aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

9 Der Grader JACK

Michael Striewe

Zusammenfassung
JACK ist ein webbasiertes Bewertungssystem für Übungs- und Prüfungsaufgaben, das seit 10 Jahren für die Bewertung von Programmieraufgaben eingesetzt wird. In diesem Kapitel werden Aufgabendesign, Feedbackmöglichkeiten und Einsatzerfahrungen vorgestellt.

9.1 Einleitung

Zur Unterstützung der Einführung in die Programmierung für Erstsemester wurde an der Universität Duisburg-Essen im Jahr 2006 ein System zur automatischen Bewertung von Programmieraufgaben entwickelt und aufgrund seiner Aufgabe als „Java Checker“ auf den Namen JACK getauft [SGB08]. Im Laufe mehrerer Jahre wurde JACK im Rahmen von Forschungsaktivitäten, Förderprojekten des BMBF und studentischen Projekten immer weiter erweitert, so dass das System als Framework für ganz verschiedene Arten von Aufgabenstellungen mit automatischem Feedback genutzt werden kann [SZG15]. Im Folgenden wird jedoch weitgehend nur das ursprüngliche Kerngeschäft von JACK, d. h. die automatische Bewertung von Programmieraufgaben in der Programmiersprache Java betrachtet.

JACK ist ein webbasiertes System, das grundsätzlich unabhängig von anderen Systemen wie Lernmanagementsystemen betrieben werden kann und das sich sowohl für den Übungs- als auch für den Prüfungsbetrieb eignet. An der Universität Duisburg-Essen wird eine Installation mit Anbindung an den LDAP-Server der Universität betrieben, so dass sich die Studierenden im Übungsbetrieb mit ihren üblichen Login-Daten anmelden können. Im Prüfungsbetrieb wird das System in einem anderen Login-Modus betrieben, in dem individuelle Einmalpassworte für die Prüfungen zum Einsatz kommen.

JACK ist nicht open-source, steht jedoch deutschen Hochschulen frei zur Verfügung, da die Entwicklung in den letzten Jahren primär über ein BMBF-Projekt

finanziert wurde. Weitere Informationen und Kontaktmöglichkeiten können der Projektwebseite[1] entnommen werden.

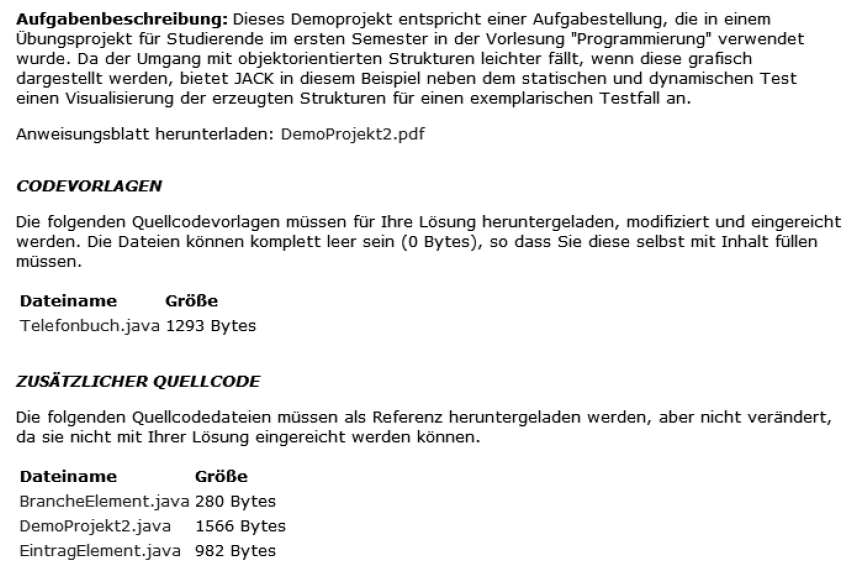
9.2 Aufgabendesign

Programmieraufgaben in JACK bestehen grundsätzlich aus einer oder mehreren Dateien Quellcode, die den Studierenden als Vorlage zur Verfügung gestellt werden und die nach der Bearbeitung auf den Server hochgeladen werden müssen. Dies erfolgt im Übungsbetrieb über den Browser, während für Prüfungen ein spezielles Plugin für die Entwicklungsumgebung Eclipse zur Verfügung steht, das die Vollständigkeit der Dateien und die richtige Einordnung in ein Projekt in der Entwicklungsumgebung sicherstellt. Auf diesem Wege können über JACK auch vollständige Eclipse-Projekte verteilt und eingesammelt werden.

Die Aufgaben können so gestaltet werden, dass nicht alle zur Verfügung gestellten Dateien bearbeitet werden müssen oder dürfen. Es kann auch nur eine Teilmenge der Dateien wieder eingesammelt werden. Vorlagendateien dürfen ferner leer sein, wenn diese von den Studierenden komplett selbst gefüllt werden sollen. Es ist jedoch nicht möglich, den Studierenden völlige Freiheit bei der Zahl und Benennung ihrer Dateien zu lassen, sofern nicht im Prüfungsmodus ganze Projekte eingesammelt werden. Ebenso wenig können Programmieraufgaben als Lückentexte im Browser angezeigt werden, in denen die Studierenden nur einzelne Zeilen ergänzen müssen. Derartige Aufgaben können in JACK über andere Aufgabentypen gestellt werden. Dann steht allerdings nicht der volle Umfang der automatischen Programmbewertung zur Verfügung.

Zu jeder Programmieraufgabe können vom Autor beliebig viele weitere versteckte Dateien definiert werden, die der Durchführung der Bewertung und der Erzeugung von Feedback dienen. Dazu werden so genannte Checker-Module für die Aufgabe konfiguriert, die jeweils einen Teil der gewünschten Funktionalität umsetzen. Für jede Aufgabe können prinzipiell beliebig viele Checker-Module konfiguriert werden, die für die Punktevergabe pro Aufgabe beliebig gewichtet werden können. Es ist auch möglich, Module ohne Gewichtung zu verwenden, so dass deren Feedback angezeigt wird, ohne Einfluss auf die Punktevergabe zu nehmen. Ebenso ist es möglich, Module mehrfach einzubinden, wenn ein Teil der von ihnen erzeugten Rückmeldung nicht in die Bewertung einfließen oder nur für Lehrende sichtbar sein soll. Lehrende haben unabhängig von der Konfiguration von Checker-Modulen immer die Möglichkeit, manuell Lösungen zu kommentieren und Punkte zu vergeben sowie automatisch erzeugte Meldungen zu unterdrücken.

1 http://www.s3.uni-duisburg-essen.de/jack/



Aufgabenbeschreibung: Dieses Demoprojekt entspricht einer Aufgabestellung, die in einem Übungsprojekt für Studierende im ersten Semester in der Vorlesung "Programmierung" verwendet wurde. Da der Umgang mit objektorientierten Strukturen leichter fällt, wenn diese grafisch dargestellt werden, bietet JACK in diesem Beispiel neben dem statischen und dynamischen Test einen Visualisierung der erzeugten Strukturen für einen exemplarischen Testfall an.

Anweisungsblatt herunterladen: DemoProjekt2.pdf

CODEVORLAGEN

Die folgenden Quellcodevorlagen müssen für Ihre Lösung heruntergeladen, modifiziert und eingereicht werden. Die Dateien können komplett leer sein (0 Bytes), so dass Sie diese selbst mit Inhalt füllen müssen.

Dateiname	Größe
Telefonbuch.java	1293 Bytes

ZUSÄTZLICHER QUELLCODE

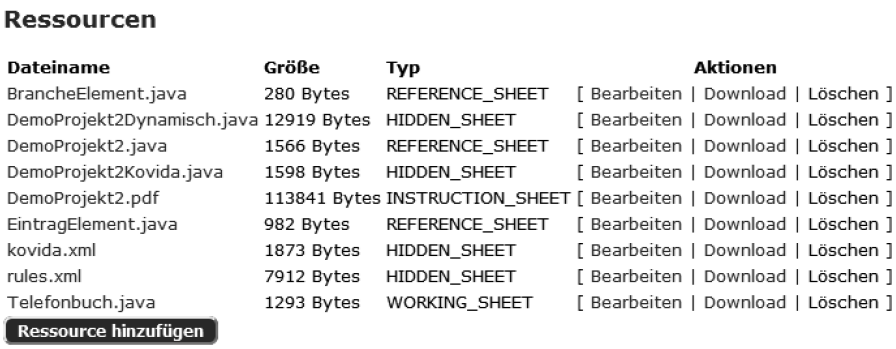
Die folgenden Quellcodedateien müssen als Referenz heruntergeladen werden, aber nicht verändert, da sie nicht mit Ihrer Lösung eingereicht werden können.

Dateiname	Größe
BrancheElement.java	280 Bytes
DemoProjekt2.java	1566 Bytes
EintragElement.java	982 Bytes

Abbildung 9.1: Beispielaufgabe aus Sicht der Studierenden. Die Aufgabe ist auf dem JACK-Demo-Server unter https://jack-demo.s3.uni-due.de/ frei verfügbar.

Als durchgehendes Beispiel zur Erläuterung der verschiedenen Möglichkeiten von JACK soll folgende Übungsaufgabe dienen: Inhalt der Aufgabenstellung ist das Erstellen einer Telefonbuchverwaltung, mit der Personen- und Brancheneinträge erstellt, geändert und gesucht werden können. Abbildung 9.1 zeigt die Aufgabenstellung mit Downloadmöglichkeiten aus Sicht der Studierenden. Sichtbar ist eine kurze Aufgabenbeschreibung sowie Links zum Herunterladen des Aufgabenblattes, der Codevorlagen und weiterer Codedateien, die für die Lösung nicht verändert werden sollen. Es ist den Studierenden überlassen, wie sie die Dateien auf ihrem Rechner ablegen und mit welchen Werkzeugen sie sie bearbeiten. Die Einreichung der Lösung erfolgt über eine weitere Seite (siehe Abbildung 9.2), über

Abbildung 9.2: Ansicht zum Hochladen einer Lösung für die Beispielaufgabe aus Abbildung 9.1.



Ressourcen

Dateiname	Größe	Typ	Aktionen
BrancheElement.java	280 Bytes	REFERENCE_SHEET	[Bearbeiten | Download | Löschen]
DemoProjekt2Dynamisch.java	12919 Bytes	HIDDEN_SHEET	[Bearbeiten | Download | Löschen]
DemoProjekt2.java	1566 Bytes	REFERENCE_SHEET	[Bearbeiten | Download | Löschen]
DemoProjekt2Kovida.java	1598 Bytes	HIDDEN_SHEET	[Bearbeiten | Download | Löschen]
DemoProjekt2.pdf	113841 Bytes	INSTRUCTION_SHEET	[Bearbeiten | Download | Löschen]
EintragElement.java	982 Bytes	REFERENCE_SHEET	[Bearbeiten | Download | Löschen]
kovida.xml	1873 Bytes	HIDDEN_SHEET	[Bearbeiten | Download | Löschen]
rules.xml	7912 Bytes	HIDDEN_SHEET	[Bearbeiten | Download | Löschen]
Telefonbuch.java	1293 Bytes	WORKING_SHEET	[Bearbeiten | Download | Löschen]

Ressource hinzufügen

Abbildung 9.3: Auflistung aller zur Beispielaufgabe aus Abbildung 9.1 gehörenden Dateien aus Sicht des Aufgabenautors.

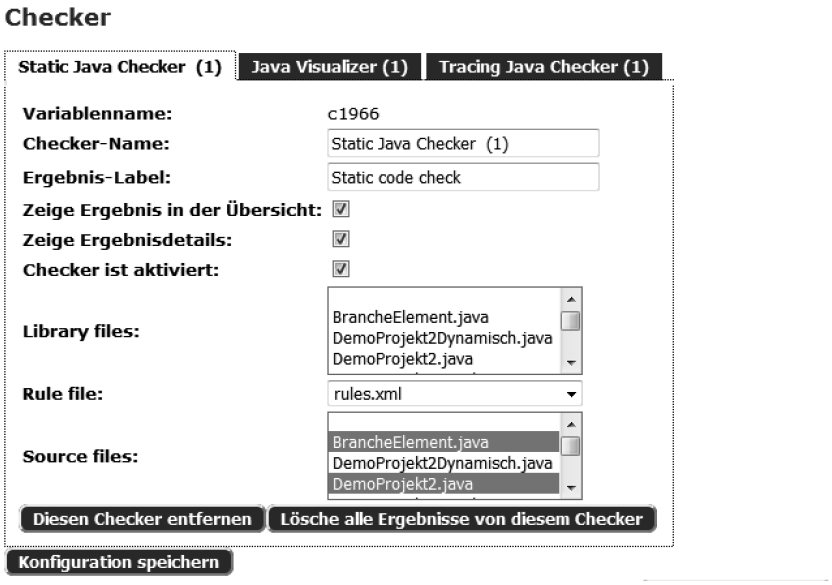


Abbildung 9.4: Konfiguration eines Checker-Moduls für die Beispielaufgabe aus Abbildung 9.1.

die nur die zur Bearbeitung vorgesehenen Codevorlagen wieder hochgeladen werden können. Aus Perspektive des Aufgabenautors besteht die Aufgabe neben der Grundkonfiguration (ohne Abbildung) insbesondere aus einer Liste von Dateien (siehe Abbildung 9.3) und der Konfiguration mehrerer Checker-Module (siehe Abbildung 9.4). Die Zuordnung von Dateien zu Checker-Modulen ist dabei beliebig und unabhängig von der Sichtbarkeit der Dateien für die Studierenden. Insbe-

sondere wird in dieser Aufgabe zwischen zu bearbeitendem Quellcode („working sheet“) und weiterem Quellcode („reference sheet“) unterschieden, aber alle diese Dateien werden allen Checker-Modulen zur Verfügung gestellt, da der Quellcode nur im Zusammenspiel aller Dateien kompilieren kann. Das Aufgabenblatt („instruction sheet“) ist dagegen für keines der Checker-Module relevant, während weitere Dateien für die Studierenden nicht sichtbar sind („hidden sheet“) und nur jeweils einem Modul zugewiesen werden, wie in den Abbildungen exemplarisch anhand der „rules.xml“ gezeigt.

9.3 Feedbackmöglichkeiten

Durch die Verwendung unabhängiger Checker-Module ist JACK sehr flexibel bezüglich der möglichen Rückmeldung zu den Lösungen einer Programmieraufgabe. Grundsätzlich müssen alle Checker-Module eine Punktzahl im Intervall von 0 bis 100 zurückgeben, wobei ein höherer Wert für eine bessere Lösung steht. Die Ergebnisse der einzelnen Checker-Module können gemäß einer Gewichtung oder eines komplexeren Ausdrucks miteinander verrechnet werden, um die Gesamtpunktzahl für die Lösung zu erhalten. Ferner können Checker-Module einen globalen Feedbacktext, eine Liste einzelner Fehlermeldungen sowie eine beliebige Menge an zusätzlichen Dateien zurückgeben, die während des Prüfvorgangs erstellt wurden. Im Folgenden werden die drei wichtigsten Checker-Module für Programmieraufgaben in Java vorgestellt.

9.3.1 Regelbasierte statische Prüfung

Die regelbasierte statische Prüfung von Programmcode zielt darauf ab, syntaktische und strukturelle Fehler und Schwächen einer Lösung zu entdecken und zu kommentieren. Sie besteht aus zwei Schritten: Im ersten Schritt wird die Einreichung kompiliert und gegebenenfalls werden die Fehlermeldungen des Compilers ohne weitere Verarbeitung an die Studierenden weitergereicht. Im zweiten Schritt erfolgt die Erzeugung eines Syntaxgraphen, auf den vom Aufgabenautor definierte Regeln angewandt werden können.

Jede dieser Regeln kann eine erwünschte oder unerwünschte Struktur definieren, die für den Fall ihres Auftretens bzw. Fehlens mit einem Feedback verknüpft wird. Zur Formulierung der Regeln kommt die Graphabfragesprache GReQL zum Einsatz, mit der beliebige Abfragen über Graphstrukturen definiert werden können. Die dazu verwendete Notation ähnelt Datenbankabfragen in SQL. Ein Editor

zur Unterstützung bei der Erstellung der Regeln ist über die o. g. Projektwebseite von JACK verfügbar.

Ein Beispiel für eine einfache Regel ist in Abbildung 9.5 angegeben. In dieser Regel wird in der `<query>` nach Methodendeklarationen gesucht, bei denen der Name mit einem Großbuchstaben beginnt, ohne dass es sich um die Deklaration eines Konstruktors handelt. Die Regel ist vom Typ `absence`, d. h. die beschriebene Struktur soll in einer korrekten Lösung nicht vorkommen. Kommt sie trotzdem vor, wird das in der Regel definierte Feedback ausgegeben. Eine solche Regel ist unabhängig von der konkreten Aufgabenstellung und kann daher für viele Aufgaben wiederverwendet werden. Regeln dieser Art können recht einfach für viele stilistische und strukturelle Aspekte eines Programms entwickelt werden. Ein weiteres Beispiel mit einer komplexeren Regel wird in Abbildung 9.6 gezeigt. In dieser Regel wird eine umfangreichere Struktur angegeben, nach der eine Methode namens `neu` den Konstruktor der Klasse `BrancheElement` aufrufen und dabei ihren eigenen Parameter an diesen Aufruf übergeben soll. Diese Struktur ist offensichtlich aufgabenspezifisch zu verwenden und müsste für andere Aufgaben angepasst werden. Die Regel ist vom Typ `presence`, so dass das Feedback ausgegeben wird, wenn die Lösung den beschriebenen Aufruf nicht enthält.

Das Erstellen derartiger Regeln erfordert vom Aufgabenautor eine gewisse Einarbeitung sowohl in die Notation der Abfragesprache als auch in den genauen Aufbau des Syntaxbaums von Java. Nach dieser Einarbeitung ermöglichen derartige Regeln aber eine hohe Flexibilität bei der Erstellung des Feedbacks. Für komplexe Fälle können dazu auch mehrere Abfragen in einer Regel kombiniert werden. Außerdem ist es möglich, Feedback auch für den positiven Fall zu geben, wenn die Studierenden nicht nur über einen Fehler, sondern auch über die Abwesenheit von Fehlern explizit informiert werden sollen. Neben der textuellen Rückmeldung

```
<rule type="absence" points="0">
    <query>from x : V{MethodDeclaration} with
        x.name=capitalizeFirst(x.name) and
        x.constructor="false"
        report 0 end
    </query>
    <feedback prefix="Hinweis (ohne Punktabzug)">
        Du verwendest Methodennamen, die mit einem Großbuchstaben
        beginnen. Das ist möglich, aber es entspricht nicht dem
        üblichen Programmierstil für Java.
    </feedback>
</rule>
```

Abbildung 9.5: Diese Regel zur statischen Codeprüfung sucht nach groß geschriebenen Methodennamen und gibt ein entsprechendes Feedback.

```
<rule type="presence" points="5">
    <query>from m : V{MethodDeclaration},
        c : V{ClassInstanceCreation},
        v : V{SingleVariableDeclaration},
        t : V{SimpleType}
        with
        m.name="neu" and
        m -->{MethodDeclarationParameters} v and
        m -->{Child}* c --> t and
        c -->&amp;{SimpleName} -->{Access} v and
        t.name="BrancheElement"
        report 0 end
    </query>
    <feedback>In der Methode "neu(String branche)" erfolgt kein
        Aufruf des Konstruktors von "BrancheElement", dem der
        übergebene String für den Namen weitergegeben wird.
    </feedback>
</rule>
```

Abbildung 9.6: Diese Regel zur statischen Codeprüfung überprüft, ob in einer bestimmten Methode ein vorgegebener Konstruktor aufgerufen wird.

kann jede Regel mit einem unterschiedlichen Punktgewicht versehen werden, um ihren Einfluss auf die Gesamtbewertung zu steuern.

9.3.2 Testfallbasierte dynamische Prüfung

Die dynamische Prüfung zielt darauf ab, die funktionale Korrektheit einer Lösung festzustellen und führt dazu Testfälle aus. Jeder Testfall muss vom Aufgabenautor durch eine Methode in einem Testtreiber definiert werden, wie dies auch bei klassischen Unit-Tests mit Werkzeugen wie JUnit der Fall ist. JACK bietet im Vergleich zu derartigen Werkzeugen jedoch einige abweichende Funktionen.

Erstens können in JACK Testfälle nicht nur bei einem Fehlschlag oder einem Erfolg eine definierte Fehlermeldung zurückgeben, sondern im Verlauf ihrer Ausführung nach Bedarf beliebig viele Meldungen erzeugen. Ebenso können Testfälle je nach Verlauf oder Ergebnis unterschiedliche Auswirkungen auf die Punktzahl haben und es können beliebige Abhängigkeiten zwischen den Testfällen erzeugt werden. Die Entscheidung, welche dieser Möglichkeiten in welchem Umfang genutzt werden, ist Sache der Aufgabenautoren und bedarf einer sorgfältigen Planung, da die Testergebnisse sonst schlecht nachvollziehbar werden können. Zweitens kann JACK während der Ausführung der Testfälle alle Programmschritte der geprüften Lösung aufzeichnen und dabei vom Testtreiber gesteuert werden. So ist

es möglich, dass in einem Testfall die Initialisierung eines Objektes geprüft wird und dabei alle Schritte aufgezeichnet werden, während in einem zweiten Testfall mehrere dieser Objekte ohne Aufzeichnung der Schritte erzeugt werden und erst anschließend für Operationen auf diesen Objekten die Aufzeichnung beginnt. Die Aufzeichnung gibt immer die jeweilige Codezeile und die Variablenwerte vor Ausführung derselben an. Ein kurzes Beispiel für eine solche Aufzeichnung ist in Abbildung 9.7 zu sehen. Die erste und letzte Spalte liefern Informationen zur Codezeile, während die übrigen Spalten Variablenwerte enthalten. Im Beispiel in der Abbildung gibt es ein Objekt mit der internen ID 160, dessen Feld `Ort` im Laufe des Aufrufs einen neuen Wert zugewiesen bekommt. Bei vielen Aufgaben können diese Aufzeichnungen Studierenden und Tutoren helfen, die Ursache einer fehlerhaften Programmausgabe im Detail nachzuvollziehen. Auch eine automatisierte Analyse der Aufzeichnungen, durch die weiteres Feedback generiert werden kann, ist bereits möglich, wird allerdings derzeit nur experimentell eingesetzt.

Unabhängig von den vom Aufgabenautor gewählten Einstellungen ist JACK immer in der Lage, Exceptions abzufangen und so zu behandeln, dass erstens ein erklärender Kommentar zu den gängigsten Exceptions erzeugt wird und zweitens die Ausführung weiterer Testfälle nicht gestört wird. Dasselbe gilt für das Auftreten von Endlosschleifen, die JACK nach einem Timeout pro Testfall beendet und automatisch mit dem nächsten Testfall fortfährt.

Ein Beispiel für einen Testtreiber ist in Abbildung 9.8 angegeben. Die Abbildung zeigt einen Ausschnitt von zwei Testmethoden aus einem größeren Testtreiber. Der erste Test führt einen Konstruktoraufruf auf der vom Lernenden zu implementierenden Klasse `Telefonbuch` durch und überprüft das Ergebnis durch zwei Vergleiche. In beiden Fällen wird bei einem Fehler eine Meldung ausgegeben und es werden keine Punkte gutgeschrieben. Ansonsten werden vier Punkte vergeben. Im zweiten Testfall wird zunächst überprüft, ob der erste Testfall erfolgreich war. Ist dies nicht der Fall, wird der zweite Testfall übersprungen und eine entsprechende Warnung ausgegeben. Anderenfalls wird als erstes die Aufzeichnung der Programmschritte ausgeschaltet, um denselben Aufruf wie im ersten Test durchzuführen. Anschließend wird die Aufzeichnung wieder eingeschaltet und es wird

Aufruf des Konstruktors der Klasse Telefonbuch durch JACK

Klasse und Zeile	Variablenwerte				Nächste ausgeführte Codezeile
	160.KopfBranche	160.KopfEintrag	160.Ort	Ort	
Telefonbuch:6	**null**	**null**	**null**	"Syke"	public Telefonbuch(String Ort){
Telefonbuch:7	**null**	**null**	**null**	"Syke"	this.Ort = Ort;
Telefonbuch:8	**null**	**null**	**"Syke"**	"Syke"	}

Abbildung 9.7: Beispielhafte Ausgabe der aufgezeichneten Programmschritte bei einem einfachen Konstruktoraufruf.

```
public class DemoProjekt2Dynamisch {

  private int[] punkte = new int[18];

  @Test(name = "Test 1")
  public void test1() {
    Telefonbuch t1 = new Telefonbuch("Syke");

    if (t1.ort() == null) {
      TracingFramework.printError(/* . . . */);
    } else if (!(t1.ort().equals("Syke"))) {
      TracingFramework.printError(/* . . . */);
    } else punkte[0] = 4;
  }

  @Test(name = "Test 2")
  public void test2() {
   if (punkte[0] == 0)
      TracingFramework.printWarning(/* . . . */);
   else {
      TracingFramework.switchOffTracing();
      Telefonbuch t1 = new Telefonbuch("Syke");
      TracingFramework.switchOnTracing();

      t1.neu("Mueller",   7978);

      if (t1.ersterEintrag() == null){
        TracingFramework.printError(/* . . . */);
      } else if (!t1.ersterEintrag.name().equals("Mueller")){
        TracingFramework.printError(/* . . . */);
        punkte[1] = 1;
      } else if (t1.ersterEintrag.nummer() != 7978){
        TracingFramework.printError(/* . . . */);
        punkte[1] = 1;
      } else punkte[1] = 3;
    }
  }

  // . . . (weitere Testmethoden)

  public int getResult() {
    // . . . (Punkte summieren und zurückgeben)
  }
}
```

Abbildung 9.8: Ausschnitt aus einem Testtreiber für den dynamischen Test einer Klasse. Die Inhalte der über `TracingFramework.printError()` ausgegebenen Fehlermeldungen sind zur besseren Lesbarkeit ausgeblendet.

eine Methoden aufgerufen. Auch hier erfolgen zur Kontrolle mehrere Vergleiche, wobei diesmal in einigen Fällen Teilpunkte vergeben werden. Alle Punkte wer-

den in ein Array geschrieben und am Ende über die Methode `getResult()` aufsummiert und zurückgegeben.

9.3.3 Testfallbasierte Programmvisualisierung

Gerade in der objektorientierten Programmierung ist ein rein textuelles Feedback in Form von Fehlermeldungen und aufgezeichneten Programmschritten schwierig zu verstehen, wenn Lösungen und die von ihnen erzeugten Datenstrukturen größer werden. JACK bietet daher zusätzlich die Möglichkeit an, grafisches Feedback in Form von Objektdiagrammen zu erzeugen. Dazu muss vom Aufgabenautor ähnlich zu den dynamischen Tests ein Testfall definiert werden, für den die Visualisierung erzeugt werden soll. In einer Konfigurationsdatei können dann Zeilennummern aus diesem Testtreiber angegeben werden, in die ein Breakpoint gesetzt wird, um einen Snapshot der Datenstruktur zu erzeugen. Es ist zwar technisch genauso möglich, die Breakpoints direkt in der eingereichten Lösung zu setzen, aber da deren Struktur zum Zeitpunkt der Konfiguration der Aufgabe unbekannt ist, sind feste Zeilennummern in diesem Zusammenhang nicht sinnvoll. Bei der Erzeugung des Snapshots berücksichtigt JACK auch Objekte, die im Programmverlauf vor dem Breakpoint erzeugt und später bereits wieder gelöscht wurden. Diese Objekte werden vom Java Garbage Collector aus dem Speicher entfernt, aber von JACK im Objektdiagramm noch mit einem roten Rand angezeigt. Im Beispiel in Abbildung 9.9 sind mehrere solcher Objekte enthalten. In Verbindung mit dem zur Visualisierung gehörenden Testfall (hier nicht gezeigt) lässt sich erkennen, dass offenbar eine Löschoperation fehlerhaft implementiert wurde, da es mehrere „hängende“ Objekte gab, die vom Garbage Collector entfernt wurden, während die verbleibenden Einträge zyklisch aufeinander verweisen.

9.3.4 Weitere Feedbackmöglichkeiten

Basierend auf der Aufzeichnung aller Programmschritte während der Ausführung der Testfälle ist JACK in der Lage, die Abdeckung des studentischen Codes durch diese Testfälle zu berechnen bzw. im Code grafisch darzustellen. Auf diese Weise können Programmteile hervorgehoben werden, die in keinem der Testfälle erreicht werden. Die Interpretation dieser Information erfordert allerdings ein fortgeschrittenes Verständnis von Testverfahren von den Studierenden, da nicht erreichte Programmzeilen sowohl auf fehlerhafte Programmierung („dead code“) als auch auf unzureichende Testfälle zurückzuführen sein können. Zudem kann

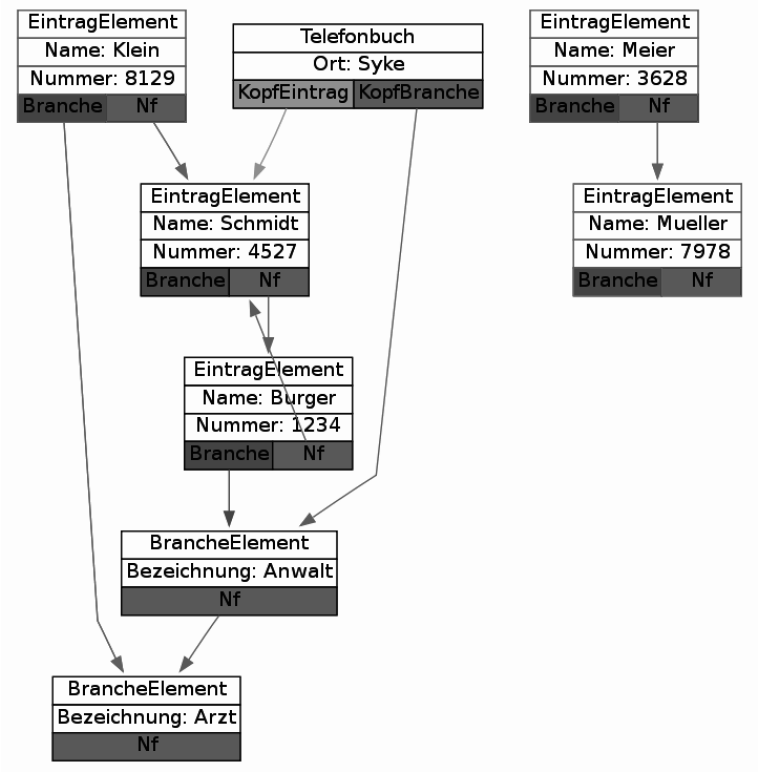
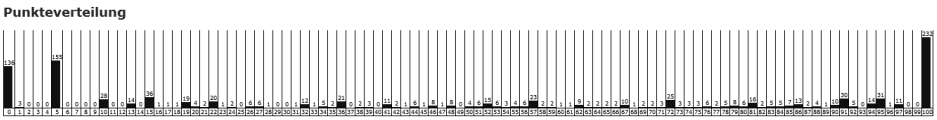


Abbildung 9.9: Beispielhafte Ausgabe eines Objektdiagramms. Objekte, die zum Zeitpunkt der Erzeugung des Snapshots bereits vom Garbage Collector entfernt worden sind, werden mit einem roten Rand dargestellt.

unerreichter Code grundsätzlich überflüssig sein, oder aufgrund eines bestimmten Fehlers nicht erreicht werden und damit ursächlich für einen fehlgeschlagenen Testfall sein. Der Grad der Testabdeckung wird daher auch nicht zur Berechnung der Punktzahl herangezogen, sondern als reine Zusatzinformation verwendet.

Als weitere Feedbackmöglichkeiten, die nur den Aufgabenautoren angezeigt werden, kann JACK auch Softwaremetriken berechnen und Aussagen zur Laufzeit einer Lösung machen. Grundsätzlich könnten diese Informationen zwar auch den Lernenden zur Verfügung gestellt werden, doch bietet JACK derzeit keine Möglichkeit, diese Informationen sinnvoll in einen Kontext einzubetten und zu interpretieren. Metriken wie die zyklomatische Komplexität einer Lösung sind daher momentan eher für Lehrende interessant, um einzelne auffällige Lösungen schneller identifizieren zu können, oder generelle Aussagen zum Vergleich von



Fehlermeldungsstatistik

DYNAMISCHER TEST 1

Fehlermeldung	Häufigkeit
Fehler in Test 15: Am haeufigsten bestellte Getraenk: Die Methode 'm.meisteGetraenk()' liefert das falsche Ergebnis: Richtig ist 14 und nicht 0	477
Fehler in Test 12: Tisch mit der meisten Anzahl an Artikeln : Die Methode 'm.maxArtikel()' liefert das falsche Ergebnis: Richtig ist 11 und nicht 0	435
Fehler in Test 13: Anzahl der freien Plaetze (belegte und nicht belegte Tische). : Die Methode 'm.freiePlaetze()' liefert das falsche Ergebnis: Richtig ist 8 und nicht 0	392

Abbildung 9.10: Auszug aus der Fehlerstatistik einer Aufgabe. Es ist zu erkennen, dass nahezu alle möglichen Punktzahlen von 0 bis 100 Punkten erreicht werden und das Bewertungsschema dieser Aufgabe somit offenbar sehr feingranular ist. Ferner werden unter dem Diagramm die am häufigsten fehlgeschlagenen Testfälle aufgeführt.

Aufgaben untereinander zu gewinnen [SG13]. Als zusammenfassendes Feedback für Lehrende bietet JACK zudem einige Statistiken an, die Auskunft über die erreichten Punktzahlen sowie die häufigsten Fehler geben. Auf diese Weise kann etwa die Granularität des Bewertungsschemas überprüft werden, um die Effektivität gegebenenfalls steigern zu können [Fal+14].

9.4 Technischer Hintergrund

JACK basiert auf einer Master-Worker-Architektur, in der der Master die Entgegennahme, Bereitstellung und Verwaltung von Aufgaben und Lösungen übernimmt, während die Worker die Bewertung und Erzeugung von Feedback durchführen. Der Master ist als EJB-Anwendung auf einem JBOSS Application Server realisiert und verwendet eine PostGreSQL-Datenbank zur Datenhaltung. Die Worker sind eine OSGi-Anwendung, in die die einzelnen Checker-Module als Plugins eingeführt werden können. Es ist insbesondere nicht notwendig, dass alle Worker-Instanzen mit denselben Plugins ausgestattet werden, so dass stattdessen dedizierte Worker für bestimmte Aufgaben konfiguriert werden können, die auf einer entsprechend angepassten Hardware laufen.

Master und Worker kommunizieren miteinander über Webservices und eine vom Master verwaltete Queue für Jobs. Zu jeder eingereichten Lösung wird für jedes in der Aufgabe konfigurierte Checker-Modul ein Job erzeugt, der von einem Worker abgeholt und bearbeitet werden kann. Die Kommunikation wird dabei

ausschließlich von den Workern initialisiert, die Jobs aus der Queue entnehmen bzw. Ergebnisse zurückmelden. Die Worker können daher aus Sicherheitsgründen in Netzwerksegmente abgelegt werden, die von außen nicht erreichbar sind.

Die Architektur hat sich in den vergangenen Jahren als sehr leistungsfähig erwiesen und ist insbesondere gut skalierbar, um beispielsweise durch das Hinzufügen weiterer Backend-Instanzen auf hohe Last zu reagieren. Das Konzept ist dabei nicht auf Programmieraufgaben beschränkt, sondern für alle Aufgabentypen in JACK nützlich [Str16].

Auch wenn JACK vollständig in Java entwickelt ist, sind Programmieraufgaben nicht grundsätzlich auf diese Sprache beschränkt. Über die Einbindung geeigneter externer Werkzeuge in den Checker-Modulen können auch weitere Programmiersprachen geprüft werden. Als prototypische Module, die im Rahmen studentischer Arbeiten entstanden sind, existieren derzeit Erweiterungen für Python [Loh15] und die .NET-Sprachfamilie [D'A15]. Eine Erweiterung für C++ wurde an der Universität Heidelberg entwickelt [HWP13].

9.5 Einsatzerfahrung

An der Universität Duisburg-Essen kommt JACK am Campus Essen seit dem Wintersemester 2006/2007 zur Bewertung von Programmieraufgaben in Java zum Einsatz. Bis einschließlich Wintersemester 2015/16 wurden von dem System gut 150.000 Lösungen zu Übungs- und Testataufgaben bewertet. Es existiert ein Aufgabenpool von etwa 70 Aufgaben, die zum Teil als Prüfungsaufgaben in mehreren Varianten vorliegen.

Der Einsatz wurde in mehreren Semestern durch eine umfangreiche Evaluation begleitet, um Einschätzungen der Studierenden über den Nutzen des Systems zu erhalten. Die wichtigste Frage dabei ist, ob JACK von den Studierenden als hilfreich für Übungen oder Testate betrachtet wird. Die Ergebnisse der Befragung sind in Tabelle 9.1 zusammengefasst. Es ist ersichtlich, dass JACK von den Studierenden mit 75% bzw. 68% vollständiger oder überwiegender Zustimmung als nützlich im Übungs- und Testatbetrieb betrachtet wird. Diese grundsätzliche Einstellung spiegelt sich auch tatsächlich im Nutzerverhalten wieder, indem im Übungsbetrieb mehrere Einreichungen durchgeführt werden, um schrittweise die durch das Feedback benannten Fehler auszubessern [SG11b].

Spezielle statistische Auswertungen für die verschiedenen Feedbackmöglichkeiten wurden in einzelnen Semestern durchgeführt. So waren im Wintersemester 2012/13 55% aller Feedbacknachrichten auf fehlgeschlagene Testfälle zurückzuführen, 23% entfielen auf Compilerfehler, 13% auf Exceptions, 6% auf stilistische

		++	+	o	-	- -	#Antworten
Einstellung bzgl. der Aussage „JACK ist im *Übungsmodus* insgesamt betrachtet nützlich.“	2008/2009	34%	44%	15%	7%	0%	61
	2009/2010	53%	29%	9%	5%	4%	77
	2010/2011	26%	43%	25%	5%	2%	61
	2011/2012	35%	27%	27%	10%	2%	63
	2012/2013	Keine Daten verfügbar					
	2013/2014	49%	33%	13%	5%	0%	61
	Schnitt	**40%**	**35%**	**17%**	**6%**	**2%**	
Einstellung bzgl. der Aussage „JACK ist im *Testatbetrieb* insgesamt betrachtet nützlich.“	2008/2009	30%	36%	23%	8%	3%	61
	2009/2010	48%	16%	12%	13%	11%	77
	2010/2011	34%	38%	16%	10%	2%	61
	2011/2012	30%	35%	19%	10%	6%	63
	2012/2013	Keine Daten verfügbar					
	2013/2014	36%	38%	13%	11%	4%	56
	Schnitt	**36%**	**32%**	**16%**	**11%**	**6%**	

Tabelle 9.1: Ergebnisse mehrerer Befragungen der Studierenden zu ihrer grundsätzlichen Einstellung zu JACK in den beiden hauptsächlichen Einsatzszenarien. Es steht ++ für die volle Zustimmung zur Aussage und - - für die vollständige Ablehnung.

Fehler, 2% auf strukturelle Fehler und 1% auf Timeouts für mutmaßliche Endlosschleifen. Die Kombination verschiedener Feedbackmöglichkeiten ist also auch tatsächlich effektiv bei der Erstellung vielfältiger Rückmeldungen.

9.6 Zusammenfassung

Nach inzwischen 10 Jahren Einsatzzeit zeigt sich JACK als stabiles System mit vielfältigen Feedbackmöglichkeiten, das von den Studierenden überwiegend als hilfreich erachtet wird. Der Aufwand für die Lehrenden bei der Erstellung und Konfiguration neuer Aufgaben liegt im Durchschnitt höher als bei anderen Systemen, aber dafür sind auch vielfältigere und detailliertere Rückmeldungen möglich. Obwohl ursprünglich nur für Programmieraufgaben in Java entwickelt, ist die Architektur von JACK so generisch, dass Checker-Module für weitere Programmiersprachen einfach ergänzt werden können.

Literatur für dieses Kapitel

[D'A15] Mario D'Amico. „Konzeption universeller .NET-Prüfkomponenten für das E-Assessment-System JACK“. In: *Workshop „Automatische*

Bewertung von Programmieraufgaben“ (ABP 2015). Bd. 1496. CEUR Workshop Proceedings. 2015.

[Fal+14] Nickolas Falkner u. a. „Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units“. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. ACM, 2014, S. 9–14. DOI: 10.1145/2538862.2538896.

[HWP13] Tom-Michael Hesse, Axel Wagner und Barbara Paech. „Automated assessment of C++ programming exercises with unit tests“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Loh15] Enno Lohmann. „Erweiterung eines E-Assessment-Systems um eine Prüfkomponente für die Programmiersprache Python“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[SG11b] Michael Striewe und Michael Goedicke. „Studentische Interaktion mit automatischen Prüfungssystemen“. In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 209–220.

[SG13] Michael Striewe und Michael Goedicke. „Analyse von Programmieraufgaben durch Softwareproduktmetriken“. In: *SEUH*. 2013, S. 59–68.

[SGB08] Michael Striewe, Michael Goedicke und Moritz Balz. *Computer Aided Assessments and Programming Exercises with JACK*. Techn. Ber. 28. ICB, University of Duisburg-Essen, 2008.

[Str16] Michael Striewe. „An architecture for modular grading and feedback generation for complex exercises“. In: *Science of Computer Programming* 129 (2016), S. 35–47. DOI: 10.1016/j.scico.2016.02.009.

[SZG15] Michael Striewe, Björn Zurmaar und Michael Goedicke. „Evolution of the E-Assessment Framework JACK“. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015.* Bd. 1337. CEUR Workshop Proceedings. 2015, S. 118–120.