Werkzeuge zur statischen Codeanalyse

Fusselfreier Code

Tools zur statischen Analyse durchkämmen selbst gestrickten Code nach Fusseln und Knötchen. Was Open-Source-Werkzeuge für C, C++, Java und Python leisten, beschreibt dieser Artikel. Tim Schürmann

Beim Testen von Software hilft der Compiler nur eingeschränkt. Er prangert meist lediglich Tipp- und Syntaxfehler an. Speicherlecks, endlos laufende Schleifen oder Race Conditions dagegen erkennt der Entwickler erst, wenn das Programm läuft und mit den zuvor aufwändig erdachten Testdaten gegen die Wand fährt.

Zumindest einen Teil der aufgezählten Probleme kann entdecken, wer den Quellcode genau unter die Lupe nimmt. Das Paradebeispiel zeigt, wie in C und C++ aus dem Vergleich

```
if ( a == b ) ...
```

ein vergessenes Gleichheitszeichen eine Zuweisung macht:

```
if ( a = b ) ...
```

Gerade bei schwer lesbarem und umfangreichem Quellcode stößt der Mensch dabei jedoch an seine Grenzen. Ein falsch gesetztes Semikolon entdecken ist ähnlich schwierig, wie eine Nadel im Heuhaufen zu finden – wenn man denn überhaupt daran denkt, alle Strichpunkte zu überprüfen. Zudem kostet eine solche manuelle Inspektion extrem viel Zeit.

Automatische Assistenten

Glücklicherweise gibt es zahlreiche Werkzeuge, die eine solche statische Codeanalyse in Sekundenbruchteilen durchführen. In der Regel parsen sie zunächst den Quellcode ähnlich wie der Compiler. Mit dem dabei aufgebauten Wissen können sie dann nicht nur den Programmablauf auf Probleme abklopfen, sondern auch eine aufwändige Datenflussanalyse durchführen.

In seltenen Fällen starten sie sogar selbst generierten Objektcode – auch wenn das eigentlich nicht mehr zur statischen Codeanalyse gehört (siehe Grundlagenartikel in diesem Schwerpunkt).

Welche Fehler sich damit prinzipiell aufdecken lassen, fasst der **Kasten „Fundstücke"** zusammen. Manche Werkzeuge prüfen allerdings nur auf einen Teil dieses Katalogs. Einige erlauben es dem Entwickler immerhin, nachträglich weitere Prüfregeln hinzuzufügen. Häufig muss er sich dabei jedoch in eine kryptische Beschreibungssprache einarbeiten oder selbst entsprechende Prüfroutinen programmieren.

Stil und Statistik

Wenn solch ein Tool schon mal den Quellcode in den Fingern hat, kann es gleich noch ein paar andere Dinge damit anstellen. So begutachten manche Werkzeuge die Formatierung des Quellcodes und übernehmen damit die Aufgaben eines Style-Checkers. Andere legen sogar umfangreiche Statistiken über den Quellcode an.

Fast jedes Werkzeug bewertet oder klassifiziert die entdeckten Probleme. Im einfachsten Fall unterscheidet ein Tool nur Fehler und Warnungen, differenzierter sind die ebenfalls beliebten Punktesysteme oder Noten. Besonders kritische Stellen erhalten einen höheren Punktwert als weniger gefährliche.

Da Werkzeuge zur statischen Codeanalyse direkt den Quellcode inspizieren, müssen sie die Eigenheiten der jeweiligen Programmiersprache und deren typische Problemzonen kennen. Während

Online PLUS

Informationen zur statischen Codeanalyse für PHP finden Sie unter [http://www.linux-magazin.de/plus/2012/08].

```
test.c:19:5: Test expression for if not boolean, type int: a = 1
  Test expression type is not boolean or int. (Use -predboolint to inhibit
  warning)
test.c:23:13: Function printtext expects arg 1 to be int gets char *: c
  Types are incompatible. (Use -type to inhibit warning)
test.c:23:3: Function printtext called with 2 args, expects 1
test.c:23:13: Variable c used before definition
test.c:23:3: Return value (type int) ignored: printtext(c, 2)
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (Use -retvalint to inhibit warning)
test.c:27:2: Path with no return in function declared to return int
test.c:3:1: Function exported but not used outside test: printtext
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
   test.c:4:2: Definition of printtext

Finished checking --- 11 code warnings
tim@ubuntu:~$
```

Abbildung 1: Das Werkzeug Splint entdeckt in Listing 1 viele zumindest problematische Codeteile. GCC in der Standardeinstellung dagegen übersetzt den Code ohne Beanstandung.

der Programmierer etwa in C und C++ manuell den Speicher verwalten muss, greift ihm in Java die Garbage Collection unter die Arme. Meist beschränken sich Werkzeuge daher auf eine einzige Programmiersprache.

Schlüpfrige Typen

Bei dynamisch typisierten (Skript-)Sprachen wie Python gestaltet sich die statische Codeanalyse etwas schwieriger. Dort steht der Typ einer Variablen erst zu Laufzeit fest und darf sich je nach Programmiersprache sogar während der Ausführung ändern. In der Folge warnen die Werkzeuge zwangsweise häufiger vor Stellen, die eigentlich korrekt sind – sie erzeugen so genannte False Positives. Übrigens sind auch statisch typisierte Programmiersprachen nicht vor Fehlalarmen gefeit. Bessere Werkzeuge erlauben deshalb, bestimmte Tests oder Codepassagen von der Prüfung auszunehmen.

Werkzeuge zur statischen Codeanalyse haben durchweg die Form von Kommandozeilen-Programmen, die der Programmierer auf den Quellcode ansetzt und dann umgehend die Verbesserungsvorschläge und Warnungen erhält. Für einige gibt es auch eine grafische Benutzeroberfläche. Deren Entwickler behandeln sie jedoch meist recht stiefmütterlich, viele GUIs nutzen zudem das eigentlich nicht mehr ganz zeitgemäße Tk-Toolkit. Nützlicher sind da schon Plugins für bekannte IDEs wie Eclipse: Im Idealfall sieht der Coder dann schon beim Tippen, wo ein Programm klemmt.

Flusensiebe

Eines der ersten Werkzeuge zur statischen Codeanalyse war Lint [1]. Es entstand bereits Ende der 70er Jahre und kümmerte sich ausschließlich um C-Programme. Sein Name leitet sich vom englischen Wort für Fusseln ab. Es fängt gewissermaßen lose Partikel aus einem Programm wie ein Flusensieb im Wäschetrockner. Lint war so bekannt, dass auch ähnlich arbeitende Werkzeuge für andere Sprachen diesen Namen erhielten. So findet man etwa im Paketmanager von Ubuntu 12.04 ein Lint für Java (namens Jlint), Python (Pylint) und sogar Haskell (Hlint). Diese Werkzeuge sind jedoch weder verschwägert noch verwandt mit dem ursprünglichen Lint.

Fundstücke

Einige typische problematische Konstrukte, die eine statische Codeanalyse aufdecken kann:

- Nicht initialisierte oder deklarierte Variablen
- Nie verwendete Variablen, Funktionen und Klassen
- Variablen, die zwischen zwei Zuweisungen nicht verwendet werden
- Zeiger ins Nichts, Zugriff auf nicht existierende Objekte
- Überschrittene Array- oder Listengrenzen
- Probleme bei der Speicherverwaltung einschließlich Speicherlecks und Buffer Overflows
- Code, der niemals erreicht werden kann
- Überflüssige If-Abfragen
- Endlos laufende Schleifen
- Ergebnisse einer Funktion oder Methode werden nicht verwendet
- Probleme beim Überladen von Methoden und Funktionen
- Fehlende oder falsch verwendete Konstruktoren oder Destruktoren
- Doppelte Codepassagen (unbedarftes Copy & Paste)
- Probleme in den Vererbungsbeziehungen (zum Beispiel Zyklen)
- Inkorrekte Implementierung von Schnittstellen
- Inkorrektes Werfen und Fangen von Exceptions
- Falsche Anzahl von Parametern und Parametertypen, insbesondere bei schwach typisierten Sprachen wie Python

C und C++

Eine moderne Fassung für C-Programme heißt Splint. Listing 1 zeigt ein mit zahlreichen trivialen Fehlern behaftetes, leicht verständliches C-Programm. Wie Abbildung 1 beweist, entdeckt Splint darin stolze elf Probleme. Der Compiler dagegen übersetzt in Standardeinstellung denselben Code, ohne zu murren. Lässt man »gcc« wie in Abbildung 2 jedoch etwas penibler zu Werke gehen, bemerkt auch dieser sechs Probleme.
Wie das Beispiel zeigt, führen heute fast alle Compiler selbst eine statische Code-

Listing 1: Fehlerhaftes »test.c«

```
01 #include <stdio.h>
02
03 printtext(text)
04 {
05         printf("%d", text); /* Rückgabewert von
   printf nicht genutzt */
06 }
07
08 int main ()
09 {
10         int a;
11         char* c;
12
13         /* Nutzung von a ohne vorherige Zuweisung: */
14         while (a != 0); /* Falsches Semikolon,
   dadurch Endlosschleife */
15         {
16                 a=a-1;
17         }
18
19         if(a=1) /* Zuweisung statt Vergleich */
20         {
21                 /* Zeiger ins Nichts übergeben */
22                 /* falsche Parameteranzahl */
23                 printtext(c, 2);
24         }
25
26         /* Kein Rückgabewert für main() */
27 }
```

```
tim@ubuntu: ~
tim@ubuntu:~$ gcc -Wall test.c
test.c:3:1: Warnung: Rückgabetyp ist auf »int« voreingestellt [-Wreturn-type]
test.c: In Funktion »main«:
test.c:19:2: Warnung: Um Zuweisung, die als Wahrheitswert verwendet wird, werden Klammern empfo
hlen [-Wparentheses]
test.c:27:1: Warnung: Kontrollfluss erreicht Ende von Nicht-void-Funktion [-Wreturn-type]
test.c: In Funktion »printtext«:
test.c:6:1: Warnung: Kontrollfluss erreicht Ende von Nicht-void-Funktion [-Wreturn-type]
test.c: In Funktion »main«:
test.c:14:8: Warnung: »a« wird in dieser Funktion uninitialisiert verwendet [-Wuninitialized]
test.c:23:12: Warnung: »c« wird in dieser Funktion uninitialisiert verwendet [-Wuninitialized]
tim@ubuntu:~$
```

Abbildung 2: Die Arbeitsteilung ist nicht so streng: Auch Compiler, wie hier der GCC, führen eine mehr oder weniger umfangreiche statische Analyse durch.

analyse durch. Meist kommen sie jedoch nicht an die speziellen Prüfwerkzeuge heran. Beim GCC-Compiler sind die Fehlermeldungen zudem teilweise verklausuliert oder nichtssagend.

Ganz anders sein Konkurrent Clang: Das C- und C++-Frontend für den modernen Compiler-Unterbau LLVM enthält bereits einen äußerst mächtigen statischen Code-Analysierer namens »scan-build« (Abbildung 3). Der nimmt aber nur C- und Objective-C-Code unter die Lupe und legt den Fokus zudem auf die Mac-OS-X-Entwicklung. Doch kann es sich lohnen, einen Blick auf die verfügbaren Compiler und Interpreter zu werfen.

Neben Splint existieren zahlreiche weitere statische Code-Analysierer für C und C++, die sich ebenfalls in den Repositories der Distributionen finden. Dabei stößt man erstaunlich oft auf ziemlich betagte Tools. Beispielsweise erschien die aktuelle Splint-Version bereits am 12. Juli 2007, die letzten Änderungen in der Entwicklerversion fanden 2009 statt. Die Dokumentation ist sogar auf das Jahr 2003 datiert.

Der Konkurrent Flawfinder wirbt zwar mit C++-Unterstützung, weist aber ein ähnlich hohes Alter wie Splint auf und kennt daher noch nicht den neuen C++11-Standard. Dafür stammt die aktuelle Version von Frama-C aus dem Oktober 2011, doch wie sein Name schon andeutet, analysiert das Werkzeug ausschließlich C-Programme, lässt sich aber über Plugins um weitere Funktionen ergänzen, Anwender können zudem kostenpflichtigen Support erwerben.

Ein echter Lichtblick ist das derzeit äußerst populäre Cppcheck, das angeblich schon Fehler in Linux-Kernel und Mplayer aufgedeckt hat. Das Werkzeug kennt bereits C++11, nimmt sich der STL an und prüft sogar, ob der Quellcode vielleicht auf 64-Bit-Systemen Probleme bereitet. Seine Meldungen kann Cppcheck nicht nur auf die Konsole schreiben, sondern sie auch in anderen Formaten ausgeben, etwa in XML (Abbildung 4). Das ist besonders nützlich, wenn der Entwickler die Meldungen in anderen Programmen weiterverarbeiten möchte.

Cppcheck lässt sich online ausprobieren [2]: Einfach den Quellcode in das Webformular eintippen und checken lassen. Die Entwickler haben sich das ehrgeizige Ziel gesetzt, alle ein bis zwei Monate eine neue Version herauszubringen. In den IDEs Code::Blocks und Code Lite ist Cppcheck bereits enthalten. Plugins für Eclipse, Hudson, Jenkins und sogar Gedit stehen auf der Homepage bereit.

Apropos Eclipse: Zu CDT, der Erweiterung für die C-Entwicklung, gehört ab Version 7.0 eine statische Codeanalyse namens Codan. Sie bekrittelt, wie in Abbildung 5 gezeigt, potenzielle Fehler bereits bei der Eingabe. Darüber hinaus darf der Anwender in den Einstellungen ganz bequem selbst entscheiden, ob ein Prüfkriterium nur eine Warnung oder gleich einen schwerwiegenden Fehler auslöst. Außerdem kann Codan einen einheitlichen Codestil sicherstellen. Tabelle 1 fasst eine kleine Auswahl freier Werkzeuge für C und C++ zusammen.

Java

Wer in Java programmiert, deckt mit einem guten Compiler schon eine ganze Reihe typischer Fehler auf. Beispielsweise bemängelt die Variante aus dem Open JDK fehlende Rückgabewerte und nicht initialisierte Variablen. Dennoch bleiben genügend Baustellen aus dem Kasten „Fundstücke" übrig, etwa endlos laufende Schleifen und unerreichbare Zweige. Bekannte kostenfreie Werkzeuge für Java stellt Tabelle 2 vor.

Besonders beliebt ist das Tool PMD, bei dem die Entwickler übrigens offen lassen, was die Abkürzung bedeuten soll. Es prangert auch umständlich aufgebaute Codeteile an, etwa komplexe Schleifen, die sich in schnellere »while«-Blöcke umwandeln ließen. Alle Tests unterteilt PMD thematisch in so genannte Regelsätze (Rulesets).

Tabelle 1: Werkzeuge für C und C++ (Auswahl)

Name	Versionsnummer	Homepage	Lizenz	Typ	Sprachen	Prüfungen erweiterbar
Codan	CDT 8.1 (April 2012)	[http://wiki.eclipse.org/CDT/designs/StaticAnalysis]	EPL 1.0	Plugin für Eclipse	C, C++	ja
Cppcheck	1.54 (vom 25.4.2012)	[http://cppcheck.sourceforge.net]	GPLv3	Kommandozeile, GUI	C, C++	nein
Flawfinder	1.27 (vom 17.1.2007)	[http://www.dwheeler.com/flawfinder/]	GPLv2	Kommandozeile	C, C++	nein
Frama-C	20111001 (vom 01.10.2011)	[http://frama-c.com]	LGPLv2	Kommandozeile, GUI	C	ja
Scan-build	3.1 (vom 22.5.2012)	[http://clang-analyzer.llvm.org]	LLVM Release License	Kommandozeilen, Teil von Clang	C, Objective-C	nein
Splint	3.1.2 (vom 12.7.2007)	[http://www.splint.org]	GPLv2	Kommandozeile	C	nein

```
tim@ubuntu: ~
tim@ubuntu:~$ scan-build gcc test.c
scan-build: 'clang' executable not found in '/usr/share/clang/scan-build/bin'.
scan-build: Using 'clang' from path: /usr/bin/clang
test.c:14:11: warning: The left operand of '!=' is a garbage value
        while (a != 0);         /* Falsches Semikolon, dadurch Endlosschleife */
               ~ ^
test.c:16:3: warning: Value stored to 'a' is never read
                a=a-1;
                ^ ~~~
2 warnings generated.
scan-build: 2 bugs found.
scan-build: Run 'scan-view /tmp/scan-build-2012-06-13-2' to examine bug reports.
tim@ubuntu:~$
```

Abbildung 3: Das Tool »scan-build« von LLVM zeigt sogar auf kritische Anweisungen und nennt nicht nur die Zeilennummer.

```
tim@ubuntu: ~
tim@ubuntu:~$ cppcheck test.c
Checking test.c...
[test.c:14]: (error) Uninitialized variable: a
tim@ubuntu:~$ cppcheck --xml test.c
<?xml version="1.0" encoding="UTF-8"?>
<results>
Checking test.c...
<error file="test.c" line="14" id="uninitvar" severity="error" msg="Uninitialized variable: a"/>
</results>
tim@ubuntu:~$
```

Abbildung 4: Cppcheck nennt immer nur den ersten Fehler im Programm, gibt diesen auf Wunsch aber auch im XML-Format aus.

Beim Aufruf muss der Benutzer eine Liste mit den anzuwendenden Regelsätzen angeben. Dummerweise gibt es standardmäßig keinen Regelsatz, der alle Prüfungen einschließt. Damit erhält der Anwender entweder an der Kommandozeile einen ziemlichen Bandwurmbefehl oder muss erst umständlich einen eigenen Regelsatz erstellen. Andererseits kennt PMD auch Tests, die auf Eigenheiten des Android-SDK oder J2EE eingehen. Gefundene Fehler gibt das Werkzeug nicht nur als reinen Text aus, sondern auch in anderen Formaten, etwa in XML oder HTML.

Checkstyle überwacht eigentlich die Einhaltung von Stilvorgaben. So meckert es, wenn Javadoc-Kommentare fehlen oder Namen von den Schreibkonventionen abweichen [3]. Seit Version 3 finden aber auch immer mehr andere Prüfungen Eingang in das Werkzeug. So prangert es mittlerweile duplizierte Codeteile an, prüft die Sichtbarkeit von Klassenmethoden und spürt leere Anweisungsblöcke auf. Ähnlich wie PMD startet Checkstyle nur, wenn der Anwender ihm eine Konfigurationsdatei mit den durchzuführenden Prüfungen an die Hand gibt.

Die Software Hammurapi rühmt sich damit, für besonders große Projekte geeignet zu sein. Das Werkzeug lässt sich entweder als Ant-Task oder über ein mitgeliefertes Eclipse-Plugin verwenden. Letztgenanntes dürfen nur nichtkommerzielle Projekte kostenlos nutzen. Außerdem wollte es in den Tests der Redaktion nicht mehr mit der aktuellen Eclipse-Version zusammenarbeiten. Generell verlangt Hammurapi eine im Hintergrund laufende HSQLDB-Datenbank sowie Apache Tomcat.

Bytecode

Zu den zahlreichen Werkzeugen für Java, die anstelle des Quellcodes den Bytecode inspizieren, zählt unter anderem Jlint. Es führt eine Datenflussanalyse durch und deckt auch Synchronisationsprobleme auf. Witzigerweise analysiert es zwar Java-Code, ist aber selbst in C++ geschrieben. Dem Jlint-Paket liegt noch das Werkzeug Anti C bei, das einfache Fehler und Probleme in C- und C++-Quellcode entdeckt.

Ebenfalls nur Bytecode betrachtet Findbugs. Es startet von Haus aus mit einer schicken Benutzeroberfläche. Die verlangt, dass der Anwender zunächst ein Projekt anlegt, dem er dann alle Bytecode-Dateien hinzufügt. Das macht das Testen in komplexen Projekten zwar bequem, ist aber umständlich, wenn man nur eine einzelne Datei prüfen möchte. Glücklicherweise arbeitet Findbug auch auf der Kommandozeile.

Für fast alle in Tabelle 2 genannten Java-Werkzeuge existieren passende Plugins für Eclipse, häufig sind auch noch Pendants für weitere Entwicklungsumgebungen wie Intelli J und Netbeans zu haben. Hier muss man allerdings etwas auf die Versionsnummern achten, insbesondere wenn die Plugins nicht von den Werkzeugentwicklern selbst stammen. Beispielsweise waren die Plugins für Checkstyle zum Redaktionsschluss weitgehend veraltet.

Python

Unter Python-Programmierern sind drei Werkzeuge besonders beliebt (Tabelle 3). Sie liegen auch den meisten großen Linux-Distributionen bei. Neben den üblichen Fehlern aus dem Kasten „Fundstücke“ weisen sie auch auf Python-typische Probleme hin, etwa wenn wegen falscher Einrückungen eine Funktion scheinbar doppelt definiert ist.

Pychecker lässt sich sogar direkt in eigene Python-Programme einbinden. Dann

Tabelle 2: Werkzeuge für Java (Auswahl)

Name	Versionsnummer	Homepage	Lizenz	Typ	Sprachen	Prüfungen erweiterbar
Checkstyle	5.5 (vom 5.11.2011)	[http://checkstyle.sourceforge.net]	LGPLv2.1	Kommandozeile	Java	ja
Findbugs	2.0.1-rc2 (vom 11.5.2012)	[http://findbugs.sourceforge.net]	LGPL	Kommandozeile, GUI	Java	nein
Hammurapi	5.7.0	[http://www.hammurapi.biz]	Apache-Lizenz 2.0, LGPL, Hammurapi Group Community License	Eclipse-Plugin, Ant-Task	Java	ja
Jlint	3.1.2 (vom 11.1.2011)	[http://jlint.sourceforge.net]	GPLv2	Kommandozeile	Java (Anti C: C, C++, Objective-C)	nein
PMD	5.0.0 (vom 1.5.2012)	[http://pmd.sourceforge.net]	eigene (BSD-ähnlich)	Kommandozeile	Java	ja

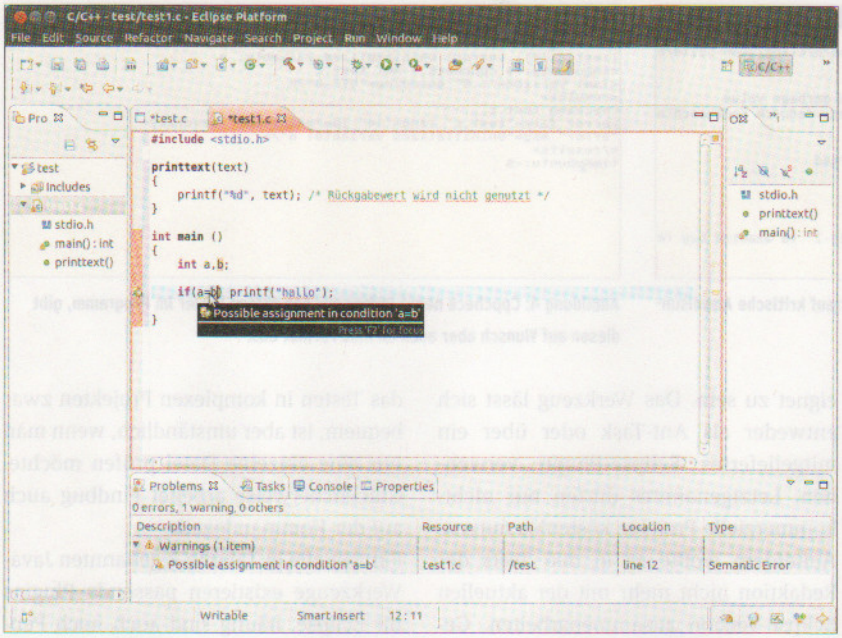

Abbildung 5: Praktisch: Plugins für integrierte Entwicklungsumgebunden, hier Codan in Eclipse, warnen schon bei der Eingabe von Quellcode vor Problemen.

prüft es automatisch alle Module, die nach der entsprechenden »import«-Zeile folgen. Doch übersetzt Pychecker das zu untersuchende Programm in Bytecode und führt diesen sogar aus. Das ist eher unerwünscht, weil das Programm dann schon bei der vorbeugenden Analyse abläuft und dabei vielleicht Nutzdaten ändert. Daneben gibt es eine GUI-Version des Programms (Abbildung 6)

Hier gibt's Noten

Als Alternative bietet sich das bereits erwähnte Pylint an. Neben den üblichen Tests prüft es auch, ob der Quellcode dem PEP8-Styleguide folgt [4]. Darüber hinaus generiert Pylint zahlreiche Statistiken und vergibt sogar eine Gesamtnote für die Codequalität. Interessant ist das mitgelieferte Werkzeug »pyreverse«, das aus dem Quellcode ein UML-Diagramm im Dot-Format von Graphviz erstellt. Pylint importiert die zu analysierenden Dateien, was unter Umständen einige ungewollte Nebenwirkungen auslöst. Der Konkurrent Pyflakes liest hingegen die Datei nur ein, bietet dafür aber keine Stilprüfung.

Eines reicht selten

Werkzeuge zur statischen Codeanalyse sorgen nicht automatisch für fehlerfreien Code. Sie zeigen lediglich potenzielle Gefahrenquellen beziehungsweise Anomalien auf. Die Hinweise müssen zudem nicht immer stimmen: Gerade bei komplexem Code melden die Werkzeuge Stellen, die eigentlich korrekt ablaufen (False Positives). Erschwerend kommt hinzu, dass die Entwickler fast nie verraten, welche Code-Anomalien ihre Werkzeuge prüfen und wann sie diese bemängeln. Häufig findet man auf der Projekt-Homepage nur eine Liste mit den wichtigsten Problemen, die das Programm abklopft. Ausführliche Erläuterungen wie bei Splint und Jlint sind selten.

Ebenfalls häufig im Unklaren bleibt, welche Sprachversion das Werkzeug unterstützt – dabei gibt es etwa zwischen C++ in der Fassung von 1998 und dem brandneuen C++11 gravierende Unterschiede. Es bietet sich folglich an, den Quellcode von mehreren Werkzeugen analysieren zu lassen. Was übrigens passieren kann, wenn man eines der Werkzeuge auf einen Konkurrenten ansetzt, zeigt ein amüsanter Blogeintrag unter [5]. *(mhu)* ■

Infos

[1] Wikipedia-Eintrag zu Lint: [http://de.wikipedia.org/wiki/Lint_%28Programmierwerkzeug%29]
[2] Online-Angebot von Cppcheck: [http://cppcheck.sourceforge.net/demo/]
[3] Wikipedia-Eintrag zu Javadoc: [http://de.wikipedia.org/wiki/Javadoc]
[4] PEP8-Styleguide: [http://www.python.org/dev/peps/pep-0008/]
[5] Luke Plant, „Pychecker vs Pylint vs Django": [http://lukeplant.me.uk/blog/posts/pychecker-vs-pylint-vs-django/]

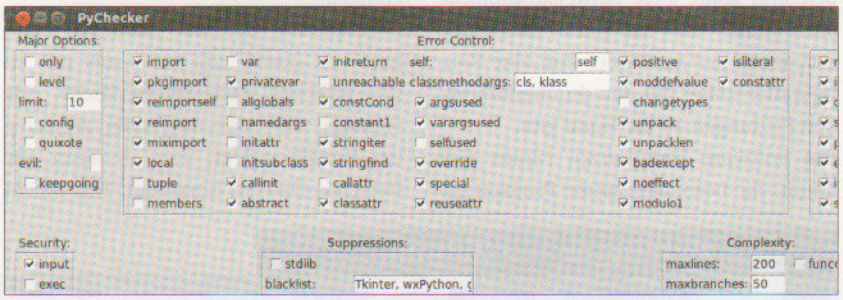

Abbildung 6: Pychecker bringt eine kleinteilige grafische Oberfläche mit, die auf Python-Tk basiert und noch deutlich in den Kinderschuhen steckt.

Tabelle 3: Werkzeuge für Python (Auswahl)

Name	Versionsnummer	Homepage	Lizenz	Typ	Sprachen	Prüfungen erweiterbar
Pychecker	0.8.19 (vom 8.1.2011)	[http://pychecker.sourceforge.net]	eigene (BSD-ähnlich)	Kommandozeile, GUI	Python	nein
Pyflakes	0.5.0 (vom 6.9.2011)	[https://launchpad.net/pyflakes]	MIT-Lizenz	Kommandozeile	Python	nein
Pylint	0.25.1 (vom 8.12.2011)	[http://pypi.python.org/pypi/pylint]	GPLv2	Kommandozeile, GUI	Python	ja