Bewertungsaspekte und Tests in Java-Programmier-aufgaben für Graja im ProFormA-Aufgabenformat

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

Robert Garmann[1]

Forschungsbericht

30. Mai 2016

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Ricklinger Stadtweg 120
30459 Hannover

[1] E-Mail: robert.garmann@hs-hannover.de

Zusammenfassung

Ein Autobewerter für von Studierenden eingereichte Programme führt die im ProFormA-Aufgabenformat sequentiell spezifizierten „Tests“ aus, um die Einreichung zu prüfen. Bzgl. der Interpretation und Darstellung der Testausführungsergebnisse gibt es derzeit keinen graderübergreifenden Standard. Wir beschreiben eine Erweiterung des ProFormA-Aufgabenformats um eine Hierarchie von Bewertungsaspekten, die nach didaktischen Aspekten gruppiert ist und Referenzen auf die Testausführungen besitzt. Die Erweiterung wurde in Graja umgesetzt, einem Autobewerter für Java-Programme. Je nach gewünschter Detailaufschlüsselung der Bewertungsaspekte müssen in der Konsequenz Testausführungen in Teilausführungen aufgebrochen werden. Wir illustrieren unseren Vorschlag unter Einsatz der Testwerkzeuge Compiler, dynamischer Softwaretest, statische Analyse sowie unter Einsatz menschlicher Bewerter.

Schlagworte

e-Assessment, computer based assessment, CBA, Grader, Bewertung, Java, Programmierung, Programmieraufgabe, Programmieranfänger, Feedback, ProFormA-Aufgabenformat, formatives Assessment, Testautomation, Autobewerter, Graja

DDC Klassifikation

004 Datenverarbeitung; Informatik

GND-Schlagworte

Programmierung, Softwaretest, E-Learning, Computerunterstütztes Lernen, Java <Programmiersprache>, Konfiguration <Informatik>, Softwarewartung, Übung <Hochschule>, Lernaufgabe, Softwarewerkzeug, JUnit

ACM CCS (2012)

• **Social and professional topics~Computer science education** • **Social and professional topics~Student assessment** • *Applied computing~Computer-assisted instruction* • *Applied computing~E-learning* • *Software and its engineering~Software testing and debugging*

1 Einleitung

1.1 Forschungsfrage und -methodik

Die Bewertung einer studentischen Einreichung zu einer Java-Programmieraufgabe kann teilweise automatisiert durch sog. Autobewerter (Grader) erfolgen. Graja[2] [1] ist ein solcher Autobewerter, der im Hintergrund Softwaretechnikwerkzeuge wie Compiler, JUnit[3] und PMD[4] nutzt. Graja verwendet das in den letzten Jahren als graderübergreifender Standard entwickelte ProFormA-Aufgabenformat [2], in dem sog. „Tests“ die Konfiguration und Durchführung von Bewertungsschritten spezifizieren. Ein „Test“ im ProFormA-Aufgabenformat wird von verschiedenen Gradern und auch von Graja als Spezifikation zur Ausführung eines Werkzeuges interpretiert. Die Ausgabe des Werkzeuges wird dann als Teil

[2] http://graja.hs-hannover.de
[3] http://junit.org
[4] http://pmd.github.io

des gesamten Bewertungsergebnisses an den einreichenden Studenten oder die einreichende Studentin zurück gemeldet.

Nun ist es aus didaktischer Sicht nicht optimal, die Ausgaben der genutzten Werkzeuge einfach unverändert als Feedback durchzureichen. In der Vergangenheit durchgeführte studentische Befragungen im Rahmen von Evaluierungen [3] offenbarten den Wunsch nach einer „übersichtlichen“ Darstellung der Bewertungsergebnisse. Als Forschungsfrage ergibt sich, unter welchen Umständen und Maßnahmen eine bessere, lernförderliche Darstellung der Bewertungsergebnisse erreicht werden kann, ohne dass der Vorteil der Nutzung bestehender und ausgereifter Softwaretechnikwerkzeuge aufgegeben werden muss.

In diesem Bericht beschreiben wir, wie Graja und das genutzte Aufgabenformat erweitert werden kann, um eine übersichtlichere Darstellung des Bewertungsergebnisses zu erreichen. Wir trennen dazu die Sicht auf „Tests“ auf in eine technische Sicht, die die Werkzeuge konfiguriert und deren Ausführung als Ganzes oder in Teilen spezifiziert, sowie eine didaktische Sicht, die mehrere Bewertungsaspekte hierarchisch angeordnet spezifiziert und dabei auf die technischen Testergebnisse Bezug nimmt. Forschungsmethodisch zeigen wir an einem konkreten Fallbeispiel auf, welche Änderungen an bestehenden Informationssystemen zur Erreichung des Forschungsziels führen. Wir führen diese Änderungen am konkreten System durch und demonstrieren die erreichten Ergebnisse.

1.2 Überblick über diesen Bericht

Wir beginnen mit einer kurzen Einführung in den relevanten Teil des ProFormA-Aufgabenformats in Abschnitt 2. Im weiteren Verlauf dieses Abschnitts zeigen wir eine Beispielaufgabe, die sich auf das ProFormA-Aufgabenformat stützt und die von Graja bewertet werden kann. Das Beispiel ist bewusst nicht minimalistisch gehalten, um die in der Realität erst in komplexeren Aufgaben auftretenden Effekte im weiteren Verlauf dieses Beitrags illustrieren zu können.

Während Abschnitt 2 auf die Spezifikation einer Aufgabe fokussiert, beschäftigt sich Abschnitt 3 mit der Darstellung des Bewertungsergebnisses. Wir zeigen zunächst die an den Testausführungsergebnissen orientierte Ausgabe (sowohl modellhaft als auch am konkreten Graja-Beispiel) um dann die Mängel dieser Form des Feedbacks zu benennen und Alternativen zu entwickeln.

Ein Ergebnis des Abschnitts 3 ist die Notwendigkeit, Test-Teilausführungen spezifizieren zu können. Abschnitt 4 erörtert zunächst am konkreten Graja-Beispiel eine diesbezügliche Möglichkeit der Spezifizierung. Dabei diskutieren wir auch die dabei relevante Frage, wie oft ein Testwerkzeug im Rahmen eines Bewertungsprozesses ausgeführt werden muss oder darf. Ein Ergebnis des Abschnitts 4 ist ein Vorschlag zur Erweiterung des ProFormA-Aufgabenformats um die Möglichkeit, Tests und Test-Teilausführungen im Zusammenhang zu spezifizieren.

Abschnitt 5 widmet sich schließlich der Umsetzung der in Abschnitt 3 entwickelten alternativen Form des Feedbacks. Dazu schlagen wir eine Erweiterung des ProFormA-Aufgabenformats um zusätzliche Domänenobjekte vor, die die Struktur des Bewertungsfeedbacks spezifizieren. Wir zeigen am konkreten Graja-Beispiel die Auswirkungen auf das generierte Feedback um argumentativ festzustellen, dass die Übersichtlichkeit und Lernförderlichkeit des Feedbacks verbessert wurde.

Abschnitt 6 fasst die Ergebnisse zusammen und nennt offene Forschungsfragen.

2 Tests im ProFormA-Aufgabenformat

Das ProFormA-Aufgabenformat [2] wurde in den letzten Jahren als ein Format entwickelt, das den Austausch von automatisiert bewertbaren Programmieraufgaben zwischen verschiedenen Lernmanagementsystemen (LMS) und Autobewertern befördert. Das Format

ist als XML-Schema spezifiziert und enthält diverse verpflichtende aber auch mehrere optionale Elemente. Weiterhin ist es möglich, in einer Aufgabe grader-spezifische Daten aufzunehmen, die nicht im ProFormA-XML-Schema spezifiziert sind.

2.1 Das Domänenobjekt *Test*

Ein zentrales Element in diesem Format ist ein sog. *Test*. Ein Test ist ein Domänenobjekt, welches die automatisierte Prüfung einer studentischen Einreichung spezifiziert. Zu einem Test gehören u. a. die folgenden Datenelemente:

- Eine *Id* zur eindeutigen Identifizierung.
- Ein *Testtyp*. Gegenwärtig (Version 1.0.1) enthält die Liste der Testtypen Tests zur Compilierung, zur (Quell-)Codeanalyse, zur Bytecodeanalyse, für (dynamische) Software-Tests, und für weitere Werkzeuge wie Codeabdeckungsanalyse und Anonymitätsprüfung.
- Einen *Titel*. Der Titel wird von einem Autobewerter genutzt, um die vom Test generierte Ausgabe gegenüber dem einreichenden Studenten mit einem aussagekräftigen Titel zu versehen.
- Eine *Testkonfiguration*. Die Testkonfiguration enthält alle Parameter, die zur Durchführung der automatisierten Prüfung benötigt werden. Konkrete Beispiele sind eine JUnit-Testklasse oder eine PMD-Rule. Allgemeiner können unter einer Testkonfiguration bspw. die folgenden Informationen spezifiziert werden: Pfadangaben, benötigte Bibliotheken, Bezeichner von benutzten externen Ressourcen, sonstige Dateien oder werkzeugspezifische Konfigurationsparameter.

Die drei erstgenannten Datenelemente sind im ProFormA-Aufgabenformat standardisiert. Das letztgenannte Datenelement ist ein Sammelbecken für graderspezifische XML-Elemente, die über das XML Schema-Element <any> aus fremden XML-Namensräumen eingebunden werden können.

Nicht zur Testkonfiguration gehört in der Regel die Definition von Bewertungsmaßstäben. Wie viele Punkte für das Bestehen oder Teilbestehen der automatisierten Prüfung vergeben werden, wird im ProFormA-Aufgabenformat nicht an standardisierter Stelle spezifiziert. Entweder können Bewertungsmaßstäbe getrennt von der eigentlichen Aufgabe an den Autobewerter geleitet werden oder als Teil des im ProFormA-Aufgabenformat definierten, inhaltlich weitgehend offen gelassenen Domänenobjekts *GradingHints*.

2.2 Eine Beispielaufgabe

Wir zeigen ein Beispiel mit vier Tests für eine Java-Programmieraufgabe, die von Graja[5] automatisiert bewertet werden kann (vgl. Abbildung 1). Der dort gezeigte XML-Ausschnitt ist keine vollständige Aufgabe. Es fehlen weitere Informationen zur Aufgabe (Beschreibung, Versionsnummer, etc.) sowie die Angabe der Dateipfade (ProFormA-Element <file>), die wir hier aus Platzgründen weglassen.

Da die XML-Darstellung teilweise etwas mühsam zu lesen ist, stellen wir die Information in UML-Form dar. Ein dem XML-Schema entsprechendes UML-Modell zeigt die Abbildung 2. Der Erweiterungsmechanismus durch das XML-Schema-Element <any> ist als Subklassenbildung dargestellt. Die allgemeine ProFormA-Spezifikation erlaubt die Angabe von Dateien (Fileref) in einer Testkonfiguration, nicht jedoch die Auszeichnung jeder einzelnen Datei mit einer Zusatzinformation, die die Rolle der Datei beschreibt. Deshalb werden zusätzlich zwei Graja-spezifische Objekte PolicyRef und ClasspathRefs aufgenommen, die einige der Dateien mit Rollen versehen. Im Beispiel-XML-Dokument spielt die Datei mit der Id „policy“ die Rolle der Security-Policy beim Start der Graja-JVM und die drei Dateien mit den Ids „grader“, „ruleset“ und „libmock“ spielen die Rolle des Klassenpfads beim Start der Graja-JVM.

[5] Die hier gezeigte Aufgabe basiert auf einer Entwicklungsversion von Graja

Das XML-Dokument aus Abbildung 1 ist in Abbildung 3 als UML-Objektmodell dargestellt. Das gezeigte Beispiel illustriert, dass je ein Test-Objekt für die Durchführung eines Softwaretechnikwerkzeuges spezifiziert wurde. Das Test-Objekt mit der id B spezifiziert eine Durchführung von JUnit, das Test-Objekt mit der id C spezifiziert eine Durchführung von PMD. Eine Sonderrolle spielt das Test-Objekt mit der id D, welches eine nachträgliche Bewertung durch einen menschlichen Gutachter spezifiziert. In diesem Beitrag betrachten wir menschliche Teilbewertungen als Ergebnisse des „Testwerkzeugs“ Mensch. Vorteil dieser Generalisierung ist eine einheitliche Darstellung im Bewertungsergebnis.

Jedes der Test-Objekte besitzt eine spezifische Testkonfiguration. Der zusätzliche, in Abbildung 3 im oberen Bildbereich dargestellte Test mit der id basecfg ist eigentlich kein zusätzlicher Test, sondern dieses Test-Objekt dient der Aufnahme von übergreifenden Informationen, die nicht eindeutig dem JUnit-Test, dem PMD-Test oder dem manuellen Test zugeordnet werden können oder sollen[6].

Eigentlich fehlt in Abbildung 1 und Abbildung 3 ein Test-Objekt A zur Spezifikation des Compilers. Da wir in Graja den Compiler sowieso immer durchführen, erübrigt sich derzeit eine diesbezügliche Testkonfiguration. Es ist jedoch geplant, den Compiler als explizit zu konfigurierendes Testwerkzeug in die Sequenz der Test-Objekte aufzunehmen.

[6] Ein alternativer Entwurf wäre gewesen, die Dateien mit den ids „grader“ und „libmock“ nicht im Test-Objekt basecfg, sondern im Test-Objekt B unterzubringen, sowie die Datei mit der id „ruleset“ im Test-Objekt C. Da diese Dateien jedoch auch bereits beim Start der Graja-JVM benötigt werden, wurde der oben gezeigte Entwurf umgesetzt.

```
<p:test id="basecfg">
  <p:title>Graja grading result</p:title>
  <p:test-type>graja</p:test-type>
  <p:test-configuration>
    <p:filerefs>
      <p:fileref refid="grader"/><p:fileref refid="policy"/><p:fileref refid="ruleset"/><p:fileref
refid="libmock"/>
    </p:filerefs>
    <p:externalresourcerefs/>
    <graja:computationResources>
      <graja:maxDiscQuotaKib>1000</graja:maxDiscQuotaKib>
      <graja:maxRuntimeSecondsWallclockTime>15</graja:maxRuntimeSecondsWallclockTime>
      <graja:maxMemMib>96</graja:maxMemMib>
    </graja:computationResources>
    <graja:policyRef><graja:ref externalOrFile="file" refid="policy"/></graja:policyRef>
    <graja:classpathRefs>
      <graja:refs>
        <graja:refs externalOrFile="file" refid="grader"/><graja:refs externalOrFile="file"
refid="libmock"/><graja:refs externalOrFile="file" refid="ruleset"/>
      </graja:refs>
    </graja:classpathRefs>
    <p:test-meta-data>
      <graja:grajaMetaData>
        <graja:grajaVersionCompatibility>1.4.0</graja:grajaVersionCompatibility>
      </graja:grajaMetaData>
    </p:test-meta-data>
  </p:test-configuration>
</p:test>
<p:test id="B">
  <p:title>JUnit result </p:title>
  <p:test-type>graja-junit</p:test-type>
  <p:test-configuration>
    <graja:grajaConfiguration>
      <graja:graderClassFqn>de.hsh.prog.shoppingcartv02.grader.Grader</graja:graderClassFqn>
    </graja:grajaConfiguration>
  </p:test-configuration>
</p:test>
<p:test id="C">
  <p:title>PMD result</p:title>
  <p:test-type>graja-pmd</p:test-type>
  <p:test-configuration>
    <graja:grajaConfiguration>
      <graja:pmdRuleset>ruleset1.xml</graja:pmdRuleset>
    </graja:grajaConfiguration>
  </p:test-configuration>
</p:test>
<p:test id="D">
  <p:title>Human grading activity</p:title>
  <p:test-type>graja-human</p:test-type>
  <p:test-configuration>
    <graja:grajaConfiguration>
      <graja:humanTestDescription>Later human reviewers will grade your submission and credit
points for the following aspects: a) String concatenation using StringBuilder, b) no code
redundancy.</graja:humanTestDescription>
    </graja:grajaConfiguration>
  </p:test-configuration>
</p:test>
```

Abbildung 1: Eine Beispielaufgabe für Graja im ProFormA-Aufgabenformat

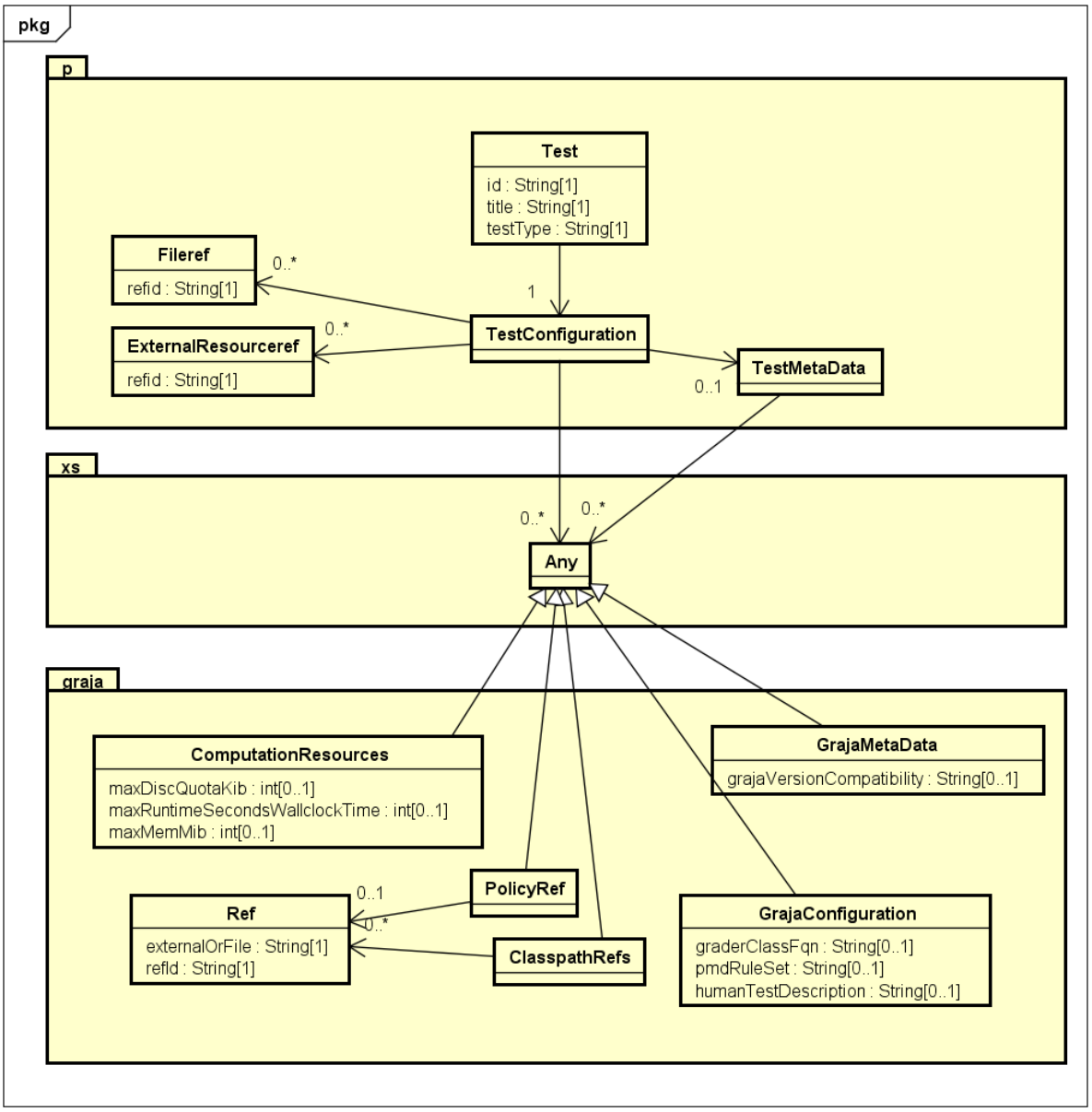

Abbildung 2: UML-Modell des ProFormA-Tests sowie der Graja-spezifischen Elemente darin.

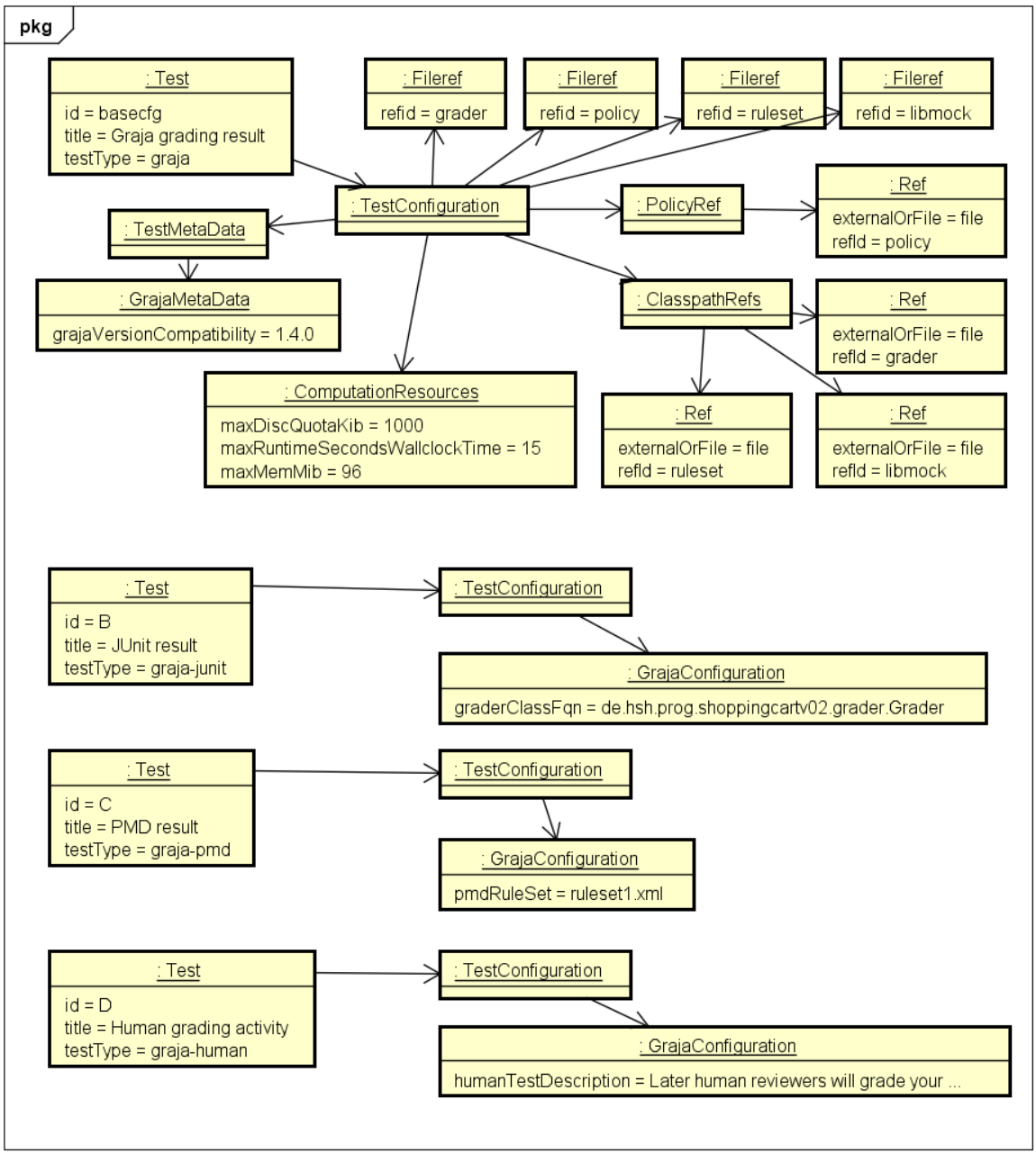

Abbildung 3: Objektmodell der Tests einer Beispielaufgabe in Graja

2.3 Teiltests

Die Tests B, C und D werden, wie oben erwähnt, durch spezielle Testkonfigurationen spezifiziert. Wir werfen einen detaillierten Blick auf die Konfigurationen der durchzuführenden Werkzeuge JUnit, PMD und Mensch. Wir beginnen mit JUnit.

2.3.1 JUnit-Test

Im gewählten Beispiel versteckt sich hinter der Datei mit der id „grader“ eine jar-Datei. Der Bytecode in dieser jar-Datei wird in diesem Beispiel aus drei[7] verschiedenen Java-Quelldateien gespeist, wovon beispielhaft eine in Abbildung 4 gekürzt wiedergegeben sei.

[7] Viele Graja-Aufgaben benötigen lediglich eine Quelldatei. Die Anzahl der Quelldateien, auf die der Aufgabenautor die JUnit-Testmethoden aufteilt, ist frei konfigurierbar.

Ohne jede Testmethode im Detail verstehen zu müssen, können wir erkennen, dass eine Testmethode attributesShouldBePrivate überprüft, ob die Einreichung ausschließlich private Attribute oder Klassenkonstanten besitzt. Dazu nutzt die Testmethode die Graja-Bibliotheksfunktionen der Klasse Support, welche wiederum Java Reflection einsetzt, um die Einreichung zu untersuchen.

Die zweite komplett dargestellte Testmethode canCalculateCostAfterSetQuantity prüft die Einreichung durch einen dynamischen Test. Eine vom Studierenden geschriebene Klasse com.mymart.CartItem, welche einen Bestellposten in einem Webshop-Warenkorb repräsentiert, soll unter Angabe von Produktname, Anzahl und Preis pro Einheit instanziiert werden können. Außerdem soll die Anzahl durch Aufruf der Methode com.mymart.CartItem#setQuantity verändert werden können. Danach soll die Methode com.mymart.CartItem#getCost den korrekten Gesamtwert des Bestellpostens berechnen.

Bei der ersten Testmethode attributesShouldBePrivate handelt es sich um eine Prüfung der Einreichung auf Einhaltung objektorientierter Prinzipien. Die zweite Testmethode canCalculateCostAfterSetQuantity prüft die funktionale Korrektheit der eingereichten Lösung.

Da beide Methoden Teil derselben JUnit-Testklasse sind und JUnit die Testklasse als eine Einheit zur Ausführung bringt, stehen die beiden Testergebnisse zu den beiden Testmethoden nachher einfach hintereinander. Zumindest dann, wenn wir keine besonderen Maßnahmen für eine „übersichtlichere“ Darstellung der Testergebnisse ergreifen.

Die Datei mit der id „libmock“ deutet an, dass der JUnit-Test aus komplexeren Testmethoden bestehen kann, die Drittbibliotheken einsetzen. Für die Zwecke dieses Beitrages werden wir uns jedoch auf die beiden oben dargestellten Beispiel-Testmethoden beschränken.

```
@DemandLocale(country= "US", language= "en")
public class CartItemTest extends AssignmentGrader {

  private static Class<? extends Object> submission;

  public static Class<?> getSubmissionClass() {
    synchronized(CartItemTest.class) {
      if (submission == null) submission= getPublicClassForName("com.mymart.CartItem");
      return submission;
    }
  }
  @BeforeClass public static void setUpOnce() {
    getSubmissionClass();
  }
  @Test public void canConstructCartItem() { ... }
  @Test public void cannotConstructInvalidCartItem() { ... }
  @Test public void cannotSetInvalidQuantity() { ... }
  @Test public void canCalculateCost() { ... }
  @Test public void shouldHaveCorrectToStringMethod() { ... }

  @Test public void attributesShouldBePrivate() {
    Support.assertAllAttributesArePrivateOrClassConstants(getSubmissionClass());
  }
  @Test public void canCalculateCostAfterSetQuantity() {
    int quantity0= 4, quantity= 6;
    Object submissionInstance= createSubmissionCartItemDogFood(quantity0);
    ReflectionSupport.invoke(submissionInstance, "setQuantity", quantity);
    double expected= getDogFoodCost(quantity);
    double observed= ReflectionSupport.invoke(submissionInstance, double.class, "getCost");
    org.junit.Assert.assertTrue(
      "Given a cart item with (pricePerUnit="+PRICE_PER_UNIT+", quantity="+quantity0+"). "
         +"Then call setQuantity("+quantity+"). Then call getCost. "
         +"Expected a value close to "+expected+". Observed: "+observed,
      Math.abs(expected-observed)<1E-5);
  }

  private static final double PRICE_PER_UNIT= 3.2;
  private static final String NAME= "Dog food";
  private double getDogFoodCost(int quantity) {
    return quantity*PRICE_PER_UNIT;
  }
  static Object createSubmissionCartItemDogFood(int quantity) {
    return createSubmissionCartItem(NAME, quantity, PRICE_PER_UNIT);
  }
  static Object createSubmissionCartItem(String name, int quantity, double pricePerUnit) {
    return ReflectionSupport.create(getSubmissionClass(), name, quantity, pricePerUnit);
  }
}
```

Abbildung 4: Eine gekürzte JUnit-Testklasse

```
<?xml version="1.0"?>
<ruleset name="Simple rules"
         xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
http://pmd.sourceforge.net/ruleset_2_0_0.xsd">
  <description>A few simple rules</description>
  <rule ref="rulesets/java/basic.xml/CollapsibleIfStatements"/>
  <rule ref="rulesets/java/naming.xml/VariableNamingConventions">
    <description>Variables should be named conventionally.</description>
  </rule>
  <rule ref="rulesets/java/naming.xml/MethodNamingConventions">
    <description>Methods should be named conventionally.</description>
  </rule>
  <rule ref="rulesets/java/naming.xml/ClassNamingConventions">
    <description>Classes should be named conventionally.</description>
  </rule>
  <rule ref="rulesets/java/strings.xml/UseEqualsToCompareStrings"/>
  <rule ref="rulesets/java/design.xml/SimplifyBooleanReturns"/>
  <rule ref="rulesets/java/design.xml/LogicInversion"/>
  <rule ref="rulesets/java/design.xml/FieldDeclarationsShouldBeAtStartOfClass"/>
  <rule ref="rulesets/java/comments.xml/CommentRequired">
    <description>
      Leading comments are required before a class and a (public or protected) method.
      'headerCommentRequirement' means a missing comment in front of a class.
      'publicMethodCommentRequirement' means a missing comment in front of a public method,
      and so on.
    </description>
    <properties>
      <property name="fieldCommentRequirement" value="Ignored"/>
      <property name="protectedMethodCommentRequirement" value="Required"/>
      <property name="publicMethodCommentRequirement" value="Required"/>
      <property name="headerCommentRequirement" value="Required"/>
    </properties>
  </rule>
  <rule ref="rulesets/java/design.xml/AvoidDeeplyNestedIfStmts">
    <properties>
      <property name="problemDepth" value="3"/>
    </properties>
  </rule>
  <rule ref="rulesets/java/naming.xml/BooleanGetMethodName"/>
  <rule ref="rulesets/java/naming.xml/ShortClassName">
    <description>Short class names with fewer than four characters are not
recommended</description>
    <properties>
      <property name="minimum" value="4"/>
    </properties>
  </rule>
</ruleset>
```

Abbildung 5: Konfiguration des PMD-Tests

2.3.2 PMD-Test

Der PMD-Test mit der id C referenziert eine Datei namens ruleset1.xml, die wir in Abbildung 5 abdrucken. Es handelt sich um eine Datei mit einigen Standardregeln sowie einigen angepassten Regeln, wie sie vom Testwerkzeug PMD unverändert importiert werden können.

Auch hier geht es nicht darum, jede Regel im Detail zu verstehen. Wir erkennen einige Regeln, die die Einreichung bzgl. der Einhaltung von Code-Konventionen prüfen (bspw.

VariableNamingConventions, FieldDeclarationShouldBeAtStartOfClass), einige Regeln beziehen sich auf unnötig aufgeblähten oder unnötig komplizierten Code (CollapsibleIfStatements, LogicInversion, SimplifyBooleanReturns, AvoidDeeplyNestedIfStmts), weitere Regeln beziehen sich auf fehlende Kommentierung (CommentRequired). Die bisher genannten Regeln können unter dem Oberbegriff der Wartbarkeit subsumiert werden. Einen anderen Prüfaspekt adressiert die Regel „UseEqualsToCompareStrings“, deren Nichteinhaltung auf einen funktionalen Fehler in der Einreichung hinweist.

Ähnlich wie die Testmethoden beim JUnit-Test sind die PMD-Regeln Teil derselben PMD-Ausführung. Die Ergebnisse stehen am Ende einfach hintereinander, zumindest dann, wenn wir keine besonderen Maßnahmen für eine „übersichtlichere“ Darstellung der Testergebnisse ergreifen.

2.3.3 Der manuelle Test

Das Test-Objekt D in Abbildung 3 spezifiziert die manuelle Prüfung einer Einreichung durch einen Menschen. In der obigen Beispielaufgabe enthält der Aufgabentext den Hinweis, dass Stringverkettungen in toString-Methoden unter Einsatz der Standardbibliotheksklasse StringBuilder erfolgen sollen. Weiterhin besteht der Anspruch, dass die Einreichung möglichst keine Coderedundanzen aufweist. Tutoren sind nach Ablauf der Einreichungsfrist damit befasst, die Einhaltung dieser und weiterer Vorgaben zu prüfen und ggf. zu kommentieren.

Eine Testkonfiguration des Testwerkzeugs Mensch ist hier eine Arbeitsanweisung. Im oben gewählten Ansatz steht die Arbeitsanweisung als Beschreibung im Attribut humanTestDescription und außerdem in weiteren unabhängigen Dokumenten, die mit der XML-Datei in einem Zip-Archiv gebündelt werden.

Wie bei den Testmethoden beim JUnit-Test und bei den Regeln im PMD-Test gilt auch hier, dass sich das Testwerkzeug Mensch mit verschiedenen zu bewertenden Aspekten der Einreichung befasst. Die Nutzung des StringBuilder bewertet Wissen und Fertigkeiten im Umgang mit der Standardbibliothek und ggf. die Kenntnis der Effizienzauswirkungen. Die Begutachtung hinsichtlich Redundanzen im Code betrifft die Fähigkeit des Einreichenden, wartbaren Code zu schreiben.

3 Bewertungsergebnis

Gegenwärtig gibt es noch kein standardisiertes Format für die Darstellung eines Bewertungsergebnisses eines Autobewerters. Eine im eCULT[8]-Projekt bereits erfolgte Betrachtung der Gemeinsamkeiten verschiedener Autobewerter lässt die folgenden Datenelemente zur Beschreibung eines Bewertungsergebnisses als sinnvoll erscheinen:

- Ein quantifizierbares Ergebnis (bestanden/nicht bestanden, oder eine Punktzahl, oder ein Erfolgsgrad im Wertebereich 0%-100%).
- Ein Kommentar (in der Regel textuell, aber auch Bilder sind denkbar).

3.1 An den Testwerkzeugen orientierte Struktur

Es ist zunächst naheliegend, das Bewertungsergebnis entlang der Testwerkzeuge zu strukturieren. Wir abstrahieren von dem konkreten Beispiel des vorhergehenden Abschnitts und geben beispielhaft einige Aspekte an, die Teil eines formativen Assessments sein können. Wie üblich beschränkt sich das Feedback nicht auf syntaktische und funktionale Aspekte, sondern auch auf diverse Qualitätsaspekte wie Wartbarkeit, Effizienz, etc. Die folgende Abbildung 6 listet einige Bewertungsaspekte auf und gruppiert diese nach den

[8] Dieser Beitrag wurde als Teil des Projekts „eCompetence and Utilities for Learners and Teachers“ (eCULT) vom Bundesministerium für Bildung und Forschung (BMBF) gefördert (Förderkennzeichen 01PL11066D).

Testwerkzeugen, die das Potential haben, den Aspekt abzudecken. Manche Aspekte verteilen sich auf verschiedene Werkzeuge. So ist das einheitliche Codelayout (Aspekt C.ii) durch eine statische Codeanalyse ermittelbar. Die Lesbarkeit des Codelayouts (Aspekt D.ii) wird vermutlich besser von einem Menschen beurteilt. Ein anderes Beispiel: die Einhaltung funktionaler Invarianten kann teilweise durch statische Codeanalyse geprüft werden (bspw. durch die PMD-Regel OverrideBothEqualsAndHashcode, Aspekt C.iii), teilweise durch dynamische Softwaretests (Aspekt B.iii).

Testwerkzeug	**Aspekt mit am Testwerkzeug orientierter Nummerierung**	
Compiler	Ist das studentische Programm syntaktisch korrekt?	A.i
Dynamischer Softwaretest	Liefert das Programm korrekte Ausgaben?	B.i
	Werden auch Grenzfälle und Fehleingaben berücksichtigt?	B.ii
	Werden funktionale Vor- und Nachbedingungen und funktionale Invarianten eingehalten?	B.iii
	Ist das Laufzeitverhalten und der Ressourcenverbrauch des Programms für große Eingaben angemessen?	B.iv
	Sind Ausgaben des Programms wie angefordert?	B.v
Statische Code-/Bytecode-analyse	Werden Bezeichner konventionsgemäß verwendet?	C.i
	Ist das Codelayout (Einrückung, Leerräume) einheitlich?	C.ii
	Werden funktionale Vor- und Nachbedingungen und funktionale Invarianten eingehalten?	C.iii
	Wurde der Code angemessen kommentiert?	C.iv
	Ist redundanter Code vorhanden?	C.v
	Werden Grundprinzipien wie lose Kopplung / hohe Kohäsion berücksichtigt?	C.vi
	Werden Daten vor unberechtigten Zugriffen geschützt?	C.vii
Mensch	Ist der Code lesbar?	D.i
	Ist das Codelayout (Einrückung, Leerräume) lesbar?	D.ii
	Werden Daten vor unberechtigten Zugriffen geschützt?	D.iii
	Sind Ausgaben des Programms verständlich?	D.iv
	Sind Bedienelemente intuitiv angeordnet?	D.v

Abbildung 6: Beispielhafte und nach Testwerkzeug gruppierte Bewertungsaspekte

Im Detail kann man über die Zuordnung von Testwerkzeugen zu Bewertungsaspekten streiten. Für jeden Aspekt lassen sich möglicherweise besser geeignete, spezialisierte Werkzeuge und Algorithmen finden, die den Aspekt einer automatischen Bewertung zuführen. Hier soll es nicht um eine endgültige Zuordnung von Testwerkzeugen zu Bewertungsaspekten gehen, sondern darum, dass jedes Testwerkzeug seine ganz spezifischen Stärken bzgl. bestimmter Bewertungsaspekte hat.

3.1.1 Beispiel

Wir illustrieren ein mögliches, nach den Testwerkzeugen strukturiertes Bewertungsergebnis zur in Abschnitt 2 beschriebenen Aufgabe in Abbildung 7, so wie es in ähnlicher Form von Graja in einer früheren Programmversion generiert werden konnte. Wir haben bewusst die eigentlich in Graja vorhandene Spalte mit Angaben zu erreichten Punkten weggelassen, weil diese noch nicht standardisiert ist und weil wir im weiteren Verlauf die Angabe von erreichbaren Punkten sowieso von der Konfiguration von Tests trennen wollen.

Graja grading result	**Score**
JUnit result	
• *Success in de.hsh.prog.shoppingcartv02.grader.CartItemTest#canConstructCartItem.*	100 %
• *Success in de.hsh.prog.shoppingcartv02.grader.CartItemTest#cannotConstructInvalidCartItem.*	100 %
• *Success in de.hsh.prog.shoppingcartv02.grader.CartItemTest#cannotSetInvalidQuantity.*	100 %
• *Success in de.hsh.prog.shoppingcartv02.grader.CartItemTest#canCalculateCost.*	100 %
• *Success in de.hsh.prog.shoppingcartv02.grader.CartItemTest#shouldHaveCorrectToStringMethod.*	100 %
• *Error in de.hsh.prog.shoppingcartv02.grader.CartItemTest#attributesShouldBePrivate:* There are at least 3 non-private attributes in com.mymart.CartItem.	0 %
• *Error in de.hsh.prog.shoppingcartv02.grader.CartItemTest#canCalculateCostAfterSetQuantity:* For a cart item with (pricePerUnit=3.2, quantity=6) your method 'com.mymart.CartItem.getCost' must return a value close to 19.2. Observed: 9.2.	0 %
PMD result	
• *Success in CollapsibleIfStatements.*	100 %
• *Error in VariableNamingConventions:* *Variables should be named conventionally.* ○ Variables should start with a lowercase character, 'Name' starts with uppercase character. `CartItem.java` `10 : public CartItem(String Name, int quantity, double pricePerUnit) {`	0 %
• *Success in MethodNamingConventions.*	100 %
• *Success in ClassNamingConventions.*	100 %
• *Error in UseEqualsToCompareStrings:* *Using '==' or '!=' to compare strings only works if intern version is used on both sides. Use the equals() method instead.* ○ Use equals() to compare strings instead of '==' or '!=' `CartItem.java` `11 : if (Name=="") throw new IllegalArgumentException("name is empty");`	0 %
• *Success in SimplifyBooleanReturns.*	100 %
• *Error in LogicInversion:* *Use opposite operator instead of negating the whole expression with a logic complement operator.* ○ Use opposite operator instead of the logic complement operator. `CartItem.java` `22 : if ( ! (quantity >= 1) ) {`	0 %
• *Success in FieldDeclarationsShouldBeAtStartOfClass.*	100 %
• *Success in CommentRequired.*	100 %
• *Success in AvoidDeeplyNestedIfStmts.*	100 %
• *Success in BooleanGetMethodName.*	100 %
• *Success in ShortClassName.*	100 %
Human grading activity	
• Later human reviewers will grade your submission and credit points for the following aspects: a) String concatenation using StringBuilder, b) no code redundancy.	0 %

Abbildung 7: Bewertungsergebnis für die Beispielaufgabe, strukturiert nach Testwerkzeugen

3.2 An didaktisch sinnvollen Überschriften orientierte Struktur

Das Bewertungsergebnis des vorhergehenden Abschnitts ist nicht völlig unübersichtlich. Dennoch äußern nicht wenige Studierende, dass sie eine gewisse Struktur vermissen. Wir vergleichen die Abbildung 6 mit dem Feedback, welches üblicherweise im Rahmen eines im Übungsraum von Menschen durchgeführten formativen Assessment geäußert wird. Das Feedback wird meist von Lehrpersonen unter gedachten Überschriften gruppiert. Denkbar sind etwa die in Abbildung 8 dargestellten, teilweise einer Softwarequalitätsnorm [4]

entlehnten Überschriften, wobei Feedback zu den im Folgenden drei erstgenannten Bereichen bei Programmieranfängern häufig den größten Raum einnimmt. Neben der Überschrift nennen wir die zugehörigen Aspekte aus Abbildung 6 zusammen mit der dort vergebenen Nummer.

Überschrift	**Aspekt**		
Sprache und Syntax	1a	Ist das studentische Programm syntaktisch korrekt?	A.i
	1b	Werden Bezeichner konventionsgemäß verwendet?	C.i
	1c	Ist das Codelayout (Einrückung, Leerräume) einheitlich / lesbar?	C.ii / D.ii
Funktion	2a	Liefert das Programm korrekte Ausgaben?	B.i
	2b	Werden auch Grenzfälle und Fehleingaben berücksichtigt?	B.ii
	2c	Werden funktionale Vor- und Nachbedingungen und funktionale Invarianten eingehalten?	B.iii / C.iii
Wartbarkeit	3a	Wurde der Code angemessen kommentiert?	C.iv
	3b	Ist redundanter Code vorhanden?	C.v
	3c	Werden Grundprinzipien wie lose Kopplung / hohe Kohäsion berücksichtigt?	C.vi
	3d	Ist der Code lesbar?	D.i
Effizienz	4a	Ist das Laufzeitverhalten und der Ressourcenverbrauch des Programms für große Eingaben angemessen?	B.iv
Sicherheit	5a	Werden Daten vor unberechtigten Zugriffen geschützt?	C.vii / D.iii
Benutzbarkeit	6a	Sind Ausgaben des Programms verständlich / wie angefordert?	B.v / D.iv
	6b	Sind Bedienelemente intuitiv angeordnet?	D.v

Abbildung 8: Beispielhafte und nach einer Qualitätsnorm gruppierte Bewertungsaspekte

Ziel dieses Beitrages ist, ein Bewertungsergebnis wie in Abbildung 8 zu erzeugen, die losgelöst von der Spezifikation von Testwerkzeugen eine eigene, nach didaktischen Kriterien konzipierte Struktur aufweist. Wir wollen in den folgenden Abschnitten untersuchen, welche Möglichkeiten bestehen, ein auf diese Weise strukturiertes Bewertungsergebnis erzeugen.

Die Struktur der Abbildung 8 ist nicht die einzige sinnvolle Struktur. Es soll in diesem Beitrag nicht darum gehen, ein Bewertungsschema als das beste vorzugeben, sondern es soll darum gehen, der Lehrperson die Freiheit zu geben, das Feedback dem Lehrkontext entsprechend zu strukturieren und dabei nicht an die Spezifikation von Testwerkzeugen gebunden zu sein. Andere Strukturierungsmöglichkeiten sind denkbar und je nach Lehrkontext didaktisch sinnvoll:

- Wenn eine Aufgabe aus mehreren Teilaufgaben besteht, könnte die Überschrift die Nummer der Teilaufgabe sein.
- Denkbar ist, dass die Lehrperson das Feedback nach Schwierigkeit strukturiert. Erst kommen leicht zu behebende Fehler, dann schwerer zu behebende.

Im weiteren Verlauf dieses Beitrages werden wir uns exemplarisch auf die in Abbildung 8 dargestellte Gruppierung beziehen. Unabhängig von der tatsächlich gewählten Strukturierungs-Strategie ist erkennbar, dass die Reihenfolge der einzelnen Bewertungsaspekte nicht notwendigerweise der Reihenfolge entspricht, in der die Aspekte in den Test-Objekten und den dazu gehörigen Testkonfigurationen spezifiziert sind.

3.3 Orthogonalität von Tests und Bewertungsaspekten

Bewertungsaspekte und Testwerkzeuge lassen sich als zwei Dimensionen einer Matrix auffassen (vgl. Abbildung 9). In dieser Sichtweise bezeichnen die Nummern in der zweiten

Spalte (z. B. B.iii) keine Bewertungsaspekte mehr, sondern sie bezeichnen Teilausführungsergebnisse des Testwerkzeugs. B.iii könnte z. B. das Ergebnis der Ausführung einer einzelnen Testmethode sein, C.iv das Ergebnis der Ausführung einer einzelnen PMD-Regel, D.i das Ergebnis der menschlichen Sichtung der Einreichung hinsichtlich eines einzelnen Aspekts.

Aspektgruppe ➞		1			2			3				4	5	6	
⬇ Testwerkzeug		a	b	c	a	b	c	a	b	c	d	a	a	a	b
Compiler	A.i	X													
Dynamischer Softwaretest	B.i				X										
	B.ii					X									
	B.iii						X								
	B.iv											X			
	B.v													X	
Statische Code-/Bytecode-analyse	C.i		X												
	C.ii			X											
	C.iii						X								
	C.iv							X							
	C.v								X						
	C.vi									X					
	C.vii												X		
Mensch	D.i										X				
	D.ii			X											
	D.iii												X		
	D.iv													X	
	D.v														X

Abbildung 9: Von Testwerkzeugen geleistete Bewertungsbeiträge zu den Bewertungsaspekten. Ein X bedeutet, dass ein Teilausführungsergebnis des Testwerkzeugs (Zeile) einen Bewertungsbeitrag zum Bewertungsaspekt (Spalte) leistet.

4 Spezifikation von Test-Teilausführungen im ProFormA-Aufgabenformat

In Abschnitt 3.3 wird offenbar, dass wir zur Veränderung der Struktur des Bewertungsergebnisses die Testausführung in Test-Teilausführungen aufbrechen müssen. Jede Test-Teilausführung muss separat adressierbar sein, um das Ergebnis der Test-Teilausführung danach an der gewünschten Stelle im Gesamtfeedback unterbringen zu können. Test-Teilausführungen sind bereits in der bisherigen Aufgabenbeschreibung adressierbar. Testmethodennamen und PMD-Regeln sind die dazu benötigten Bezeichner. Um eine einheitliche Adressierung zu erreichen, wollen wir Test-Teilausführungen als separate Test-Objekte mit einer eigenen id in die XML-Datei (vgl. Abbildung 1) aufnehmen.

4.1 Test-Teilausführungen sind Tests

Dazu erweitern wir das UML-Klassenmodell der Abbildung 2 um die Möglichkeit, Teilausführungen von JUnit-Tests bzw. von PMD-Tests zu adressieren (vgl. Abbildung 10). In unserem Fall sind dies einzelne Testmethoden und einzelne PMD-Regeln, die wir mit den Attributen methodName und pmdRule bezeichnen. Außerdem zerlegen wir die humanTestDescription in mehrere humanAspectDescriptions, um auch für das Testwerkzeug Mensch Teilausführungen für jeden einzelnen zu begutachtenden Aspekt spezifizieren zu können.

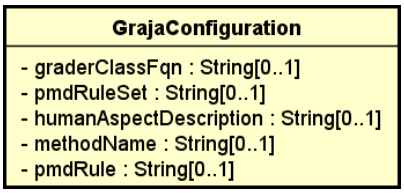

Abbildung 10: Erweitertes UML-Modell

Nun nutzen wir diese neue Möglichkeit und konfigurieren Tests zusammen mit ihren Test-Teilausführungen. Beispielhaft ist dies in Abbildung 11 für das Testwerkzeug JUnit dargestellt. Jede Testmethode erhält eine eigene id in einem eigenen Test-Objekt.

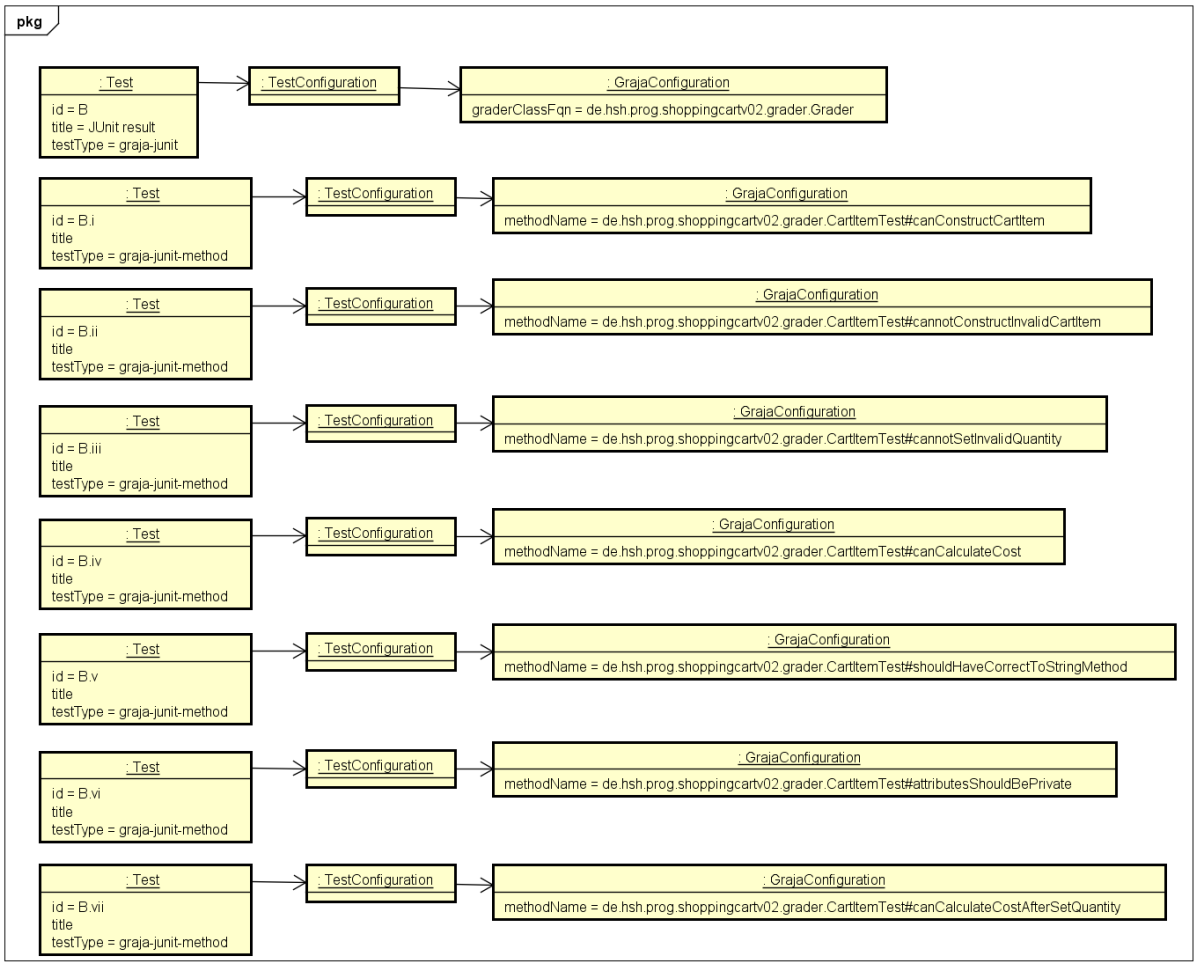

Abbildung 11: Objektmodell der Beispielaufgabe mit Test-Teilausführungsspezifikation.

Das Test-Objekt B repräsentiert eine JUnit-Testausführung. Die zusätzlichen Test-Teilausführungs-Objekte deklarieren, dass der JUnit-Runner während der Ausführung für die benannten Testmethoden das jeweilige Testergebnis so ablegen muss, dass es für einen späteren Abruf im Rahmen der Erzeugung des Gesamtfeedbacks zur Verfügung steht.

Warum haben wir in Abbildung 11 das Test-Objekt B überhaupt noch separat aufgenommen? Stattdessen hätten wir dieses ja entfernen können und die Konfiguration des graderClassFqn in jedes Teiltest-Objekt duplizieren können. Neben der redundanten Konfiguration, die das mit sich brächte, sprechen weitere Gründe für die separate Konfiguration des JUnit-Tests, die wir im weiteren Verlauf dieses Beitrags diskutieren werden.

Für PMD können entsprechend Test-Teilausführungen spezifiziert werden, indem für jede PMD-Regel ein eigenständiges Test-Objekt in der XML-Datei spezifiziert wird (vgl. Abbildung 13). Und auch für das Testwerkzeug Mensch bietet es sich an, den Testvorgang, der ja eigentlich zwei Aspekte umfasst, explizit in zwei Test-Objekte aufzufächern (vgl. Abbildung 12).

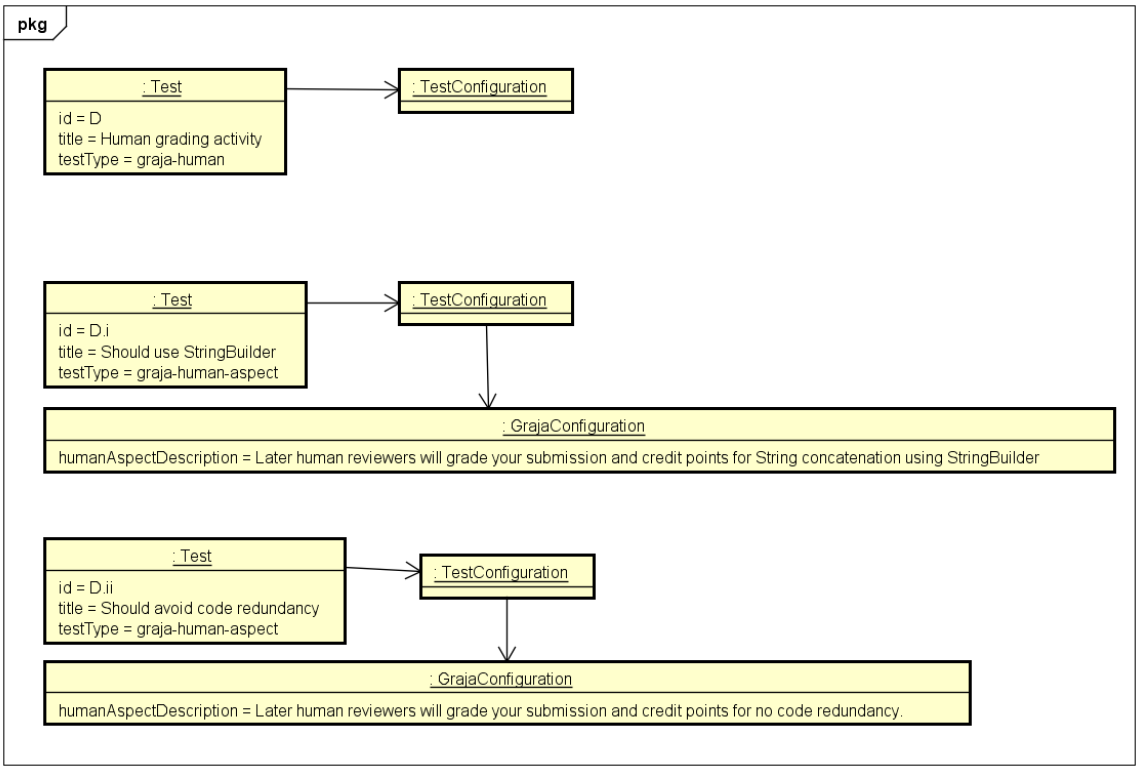

Abbildung 12: Objektmodell der Beispielaufgabe mit Test-Teilausführungsspezifikationen für die einzelnen von Menschen zu bewertenden Aspekte.

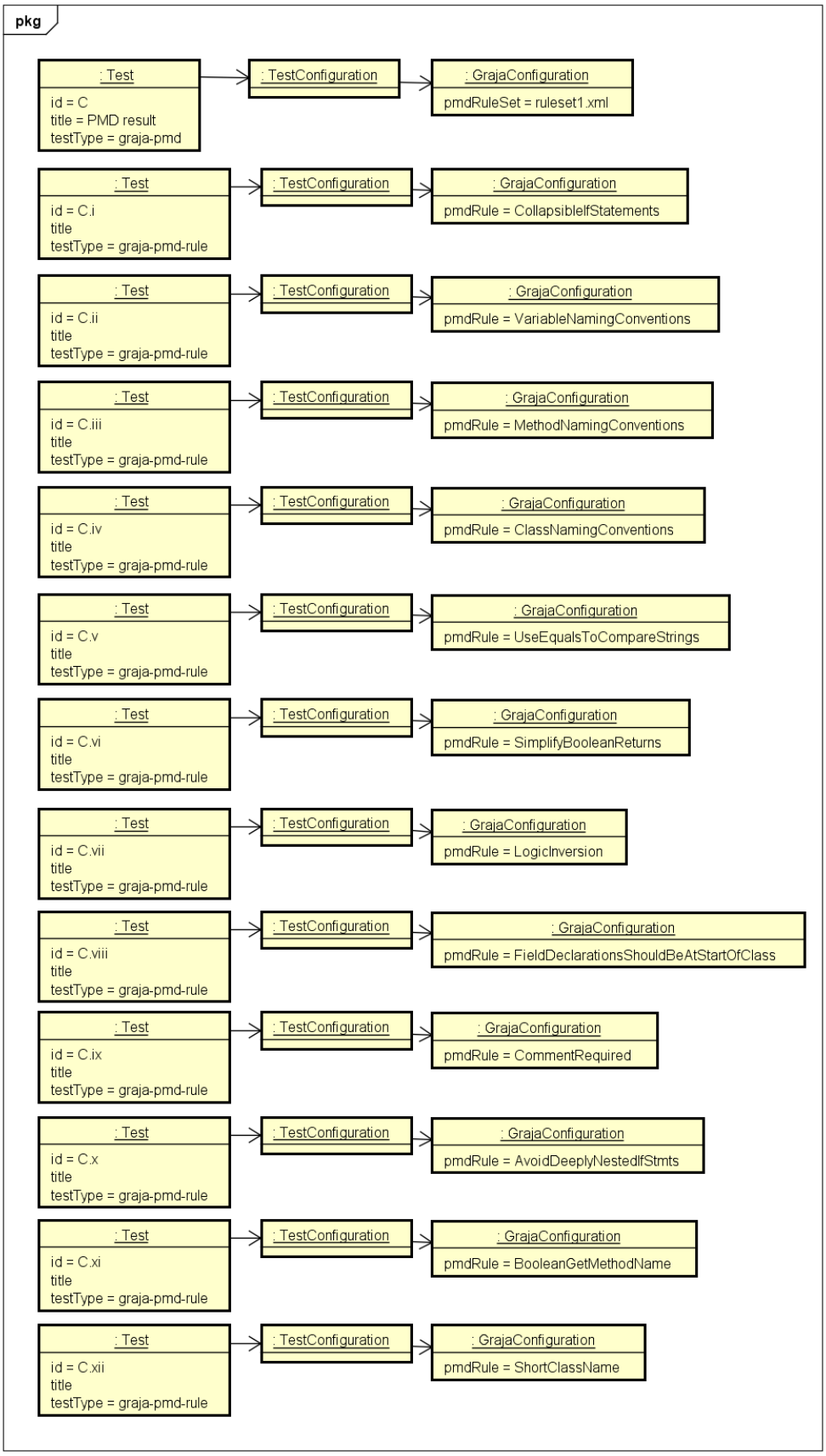

Abbildung 13: Objektmodell der Beispielaufgabe mit Test-Teilausführungsspezifikationen für die einzelnen PMD-Regeln.

Wir widmen uns nun den Zusammenhängen innerhalb von Testausführungen und Test-Teilausführungen. In diesem Kontext ist es wesentlich zu wissen, ob ein Testwerkzeug

einmal oder mehrmals ausgeführt werden soll. Wird es nur einmal ausgeführt, ist die Zuordnung von Test-Teilausführungsergebnissen zu einer Testausführung trivial. Ansonsten benötigt man eine Datenstruktur, die die Zuordnung garantiert.

4.2 Welche Testwerkzeuge werden wie oft ausgeführt?

Zunächst müssen wir dazu etwas ausholen. In Abschnitt 3.1 haben wir die vier Werkzeugklassen Compiler, Dynamischer Softwaretest, Statische Codeanalyse und Mensch geprägt. Welches dieser Werkzeuge wollen wir wie oft ausführen? Auf den ersten Blick erscheint die Frage überflüssig und trivial: jedes Werkzeug einmal. Der Compiler muss in der Regel nur einmal gestartet werden und übersetzt dabei alle eingereichten Quelltexte. Ebenso untersucht das Werkzeug der statischen Codeanalyse alle Quelltexte in einem Durchgang und führt der dynamische Softwaretest alle durchzuführenden Testroutinen in einem Durchgang aus. Der Mensch als Testwerkzeug wird sich auch nur einmal an eine Einreichung setzen und alle zu begutachtenden Aspekte sofort und in einer Sitzung prüfen.

Wozu also die Frage? Nun, es gibt Situationen, in denen die mehrfache Ausführung eines Testwerkzeuges geboten ist, und andere Situationen, in denen die mehrfache Ausführung eines Testwerkzeuges nicht möglich oder zumindest nicht sinnvoll ist.

4.2.1 Mehrfache Ausführung des Testwerkzeugs PMD

PMD verwendet sog. Rulesets zur Konfiguration einer Ausführung. Rulesets bestehen in der Regel aus Referenzen auf bestehende, mit PMD mitgelieferte Regeln (vgl. Abschnitt 2.3.2). Einige der Regeln können im Rahmen der Referenzierung für den jeweiligen Kontext parametriert werden. So zeigt Abbildung 5 z. B. die Parametrierung der Regel CommentRequired durch sog. Properties, die beschreiben, welche Kommentare verpflichtend sind. Es ist nun denkbar, dass dieselbe Regel CommentRequired mehrfach, aber in verschiedenen Parametrierungen genutzt werden soll. Die Parametrierung, die Klassenkommentare verpflichtend macht, soll am Ende mit einem Punktgewicht von X in das Gesamtbewertungsergebnis eingehen, die Parametrierung, die Kommentare vor öffentlichen Methoden verpflichtend macht, soll am Ende das Punktgewicht Y erhalten. Genau dies ist nun leider nicht möglich, in einem einzigen Durchlauf von PMD zu bewerkstelligen. Die referenzierte Regel CommentRequired kann nicht in zwei verschiedenen Parametrierungen im gleichen PMD-Durchgang verwendet werden. Dies mag man als Mangel des Werkzeuges PMD betrachten. Grundsätzlich kann eine solche Einschränkung jedoch bei jedem Testwerkzeug bestehen.

Wir halten also fest: es kann notwendig sein, ein und dasselbe Testwerkzeug mehrfach in dann vermutlich unterschiedlicher Parametrierung durchzuführen.

4.2.2 Einfache Ausführung des Testwerkzeugs JUnit

JUnit kann zumindest in neueren Versionen gezielt einzelne Testmethoden zur Ausführung bringen. Es wäre denkbar, jedes einzelne in Abbildung 11 dargestellte Test-Objekt mit Testtyp graja-junit-method durch einen separaten JUnit-Runner zur Ausführung zu bringen. Der Vorteil wäre, dass jedes Test-Teilausführungsergebnis leicht abgreifbar wäre. Der Nachteil ist ein deutlicher Effizienzverlust, wenn JUnit mehrfach gestartet werden muss.

Schwerwiegender, aber noch nicht abschließend analysiert, wiegt ein anderes Argument: Wenn JUnit mehrfach gestartet wird, heißt das nicht, dass jede Testmethode vollkommen isoliert getestet werden kann. Beispielsweise muss eine studentische Klasse zuerst einmal vom Klassenlader geladen werden. Schon dabei können Fehler auftreten, die der allerersten Ausführung von JUnit angelastet würden, obwohl es eigentlich Fehler übergreifender Natur sind. Weiterhin enthält eine Testklasse üblicherweise eine Methode, die mit @BeforeClass annotiert ist. Die dort durchgeführten Initialisierungen sollen nur einmal für den gesamten Testlauf ausgeführt werden. Was passiert, wenn JUnit nun mehrfach gestartet wird. Ist die in der @BeforeClass-Methode implementierte Initialisierung robust gegen mehrfache Ausführung? Die Programmierung von Testklassen würde für Aufgabenautoren erheblich

komplexer und fehleranfälliger, wenn er damit rechnen müsste, dass eine JUnit-Testklasse mehrfach ausgeführt wird. Man mag dies als Mangel des Werkzeuges JUnit betrachten. Grundsätzlich kann bei jedem Testwerkzeug die Einschränkung bestehen, dass die mehrfache Ausführung des Werkzeuges dessen Verwendung verkompliziert oder gar unmöglich macht.

Wir halten also fest: es kann notwendig sein, ein Testwerkzeug nur ein einziges Mal auszuführen, auch wenn dessen Test-Teilausführungsergebnisse an verschiedenen Stellen des Bewertungsfeedbacks einfließen sollen.

4.3 Konsequenz: eine Hierarchie von Tests

Testausführungsobjekte und Test-Teilausführungsobjekte müssen einander zugeordnet werden. Die flache Sammlung von Test-Objekten im ProFormA-Aufgabenformat bietet dazu keinen expliziten Mechanismus an. Die Verwendung von hierarchischen Schlüsseln (Attribut id in Test) wäre denkbar, würde jedoch die Generierung von Schlüsseln verkomplizieren.

Ein naheliegendes Modell zur Strukturierung der Testausführung ist ein Baum von Test-Objekten. Beispielhaft zeigen wir im Folgenden einen solchen Baum:

- Tests
  - Test B
    - Test B.i
    - Test B.ii
    - Test B.iii
    - Test B.iv
    - Test B.v
    - Test B.vi
    - Test B.vii
  - Test C
    - Test C.i
    - Test C.ii
    - Test C.iii
    - Test C.iv
    - Test C.v
    - Test C.vi
    - Test C.vii
    - Test C.viii
    - Test C.ix
    - Test C.x
    - Test C.xi
    - Test C.xii
  - Test D
    - Test D.i
    - Test D.ii

Wären in unserem Beispiel mehrere unterschiedlich parametrierte Instanzen derselben PMD-Regel enthalten, würde der Baum etwas anders aussehen, etwa so (Änderungen sind hervorgehoben):

- Tests
  - Test B
    - Test B.i
    - Test B.ii
    - Test B.iii
    - Test B.iv
    - Test B.v
    - Test B.vi
    - Test B.vii
  - Test **C‘**
    - Test C.i
    - Test C.ii
    - Test C.iii
    - Test C.iv

    - Test C.v
    - Test C.vi
    - Test C.vii
    - Test C.viii
    - Test **C.ix‘**
    - Test C.x
    - Test C.xi
    - Test C.xii
- **Test C“**
    - **Test C.ix“**
- Test D
    - Test D.i
    - Test D.ii

4.4 Vorschlag zur Erweiterung des ProFormaA-Aufgabenformats

Zur Umsetzung einer solchen Testhierarchie schlagen wir vor, das ProFormA-Aufgabenformat um ein Attribut im Domänenobjekt *Test* zu erweitern. Ein Test soll eine optionale *parentId* als Attribut erhalten, die den zugehörigen, übergeordneten Test referenziert. Tests, die eine parentId besitzen, nennen wir Subtests oder Teiltests oder Test-Teilausführungen. Einen Subtest könnte man in unserem Beispiel zur Aufnahme der Spezifikation einer JUnit-Testmethode bzw. zur Spezifikation einer PMD-Regel bzw. zur Spezifikation eines manuell zu bewertenden Aspekts nutzen.

Inwieweit die Hierarchie mehr als eine Ebene umfassen muss, ist noch nicht abschließend analysiert. Bisher haben wir lediglich Anwendungsfälle für die Konfiguration einer Subtest-Ebene unterhalb der Test-Ebene gefunden.

5 Spezifikation der Bewertungsstruktur

In Abschnitt 3 haben wir dargelegt, dass die Testausführungsstruktur nicht notwendigerweise eine gute Strukturvorlage für eine lernförderliche Aufbereitung des Bewertungsergebnisses ist. D. h. dass der Vorschlag des Abschnitts 4.4, Strukturen innerhalb von Tests und Subtests im ProFormA-Aufgabenformat explizit zu machen, nicht ausreicht. Wir benötigen eine unabhängige Strukturvorgabe für das Bewertungsfeedback.

5.1 Eine Hierarchie von Bewertungsaspekten

Ein naheliegendes Modell zur Strukturierung des Bewertungsfeedbacks ist ein Baum von Bewertungsaspekten. Jeder Bewertungsaspekt wird aus einer Testausführung oder Test-Teilausführung gespeist. Beispielhaft zeigen wir im Folgenden einen solchen Baum, dessen innere Knoten ähnlich wie in Abbildung 8 gewählt wurden:

- GradingAspectGroup, Title=Graja grading result
    - GradingAspectGroup, Title=Functional correctness
        - GradingAspect, Title=Can construct CartItem ➞ Test B.i
        - GradingAspect, Title=Can calculate cost ➞ Test B.iv
        - GradingAspect, Title=Should have correct toString method ➞ Test B.v
        - GradingAspect, Title=Can calculate cost after calling setQuantity ➞ Test B.vii
        - GradingAspect, Title=String comparison ➞ Test C.v
    - GradingAspectGroup, Title=Maintainability
        - GradingAspectGroup, Title=Encapsulation
            - GradingAspect, Title=All attributes in class CartItem should be private ➞ Test B.vi (*)
        - GradingAspectGroup, Title=Code conventions:
            - GradingAspect, Title=Variable naming conventions ➞ Test C.ii
            - GradingAspect, Title=Class naming conventions ➞ Test C.iv
            - GradingAspect, Title=Method naming conventions ➞ Test C.iii
            - GradingAspect, Title=Fields (attributes) should be at the start of the class ➞ Test C.viii
            - GradingAspect, Title=Boolean accessors should not be named get... ➞ Test C.xi
        - GradingAspectGroup, Title=Code Complexity
            - GradingAspect, Title=Collapsible if statements ➞ Test C.i

        - GradingAspect, Title=Unnecessary if-then-else ➞ Test C.vi
        - GradingAspect, Title=Bad use of logic complement operator ➞ Test C.vii
        - GradingAspect, Title=Avoid deeply nested if-statements ➞ Test C.x
        - GradingAspect, Title=(no title, borrowed from test) ➞ Test D.ii (**)
    - GradingAspectGroup, Title=Readability
        - GradingAspect, Title= Comments needed in front of methods and classes ➞ Test C.ix
        - GradingAspect, Title=Short class name ➞ Test C.xii
- GradingAspectGroup, Title=Reliability
    - GradingAspect, Title= Should monitor CartItem invariant in constructor ➞ Test B.ii
    - GradingAspect, Title= Should monitor CartItem invariant in setter ➞ Test B.iii
- GradingAspectGroup, Title=Security
    - GradingAspect, Title=Non-private attribute is a security risk ➞ Test B.vi (*)
- GradingAspectGroup, Title=Efficiency
    - GradingAspect, Title=(no title, borrowed from test) ➞ Test D.i (**)

Häufig wird es so sein, dass der Baum der Bewertungsaspekte lediglich eine Umordnung der (Sub-)Tests vornimmt. Jedoch bieten sich durch die separate Modellierung einer Bewertungshierarchie noch weitere Möglichkeiten:

1. Ergebnisse von Tests bzw. Subtests können mehrfach für verschiedene Bewertungsaspekte heran gezogen werden. Im obigen Beispiel ist dies für die Test-Teilausführung B.vi (Testmethode de.hsh.prog.shoppingcartv02.grader.CartItemTest#attributesShouldBePrivate) beispielhaft dargestellt. Die Verwendung von nicht-privaten Attributen hat sowohl ungünstige Auswirkungen auf die Wartbarkeit des Programms als auch auf die Sicherheit. Beide Aspekte werden im obigen Beispiel ins Feedback aufgenommen (s. die mit dem Symbol (*) gekennzeichneten Zeilen).
2. Die Beschreibung eines Bewertungsaspekts gliedert sich normalerweise in eine Überschrift und eine Detailbeschreibung. Den aus didaktischer Sicht relevanten Aspekt kann man gut in einem *Title*-Attribut des Bewertungsaspekts unterbringen. Der Title eines Bewertungsaspekts kann, muss aber nicht mit dem Title des referenzierten Tests übereinstimmen. Ein Test besitzt einen Title, der die technische Ausführung des Tests beschreibt. Der Title des Bewertungsaspekts fokussiert auf eine lernförderliche Formulierung. Während der Title des Tests eher in einem Ablaufprotokoll enthalten sein könnte, das es einer Lehrkraft erlaubt, den Ablauf des automatisierten Bewertungsprozesses zu kontrollieren, ist der Title des Bewertungsaspekts geeignet, direkt an den einreichenden Studierenden zurück gemeldet zu werden. Im obigen Beispiel sind in den mit (*) gekennzeichneten Zeilen zwei unterschiedliche Titles für ein und denselben referenzierten Test verwendet worden. Zudem ist in den mit (**) gekennzeichneten Zeilen illustriert, dass es möglich ist, auf einen separaten Bewertungsaspekt-Title zu verzichten, wenn der Title des Tests bereits ausreichend ist. Hier besitzen die beiden referenzierte Tests D.i und D.ii bereits für das Bewertungsfeedback einsetzbare Titel (vgl. Abbildung 12).
3. Nicht dargestellt sind Möglichkeiten der quantitativen Bewertung. Durch die Einführung einer von der Testhierarchie getrennten Bewertungshierarchie ist es möglich, einzelne Testausführungsergebnisse unterschiedlich zu bewerten. Es wäre möglich, für nicht-private Attribute 10 Strafpunkte aus Wartbarkeitssicht und nur 5 Strafpunkte aus Sicherheitssicht zu vergeben. Wir haben uns entschieden, Fragen der quantitativen Bewertung zunächst außen vor zu lassen. Denkbar ist jedoch, an jedem inneren Knoten und an jedem Blatt der Bewertungshierarchie Informationen zur quantitativen Bewertung zu hinterlegen. Im einfachsten Fall sind das Gewichte zur Bildung gewichteter Punktsummen. In komplizierteren Fällen stellen wir uns Bewertungsskripte der Art „wenn der Teilaspekt 1 nicht mindestens 50% erreicht, kann Teilaspekt 2 nicht mehr als 30 Punkte erhalten“ vor.

5.2 Beispiel in Graja

Unter Einsatz einer von der Testhierarchie getrennten Bewertungshierarchie haben wir in Graja das in Abbildung 15 dargestellte Bewertungsfeedback generiert. Im Vergleich mit Abbildung 7 stellen wir fest, dass das Ergebnis übersichtlicher und lernförderlicher gestaltet ist.

In der aktuellen Umsetzung in Graja haben wir die Bewertungshierarchie lediglich zweistufig angelegt. Unteraspektgruppen wie der in Abschnitt 5.1 dargestellte Aspekt „Readability“ unterhalb von „Maintainability“ sind derzeit noch nicht möglich. Eine diesbezügliche Erweiterung von Graja ist geplant.

Graja ermöglicht abstraktere Sichten auf das gesamte Bewertungsergebnis. So ist es möglich, eine Tabelle des Bewertungsergebnisses zu generieren, in der lediglich die gefundenen Mängel der Einreichung jeweils mit einer Überschrift aufgelistet sind (vgl. Abbildung 14). Es liegen noch keine Evaluierungen mit studentischen Befragungen vor. Gegenwärtig führen wir lediglich den gesunden Menschenverstand als Beleg dafür an, dass die neuen von Graja erzeugten Bewertungsübersichten dem einreichenden Studenten eine bessere, lernförderliche Orientierung bieten.

Category	**Aspect**	**Result**
Functional correctness	*Can calculate cost after calling setQuantity.*	wrong
	String comparison.	wrong
Maintainability	*All attributes in class CartItem should be private.*	wrong
	Variable naming conventions.	wrong
	Bad use of logic complement operator.	wrong
	Should avoid code redundancy.	delayed
Security	*Non-private attribute is a security risk.*	wrong
Efficiency	*Should use StringBuilder.*	delayed

Abbildung 14: Übersicht über die Mängel einer Einreichung, strukturiert nach Bewertungsaspekten

Category	Aspect	Src.	Result	Achieved	Max.
Functional correctness	*Can construct CartItem.*	JUnit	ok	0.75	0.75
	Can calculate cost.	JUnit	ok	1.00	1.00
	Should have correct toString method.	JUnit	ok	1.00	1.00
	Can calculate cost after calling setQuantity. For a cart item with (pricePerUnit=3.2, quantity=6) your method 'com.mymart.CartItem.getCost' must return a value close to 19.2. Observed: 9.2.	JUnit	wrong	0.00	0.50

	String comparison.
	Using '==' or '!=' to compare strings only works if intern version is used on both sides. Use the equals() method instead.
	○ Use equals() to compare strings instead of '==' or '!='
	`CartItem.java`
	`11 : if (Name=="") throw new IllegalArgumentException("name is empty");`
		PMD	wrong	0.00	0.50

				2.75	*3.75*
Maintainability	*All attributes in class CartItem should be private.* There are at least 3 non-private attributes in com.mymart.CartItem.	JUnit	wrong	0.00	0.25

	Variable naming conventions.
	Variables should be named conventionally.
	○ Variables should start with a lowercase character, 'Name' starts with uppercase character.
	`CartItem.java`
	`10 : public CartItem(String Name, int quantity, double pricePerUnit) {`
		PMD	wrong	0.00	0.50

	Class naming conventions.	PMD	ok	0.50	0.50
	Method naming conventions.	PMD	ok	0.50	0.50
	Fields (attributes) should be at the start of the class.	PMD	ok	0.50	0.50
	Boolean accessors should not be named get...	PMD	ok	0.25	0.25
	Collapsible if statements.	PMD	ok	0.50	0.50
	Unnecessary if-then-else.	PMD	ok	0.50	0.50

	Bad use of logic complement operator.
	Use opposite operator instead of negating the whole expression with a logic complement operator.
	○ Use opposite operator instead of the logic complement operator.
	`CartItem.java`
	`22 : if ( ! (quantity >= 1) ) {`
		PMD	wrong	0.00	0.50

	Avoid deeply nested if statements.	PMD	ok	0.25	0.25
	Should avoid code redundancy. Later human reviewers will grade your submission and credit points for no code redundancy.	Non-automated activity	delayed	0.00	1.00
	Comments needed in front of methods and classes.	PMD	ok	0.50	0.50
	Short class name.	PMD	ok	0.25	0.25
				3.75	*6.00*
Reliability	*Should monitor CartItem invariant in constructor.*	JUnit	ok	0.50	0.50
	Should monitor CartItem invariant in setter.	JUnit	ok	0.50	0.50
				1.00	*1.00*
Security	*Non-private attribute is a security risk.* There are at least 3 non-private attributes in com.mymart.CartItem.	JUnit	wrong	0.00	0.50
				0.00	*0.50*
Efficiency	*Should use StringBuilder.* Later human reviewers will grade your submission and credit points for String concatenation using StringBuilder.	Non-automated activity	delayed	0.00	0.50
				0.00	*0.50*

Abbildung 15: Bewertungsergebnis für die Beispielaufgabe, strukturiert nach Bewertungsaspekten

5.3 Vorschlag zur Erweiterung des ProFormaA-Aufgabenformats

Wir schlagen aufgrund der Vorüberlegungen dieses Beitrags vor, das ProFormA-Aufgabenformat um zwei neue Domänen-Entitätstypen zu erweitern:

- *GradingAspectGroup*: repräsentiert einen inneren Knoten der Bewertungshierarchie. Attribute: ein *title* sowie eine eigene *id*. Kindelemente: beliebig viele GradingAspectGroups oder GradingAspects.
- *GradingAspect*: repräsentiert ein Blatt der Bewertungshierarchie. Attribute: ein *title*, eine Referenz auf einen Test (*testref-id*), und eine eigene *id*.

Ein sinnvoller Ort für diese neuen Objekte ist das sowieso im ProFormA-Format vorgesehene Element <grading-hints>.

6 Zusammenfassung und Ausblick

Wir haben den Unterschied zwischen Testausführung auf der einen Seite und Interpretation des Testergebnisses im Sinne eines Bewertungsaspekts auf der anderen Seite erörtert. Die verschiedenen Sichten auf ein Testergebnis führten zum Vorschlag, im ProFormA-Aufgabenformat neben Domänenobjekten zur Spezifikation einer Testausführung zusätzliche Domänenobjekte vorzusehen. Zum einen schlagen wir vor, Testteilausführungen als separate Objekte zu spezifizieren und mit dem Objekt, welches die Ausführung des Testwerkzeugs spezifiziert, über eine *parentId* zu verknüpfen. Zum anderen schlagen wir vor, zusätzliche Domänenobjekte *GradingAspect* und *GradingAspectGroup* zur Spezifikation der Bewertungsaspekte vorzusehen.

Offen geblieben sind u. a. folgende Fragen

- Wie wird ein Ergebnis einer Teilausführung eines Tests dargestellt?
- Wie können quantitative Bewertungen an inneren Knoten vom Typ GradingAspectGroup aggregiert werden?
- Wie bewährt sich die vorgeschlagene neue Bewertungsstruktur in der Praxis?

Zum ersten Spiegelpunkt können wir anführen, dass in Graja ein Test-Teilausführungsergebnis derzeit als zusammengesetzter Datentyp mit den Elementen Erfolgsgrad (0-100%) und Kommentar gespeichert wird. Die Entwicklung eines grader-übergreifend verwendbaren Antwortformats mit Bewertungsergebnissen ist Gegenstand aktueller Forschung im eCULT-Projekt.

Zum zweiten Spiegelpunkt nutzt Graja in der Hierarchie der Bewertungsaspekte zurzeit einfach gewichtete Summen der Bewertungsergebnisse in Unteraspekten. Dies erleben wir als in bestimmten Bewertungssituationen zu unflexibel und soll zukünftig überarbeitet werden.

Zum dritten Spiegelpunkt halten wir fest, dass Graja mit der neuen Bewertungsstruktur erstmals im Sommersemester 2016 in einem Programmierkurs des zweiten Semesters (Bachelor Angewandte Informatik, ca. 80 Teilnehmerinnen und Teilnehmer) eingesetzt wird. Wir wollen sowohl qualitativ als auch quantitativ beobachten, wie die Studierenden mit dem neuen Feedback zurechtkommen.

Literaturverzeichnis

[1] Garmann, R.: E-Assessment mit Graja – ein Vergleich zu Anforderungen an Softwaretestwerkzeuge. In: Workshop „Automatische Bewertung von Programmier-aufgaben“, 6.11.2015, Wolfenbüttel, 2015.

[2] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, O.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks. eleed e-learning & education, 11(1), 2015.

[3] Werner, P.; Garmann, R.; Heine, F.; Kleiner, C.; Reiser, P.; De Vere Peratoner, I.; Grzanna, S.; Wübbelt, P.; Bott, O.: Grading mit Grappa – Ein Werkstattbericht. In: Workshop „Automatische Bewertung von Programmieraufgaben“, 6.11.2015, Wolfenbüttel, 2015.

[4] ISO/IEC 25010:2011: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.