Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein System entwickelt, mit dem sich falsch-positive Warnmeldungen von FindBugs, einem Werkzeug für die statische Code-Analyse, unterdrücken lassen. Der dafür verwendete Ansatz ist auch auf andere Werkzeuge der statischen Code-Analyse übertragbar. Basis des Entwurfs ist ein selbstlernender Algorithmus, der anhand der Bug-Klassifizierungen eines Benutzers lernt, in welchen Situationen eine Warnmeldung als falsch-positiv erachtet wird. Mit diesem Wissen kann der Algorithmus in zukünftigen Analysen die Situationen wiedererkennen und falsch-positive Warnmeldungen vor einer Ausgabe an den Benutzer unterdrücken. Die Situationen werden anhand individueller Merkmale des Source-Codes wiedererkannt, daher ist der Aspekt des Ähnlichkeitsvergleichs von Source-Code ein zentraler Bestandteil dieser Arbeit.

Abstract

In this thesis a system was developed to suppress false positive error reports of FindBugs, a tool for static analysis. The used approach can be adapted to other static analysis tools as well. The core of the concept is a self-learning algorithm. Based on the developer's bug classifications, the system will learn, in which situations the error reports can be considered being false positive. Based on this knowledge, the algorithm is able to recognize these situations in future analysis passes and thereby suppress the corresponding error reports. The situations get recognized on the base of individual features of the source code. Therefore the aspect of similarity measurement of source code fragments is an essential element of this approach.

INHALTSVERZEICHNIS

1. Einleitung 1
2. Aufgabenstellung 4
3. Grundlagen 7
    3.1. Statische Code-Analyse 7
    3.2. Byte-Code Libraries 10
    3.3. FindBugs 10
    3.4. Maschinelles Lernen 14
        3.4.1. Was versteht man unter Maschinellem Lernen? 14
        3.4.2. Was sind Anwendungsgebiete des Maschinellen Lernens? 15
        3.4.3. Welche Ansätze des Maschinellen Lernens gibt es? 17
    3.5. Case-Based Reasoning 18
4. Ursprünglicher Lösungsansatz 20
5. Lösungsansatz 22
6. Lösungsbeschreibung 27
    6.1. Wiederherstellung von Klassifizierungen 27
    6.2. Feature-Vektoren und Locality-Sensitive Hashing 31
    6.3. Anti-Unification 34
    6.4. Automatische und benutzergesteuerte Kontext-Spezifizierung 37
    6.5. Konvertierung und Speicherung von Kontextinformationen 40
    6.6. Ähnlichkeitsvergleich von Kontextinformationen 42
    6.7. Der Reasoning-Vorgang 43
7. Evaluation 48
8. Weiterführende Arbeiten 54
9. Fazit 57
    9.1. Kritischer Rückblick auf den Verlauf der Arbeit 57
    9.2. Inhaltliches Fazit 58
Danksagungen 60
Literaturverzeichnis 61

1. EINLEITUNG

Diese Diplomarbeit befasst sich mit dem Thema der statischen Code-Analyse, einem Verfahren zur Fehlersuche und Code-Optimierung, das bei der Software-Entwicklung zum Einsatz kommt. Mit diesem Verfahren wird der Programmcode einer Software analysiert, um potentielle Programmierfehler aufzuspüren. Je mehr Fehler durch ein solches Verfahren aufgespürt werden, desto größer ist der Nutzen und desto lohnenswerter ist es, ein solches Verfahren in die Entwicklungsprozesse einer Software zu integrieren. Werden Fehler durch die statische Code-Analyse nicht erkannt, spricht man von sogenannten falsch-negativen Anzeigen. Es existiert aber ein weiteres Problem, das ebenfalls von großer Relevanz für die statische Code-Analyse ist, die falsch-positiven Anzeigen. Es wird ein Fehler angenommen, obwohl in Wirklichkeit keiner vorliegt. Dieses Problem wurde wie folgt durch die Ausschreibung meiner Diplomarbeit benannt:

„Die Werkzeuge für die automatische statische Code-Analyse können die Intentionen des Entwicklers nicht kennen. Daher kommt es immer wieder zu falsch-positiven Anzeigen.“

Werkzeuge, die in der statischen Code-Analyse angewandt werden, können hohe Raten von falsch-positiven Anzeigen aufweisen. FindBugs ist ein häufig eingesetztes Tool in der statischen Code-Analyse. Mit diesem Tool befasse ich mich in dieser Arbeit. Die Entwickler von FindBugs haben in ihrer ersten Veröffentlichung eine Rate falsch-positiver Anzeigen von bis zu 50% angegeben (Hovemeyer & Pugh, 2004). Das zentrale Thema dieser Arbeit ist es, Möglichkeiten der Vermeidung von falsch-positiven Anzeigen zu erörtern und diese als Erweiterungen in FindBugs zu integrieren.

Falsch-positive Anzeigen sind nicht alleine die Folge von Unzulänglichkeiten in der Analyse. Manche Fehler, die nach formalen Kriterien tatsächlich als Fehler zu bewerten sind, jedoch nicht kritisch für die Ausführbarkeit eines Programms sind, können möglicherweise nach Ansicht des Entwicklers in einem bestimmten Kontext ver-

nachlässigt werden. Der Entwickler hat in der Regel ein spezielles Wissen über die Semantik seines Programms. Dieses Wissen wäre nötig, um entscheiden zu können, ob ein formaler Fehler in einem gegebenen Kontext vernachlässigbar ist oder nicht. Dies kann zum Beispiel das Wissen über die möglichen Werte und Typen von generischen Parametern sein, das dem Entwickler zur Verfügung steht, dem Programm, das die statische Code-Analyse durchführt, jedoch nicht.

Falsch-positive Anzeigen können für den Entwickler zu einem Problem werden, wenn sie sich häufen. Eine Häufung solcher Anzeigen, die der Entwickler letztlich als vernachlässigbar einstuft, ist sogar wahrscheinlich. Viele Programme beinhalten sich wiederholende Code-Fragmente, wiederkehrende Teil-Aufgaben, wie zum Beispiel die Abfrage von Array-Elementen innerhalb einer Schleife.

Menschen sind, einem allgemein bekannten Sprichwort nach, Gewohnheitstiere. Hat sich ein Entwickler einen gewissen Programmierstil angewöhnt, der zu einer falsch-positiven Anzeige führt, so ist es wahrscheinlich, dass sich der vermeintliche Fehler im Programmcode des Entwicklers wiederholt. Die Ursache sind Programmiergewohnheiten, mitunter auch schlechte Programmiergewohnheiten, die der Entwickler selber aber nicht als Problem ansieht. Der Entwickler wird in einem solchen Szenario immer wieder mit Falschmeldungen konfrontiert, was sehr zeitraubend ist und ihn dazu verleiten kann, die Werkzeuge nicht mehr zu verwenden. Die Akzeptanz des durchaus nützlichen Verfahrens leidet unter der Häufung falsch-positiver Anzeigen. Eine Senkung der hohen falsch-positiv Raten könnte den Nutzen der Werkzeuge erheblich steigern. Sie stellen eine gute Ergänzung zu manuellen Code-Inspektionen dar (Wagner, 2011).

Eine sinnvolle Erweiterung der Verfahren zur statischen Code-Analyse wäre, dass einmal begutachtete und durch den Entwickler als falsch-positiv identifizierte Warnungen in zukünftigen Analysen desselben Programms nicht mehr auftauchen würden. Somit würde der Benutzer während mehrerer Analysedurchgänge nicht immer wieder auf dieselben schon untersuchten Code-Abschnitte hingewiesen.

Darüber hinaus kann es Code-Abschnitte geben, die falsch-positive Warnmeldungen auslösen und einen ähnlichen Kontext besitzen wie andere Abschnitte, bei denen der Entwickler bereits eine falsch-positive Warnmeldung identifiziert hat. Es wäre wünschenswert, wenn auch diese sich wiederholenden falsch-positiven Warnmeldungen auf Basis ihres Kontexts als solche erkannt würden.

Im einfachsten Fall handelt es sich bei einem Code-Abschnitt um eine exakte Kopie eines anderen Abschnitts, der bereits eine falsch-positive Warnmeldung ausgelöst hat. Es kann aber auch vorkommen, dass es sich nicht um exakte Kopien handelt, aber ein ähnlicher Kontext gegeben ist und derselbe Fehlertyp vorliegt. Will man in beiden Fällen Warnmeldungen als falsch-positiv wiedererkennen, so gilt es, relevante Kontexinformationen aus dem Code zu extrahieren, um die Charakteristik eines Fehlers beschreiben zu können. Auf Basis dieser Beschreibung ist es möglich, verschiedene Fehler zu vergleichen, und im Falle einer ausreichenden Ähnlichkeit, falsch-positive Warnmeldungen zu unterdrücken.

Die Kernfrage dieser Arbeit wird daher sein:

„Was sind relevante Kontextinformationen für die Klassifizierung von Fehlern, um eine bestmögliche Wiedererkennung zu ermöglichen und wie können diese Kontextinformationen erfasst, gespeichert und genutzt werden?“

2. AUFGABENSTELLUNG

In dieser Diplomarbeit befasse ich mich mit Ansätzen zur Senkung der Rate falsch-positiver Anzeigen bei der statischen Code-Analyse. Diese Ansätze werden in das Tool FindBugs integriert. FindBugs ist ein Programm zur statischen Code-Analyse von Java-Programmcode. FindBugs ist in der Programmiersprache Java geschrieben und kann als Standalone-Tool verwendet werden. Es besitzt aber auch eine Integration in die Entwicklungsumgebung Eclipse, in Form eines Plugins. Grundlage der Entwicklung meines Systems ist die, zum Zeitpunkt des Verfassens dieser Arbeit, aktuelle Version 2.0.2. des Plugins.

Bei der Verwendung des Eclipse-Plugins werden dem Anwender von FindBugs die gefundenen Fehler direkt im Eclipse-Code-Editor angezeigt. Dies geschieht durch sog. Marker in der Programmzeile, die den Fehler enthält. Das Plugin bietet auch eine eigene Eclipse-Perspektive an. Die in dieser Perspektive enthaltenen Views zeigen dem Entwickler detaillierte Informationen über gefundene Fehler an. In der Bug-Explorer-View kann der Entwickler in einer Baumstruktur durch alle gefundenen Fehler navigieren.

Die Ansätze, die ich in dieser Diplomarbeit untersuche, bedürfen stets eines Feedbacks durch den Entwickler. Der Entwickler muss falsch-positive Anzeigen in seinem Programm als solche identifizieren. Außerdem soll es möglich sein, dass der Entwickler Informationen über den Kontext einer falsch-positiven Anzeige bereitstellt. Dadurch sollen relevante Informationen für die Wiedererkennung erfasst werden. Durch Markieren eines mehrzeiligen Code-Abschnittes soll der Entwickler einen Kontext in seinem Programm kennzeichnen, der die relevanten Informationen für die Wiedererkennung enthält.

Der markierte Code-Abschnitt enthält ggf. auch irrelevante Informationen. Eine Aufgabe meiner Arbeit ist es, die für die Wiedererkennung der falsch-positiven Anzeige relevanten Informationen aus den vom Entwickler bereitgestellten Kontextinformationen zu extrahieren. Mein Programm soll durch wiederholtes Feedback lernen, welche Informationen relevant sind.

Das Feedback des Entwicklers soll zeitlich nach der Analyse von FindBugs stattfinden. Die Analyse ergibt in jedem Durchlauf eine Liste potentieller Fehler, die dem Entwickler in der Entwicklungsumgebung angezeigt werden. Der Entwickler kann nun jeden potentiellen Fehler untersuchen und er soll die Möglichkeit erhalten, einzelne Fehler als falsch-positive Anzeigen zu kennzeichnen und ggf. relevante Informationen über den Kontext bereitzustellen. Bei der nächsten FindBugs-Analyse, die der Entwickler startet, soll nun das Feedback der vorherigen Durchläufe berücksichtigt werden und eine Wiedererkennung von falsch-positiven Anzeigen stattfinden. Dafür soll das eigentliche Analyse-Verfahren von FindBugs nicht verändert werden, sondern in einer Nachverarbeitung ein zusätzlicher Filter implementiert werden. Dieser Filter soll auf Basis des Entwickler-Feedbacks aus den vorausgegangenen Analysen falsch-positive Anzeigen erkennen und kenntlich machen.

Werden falsch-positive Anzeigen erkannt, sollen diese in der Fehlerübersicht von FindBugs mit geeigneten visuellen Mitteln deutlich für den Benutzer gekennzeichnet werden. Die Implementierung des Filters und die Modifikation der Benutzeroberfläche von FindBugs ist Teil meiner Arbeit.

Neben dem Erfassen des Entwickler-Feedbacks ist eine wichtige Aufgabenstellung der Arbeit, das Wissen des Entwicklers in einer geeigneten Form zu speichern. Sein Wissen stellt er bei jeder Analyse durch sein Feedback bereit. Dadurch entsteht eine Wissensbasis, die für spätere Analysen zur Verfügung steht und stetig erweitert werden kann. Die Wissensbasis soll nicht nur für verschiedene Analysen desselben Programms, sondern auch projektübergreifend verwendet werden können. Der Aufbau der Wissensbasis soll einem Expertensystem gleichen, das über das kontinuierliche Feedback des Entwicklers trainiert wird.

Den von mir verfolgten Ansätzen liegt die Annahme zugrunde, dass die vom Entwickler vorgenommenen Klassifizierungen von Fehlern abhängig von den Kontexten der Fehler sind. Fehler treten demnach in Situationen auf, die durch Kontextinformationen beschreibbar und erfassbar sind. Diese Kontextinformationen können, der Annahme folgend, dazu verwendet werden, fehlerauslösende Situationen durch einen Ähnlichkeitsvergleich wiederzuerkennen. Die Kontexte der Fehler werden vom Entwickler geschaffen. Wie charakteristisch die Kontexte sind und ob sie sich in ähnlicher Form an anderen Stellen wiederholen, hängt daher von den Programmiergewohnheiten des Entwicklers ab.

Eine weitere Annahme, die getroffen wird, ist, dass sehr viele Entwickler einen Programmierstil anwenden, der sich in den Kontexten von Fehlern durch erkennbare Merkmale widerspiegelt. Mein Programm soll daher für jeden individuellen Entwickler, der es verwendet, eine Senkung der falsch-positiv Raten bewirken. Um die beschriebenen Hypothesen zu verifizieren, soll diese Arbeit mit einer Evaluation des realisierten Gesamt-Systems abschließen. Eine Testgruppe soll das erweiterte Tool in Tests realer Projekte einsetzen und dessen Effektivität bewerten. Mit den Ergebnissen dieser Evaluation sollen Abschätzungen für die Nützlichkeit des Systems und kritische Betrachtungen der gewählten Ansätze möglich sein.

3. GRUNDLAGEN

3.1. STATISCHE CODE-ANALYSE

Die statische Code-Analyse ist ein falsifizierendes Software-Testverfahren, das Fehler (Bugs) im Quell-Code von Software nachweisen kann. Im Gegensatz zu dynamischen Tests ist sie dabei unabhängig von einer Ausführung des Programms. Mit der statischen Code-Analyse möchte man nachweisen, dass ein Programm bestimmte Eigenschaften erfüllt. Beispiele dafür sind:

„Das Programm dereferenziert keinen Null-Pointer“

„Jede Variable wurde initialisiert, bevor auf sie zugegriffen wird“

Solche Aussagen lassen sich aber im Allgemeinen nicht entscheiden, wie aus dem Satz von Rice[1] bekannt ist. Dieser besagt, *„dass jede nichttriviale Aussage über die von einer Turingmaschine berechnete Funktion – und damit natürlich ebenso für die von einem beliebigen Programm berechnete Funktion – unentscheidbar ist“* (Socher, 2008).

Aufgrund dieses Dilemmas kann die statische Code-Analyse nur unpräzise Antworten darauf geben, ob ein Programm bestimmte Eigenschaften erfüllt. Dennoch lassen sich mit geeigneten Techniken nützliche Informationen ableiten. Für das Verhalten eines Programms werden Unter- oder Überapproximationen angewendet, in denen Eigenschaften eines Programms entscheidbar werden. Mit dem folgenden Code-Beispiel kann solch eine Approximation veranschaulicht werden (x, y seien Variablen, S sei ein Statement, das keine Zuweisung an y enthält).

```
read(x); (if x>0 then y:=1; else (y:=2;S)); z:=y;
```

[1] Nach **Henry Gordon Rice**

Eine Analyse könnte als Annahme liefern, dass *y* nur mit einem Wert von 1 die Zuweisung *z:=y* erreichen kann. Diese Annahme ist jedoch nur korrekt, falls *S* nicht terminiert. Da im Allgemeinen aber unentscheidbar ist, ob ein Code-Abschnitt terminiert, ist eine sichere (**safe**) aber unpräzise Antwort, dass der Wert 1 oder 2 sein kann (Nielson, et al., 2005). Das entspricht einer Überapproximation, falls *S* nicht terminiert. Solche Approximationen können aber trotzdem nützliche Informationen liefern. In obiger Situation wüsste das Analyseverfahren beispielsweise, dass der Wert von *y* auf jeden Fall positiv bei Erreichen der Zuweisung *z:=y* ist.

Überapproximationen (Abbildung 1) des Programmverhaltens werden von statischen Code-Analysen vorgenommen, die man als **sound** bezeichnet. Eine statische Code-Analyse, die sound ist, würde alle Verletzungen einer gewünschten Eigenschaft finden, dabei aber unter Umständen auch falsch-positive Warnmeldungen ausgeben.

Unterapproximationen (Abbildung 1) des Programmverhaltens werden von statischen Code-Analysen vorgenommen, die man als **complete** bezeichnet. Eine statische Code-Analyse, die complete ist, würde nur tatsächliche Verletzungen der gewünschten Eigenschaften finden, also keine falsch-positiven Warnmeldungen ausgeben. Bei dieser Art Analyse gibt es aber keine Garantie, dass alle Verletzungen der Eigenschaft gefunden werden.

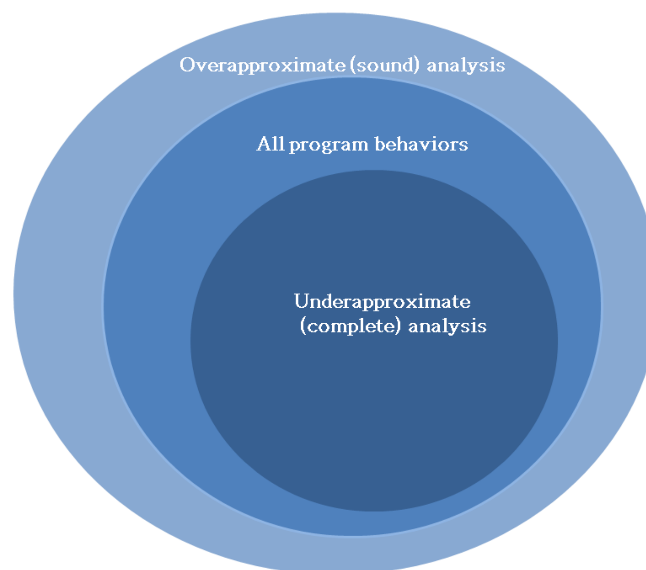


Abbildung 1 - Unter- und Überapproximation

Wenn bei einer statischen Code-Analyse, die sound ist, keine Warnmeldungen ausgegeben werden, ist es bemerkenswert, dass das betrachtete Programm garantiert keine Verletzungen der untersuchten Eigenschaft enthält. Das ist eine starke Zusicherung. Daher werden statische Code-Analyse-Verfahren eher dahingehend entworfen, sound zu sein, als complete. Am ehesten sollte eine zufriedenstellende Kombination der Faktoren Präzision, Soundness und Skalierbarkeit angestrebt werden (Dillig, et al., 2008).

Zu einer statischen Code-Analyse von Programm-Code gehören drei Komponenten: Der Parser, die interne Repräsentation, sowie die Komponente, die die Analyse auf der internen Repräsentation durchführt. Der Parser verarbeitet und überführt den Programm-Code in die interne Repräsentation. FindBugs transformiert hierfür den Byte-Code von Java-Programmen mit Bibliotheken wie BCEL[2] oder ASM[3].

Als interne Repräsentation bezeichnet man eine abstrakte Syntax, die besser für die Analyse geeignet ist als der Programm-Code selbst. Interne Repräsentationen gibt es nahezu genauso viele, wie es Analyse-Methoden gibt. Bekannte Beispiele sind der **Kontroll-Fluss-Graph** (CFG, nach englisch control flow graph) und der **Abstrakte Syntaxbaum** (AST, nach englisch abstract syntax tree) (Binkley, 2007). FindBugs verwendet als interne Repräsentation unter anderem einen Kontroll-Fluss-Graph.

Die Analyse-Komponente untersucht die interne Repräsentation des Programm-Codes auf das Vorliegen einer bestimmten Eigenschaft. Eine Analyse-Komponente kann beispielsweise eine Datenfluss-Analyse durchführen. Diese Variante der statischen Code-Analyse-Verfahren wird auch von FindBugs durchgeführt. Dabei ist es üblich, sich das Programm als einen Graph vorzustellen. Die Knoten entsprechen den elementaren Blöcken des Programms (Zuweisungen, Bedingungen etc.) und die Kanten beschreiben, wie die Kontrolle von einem Block zum nächsten übergeben wird (Nielson, et al., 2005). Die Datenfluss-Analyse verwendet daher als interne Repräsentation einen Kontroll-Fluss-Graph. Mit diesem kann festgestellt werden, wie der Wert einer Variablen in einem Programm propagiert wird. Dafür werden Gleichungen aufgestellt, die Eintritts- mit Austrittsinformationen der elementaren Blöcke in Beziehung setzen. So entsteht ein Gleichungssystem, für dessen Lösung wieder verschiedene Ansätze existieren (Nielson, et al., 2005).

[2] http://commons.apache.org/bcel/
[3] http://asm.ow2.org/

3.2. BYTE-CODE LIBRARIES

Die Byte-Code Engineering Library (BCEL) und ASM sind open-source Bibliotheken, mit denen sich Java Byte-Code Dateien (.class-Dateien), die Kompilate des Java Source-Codes, analysieren, erstellen und manipulieren lassen. Sie können daher als Parser-Komponente einer statischen Code-Analyse verwendet werden, die auf Java Byte-Code operiert. Mittels ihrer einfachen Schnittstellen kann auf die Bestandteile einer class-Datei als Objekte zugegriffen werden. Die Bibliotheken bilden dabei alle Elemente der class-Dateien ab, die auch in der Java Virtual Machine Specification definiert sind, wie z.B. den Konstanten-Pool.

BCEL und ASM unterstützen das Visitor-Design-Pattern. So lassen sich auf einfache Weise Visitor-Klassen implementieren, deren Instanzen die Elemente einer class-Datei traversieren und analysieren.

3.3. FINDBUGS

FindBugs ist ein Werkzeug für die statische Code-Analyse, das von David Hovemeyer im Rahmen seiner Doktorarbeit an der Universität von Maryland entwickelt wurde (mit Unterstützung seines Doktorvaters William Pugh). FindBugs untersucht den Byte-Code von Java-Anwendungen auf Fehler-Muster (Bug-Patterns). Seit der ersten Publikation (Hovemeyer & Pugh, 2004) zu FindBugs, im Jahr 2004, wurde das Werkzeug stetig erweitert. Waren anfangs Detektoren für knapp 50 Fehler-Muster integriert, so findet FindBugs mit der zum jetzigen Stand aktuellen Version 2.0.2 einige hundert Fehler-Muster. FindBugs ist freie Software und unter der Lesser GNU Public License verfügbar. Es kann als Standalone-Tool verwendet werden, es sind aber unter anderem auch Plugins für Eclipse und Netbeans verfügbar. FindBugs ist schon über 500.000 mal heruntergeladen worden und es wurde bereits in den Entwicklungs-Prozessen großer Firmen wie Google eingesetzt (Ayewha, et al., 2008).

Zu beachten ist, dass FindBugs nicht garantiert, alle Fehler einer bestimmten Kategorie zu finden, und auch keine Sicherheit gibt, dass eine Software keine Fehler eines bestimmten Typs enthält (Ayewha, et al., 2008). Die Analysen von FindBugs sind also im Allgemeinen weder sound noch complete (dies muss nicht notwendigerweise für jeden verwendeten Detektor im Einzelnen gelten; siehe Abschnitt 3.1. STATISCHE CODE-ANALYSE). FindBugs findet daher nur potentielle Bugs, es kann bei der Suche echte Bugs übersehen und auch falsche Warnmeldungen anzeigen (Hovemeyer & Pugh, 2004).

Im Gegensatz zu Werkzeugen, die Sicherheitsgarantien machen, ist FindBugs also eher dafür konzipiert, effektiv einfach aufzuspürende Fehler zu identifizieren (die „leichte Beute" abzugreifen) (Ayewha, et al., 2008). Nichtsdestotrotz sehen Ayewah et al. die von FindBugs gefundenen Fehler als solche an, bei denen in ausreichend vielen Fällen für einen Entwickler ein Interesse an der Begutachtung und Behebung besteht (Ayewha, et al., 2008).

Viele der Fehler-Muster, nach denen FindBugs sucht, entsprechen bekannten Fallstricken in der Java-Programmierung oder der Verletzung allgemeiner Coding-Standards. Das folgende Code-Beispiel veranschaulicht ein FindBugs Fehler-Muster:

```
String dateString = getHeaderField(name);
dateString.trim();
```

Dieses Fehler-Muster wird von FindBugs bezeichnet als „Method ignores return value (RV_RETURN_VALUE_IGNORED)". Der Auslöser des Fehlers ist in der zweiten Code-Zeile zu finden. Da Strings in Java *immutable* sind, gibt die Funktion *trim()* den modifizierten String als Rückgabewert zurück. Der Rückgabewert wird keiner Variablen zugewiesen und somit ignoriert. Dieser Fehler ist nicht kritisch für die Ausführung des Programms, da der Ausdruck *„dateString.trim();"* eine gültige Anweisung darstellt. Die Anweisung hat jedoch keine Auswirkung und es ist zu vermuten, dass die Zuweisung zu einer Variablen vergessen wurde. Dies ist ein typisches Fehler-Muster, das FindBugs erkennt. Obwohl kein syntaktischer Fehler vorliegt, impliziert dieses Code-Beispiel, dass der Entwickler eine andere Intention hatte.

FindBugs nimmt neben der Unterteilung in Bug-Patterns weitere Klassifizierungen der Bugs vor. Für jeden Bug wird die Zugehörigkeit zu einer der folgenden Kategorien angegeben:

- *Malicious Code Vulnerability:* Referenzen auf Felder oder Variablen werden an Klassen übergeben, die diese nicht benutzen sollten
- *Dodgy Code:* Verwirrender, ungewöhnlicher oder fehleranfälliger Code
- *Bad Practice*: Verstöße gegen empfohlene Coding-Standards (schlechter Programmierstil)
- *Correctness:* Offensichtliche Programmierfehler
- *Internationalization:* Verwendung von nicht lokalisierbaren Methoden
- *Performance:* Ineffiziente Speichernutzung etc.
- *Security:* Sicherheitskritischer Code
- *Multithreaded Correctness:* Probleme, die die Synchronisation von Threads betreffen
- *Experimental:* Fehler, die von neuen, experimentellen Detektoren gefunden werden

Für jeden Bug wird außerdem ein Rang festgelegt. Angefangen bei Rang 1, den schwerwiegendsten Fehlern, bis hin zu Rang 20, den harmloseren Fehlern. Der Rang soll einen Hinweis auf die Brisanz der Fehler oder deren Auswirkung geben. Die Ränge werden wiederum in die vier folgenden Gruppen zusammengefasst:

- *Scariest:* Rang 1-4
- *Scary:* Rang 5-9
- *Troubling:* Rang 10-14
- *Of Concern:* Rang 15-20

Jeder Warnmeldung eines Bugs wird eine Priorität zugewiesen. High (hoch), Medium (mittel) und Low (niedrig). Die Prioritäten werden mit gewissen Heuristiken bestimmt. Diese sind für alle Detektoren und teils auch für Bug-Patterns unterschiedlich. Die Prioritäten von Bugs unterschiedlicher Bug-Patterns sind daher nicht zwangsläufig vergleichbar (Ayewha, et al., 2008).

Für die Analyse eines Java-Programms untersucht FindBugs den Java Byte-Code. Für das Parsen des Byte-Codes wurde anfangs ausschließlich die BCEL verwendet, seit Version 1.1 wird auch das ASM Byte-Code Framework unterstützt. Beide unterstützen das Visitor-Design-Pattern, mit dem es Detektoren von FindBugs möglich ist,

die Komponenten eines class-Files und/oder den Byte-Code von Methoden zu traversieren.
FindBugs setzt auf eine Plugin-Architektur, innerhalb derer Detektoren definiert und integriert werden können. Jeder Detektor kann Fehler eines oder mehrerer Bug-Patterns finden. Die Detektoren benutzen verschiedene Techniken der statischen Code-Analyse. Die einfachsten Detektoren betrachten nur Klassenstrukturen und Vererbung von Klassen, ohne den eigentlichen Code zu untersuchen. Anspruchsvollere Detektoren untersuchen die Anweisungen im Code in einer linearen Abfolge. Dabei wird der Code Zeile für Zeile betrachtet, ohne den Kontrollfluss des Programms zu berücksichtigen. Die anspruchsvollsten Detektoren verwenden Datenfluss-Analysen, um die Abfolge der Anweisungen und den Zustand der Variablen eines Programms bei der Analyse in Betracht zu ziehen. Je anspruchsvoller die Implementierung eines Detektors ist, desto langsamer ist er in der Ausführung.

FindBugs bietet seit einigen Jahren auch ein Eclipse-Plugin an. Damit kann FindBugs leicht in den Entwicklungsprozess integriert werden. Das Plugin verwendet beispielsweise das Marker-Interface (*IMarker*) von Eclipse, um im Code-Editor Warnmeldungen über gefundene Bugs an der passenden Source-Line anzuzeigen. Das Plugin erweitert Eclipse zusätzlich um eine eigene Perspektive. Mit den darin enthaltenen Views fällt die Inspizierung der gefundenen Bugs leichter. Informationen über gefundene Bugs, wie Bug-Pattern, Bug-Typ, Kategorie, Rang, Priorität, den Fundort und eine allgemeine Beschreibung des Fehler-Musters, werden von FindBugs in der Bug-Info-View angezeigt (Abbildung 2).

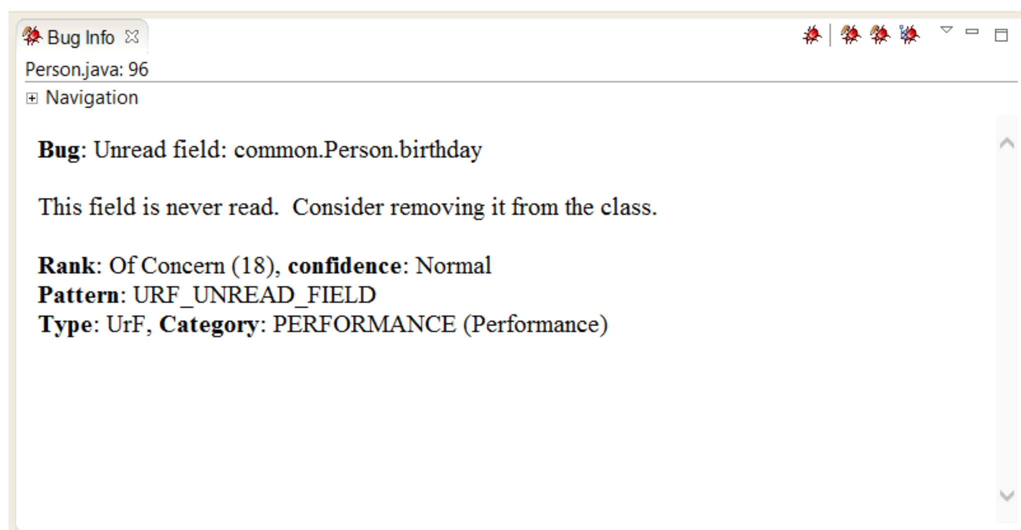


Abbildung 2 – Bug-Info-View von FindBugs

Anhand der Merkmale wird eine Unterteilung der Elemente der Bug-Explorer-View vorgenommen. Dort kann der Entwickler in einer Baumstruktur durch eine Übersicht aller gefundenen Bugs navigieren (Abbildung 3).

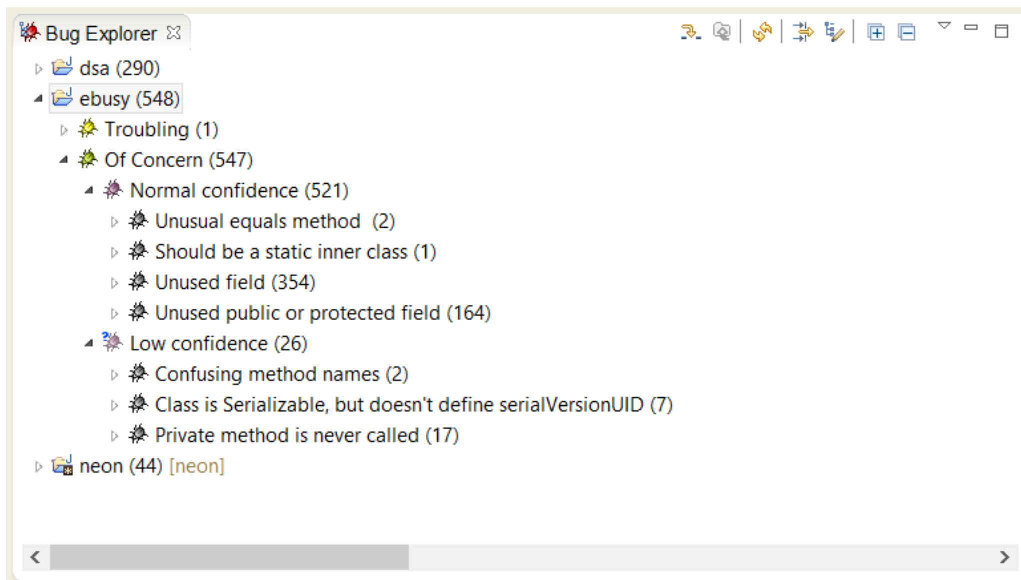


Abbildung 3 - Bug-Explorer-View von FindBugs

3.4. Maschinelles Lernen

3.4.1. Was versteht man unter Maschinellem Lernen?

Das Forschungsgebiet des Maschinellen Lernens beschäftigt sich mit dem Entwurf von Algorithmen, die auf Basis von Erfahrungen Wissen erwerben, um ein gegebenes Problem besser lösen zu können. Der Begriff Erfahrung bezieht sich hierbei auf Informationen aus der Vergangenheit, die dem Lernverfahren zugänglich gemacht werden, typischerweise in Form von Paaren von Ein- und Ausgabedaten, die gesammelt und für eine Analyse verfügbar gemacht wurden. Dies können unter anderem manuell erstellte Trainingsdaten sein, oder Daten, die in einer Interaktion mit der Umwelt erfasst wurden. Der Algorithmus versucht Gesetzmäßigkeiten in den Daten zu erkennen, um so für neue Eingabedaten akkurate Voraussagen treffen zu können.

Für alle Lernverfahren sind Qualität und Größe der Daten ein kritischer Faktor für den Erfolg der Voraussagen (Mohri, et al., 2012).

Die Verfahren des Maschinellen Lernens sind stark geprägt durch die verwendeten Daten und kombinieren grundlegende Konzepte der Informatik mit Ideen aus den Gebieten Statistik, Wahrscheinlichkeitstheorie und Optimierung (Mohri, et al., 2012). Der Großteil der Forschung auf dem Gebiet des Maschinellen Lernens konzentriert sich auf Ansätze, bei denen aus Beispielen gelernt wird. Dabei liegen die Eingaben in einer faktorisierten Darstellung (einem Vektor von Attributwerten) vor und die Ausgaben entweder als stetige numerische Werte oder als diskrete Werte. Es geht hierbei darum, aus einer gegebenen Menge von Ein-/Ausgabepaaren eine Funktion zu lernen, die die Ausgabe für neue Eingaben voraussagen kann. Sich nur auf diese Klasse von Lernproblemen zu konzentrieren, mag einem als Einschränkung erscheinen, tatsächlich aber besitzt diese Klasse ein sehr großes Anwendungsgebiet (Russell & Norvig, 2012).

Drei Arten des Feedbacks werden für die Unterteilung der drei Hauptarten des Lernens verwendet. **Nicht überwachtes Lernen** setzt keinerlei Feedback voraus. Hier versucht eine Maschine selbstständig, Muster in den Eingabedaten zu erkennen. **Das verstärkende Lernen** nutzt Belohnungen und Bestrafungen, um eine Maschine auf korrektes Verhalten oder Fehlverhalten hinzuweisen. Beim **überwachten Lernen** beobachtet die Maschine Eingabe-/Ausgabe-Paare und hat so Feedback gesammelt, mit dem sie versuchen kann, eine Funktion zu lernen, die Eingaben auf Ausgaben abbildet (Russell & Norvig, 2012).

3.4.2. Was sind Anwendungsgebiete des Maschinellen Lernens?

Warum ist es für spezielle Probleme notwendig, eine Maschine in die Lage zu versetzen, zu lernen? Kann man eine Maschine nicht von vorneherein mit genug Wissen ausstatten, dass sie ein gegebenes Problem zufriedenstellend lösen kann? Tatsächlich gibt es Probleme, die so komplex sind, dass es viel zu aufwändig oder schlichtweg nicht möglich ist, eine Funktion zu finden, die das Problem ausreichend genau beschreibt. Auch Änderungen, die sich über die Zeit ergeben, können unvorhersehbar sein (Russell & Norvig, 2012). Eine Maschine soll sich idealerweise auch in neuen und unbekannten Umgebungen autonom „zurechtfinden“ können.

Die Hauptziele in einer praktischen Anwendung von Maschinellem Lernen liegen einerseits darin, akkurate Voraussagen für noch nicht betrachtete Elemente geben zu können und andererseits im Entwurf von effizienten und robusten Algorithmen, die diese Voraussagen erstellen. Dabei treten eine Reihe von theoretischen Fragen auf, sowie Fragen, die die Umsetzung in einem konkreten Algorithmus betreffen. Fundamentale Fragen sind:

- *Welche Konzepte können überhaupt gelernt werden und unter welchen Bedingungen?*
- *Wie gut können diese Konzepte von einer Maschine gelernt werden?*

Verfahren des Maschinellen Lernens werden erfolgreich in vielen Anwendungsfeldern eingesetzt. Dazu gehören unter anderem:

- ***Text- oder Dokument-Klassifikation*** (z.B. Spam-Filter)
- ***Spracherkennung***
- ***Spiele*** (Schach, Backgammon)
- ***Medizin*** (für Diagnosen)
- ***Suchmaschinen***

In den Anwendungsfeldern treten verschiedene Arten von Lernproblemen auf, für die jeweils unterschiedliche Ansätze am geeignetsten sein können. Beispiele für auftretende Lernprobleme sind unter anderem (Mohri, et al., 2012):

- ***Klassifikation***: Einem Element soll eine Kategorie zugewiesen werden (z.B. Klassifikation von Bildern in Kategorien wie Landschaft, Portrait, Tiere, etc.)
- ***Regression***: Für jedes Element soll ein Realwert vorausgesagt werden (z.B. Aktienkurse)
- ***Ranking***: Die Elemente sollen anhand eines bestimmten Kriteriums geordnet werden (z.B. bei Suchmaschinen, für eine Anfrage alle gefundenen Seiten nach Relevanz anordnen)
- ***Clustern***: Die Elemente sollen in homogene Regionen eingeteilt werden (z.B. bei der Social-Network-Analyse, Gemeinschaften innerhalb einer großen Gruppe von Personen identifizieren)

3.4.3. WELCHE ANSÄTZE DES MASCHINELLEN LERNENS GIBT ES?

Es gibt heutzutage eine große Vielfalt an Ansätzen, um einer Maschine die Fähigkeit des Lernens zu ermöglichen. Zu bekannten Ansätzen des Maschinellen Lernens gehören unter anderem Entscheidungsbäume und künstliche neuronale Netze.

Ein **Entscheidungsbaum** wird mit einem gegebenen Beispieldatensatz, bestehend aus Paaren von Ein- und Ausgaben, erstellt. Die Eingaben liegen als Vektoren von Attributwerten vor. Jeder Knoten des Entscheidungsbaums repräsentiert einen Test des Werts eines der Eingabeattribute. Die abgehenden Zweige eines Knotens entsprechen den möglichen Werten des zugehörigen Attributs. Für eine neue Eingabe wird entlang der Wurzel bis zu einem Blattknoten eine Reihe von Tests auf ihren Attributen durchgeführt, an deren Ende die Ausgabe der von dem Entscheidungsbaum gelernten Funktion steht.

Künstliche neuronale Netze sind Sammlungen von Einheiten, den (künstlichen) Neuronen, die durch gerichtete Verknüpfungen miteinander verbunden sind. Die (künstlichen) neuronalen Netze sind eine Abstraktion der Aktivitäten in Netzen von Gehirnzellen. Die Eigenschaften eines neuronalen Netzes werden durch seine Topologie und die Eigenschaften der Neuronen bestimmt. Die Neuronen berechnen eine Eingabefunktion, die durch die Eingabewerte und Gewichte der Verbindungen zu anderen Neuronen bestimmt ist. Liegt der gewichtete Eingabewert über einem bestimmten (harten oder weichen) Schwellenwert, „feuert“ das Neuron und gibt die Ausgabe an die nächsten Neuronen weiter, mit denen es verbunden ist. Die Schwellenwertprüfung wird auch als Aktivierungsfunktion bezeichnet (Russell & Norvig, 2012). Die unterschiedlichen Attribute von Eingaben bewirken unterschiedliche Aktivierungen der Neuronen und damit unterschiedliche Ausgaben des Netzes bzw. der gelernten Funktion.

Welche Technik letztendlich am besten geeignet ist, hängt unter anderem auch davon ab, welches A-priori-Wissen die Maschine besitzt, welche Darstellung für die Daten verwendet wird und welches Feedback verfügbar ist, von dem gelernt werden kann (Russell & Norvig, 2012).

3.5. CASE-BASED REASONING

Das Schlussfolgern (**engl. Reasoning**) ist eine für uns Menschen wichtige Fähigkeit, weil sie uns intelligentes Handeln ermöglicht. Intelligenz ist nicht durch reine Reflex-Mechanismen gekennzeichnet, sondern durch Schlussfolgerungsprozesse, die mit dem gespeicherten Wissen arbeiten, das wir im Laufe unseres Lebens angesammelt haben (Russell & Norvig, 2012). Um ähnliche Schlussfolgerungsprozesse von einer Maschine durchführen zu lassen, muss das Schlussfolgern formalisiert werden. Mit Hilfe der Kalküle der **Logik** (die auch als die Wissenschaft des Schlussfolgerns bezeichnet werden kann) ist dies machbar. Sie zeigt uns, was gültige Schlussfolgerungen sind und gibt uns die Regeln vor, wie aus gegebenen Aussagen weitere Aussagen (neues Wissen) abgeleitet werden können.

Die Bestandteile einer logischen Schlussfolgerung sind einzelne logische Schlüsse (engl. Inferences). In den Kalkülen der **Logik** wird das deduktive Schließen angewendet, das einen Schluss von gegebenen Prämissen auf die logisch zwingenden Konsequenzen bedeutet. Andere Arten des Schlussziehens, die aber keine zwingenden, sondern nur mögliche Schlüsse darstellen, sind das induktive und abduktive Schließen. Aus dem Prinzip des logischen Schließens haben sich in der Informatik spezielle Verfahren im Forschungsbereich der künstlichen Intelligenz abgeleitet, die versuchen, aus einer vorhandenen **Wissensbasis** Schlussfolgerungen zu ziehen, um neues Wissen abzuleiten.

Bereits in den 70er Jahren wurden im Forschungsbereich der künstlichen Intelligenz verstärkt sogenannte regelbasierte **Experten-Systeme** eingesetzt. Diese Systeme versuchen, menschliches Expertenwissen in Form von Regelsätzen (Wenn-Dann-Beziehungen) für die maschinelle Verwendung verfügbar zu machen. Das Ziel ist es, dass eine Maschine durch Verwendung dieser Regelsätze in der Lage ist, Schlussfolgerungen zu ziehen und so neue Problemstellungen, genauso gut wie ein menschlicher Experte, lösen kann. Bestandteil eines Expertensystems ist eine sogenannte Inferenzmaschine, die ein automatisiertes Schließen (*automatic Reasoning*) auf den Daten der Wissensbasis durchführt. Solche Experten-Systeme werden in verschiedensten Anwendungsgebieten eingesetzt wie z.B.:

- *Medizinische Diagnostik*
- *Spracherkennung*
- *Erdbeben-Prognosen*

Die Konstruktion solcher Experten-Systeme kann sich, abhängig von der Problemstellung, sehr schwierig gestalten. Das Problem kann so komplex werden, dass ein sehr großer Regelsatz nötig ist, um es vollständig zu modellieren. Beschreibt der Regelsatz das Problem nicht vollständig, so können Fälle auftreten, in denen das Experten-System nicht in der Lage ist, für ein unvorhergesehenes Problem eine Lösung zu bestimmen.

Das Verfahren des sog. Case-Based Reasonings geht zurück auf die Arbeiten von Schank, Carbonell, Kolodner und Rissland (Schank, 1982) (Carbonell, 1983) (Kolodner, 1983) (Rissland, 1983). Es ermöglicht eine übersichtliche Konstruktion und ist in der Lage, durch Maschinelles Lernen auch unvorhergesehene Fälle zu behandeln. Beim Verfahren des Case-Based Reasonings wird versucht, menschliches Verhalten bei der Lösung von Problemen nachzuahmen. Ein Problem wird gelöst, indem ein Bezug zu ähnlichen Fällen hergestellt wird, für die eine Problemlösung bekannt ist.

Überträgt man diese Vorgehensweise auf ein Computerprogramm, so stellen sich zwei Aufgaben. Zum einen müssen Probleme und Problemlösungen in einer geeigneten Form gespeichert werden, wofür beim Case-Based Reasoning eine sog. Case-Base (Falldatenbank) verwendet wird. Zum anderen muss bei dem Versuch, ein neues Problem zu lösen, ein Abgleich mit der Case-Base stattfinden, um potentielle Problemlösung zu finden. In der Case-Base gespeicherte Datensätze stellen eine Kombination aus Problemstellung und Problemlösung dar.

Bei der Suche nach Problemlösungen in der Case-Base müssen Problemstellungen miteinander verglichen werden. Daher ist der Entwurf der Case-Base maßgeblich für den Erfolg eines solchen Systems. Maschinelles Lernen wird in einem solchen System durch das Feedback des Anwenders realisiert. Durch das Feedback wird ein Fall erstellt, der dem System neues Wissen bereitstellt. Dieses Wissen steht für künftige Problemlösungen bereit. Durch die immer größer werdende Case-Base wird das System stetig erweitert und verbessert.

4. Ursprünglicher Lösungsansatz

In diesem Kapitel beschreibe ich die ursprünglichen Ideen und Ansätze, die ich zur Umsetzung der Aufgabenstellungen in Betracht gezogen habe.

Zu Beginn meiner Diplomarbeit hatte ich die Aufgabe, zu prüfen, ob sich der Lernalgorithmus mittels Reasoning auf einer Ontologie durchführen lässt.

Der Begriff der Ontologie hat seinen Ursprung in der Philosophie als Teil der allgemeinen Metaphysik. In der Informatik gibt es für Ontologien viele verschiedene Definitionen. Eine der gängigsten lautet:

„An ontology is an explicit specification of an conceptualization“ (Guarino, et al., 2009)

Ontologien sind ein Mittel, um die Struktur eines Systems formal zu modellieren, dies sind die relevanten Entitäten (Einheiten) und Beziehungen, die aus der Beobachtung des Systems hervorgehen. Die Basis einer Ontologie besteht aus einer Generalisierungs-/Spezialisierungs-Hierarchie, einer Taxonomie (Guarino, et al., 2009). Mit Ontologien kann ein Netzwerk von Informationen mit logischen Relationen erstellt werden. Diese Tatsache und die Verfügbarkeit von Ableitungsregeln prädestinieren sie dafür, als Wissensbasis eines Reasoning-Systems zu fungieren. Eine Ontologie setzt sich zusammen aus:

- Einem Vokabular, das dazu verwendet wird, ein Anwendungsgebiet (eine Domäne) zu beschreiben oder evtl. auch eine bestimmte Sicht darauf
- Einer expliziten Spezifikation der beabsichtigten Bedeutung des Vokabulars. Diese beinhaltet meist, wie Begriffe klassifiziert werden
- Regeln, die zusätzliches Wissen über die Domäne erfassen

Da eine Ontologie die Dinge modelliert, wie sie sind, und nicht, wie sie sein könnten, sollte als Ergänzung, um die Unsicherheiten in meinem Anwendungsgebiet zu erfassen, die Ontologie um eine probabilistische Komponente erweitert werden.

Aber wie modelliert man Konzepte und Beziehungen, wenn man nicht sicher sagen kann, was relevant ist? Die einzigen Informationen, die für das Reasoning in meinem Anwendungsfall zur Verfügung stehen, sind die Kontextinformationen aus dem Source-Code. Der Reasoning-Prozess sollte anhand dieser Informationen zur Klassifikation von Warnmeldungen, von einem Bug und seinem Kontext auf die Ähnlichkeit zu einem anderen Bug und seinem Kontext schließen. Aber was sind relevante Konzepte? Wie stehen Konzepte bzw. Kontexte von Bugs in Beziehung? Auf diese Fragen konnte ich in der Literatur keine befriedigenden Antworten finden. Konzepte und Beziehungen ohne dieses Wissen explizit zu formalisieren, schien mir nicht umsetzbar.

Für die Implementierung der Ontologie war vorgesehen, die Topic-Maps Engine TinyTim zu verwenden. Für ein automatisches Reasoning auf Ontologien sind Topic-Maps aber nicht ausgelegt. Aufgrund all dieser Einschränkungen und Schwierigkeiten habe ich mich gegen diesen Weg entschieden.

5. LÖSUNGSANSATZ

Im Forschungsgebiet des Software Engineering gibt es kein allgemein anerkanntes Modell dafür, warum Programmierer Fehler bei der Erstellung von Software machen (Kim, et al., 2007). Kim et al. nehmen als Ursache eine Kette von Einbrüchen der geistigen Leistungsfähigkeit an. Sie gehen außerdem davon aus, dass Fehler nicht einzeln auftreten, sondern sich unter anderem zeitlich und in naher Umgebung von anderen Fehlern häufen. Sie sprechen hier von **Bug Locality**, die sich demnach als Temporal Locality oder Spatial Locality äußern kann (Kim, et al., 2007). Sie betrachten dabei die Fehler aus Sicht des Programmierers.

Ich betrachte in dieser Arbeit die potentiellen Fehler sowohl aus Sicht des Programmierers als auch aus Sicht des Werkzeugs, das sie aufdeckt, und gehe davon aus, dass sich Fehler in ähnlichen Kontexten wiederholen. Dies kann dabei sowohl am Programmierer liegen, der aufgrund seiner Programmiergewohnheiten immer wieder ähnliche Kontexte schafft, als auch am Werkzeug, das ständig in ähnlichen Situationen denselben Fallstricken in einer komplexen Analyse erliegt, wobei diese Situationen wiedererkennbar aber nicht benutzerspezifisch sind.

Als Grundlage der Realisierung eines Systems mit den im Kapitel „Aufgabenbeschreibung" genannten Anforderungen muss ein Weg gefunden werden, Bugs, beziehungsweise die Kontexte, in denen sie auftreten, zu vergleichen. Es gilt eine Ähnlichkeitsrelation zu definieren. Da FindBugs den Byte-Code von Java-Anwendungen analysiert, scheint es naheliegend zu sein, den Kontext von Bugs auch anhand von Byte-Code Informationen zu definieren. FindBugs hat aber eine große Hierarchie an Detektoren, die teilweise verschiedene und meist auch nicht alle Komponenten des Byte-Codes von Klassen berücksichtigen. Eine einheitliche Definition des Kontexts und damit eine generische Lösung für alle Detektoren (und Bug-Typen) ist somit ausgeschlossen, da es keine gemeinsame Grundlage an Informationen für die Detektoren gibt. Ich habe mich daher für einen anderen Weg entschieden, in dem ich dafür auf den Source-Code zurückgreife. Mit dem Source-Code habe ich auch die ganze Fülle an Informationen zur Verfügung, die der Entwickler in die Programmierung hat einfließen lassen (Kommentare gehen

beispielsweise bei der Übersetzung in Byte-Code verloren) und kann die Kontextinformationen für alle Bug-Typen auf einer einheitlichen Basis definieren.

Zu klären bleibt die Frage, welche Informationen zum Kontext eines Bugs gehören, anders ausgedrückt, wie groß ist die Kontextbreite eines Bugs? Ist es beispielweise von Belang zu wissen, wie die Klasse heißt, in der der Bug auftritt, welche Interfaces diese Klasse implementiert oder welche Klasse sie erweitert? Falls der Bug in einer Methode auftritt, ist es aufschlussreich zu wissen, wie die Methode heißt, welche Parameter die Methode besitzt oder wie die *Modifier* definiert sind? Es gibt eine unübersichtlich große Menge von potentiellen Einflussfaktoren. Noch erschwerend kommt hinzu, dass für jeden Programmierer die Kontextabhängigkeiten von Bugs anders aussehen können aufgrund unterschiedlicher Programmierstile. Somit ist der Kontext von Bugs benutzerabhängig. Nach meinem Wissen gibt es derzeit keine allgemein anerkannte Definition dafür, was relevante Kontextinformationen von Bugs sind.

Ich beschränke mich in meiner Arbeit auf Bugs, die innerhalb von Methoden auftreten, da sich bei diesen der Kontext meiner Vermutung nach am ehesten fast vollständig in der direkten Umgebung der Bugs befindet. Der Kontext ist damit leichter zu erfassen und der Ansatz kann zunächst mit diesen besser zu handhabenden Testfällen erprobt werden. Sollten sich vielversprechende Resultate zeigen, kann über eine Erweiterung und Anpassung der Methode für alle anderen Vorkommen von Bugs im Anschluss an meine Diplomarbeit nachgedacht werden.

Da der Kontext in Form von Source-Code-Fragmenten zur Verfügung steht, habe ich mich entschieden, auf Techniken der Clone-Detection (auf Syntax-Basis) zurückzugreifen, um gleiche und auch ähnliche Kontexte (inkonsistente Klone, (Juergens, et al., 2009)) erkennen und finden zu können.

Die Techniken, die in der Clone-Detection Anwendung finden, lassen sich gut anhand der Programm-Repräsentation, auf der sie arbeiten, unterscheiden. Text-, Token-, AST- und PDG[4]-basierte sind die gängigsten Repräsentationen. Die von Eclipse verwendete, interne Programm-Repräsentation ist der Abstrakte Syntax-Baum (AST). Ein AST abstrahiert für den Vergleich von Kontextinformationen unwichtige Unterschiede in den Formatierungen des Source-Codes (Whitespaces etc.), behält aber die hierarchische Beziehung der Code-Bestandteile bei. Aufgrund dieser Eigenschaften und der schon bestehenden Integration in Eclipse samt AST-Parser,

[4] Nach engl. Program Dependence Graph

verwende ich als Repräsentationsform des Kontexts von Bugs ebenfalls Abstrakte Syntax-Bäume.

Die Wahl des Lernverfahrens spielt für den Erfolg des Systems eine entscheidende Rolle. Ich setze hier die Idee des Case-Based Reasonings (fallbasiertes Schließen) um. Eine Wissensbasis, in der fortwährend neue Fälle abgelegt werden, angestoßen durch das Feedback des Benutzers (bzw. dessen Klassifizierungen), ermöglicht dem System im Idealfall eine immer besser werdende, eigenständige Klassifizierung von Bugs anhand ihres Kontexts. Es stehen hier im Laufe der Zeit immer mehr Vergleichsmöglichkeiten (Fälle) zur Verfügung und damit auch potentiell gleiche oder ähnliche, beziehungsweise wiederkehrende Bug-Kontexte. Das stetige Anwachsen der Wissensbasis mit Wissen über den Benutzer und seine Klassifizierungen stellt dabei den Lernprozess des Systems dar.

Ein Case-Based Reasoning scheint der am besten geeignete Ansatz für meine Arbeit zu sein, da es eine quantifizierbare Ähnlichkeit der Elemente voraussetzt. Das gilt auch für die meiner Arbeit zugrundeliegende Annahme, dass sich das Klassifizierungsproblem (Klassifizierungen von Bugs) auf das Finden ähnlicher Kontexte reduzieren lässt. Das Finden und Vergleichen von ähnlichen Kontexten beinhaltet, dass die Ähnlichkeit messbar ist.

Case-Based Reasoning (CBR) kann außerdem als Alternative zu Regel-basierten Experten-Systemen gesehen werden und ist besonders dann geeignet, wenn die Anzahl der Regeln, die benötigt würden, um das Experten-Wissen formal zu erfassen, nicht mehr handhabbar ist oder das Wissen (die Theorie) über die Domäne zu gering oder unvollständig ist. CBR wird auch oft in Bereichen angewandt, in denen das zugrundeliegende Modell, das für Problem-Lösungen benutzt wird, noch nicht besonders gut verstanden wird (Mantaras, 2001).

All das trifft auch auf meine Arbeit zu, denn dem System ist nie bekannt, welche Faktoren tatsächlich den Benutzer veranlasst haben, einen bestimmten Bug als falsch-positiv zu klassifizieren. Das Wissen darüber ist nicht unmittelbar verfügbar, sondern nur indirekt und unvollständig in dem Kontext der Bugs für das System zugänglich. Außerdem gibt es auch kein Modell, mit dem das System arbeiten und aus dem es ableiten könnte, unter welchen Umständen der Benutzer einen Bug als falsch-positiv klassifizieren würde. Automatisiert Regeln durch das System erstellen zu lassen, ist zudem eine große Herausforderung, wenn die relevanten Kontextinformationen nicht exakt definiert sind.

Die vielen Methoden des Maschinellen Lernens, die dafür gedacht sind, Funktionen zu lernen, wie neuronale Netze, schienen mir für meinen Ansatz weniger geeignet, denn diese setzen Trainingsdaten voraus, mit denen das System für das Lösen der zukünftigen Aufgaben trainiert bzw. konfiguriert und justiert werden kann. Im Falle der automatisierten Erkennung handschriftlicher Ziffern wurde beispielsweise mit der Datenbank des NIST (*United States National Institute of Science and Technologie*), die 60.000 ausgewertete Ziffernproben enthält, ein neuronales Netz mit 123.000 Gewichten, trainiert, das anschließend im Einsatz eine Fehlerrate von nur 1,6% aufwies (Russell & Norvig, 2012).
Trainingsdaten wie die des NIST stehen mir nicht zur Verfügung. Für Varianten in der Schreibweise von Ziffern mag so eine Datenbank wohl auch von großem Wert sein, denn bei lediglich 10 Grundelementen (die Ziffern von 0-9) erfasst diese Datenbank mit 60.000 Einträgen sicher eine ausreichende und repräsentative Menge an unterschiedlichen Schreibweisen.

Anders sieht es in meinem Anwendungsgebiet aus, da die Klassifizierungen benutzer- und kontextabhängig und somit auch nicht verifizierbar sind. Eine systematische Erfassung gestaltet sich so äußerst schwierig. Das Erstellen einer solchen Datenbank würde auch erst Sinn machen, wenn es vielversprechende, allgemein anerkannte Ansätze gäbe, die von Testdaten profitieren würden. Eine Datenbank zur Verwendung von selbstlernenden Algorithmen zur Erkennung falsch-positiver Warnungen von Werkzeugen der statischen Code-Analyse ist heute nicht verfügbar.

Diesen Nachteil kann das Case-Based Reasoning kompensieren, da es den Vorteil besitzt, direkt eingesetzt werden zu können, ohne dass die Case-Base bereits viele Fälle enthalten muss. Das Wissen wächst ab dem ersten Einsatz stetig an und damit auch die Wahrscheinlichkeit, Kontexte wiederzuerkennen. Im Gegensatz zu beispielsweise neuronalen Netzen, bei denen neue Testdaten eine Anpassung der Struktur des Netzes nach sich ziehen können, muss an der Struktur des Reasoners nichts weiter geändert werden, sobald er erst einmal implementiert ist.

Auf Basis aller vorangegangenen Überlegungen lassen sich nun die wesentlichen Komponenten des Systems definieren:

1. Eine Komponente, die Kontextinformationen von Warnmeldungen und den zugehörigen Bugs aus dem Source-Code in Form eines AST extrahiert

2. Eine Komponente, die Kontextinformationen vergleicht und einen korrespondierenden Ähnlichkeitswert ausgibt
3. Eine Komponente, die Klassifizierungen von Warnmeldungen anhand ähnlicher, bereits klassifizierter Fälle und deren Kontextinformationen vornimmt
4. Eine Komponente, die die zu einer Klassifizierung gehörenden Kontextinformationen und alle weiteren relevanten Daten speichert
5. Eine Komponente, die unter Verwendung der gespeicherten Daten Klassifizierungen wiederherstellen kann

Im folgenden Kapitel werde ich im Detail darauf eingehen, wie ich diese Komponenten und alle weiteren Bestandteile des Systems umgesetzt habe.

6. LÖSUNGSBESCHREIBUNG

Zwei wesentliche Aufgabenstellungen müssen gelöst werden. Zum einen müssen dem System bereits bekannte Bugs wiedererkannt und die Klassifizierung des Benutzers wiederhergestellt werden. Zum anderen muss das System ähnliche Kontexte von noch nicht bekannten Bugs erkennen und mit Hilfe des Case-Base Reasonings automatisch Klassifizierrungen vornehmen.

Zunächst wird der Aspekt des Wiedererkennens von bekannten Bugs und die Wiederherstellung der Klassifizierung beschrieben, danach der Aspekt des Erkennens von ähnlichen Kontexten und die automatische Klassifizierung im Case-Base Reasoning.

6.1. WIEDERHERSTELLUNG VON KLASSIFIZIERUNGEN

FindBugs analysiert den Byte-Code von Java-Anwendungen. Werden während der Analyse Bugs entdeckt, zeigt das Tool die zugehörigen Warnmeldungen, im Falle des Eclipse-Plugin von FindBugs, am Rand des Editors mit einem Problem-Marker an. Damit wird ein Mapping zwischen Byte- und Source-Code umgesetzt. Doch genau darin liegen das Problem und die Herausforderung der Aufgabe. Die Kompilierung in Byte-Code erfolgt nicht unmittelbar. Wenn der Benutzer sich entschließt, eine Warnmeldung als falsch-positiv zu markieren, danach Änderungen am Source-Code vornimmt und erneut eine Analyse von FindBugs startet, kann es sein, dass der Teil des Byte-Codes, der die Warnmeldung zuvor ausgelöst hat, nicht mehr da oder an einer anderen Stelle zu finden ist. Dann würde auch die Warnmeldung nicht mehr oder an einer anderen Stelle im Source-Code auftauchen.

Wenn die Warnmeldung an einer anderen Stelle auftaucht, kann FindBugs nicht unterscheiden, ob es sich um den identischen Bug handelt, oder um einen gleichen, der an anderer Stelle durch den gleichen Programmierfehler oder gleiche Umstände wiederholt wurde.

Grundsätzlich gibt es keine Gewissheit und es bleibt nur die Möglichkeit einer probabilistischen Lösung für das Problem.

Neben den Informationen wie Bug-Pattern, Bug-Typ und Kategorie des Bugs, verwendet FindBugs sogenannte Bug-Annotationen, um den Ursprung eines Bugs innerhalb des Source-Codes zu beschreiben. Ein Bug kann beispielsweise mit Informationen darüber versehen werden, in welcher Klasse er aufgetreten ist, innerhalb welcher Methode, welche Variable betroffen ist und in welcher Source-Line er gefunden wurde. Dazu würden entsprechend Class-, Method-, Field- und Source-Line-Annotationen verwendet. Diese Informationen wurden von mir dafür benutzt, um einen Wahrscheinlichkeitswert dafür zu berechnen, ob es sich bei einem von FindBugs entdeckten Bug um einen dem System bereits bekannten Bug handelt.

Angenommen, ein Bug stimmt sowohl im Bug-Pattern als auch in den Class-, Method- und Field-Annotationen mit einem Bug, der in einem vorherigen Analysevorgang gefunden wurde, überein, nur die Source-Line-Annotation sei unterschiedlich: Dann ist es sehr wahrscheinlich, dass sich entweder die ganze Methode innerhalb der Klasse durch Einfügen oder Löschen von Source-Code verschoben hat, oder der Programmierer überladende Methoden verwendet hat und ein gleichartiger Bug in einer Methode mit gleichem Namen auftaucht.

Im ersten Fall könnte man annehmen, dass es sich um den identischen Bug handelt und die Klassifizierung übernehmen, falls vorhanden. Im zweiten Fall könnte man schließen, dass beiden Bugs eine ähnliche Ursache zugrunde liegt und davon ausgehen, dass deswegen auch hier die Klassifizierung übernommen werden kann, falls eine vorliegt.

Jedoch sind Einfüge- und Lösch-Operationen nicht die einzigen Modifikationen am Source-Code, die eine Veränderung der Bug-Annotationen nach sich ziehen können. Auch ein Refactoring der Bezeichner von Klassen, Methoden und Variablen kommt in Frage.

Angenommen, ein Bug stimmt diesmal sowohl im Bug-Pattern als auch in den Method-, Field- und Source-Line-Annotationen mit einem bereits gefundenen Bug überein, nur die Class-Annotation sei unterschiedlich: In diesem Fall wäre es durchaus denkbar, dass der Unterschied durch ein Refactoring des Klassennamens zustande gekommen ist. Dies scheint generell wahrscheinlicher, als dass in einer anderen Klasse ein gleicher Bug an derselben Source-Line in Verbindung mit einer

Variablen und Methode desselben Namens auftritt. Daher dürfte auch hier die Klassifizierung des Bugs, falls vorhanden, übernommen werden.

Analoge Überlegungen lassen sich auf alle weiteren Kombinationen von Übereinstimmungen und Unterschiede in den Bug-Annotationen zweier Bugs anwenden. Um alle diese Fälle abzudecken, habe ich mich zunächst für eine Lösung mit einem Ähnlichkeitsvergleich der String-Repräsentationen aller Bug-Annotationen entschieden, um einen kumulativen Wahrscheinlichkeitswert zu berechnen. Dieser Wert sollte angeben, wie hoch die Wahrscheinlichkeit ist, dass es sich bei zwei in unterschiedlichen Analysen gefundenen Bugs tatsächlich um ein und denselben Bug handelt.

Für den String-Ähnlichkeitsvergleich habe ich die Definition der Levenshtein-Distanz[5], auch Editier-Distanz, verwendet. Sie ist eine einfache, intuitive und rein syntaktische Definition der Text-Ähnlichkeit. Präzise gesagt, gibt sie an, wie viele Einfüge-, Lösch- und Editier-Operationen minimal nötig sind, um zwei Strings anzugleichen. Die Distanz der Strings „Test" und „Text" beträgt beispielsweise 1. Entweder wird in dem String „Test" der Buchstabe „s" durch ein „x" ersetzt, oder in dem String „Text" wird das „x" durch ein „s" ersetzt.

In meiner Implementierung der Levenshtein-Distanz hatte ich berücksichtigt, dass die Distanz für die Ähnlichkeit zweier Strings eine immer größere Rolle spielt, je kürzer die Strings sind. So sind sich beispielsweise die Strings „Person" und „Personen" ähnlicher als die Strings „Bar" und „Tag", obwohl beide Paare zueinander die Distanz 2 aufweisen. Das erste Paar hat ein größeres Verhältnis von gemeinsamen zu unterschiedlichen Buchstaben. Wie in folgender Formel dargestellt, habe ich die Ähnlichkeit zweier Strings *s1* und *s2* mit Hilfe der Levenshtein-Distanz berechnet:

$$Similarity(s1, s2) = 1 - \frac{LevenshteinDist(s1, s2)}{Max(Length(s1), Length(s2))}$$

Mit dieser Formel kann die Wahrscheinlichkeit dafür berechnet werden, dass es sich bei zwei Bugs aus unterschiedlichen Analysen um den identischen Bug handelt. Die Wahrscheinlichkeit, dass es sich um dieselben Bugs handelt, ergibt sich als mittlere

[5] Benannt nach **Vladimir I. Levenshtein**

Ähnlichkeit der String Repräsentationen aller Paare von Bug-Annotationen desselben Typs.

Hatte ein Bug keine Annotation eines bestimmten Typs, die der zu vergleichende Bug besaß, so ging dafür trotzdem der Wert 0 mit in die Berechnung der Wahrscheinlichkeit ein. Der letzte Schritt war der Vergleich des berechneten Werts für die Wahrscheinlichkeit mit einem Threshold. Wurde der Threshold überschritten, wurden die Bugs von dem System als identisch betrachtet und etwaige Klassifizierungen übernommen bzw. wiederhergestellt.

Ich verwarf diesen Ansatz nach einer ersten Verwendung wieder, da ich vermutete, dass Unterschiede in den Bug-Annotationen, die durch ein Refactoring zustande kommen, nur einen sehr geringen Anteil ausmachen. Ich nahm auch an, dass dieser Ansatz nur für kleine, Refactoring bedingte Änderungen der Bezeichner geeignet ist, wie z.B. die Korrektur von Tippfehlern. Nur hier ist die Ähnlichkeit der Strings groß genug, um an eine Übereinstimmung zu denken. Bei geringeren Übereinstimmungen im mittleren Bereich können die Übereinstimmungen einfach nur daher rühren, dass die referenzierten Objekte und Methoden ähnliche Aufgaben erledigen, auf gleichen/ähnlichen Daten operieren usw. Dies ist nicht etwa die Ausnahme, sondern eher der Regelfall in der objektorientierten Programmierung. Ein Beispiel ist das Bezeichner-Paar „SelectionListener“ und „SelectionHandler“. Dies sind zwei häufig verwendete Namen für Objekte, die auf Selektionen eines Benutzers reagieren und die dazugehörigen Daten verarbeiten. Sie verwenden die gleichen Daten, was in ihren Bezeichnern zum Ausdruck gebracht wurde. Man sollte in diesem Fall aber nicht wegen einer großen Ähnlichkeit ihrer Bezeichner darauf schließen, dass der String „SelectionHandler“ durch ein Refactoring aus dem Namen „SelectionListener“ entstanden ist. Dass zwei Bezeichner dasselbe Präfix oder Suffix aufweisen ist also meiner Annahme nach eher selten ein Indiz dafür, dass es sich auch um denselben Bug handeln könnte.

Das zentrale Problem für das Wiedererkennen von Bugs ist die Verschiebung der Bug-Source-Lines: Also das Problem, dass Bugs, die in einer bestimmten Source-Line gefunden wurden und durch Editieren des Source-Files an eine andere Position verschoben werden.

FindBugs selbst verwendet einen MD5-Hashwert, den es aus den signifikanten Bug-Annotationen berechnet, als Identitätsbeweis für Bugs. Nur Bugs mit übereinstimmenden Hashs werden als identisch betrachtet. Da für die Berechnung auch die

Source-Line-Annotation verwendet wird, ist ein Wiederkennen von Bugs, die in anderen Source-Lines wieder auftauchen, für FindBugs nicht machbar.

Ich verwende in meiner Implementierung daher eine modifizierte Variante der Hash-Funktion, indem ich die Source-Line-Annotation nicht mit berücksichtige und stattdessen einen Hashwert des AST-Teilbaums, des Statements, das den Bug provoziert hat, verwende. Solange sich an den Bug-Annotationen (außer der Source-Line-Annotation) und dem Statement nichts geändert hat, ist mein System so in der Lage, einen Bug an jeder beliebigen Position innerhalb der Klasse wiederzuerkennen.

Sollte sich das Statement geändert haben, ist es sehr wahrscheinlich, dass FindBugs in einem erneuten Analysevorgang dort keinen Bug mehr finden wird.

Eine Einschränkung bzw. ein Nachteil dieses Ansatzes ist es, dass zwei gleiche Statements, die in derselben Methode auftreten, denselben Hashwert zugewiesen bekommen. Sollte also ein Fall auftreten, in dem zwei Bugs desselben Typs innerhalb einer Methode auftreten und durch zwei exakt übereinstimmende Statements hervorgerufen wurden, so würden diese Bugs stets dieselbe Klassifizierung aufweisen. Vermutlich tritt solch eine Situation sehr selten auf und noch unwahrscheinlicher ist es, dass diese Bugs unterschiedlich als falsch-positiv oder nicht falsch-positiv zu klassifizieren wären.

In den folgenden Abschnitten werde ich nacheinander die Aspekte des Ansatzes, den ich für den Lernalgorithmus gewählt habe, erklären, um anschließend eine Gesamtübersicht des Systems aufzeigen zu können.

Der Ansatz stützt sich auf zwei Definitionen für die Ähnlichkeit von abstrakten Syntax-Bäumen: Feature-Vektoren und Anti-Unifier.

6.2. Feature-Vektoren und Locality-Sensitive Hashing

Die Berechnung von Feature-Vektoren, auch charakteristische Vektoren genannt, ist eine bekannte Methode um einen Fingerprint (Hashwert) für einen AST zu berechnen. Feature-Vektoren erfassen die strukturellen Eigenschaften eines AST. Für

ihre Berechnung werden alle enthaltenen Knoten berücksichtigt. Die Berechnung von Feature-Vektoren ist ein summierendes Hashing. Für die Berechnung des Hashwerts eines AST-Knotens sind nur die Hashwerte seiner direkten Kind-Knoten nötig. Die Berechnung kann daher in linearer Zeit (in Abhängigkeit der Anzahl der Knoten) durchgeführt werden.

Wie Jiang et al. definiere ich den Feature-Vektor eines AST als einen Punkt im euklidischen Raum, wobei jede Stelle des Vektors die Vorkommen eines bestimmten Tree-Patterns innerhalb des Teilbaums zählt (Jiang, et al., 2007).

Als Tree-Pattern verwende ich den Knoten-Typ. Da ich mit dem Document-Object-Model (DOM) von Eclipse arbeite, sind die verschiedenen Knoten-Typen alle die, die im package *org.eclipse.jdt.core.dom.ast* zu finden sind. In der Basis-Klasse *ASTNode* sind die Konstanten für alle Knoten-Typen definiert. Das DOM von Eclipse kennt genau 83 unterschiedliche Knotentypen. Die von mir verwendeten Feature-Vektoren besitzen daher 83 Stellen, beziehungsweise 84, weil ich einen eigenen Knoten-Typ definiert habe, der innerhalb des Anti-Unifiers benötigt wird, worauf ich in dem entsprechenden Abschnitt noch genauer eingehen werde.

Ein Feature-Vektor entfernt definitionsbedingt alle Bezeichner, die in einem AST vorkommen, da die Bezeichner selbst keine strukturelle Information beinhalten. Aber auch eine weitergehende Abstraktion bestimmter Knotentypen wäre denkbar, zum Beispiel könnten alle Knoten-Typen, die Schleifen-Konstrukte darstellen, auf denselben Hashwert abgebildet werden (eine while-Schleife lässt sich auch als for-Schleife formulieren).

Wie Jiang et al. verwende ich die Feature-Vektoren und die euklidische Distanz zwischen ihnen, um mittels eines **Locality-Sensitive Hashing** ähnliche Vektoren und damit auch strukturell ähnliche Code-Fragmente effizient zu clustern. Die Code-Fragmente sind in meinem Anwendungsfall die Kontextinformationen von Bugs, für die ich ähnliche Kontexte finden und erkennen möchte.

Da die Feature-Vektoren nur die verwendeten elementaren syntaktischen Einheiten zählen, kann es passieren, dass auch vollkommen unterschiedliche Code-Fragmente geclustert werden. Dies kann bei Code-Fragmente passieren, bei denen die Anzahl der Vorkommen der elementaren syntaktischen Einheiten übereinstimmt, diese aber auf unterschiedliche Weise größere Einheiten (Statements, Blöcke, usw.) bilden. Das Clustern anhand der Feature-Vektoren ist daher in meinem System nur eine

Vorselektion möglichweise ähnlicher Kontextinformationen. Eine definitive Entscheidung über die Ähnlichkeit der Code-Fragmente übernimmt eine nachgeschaltete Einheit.

Für das von Jiang et al. vorgeschlagene Locality-Sensitive Hashing wird eine Hash-Funktion benötigt, die gewollt Kollisionen für Vektoren erzeugt, deren euklidische Distanz unter einem Threshold liegt. Bei der Konzeption der meisten heute bekannten und verwendeten Hash-Funktionen wird versucht, zu verhindern, dass Kollisionen bewusst erzeugt werden können. Dies würde ein enormes Risiko in der Kryptographie darstellen, dem Anwendungsfeld, in dem die meisten Hash-Funktionen ihren Ursprung haben. Das Locality-Sensitive Hashing nimmt also eine gewisse Sonderrolle unter allen Hash-Funktionen ein.

Definition des (p_1, p_2, r, c)-Sensitiven Hashings:
Eine Familie $\mathcal{F}$ von Hashfunktionen $h : \mathcal{V} \rightarrow \mathcal{U}$ wird $(p_1, p_2, r, c) - sensitiv$ $(c \geq 1)$bezeichnet, falls gilt, $\forall\, v_i, v_j \in \mathcal{V}$,

$$\begin{cases} \text{falls } \mathcal{D}(v_i, v_j) < r & \text{dann ist } Prob\left[h(v_i) = h(v_j)\right] > p_1 \\ \text{falls } \mathcal{D}(v_i, v_j) > cr & \text{dann ist } Prob\left[h(v_i) = h(v_j)\right] < p_2 \end{cases}$$

Es muss also gelten: Ist die Distanz der Vektoren v_i und v_j kleiner als r (Radius), so ist die Wahrscheinlichkeit, dass beide Vektoren auf denselben Hashwert abgebildet werden, größer als p_1. Umgekehrt muss die Wahrscheinlichkeit, dass beide Vektoren denselben Hashwert haben, kleiner als p_2 sein, falls die Distanz der Vektoren größer als ein konstant Vielfaches von r ist.

Datar et al. haben beispielsweise für die folgende Familie von Hash-Funktionen, die Abbildungen von Vektoren auf Integer-Werte vornehmen, gezeigt, dass sie locality-sensitive ist (Datar, et al., 2004).

.

$$\left\{ h_{\alpha,b} : \mathbb{R}^d \rightarrow \mathbb{N} \;\middle|\; h_{\alpha,b}(v) = \left\lfloor \frac{\alpha \cdot v + b}{w} \right\rfloor, w \in \mathbb{R}, b \in [0, w] \right\}$$

Die Hashwerte der Feature-Vektoren dienen in meinem System als Index auf eine Datenbank, die Feature-Vektoren mit denselben Hashwerten in denselben Clustern ablegt. Eine Query an die Datenbank mit einem Feature-Vektor als Parameter liefert

dann alle Feature-Vektoren, die in dem zum Hashwert des Vektors korrespondierenden Cluster liegen. Liefert die Query eine zu große Ergebnismenge, kann vorher noch eine n-nearest-Neighbor Search Routine durchlaufen werden.

6.3. ANTI-UNIFICATION

Als Methode für den Ähnlichkeitsvergleich von abstrakten Syntax-Bäumen benutze ich die Anti-Unification, dafür wird eine Distanz zwischen den Bäumen berechnet. Ich orientiere mich in meiner Implementierung an dem von Bulychev und Mineas vorgeschlagenen Ansatz, den sie in ihrem Clone-Detection Tool Clone-Digger umgesetzt haben (Bulychev & Minea, 2008). Die Anti-Unification wurde zuerst in den Arbeiten von Plotkin und Reynolds beschrieben (Plotkin, 1970) (Reynolds, 1970). In der Anti-Unification wird aus zwei zu vergleichenden Teilbäumen ein Anti-Unifier erstellt. Der Anti-Unifier ist eine Generalisierung der Teilbäume, präziser gesagt die spezifischste Generalisierung beider Teilbäume. Alle Gemeinsamkeiten bleiben erhalten, alle Unterschiede gehen in der Generalisierung verloren.

Knoten, die Wurzeln von Teilbäumen sind, die nicht übereinstimmen, werden während der Anti-Unification durch spezielle Platzhalter-Knoten substituiert. Mit den Substitutionen lässt sich dann eine Editier-Distanz als Ähnlichkeitsmaß für abstrakte Syntax-Bäume berechnen (anhand der Größe der substituierten Knoten bzw. der Größe der Teilbäume, deren Wurzeln die substituierten Knoten sind). Die Anti-Unification erfasst die strukturellen Unterschiede zwischen zwei Bäumen. Sie erlaubt beispielsweise die Ersetzung einer Variablen durch einen komplexeren Ausdruck, aber sie unterscheidet auch beispielsweise zwischen Funktionen mit verschieden großer Anzahl an Parametern (Bulychev & Minea, 2008).

Bulychev und Mineas bezeichnen den Anti-Unifier zweier Bäume (wie ASTs) als deren „Skelett“. Der Anti-Unifier einer Menge von Bäumen kann, ihrer Vorstellung nach, als das spezifischste Pattern, das zu jedem Baum in der Menge passt, gesehen werden. Für sie kann der Anti-Unifier daher dazu benutzt werden, den Durchschnittswert einer Menge von Bäumen zu speichern (Bulychev & Minea, 2008).

Diese Eigenschaften will ich mir zunutze machen. Angenommen man habe eine Menge von abstrakten Syntax-Bäumen, die alle ähnliche Kontextinformationen zu einem bestimmten Bug-Typ enthalten. Darüber hinaus könnten diese ASTs aber auch völlig irrelevante Informationen enthalten, die nicht zum Kontext eines Bugs, des bestimmten Bug-Typs, gehören. Meiner Annahme nach wird die Anti-Unification in der wiederholten Generalisierung der AST-Teilbäume alle irrelevanten Informationen eliminieren und am Ende nur die wichtigen Kontextinformationen, die in allen Teilbäumen enthalten sind, übrig lassen.

Um den Generalisierungsvorgang zu begrenzen und um dabei nicht zu viele Informationen zu verlieren, definieren Bulychev und Mineas eine Kostenfunktion (Bulychev & Minea, 2008). Um zu überprüfen, ob ein Anti-Unifier mit einem weiteren abstrakten Syntax-Baum zusammengeführt werden sollte, wird mit der Kostenfunktion ein dynamischer Kostenwert berechnet. Dabei wird die Größe der benötigten Substitutionen für den Anti-Unifier mit einem Faktor skaliert. Als Faktor verwenden sie einen Integer, der angibt, wie oft der Anti-Unifier schon mit anderen abstrakten Syntax-Bäumen in einer Anti-Unification generalisiert wurde. Anders ausgedrückt, auf wieviele Bäume dieser Anti-Unifier schon gepasst hat. Sie wollen damit zum Ausdruck bringen, dass sich ein Anti-Unifier mit der Anzahl der Matches immer weniger ändern sollte, da er schon ein Pattern angenommen hat, das für viele ASTs passt.

Die Anti-Unification wird mit einer simultanen Top-Down-Traversierung zweier zu vergleichender abstrakter Syntax-Bäume durchgeführt. Angefangen wird bei den Wurzelknoten und es wird überprüft, ob die Knoten matchbar sind. Dies hängt unter anderem davon ab, ob der Knotentyp und die Anzahl der Kind-Knoten übereinstimmen. Ist dies nicht der Fall, werden beide Knoten durch einen Platzhalter-Knoten ersetzt. Falls sie matchbar sind, wird rekursiv für die Kind-Knoten der beiden Knoten überprüft, ob sie matchbar sind. Wird ein Knoten eines AST durch einen Platzhalter ersetzt, werden dabei auch alle seine Kind-Knoten aus dem AST entfernt. Die Traversierung hat eine lineare Laufzeit in Abhängigkeit der Anzahl der Knoten der Bäume.

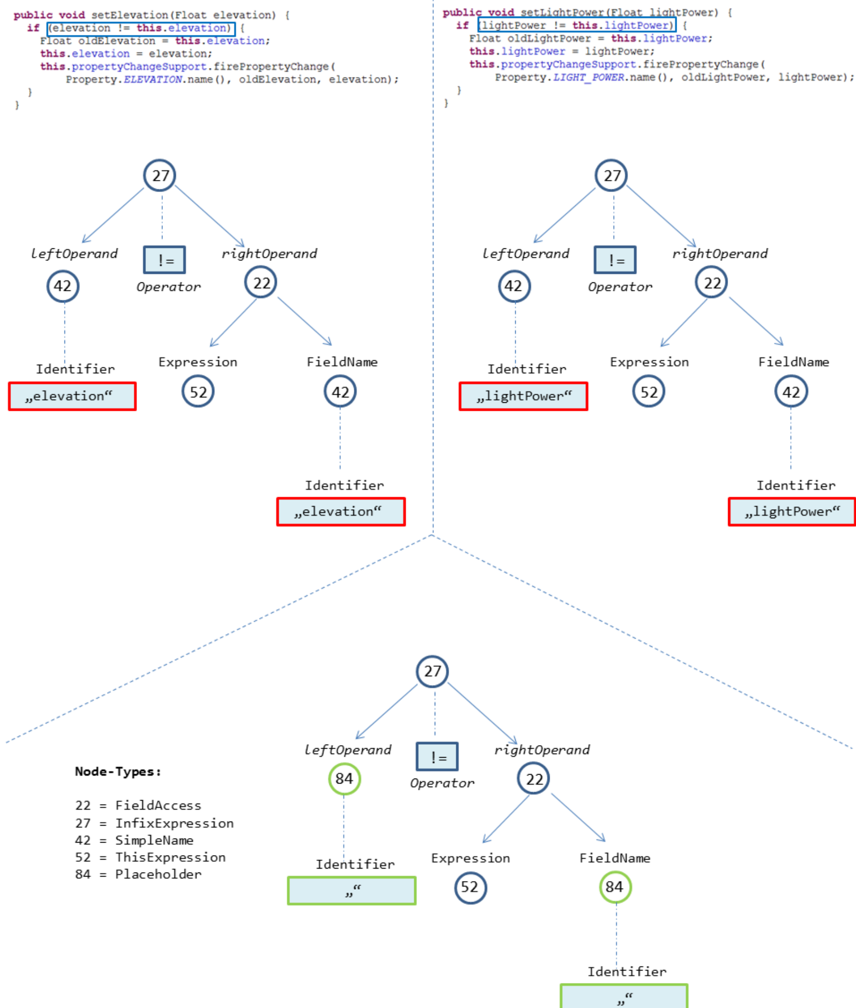
Eine Einschränkung bei der Verwendung der Anti-Unification ergibt sich durch ihre Rigorosität. Die Anti-Unification kann keine Unterschiede in der Anzahl der Kind-Knoten handhaben, oder einfache Vertauschungen der Kind-Knoten. Das spiegelt sich auch in dem von Bulychev und Mineas implementierten Tool Clone-Digger wieder. Das Tool sucht nach Sequenzen übereinstimmender IDs (in einem ersten

Durchlauf werden alle Statements mit der ID des am besten passenden Anti-Unifiers markiert) (Bulychev & Minea, 2008). Vertauschungen einzelner Statements oder Unterschiede in der Anzahl der Statements können hier zur Folge haben, dass gar keine Klone gefunden werden oder dass die von dem Tool gefundenen Klone Löcher aufweisen. Für das Finden exakter Klone ist dieses Verhalten sicher unproblematisch, eher sogar erwünscht.

Mit meinem System möchte ich mich jedoch auch auf die ähnlichen und nicht exakt übereinstimmenden Klone (die sich wiederholenden Bug-Kontexte) fokussieren. Daher verwende ich diese rigorose Definition der Anti-Unification nur auf Ebene einzelner Statements. Hier ist es meiner Ansicht nach für das Erkennen von Bug-Kontexten besonders wichtig, Übereinstimmungen in den Bezeichnern zu erfassen. Es könnte beispielsweise von dem System erkannt werden, dass ein Bug stets als falsch-positiv von einem Benutzer klassifiziert wird, wenn in einem der kurz vor der Bug-Source-Line auftretenden Statements eine bestimmte Bibliotheksfunktion aufgerufen wird.

In der Programmierung von Software ist es gang und gäbe, kurze Beschreibungen der Semantik bestimmter Code-Abschnitte in Bezeichner einfließen zu lassen. Ein Methodenaufruf wie *„getFactorial(n)"* verrät dem Betrachter, dass hier von der Methode höchstwahrscheinlich die Fakultät der Zahl *n* berechnet wird. So ist eine strukturelle Übereinstimmung zweier Statements mit zusätzlicher Übereinstimmung in den verwendeten Bezeichnern für die Kontextähnlichkeit sicher stärker zu gewichten als eine lediglich strukturelle Übereinstimmung.

Folgender Abschnitt zeigt an einem realen Code-Beispiel den Vorgang der Anti-Unification. Die if-Statements der beiden Methoden rufen bei einer Analyse mit FindBugs Warnmeldungen hervor, da hier Fließkomma-Werte nicht als primitive Datentypen, sondern als Float-Objekte mit dem Ungleich-Operator anstatt einer equals-Methode verglichen werden. Wie in der folgenden Abbildung dargestellt, erhält die Anti-Unification die Übereinstimmungen des AST-Teilbaums bei und eliminiert alles andere. Die Abbildung zeigt den Vorgang exemplarisch für den Teilbaum der if-Bedingung (einer InfixExpression). Wird dies analog mit allen anderen Komponenten der Methoden durchgeführt, so bleibt am Ende eine „Schablone" übrig, die für weitere solcher Fälle passend ist und zur Wiedererkennung herangezogen werden kann.



6.4. AUTOMATISCHE UND BENUTZERGESTEUERTE KONTEXT-SPEZIFIZIERUNG

Ein wichtiger Aspekt des von mir implementierten Systems ist es, in Verbindung mit einer vom Benutzer vorgenommenen Klassifizierung eines Bugs, den zugehörigen Kontext im Source-Code festzulegen. Im ersten Entwurf meines Systems und der zugehörigen Realisierung hatte ich noch den Ansatz verfolgt, zunächst die ganze Methode als Kontextinformation zu speichern, dies in der Annahme, dass nach einer

mehrfachen Anti-Unification nur noch die relevanten Kontextinformationen übrig bleiben.

In ersten Tests hat sich herausgestellt, dass dies nur auf kleine Methoden mit wenigen Statements überhaupt zutreffen könnte. Bei großen Methodenrümpfen geht die Ähnlichkeit der Bug-Kontexte schnell gegen einen sehr geringen Wert, bei dem man nicht mehr von einer Übereinstimmung reden kann, beziehungsweise bei der man nicht beurteilen kann, ob die geringe Ähnlichkeit nur durch irrelevante Statements bedingt ist. Ich habe diesen Ansatz daher wieder verworfen und verfolge nun einen anderen Weg, den Kontext von Bugs zu spezifizieren.

Mit meiner Anwendung kann die Spezifizierung des Bug-Kontexts auf zwei verschiedene Weisen geschehen, automatisch durch das System, oder durch eine Selektion eines Code-Abschnitts durch den Benutzer. Im ersten Fall nimmt der Benutzer zwar eine Klassifizierung vor, weil er aufgrund bestimmter Kenntnisse davon ausgeht, dass es sich um eine falsch-positive Warnmeldung über einen Bug handelt. Er kann sie in diesem Fall aber nicht auf einen bestimmten Kontext innerhalb des Source-Codes zurückführen. Daher muss hier das System eine bestmögliche eigenständige Spezifizierung des Kontexts vornehmen. Im Gegensatz dazu markiert der Benutzer im zweiten Fall eine oder mehrere Source-Lines innerhalb der Methode, in der der Bug aufgetreten ist, und setzt diese mit dem Bug in Beziehung (dies geschieht über eine Funktion im Kontext-Menü des Eclipse-Editors).

Die vom System eigenständig vorgenommene Kontextspezifizierung ist durch einen einzelnen Konfigurationsparameter gesteuert. Dieser gibt an, wie groß der zu extrahierende Kontext in Abhängigkeit von der Anzahl der Statements sein soll. Innerhalb eines Methodenrumpfs sind hauptsächlich die Statements die Teile des abstrakten Syntax-Baums, die die Kontextinformationen tragen. Eine Begrenzung der Anzahl der Statements begrenzt somit direkt den Informationsgehalt des Kontexts. Ich beziehe diese Anzahl jedoch auf alle in einem AST-Teilbaum enthaltenen Statements. Ein einzelnes Statement, gegeben durch einen AST-Knoten, kann weitere Statements in seinen Kind-Knoten enthalten (Schleifen, if-else-Konstrukte etc.), daher berechne ich die Kontextgröße für den gesamten AST-Teilbaum eines Statements.

Eine einfache Limitierung wie die Festlegung auf eine bestimmte Anzahl der durch den Kontext erfassten Source-Lines halte ich nicht für sinnvoll, da ich annehme, dass sie zu sehr vom Formatierungsstil des Source-Codes abhängt und die hierarchische Struktur des Source-Codes außer Acht lässt.

Die Bug-Source-Line muss sich stets innerhalb des automatisch erfassten Kontexts befinden. Das habe ich in dieser Weise festgelegt, in der Annahme, dass sich der Kontext eines Bugs im Allgemeinen fast vollständig in direkter Umgebung der Bug-Source-Line befindet.

Der von mir implementierte Algorithmus versucht stets, Statements auf der geringstmöglichen Verschachtelungstiefe als Kontext zu erfassen, ohne dabei den Grenzwert für die maximale Kontextgröße zu überschreiten. Diesem Vorgehen liegt die Annahme zugrunde, dass für den Großteil der Bugs, die innerhalb von Blöcken (if-else-Konstrukte, Schleifen, Try-Catch-Blöcke, etc.) auftreten, auch die Blöcke selbst relevante Kontextinformationen enthalten (z.B. die Bedingung bei einem if-Statement).

Bei der automatischen Erfassung des Kontexts wird die hierarchische Struktur innerhalb der Methode mit verschachtelten Anweisungs-Blöcken berücksichtigt. Es werden nur Statements erfasst, deren AST-Knoten alle denselben Parent-Knoten referenzieren, um zu verhindern, dass Statements aus benachbarten aber unterschiedlichen Anweisungs-Blöcken im Bug-Kontext zu direkten Geschwistern werden. Umgesetzt habe ich all diese Überlegungen mit einem Sliding-Window-Verfahren, das sich durch folgenden Pseudo-Code grob beschreiben lässt:

```
for (depth in 0 upto methodDepth) do

    for (statementsCount in MAX_CONTEXT_SIZE downto 1) do

        statements:= getMethodStatementsOfDepth(depth);

        for(index in 0 upto sizeof(statements)-statementsCount) do

            slidingWindow:=sublist(statements,index,index+
                                   statementsCount);

            if(coversBugLine(slidingWindow)and
             isEqualOrLessThanMaxContextSize(slidingWindow)and
            statementsAreSiblings(slidingWindow))then


                   return (context := slidingWindow);
```

Bei einer durch das System automatisch durchgeführten Kontexterfassung besteht natürlich immer die Gefahr, dass zu viele irrelevante bzw. zu wenig relevante Informationen mit in den Kontext aufgenommen werden, wenn für alle Bugs einheitlich

die Kontextbreite auf einen bestimmten Wert festgelegt wird. Um dieses Dilemma zu umgehen, bietet das System dem Benutzer die Möglichkeit, den Kontext eines Bugs selbst zu spezifizieren. Damit wird zusätzlich zu dem Selbstlern-Aspekt ein Trainingsmechanismus für das System etabliert. Es ist nicht nur darauf beschränkt, selbst zu lernen, welche Informationen als Kontext relevant sind, sondern es kann dies auch direkt durch den Benutzer mitgeteilt bekommen und dadurch trainiert und stetig verbessert werden.

6.5. Konvertierung und Speicherung von Kontextinformationen

Im Abschnitt *„Automatische und benutzergesteuerte Kontextspezifizierung"* habe ich gezeigt, wie das von mir entwickelte System den Kontext eines Bugs aus einem AST extrahiert. Für die Anti-Unification wird ein spezieller Knotentyp benötigt, ein Platzhalter, der als Ersatz für nicht passende Teilbäume eingefügt wird. Der Platzhalter erlaubt dann für zukünftige Anti-Unification Vorgänge einen gewissen strukturellen Freiheitsgrad für zu vergleichende abstrakte Syntax-Bäume.

Innerhalb des Eclipse DOM-Package (*org.eclipse.jdt.core.dom)* einen eigenen Knotentyp zu definieren, kam für mich nicht in Betracht. Dies hätte eine Reihe von Änderungen in Klassen des *Package* nach sich gezogen. Eclipse ist zwar ein quelloffenes Werkzeug, somit wären diese Änderungen möglich, doch damit würde mein Tool nur für Anwender benutzbar, die bereit sind, eine modifizierte Eclipse-Version zu benutzen.

Ein weiterer Grund ist die bei Eclipse AST-Knoten fehlende Kind-Relation. Die abstrakte Basis-Klasse *ASTNode* des Eclipse DOM-Package definiert lediglich eine Eltern-Relation über die Methode *getParent().*

Eine Top-Down Navigierung ist nur mit expliziter Kenntnis über den Knotentyp möglich. Ein Knoten vom Typ Block kennt beispielsweise die enthaltenen Statements und kann auf diese über die Methode *statements()* zugreifen. Um eine einfachere Top-Down Traversierung von Bäumen (in der Anti-Unification) zu ermöglichen, habe ich eine eigene Knoten-Klasse TreeNode erstellt, die zusätzlich eine Kind-Relation und eine entsprechende *getChildren()* Methode definiert. Die AST-Knoten von Eclipse

unterstützen zwar eine Top-Down Traversierung mit einem Visitor-Pattern basierten Ansatz, damit ließe sich der Anti-Unification Vorgang aber nur sehr umständlich formulieren.

Desweiteren wollte ich eine leichtgewichtigere und serialisierbare Version eines AST-Knotens definieren, um auch die Möglichkeit zu bieten, mit der Java Objekt-Serialisierung die Daten persistent speichern zu können. Viele Felder von Eclipse AST-Knoten sind aufgrund ihres Typs nicht serialisierbar. In meiner TreeNode Klasse werden daher alle relevanten Felder bei der Erstellung eines Objekts in die Java Basis-Typen konvertiert. Beispielsweise kennt die Klasse InfixExpression verschiedene Operatoren (&&, ||, !=, >=, …). Die Operatoren werden intern als eigene Objekte vom Typ Operator abgelegt. Diese Objekte sind nicht serialisierbar, aber für einen Vergleich zweier AST-Knoten vom Typ InfixExpression kann genauso gut auch die String-Repräsentation der Operatoren verwendet werden.

Analog kann für sämtliche AST-Knotentypen eine Übereinstimmung (Matching) auf den Vergleich von Basis-Typen zurückgeführt werden. Deshalb ist die konvertierte Version der AST-Knoten, die ich mit der Klasse TreeNode umsetze, für den Ähnlichkeitsvergleich völlig ausreichend. Es gibt überhaupt nur wenige AST-Knoten im Eclipse DOM-Package, die außer ihrem Knoten-Typ Eigenschaften besitzen, die für ein Matching relevant sind. Bei den meisten wird ein Matching auf ein Matching der Kind-Knoten zurückgeführt.

Aus den genannten Gründen habe ich mich entschieden, den Kontext in Form eines AST vor der Speicherung zu konvertieren. Um nun einen abstrakten Syntax-Baum zu konvertieren, wird dieser in meiner Anwendung mit einem ASTVisitor-Objekt traversiert. Für jeden besuchten ASTNode wird ein passender TreeNode mit allen relevanten Eigenschaften erstellt und die parent- und child-Referenz gesetzt. Die Klasse ASTNode definiert 83 Integer-Konstanten für alle AST-Knoten-Typen. Die Klasse TreeNode verwendet diese ebenfalls und definiert eine weitere 84. Integer-Konstante, die den für die Anti-Unification benötigten Platzhalter repräsentiert.

6.6. ÄHNLICHKEITSVERGLEICH VON KONTEXTINFORMATIONEN

Im Abschnitt *„Konvertierung und Speicherung von Kontextinformationen"* habe ich gezeigt, dass ein aus dem Source-Code extrahierter AST-Teilbaum, der den Kontext eines Bugs darstellt, vor der Speicherung und weiteren Verwendung durch das System konvertiert wird. Es wird ein korrespondierender Baum aus Objekten des Typs TreeNode erstellt. Im Abschnitt *„Automatische und benutzergesteuerte Kontextspezifizierung"* habe ich außerdem erläutert, dass die Anti-Unification nur auf Ebene einzelner Statements für den Ähnlichkeitsvergleich herangezogen wird. Es bleibt also zu klären, wie die vollständigen Kontexte von Bugs, die mehrere Statements (Sequenzen von Statements) enthalten können, verglichen werden.

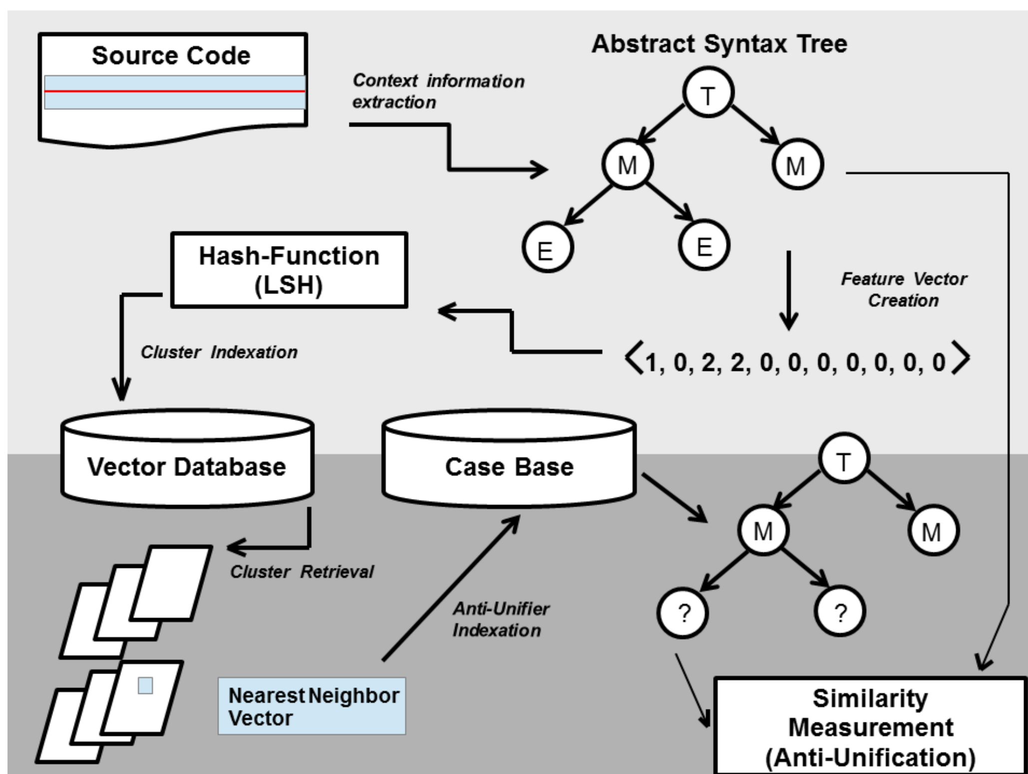
Die Ähnlichkeit von Sequenzen von Statements wird schlichtweg durch die Ähnlichkeit der einzelnen Statements berechnet. Es werden dabei nur Statements verglichen, die denselben Knoten-Typ aufweisen und damit matchbar sind. Der Vergleich der einzelnen Statements wird dann mit der Anti-Unification durchgeführt mit der Ausnahme für Statements, die Blöcke mit tiefer verschachtelten Statements enthalten können (Schleifen, if-else-Konstrukte, try-catch-Blöcke, etc.). Bei diesen Knoten-Typen wird die Ähnlichkeit rekursiv durch die Ähnlichkeit der Sequenz ihrer enthaltenen Statements berechnet.

Die Gesamt-Ähnlichkeit der Kontexte wird daher letzten Endes immer durch die Ähnlichkeit der einzelnen Statements bestimmt. Die Anti-Unification erzeugt während ihrer Durchführung eine Liste von Knoten, die durch Platzhalter substituiert wurden. Anhand dieser Substitutionen lässt sich der Gesamtwert für die Ähnlichkeit annähern. Es wird dafür die Anzahl der übereinstimmenden (nicht substituierten) Knoten der Kontexte im Verhältnis zur Gesamt-Anzahl der Knoten der Kontexte betrachtet. Für zwei Kontexte *c1* und *c2* lässt sich die Ähnlichkeit *Sim(c1, c2)* folgendermaßen definieren (*c1.subs* und *c2.subs* seien dabei die Listen der in der Anti-Unification substituierten Knoten für Kontext *c1* und *c2*):

$$Sim\ (c1, c2) = \frac{\big(size(c1) + size(c2) - size(c1.subs) - size(c2.subs)\big)}{size(c1) + size(c2)}$$

6.7. DER REASONING-VORGANG

Nachdem ich in den vorangegangenen Abschnitten jeden Aspekt des Systems einzeln beschrieben habe, fahre ich damit fort, deren Funktion und Zusammenspiel im Gesamtkontext des Case-Based Reasonings zu erklären.
Der Reasoning-Prozess orientiert sich an folgendem schematischem Ablauf:



Der Case-Based Reasoner wird durch das FindBugs Eclipse-Plugin am Ende eines Analysevorgangs aufgerufen und bekommt die Liste der von FindBugs gefundenen Bugs übergeben.
Der Case-Based Reasoner führt dann folgende Schritte für jeden Bug aus:

1. Hole AST der Klasse, in der der Bug auftritt
2. Überprüfe, ob der Bug in einer Methode auftritt
    a. Falls ja, fahre fort mit Schritt 3
    b. Falls nein, fahre fort mit dem nächsten Bug bei Schritt 1
3. Überprüfe, ob Bug wiedererkannt werden kann und ob der Bug bereits vom Benutzer klassifiziert wurde

   a. Falls ja, übernehme Klassifizierung aus der Datenbank
   b. Falls nein, fahre fort mit Schritt 4
4. Frage alle vorhandenen Kontextgrößen für Bugs desselben Typs aus der Datenbank ab
5. Extrahiere für alle Kontextgrößen alle möglichen Kontexte aus der Methode des Bugs
6. Berechne für jeden Kontext einen Feature-Vektor und frage Vektor-Datenbank nach Nearest-Neighbors ab
7. Benutze die Nearest-Neighbor Vektoren für eine Anfrage an die Case-Base
8. Berechne Ähnlichkeiten zwischen den Kontexten der Cases und den aus der Methode extrahierten Kontexten
   a. Sind die Kontexte nicht ähnlich genug, lasse den Bug unklassifiziert und fahre mit dem nächsten Bug bei Schritt 1 fort
   b. Andernfalls, fahre fort mit Schritt 9
9. Benutze die ähnlichen Kontexte zur Lösung des Klassifizierungsproblems des Bugs

Der Case-Based Reasoner arbeitet mit Kontexten, die aus dem Source-Code extrahiert werden. Der erste Schritt ist daher das Parsen des Source-Files mit einem AST-Parser, um an den AST der Klasse zu kommen. Dies muss natürlich für eine Menge von Bugs, die alle in derselben Klasse auftreten, nur einmal getan werden. Ich habe daher einen AST-Cache nach dem FIFO(First In, First Out)-Prinzip erstellt, der einen AST für ein Klasse solange vorhält, bis dieser in seinem internen Speicher durch andere ASTs, die später hinzugefügt wurden, ersetzt wird.

Der AST-Cache muss jedoch vor jedem Reasoning-Vorgang zurückgesetzt (geleert) werden, sonst würde die Gefahr bestehen, dass die FindBugs-Analyse und der Reasoner mit inkonsistenten Daten arbeiten, wenn zwischen Reasoning und einer FindBugs-Analyse die Source-Files bearbeitet wurden. Das würde einige Probleme wie ein falsches Mapping der Bug-Source-Line auf AST-Knoten zur Folge haben.

Wie ich im Kapitel „*Lösungsansatz*“ bereits erläutert habe, beschränke ich mich mit meinem System auf Bugs, die innerhalb von Methoden auftreten. Um dies in einem zweiten Schritt zu überprüfen, kann das von der Klasse *CompilationUnit* aus dem Eclipse-DOM Package umgesetzte Mapping zwischen AST-Knoten und Source-Lines

verwendet werden. Ein AST-Visitor traversiert dafür den AST der gesamten Klasse und bei jedem Besuch eines Knotens des Typs *MethodDeclaration* wird getestet, ob sich die Bug-Source-Line innerhalb der Source-Range des Methoden-Rumpfs befindet.

Auf die in der Case-Base abgelegten Fälle (Cases) von Bug-Klassifizierungen kann auf zwei verschiedene Weisen zugegriffen werden. Ich habe hier zwei Arten der Indizierung etabliert, um ein schnelles Wiedererkennen von Bugs zu ermöglichen. Eine Indizierungsart verwendet Hash-Werte der Feature-Vektoren und ist nur für das Finden von ähnlichen Bug-Kontexten gedacht. Die zweite Indizierungsvariante benutzt die Hashwerte der Bugs. Im Abschnitt *„Wiedererkennung von Bugs und Wiederherstellung von Klassifizierungen“* habe ich beschrieben, wie ich eine modifizierte Variante der von FindBugs implementierten Hashwert-Berechnung verwende, um Bugs unabhängig von ihrer Source-Line innerhalb der Klasse wiederzuerkennen.

Diese Hashwerte dienen zur Indizierung der Cases in der Case-Base. Will der Case-Based Reasoner nun im dritten Schritt überprüfen, ob es für einen Bug schon eine Klassifizierung gibt, frägt er an, ob die Case-Base einen Case für den Hash des Bugs gespeichert hat. Falls nicht, nimmt das System an, der Bug wurde noch nicht klassifiziert und versucht durch ein Schließen von ähnlichen Fällen auf diesen Fall das Klassifizierungsproblem zu lösen. Falls es einen Case gibt, kann die Klassifizierung anhand der gespeicherten Daten wiederhergestellt werden.

Da mein System wie im Abschnitt *„Automatische und benutzergesteuerte Kontext-spezifizierung“* dargestellt, dem Benutzer die Möglichkeit bietet, den Kontext von Bugs selbst festzulegen, wird der Reasoning-Vorgang ab Schritt 4 wesentlich komplexer als es ohne diese Möglichkeit nötig wäre. Angenommen, es gäbe diese Möglichkeit für den Benutzer nicht und man würde sich auf eine einfache Definition des Kontexts, wie beispielsweise fünf Source-Lines direkt vor der Bug-Source-Line festlegen, dann könnte der Reasoner einfach für einen Bug den entsprechenden Kontext aus dem AST auslesen und ihn mit den Kontexten in der Case-Base vergleichen.

Gibt man dem Reasoner dagegen keinerlei Informationen wonach er suchen soll, so bleibt ihm nur ein Brute-Force Ansatz mit der Überprüfung jedes möglichen Kontextes. Die Obergrenze für die Anzahl der möglichen Kontexte ist die Fakultät der Statements der Methode. Diese Obergrenze wird zwar nie erreicht, weil mein System nur aufeinanderfolgende Statements mit demselben Parent-Knoten als Kontext

erlaubt, trotzdem könnte diese erschöpfende Suche zu einem Problem für die Effizienz des Reasonings werden.

Ich habe mich daher entschieden, in die Case-Base zusätzliche Informationen über die enthaltenen Kontexte zu speichern. Wenn der Reasoner also vor der Aufgabe steht, nach Kontexten in der Methode zu suchen, frägt er vorher bei der Case-Base an, welche verschiedenen Größen es für gespeicherte Kontexte zu Bugs desselben Typs gibt (gemessen in Anzahl der Statements). Wenn er sich nur auf diese Kontextgrößen in der Suche beschränkt, kann dies den Aufwand deutlich reduzieren.

Der Reasoner wird nun in Schritt 5 Sliding-Windows dieser Kontextgrößen nacheinander über die Statements der Methode in jeder Verschachtelungstiefe schieben und diese als mögliche Kontextinformationen betrachten, falls sie eine Reihe von Kriterien erfüllen. Dazu zählen, dass die Knoten der Statements Geschwister sein müssen, die Gesamtzahl aller durch den Kontext erfassten Statements unter einem Threshold liegt und der Kontext vor der Bug Source-Line beginnen muss. Erfüllt der betrachtete Kontext die Kriterien, wird in Schritt 6 ein Feature-Vektor für diesen Kontext berechnet.

Der Abschnitt *„Feature-Vektoren"* hat gezeigt, dass die Feature-Vektoren die Vorkommen aller Knoten-Typen in einem AST zählen. Sie erfassen damit strukturelle Eigenschaften von abstrakten Syntax-Bäumen. Ich verwende sie an dieser Stelle, um eine gewisse Vorauswahl an potentiell ähnlichen Kontexten zu treffen. Dass Kontexte ähnliche Feature-Vektoren aufweisen ist zwar keine Garantie dafür, dass zwischen ihnen eine syntaktische Übereinstimmung besteht, es ist aber eine Voraussetzung dafür.

Mit dem Feature-Vektor des extrahierten Kontexts wird eine Anfrage an die Vektor-Datenbank nach dessen nächsten Nachbarn veranlasst. Dafür wird zwischen allen in Frage kommenden Vektoren (Vektoren mit demselben Hashwert) paarweise die euklidische Distanz berechnet.

In Schritt 7 wird jeweils mit den Hashwerten der Feature-Vektoren (nächste Nachbarn) eine Anfrage an die Case-Base geschickt. Jeder dieser Feature-Vektoren referenziert genau einen Case. Jeder Case enthält die Kontextinformationen des Bugs, der mit dem Case klassifiziert wurde. Der in Schritt 5 gefundene Bug-Kontext wird dann, wie im Abschnitt *„Ähnlichkeitsvergleich von Kontextinformationen"* beschrieben, in

Schritt 8 mit dem Kontext des Cases verglichen. Der Ähnlichkeitswert muss über einem Threshold liegen, damit das System den Kontext als übereinstimmend erkennt. Einer Besonderheit hierbei ist, dass dieser Threshold mit einem Faktor skaliert wird, der davon abhängt, wie oft der Kontext des Cases in einem Reasoning-Vorgang mit anderen Kontexten gematcht wurde. Bei jedem Matchen mit einem neuen Kontext kann sich der Kontext des Cases durch die Anti-Unification, die während des Ähnlichkeitsvergleichs durchgeführt wird (siehe Abschnitt *„Anti-Unification"*), verändern. Diese Veränderungen zu beschränken, um keine relevanten Informationen zu verlieren, ist der Gedanke hinter diesem skalierten Threshold. Je öfter der Kontext des Cases gematcht wurde, desto größer wird der beschriebene Threshold. Das heißt, dass jeder neue Match eine immer größere Übereinstimmung mit dem Kontext des Cases aufweisen muss. Das bringt zum Ausdruck, dass ein Kontext, der schon in vielen Fällen gepasst hat, nicht mehr verändert werden sollte, weil er scheinbar eine immer wiederkehrende Situation darstellt und schon alle irrelevanten Informationen in der Anti-Unification entfernt bekommen hat.

Schritt 9, der letzte Schritt des Reasoning-Vorgangs, muss mit den vorher erfassten Informationen eine Lösung für das Klassifizierungsproblem des Bugs finden. Es stellt sich die Frage, ob der Bug analog zu den ähnlichen Fällen klassifiziert werden sollte. Reicht also eine, über einem definierten Threshold liegende Ähnlichkeit zu einem anderen Kontext als Begründung für die Übernahme der Klassifizierung aus?

Bei einer Festlegung des Kontextes durch den Benutzer gehe ich davon aus, dass er diesen so wählt, dass er charakteristisch und ausreichend für eine Klassifizierung des entsprechenden Bugs ist. Anders sieht es aus, wenn es um die von dem System automatisch erfassten Kontexte geht. Bei diesen besteht natürlich keine Garantie, dass sie auch relevante Informationen für eine Klassifizierung erfassen. Dass bestimmte Code-Fragmente immer wieder in der Umgebung von Bugs auftreten, die der Benutzer auf eine bestimmte Weise klassifiziert hat, kann man lediglich als Hinweis auf eine potentielle Relevanz werten.

Eine einfache und in meinem System umgesetzte Lösung des Problems besteht darin, den Bug auf die gleiche Weise zu klassifizieren wie der ähnlichste gefundene Fall (Case) aus der Case-Base.

7. EVALUATION

Eine Gruppe von drei Teilnehmern (Studenten und wissenschaftliche Mitarbeiter der Universität Stuttgart) hat im Rahmen einer Evaluation die Funktionalität des von mir erweiterten FindBugs Eclipse-Plugin getestet und dessen Nutzen bewertet.

Für die Evaluation wurde das erweiterte FindBugs Eclipse-Plugin, der zu analysierende Source-Code der frei verfügbaren Software SweetHome3D und ein Fragebogen bereitgestellt. Es wurden keine Vorkenntnisse der Teilnehmer in der Verwendung von FindBugs oder des Eclipse-Plugins von FindBugs vorausgesetzt. Die Teilnehmer bekamen vor Beginn der Evaluation eine Einführung in die Verwendung von FindBugs mittels des erweiterten Eclipse-Plugins. Das Plugin wurde mit der aktuellen Version Eclipse Standard 4.3 verwendet.

Mit dem erweiterten Eclipse-Plugin haben die Teilnehmer den Java Programm-Code des Innenraum-Planers SweetHome3D[6] auf Fehler untersucht und diese klassifiziert. Folgende Tabelle gibt anhand einiger Code-Metriken eine Übersicht über Größe und Struktur der untersuchen Software.

SweetHome3D (Version 4.1) - Übersicht

METRIK	WERT
Anzahl Packages	10
Anzahl Klassen	587
Anzahl Methoden	4801
Methoden LOC (Lines of Code)	59635
Gesamt LOC	82439

Nachfolgende Tabellen zeigen die Anzahl der von FindBugs gefundenen Bugs im Programm-Code von SweetHome3D in Abhängigkeit unterschiedlicher Konfigurationen, unterteilt nach den Rängen der gefundenen Bugs. Die Teilnehmer verwendeten ausschließlich die Standard-Konfiguration von FindBugs.

[6] http://www.sweethome3d.com

FindBugs Analyse: SweetHome3D (Vers. 4.1)

Configuration: (Min. Rank = 20, Min. Confidence = Low, Detectors = All, Analysis-Effort = MAX)

Rank	Total	High Confidence	Normal Confidence	Low Confidence
Scariest	39	39	0	0
Scary	28	2	10	16
Troubling	35	12	19	4
Of Concern	2186	32	170	1984
Sum	**2288**	**85**	**199**	**2004**

Configuration: (Min. Rank = 20, Min. Confidence = Medium, Detectors = Default, Analysis-Effort = Default)

Rank	Total	High Confidence	Normal Confidence	Low Confidence
Scariest	39	39	0	0
Scary	12	2	10	0
Troubling	31	12	19	0
Of Concern	197	32	165	0
Sum	**279**	**63**	**194**	**0**

Default Configuration: (Min. Rank = 15 , Min. Confidence = Medium, Detectors = Default, Analysis-Effort = Default)

Rank	Total	High Confidence	Normal Confidence	Low Confidence
Scariest	39	39	0	0
Scary	12	2	10	0
Troubling	30	12	18	0
Of Concern	10	10	0	0
Sum	**91**	**63**	**28**	**0**

Zur Bewertung des erweiterten Eclipse-Plugins beantworteten die Teilnehmer nach der Anwendung die folgenden Fragen:

1. *Konnte das System Bugs, die in ähnlichen Kontexten auftreten, erkennen und selbstständig klassifizieren?*

Antworten: Einstimmige Meinung der Teilnehmer ist, dass dies nur bedingt gelang. Einige, sich ähnelnde Kontexte wurden wiedererkannt, bei anderen, mit einer vergleichbaren Ähnlichkeit geschah dies nicht. Dabei war für die Teilnehmer nicht ersichtlich, warum die Wiedererkennung in diesen Fällen nicht erfolgreich war.

2. *In welchen Situationen gelang das Erkennen von Bugs mit ähnlichen Kontexten nicht?*

Antworten: Die Teilnehmer konnten die Situationen nicht auf bestimmte Merkmale des Source-Codes zurückführen, lediglich die Bug-Patterns der von ihnen begutachteten Bugs wurden als Hinweise genannt. Eine Beschreibung der Situationen war für die Teilnehmer nur schwer möglich, da für sie die Wiedererkennung nicht nachvollziehbar war.

3. *Was sind Ihrer Meinung nach relevante Kontextinformationen für die Klassifizierung von Bugs und lässt sich eine allgemeine Aussage für alle Bug-Typen treffen?*

Antworten: Die Teilnehmer nannten den Methodenaufbau und Informationen über die Art der Anwendung, beispielweise ob es sich um eine Multithreading-Anwendung handelt, als relevante Informationen. Übereinstimmend wurde genannt, dass sich keine allgemeinen Aussagen für alle Bug-Typen treffen lassen.

4. *Traten Situationen auf, in denen Sie Ihre Klassifizierung nicht auf einen bestimmten Abschnitt im Source-Code zurückführen konnten?*

Antworten: Bei einem Teilnehmer trat eine solche Situation auf.

5. *Traten während der Anwendung des Systems Fehler auf?*

Antworten: Die Teilnehmer beobachteten keine Programmabstürze oder sonstige unerwartete Fehler in der Programmausführung.

6. *Wieviel Erfahrung haben Sie mit der Programmiersprache Java?*
7. *Wieviel Erfahrung haben Sie mit FindBugs?*

Antworten: Alle Teilnehmer haben Erfahrung mit der Programmiersprache Java und bereits eigene Projekte durchgeführt. Alle Teilnehmer hatten FindBugs bereits zuvor verwendet. Einer der Teilnehmer beschäftigt sich in eigenen wissenschaftlichen Arbeiten mit FindBugs.

8. *Wie zufrieden waren Sie mit der Bedienbarkeit des Systems?*

Antworten: Die Teilnehmer bewerteten die Bedienbarkeit überwiegend mit „verbesserungsfähig". Einer der Teilnehmer war zufrieden. Kritikpunkte waren nicht verstandene Dialog-Meldungen, eine nicht vorhandene Möglichkeit, Klassifizierungen rückgängig zu machen und ein fehlendes Gruppieren von wiedererkannten Bugs. Das Festlegen des Kontextes durch eine Markierung von Source-Code war für einen der Teilnehmer umständlich und nicht intuitiv umgesetzt.

9. *Würden Sie solch ein erweitertes System in der Praxis einsetzen?*

Antworten: Einer der Teilnehmer würde solch ein System auf jeden Fall einsetzen. Alle anderen Teilnehmer gaben an, dies unter Umständen zu tun.

10. *Glauben Sie, das System ermöglicht Ihnen eine Zeitersparnis bei der Arbeit mit FindBugs?*

Antworten: Die Teilnehmer gaben überwiegend an, dass Ihnen das System in der vorliegenden Fassung, keine oder nur marginale Zeitersparnisse bei der Arbeit mit FindBugs ermöglicht. Nur einer der Teilnehmer rechnet mit einer deutlichen Zeitersparnis.

11. *Wie bewerten Sie allgemein den Nutzen eines solchen selbstlernenden Systems?*

Antworten: Die Teilnehmer sprechen einem solchen selbstlernenden System einen deutlichen Nutzen zu. Einer der Teilnehmer geht von einem sehr großen Nutzen aus.

12. *Haben Sie sonstige Anmerkungen?*

Antworten: Einer der Teilnehmer gab an, dass das System in der aktuellen Fassung noch eine ständige manuelle Kontrolle der automatisch klassifizierten Bugs erfordert, um sicherzugehen, dass keine wirklichen Bugs „maskiert“ werden.

Auswertung: Diese Evaluation kann wegen der begrenzten Personenanzahl kaum als repräsentativ angesehen werden. Dennoch liefert sie einige wichtige Spotlights.

Zunächst hat sich gezeigt, dass das System noch nicht ausgereift ist. Durch die Analyse der nicht erfolgreichen Fälle konnten einige Programmierfehler aufgedeckt werden, die die Effektivität des Systems beeinträchtigt haben. Als Bewertung für die Effektivität des Ansatzes an sich, ist die vorliegende Evaluation daher nicht geeignet. Nach einem Code-Review mit anschließenden Korrekturen, war das System bei einem Großteil der vorher nicht wiedererkannten Kontexte erfolgreich. In einer weiteren Evaluation würde daher die Effektivität des Systems sicher besser bewertet werden.

In der Evaluation kam ein Prototyp einer Software zum Einsatz, der aufgrund des Zeitrahmens einer Diplomarbeit vorher nicht ausreichend getestet werden konnte.

Allerdings konnte die Evaluation zeigen, dass ein generelles Interesse an solch einer Software besteht. Auch wurde die Annahme, dass sich Bugs anhand ihres Kontexts wiedererkennen lassen, weitestgehend bestätigt. Auch die Annahme, dass sich keine allgemeinen Aussagen zur Relevanz von Kontextinformationen machen lassen, wurde von den Teilnehmern bekräftigt.

Da es ein Prototyp ist, kann man keine Bedienbarkeit erwarten, wie bei einem ausgereiften System. Generell sank die Akzeptanz des Systems aufgrund der Unzufriedenheit mit der Bedienbarkeit.

In der Verwendung des Systems hat sich bei den Teilnehmern ein gewisses Misstrauen gegenüber solch selbstlernenden und den Benutzer „bevormundenden" Systemen gezeigt. Die Teilnehmer wollten die Schlussfolgerungen, die das System zieht, bzw. das Wiedererkennen, nachvollziehen. Die Teilnehmer hätten erwartet, angezeigt zu bekommen, auf Basis welcher Kontextinformationen eine Ähnlichkeit berechnet wurde. Sie gaben an, dass nur damit der Aufbau eines Vertrauens gegenüber solch einem System möglich wäre. Sonst würde dem System etwas „Orakelhaftiges" anlasten. Diese Transparenz des Systems erscheint mir relevant und ist ein sinnvoller Ansatz für weiterführende Arbeiten.

8. WEITERFÜHRENDE ARBEITEN

Die wichtigste Voraussetzung für eine Weiterführung des Projekts, das mit dieser Diplomarbeit begonnen wurde, ist ein umfangreicher Software-Test. Die Software ist noch im Zustand eines Prototyps. Sobald das System einen „stabilen" Zustand erreicht hat, kann damit begonnen werden, das System im produktiven Einsatz zu erproben.

Ein erster Ansatz für eine weiterführende Arbeit wäre, zu untersuchen, wie sich das Locality-Sensitive Hashing mit unterschiedlichen Konfigurationsparametern auf das Clustern von (ähnlichen) Kontexten auswirkt. Es muss ein Kompromiss gefunden werden, zwischen einem zu großen Radius (vgl. Abschnitt 6.2. Feature-Vektoren und Locality-Sensitive Hashing), um große Kontexte mit nur relativ kleiner Übereinstimmung zu clustern und einem zu kleinen Radius, der nur strukturell exakt übereinstimmende Kontexte clustert. Im Falle eines zu großen Radius besteht die Gefahr, dass zu viele Kontexte für Vergleiche ausgewählt werden, was wiederum die Performanz merklich beeinträchtigen könnte. Dies sollte ebenfalls untersucht werden. Im Falle eines zu kleinen Radius wäre das System nur in der Lage, Bug-Kontexte wiederzuerkennen, die sich nur in den Bezeichnern unterscheiden, oder solche, die durch copy&paste wiederholt wurden.

Die Berechnung der Ähnlichkeit von Kontexten bietet ebenfalls viel Spielraum für Optimierungen. Auf unterster Ebene (Code, der keine Verschachtelungen enthält) wird ein Vergleich mittels Anti-Unification durchgeführt, auf den restlichen Ebenen wird eine modifizierte Variante der Anti-Unification verwendet. In beiden Fällen wird die Ähnlichkeit jedoch anhand der Menge der substituierten Knoten ermittelt. Dieser Wert kann als Basiswert, bzw. erster Näherungswert betrachtet werden. Es sind noch eine Reihe von Anpassungen denkbar, die sich mit einer Skalierung des Basiswerts umsetzen lassen würden. Die verwendete Definition (vgl. die Formel in Abschnitt „Ähnlichkeitsvergleich von Kontextinformationen") für die Ähnlichkeit betrachtet lediglich die Übereinstimmung in der Gesamtzahl der Knoten, vernachlässigt aber Übereinstimmungen einzelner Statements. Angenommen zwei Paare von Kontexten hätten nach genannter Definition denselben Ähnlichkeitswert, doch bei einem der

Paare wurden nicht alle Statements gematched. Dann wäre es durchaus nachvollziehbar, das Paar, bei dem nicht alle Statements gematched werden konnten, als weniger ähnlich zu betrachten, da die entfernten Statements relevante Kontextinformationen enthalten können.

Es wäre auch vorstellbar, die Positionen der gematchten Statements zu berücksichtigen. Denn eine bestimmte Sequenz von Statements behält durch Vertauschung einzelner Statements nur dann ihre ursprüngliche Semantik, wenn keinerlei Abhängigkeiten zwischen den vertauschten Statements bestehen. Abhängigkeiten zwischen Statements können aber theoretisch immer bestehen, daher ist in einer Sequenz die Position jedes einzelnen Statements potentiell charakteristisch für den Kontext. Ein Paar von Kontexten, bei dem zwar alle Statements gematched wurden, aber keines der gematchten Statements an derselben Position innerhalb der jeweiligen Sequenz auftritt, könnte daher als sich weniger ähnlich angesehen werden als ein Paar, bei dem die gematchten Statements an derselben Position auftreten.

Es besteht generell die Gefahr, dass zwei Kontexte lediglich aus denselben syntaktischen Einheiten aufgebaut sind. Dass nur Übereinstimmungen in Statements sehr unterschiedlicher Positionen gegeben sind, könnte ein Hinweis darauf sein. Eine stärkere Gewichtung von Übereinstimmungen in Statements der gleichen Position scheint daher für mich aufgrund der vorangegangen Überlegungen gerechtfertigt und könnte in einer Weiterentwicklung des Systems durchgeführt werden.

Im Abschnitt „Anti-Unification" bin ich bereits auf die Bedeutung der Übereinstimmung von Bezeichnern für die Ähnlichkeit von Kontexten eingegangen. Wiederkehrende Aufrufe bestimmter Bibliotheksfunktionen könnten beispielsweise charakteristisch für Bug-Kontexte sein. Verwendet der Benutzer in seinen Projekten bestimmte Frameworks, so zeigt sich das in den Bezeichnern von Variablen, Methoden und Klassen. Vieles spricht dafür, Übereinstimmungen in Bezeichnern stärker zu gewichten. Denkbar wäre es, dies noch in Abhängigkeit der Länge der Bezeichner zu tun. Die gängigen Bezeichner **i** und **j** für Schleifenindizes findet man sicher in vielen Kontexten, ohne dass dies ein Hinweis auf eine große Ähnlichkeit der Kontexte wäre.

Der Abschnitt „Der Reasoning-Vorgang" hat gezeigt, dass ein Bug analog zu dem ähnlichsten Fall klassifiziert wird. Bei einer automatischen Extraktion des Kontextes kann es passieren, dass der Kontext nur irrelevante Informationen enthält. Wenn das

System anhand dieser irrelevanten Informationen Ähnlichkeiten von Kontexten berechnet, können falsche Klassifizierungen die Folge sein. Der Benutzer sollte daher für jede vom System automatisch vorgenommene Klassifizierung angezeigt bekommen, anhand welcher Kontextinformationen dies geschah. Sieht er, dass der gespeicherte Kontext nur irrelevante Informationen enthält, sollte er die Möglichkeit haben, den entsprechenden Case aus der Case-Base zu löschen. In der aktuellen Fassung des Systems wurde beides aus Zeitgründen nicht berücksichtigt. Der Benutzer kann lediglich den automatisch erfassten Kontext eines Cases mit einem selbstgewählten Kontext überschreiben.

Der Reasoning-Vorgang ist bisher beschränkt auf Bugs, die innerhalb von Methoden auftreten. In einer Erweiterung des Systems wäre es denkbar, alle anderen Vorkommen von Bugs zu berücksichtigen. Für diese neuen Fälle muss dann eine eigene Definition eines Standard-Kontextes gefunden werden (vgl. Abschnitt „Automatische und benutzergesteuerte Kontextspezifizierung“). Auch die Verfahren zur Erfassung des Kontextes (automatisch und durch den Benutzer veranlasst) müssen dafür überarbeitet werden.

9. FAZIT

9.1. KRITISCHER RÜCKBLICK AUF DEN VERLAUF DER ARBEIT

Die erste Hälfte der Bearbeitungszeit dieser Diplomarbeit war geprägt von Literatursuche und –studium und einer Einarbeitung in das Werkzeug FindBugs. Die Funktionsweise von FindBugs zu analysieren nahm einen sehr großen Anteil der Zeit in Anspruch. Das Werkzeug ist über die Jahre seit seiner Initiierung zu einem großen open-source Projekt geworden. Allein das Core-Projekt FindBugs hat über 100.000 Zeilen Code.

Zu den Grundlagen der statischen Code-Analyse sowie der künstlichen Intelligenz gibt es gute und vielfältige Literatur, hier bestand wenig Schwierigkeit, einen Einstieg in die Thematik zu finden. Im Gegensatz dazu gestaltete sich eine Suche nach guter Literatur zu dem Werkzeug FindBugs schwierig. Die meisten wissenschaftlichen Veröffentlichungen verweisen zurück auf die Arbeiten der Entwickler von FindBugs. Deren Beschreibungen sind wertvoll, um einen Überblick zu bekommen, was FindBugs ist und was es kann. Die zugrundeliegenden Ideen werden gut erklärt. Steht man jedoch vor der Aufgabe, FindBugs zu erweitern, muss man sich in viele Konzepte selbständig einarbeiten. Außer dem Java-Doc ist nur sehr wenig an Dokumentation verfügbar. Das Architekturdokument ist veraltet (dieses Dokument bezieht sich immer noch auf die Version 0.9.4 von FindBugs) und lediglich in Fließtext verfasst. Es gibt keine UML-Diagramme, die ein Verständnis erleichtern würden. So ist man gezwungen, sich durch ein intensives Studium des Source-Codes eine Übersicht über das System zu verschaffen. Auch hier gab es einige Hürden. Die Untergliederung der Klassen in Pakete ist teilweise verwirrend und nicht nachvollziehbar, große Mengen von Klassen, zwischen denen kein direkt ersichtlicher Zusammenhang besteht, liegen direkt im Wurzelverzeichnis des Projekts.

Es gelang mir noch vor Ablauf der ersten Hälfte der für diese Arbeit verfügbaren Zeit, den Entwurf für das Gesamt-System fertigzustellen. Diesen konnte ich in einem

Zwischenvortrag meinem Prüfer, meinem Betreuer und interessierten Studenten präsentieren. Nach positiver Rückmeldung stand die zweite Hälfte der Diplomarbeit im Zeichen der Umsetzung und Integrierung des gewählten Ansatzes in das Eclipse-Plugin von FindBugs und dem Verfassen dieser Ausarbeitung. Für die Umsetzung musste ich mich mit der Plugin-Programmierung mit Eclipse vertraut machen. Hilfreich dabei waren sowohl die verfügbare, gute Literatur (erwähnt sei an dieser Stelle (Clayberg & Rubel, 2009)) als auch die professionelle Dokumentation von Eclipse.

9.2. INHALTLICHES FAZIT

Mit der vorliegenden Arbeit befasste ich mich mit dem Problem der hohen falsch-positiv Raten bei Werkzeugen der statischen Code-Analyse. Ich habe einen möglichen Ansatz zur Lösung des Problems aufgezeigt. Exemplarisch habe ich diesen Ansatz in einer Erweiterung des Eclipse-Plugins von FindBugs umgesetzt.

Ich hatte die Aufgabe, mit meinem Ansatz zu prüfen, ob sich ein selbstlernender Algorithmus dazu verwenden lässt, die Klassifizierungen eines Benutzers zu lernen, um diese automatisch im Sinne des Benutzers durchführen zu lassen.

Dies ist ein neuartiger Ansatz, denn bisher beschriebene Problemlösungen zielen hauptsächlich darauf ab, die Analysemethoden der Werkzeuge zu verbessern und so deren falsch-positiv Rate zu senken. Beide Wege, das Problem anzugehen, ergänzen sich jedoch gut. Ansätze, wie der von mir verfolgte, unterdrücken zwar lediglich die Warnmeldungen für gefundene falsch-positive Bugs, haben jedoch den Vorteil, dass sie universell (unabhängig von Art der Analyse und unabhängig vom verwendeten Werkzeug) anwendbar sind. Die Werkzeuge werden ständig erweitert und neue Detektoren (im Falle von FindBugs), die neue Analysemethoden anwenden, werden hinzugefügt. Ein davon unabhängiger Ansatz bietet somit einen klaren Vorteil.

Ein anderer Ansatz, um Warnmeldungen aufgrund bestimmter Kriterien zu unterdrücken, ist beispielsweise der von Shen et al. verfolgte Weg (Shen, et al., 2011). Sie führen eine Sortierung der Bugs nach aufsteigender falsch-positiv Wahrscheinlichkeit durch. Die Berechnung der Wahrscheinlichkeiten findet in zwei Phasen statt. Anhand einer manuellen Durchsicht der FindBugs-Warnmeldungen

großer open-source Software legen sie zunächst für jedes Bug-Pattern einen Default-Wert fest, der angibt, mit welcher statistischen Wahrscheinlichkeit ein Bug dieses Patterns falsch-positive Warnmeldungen verursacht. Die Bugs werden in Phase 1 anhand dieser Default-Werte sortiert. In Phase 2 werden durch Klassifizierungen des Benutzers und anhand einfacher Korrelationen zwischen Bugs desselben Patterns und teilweise auch desselben Typs, die Wahrscheinlichkeiten angepasst.

Wird man vor die Aufgabe gestellt, einen selbstlernenden Algorithmus zu entwerfen, so muss man sich, unabhängig von dem konkreten Anwendungsgebiet, mit grundlegenden Fragen des Maschinellen Lernens auseinandersetzen. Was ist beispielsweise von einem Algorithmus erlernbar und wie kann man einem Algorithmus die Bedeutung von Daten vermitteln? Kann sich ein Algorithmus selbst-adaptiv sich verändernden Situationen anpassen, oder verlangt dies immer nach einem Eingreifen durch einen Programmierer? Während der Durchführung meiner Diplomarbeit stellte ich fest, dass sich das Vorgehen von Menschen beim Treffen komplexer Entscheidungen (Bug-Klassifizierungen in meinem Anwendungsfall), die außerdem von vielen Einflussfaktoren abhängen, nur sehr schwer in mathematische Funktionen „gießen" lässt bzw. mit Hilfe logischer Terme ausgedrückt werden kann, auf Basis derer ein Algorithmus Schlussfolgerungen ziehen kann. Ich habe in meiner Lösung daher einen anderen, aber ebenfalls menschliches Verhalten nachahmenden Ansatz verwendet. Eine Entscheidung (Schlussfolgerung) wird auf eine andere Entscheidung zurückgeführt, die in einem ähnlichen Kontext getroffen wurde.

Es hat sich gezeigt, dass dieser Ansatz prinzipiell vielversprechend ist.

DANKSAGUNGEN

Ich bedanke mich bei Herrn Prof. Dr. Stefan Wagner und Herrn Dipl.-Ing. Jan-Peter Ostberg für die Betreuung dieser Diplomarbeit.

Meinen Eltern, Dr. Rainer und Marianne Schneider, bin ich zutiefst dankbar für deren aufopfernde Mühe bei der Durchsicht und Korrektur dieser Ausarbeitung.

Ganz besonderer Dank gilt auch meinem Bruder Wolfgang Schneider, der mir in vielen Situationen mit konstruktiver Kritik beiseite stand.

LITERATURVERZEICHNIS

Ayewha, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W. (2008): Experiences Using Static Analysis to Find Bugs. **IEEE Software**, pp. 22-29.

Binkley, D. (2007): Source Code Analysis: A Road Map. In: Proc. **Future of Software Engineering** (FOSE '07), IEEE Comp. Society, pp. 104-119.

Bulychev, P. Minea, M. (2008): Duplicate code detection using anti-unification. In: Proc. **Spring Young Researchers Colloquium on Software Engineering**, pp. 51-54.

Carbonell, J. G. (1983): Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In: **Machine Learning: An Artificial Intelligence Approach.** Morgan Kaufmann Publishers, pp. 137-161.

Clayberg, E., Rubel, D. (2009): **Eclipse Plug-ins**. Addison-Wesley.

Datar, M., Immorlica, N., Indyk, P., Mirrokni, V. S. (2004): Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In: Proc. **20th annual symposium on Computational geometry** (SCG '04), ACM, pp. 253-262.

Dillig, I., Dillig, T., Aiken, A. (2008): Reasoning About the Unknown in Static Analysis. In: **Communications of the ACM**. 53(8):115-123.

Guarino, N., Oberle, D., Staab, S. (2009): What Is an Ontology?. In: **Handbook on Ontologies**. pp. 1-17. Springer, Berlin.

Hovemeyer, D., Pugh, W. (2004): Finding Bugs is Easy. In: **ACM SIGPLAN Notices**, 39(12):92-106.

Jiang, L., Misherghi, G., Su, Z., Glondu, S. (2007): DECKARD: Scalable and Accurate Tree-based Detection of Code-Clones. In: Proc. **29th Intern. Conf. on Software Engineering** (ICSE '07), IEEE Computer Society, pp. 96-105.

Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S. (2009): Do Code Clones Matter?. In: Proc. **31st Intern. Conf. on Software Engineering** (ICSE '09), IEEE Comp. Society, pp. 485-495.

Kim, S., Zimmerman, T., Whitehead, E. J. J., Zeller, A. (2007): Predicting Faults from Cached History. In: Proc. **29th Intern. Conf. on Software Engineering** (ICSE '07), IEEE Comp. Society, pp. 489-498.

Kolodner, J. L. (1983): Reconstructive memory: A computer model. In: **Cognitive Science,** Vol. 7, pp. 281-328.

Mantaras, R. L. d. (2001): Case-Based Reasoning. In: **Machine Learning and Its Applications**. Volume 2049 of LNCS, pp. 127-145. Springer, Berlin.

Mohri, M., Rostamizadeh, A., Talwalkar, A. (2012): **Foundations of Machine Learning**. The MIT Press, Cambridge, Massachusetts.

Nielson, F., Nielson, H. R., Hankin, C. (2005): **Principles of Program Analysis.** Springer, Berlin.

Plotkin, G. P. (1969): A note on inductive generalization. In: **Machine Intelligence , Vol. 5**, Edinburgh University Press.

Reynolds, J. (1970): Transformational systems and the algebraic structure of atomic formulas. In: **Machine Intelligence, Vol. 5**. Edinburgh University Press.

Rissland, E. L. (1983): Examples in legal reasoning: legal hypotheticals. In: Proc. of the 8th **Intern. joint Conf. on Artificial Intelligence,** Vol. 1, Morgan Kaufmann Publishers, pp. 90-93.

Russell, S., Norvig, P. (2012): **Künstliche Intelligenz - Ein moderner Ansatz**. Pearson, München.

Schank, R. (1982): **Dynamic Memory: A theory of learning in computers and people**. Cambridge University Press.

Shen, H., Fang, J. & Zhao, J. (2011): EFindBugs: Effective Errror Ranking for FindBugs. In: Proc. **4th Intern. Conf. on Software Testing, Verification and Validation** (ICST '11), IEEE Computer Society Press, pp. 299-308

Socher, R. (2008): **Theoretische Grundlagen der Informatik**. Carl Hauser, München.

Wagner, S. (2011): Empirische Analyse von Fehlermusterwerkzeugen. In: Proc. 4. **Symposium Testen im System und Software-Life-Cycle**, Technische Akademie Esslingen.