Statische Analyse

Holger Hans Peter Freyther[1]

[1]Freie Universität Berlin

Seminar zu Ursachen und Vermeidung von Fehlern in der Softwareentwicklung, 2006

Übersicht

Übersicht

Übersicht

Dynamische Analyse

- Ausführen des Programmes
- Resultate einzelner Komponenten testen
- Zur Laufzeit Invarianten prüfen

Statische Analyse

- Betrachtung des Codes ohne ihn laufen zu lassen

Finde den Fehler 1

```
System.out.println("Wo ist der Fehler?");
```

- Wo ist der Fehler?
- Wie beweise ich, dass kein Fehler vorhanden ist?

Finde den Fehler 2

```
int fib(unsigned int anzahl) {
if( anzahl ==  0 )return 0;
else if( anzahl == 1 ) return 1;
else return fib(anzahl-1)+fib(anzahl-2); }
```

- Wo ist der Fehler?
- Kann dies ein Werkzeug erkennen?

Finde den Fehler 3

```
int a [] = getArray();
for( int i = 0; i < 10*a.length; ++i)
 System.out.println(a[i]);
```

- Wo ist der Fehler?
- Kann dies ein Werkzeug erkennen?

Werkzeuge - ein Überblick

- Hoare Kalkül
- Compiler (GNU Compiler Collection)
- Interprozedurale und einfache intraprozedurale Werkzeuge (PREfix, coverity, PREfast, findbugs)
- **Heute** betrachten wir das intraprozedurale Werkzeug findbugs genauer

Was ist findbugs?

- Einfach
- Intraprozedural
- Perspektive der Betrachtung von Detektoren
  - Sind sie richtig oder falsch?
  - Effektiv oder Ineffektiv?

Funktionsweise

- findbugs liest Java Bytecode
- Aus dem Bytecode werden Klassen, Methoden extrahiert
- Detektoren suchen nach bekannten fehleranfälligen Spracheigenarten
- Gefunde Eigenarten werden berichtet
- Berichte können per GUI betrachtet werden oder als XML weiterverarbeitet werden.

Detektoren Interface

- Detector und StatelessDetector Interface
- visitClassContext(ClassContext) Einstieg der Detektoren
- Helferklassen dienen zur besseren Bearbeitung
- Wie z.B. DismantleBytecode, BytecodeScanningDetector

Bekannte Fehlermuster

- Es werden über 200 unterschiedliche Fehlermuster erkannt
- Nullzeiger Derefenzierung
- Unitialisierte Variablen
- Ungenügende Sperrsynchronisation
- Und viele andere fehleranfällige Konstrukte
- Es ist einfach zu erweitern

Besonders zuverlässige Detektoren

- Klassen die **equal** implementieren müssen auch **hashCode** implementieren (HE)
- Klasse führt eine Kovariante Implementierung von equals ein (Eq)
- Verweise auf interne Daten werden nach aussen gegeben (MS)

Fehlermuster Cloneable - Übersicht

```
public class WrongClonable implements
  java.lang.Conleable {}
```

- Nach Konvention sollen Klassen die Cloneable implementieren **clone()** implementieren
- Der Compiler prüft es jedoch nicht
- Zu dem muss super.**clone()** aufgerufen werden
- Kann das ein Fehler sein?

Fehlermuster Cloneable - Detetkor

- Implementierung in CloneIdiom.java
- Stelle fest das es kein Interface und keine Abstrakteklasse ist
- Warte auf die **clone** Methode
- Warte auf clone Aufruf in der **clone** Methode
- Am Ende erstelle Bericht abhängig von den Beobachtungen

Fehlermuster nicht gelesener Speicher

```
public void dls(int i) {
 i = i++;
 int j = 0;
 j += 10;
 berechne_j( i );
}
```

- Die Variable **j** wird nie gelesen
- Die Variable **i** wird geschrieben aber sofort wieder überschrieben
- Können solche Fehler auftreten?

Fehlermuster nicht gelesener Speicher - Genauer

```
public void dls(int);
  Code:
   0:   iload_1
   1:   iinc    1, 1
   4:   istore_1
   5:   iconst_0
   6:   istore_2
   7:   iinc    2, 10
   10:  aload_0
   11:  iload_1
   12:  invokespecial   #18
   15:  return
```

Fehlermuster nicht gelesener Speicher - FindDeadLocalStore.java

- Betrachte alle Methoden einer Klasse
- Bestimme Anzahl der Lade-, Speicher- und Inkrementieroperationen pro Methode
- Probiere vom Compiler automatisch generierten Code zu vermeiden
- Betrachte Parameter die nur überschrieben werden

Kurze Demonstration von findbugs

- Fehler existieren auch in grossen Projekten [1]

[1] http://findbugs.sourceforge.net/docs/oopsla2004-slides.pdf

Kategorien der Fehler

- Fehler werden in sechs Kategorien geordnet
- Korrektheit
- Internationalisierung
- Nebenläufigkeit und Synchronisation
- Performance
- Stil
- Sicherheit

Nullzeiger Derefenzierung - Korrektheit

```
if( in == null )
 try {
  in.close();
 } catch ( IOException e1 ) {}
```

- eclipse 3.0.1, org.eclipse.update.internal.core.ConfiguredSite
- Und dies ist in einem **finally** Block
- Er existiert nur auf Grund einer Schwäche von Java (OS)

Nullzeiger Derefenzierung - Korrektheit

```
if ( local != null ||
 local.getType() == IResource.FILE) {
```

- Eclipse 3.0.1, org.eclipse.team.internal.ccvs.core.CVSSyncInfo

Synchronisation

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

- GNU Classpath 0.08 java.util.Vector

Synchronisation

```
// If we are not enabled, then wait
if (!enabled) {
 try {
 log.debug("Disabled, waiting for notification");
 synchronized (lock) {
  lock.wait();
 }
```

- JBoss 4.0.0RC1, org.jboss.deployment.scanner.AbstractDeploymentScanner

Veränderbare statische und private Felder

```
public class JarEntry extends ZipEntry {
 Certificate[] certs;
 public Certificate[] getCertificates() {
  return certs;
 }
}
```

- J2SE 1.4.1
- Sicherheitskritisch

Kein Fehler ist so offensichtlich, dass er nicht gemacht wird

- etliche Fehler in SUNs Java Implementierungen gefunden
- etliche weitere Nullzeiger Derefenzierungen (SUN, eclipse, classpath)
- Inkonsistente Synchronisation (SUN, classpath)

Genauere Betrachtung der eclipse Quellen

- Umfang
- Zeitaufwand für die Übersetzung
- Zeitaufwand für die Analyse
- Auswertung der Ergebnisse

Umfang - eclipse

- eclipse 3.0.0 [2]
- 95 jar Dateien
- 16496 Klassen in 62 Minuten analysiert
- 6013 mögliche Fehler, 59 unterschiedliche Fehlertypen
- 30 MB an Programmdateien
- 52 MB an Quelldateien

[2] eclipse-SDK-3.0-macosx-carbon.tar.gz

Auswertung - eclipse

- Auswertung von findbugs übernommen [3]
- eclipse ist an vielen Stellen durch UnitTests abgedeckt
- Trotzdem wurden echte Fehler entdeckt, 3 Endlosschleifen, 3 Eq, 55 HE, 1000 MS
- Dennoch gibt es Detektoren die nur Fehler reporten

[3] oopsla2004-slides.pdf

Betrachtung Software Projekt talk

- Umfang
- Zeitaufwand für die Analyse
- Zeitaufwand für die Auswertung
- Einschätzung des Verhältnises

Umfang - talk

- 7 Entwickler
- 82 Klassen
- Vier Gruppen
- Unterschiedliche Erfahrungen
- Wenig bis kein kein dynamisches Testen

Zeitaufwand - talk

- 25 Sekunden für 82 Klassen
- 76 potentielle Fehler
- 21 unterschiedliche Fehlertypen
- 1 schwerer Fehler sofort entdeckt
- 20 Performance- und 20 Stilprobleme

Bewertung - talk

- 60 von 60 möglichen Fehlern lohnenswert zu berichtigen
- 16 Fehler mit Synchronisation von einer Gruppe ohne Erfahrung mit Nebenläufigkeit
- Guter Anlass noch ein Mal nachzuschauen
- Echte Fehler wurden gefunden - zum Glück

Was ist es nicht?

- Ersatz für den Korrektheitsbeweis
- Ersatz für die dynamische Analyse
- Ersatz für eigene Gedanken und Überlegungen

Was sind die Schwächen?

- Per Entwurf ungenaue Ergebnisse
- Möglicherweise überwiegen die Falschmeldungen für einige Detektoren

Was sind die Stärken?

- Geschwindigkeit der Analyse
- Echte Fehler werden trotz der einfachen Detektoren gefunden
- Gute Integration in eclipse und ant
- Alle Schwächen sind beherrschbar

Empfehlung

- findbugs bietet die Chance Fehler zu finden
- findbugs findet Fehler die durch Hinschauen schwer zu finden sind
- findbugs findet unabhängig von der Qualität der Testfälle Fehler
- findbugs ist eine Chance bessere Software zu schreiben
- Gute Programmierer erweitern findbugs
- Gute Programmierer denken nach
- All dies gilt für andere Sprachen und Werkzeuge