paluno
The Ruhr Institute for Software Technology
Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen

Seminararbeit

Prüfung von Übungsaufgaben durch Graphabfragen

Michael Krane
2233018

Mülheim an der Ruhr
08.09.2014

Betreuung: Dipl. Inf. Michael Striewe
Studiengang: Angewandte Informatik – Systems Engineering

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mülheim an der Ruhr, am 08.09.2014

Zusammenfassung

Im Rahmen dieser Seminararbeit wird das an der Uni Duisburg-Essen entwickelte JACK System untersucht.

Bei JACK handelt es sich um ein e-Assessment System, das in verschiedenen Vorlesungen für Test- und Korrekturzwecke eingesetzt wird.

In dieser Seminararbeit werden für den Einsatz von JACK in der Vorlesung „Programmierung" Regeln zur statischen Codeanalyse entwickelt und vorgestellt. Darüber hinaus wird eine Erweiterung für das JACK-System beschrieben.

Ziel dieser Arbeit ist dabei nicht nur die Entwicklung von allgemein einsetzbaren Regeln, sondern auch die Diskussion der Frage, inwiefern es sich lohnt, solche Regeln weiterzuentwickeln.

Abbildungsverzeichnis

Inhaltsverzeichnis

1 Einleitung und Motivation

Im Zuge der Vorlesung „Programmierung" an der Universität Duisburg-Essen wird JACK eingesetzt. JACK überprüft anhand von statischen und dynamischen Checks Übungsaufgaben und Testate.

Ziel dieser Seminararbeit ist es, die statischen Checks zu erweitern und zu ergänzen, um dadurch das Feedback an die Studierenden und Lehrenden über die abgegebenen Programmieraufgaben zu verbessern. Aktuell existiert nur eine sehr kleine Datenbank von statischen Checks, die hauptsächlich sicherstellen, dass die Struktur der Lösungen korrekt ist.

JACK benutzt für die statische Überprüfung der Programmieraufgaben einen Abstract-Syntax Graph (ASG). Mit diesem Graphen und dem jGralab-Framework ist es möglich, die Struktur der zu überprüfenden Programme zu analysieren und dadurch gezielt Hinweise oder Fehlernachrichten zu generieren.

Diese Hinweise bzw. Fehlermeldungen, die durch den Java-Compiler nicht erkannt werden können und daher nicht rückgemeldet werden, unterstützen Lernende und Lehrende:

Zum einen unterstützen sie die Studierenden, die selbständig mit JACK arbeiten und durch detaillierte Rückmeldungen Fehler selber und schneller finden.

Zum anderen unterstützen sie die Lehrenden in der Programmierberatung, da diese direkt auf die von JACK erzeugten Fehlermeldungen eingehen können und sich nicht erst in die von den Studierenden erstellten Lösungen einlesen müssen.

Außerdem ist es durch die erweiterte, verbesserte Analyse des Codes möglich festzustellen, ob ein Studierender evtl. vorgegebene Einschränkungen in der Aufgabenstellung eingehalten hat oder nicht, z. B. die Einschränkung, ob eine bestimmte Methode rekursiv implementiert werden musste oder nur eine bestimmte Art von Schleifen genutzt werden durfte.

2 Theoretische Grundlagen

2.1 JACK

JACK ist ein von der Arbeitsgruppe "Specifications of Software Systems" entwickeltes webbasiertes e-Assessment System, das seit dem WS 2006/2007 an der Universität Duisburg-Essen eingesetzt wird [2]. Aktuell wird das System in mehreren Vorlesungen zur Unterstützung des Lehrbetriebes genutzt, ursprünglich wurde es für die Erstsemester-Vorlesung "Programmierung" entwickelt. Mit JACK ist es möglich, abgegebene Übungs- und Testat-Aufgaben der Studierenden automatisch zu kontrollieren.

In [6] wird der Aufbau von JACK beschrieben:

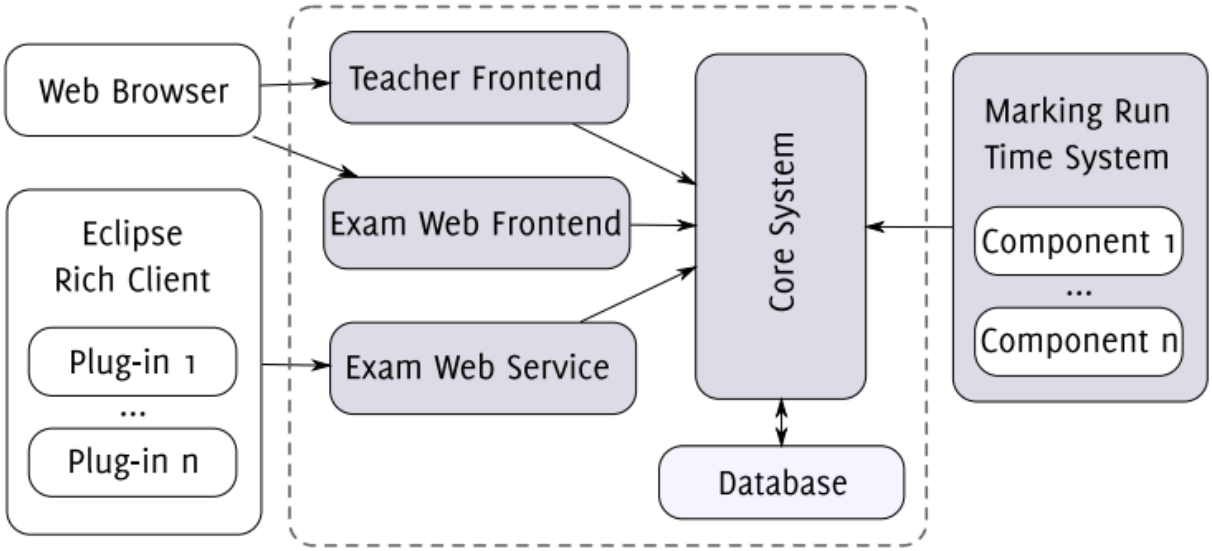


Abbildung 1: Übersicht der Architektur von JACK. Siehe [6]

Das JACK-System besteht dabei aus 3 Komponenten: einem Core-System, einem oder mehrere Clients und einem oder mehreren Checkern.

Das Core-System ist für die Verwaltung von Benutzern, Aufgaben und Ergebnissen zuständig und besteht hauptsächlich aus einem DBMS, das für die Verwaltung der vorhandenen Übungen und den von den Studierende abgegeben Lösungen zuständig ist. Die verschiedenen Clients sind über verschiedene Schnittstellen mit JACK verbunden und ermöglichen es, das System zu administrieren, neue Aufgaben zu erstellen oder Lösungen in JACK hochzuladen.

Durch den modularen Aufbau von JACK ist es möglich, über mehrere Front-Ends auf dasselbe Core-System zuzugreifen. Dadurch können über verschiedene Methoden Lösungen abgegeben werden, z. B. über ein Web-Interface für normale Übungsaufgaben oder direkt über eine spezielle Version von Eclipse für Testate oder Prüfungen.

Die Checker bilden das Backend von JACK und sind für die Auswertung und Bewertung der Aufgaben zuständig. Die Modularität erlaubt es, nicht nur Programmieraufgaben zu überprüfen, sondern auch Multiple-Choice Fragen auszuwerten oder mit mehreren Checkern verschiedene Aspekte einer Aufgabe zu testen.

Dadurch ist es möglich, dass zuerst die Struktur des abgegeben Programms (statischer Checker) und danach das Verhalten des Programms (dynamischer Checker) getestet wird.

Diese Seminararbeit betrachtet im Folgenden nur den statischen Checker für Programmieraufgaben aus der Vorlesung „Programmierung".

2.2 Statische Checks

Eine Möglichkeit, um Code zu analysieren und zu bewerten, ist die rein syntaktische Analyse. Bei dieser spielt die Semantik des Codes, also das, was das Programm bewirkt, nur eine untergeordnete Rolle. Es ist aber trotzdem möglich, anhand der Code-Struktur Rückschlüsse auf Eigenschaften und Qualität des Programms zu ziehen.

Abstract-Syntax-Trees (ASTs) machen es möglich, die Struktur eines Programmes darzustellen. Dabei wird der Code in eine hierarchische Struktur eingefügt, die es möglich macht, die Programmstruktur nachzuvollziehen.

Aufgrund der Baumstruktur gehen allerdings wertvolle Informationen der Programmstruktur verloren, da zum Beispiel Referenzen nicht aufgelöst werden können. Dieser Nachteil kann dadurch ausgeglichen werden, dass dem AST weitere Kanten hinzugefügt werden, um die fehlenden Referenzen aufzulösen. Dadurch wird aus dem Syntax-Tree ein Syntax-Graph.

In der Abbildung 3 wurden zum Beispiel die Kanten e18 und e19 hinzugefügt um darzustellen, dass in Zeile 5 sowohl auf ein Feld der Klasse als auch auf einen Parameter zugegriffen wird.

Der GReQL-Checker von JACK arbeitet nach der oben beschriebenen Technik und nutzt als grundlegende Datenstruktur TGraphen.

```
1  public class Example {
2      int i;
3
4      void setInt(int i) {
5          this.i = i;
6      }
7  }
```

Abbildung 2: Java Source-Code

Das folgende Beispiel zeigt den aus dem Beispielcode in Abbildung 2 erzeugten Syntaxgraphen.

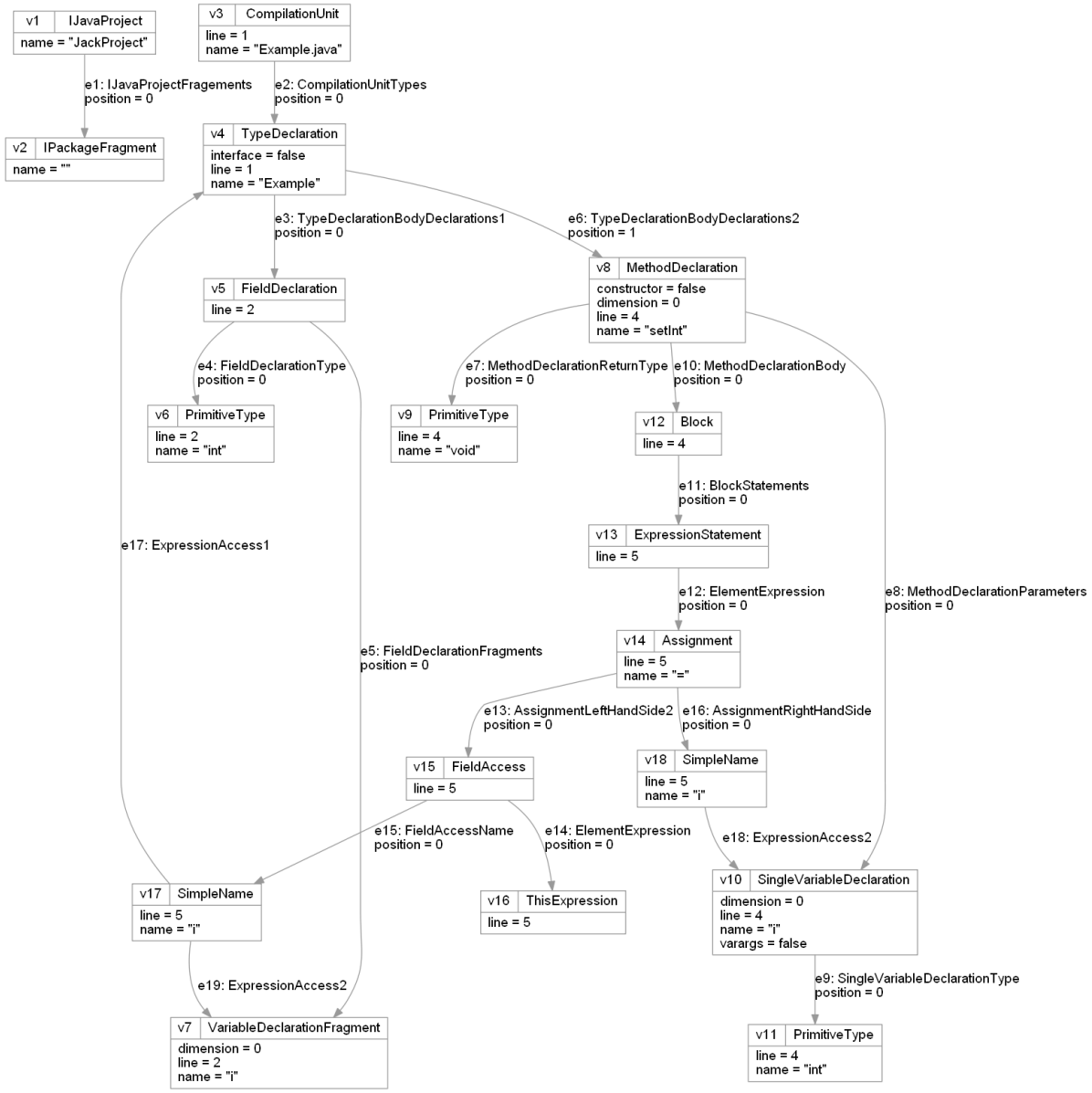


Abbildung 3: Syntax-Graph der Abbildung 2

2.3 TGraphen

Der GReQL-Checker benutzt als Datentyp TGraphen. Dabei sind TGraphen spezielle gerichtete Graphen, die folgende Eigenschaften [4] erfüllen müssen: Sie sind

1. gerichtet - Alle Knoten sind über gerichtete Kanten verbunden. Dadurch ist es immer möglich festzustellen, welcher Knoten der Startknoten und welcher Knoten der Endknoten einer Kante ist.

2. typisiert - Jedes Element im Graphen ist typisiert, d.h. jedem Knoten und jeder Kante ist eine Klasse zugeordnet. Hierdurch lassen sich die Klassen in eine Vererbungshierarchie einsortieren.

3. attribuiert - Jedes Element besitzt, abhängig von der Vererbungshierarchie, verschiedene Attribute-Werte-Paare. Dadurch wird es möglich, zusätzliche Informationen im Knoten zu speichern.

4. geordnet - In jeder Menge von Knoten oder Kanten existiert eine Ordnung, d.h. es ist möglich, jedem Element eine eindeutige Bezeichnung zuzuweisen.

2.4 GReQL

Der statische Checker nutzt zur Überprüfung des Graphen die Graphanfragesprache GReQL 2 (Graph Repository Query Language 2). Diese wurde 2006 im Zuge einer Diplomarbeit als Grundlage für die Graphenbibliothek jGraLab entwickelt [4].

GReQL 2 ist eine SQL ähnliche Anfragesprache, die es ermöglicht, Abfragen und reguläre Pfadausdrücke auf TGraphen auszuwerten. Außerdem ist es möglich, auf die in 2.3 beschriebenen Attribute der Graphen zuzugreifen.

Die vollständige Syntax von GReQL 2 ist in [4] zu finden. In den letzten acht Jahren wurde GReQL 2 aktiv weiter entwickelt und verändert, deshalb gibt es zusätzlich eine GReQL-Referenzkarte [1].

Die im Zuge dieser Seminararbeit entwickelten Anfragen werden alle durch FWR-Ausdrücke beschrieben. Ähnlich wie die SFW-Konstrukte [4] von SQL bestehen auch FWR-Ausdrücke aus drei Teilen:

1. FROM-Klausel

Hier werden die Variablen deklariert, die in der Query genutzt werden. Mit V(TypName) werden Knoten und mit E(TypName) werden Kanten gekenn-

zeichnet. Hierbei ist zu beachten, dass bei der Auswertung die Variable nacheinander alle Werte vom gegebenen Typ annimmt.

2. WITH-Klausel

In der optionalen WITH-Klausel werden Anforderungen an die in der FROM-Klausel definierten Variablen gestellt. Es besteht zum einen die Möglichkeit Kanten und Knoten anhand ihrer Attribute zu filtern und zum anderen, mittels regulären Pfadbeschreibungen Pfade im TGraphen zu beschreiben.

3. REPORT-Klausel

In der REPORT-Klausel wird das Ergebnis der Query beschrieben. Als Rückgabewert ist es möglich, statische Werte oder Attribute der vorher definierten Variablen anzugeben.

Folgendes Beispiel zeigt, wie mit einer GReQL-Query in dem Java-Code von Abbildung 4 die leere For-Schleife in Zeile 3 gefunden werden kann.

```
1 public class Example2 {
2     public static void loop() {
3         for (int i = 0; i <= 3; i++) {
4         }
5     }
6 }
```

Abbildung 4: Leere For-Schleife

Die folgende GReQL-Query erkennt zwei mögliche Strukturen für leere For-Schleifen:

```
from
  t : V{ForStatement}
with
   not isEmpty(t -->{ForStatementBody} & {EmptyStatement}) or
  (not isEmpty(t -->{ForStatementBody} & {Block}) and
   isEmpty(t -->{ForStatementBody} & {Block} -->{Child}))
report
  t.line as "line" end
```

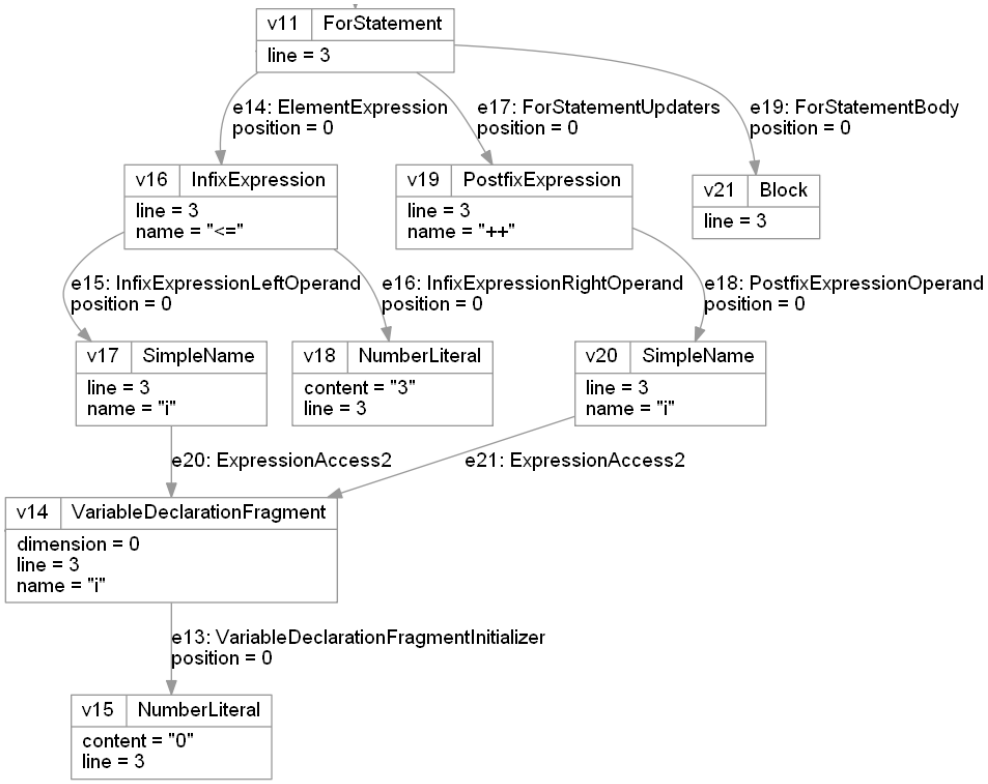


Abbildung 5: Ausschnitt TGraph von Abbildung 4

Zunächst wird in der FROM-Klausel die Variable t vom Typ ForStatement definiert. Danach werden mittels der WITH-Klausel Anforderungen an die Variable t gestellt. Im o. g. Beispiel trifft der zweite Teil der OR-Bedingung zu, wobei zwei Bedingungen erfüllt werden müssen: Zum einen muss die Variable t über eine ForStatementBody-Kante mit einem Block-Knoten verbunden sein und zum Anderen darf von der Variable t kein Pfad über einen ForStatementBody-Kante und über einem Block Knoten zu einem Child-Knoten führen.

Das erste Kriterium wird durch den Pfad v11 → e19 → v21 erfüllt, das Zweite ist dadurch erfüllt, das es maximal nur eine ForStatementBody-Kante gibt und der Knoten v21 keine ausgehenden Kanten hat.

Zuletzt wird das Attribute line von allen zutreffenden ForStatements mittels der REPORT-Klausel zugegeben und im Feedback weiter benutzt werden.

2.5 Regeln für den GReQL-Checker

Für den statischen Test (siehe 2.2) wird für jeden GReQL-Checker eine Regel-Datei benötigt. In dieser Datei ist jede anzuwendende GReQL-Abfrage zusammen mit dem dazu gehörigen Feedback als eine Regel zusammengefasst. Bei der Überprüfung wird jede Regel nacheinander auf den TGraphen der abgegebenen Lösung angewendet. Um die Funktionalität der Regeln genauer beschreiben zu können, existieren eine Reihen von Attributen.

Die Regel-Datei benutzt eine XML-artige Syntax:

```
<checkerrules>
  <rule type="absence" points="1">
    <query> from p : V{PackageDeclaration}
      report 0 end
    </query>
    <feedback prefix="Paketdefinition">
      Deine Lösung beinhaltet eine package-Angabe. Bitte entferne sie.
    </feedback>
  </rule>

  <rule type="absence" points="1" multiple="true">
    <query>
      from m : V{MethodDeclaration}
      with
        m.name = "fibo" and
        isEmpty(m -->* &amp;amp; {ForStatement})
      report m.line as "line", m.name as "name" end
    </query>
    <feedback>
      Die Methode {name} in Zeile {line} soll mit einer For-Schleife implementiert werden.
    </feedback>
  </rule>
</checkerrules>
```

Abbildung 6: Beispiel für eine Regel-Datei für den GReQL-Checker

<checkerrules> - umschließt alle Regeln in der Regel-Datei.

<rule> - eine Regel, zusammen gesetzt aus <query> und <feedback>. Folgende Attribute sind möglich:

- type – der Typ der Regel:

  absence: gibt das Feedback aus, wenn die Query EIN oder mehrere Ergebnis hat.

  presence: gibt das Feedback aus, wenn die Query KEIN Ergebnis hat.

- points – gibt an, wie stark die Regel gewichtet wird.

- Multiple – gibt an, ob alle Ergebnisse ausgegeben werden sollen [optional, false wenn nicht gesetzt] (siehe 3.2)

<query> - eine oder mehrere GReQL-Abfragen

<freedback> - das Feedback, das beim Zutreffen der Regel ausgeben wird. Über das Attribut prefix kann eine Kategorie festgelegt werden.

Die Regel-Datei selber wird als ein .xml-Dokument gespeichert. Dabei muss darauf geachtet werden, dass die GReQL-Syntax nicht als HTML-Element erkannt wird [7]. Aus diesem Grund müssen alle Sonderzeichen maskiert werden, die Teil der HTML-Syntax sind. Für GReQL müssen das „Kleiner als"-Zeichen < durch ein < und das Ampersand & durch ein & ersetzt werden.

3 Erweiterung JACK

3.1 Änderungen jGralab

Die dieser Arbeit zugrundeliegende Version von JACK basiert auf einer veralteten Version der jGralab Bibliothek aus dem Jahr 2012. In den letzten zwei Jahren wurde das Projekt allerdings weiterentwickelt. Dadurch besitzt JACK nicht alle in der aktuellen Referenzcard [3] beschriebenen Funktionen oder benutzt Funktionen mit anderen Parametern.

Die jGralab-Bibliothek wird dazu genutzt, um in Java mit dem TGraphen arbeiten zu können. Durch diese Schnittstelle ist es möglich, den Umfang von GReQL 2 zu erweitern und zu verändern, unter anderem können so fehlende Funktionen selbst hinzugefügt werden.

Im Abschnitt 4.2.1 [STYLE03] wird überprüft, ob ein String komplett aus Großbuchstaben besteht. Diese Funktion fehlt in der für die Arbeit benutzten JACK-Version, kann aber (wie in Abbildung 7: Code-Ausschnitt isUpperCase.java beschrieben) einfach hinzugefügt werden.

```
64 public class IsUpperCase extends Greql2Function {
65     {
66         JValueType[][] x = { { JValueType.STRING, JValueType.BOOL } };
67         signatures = x;
68         description = "Checks whether a string is all uppercase.";
69 
70         Category[] c = { Category.STRINGS };
71         categories = c;
72     }
73 
74     @Override
75     public JValue evaluate(Graph graph,
76             AbstractGraphMarker<AttributedElement> subgraph, JValue[] arguments)
77             throws EvaluateException {
78         if (checkArguments(arguments) < 0) {
79             throw new WrongFunctionParameterException(this, arguments);
80         }
81 
82         String s = arguments[0].toString();
83         return new JValueImpl(s.toUpperCase().equals(s));
84     }
```

Abbildung 7: Code-Ausschnitt isUpperCase.java

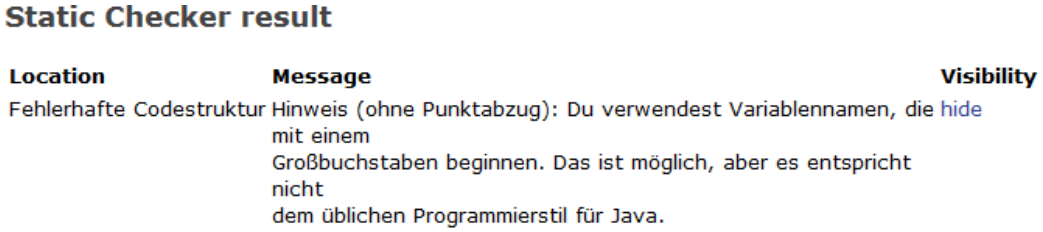
Damit die neue Funktion in GReQL benutzt werden kann, muss sie der Funktions-Bibliothek hinzugefügt werden:

```
Greql2FunctionLibrary.instance().registerUserDefinedFunction(
                                IsUpperCase.class);
```

3.2 Änderungen GReQL-Checker

Im JACK System kann eine Regel maximal ein Feedback erzeugen. Hierdurch wird, obwohl die eine Regel mehrfach verletzt wurde, immer nur eine der Verletzungen angezeigt.

Diese Vorgehensweise hat durchaus Vorteile, weil z.B. Regeln nicht so streng formuliert werden müssen, da keine Duplikate angezeigt werden. Diese Methode erschwert das Auffinden der Fehlerquelle, da die Regeln meist nur angeben, dass sie etwas gefunden haben, aber nicht wo (siehe Abbildung 8).


Static Checker result

Location	Message	Visibility
Fehlerhafte Codestruktur	Hinweis (ohne Punktabzug): Du verwendest Variablennamen, die mit einem Großbuchstaben beginnen. Das ist möglich, aber es entspricht nicht dem üblichen Programmierstil für Java.	hide

Abbildung 8: Fehlende Zeilenangabe im Feedback

Um bessere Hinweise zur Fehlerkorrektur geben zu können, müssen drei Probleme gelöst werden:

1. Eine Möglichkeit, alle Verletzungen einer Regeln anzuzeigen

2. Rückwärtskompatibilität mit alten Regeln

3. Duplikate derselben Verletzung auszublenden

Die Punkte 1 und 2 werden durch eine Erweiterung der Regelbeschreibung gelöst.

Dazu wird der XML-Tag <rule> um das Attribute multiple = „true|false“ erweitert. Um die Rückwärtskompatibilität sicherzustellen, muss multiple optional sein und standartmäßig den Wert false haben.

Punkt 3 lässt sich durch die Einführung eines Zwischenspeichers für das Feedback beheben. Dabei wird das Feedback zuerst komplett erstellt und auf Duplikate geprüft. Die Ausgabe erfolgt erst, wenn alle Duplikate gelöscht wurden.

Mit der Wahl einer Datenstruktur, die keine Duplikate erlaubt (z. B. einem HashSet), ist das Prüfung auf Duplikate nicht nötig und kann deshalb übersprungen werden.

Um die geplanten Änderungen zu implementieren, wird der Klasse `CheckerRule` um ein Boolean-Attribut multiple erweitert und die Klasse `Checker` wie folgt geändert:

```
89                  if (evalResult.toString().length() > 2
90                          && rule.getType() == RuleType.ABSENCE) {
91                      String feedback = rule.getFeedback();
92                      logger.debug(evalResult);
93
94                      JValueTable t = evalResult.toJValueTable();
95                      JValueTuple names = t.getHeader();
96                      JValueCollection data = t.getData().toCollection();
97
98                      if (data.size() == 1 || !rule.getMultiple()) {
99
100                         for (int i = 0; i < names.size(); i++) {
101                             feedback = feedback.replace(
102                                     "{" + names.get(i).toString() + "}",
103                                     ((JValueTuple) data.iterator().next()).get(
104                                             i).toString());
105                         }
106
107                         errorLines.add(rule.getFeedbackPrefix());
108                         errorLines.add(feedback);
109                         logger.info("Match found for ABSENCE rule. Reporting rule violation: "
110                                 + feedback);
111                     } else {
112                         HashSet<String> feed = new HashSet<String>();
113                         for (JValue d : data.toCollection()) {
114                             feedback = rule.getFeedback();
115                             for (int i = 0; i < names.size(); i++) {
116                                 feedback = feedback.replace("{"
117                                         + names.get(i).toString() + "}",
118                                         ((JValueTuple) d).get(i).toString());
119                             }
120
121                             feed.add(feedback);
122                         }
123                         for (String s : feed) {
124                             errorLines.add(rule.getFeedbackPrefix());
125                             errorLines.add(s);
126                             logger.info("Match found for ABSENCE rule. Reporting rule violation: "
127                                     + s);
128                         }
129                     }
130                 } else if (evalResult.toString().length() <= 2
```

Abbildung 9: Codeauschnitt Checker.java

In Zeile 98 wird zunächst überprüft, ob die Regel genau ein Ergebnis hat oder ob das Multiple-Attribut nicht gesetzt ist. Falls ersteres zutrifft, wird nach dem alten System verfahren. Wenn die Regel aber mehr als ein Ergebnis hat, wird das neue System eingesetzt.

Die Methoden unterscheiden sich darin, dass bei gesetzter Multiple-Flag zuerst das komplette Feedback in einem HashSet gespeichert wird und erst danach ausgegeben wird. Im Gegensatz dazu wird ohne Flag nur das erste Ergebnis ausgegeben.

4 Allgemeine Graphenabfragen

In den folgenden Abschnitten werden die im Zuge dieser Arbeit entwickelten Regeln zur Unterstützung von Studierenden und Lehrenden vorgestellt und erläutert. Abhängig von der Komplexität der Regel, wird zusätzlich noch auf deren Funktionsweise eingegangen. Die Regeln werden nach folgendem Schema dargestellt:

Name: der Name der Regeln und ihr Identifikationscode

Beschreibung: eine Beschreibung, wozu die Regel genutzt werden kann und warum sie nötig ist

Beispiel: ein syntaktisch korrektes Codebeispiel, welches einen Fehler enthält den die Regel findet

Regel: der Code der Regel in einem Format, das von JACK ge nutzt wird

Erklärung: (optional) eine Erklärung der vorgestellten Regel anhand eines oder mehreren TGraphen.

4.1 Unterstützung der Studierenden

4.1.1 Konstruktoren

Name:

Konstruktor – Parameter überdeckt Feld [KON01]

Beschreibung:

Die Java Syntax ermöglicht es, dass lokale Variable oder Parameter denselben Namen wie globale Variablen oder Felder haben. Dieses Feature ist zwar nützlich, aber gerade für Programmieranfänger oft verwirrend. In Konstruktoren kann es dadurch zu schwerwiegenden Fehlern kommen, wenn z. B. Attribute der Klasse nicht korrekt gesetzt und dann später aber benutzt werden.

Beispiel:

```
public class A {
  private int foo;

  public A(int foo) {
    foo = foo;
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
      from con :V{MethodDeclaration},
           a   :V{Assignment},
           dec :V{SingleVariableDeclaration}
      with
        con.constructor = true and
        con -->* a and
        a-->{AssignmentLeftHandSide3}-->{ExpressionAccess2} dec and
        a-->{AssignmentRightHandSide}-->{ExpressionAccess2} dec
      report a.line as "line", a.name as "name" end
  </query>
  <feedback prefix="Konstruktor">
    In Zeile {line} wird der Parameter {name} sich selber und nicht dem
    Feld zugewiesen.
  </feedback>
</rule>
```

Name:

Konstruktor – Feld existiert nicht [KON02]

Beschreibung:

In Java ist es zwar guter Stil, dass die Argumente des Konstruktors genauso wie die passenden Attribute heißen, allerdings wird dies nicht kontrolliert. So ist es z. B. durch Copy&Paste gerade bei Programmieranfänger möglich, dass im Konstruktor folgende Struktur sichtbar wird:

Beispiel:

```
public class A {
  private int foo;

  public A(int initFoo) {
    this.foo = foo;
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from td  : V{TypeDeclaration},
         con : V{MethodDeclaration},
         t   : V{Assignment},
         m   : V{VariableDeclarationFragment}
    with
      con.constructor = true and
      td -->{TypeDeclarationBodyDeclarations2} con and
      con -->* t and
      td -->{TypeDeclarationBodyDeclarations1}-
            >{FieldDeclarationFragments} m and
      t-->{AssignmentLeftHandSide2}-->{FieldAccessName}-
            >{ExpressionAccess2} m and
      t-->{AssignmentRightHandSide}-->{ExpressionAccess2} m
    report t.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Konstruktor">
    In Zeile {line} existiert der Parameter {name} nicht, deshalb wird das
    Feld {name} sich selber zugewiesen.
  </feedback>
</rule>
```

Name:

Konstruktor – Parameter wird auf Feld gesetzt [KON03]

Beschreibung:

Die Zuweisung in Java weist dem linken Argument den Inhalt des rechten zu. Gerade bei Konstruktoren führt ein Vertauschen der Zuweisung zu einer Exception, wenn das Programm ausgeführt wird und dem Parameter ein nicht initialisiertes Feld zugewiesen wird.

Beispiel:

```
public class A {
  private int foo;

  public A(int foo) {
    foo = this.foo;
  }
}
```

Regel:

```
<rule type="absence" points="1">
  <query>
    from con : V{MethodDeclaration},
         t   : V{Assignment},
         m   : V{SingleVariableDeclaration}
    with
      con.constructor = true and
      con -->* t and
      con-->{MethodDeclarationParameters} m and
      t-->{AssignmentLeftHandSide3}-->{ExpressionAccess2} m and
      not isEmpty(t-->{AssignmentRightHandSide} &amp; {FieldAccess})
    report t.line as "line", m.name as "name" end
</query>
  <feedback prefix="Konstruktor">
    Du versuchst in Zeile {line} den Parameter {name} auf ein (eventuell)
    nicht initialisiertes Feld zusetzen.
  </feedback>
</rule>
```

4.1.2 Leere Codeblöcke

Name:

Schleifen – Leere For-Schleifen [EMPTY01]

Beschreibung:

Bei der Syntax der For-Schleife wird nicht überprüft, ob der Körper der Schleife leer ist oder durch ein Semikolon von der Schleife getrennt wird.

Beispiel:

```
public class A {
  public static void test(){
    for(int i = 0; i <= 3; i++);
    for(int i = 0; i <= 3; i++); { }
    for(int i = 0; i <= 3; i++)  { }
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from t : V{ForStatement}
    with
      not isEmpty(t -->{ForStatementBody} &amp;amp; {EmptyStatement}) or
     (not isEmpty(t -->{ForStatementBody} &amp;amp; {Block}) and
      isEmpty(t -->{ForStatementBody} &amp;amp; {Block} -->{Child}))
    report t.line as "line" end
  </query>
  <feedback prefix="leere Blöcke">
    In der Zeile {line} befindet sich eine For-Schleife, die keinen Körper
    hat und deshalb sinnlos ist.
  </feedback>
</rule>
```

Name:

Schleifen – Leere While-Schleifen [EMPTY02]

Beschreibung:

Bei der Syntax der While-Schleife wird nicht überprüft, ob der Körper der Schleife leer ist oder durch ein Semikolon von der Schleife getrennt wird.

Beispiel:

```
public class A {
  public static void test(boolean t){
    while(t);
    while(t); {}
    while(t)  {}
  }
}
```

Regel:

```
<rule type="absence" points="2" multiple="true">
  <query>
    from x : V{WhileStatement}
    with
       not isEmpty(x -->{WhileStatementBody} &amp;amp; {EmptyStatement}) or
      (not isEmpty(x -->{WhileStatementBody} &amp;amp; {Block}) and
       isEmpty(x -->{WhileStatementBody} &amp;amp; {Block} -->{Child})
      )
    report x.line as "line" end
  </query>
  <feedback prefix="leere Blöcke">
    In der Zeile {line} befindet sich eine While-Schleife, die keinen
    Körper hat und deshalb sinnlos ist.
  </feedback>
</rule>
```

Name:

If-Then-Block – Leere Blöcke [EMPTY03]

Beschreibung:

Ähnlich wie in [EMPTY01] besteht die Möglichkeit, dass nach einem IF-Statement ein leerer Code-Block kommt. Es ist zu überprüfen, dass entweder kein ELSE-Block vorhanden ist oder beide Zweige leer sind.

Beispiel:

```
public class A {
  public static void test(boolean t){
    if (t) { }
    if (t) { } else { }
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from t : V{IfStatement}
    with
      isEmpty(t--> {IfStatementThenStatement}-->{Child}) and
      isEmpty(t--> {IfStatementElseStatement}-->{Child})
    report t.line as "line" end
  </query>
  <feedback prefix="leere Blöcke">
    In Zeile {line} befindet sich ein IF-Block, der keinen Effekt (leerer
    Block) oder einen nicht gewuenschten Effekt hat (Semikolon vor den
    bedingten Anweisungen).
  </feedback>
</rule>
```

Name:

If-Then-Block –Blöcke mit TRUE [EMPTY04]

Beschreibung:

Der Java-Compiler erkennt toten Code. Bei totem Code handelt es sich um Code, der nie aufgeführt wird. Im Gegensatz dazu wird Code in einem IF-Statement, welches immer ausgeführt wird, nicht als solcher erkannt.

Beispiel:

```
public class A {
  public static void test(boolean t){
    if (true) t = !t;
    if (true) {t = !t;}
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from t : V{IfStatement}
    with
      not isEmpty(t -->{ElementExpression} &amp;
                       {BooleanLiteral @ thisVer  tex.content = "true"})
    report t.line as "line" end
  </query>
  <feedback prefix="leere Blöcke">
    In Zeile {line} befindet sich der Start eines IF-Blocks der IMMER
    ausgeführt wird.
  </feedback>
</rule>
```

4.2 Unterstützung der Lehrenden

4.2.1 Style Guide

Die Regeln in diesem Bereich untersuchen vor allem die Groß- und Klein-Schreibung der Namen von Variablen, Attributen und Konstanten. Als Orientierung gilt die offizielle Oracle Java Style Guide [5].

Name:

Style Guide – Variablennamen [STYLE01]

Beschreibung:

Laut Style Guide beginnen alle Variablennamen mit einem Kleinbuchstaben.

Beispiel:

```
public class A {
  public static void test() {
    int VarTest = 1;
    final int FinalTest = 10;
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from x : V{VariableDeclarationStatement},
         i : V{VariableDeclarationFragment}
    with
      (isEmpty(x--> &amp;amp; {Modifier @ thisVertex.name = "final"}) or
       isEmpty(x--> &amp;amp; {Modifier @ thisVertex.name = "static"})) and
      x --> i and
      i.name=capitalizeFirst(i.name)
    report x.line as "line" end
  </query>
  <feedback prefix="Style">
    In Zeile {line} verwendest du einen Variablennamen, der mit einem
    Großbuchstaben beginnt.
  </feedback>
</rule>
```

Name:

Style Guide - Attributname [STYLE02]

Beschreibung:

Laut Style Guide beginnen alle Attributnamen mit einem Kleinbuchstaben.

Beispiel:

```
public class A {
  int VarTest = 1;
  final int FinalTest = 2;
  static int StaticTest = 3;
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from x : V{FieldDeclaration},
         i : V{VariableDeclarationFragment}
    with
     (isEmpty(x--> &amp;amp; {Modifier @ thisVertex.name = "final"}) or
      isEmpty(x--> &amp;amp; {Modifier @ thisVertex.name = "static"})) and
     x --> i and
     i.name=capitalizeFirst(i.name)
    report x.line as "line" end
  </query>
  <feedback prefix="Style">
    In Zeile {line} verwendest du einen Attributenamen, der mit einem
    Großbuchstaben beginnt.
  </feedback>
</rule>
```

Name:

Style Guide – Konstanten Name [STYLE03]

Beschreibung:

Laut Style Guide werden Konstanten, also Attribute mit dem Modifier final static, als Ganzes groß geschrieben.

Beispiel:

```
public class A {
  final static int tEST_KONSTANTE = 1;
  final static int konstetest = 2;
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from x : V{FieldDeclaration},
         i : V{VariableDeclarationFragment}
    with
      not isEmpty(x --> &amp;amp; {Modifier @ thisVertex.name = "final"}) and
      not isEmpty(x --> &amp;amp; {Modifier @ thisVertex.name = "static"}) and
      x --> i and
      not isUpperCase(i.name)
    report x.line as "line" end
  </query>
  <feedback prefix="Style">
    In Zeile {line} verwendest du einen Konstantennamen, der nicht
    großgeschrieben ist.
  </feedback>
</rule>
```

Erklärung:

Die Funktion isUpperCase, ist in der von JACK genutzten jGralab-Bibliothek nicht vorhanden. Deshalb wird hier die in 3.1 entwickelte Funktion genutzt.

4.2.2 Statische Return Values

Name:

Statische Rückgabe – Literal Type[STATIC01]

Beschreibung:

In Prüfungssituationen, insbesondere in der ersten Hälfte der Programmiervorlesung, wird ein großer Teil der Punkte über Black-box-Tests vergeben. Besonders dann, wenn Funktionen getestet werden, die als Rückgabe-Typ einen Boolean-Wert haben, können viele Punkte über eine statische Rückgabe „erschummelt" werden.

Beispiel:

```
public class A {
  public static boolean boolTest() { return true;}

  public static int intTest() { return 0;}

  public static String stringTest() { return "test";}

  public static char charTest() { return 'a';}
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from i : V{MethodDeclaration}
    with
      not isEmpty(i-->*&amp;{ReturnStatement}) and
      isEmpty(i -->* &amp; {ReturnStatement} -->* &amp;{SimpleName}) and
      isEmpty(i (-->* &amp; {IfStatement} |
                 -->* &amp; {SwitchStatement})
                -->* &amp; {BooleanLiteral}) and
      isEmpty(i (-->* &amp; {IfStatement} |
                 -->* &amp; {SwitchStatement})
                -->* &amp; {NumberLiteral}) and
      isEmpty(i (-->* &amp; {IfStatement} |
                 -->* &amp; {SwitchStatement})
                -->* &amp; {StringLiteral}) and
      isEmpty(i (-->* &amp; {IfStatement} |
                 -->* &amp; {SwitchStatement})
                -->* &amp; {CharacterLiteral})
    report i.name as "name" end</query>
  <feedback>
    Statischer Rückgabewert in der Methode {name}
  </feedback>
</rule>
```

Erklärung:

Die Regel sucht im TGraphen nach einem konstanten Rückgabewert, z.B „true" oder „false" für Boolean-Rückgabewerte. Es ist allerdings möglich, dass diese konstanten Werte in legitimem Code auftauchen, z.B. innerhalb von if-Abfragen (siehe Abbildung 9). Diese Strukturen werden dadurch erkannt, dass sich ein return-Statement innerhalb einer if-Abfrage oder einem switch-Statement befindet.

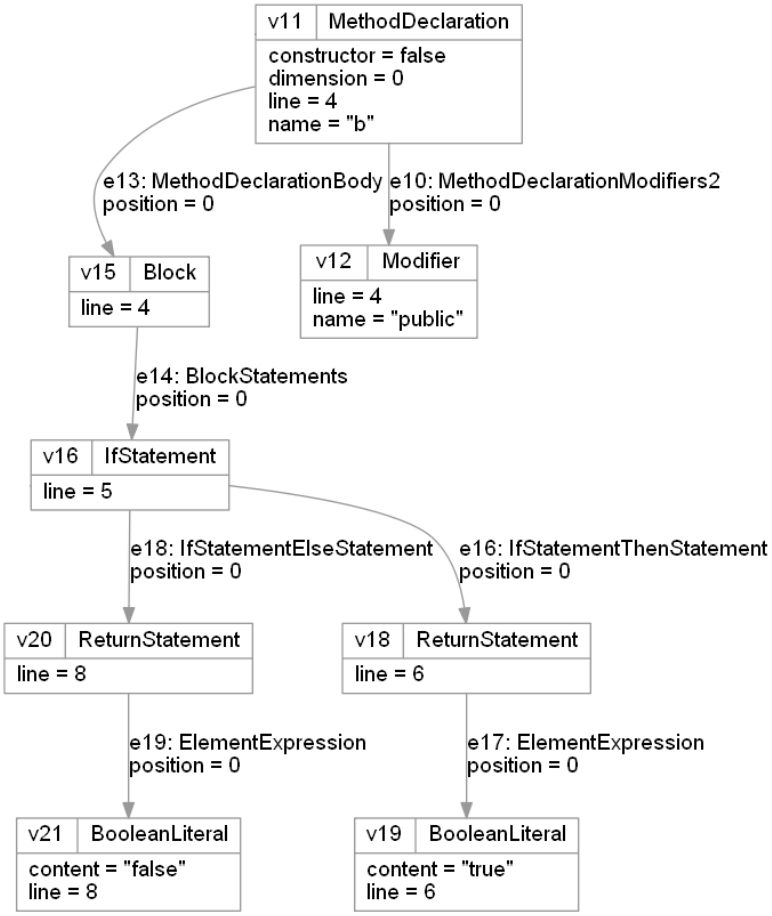


Abbildung 10: Syntax-Graph If-Abfrage mit konstanten Rückgabewerten

5 Spezielle Graphenabfragen

5.1 Vorgaben erfüllen

Name:

Vorgabe –Schleife [VORG01]

Beschreibung:

Die Regel erkennt, wenn die Methode "add" ohne Schleife implementiert wurde.

Beispiel:

```
public class A {
  public int add(int n) {
    return n * (n + 1) / 2;
  }
}
```

Regel:

```
<rule type="absence" points="1">
  <query>
    from m : V{MethodDeclaration}
    with
      m.name = "add" and
      isEmpty(m -->* &amp;amp; {ForStatement}) and
      isEmpty(m -->* &amp;amp; {WhileStatement}) and
      isEmpty(m -->* &amp;amp; {DoStatement})
    report m.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Vorgabe">
    Die Methode {name} in Zeile {line} wurde ohne eine Schleife
    implementiert.Laut Aufgabenstellung, soll die Methode aber mit einer
    Schleife gelöst werden.
  </feedback>
</rule>
```

Name:

Vorgabe – For-Schleife [VORG02]

Beschreibung:

Die Regel erkennt, wenn die Methode “add” ohne For-Schleife implementiert wurde.

Beispiel:

siehe [VORG01]

Regel:

```
<rule type="absence" points="1">
  <query>
    from m : V{MethodDeclaration}
    with
      m.name = "add" and
      isEmpty(m -->* &amp;amp; {ForStatement})
    report m.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Vorgabe">
    Die Methode {name} in Zeile {line} soll mit einer For-Schleife
    implementiert werden.
  </feedback>
</rule>
```

Name:

Vorgabe – While-Schleife [VORG03]

Beschreibung:

Die Regel erkennt, wenn die Methode “add” ohne While-Schleife implementiert wurde.

Beispiel:

```
public class Count {
  public void add(int n) {
    int erg = 0;
    for(; n > 0; n--) { erg += n;}
    return erg;
  }
}
```

Regel:

```
<rule type="absence" points="1">
  <query>
    from m : V{MethodDeclaration}
    with
      m.name = "add" and
      isEmpty(m -->* &amp;amp; {WhileStatement})
    report m.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Vorgabe">
    Die Methode {name} in Zeile {line} soll mit einer While-Schleife
    implementiert werden.
  </feedback>
</rule>
```

5.2 Rekursion vs. Iterativ

Name:

Rekursion – Methode ist iterativ, soll rekursiv sein [REGIT01]

Beschreibung:

Die Regel erkennt, wenn die Methode "fibo" ohne Rekursion implementiert wurde und nicht -wie vorgegeben- rekursiv.

Beispiel:

```
public class Rec {
  public int fibo(int n) {
    int m = 1;
    for (int i = 0; i < n; i++)
      m *= n;
    return m;
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from m : V{MethodDeclaration}
    with
      m.name = "fibo" and
      not (m -->* &amp;amp; {MethodInvocation @ thisVertex.name = m.name}
             --> {ExpressionAccess1} m)
    report m.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Vorgabe">
    Die Methode {name} in Zeile {line} wurde ohne Rekursion implementiert.
    Laut Aufgabenstellung, soll die Methode aber rekursiv gelöst werden.
  </feedback>
</rule>
```

Name:

Rekursion – Methode ist rekursiv, soll iterativ sein [REGIT02]

Beschreibung:

Die Regel erkennt, wenn die Methode "fibo" mit Rekursion implementiert wurde und nicht wie vorgegeben iterativ.

Beispiel:

```
public class Rec {
  public int fibo(int n) {
    if (n == 0) return 1;
    return n * fibo(n-1);
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from m : V{MethodDeclaration}
    with
      m.name = "fibo" and
      (m -->* &amp;amp; {MethodInvocation @ thisVertex.name = m.name}
         -->{ExpressionAccess1} m)
    report m.line as "line", m.name as "name" end
  </query>
  <feedback prefix="Vorgabe">
    Die Methode {name} in Zeile {line} wurde mit Rekursion implementiert.
    Laut Aufgabenstellung, soll die Methode aber iterativ gelöst werden.
  </feedback>
</rule>
```

5.3 Arrays

Name:

Arrays – Falscher Abbruch in For-Schleifen [ARR01]

Beschreibung:

In Java sind Arrays 0-basiert, das letzte Element in einem Array der Länge 4 hat damit den Index 3. Bei einer For-Schleife bedeutet das, dass die Schleife nur bis Länge – 1 laufen darf, da sonst eine Array-OutOfBounds-Exception ausgelöst wird.

Beispiel:

```
public class A {
  public static void test(){
    int[] b = new int[10];
    for(int i = 0; i <= b.length; i ++) {int j = b[i]; }
    for(int i = 0; i == b.length; i ++) {int j = b[i]; }
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from t : V{ForStatement},
         a : V{ArrayCreation}
    with
      not isEmpty(t
            (--> &amp; {InfixExpression @ thisVertex.name = "&lt;="} |
             --> &amp; {InfixExpression @ thisVertex.name = "=="})
             -->{InfixExpressionRightOperand}
             -->{QualifiedNameName} &amp;
                 {SimpleName @ thisVertex.name = "length"}) and
      t --> &amp;{InfixExpression} -->* {ExpressionAccess2} -->* a
    report t.line as "line" end
  </query>
  <feedback prefix="Arrays">
    In Zeile {line} ist die Abbruchbedingung, der for-Schleife für den
    Zugriff auf ein Array fehlerhaft.
  </feedback>
</rule>
```

Erklärung:

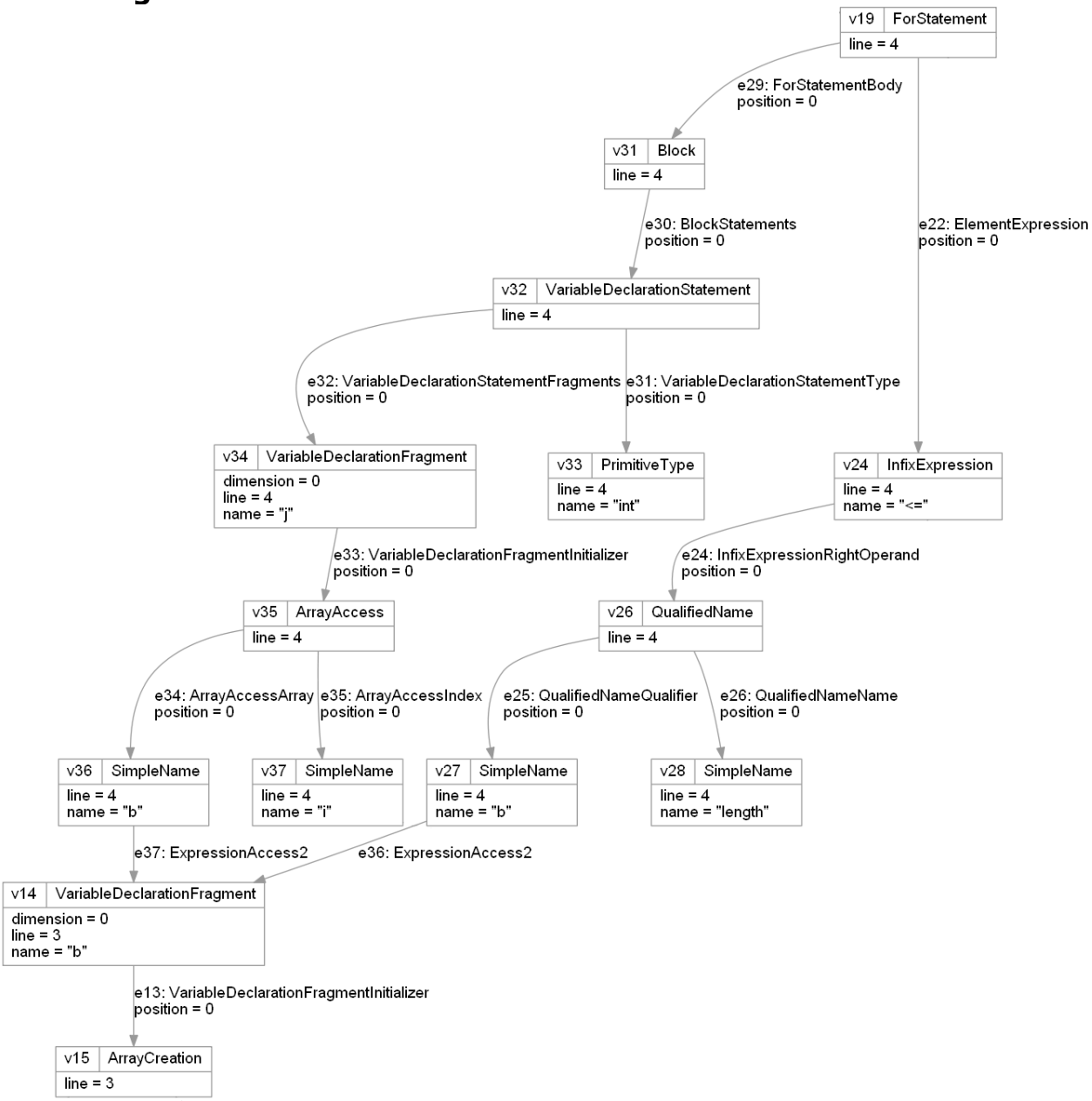


Abbildung 11: Syntax-Graph Beispiel [ARR01]

Die Regel findet den Pfad v19→ v24 →e24 →e26 → v28. Außerdem wird, durch die Verbindung von t → a über v19 → v24 → v26 → v27 → e36 → v15 sichergestellt, dass es sich wirklich um ein Array handelt und nicht um eine Klasse mit einem Feld length.

Name:

Arrays – Falscher Startwert für Rückwärts-Schleifen [ARR02]

Beschreibung:

Eine For-Schleife, die ein Array rückwärts durchläuft, muss mit dem letzen Element beginnen. Da Arrays 0-basiert sind, ist der Index des letzen Elements Länge -1.

Beispiel:

```
public class A {
  public static void test() {
    for(int i = a.length; i >=0; i--) {int j = 0;}
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from t : V{ForStatement},
         a : V{ArrayCreation}
    with
      not isEmpty(t -->{ForStatementInitializers} &amp;
                           {VariableDeclarationExpression}
                    --> &amp; {VariableDeclarationFragment}
                    --> &amp; {QualifiedName}
                    --> {QualifiedNameName} &amp;
                          {SimpleName @ thisVertex.name = "length"}) and
      t --> &amp;{InfixExpression} -->* {ExpressionAccess2} -->* a
    report t.line as "line" end
  </query>
  <feedback prefix="Arrays">
    In Zeile {line} ist die Abbruchbedingung, der for-Schleife für den
    Zugriff auf ein Array fehlerhaft.
  </feedback>
</rule>
```

5.4 Endlosschleifen

Name:

Endlosschleife – While-Endlosschleifen [END01]

Beschreibung:

Endlosschleifen sind nützlichen um z. B. einen fortlaufenden Prozess darstellen zu können. Es muss aber sichergestellt werden, dass die Schleife abgebrochen werden kann.

Beispiel:

```
public class A {
  public static void testWhile(int t) {
    while(true) { t++; }
  }

  public static void testDo(int t) {
    do { t++; } while (true);
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from x : V{WhileStatement, DoStatement}
    with
      isEmpty(x --> {ElementExpression}
                --> &amp;{BooleanLiteral @ thisVertex.content = "true"}) and
      isEmpty(x (--> {WhileStatementBody} | --> {DoStatementBody})
                 -->* &amp; {BreakStatement, ReturnStatement})
    report x.line as "line" end
  </query>
  <feedback prefix="Endlosschleifen">
    In Zeile {line} befindet sich eine Endlosschleife.
  </feedback>
</rule>
```

Erklärung:

Die Regel sucht nach While-/Do-Schleifen mit true als Bedingung und ohne return- oder break-Statement im Schleifenkörper.

Die Regel erzeugt bei zwei Konstrukte fehlerhafte Ergebnisse: Zum einen werden break-Statements in verschachtelten Schleifen nicht der korrekten Schleife zugeordnet und zum anderen werden break-Statements für benannte Schleifen ignoriert.

Name:

Endlosschleife – For-Endlosschleifen [END02]

Beschreibung:

Endlosschleifen sind nützlichen um z. B. einen fortlaufenden Prozess darstellen zu können. Es muss aber sichergestellt werden, dass die Schleife abgebrochen werden kann.

Beispiel:

```
public class A {
  public static void test(int t) {
    for (int i = 0;true; i++) { t = i; }
  }

  public static void test2(int t) {
    for (int i = 0; ; i++) { t = i;}
  }
}
```

Regel:

```
<rule type="absence" points="1" multiple="true">
  <query>
    from x : V{ForStatement}
    with
      (isEmpty(x --> {ElementExpression}
                 --> &amp;{BooleanLiteral @ thisVertex.content = "true"}) or
       isEmpty(x --> {ElementExpression} --> {Child})) and
      isEmpty(x --> {ForStatementBody}
                 -->* &amp; {BreakStatement, ReturnStatement})
    report x.line as "line" end
  </query>
  <feedback prefix="Endlosschleifen">
    In Zeile {line} befindet sich eine Endlosschleife.
  </feedback>
</rule>
```

Erklärung:

Zusätzlich zu [END01], ist es bei for-Schleifen möglich, das die Abbruchbedingung leer ist (siehe `test2`).

6 Fazit und Bewertung

Die Verbesserung von Regeln und die Entwicklung von weiteren Regeln, sowohl zur Unterstützung des Selbststudiums der Studierenden, als auch der Lehrenden, ist grundsätzlich sinnvoll und notwendig.

Insbesondere bieten die statischen Tests eine wichtige Verbindung zwischen dem Java-Compiler und den dynamischen Blackbox-Tests von JACK. Die Kombination von statischen und dynamischen Tests ermöglicht es, einen Großteil der vorhandenen Fehler in den Programmen zu finden und Hilfen für die Korrektur zu bieten.

Diese Funktion ist aus der Sicht der selbstständig lernenden Studierenden eine große Hilfe, da Fehler eigenständig schnell gefunden und behoben werden können.

Aus der Perspektive der Lehrenden, aber sind die statischen Tests nur bedingt als zusätzliche Unterstützung anzusehen. Triviale Fehler können meistens ohne Hilfe von JACK erkannt werden. Seltene Fehler, die auf grundlegende Fehler in der Architektur des Programms zurückzuführen sind, sind nur schwer mit statischen Tests erkennbar.

Einen Vorteil bieten die Tests in Prüfungssituationen, in denen bestimmte Vorgaben eingehalten werden müssen. Mit reinen Blackbox-Tests ist es nicht möglich, den Programmablauf zu analysieren und zu erkennen, ob die Vorgaben erfüllt wurden oder nicht.

Insgesamt sind drei Problembereiche bzw. Aufgabenbereiche für weitere Untersuchungen erkennbar:

1. Exotische Fehler

Ein Problem besteht darin, dass es nicht möglich ist, dass jeweils alle möglichen Fehler mit Regeln erkannt werden. Gerade Programmieranfänger machen oft viele verschiedenartige Fehler, da gewissen Programmierkonzepte noch nicht bekannt sind oder falsch verstanden wurden. Dadurch ist die Menge und Unterschiedlichkeit möglicher Fehler sehr groß.

2. Zeitproblem

Ein weiteres Problem ist die Tatsache, dass das Entwickeln der Regeln sehr zeitaufwendig ist: Denn nach jeder Änderung müssen die Regeln auf JACK aktualisiert werden, der GReQL-Checker neu gestartet werden und dann das Ergebnis der Auswertung mit dem Soll-Ergebnis verglichen werden.

3. Regeldatenbank

Das Erstellen von neuen Programmieraufgaben gestaltet sich als aufwendig und zeitintensiv, da die Regeldatei manuell erstellt werden muss.

Die beschrieben Probleme können durch verschiedenen Vorgehensweisen behoben oder verbessert werden:

1. Exotische Fehler

Exotische Fehler benötigen eine reaktive Vorgehensweise, da es nicht möglich ist, im ersten Ansatz alle exotischen Fehler aufzudecken. Eine Möglichkeit wäre es, durch ein manuelles Review von falschen Lösungen neue Fehlerarten zu finden. Dieser Prozess könnte durch eine Erweiterung von JACK optimiert werden: Hierbei wird eine Funktion implementiert, die für jede Aufgabe eine Liste erzeugt mit Lösungen, die mit einem korrekten statischen Test nicht die maximal Punktzahl erreicht haben.

2. Zeitproblem

Das Entwickeln und Testen von Regeln für JACK könnte durch die Benutzung von Tools vereinfacht werden. Zum Beispiel wäre ein REPL für GReQL-Query sehr nützlich, denn dadurch muss JACK nicht aktualisiert werden, um eine neue Regel zu testen, statt dessen ist das Ergebnis direkt ablesbar. Durch ein Update der jGralab-Bibliothek könnte das Problem gelöst werden, da in ihrer neusten Version neue Tools existieren. Diese machen es möglich, TGraphen direkt einzulesen und GReQL-Abfragen direkt auszuführen.

3. Regeldatenbank

Um das Erstellen von Regeldokumenten für neue Programmieraufgaben zu vereinfachen, ist folgendes denkbar: Es wird ein Programm entwickelt, das es möglich macht, aus einer Liste von Regeln zu wählen und daraus dann ein fertiges Regeldokument zu erstellen.

7 Literatur

[1] Daniel Bildhauer, Tassilo Horn, Eckhard Großmann: GReQL-Reference Card, 2014

[2] Michael Goedicke, Michael Striewe, Moritz Balz: Computer aided assessments and programming exercises with JACK; ICB-Research Report No 28, 2008

[3] Joachim Goll: Methoden und Architekturen der Softwaretechnik. Vieweg+Teubner Verlag, 2011, S. 39

[4] Katrin Marchewka : Entwurf und Definition der Graphanfragesprache GReQL 2. Zugl.: Koblenz, Univ., Dipl., 2006

[5] Achut Reddy: Java™ Coding Style Guide, Sun MicroSystems, 2000, <http://www.scribd.com/doc/15884743/Java-Coding-Style-by-Achut-Reddy>, Zugriff am 3.9.2014

[6] Michael Striewe, Moritz Balz, Michael Goedicke: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: CSEDU (2), 2009, S. 54-61.

[7] World Wide Web Consortium: HTML 4.01 specification, World Wide Web Consortium, 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>, Zugriff am 28.8.2014