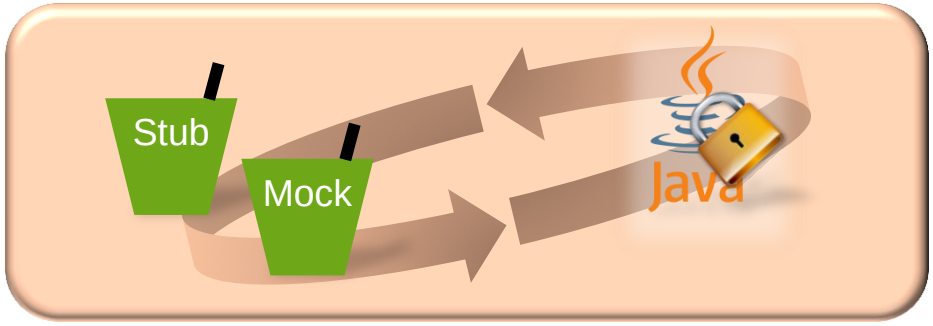
Alle Schutzbereiche mit weitgehenden Rechten ausstatten



Lösungsansatz: Privilegierter Aufruf



Inhaltsverzeichnis



Lösungsidee

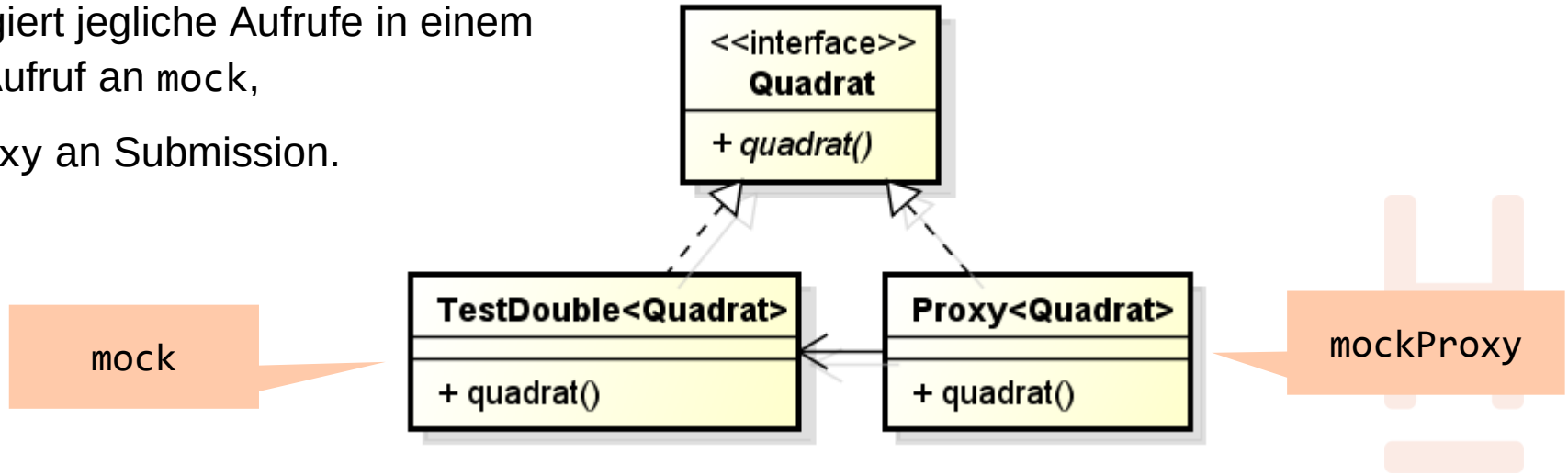
Proxy führt Test Double privilegiert aus

AssignmentGrader-Methode `pruefe…` in der bisherigen Version:

```
Quadrat mock= Mockito.mock(Quadrat.class);
ReflectionSupport.invokeStatic(getClassForName("Hypo"), double.class, "hypo", mock, a, b);
```

Neue Version:

- erzeuge ein Proxy-Objekt `mockProxy`,
- `mockProxy` delegiert jegliche Aufrufe in einem `doPrivileged`-Aufruf an `mock`,
- übergib `mockProxy` an Submission.



MockitoWrap

kapselt privilegierte Ausführung von Test Doubles

- Ziel: Proxy-Erzeugung verlagern in Bibliothek
  - Keine “Verschmutzung” des Bewertungscodes mit Sicherheitsdetails
  - Ersteller des Bewertungscodes benötigt nicht das entsprechende KnowHow.
  - Die bestehende Codebasis des AssignmentGraders erfährt minimale Änderungen. Die einfache Nutzbarkeit verbessert die Nutzungswahrscheinlichkeit und damit die Sicherheit des Bewertungssystems.

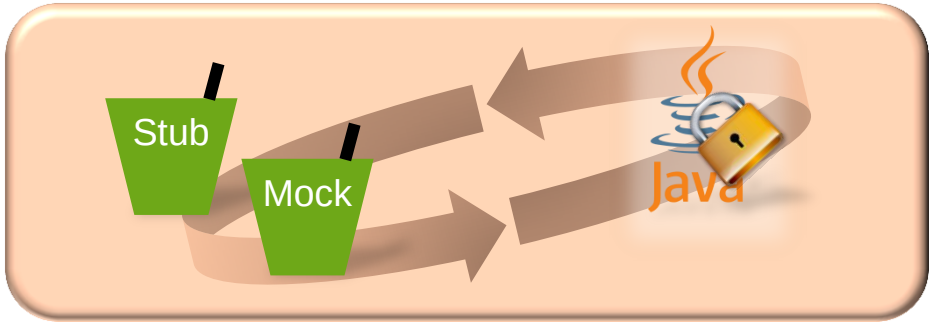
- Nutzung seitens des AssignmentGraders:
  - Im Bewertungscode: ~~`Mockito.mock`~~ → `MockitoWrap.mock`
  - `MockitoWrap` erzeugt das genannte Proxy-Objekt.
  - Ähnliche Wrapper stehen für weitere mockito-Klassen zur Verfügung (`OngoingStubbing<T>`, …)

MockitoWrap

Umsetzung und Status

- Umsetzung
  - Kompakte technische Umsetzung möglich
  - Entwickler benötigen Detailkenntnis und Erfahrung mit fortgeschrittenen Java Features (Proxy / Reflection in Kombination mit Generics).
- Status:
  - MockitoWrap wird im Zuge der Erstellung neuer Programmieraufgaben sukzessive an der Hochschule Hannover weiter entwickelt.

Inhaltsverzeichnis



Zusammenfassung

- Gezeigt, wie man interne Programmschnittstellen der Submission mit Test Doubles dynamisch analysiert.
- Motiviert, dass die Submission durch ihr hohes Bedrohungspotential restriktive Rechte besitzen sollte – überwacht durch den Java SecurityManager.
- Test Doubles benötigen weitgehende Rechte, erben jedoch im Aufrufbaum die Submission-Restriktionen.
- Privilegierte Ausführung ist kein Feature von mockito (unüblich bei herkömmlichen Softwaretests)
- Privilegierte Ausführung kann in eine einfach zu nutzende Bibliothek ausgelagert werden (MockitoWrap).

Vielen Dank für Ihre Aufmerksamkeit!

Fragen / Anmerkungen / Feedback?