Statische Analyse

Holger Hans Peter Freyther

31. März 2006

Die Softwareentwicklung scheint die einzige Ingenieur-Wissenschaft zu sein wo ein hohes Maß an Defekten gesellschaftliche Akzeptanz erlangt hat. Für Software Entwickler sollte dies keine befriedigende Situation darstellen. Erschwerend kommt die scheinbare Fehleranfälligkeit des Menschen hinzu.
Die statische Analyse verspricht, eine gewisse Anzahl von Fehlern zu finden und somit die Anzahl der Defekte in Produkten zu senken. Dies geschieht durch die systematische Analyse des Quelltextes und oder des Programmecodes und dem Suchen nach bekannten fehler- und fehleranfälligen Mustern. Hier beschäftigen wir uns mit zwei Beispielen mit der genaueren Funktionsweise, der Klasse erkennbarer Fehler und der Bedeutung der statischen Analyse um die Fehlerhäufigkeit zu senken.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Rahmen des Seminares zu "Ursachen und Vermeidung von Fehlern in der Softwareentwicklung" im Wintersemester 2005/2006 an der Freien Universität zu Berlin beschäftigen wir uns mit dem Entstehen von Fehlern und deren Vermeidung.
Bei der statischen Analyse gehen wir davon aus, dass bereits Fehler beim Schreiben des Quelltextes gemacht worden sind. Die Gründe dafür sind vielfältig. "Finding Bugs is Easy" [8] nennt die Komplexität moderner Programmiersprachen als Hauptgrund, aus früheren Veranstaltungen kommen Denkfallen, menschliches Versagen und die Arbeitsumgebung ([6, 5, 3]) als mögliche Gründe hinzu.
Die Motivation der statischen Analyse Methode ist daher, die bereits gemachten Fehler so früh wie möglich im Entwicklungsprozess zu erkennen. Im besten Fall bevor der Entwickler seine Änderungen in ein Verwaltungssystem einpflegt oder bei dem täglichen Bauen und Testen des Systems. (Unternehmen mit vorhandenen Strukturen werden vorrausgesetzt und diese sollten tatsächlich über Quellcode Verwaltungssysteme und automatisches Bauen und Testen verfügen [15]).

1.2 Grenzen des Möglichen

Wünschenswert wäre ein Werkzeug, das sich unsere Quellen anschaut, erkennt was wir meinen und die logischen Fehler und Implikationen – wie z.B fehlende Skalierung, zu hoher Speicherverbrauch, zu langsame Verarbeitung – feststellt und Lösungsvorschläge zum Beheben dieser Probleme anzeigt.
Ein Werkzeug, das erkennt, dass der Code mit zehn oder einer Million Datensätzen, oder einem, oder hunderten von Benutzern zu recht kommt und das auch noch über Bibliotheksgrenzen hinweg zuverlässig arbeitet, ist aus heutiger Sicht unrealitisch. Für diesen Zweck werden Beschreibungs-Sprachen wie die Uniform Modeling Langauge (UML) verwendet um Benutzungsweise und Anforderungen formal zu beschreiben.
Man sollte sich im Klaren sein, dass die hier betrachtete Form der computerisierten statischen Analyse keinen Korrektheits-Beweis darstellt. Es können also nur Fehler gefunden werden, die Abwesenheit von Meldungen beweist nicht die Abwesenheit von Defekten.

1.3 Statische Analyse

Unter den Techniken zum Auffinden von Fehlern können wir zwischen der dynamischen und statischen Analyse unterscheiden. Bei der dynamischen Analyse werden durch Testfälle und speziellen Programmkonstrukten, wie z.B. Assertions zur Laufzeit die Funktion des Systems getestet. Es besteht die Gefahr, dass bestimmte Teile des Systems mangels geeigneter Testfälle niemals getestet werden.
Die statische Analyse hingegen leidet nicht unter diesem Problem. Das System – der Quelltext oder ByteCode – wird direkt betrachtet. Die Königsdisziplin der statischen Analyse ist der formale Korrektheits-Beweis. Dies ist jedoch sehr zeitaufwendig, kompliziert und kann zur Zeit nur durch Menschen durchgeführt werden und wird daher hier nicht betrachtet – zu dem ist der Prozess und die Funktionsweise gut verstanden [7]. Auf der anderen Seite des Spektrums liegen die sogenannten Style-Checker. Hierbei wird zeielenweise bzw. blockweise der Stil des Programmes untersucht. Ein berühmter Repräsentant dieser Kategorie ist lint [9]. Auch diese Art von Analyse findet echte Probleme in Systemen.
Jedoch werden hier Werkzeuge betrachtet, die ein etwas tieferes Verständnis der Programmiersprache und der Semantik haben, jeodch nicht auf der Ebene des vollständigen Programmes mit all seinen möglichen Zuständen und Effekten, sondern auf Methoden und Klassen Ebene limitiert.

1.3.1 Übersicht

PREfix und PREfast sind zwei von Microsoft entwickelte Werkzeuge zur statischen Analyse. Sie speziell für C/C++ und die Windowsklassen Bibliotheken programmiert worden. findbugs ist ein in Java geschriebenes Werkzeuge der Universität von Maryland und untersucht Java Programme an Hand des ByteCodes. Coverity PREVENT ist eine Weiterentwicklung eines der Universität von Standford entwickelten Analyse Werkzeuges. Es wurde mehrere male auf die Quellen des Linux Kernels angewendet und hat zahlreiche Arten von Fehlern entdeckt. In diesem Jahr hat das U.S. amerikanische Verteidigungsministerium coverity beauftragt ,zahlreiche Freie Software Projekte zu analysieren. Es hat wiederrum zahlreiche Fehler, auch sicherheitsrelevante Fehler gefunden.
Der Text wird sich auf PREfast und findbugs konzentrieren.

1.3.2 Methode

findbugs und PREfast sind zwei Analysewerkzeuge zur statischen Analyse für zwei verschiedene Programmiersprachen. Beide Werkzeuge sind in der gleichen Klasse der statischen Analyse zu finden und sind vom Aufbau und der Funktionsweise ziemlich ähnlich. Im Kern sind sie in der Lage das Programm einzulesen Klassen,

Methoden, Variablen, Aufrufe, Sprünge, Schleifen und andere Sprachkonstrukte zu identifizieren und verfügbar zu machen. Der Kern bietet zu dem die Möglichkeit mögliche Programmabläufe und Variablenbelegungen zu kennen oder zumindest zu schätzen. Was beide Werkzeuge in die gleiche Klasse einordnet, ist der Umfang und Kontext der Analyse. Die Analyse verläuft auf Methodenebene, globale Variablen, Effekte anderer Methoden etc. werden bei der Analyse nicht betrachtet und sind auch nicht bekannt, dies wird als **intraprozedural** [13] bezeichnet. Im Gegensatz dazu steht die interprozedurale statische Analyse.
Um die Analyse durchzuführen existieren Detektoren, diese sind entweder in einer Programmiersprache geschrieben oder in einer Beschreibungsprache. Diese Detektoren prüfen einzelne Eigenschaften der Klasse, der Vererbungshierachie nach bestimmten sprachspezifischen Eigenarten (**Idiome**). Diese Eigenarten stellen in der Regel Fehler dar, können aber auch legitim sein. Die Detektoren arbeiten sehr nah an der Programmiersprache und an der Semantik der benutzten Bibliotheken. Die Detektoren benutzen meistens heuristiken, teilweise mit konfigurierbaren Schwellenwerten um Programmspezifische Faktoren berücksichtigen zu können.
Dies stellt eine systematische Schwäche dieser Form der statischen Analyse dar. Der formale Korrektheitsbeweis eines Programmes ist **vollständig** (complete), es werden alle Fehler entdeckt, und **korrekt** (sound), es werden nicht mehr Fehler als tatsächlich vorhanden berichtet. Die interprozedurale und intraprozedurale statische Analyse hingegen sind **nicht vollständig**, es werden echte Fehler übersehen, und **nicht korrekt** (not sound, unsound), es werden Fehler berichtet, die keine Fehler sind . Dies wird oft auch als **falsche positive** (false positive) oder **lärm** bezeichnet. Diesem Problem widmen sich spätere Kapitel [14].
Die grosse Stärke der intraprozeduralen und interprozeduralen Werkzeuge ist, direkt ohne besondere Änderung am Programm, die Analyse durchzuführen. Es werden keine speziellen Invarianten oder Notationen im Programm gefordert. Die Stärke in den intraprozeduralen Werkzeugen liegt in ihrer Einfachheit, sie sind einfacher zu programmieren und einfacher zu erweitern und sind schneller in ihrer Analyse. Der Frage, ob die Werkzeuge trotz dieser Eigenschaften und Einschränkungen Fehler entdecken wird im folgendem Text nachgegangen.

2 Findbugs

2.1 Ziele und Funktion

Das Ziel der findbugs Entwickler ist es mit einfachen Mitteln Fehler zu identifizieren. Es wird bewusst auf die interprozedurale Analyse verzichtet, da Fehler, bzw. Fehlermuster als Spracheigenarten angesehen werden, die mit einfachen Mitteln gefunden werden können. Findbugs enthält Detektoren die bestimmte Spracheigenarten von Java erkennen und diese berichten.
Anders als Analysewerkzeuge für Sprachen wie C und C++ analysiert findbugs den Bytecode von Java. Dies heisst, eine statische Prüfung wurde bereits vom Compiler vorgenommen und findbugs kann diese Prüfung nun verfeinern. Die Verwendung von Bytecode macht es einfacher, Methoden und Klassen einzulesen und es existierten bereits Werkzeuge, die dies tun.
Die einfachen findbugs Detektoren probieren in Methoden, Klassen und der Vererbungshierachie bestimmte Java spezifische Probleme zu finden, kompliziertere Detektoren betrachten den Inhalt, der zu untersuchenden Methode und die Schwierigsten betrachten den Kontrollfluss, wie z.B. Schleifen, Verzweigungen und Sprünge, und die möglichen Variablenbelegungen innerhalb der Methode. Ob diese Spracheigenarten tatsächlich Fehler darstellen und für das Programm relevant sind, kann und soll von findbugs nicht erkannt werden, z.B. ob ein mögliches Sicherheitsproblem tatsächlich für die Applikation relevant ist.
findbugs bietet eine graphische Oberfläche mit der man einzelne Projekte verwalten kann und regelmässig nach Fehlern suchen kann. Die Ergebnisse können auf mehrere Weisen betrachtet werden und zu jeder Fehlerklasse existiert eine Beschreibung der Ursache und mögliche Auswirkungen des Fehlers auf das Programm. Zusätzlich kann man findbugs auch ohne graphische Oberfläche benutzen und die Resultate der Analyse in eine XML-Datei speichern lassen, es existieren Programme die Unterschiede zwischen zwei Analysedurchläufen anzeigen können. Dies ist besonders praktisch, wenn man findbugs regelmässig auf ein Programm anwendet, da man die Wahl hat, nur neue Probleme zu betrachten oder doch das komplette Ergebnis.

2.2 Fehlerklassen

findbugs ist in der Lage mehr als 200 verschiedene Arten von Fehlern zu erkennen, die in einzelne Kategorien eingeordnet sind. Wie Hovemeyer [8] werden hier nur einige Detektoren kurz betrachtet. Für eine genaue Liste der Fehler und Ihre Bedeutung empfehle ich die findbug Webseite [1].

2.2.1 Invariante equal Implementierung (Eq)

Die Basisklasse **Object** bietet die **equals(Object)** Methode. Diese wird unter anderem von den Kontainerklassen benutzt. Java erlaubt es auch diese Methode zu überladen, nur wird diese Methode von den in Java eingebauten Klassen nicht zum Vergleichen von zwei Exemplaren aufgerufen.
Der Equals-Detektor (Eq) betrachtet Methoden mit den Namen **equals** und überprüft die statischen Typen des Parameters. Sollte dieser nicht direkt **Object** sein wird dies als möglicher Fehler vermerkt und berichtet.

2.2.2 hashCode Implementierung vergessen (HE)

In Java wird es gefordert, dass gleiche Objekte den gleichen **hashCode()** haben. Ein typischer Fehler ist es die equals() Methode zu überschreiben, aber die **hashCode** Methode nicht anzupassen.
Der HashCode-Detektor (HE) überprüft, ob eine Reimplementierung der equals() Methode vorhanden ist und prüft ob eine Reimplementierung für **hashCode()** ebenfalls vorhanden ist.

2.2.3 Null Verweise

```
class NullPointer {
    public void someMethod( Input in ) {
        if( in == null ) {
            in.anotherMethod();
        }
    }
}
```
(NP)

Die Erkennung der Dereferenzierungen von Null Verwiesen (NP) stellt einen komplexeren Detektor dar. Im Gegensatz zu den beiden oben genannten Detektoren werden hier nicht Methodennamen oder Klassenhierachien betrachtet sondern der Detektor baut sich Informationen zum Programmfluss und Speicherbelegung innerhalb der Methode auf. Rückgabewerte von anderen Funktionen oder Werte anderer Klassen und Methoden werden nicht in betracht gezogen.

Im obigen Beispiel sehen wir das „**in**" genau dann derefenziert wird, wenn es ein Verweis auf **null** ist. Diese Derefenzierung führt zur Laufzeit zu einer NullPointer-Exception. Mit dem Detektor bekannten Daten ist er in der Lage diesen möglichen Fehler zu erkennen und zu berichten.

2.2.4 Semantik von Ein- und Ausgabeklassen (OS)

Die Java Klassen für Ein- und Ausgaben haben im Zusammenhang mit der Speicherverwaltung eine semantische Schwäche. Java gibt keine Garantien wann nicht mehr benutzte Exemplare aus dem Speicher entfernt werden. Sollte die **close()** Methode auf dem Ein- oder Ausgabestrom nicht aufgerufen werden gibt es keine Garantien wann der **finalizer** des Exemplares dies tut. Dies führt zu zwei Problemen, einerseits werden Betriebssystem Mittel länger als nötig in Anspruch genommen und andererseits kann es passieren, dass Puffer nicht geleert werden und somit die Datei unvollständig ist.
findbugs bietet ebenfalls einen Detektor (OpenStream, OS) für diesen möglichen Fehler. Es werden lokal erzeugte Ein- und Ausgabeexemplare betrachtet und mit Hilfe des Kontrollflusses wird entschieden ob diese Ströme in allen möglichen Kombinationen des Programmflusses innerhalb der Methode auch geschlossen werden. Ist dies nicht der Fall wird ein möglicher Fehler berichtet.

2.2.5 Inkonsistente Sperren (IS2)

Der folgende Detektor (IS2) ist einer der schwierigsten und aufwendigsten Detektoren und stellt zu dem eine Ausnahme dar. Er betrachtet zwar jede Klasse und Methode für sich, wie es in der intraprozeduralen statische Analyse üblich ist, hält aber Zwischenergebnisse aller gesehenen Klassen, ihrer Methoden und Variablen vor und benutzt dies für die endgültige Auswertung. Dies macht es zu einem sehr aufwändigen und schwierigen Detektor. Die Herausforderung besteht darin zu erkennen welche Exemplare und primitiven Typen im Konflikt stehen, dass heisst nebenläufig benutzt werden. Dem Detektor stehen keine Informationen über die Anzahl der Programmfäden oder dem Lebenszyklus dee Exemplare bereit.
Daher sammelt der Detektor zu jedem Exemplar in jeder Methode Informationen, ob lesend oder schreibend zugegriffen wurde und ob ein Monitor und welcher Monitor betreten wurde. Um genauer zu arbeiten, werden Variablen, die als **volatile** oder **public** qualifiziert sind, ignoriert. Zu dem werden private Methoden und Variablen bei denen man sich beim Lesen immer in einem Monitor befindet ignoriert. Ausserdem werden Variablen ignoriert, bei denen um $\frac{1}{3}$ mehr ohne betretenen Monitor drauf zugegriffen wurden als im betreten Monitor, Konstruktoren und finalizer werden ebenfalls ignoriert.

2.2.6 Schadhafter Code (MS,EI)

```
public class MS {
    public static final Object[] cursor= { null, null, null};
    private static final Object[] certificates = { null, null, null};

    public Object[] getCertificates() {
        return certificates;
    }
}
```

Klassen die (static) Felder entweder jedem zugänglich machen (public) oder durch öffentliche Methoden Zugang verschaffen sind potentielle Sicherheitsrisikos. Findbugs erkennt diese Art von Fehler (MS, EI) durch eine einfache Klassenanalyse in dem es die (statischen) öffentlichen Variablen und die zurückgegeben Exemplare prüft.
Das Sicherheitsrisiko besteht darin, dass unerlaubter Code die Funktionsweise einer Klasse verändert und so, z.B. mehr Möglichkeiten bekommt. So ist es z.B. möglich, dass ein Applet die Cursor der AWT Klasse verändert. Noch schlimmer war ein durch findbugs entdecktes Problem in der Implementierung der jar Klasse in J2SE 1.4.1, java.util.jar.JarEntry. Wie im obigen Code angedeutet gibt die **getCertificates** Methode eine Referenz auf ein Feld mit Zertifikaten. Das Feld an sich kann nicht verändert werden, jedoch der Inhalt des Feldes. So war es möglich ein Zertifikat durch ein anderes auszutauschen.

2.2.7 Zugriff auf nicht geschriebene Variablen (UR)

Der Konstruktor dient der Erstellung und Initialisierung eines Exemplares. Java initialisiert Klassenvariablen mit Standardwerten daher sind die aus anderen Sprachen bekannten Probleme mit nicht initialisierten Verweisen nicht so gravierend es treten jedoch Probleme auf. Dieser Detektor (UR) prüft ob bei einer Zuweisung im Konstruktor die rechte Seite schon ein Mal geschrieben wurde. Es werden wiederum nur lokale und Klassenvariablen betrachtet, die Belegung von Parametern ist ungewiss.

2.2.8 Endlosschleifen

```
public class RecTest {

    public void terminating3( int i ) {
        if( i < 0 )
            return;
        terminating2( i+1 );
```

```
    }

    public void terminating2( int i ) {
        if( i < 0 )                                                    10
            return;
        terminating2( i-1 );
    }

    public void terminating( int i ) {
        if( i < 0 )
            return;
        terminating( i );
    }
                                                                       20
    public void otherrec( int i ) {
        otherrec( i-1 );
    }

    public void rec(int i ) {
        rec( i );
    }

    public void verschraenkt1( int i ) {
        verschraenkt2( i );                                            30
    }

    public void verschraenkt2( int i ) {
        verschraenkt1( i );
    }
}
```

Ein recht neuer Detektor dient zum Auffinden von Endlosschleifen. Es wird nach Methoden, unter Verwendung des Programmflusses, gesucht, die sich mit den gleichen Parametern aufrufen. So werden **otherrec**, **terminating**, **rec** als Endlosschleifen identifiziert. **terminating** ist dabei besonders gemein, da es über eine Abbruchsbedingung verfügt der Parameter jedoch nicht verändert wird. Verschränkte endlose Rekursion oder der Fehler in **terminating3** wird nicht entdeckt.

2.2.9 Nicht beachte Länge der Eingabe (RR)

Die Java Klassen zur Eingabe, **InputStreams** und **Reader**, bieten die Möglichkeit, eine Anzahl von bytes in ein Feld zu lesen. Es besteht die Möglichkeit, dass das Feld nicht ganz gefüllt wird, daher geben die Lesemethoden die Anzahl der gelesenen Bytes zurück. Ein Detektor (RR) überprüft, ob der Rückgabewert benutzt, z.B. gespeichert, oder direkt verworfen wird. Ob der Wert semantisch korrekt verwendet wird, wird nicht geprüft.

2.2.10 Rückgabewert nicht beachtet (RV)

Ähnlich wie beim obigen Detektor wird geprüft ob ein Rückgabewert benutzt wird oder direkt verworfen wird. Der Detektor (RV) probiert häufige Fehler bei der Benutzung von String, StringBuffer.toString und InetAddress zu finden. Exemplare der String Klasse sind nicht veränderbar, um den Wert zu verändern, müssen neue Exemplare mit dem geänderten Wert erzeugt werden. Die findbugs Entwickler gehen davon aus, dass ein Verwerfen eines neuen Exemplares einen Fehler darstellt. Sollte das Ergebnis mit Absicht verworfen werden, zeigt dies eine semantische Schwäche der Klasse bzw. Methode, da der Aufrufer nur am Effekt und nicht an der Rückgabe interessiert ist.

2.2.11 Sperrsynchronisation (Wa, UW)

Der Aufruf von **Object.wait()** dient meist dazu auf die Erfüllung einer Bedingung zu warten (Wa). Robuster Quellcode sollte nach dem aufwachen prüfen ob diese Bedingung tatsächlich erfüllt ist. Z.B. könnte ein Puffer nach dem Aufwachen immer noch nicht genug Platz haben. Daher sollte man meist das Warten und die Bedingung in einer Schleife wiederholt prüfen und bei Erfüllung fortfahren.
Ein oft verwendetes Idiom ist die Bedingung zu prüfen, dann den Monitor zu betreten und sofort ohne Prüfung einer Bedingung zu warten (UW). Das Java Speichermodell gibt für diese Funktionsweise jedoch keine Garantie. Nach dem Prüfen und vor dem Betreten des Monitores kann ein anderer Prozess den Wahrheitswert dieser Bedingung verändern und der eigentliche Prozess wartet dennoch auf die Erfüllung.
findbugs ist in der Lage beide Idiome zu erkennen und zu berichten.

2.3 Anwendung

Hovemeyer hat verschiedenen quelloffene Java Applikationen untersucht. Näher wird hier jedoch nur eclipse und der GNU classpath betrachtet. Die Resultate wurden aus Hovemeyer's Arbeit übernommen [8] und mit eigenen Erfahrungen erweitert.

	GNU classpath 0.06				
	Warnungen	Ernst	Harmlos	Dubious	Falsch
IS2	18	72%	16 %	0 %	11%
NP	7	85%	0%	0%	14%
OS	9	22%	33%	22%	22%
RR	7	100%	0%	0%	0%
RV	11	45%	0%	0%	54%
UR	3	100%	0%	0%	0%
UW	2	0%	0%	0%	100%
Wa	2	0%	0%	0%	100%

	eclipse 2.1.0				
	Warnungen	Ernst	Harmlos	Dubious	Falsch
NP	43	93%	0%	6%	0%
OS	16	6%	6%	18%	68%
RR	22	4%	0%	0%	0%
RV	9	100%	0%	0%	0%

2.3.1 eclipse

eclipse 2.0.1 wurde untersucht und Hovemeyer konnte beobachten, dass zwei Detektoren sehr gute und genaue Resultate geliefert haben und Fehler entdeckt wurden. Der OpenStream und RR Detektor hingegen waren sehr ungenau und haben mehr falsche Ergebnisse als echte geliefert. Der OpenStream Detektor ist nicht in der Lage die geänderte Semantik von den Strömen in eclipse zu erkennen und leidet daher unter diesem Problem.
Ansonsten hat die Analyse echte Probleme in einem grossen Projekt gefunden, die zu echten Abstürzen und Versagen führen konnten. Besonders akurat ist der Nullverweis Test (NP). Das deutet auf eine sehr gute Programmfluss und Datenfluss Analyse durch findbugs hin.
Ich habe ähnlich wie Hovemeyer zusätzlich eclipse 3.0.1 untersucht. Der grösste Aufwand in der Benutzung bestand im Hinzufügen der JAR-Archive von eclipse. Die Analyse von 95 Archiven mit 16496 Klassen benötigte 62 Minuten. Eine genaue Auswertung der einzelnen Fehler wurde nicht durchgeführt. Einige Fehler wurden stichprobenartig überprüft. So wurden drei mögliche Endlosschleifen von findbugs berichtet. Jede dieser Berichte war korrekt. Besonder heraustechend war ein Nullverweis Fehler in **org.eclipse.update.internal.core.ConfiguredSite**.

```
/*******************************************************************************
 * Copyright (c) 2000, 2004 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Common Public License v1.0
```

```
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/cpl-v10.html
 *
 * Contributors:
 *     IBM Corporation - initial API and implementation
 ******************************************************************************/
package org.eclipse.update.internal.core;

/**
 * A Configured site manages the Configured and unconfigured features of a Site
 */
public class ConfiguredSite extends ConfiguredSiteModel implements IConfiguredSite {
        private static String getProductIdentifier(String identifier, File propertyFile) {
                String result = null;
                if (identifier == null)
                        return result;
                InputStream in = null;
                try {
                        in = new FileInputStream(propertyFile);
                } catch (IOException e) {
                } catch (MissingResourceException e) {
                } finally {
                        if (in == null)
                                try {
                                        in.close();
                                } catch (IOException e1) {
                                }
                }
                return result;
        }
}
```

Besonders ist ,,**in**" im finally Block zu beobachten. Sollte **in** an dieser Stelle **null** sein, wird eine NullPointerException geworfen, aber auf jeden Fall wird die Eingabe nie geschlossen. Was diesen Fehler so herausstechend macht ist die Tatsache, dass dieser **finally** Block existiert um das oben besprochene semantische Problem mit Ein- und Ausgabeströmen (OS) zu vermeiden.

2.3.2 GNU Classpath

GNU classpath 0.06 wurde untersucht. Es ist eine recht frühe Version gewesen und bei weitem nicht so oft benutzt wie andere Projekte z.B, eclipse. Jedoch ist Sicherheit, Robustheit und Korrektheit bei einer so fundamentalen Bibliothek entscheidend. findbugs hat echte Probleme gefunden die zu vor nicht durch Testen entdeckt wurden.

Die Stärke von statischen Analyse Mitteln ist, dass sie sofort auch in frühen Versionen eingesetzt werden können, dies kann man hier ganz gut erkennen.
Es ist interessant zu erkennen das abhängig von der untersuchten Applikation einzelne Detektoren stark unterschiedliche Ergebnisse liefert. So ist der OpenStream Test bei eclipse nicht brauchbar, beim GNU classpath hingegen überwiegt die Anzahl der echten Probleme.
Besonders herausstechend hier war ein Problem bei der Nebenläufigkeit. Der IS2 Test war in der Lage **elementCount** als mangelhaft geschützt zu identifizieren. Es besteht die Möglichkeit, dass **elementCount** z.B. durch ein Entfernen eines Elementes verändert wird. Somit könnte eine ArrayOutOfBoundsException geworfen werden, dies würde aber gegen den Schnittstellenvertrag verstossen.

```
// GNU classpath 0.06
// java.util
// Vector.java, line 354

public int lastIndexOf(Object elem) {
    return lastIndexOf( elem, elementCount - 1 );
}
```

3 PREFast

3.1 Ziele

PREfix [13] ist ein interprozedurales Werkzeug. Die Geschwindigkeit und die Systemanforderungen verhinderten den Einsatz bei den Entwicklern. Stattdessen gab es ein zentrales System, dass PREfix ausführte und die Berichte automatisch in das Fehlermanagementsystem bei Microsoft eintrug. Die berichteten Fehler waren nicht einfach zu finden und zu filtern. Das Hauptziel von PREfast, ein intraprozedurales Werkzeug, war weniger als PREfix zu können, aber dafür deutlich schneller zu arbeiten, um den Entwicklern die Möglichkeit zu geben Fehler zu finden bevor sie in das Versionsverwaltungs-System eingepflegt werden [13], dem Entwickler bessere Möglichkeiten bei der Fehleranalyse geben. Ausserdem sollte es einfach zu erweitern sein.
Der Hauptfokus lag zu erst in der Implementierung von Detektoren, die sicherheitsrelevante Fehler aufdecken können. Die Geschwindigkeit und Bedienbarkeit bewirkten eine hohe Akzeptanz bei den Entwicklern. So wurde PREfast kontinuerlich um weitere Detektoren erweitert. So kommt es, dass PREfix 40 Detektoren besitzt und PREfast über 100 und PREfast ein höhere Akzeptanz bei den Entwicklern hat [11].

3.2 Funktionsweise

Über die genaue Implementierung von PREfast ist wenig in Erfahrung zu bringen. Offensichtlich existiert wie bei findbugs eine Komponente um das Projekt einzulesen. Im Falle von PREfast wird der Quelltext direkt eingelesen und geparsed. Die Reihenfolge der Abarbeitung spielt keine Rolle da alle Analysen lokal sind und nicht auf externe Klassen und Methoden oder globale Variablen zurückgreifen. Ähnlich wie bei findbugs werden sehr wahrscheinlich die Detektoren auf die Methode bzw. Klasse angesetzt um Spracheigenarten zu identifizieren. So werden printf Varianten untersucht und geschaut ob die Formatierung verträglich mit den angegebenen statischen Typen ist, oder es wird beim Kopieren von zwei Feldern darauf geachtet, dass kein grösseres Feld in ein kleines kopiert wird.
Die Ergebnisse von PREfast werden gespeichert und mit einem **prefast view** Aufruf ist man in der Lage, sich jeder Zeit die Ergebnisse im Internet Explorer

anzusehen.

3.3 Anwendung bei Microsoft

3.3.1 Sicherheitsfehler

PREfast wurde mit der Hoffnung entwickelt spezielle Konstrukte zu identifizieren die mögliche Sicherheitsfehler enthalten. So werden Puffergrössen, Nullterminierung, Initialisierung von Variablen, Zugriff auf Verweise und Speichermanagement überprüft. Das Auffinden dieser Fehler kann Sicherheitsfehler vermeiden. Dies ist eine Strategie die das OpenBSD Projekt mit seinen Durchsichten betreibt.

3.3.2 Treiber Testen

Microsoft hat auf Grund der grossen Nachfrage nach PREfast ein spezielles Profil für die Treiberentwicklung geschrieben. Ein Profil besteht aus einer Menge angepasster und neuer Detektoren für PREfast. Es wird PREfast for Drivers genannt und ist im Treiber Entwicklungs-Kit (DDK) für Windows 2003 enthalten. Auf der MSDN Seite [10] werden zahlreiche Einführungen zu diesem Werkzeug gegeben, PREfix und das normale PREfast bleiben weiterhin intern.
Da das Treiber Entwicklungs Kit frei herunterladbar ist, stellt PREfast das einzige intraprozedurale statische Analyse Werkzeug für C und C++ dar, dass frei benutzt werden kann. Auf Grund des treiberspezifischen Profiles eignet es sich nicht für andere Anwendungen.

3.4 Fehlerklassen

Über die Fehlerklassen des internen PREfast Werkzeuges ist wenig bekannt. Es existieren über 100 Detektoren die unterschiedliche Probleme identifizieren können. In der Einführung in PREfast für Drivers werden einige Fehlermeldungen erklärt [10]. Die erkennbaren Fehlerklassen beinhalten unter anderem:

- Unitialisierte Variablen
- Nullzeiger Derefenzierung
- Formatierungsfehlern bei printf Funktionen
- Vermischung von free, delete, new, malloc, new[], malloc[]
- Speicherlecks

- Pufferüberläufe und falsche Grössenangaben.
- Fehlende Überprüfung der Rückgabe von Methoden, die Speicher allozieren
- Vermischung von falschen Typen

Die Menge der erkennbaren Fehler und die teilweise sehr spezifische Analyse von Funktionen mit besonderen semantischen Eigenschaften schaffen den Eindruck, dass PREfast einfach zu erweitern und an spezielle Gegebenheiten anzupassen ist. Die Ziele und die PREFast for Drivers Veröffentlichung scheinen diesen Eindruck zu bekräftigen.
Die erwähnten Fehlerklassen sind meist für Abstürze, Speicherkorruption und Sicherheitsprobleme verantwortlich. Die Möglichkeit einige dieser Probleme zu erkennen stellt einen gewaltigen Fortschritt dar. PREfast ist Teil der Sicherheitsstrategie von Microsoft, nähere Informationen über den Erfolg des Werkzeuges bei Microsoft sind leider nicht verfügbar.

4 Schwächen und Probleme?

4.1 Schwächen

4.1.1 Falschmeldungen

Ein Problem von Werkzeugen ist ihre Akzeptanz bei den Entwicklern, Testern und anderen Angestellten (Benutzern). Ein Aspekt des Werkzeuges ist, wie viele Meldungen tatsächlich für den Benutzer relevant sind. In [12] wird die Gefahr beschrieben, dass das Werkzeug viele Meldungen ausgibt, aber für den Benutzer als nicht zutreffend angesehen werden. Im Allgemeinen ist dieser Lärm schlimmer als ein nicht gefundener Defekt. Für den Benutzer sorgt zu viel Lärm zu der Ablehnung des Werkzeuges. Damit besteht die Gefahr das alle erkennbaren Defekte in den Quellen verbleiben.
Eine Ursache des Lärmes sind laut Jonathan Pincus [13, 8] Entwurfsentscheidungen der intraprozeduralen Werkzeuge. Da Methoden nicht im Zusammenhang mit globalen Variablen, Singletons oder ähnlichen betrachtet werden, können einzelne Detektoren unter bestimmten Umständen Falschmeldungen liefern. Zu diesen Fällen gehören z.B. Variablen, die als uninitialisiert bezeichnet werden, weil sie erst in Schleifen initialisiert werden und das Werkzeug nicht feststellen konnte, dass die Schleife mindestens ein Mal durchlaufen wird. Dies kann z.B. auf Grund globaler Variablen zugesichert sein [10]. Tests mit genaueren Ergebnissen brauchen deutlich mehr Zeit, können aber weiterhin Lärm produzieren, denn auch sie bleiben nicht vollständig (not complete) und nicht korrekt (not sound), und senken die Performance der Werkzeuge entscheidend und er [12] führt weiterhin die Qualität des Parsers zur Analyse des Quelltextes und die Qualität und Genauigkeit der Ausgaben des Werkzeuges – wie z.B. für den Benutzer nicht verständliche Fehlermeldungen – als weitere Gründe an.
Pincus [12] schlägt vor in seltenen Fällen die Analyse zu verfeinern, dies würde sich z.B. für die eclipse Code Basis und den ,,Open Stream" Test anbieten, da die eclipse Stream Implementierung nicht die semantischen Schwächen der Standardimplementierung hat. In der Regel sollten aber Meldungen mit einem mehrdimensionalen Rang versehen werden wobei Genauigkeit des Fehler, Verständlichkeit der Meldung und Auswirkung des Fehlers eine Rolle haben sollten. Zusätzlich sollte die Geschichte in Betracht gezogen werden. Das heisst ein Mal als Lärm einge-

stufte Meldungen sollen in Zukunft nicht angezeigt werden, neu hinzugekommen Meldungen sollen deutlich sichtbar gemacht werden. Zudem sollen die Werkzeuge über eine gute Oberfläche verfügen, die Sortieren und Filtern ermöglicht.
Daher bietet findbugs die Sortierung der Berichte nach Dateien, Paketen und Art des Fehlers an. Zusätzlich wird ein Werkzeug bereitgestellt, dass die Unterschiede zwischen zwei findbugs Läufen darstellen kann. Auf diese Weise muss man die Falschmeldungen nur ein Mal sehen.

4.1.2 Integration

Ein Werkzeug erfüllt nur dann seinen Zweck, wenn es benutzt wird. Sowohl PREfast als auch findbugs sind gedacht schnell zu sein um auf dem Rechner des Entwicklers ohne grosse Verzögerung Resultate zu zeigen. Beide Programme wurden geschrieben um eine gute Integration in den normalen Arbeitsablauf zu haben, sie sind in die Compiler bzw. Entwicklungsumgebung integriert und liefern schnell Resultate. Auch grosse Projekte sind mit beiden Programmen beherrschbar.
So bietet PREfast die direkte Integration in den Übersetzer. Die statische Analyse ist also nach jedem Übersetzen vorhanden. findbugs bietet ähnliche Integration in die integrierte Entwicklungsumgebung eclipse.

4.2 Probleme

4.2.1 Wie viel Prozent der vorhanden Fehler werden gefunden?

Zur Erinnerung die hier betrachteten Werkzeuge sind nicht vollständig (not complete), es stellt sich die Frage wie viel Prozent der vorhanden Fehler gefunden wurden. Es existieren Verfahren um dies abzuschätzen jedoch beschäftige ich mich an dieser Stelle und zu diesem Zeitpunkt nicht mit diesen. Eine mögliche Methode zur Bestimmung der vorhanden Fehler stellt COQUALMO [4] dar.
Bei den empirischen Untersuchungen von Hovemeyer [8] ist zu sehen das bei Industrieprojekten mit regelmässigen Durchsichten und automatisierten dynamischen Analysen durch findbugs neue Fehler aufgedeckt wurden.
Ein anderer Aspekt ist die Bestimmung der Qualität anhand gefundener Fehler durch die statische Analyse. Mit Hilfe des PREfast und PREfix Werkzeuges und über Jahren automatisch gesammelten Daten der beiden Werkzeuge beschreibt Thomas Ball in seiner Arbeit [2] wie mit einer Genauigkeit von 90% Aussagen über die Fehleranfälligkeit von Modulen getroffen werden können. Aufgrund der fehlenden Daten zur Verifikation seiner Ergebnisse und die Umstellung der Prozesse bei Microsoft zur systematischen Nutzung von PREfast am Arbeitsplatzrechner und der daraus zwangsläufigen resultierenden Senkung von statisch entdeckbaren

Fehlern durch das PREfix Werkzeug kann man die Ergebnisse von Ball sehr wahrscheinlich nicht auf andere Projekte übertragen und auch nicht auf Projekte die bereits statische Analyse am Arbeitsplatzrechner benutzen.

4.2.2 Bedeutung der gefundenen Fehler

Hovemeyer [8] macht deutlich, das echte Defekte in Produkten durch Detektoren von findbugs erkannt werden. Das heisst, durch die regelmässige Benutzung von findbugs bei Java Applikationen treten im Produkt weniger Defekte auf. Es werden auch offensichtliche Probleme gefunden die bei Durchsichten nur sehr schwer entdeckt werden können. Hier werden besonders Verdeckungsprobleme von Variablen angeführt.
Die Art der gefundenen Fehler kann tiefe Konsequenzen für die Sicherheit und Robustheit der Applikation und der Benutzer haben. Statische Analyse ist in der Lage, gravierende Probleme aufzudecken. Man überlege die Konsequenzen eines falsch gesetzten Semikolons in der Steuerung von teuren Objekten. Statische Analyse ist in der Lage solche Fehler aufzudecken.

5 Bedeutung der statischen Analyse

5.1 Für Microsoft

5.1.1 Integration in die Prozesse

PREfast war das erste Werkzeug bei Microsoft das einerseits auf den Rechnern der Entwicklern zeitnah laufen konnte, gute Ergebnisse lieferte und die Resultate in einer verständlichen Art und Weise darstellt [11]. Durch die Flexibilität und anfänglichen Erfolge des PREfast Werkzeuges wurden Prozesse bei Microsoft verändert. Dies bewirkte die Einführung der Null Toleranz, gegenüber sicherheitskritischen Fehlern, die PREfast aufdecken kann.
Die konsequente Integration von PREfast in den Arbeitsprozess zeigt die Bedeutung der statischen Analyse für die Qualität von Produkten. Mit speziellen Versionen von PREfast ist man in der Lage spezielle Probleme sehr genau zu identifizieren und zu fixen. Eine Menge gravierender menschlicher Fehler wie Pufferüberläufe, Zeichenketten-Formatierungsfehler sind auf ein Mal nahezu vermeidbar. Sollte eine Klasse von Sicherheitsfehlern nicht erkannt werden, soll es relativ einfach sein, PREfast um einen Detektor zu erweitern. Für Microsoft bedeutet dies, eine Klasse von ein Mal begangenen Sicherheitsfehlern sind auf ein Mal vermeidbar, da automatisch erkennbar.

5.1.2 Für Treiberhersteller

PREfast wurde auf Wunsch der Kunden nach besseren Werkzeugen geöffnet. Für die Windowstreiberentwicklung wurden spezielle Detektoren geschrieben, bzw. Detektoren angepasst. PREfast ist konsequent in den Treiberentwicklungsprozess integriert und ist Bestandteil des Rahmenwerkes zur Treiberentwicklung. Das PREfast Werkzeug und seine Meldungen sind gut erklärt und die Ergebnisse einfach für den Web-Browser einzusehen. Die Ergebnisse stehen nach jedem Bauen des Treibers zur Verfügung.
Speicherleaks, Bufferflows, falsche Verweise in Treibern könnten dadurch bald ein Ende haben. Auf jeden Fall hilft die statische Analyse in Form von PREfast Treiberentwicklern robustere Treiber zu schreiben. Davon profitieren die Treiberhersteller, die Kunden und Microsoft, da Treiber schneller geschrieben und getestet werden können, dem Kunden bringt es eine verbesserte Stabilität des Systemes.

5.2 Für Freie Software

Die Qualitätssicherung nimmt in Freien Software Projekten zu, einerseits durch das gestiegene Interesse an besserer Qualität durch die grössere Anzahl von Benutzern und Fehlerberichten, andererseits durch die Verbesserung der Werkzeuge. So führen heutzutage Compiler für statisch getypte Sprachen wesentlich mehr Überprüfungen durch die vorher in der Domäne anderer Werkzeuge lagen. Es besteht ein Wunsch der Projekte nach mehr Qualität, so setzt die Mozilla Foundation ein Tool zur dynamischen Analyse namens Tinderbox (http://tinderbox.mozilla.org) ein und das KDE Projekt (http://www.kde.org) hat mit dem English Breakfast Network (http://www.englishbreakfastnetwork.org/) ausschliesslich statische Analyse im Einsatz, mit dem Wunsch immer mehr Aspekte des Projektes statisch zu prüfen. So ist ein C++ Parser entstanden um die Entwicklung eines freien Analyse Werkzeuges zu fördern.
Andererseits besteht eine Wechselwirkung zwischen quelloffener Software und Entwticklern von proprietären und freien Analyse Werkzeugen. So hat die Firma coverity ihre Analyse Techniken am Linux Kernel ausprobiert und so viele Probleme - auch kritische - aufgedeckt, ihr Tool selbst weiterentwickelt und kann so öffentlich den Erfolg ihrer Werkzeuge nachweisen. Gleiches haben die findbugs Entwickler bei den eclipse, classpath, jboss Projekten getan. Bei den Entwicklern und auch Beobachtern dieser Projekte führen solche Demonstrationen meistens zu einer höheren Akzeptanz dieser Kategorie von Werkzeugen.
Die Anwendung von findbugs auf die GNU classpath Quellen hat tatsächliche Fehler gefunden, Fehler, die durch systematisches Testen nicht gefunden wurden. Fehler die auch nur schwer durch testen auffindbar sind, da sie nichtdeterministisch sind.

5.3 Für uns

Statische Analyse ist wie gesehen tatsächlich in der Lage, bestimmte Fehler zu entdecken. Auch das freie Werkzeug findbugs ist in der Lage, echte Fehler auch in grossen und bekannten Applikationen zu enthüllen und die erwähnten Schwächen und Probleme sind bekannt, beherrschbar oder nicht vorhanden. So bietet findbugs eine gute Integration in integrierte Entwicklungsumgebungen, Fehlerklassen kann man leicht filtern und sie werden gut beschrieben und die Menge an Falschmeldungen kann man durch geschicktes Filtern ebenfalls unterdrücken.
Für uns sollte dies Anlass sein, sich dafür zu entscheiden, seine eigenen Werke ohne statisch erkennbare Fehler auszuliefern, bzw. ohne statisch erkennbare Fehler in das Versions-Kontroll-System einzupflegen. Die Schnelligkeit und hohe Integration in Entwicklungsumgebungen von findbugs macht die Benutzung in unseren

eigenen Kreationen möglich.
Auf der anderen Seite muss man sich über die erwähnten Schwächen im Klaren sein. Bei der behandelten Form von statischer Analyse handelt es sich um keinen Korrektheits Beweis, es werden nur Fehler Muster entdeckt und berichtet. Das bedeutet, auch wenn die Werkzeuge keine Meldungen liefern können und werden Fehler existieren. Daher wäre es fatal sich einzig auf die einfache Form der statischen Analyse zu verlassen. Die Statische Analyse sollte ergänzend zur Dynamischen betrachtet und benutzt werden.
Der Markt an Statischen Analyse Werkzeugen wächst und sie werden immer leistungsfähiger, wir sollten uns daher jetzt bemühen, diese Werkzeuge als sinnvolles Mittel gegen Fehler anzuerkennen und in unseren Arbeitsablauf zu integrieren. Die Qualität unserer Werke kann sich durch die kontinuierliche Anwendung nur verbessern. Wir sollten die ihre Anweundung als Chanche sehen so lange der Grundsatz gilt: ,,Denken schadet nicht".

Literaturverzeichnis

[1] *http://findbugs.sf.net*, 2006.

[2] Thomas Ball and Nachiappan Nagappan, *Use of relative code churn measures to predict system defect density*, 2001.

[3] Petra Fiedler, *Arbeitsumgebung*, 2004.

[4] Center for Software Engineering, *Coqualmo*, 2001.

[5] Matthias Gafert, *Menschlisches versagen*, 2004.

[6] Blörn Gustavs, *Denkfallen*, 2004.

[7] C.A.R Hoare, *An axiomatic basis for computing programming*, Communications ACM 12,10, 1969.

[8] David Hovemeyer and William Pugh, *Finding bugs is easy*, Dept. of Computer Science, University of Maryland.

[9] S.C. Johnson, *Lint, a c program checker*, Programmer's Supplementary Documents Volume 1, 1986.

[10] MSDN, *Prefast with driver-specific rules*, Microsoft, 2004.

[11] Jonathan Pincus, *Prefix, prefast, and other tools and technologies*, PREfix, PREfast, and other tools and technologies.

[12] _____, *User interaction issues in defect detection tools*, User Interaction Issues in Defect Detection Tools, 2001.

[13] _____, *Steering the pyramids*, Steering the Pyramids, Microsoft Research, 2002.

[14] Valentin Weckerle, *Semi-formale methoden*, (2004).

[15] Edward Yourdan, *Death march*, vol. 2, 2001.