Ein Eclipse-Plug-in zur statistischen Analyse der Verwendung von Typen in Java-Programmen

Master-Arbeit im Studiengang
Master of Computer Science

vorgelegt von

Gerd Eichhorn
Marco-Polo-Weg 30
70439 Stuttgart
Matrikelnummer 7117795
Gerd.Eichhorn@Fernuni-Hagen.de

Betreuer:

Prof. Dr. F. Steimann

26. Februar 2008
Danksagungen

Mein besonderer Dank gilt Herrn Prof. Dr. Friedrich Steimann für die Übernahme und Betreuung dieser Arbeit und die zahlreichen weiterführenden Hinweise.

Außerdem geht mein Dank an Herrn Dr. Eugen Grycko vom Fachbereich Mathematik für die großzügige Unterstützung bei der statistischen Analyse und Auswertung.

Danke auch an Frau Dr. Ursula Scheben und Frau Dr. Daniela Keller für die Durchsichten, Korrekturen und Verbesserungsvorschläge dieser Arbeit.
# Inhaltsverzeichnis

1 Inhaltsangabe 1

2 Einleitung 2
   2.1 Motivation 2
   2.2 Beitrag der Arbeit 2
   2.3 Aufbau der Arbeit 3

3 Grundlagen 4
   3.1 Einleitung 4
   3.2 Metrik - Number of Declarations (NOD) 4
   3.3 Abstract Syntax Tree 7
   3.4 Besucher-Entwurfsmuster 10

4 Statistik 12
   4.1 Motivation 12
   4.2 Grundlagen der Bayes-Statistik 12
   4.3 Grundbegriffe der Wahrscheinlichkeitsrechnung: Zufallsvariable, Verteilungsfunktion und Dichtefunktion 14
   4.4 Wahl der Verteilungsfunktion 16
      4.4.1 Zipf/Estoup-Verteilung 16
      4.4.2 Yule-Verteilung 17
   4.5 Statistisches Analyse-Programm 18
   4.6 Rekursionsalgorithmus 20
   4.7 Untersuchung der Beispielprogramme 22
      4.7.1 Beispielprogramme 22
      4.7.2 Untersuchung 23

5 Ergebnisse 25
   5.1 Verteilungen 25
   5.2 Abweichungen 29
   5.3 Fazit 29

6 Typ-Analyse-Plug-in (TAP) 31
   6.1 Anforderungen 31
      6.1.1 Type Diagram View für die grafische Ausgabe der NOD- und NOI-Metrik 33
A.4.3 Erstellen der JAR-Datei ........................................ 70
A.4.4 Erstellen der EXE-Datei ....................................... 74

Erklärung ................................................................. 78
1 Inhaltsangabe


Als Entwicklungsplattform wurde Eclipse, eine Open-Source-Entwicklungsumgebung, ausgewählt, die vor allem einen guten Erweiterungsmechanismus basierend auf Plugins bietet. Das statistische Analyseprogramm wurde ebenfalls mit Hilfe von Eclipse als Erweiterung des JDT’s als eigenständige Java-Applikation realisiert.
2 Einleitung

2.1 Motivation


2.2 Beitrag der Arbeit


Die hierbei gewonnenen Daten können auch als Eingabe für weitere Programme/Plug-ins dienen. So könnten z.B. automatisierte Refactorings auf Java-Quellen durchgeführt werden, die als Eingabe die Daten der Typ-Analyse benötigen.

Für den zweiten, statistischen Teil wurde ein Java-Programm entwickelt, welches eine Aussage über die, aus der Literatur bekannten Verteilungen erlaubt. Hierzu wird auf die sog. „Bayessche Statistik“ zurückgegriffen, die eine hinreichend genaue Interpretation der Ergebnisse erlaubt.

2.3 Aufbau der Arbeit

3 Grundlagen

3.1 Einleitung


3.2 Metrik - Number of Declarations (NOD)

Die Metrik, die die Verwendung eines Typs zählt, basiert auf einer statischen Typanalyse von Java-Programmen und kann folgendermaßen angegeben werden:

\[ NOD(A) = | \{ d \mid d \in D \land A \in d \} | \]

wobei A ein Typ ist und D die Menge aller betrachteten Deklarationselemente.

Zu D gehören in Konstruktoren und Methodenden vorkommende Parameterdeklarationen, Deklarationen des Rückgabetyps von Methoden sowie Variablendeklarationen wie in Kapitel 3.1 eingeführt.

- Falls d eine Parameter- oder Variablendeklaration repräsentiert, bedeutet A ∈ d, daß A in d als der statische Typ des Parameters bzw. der Variablen festgelegt wird.

- Falls d die Deklaration des Rückgabetyps einer Methode repräsentiert, bedeutet A ∈ d, daß A als Rückgabetyp angegeben ist.
Der folgende Algorithmus in Form von Pseudocode beschreibt, wie die Anzahl der Verwendungen innerhalb des Analyse-Plug-ins bestimmt wird:

```
for each declarationElement of actualProject do {
    Type := getTypeOfDeclarationElement;
    CountingArray[Type]++;
}
```

Listing 3.1: Algorithmus zur Berechnung der Metrik NOD

Unter die bei der Analyse erfassten Typen fallen nicht nur die im untersuchten Quelltext deklarierten Schnittstellen-, Klassen- und Aufzählungstypen, sondern sämtliche, im Quelltext i.o. Sinne verwendeten Typen, also auch Basistypen wie z.B. int und float etc. oder vordefinierte Typen wie java.lang.String etc. Die im untersuchten Quelltext deklarierten Typen besitzen nur insofern eine Sonderstellung, als daß sie später bei der Auswertung der Analyseergebnisse auch dann aufgeführt werden, wenn sie im untersuchten Quelltext an keiner Stelle verwendet werden. Ihre Verwendung wird dann mit 0 angegeben. Der Sinn dieser Sonderstellung wird weiter unten erläutert.

Die folgende Aufzählung gibt noch einmal alle Typen an, die in die Analyse eingehen:

- Rückgabetypen von Methoden
- Typen von Methoden- und Konstruktorparametern
- Typen von lokal in Methoden deklarierten Variablen
- Typen in Attributdeklarationen
- Klassentypen, die im untersuchten Quelltext deklariert werden
- Schnittstellentypen, die im untersuchten Quelltext deklariert werden
- Aufzählungstypen, die im untersuchten Quelltext deklariert werden.

Um diese Typen zu bestimmen, müssen bei der Analyse also folgende Javakonstrukte untersucht werden:

- Konstruktordeklarationen
- Methodendeklarationen
- Deklarationen von Variablen innerhalb von Methodenrumpfen
- Attributdeklarationen
- Klassendeklarationen
- Schnittstellendeklarationen
• Deklarationen von Aufzählungstypen

Das folgende Beispiel dient zur Verdeutlichung des Verfahrens:

```java
public class SquareRoot {
    private double squareRoot;
    public double getSquareRoot() { return squareRoot; }

    public void calcSquareRoot(double d) {
        String s;
        squareRoot = Math.sqrt(d);
        s = "Die Quadratwurzel von " + d + " ist: " + squareRoot;
        System.out.println(s.toString());
    }
}
```

Listing 3.2: Beispielcode mit relevanten Deklarationselementen

Das Beispiel hat das folgende Ergebnis:

<table>
<thead>
<tr>
<th>Typ</th>
<th>Anzahl der Verwendungen des Typs</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>3</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>1</td>
</tr>
<tr>
<td>SquareRoot</td>
<td>0</td>
</tr>
</tbody>
</table>

Tabelle 3.1: Auswertung des Beispielcodes

Eine Besonderheit ist hier anzumerken: Der letzte Typ der Tabelle, in Form der Klasse „SquareRoot“, wird hier aufgelistet, hat aber eine Anzahl von null Verwendungen, da der Typ zwar deklariert, nicht aber in anderen Deklarationselementen verwendet wird. Dies könnte bei einer späteren Untersuchung eines Programms ein Hinweis auf eine nicht benötigte Klasse sein.
3.3 Abstract Syntax Tree


Abbildung 3.1: AST-Workflow

- 1. Zu parsing Java-Quelle
- 2. Aufruf des Parsers mit Hilfe der Klasse „org.eclipse.jdt.core.dom.ASTParser“
- 3. Der „Abstract Syntax Tree“ als Resultat von Schritt 2
4a. Manipulation des ASTs entweder direkt, z.B. Ändern von Knoteneigenschaften

4b. Modifikationen in einem separaten sogenannten „Protokoll“, d.h. mit dem Aufruf von „ASTRewrite“ wird eine Kopie des ASTs erzeugt, wo anschließend die Modifikationen durchgeführt werden

5. Zurückschreiben der Änderungen

6. IDocument, ist ein sogenannter „Wrapper“ für den Quellcode und wird bei Punkt 1. und 5. verwendet


Einige beispielhaft aufgeklappte Knoten zeigen die zugehörigen Details. Z.B. zeigt die „TypeDeclaration“ den Name des Typs in der untersuchten Quelle, in diesem Fall der Name der Klasse „SquareRoot“ im Knoten „SimpleName“. Ein sogenannter „SimpleName“ ist irgendein String einer Java-Quelle, die kein Schlüsselwort, ein Boolesches Literal (true oder false) oder das „null“-Literal ist. In den weiteren Knoten innerhalb des Rumpfes „BODY_DECLARATIONS“ folgen die „FieldDeclaration“ der Membervariable „squareRoot“. Anschließend folgt die Funktion „getSquareRoot“ innerhalb einer „MethodDeclaration“. Hier ist der „Modifier“ in Form von „public“ zu erkennen, der Rückgabetyp „double“ in einem Knotentyp namens „PrimitiveType“. Primitive Typen sind alle Basistypen, die die Sprache Java zur Verfügung stellt, z.B. int, long, double, usw.
Abbildung 3.2: AST-View
KAPITEL 3. GRUNDLAGEN

3.4 Besucher-Entwurfsmuster


```java
public class TypeDeclarationVisitor extends ASTVisitor {
    ...
    public boolean visit(VariableDeclarationFragment node) {
        ...
    }
    // Überschreiben der Besucherfunktion mit dem Parameter
    // "MethodDeclaration", um an Infos zu diesem Typ zu gelangen
    public boolean visit(MethodDeclaration node) {
        // Resolve binding
        IMethodBinding methodBinding = node.resolveBinding();
        // Retrieve parameter types
        ITypeBinding[] paramTypes = methodBinding.getParameterTypes();
        ...
        // Retrieve return type
        ITypeBinding returnType = methodBinding ReturnType();
        ...
        return super.visit(node);
    }
    public boolean visit(EnhancedForStatement node) {
        ...
    }
    public boolean visit(TypeDeclaration node) {
        ...
    }
}
```

Listing 3.3: Beispiel für Besucherfunktionen zum Auswerten von AST-Daten


4 Statistik

In diesem Kapitel wird die Untersuchung der mit dem Typ-Analyse-Plug-in gewonnenen Daten mit Hilfe der Bayesschen Statistik gemacht. Es werden zwei Verteilungen angegeben und untersucht, ob die gewonnenen Daten diesen Verteilungen gehorchen.

4.1 Motivation


Mit Hilfe der Bayes-Statistik können willkürlich erhobene Daten, z.B. aus Häufigkeitszählungen von Typen in Java-Programmen, statistisch untersucht werden. Insbesondere ist mit der Bayesschen Methodik das Konzept der sogenannten „A-Priori“-Informationen verbunden. Das heißt, daß in die statistische Datenanalyse Informationen mit einbezogen werden, die bereits vor der Erhebung der Daten vorliegen. Das Ziel der Bayesschen Methodik ist die Kombination der beiden Informationsquellen, d.h. der A-Priori-Informationen, in Form von Erfahrungen oder Expertenwissen, und der Daten selbst. Im Mittelpunkt der Bayes-Statistik steht das sogenannte „Bayes-Theorem“, mit dem sich unbekannte Parameter der zugrunde liegenden und vermuteten Verteilungsfunktion schätzen lassen. Als Ergebnis einer solchen Analyse ergibt sich die sogenannte „A-Posteriori-Verteilung“. Wie der Name bereits sagt, enthält die A-Posteriori-Verteilung die durch die Beobachtung von Stichproben der zu analysierenden Daten gewonnene Information über den oder die unbekannten Parameter. Dadurch kann eine qualitative Aussage über die Wahrscheinlichkeit gemacht werden, ob die analysierten Daten einer bestimmten Verteilungsfunktion gehorchen oder auch nicht.

4.2 Grundlagen der Bayes-Statistik

In der Bayes-Statistik wird die Wahrscheinlichkeit von Aussagen definiert, wobei die Wahrscheinlichkeit ein Maß für die Plausibilität der Aussage liefert. Die Bayes-Statistik beruht auf dem sogenannten „Bayes-Theorem“. Mit seiner Hilfe lassen sich unbekannte Parameter schätzen, Konfidenzregionen für die unbekannten Parameter festlegen und die Prüfung von Hypothesen für diese Parameter ableiten. Eine Aussage hängt im allgemeinen davon ab, ob eine weitere Aussage wahr ist. Deshalb arbeitet die Bayes-Statistik ausschließlich mit bedingten Wahrscheinlichkeiten:
In der Wahrscheinlichkeitsrechnung ist \( P(A|B) \) die Wahrscheinlichkeit für ein Ereignis \( A \) unter der Bedingung, daß \( B \) eintritt. In der Bayes-Statistik gibt diese bedingte Wahrscheinlichkeit ein Maß für die Plausibilität der Aussage \( A|B \) an, d.h. daß die Aussage \( A \) richtig ist, wenn \( B \) gilt. Das Bayes-Theorem der Wahrscheinlichkeitsrechnung hat die folgende Gestalt:

Sei \( \Omega \) die Menge aller möglichen Ergebnisse (Ergebnisraum), \( E \) eine Familie von Teilmengen von \( \Omega \), die eine \( \rho \)-Algebra bildet (Ereignisalgebra) und \( P : E \rightarrow \mathbb{R}_+ \) ein Wahrscheinlichkeitsmaß. Sei \( A \in E \) mit \( P(A) > 0 \) und \( \{B_1, ..., B_n \} \) mit \( B_j \in E \) eine disjunkte Zerlegung von \( \Omega \) mit \( P(B_i) > 0 \) für \( i = 1, ..., n \). Dann gilt:

\[
P(B_k|A) = \frac{P(A|B_k) \cdot P(B_k)}{\sum_{i=1}^{n} P(A|E_i) \cdot P(E_i)}
\]

(4.1)

Eine typische Anwendung wie zum Beispiel in der Medizin ergibt sich, wenn man die Wahrscheinlichkeit \( P(\text{Ursache}|\text{Symptom}) \) bestimmen will, aber viel leichter etwas über \( P(\text{Symptom}|\text{Ursache}) \) herausfinden kann.

Praktisches Beispiel:
Es werden die folgenden Bezeichnungen verwendet:
\( B_1 \) sei das Ereignis eines defekten Lenkgetriebes.
\( B_2 \) sei das Ereignis eines \textbf{nicht} defekten Lenkgetriebes.
\( A \) sei das Ereignis eines ausgewechselten Lenkgetriebes.

Die folgenden Informationen sind gegeben:
\[
P(B_1) = \frac{20}{200000} = 0.0001 \quad P(A|B_1) = 0.99 \quad P(A|B_2) = 0.01
\]

Gesucht ist die Wahrscheinlichkeit
\[
P(B_1|A) = P(\text{Lenkgetriebe defekt}|\text{Lenkgetriebe ausgewechselt})
\]

Mit dem Theorem von Bayes folgt nun:

\[
P(B_1|A) = \frac{P(A|B_1) \cdot P(B_1)}{\sum_{i=1}^{n} P(A|E_i) \cdot P(E_i)}
\]
\[
= \frac{0.99 \cdot 0.0001}{0.99 \cdot 0.0001 + 0.01 \cdot 0.9999} \approx 0.0098
\]

13
Fast alle ausgewechselten Lenkgetriebe waren demnach nicht defekt.

In der Bayes-Statistik wird diese Formel noch durch eine Aussage über zusätzliches Wissen ergänzt. Der Einfachheit halber schreiben wir diese Formel nur für die Ergebnisse A, B und C auf:

\[
P(A|BC) = \frac{P(A|C) \cdot P(B|AC)}{P(B|C)}
\] (4.2)

Dabei bedeutet:
A die Aussage über ein unbekanntes Phänomen
B die Aussage, die Information über das unbekannte Phänomen enthält
C eine Aussage über zusätzliches Wissen

Man bezeichnet:
P(A|C) als Priori-Wahrscheinlichkeit
P(A|BC) als Posteriori-Wahrscheinlichkeit und
P(B|AC) als Likelihood

Beispiel:
Laplace schätzte die Masse des Saturns anhand vorhandener astronomischer Beobachtungen seiner Umlaufbahn. Diese Daten aus dem 18. Jahrhundert ließen eine Schätzung der Masse und eine Vorhersage auf ein Prozent Genauigkeit zu. Dieser Wert wurde bis heute nur um 0.63 Prozent korrigiert.

In Formel 4.2 ist dabei:
A: Die Masse des Saturn MS liegt in einem bestimmten Intervall
B: Daten von Observatorien über gegenseitige Störungen von Jupiter und Saturn
C: „common sense“, daß die Masse des Saturn weder so klein sein kann, daß er seine Ringe verliert, noch so groß, daß er das Sonnensystem zerstört

4.3 Grundbegriffe der Wahrscheinlichkeitsrechnung:
Zufallsvariable, Verteilungsfunktion und Dichtefunktion

In vielen Fällen benötigt man quantitative Aussagen über das Auftreten bestimmter Zahlenwerte als Ergebnis eines Zufallsversuchs. Das „Ereignis“ besteht in diesem Fall darin, daß eine „zufällige“ Größe X einen vorher fixierten Wert annimmt oder (etwas allgemeiner formuliert) einen Wert in einem festgelegten Intervall liefert. Dies führt auf den Begriff der Zufallsvariablen:
Eine Funktion \( X : \Omega \rightarrow \mathbb{R} \) heißt Zufallsvariable, wenn für jedes \( c \in \mathbb{R} \) die Menge

\[
E_c := \{ \omega \in \Omega \mid X(\omega) \leq c \}
\]

(4.3)
ein Ereignis in dem betrachteten Wahrscheinlichkeitsraum ist.

Um beschreiben zu können, welche Werte eine Zufallsvariable \( X \) mit einer bestimmten Wahrscheinlichkeit annimmt, macht man folgende Begriffsbildung:

Sei \( X \) eine Zufallsvariable. Dann heißt die durch

\[
F_X(x) = P(\{ \omega \in \Omega \mid X(\omega) \leq x \}), \quad x \in \mathbb{R}
\]

(4.4)
gegebene Funktion \( F_X \) die Verteilungsfunktion der Zufallsvariablen \( X \).

Für \( b > a \) gilt:

\[
F_X(b) - F_X(a) = P(a < X \leq b)
\]

(4.5)

Eine Zufallsvariable heißt kontinuierlich (oder stetig), falls eine Darstellung der Form

\[
F_X(x) = \int_{-\infty}^{x} f(t) \, dt
\]

(4.6)

mit einer nichtnegativen integrierbaren Funktion \( f \) möglich ist. \( f \) nennt man die Dichtefunktion (oder Dichte) der Zufallsvariablen \( X \).

Wenn eine Zufallsvariable \( X \) eine Dichtefunktion \( f_X \) besitzt, dann gilt für \( a < b \):

\[
P(a < X \leq b) = \int_{a}^{b} f_X(t) \, dt.
\]

(4.7)

Verteilungsfunktion und Dichtefunktion enthalten die gleichen Informationen über eine kontinuierliche Zufallsvariable. Das Bild einer Dichtefunktion ist aber sehr viel aussagekräftiger. Man beachte aber, daß die Dichtefunktion selbst keine direkte Interpretation als Wahrscheinlichkeit erlaubt, erst die Integration führt zu Wahrscheinlichkeiten.
4.4 Wahl der Verteilungsfunktion


4.4.1 Zipf/Estoup-Verteilung

Die Zipf/Estoup-Verteilung, in der Literatur teilweise auch „Zeta“-Verteilung genannt, ergibt sich aus der Näherungsformel, wie sie in Johnson [6] beschrieben ist:

\[
\begin{align*}
  n_r &\propto r^{-(\rho+1)} \quad (r = 1, 2, \ldots; \rho > 0) \\
  P[x=r] &\propto cr^{-(\rho+1)} \quad (r = 1, 2, \ldots)
\end{align*}
\]

wobei \( n_r \) die Anzahl der Worte ist, die \( r \) mal auftreten:

\[
P[x=r] = cr^{-(\rho+1)} \quad (r = 1, 2, \ldots)
\]

mit

\[
c := [\zeta(\rho + 1)]^{-1}
\]

wobei \( \zeta(...) \) die Zeta-Funktion

\[
\zeta(x) = \sum_{n=1}^{\infty} \frac{1}{n^x} = 1 + \frac{1}{2^x} + \frac{1}{3^x} + \frac{1}{4^x} + \cdots
\]

beschreibt. Diese Verteilungsfunktion wird auch Zeta-Verteilung genannt.
4.4.2 Yule-Verteilung

Die Yule-Verteilung ist ebenfalls in Johnson [6] beschrieben und lautet wie folgt:

\[ P[x = r] = A_\rho B(r, \rho + 1) \quad (r = 1, 2, \ldots) \quad (4.12) \]

mit

\[ A_\rho = \left( \sum_{r=1}^{\infty} B(r, \rho + 1) \right)^{-1} \quad (4.13) \]

und

\[ B(\alpha, \beta) = \frac{\Gamma(\alpha) \cdot \Gamma(\beta)}{\Gamma(\alpha + \beta)} \quad (4.14) \]

als Beta-Funktion, wobei die Gammafunktion durch

\[ \Gamma(t) = \left( t \cdot e^{\gamma t} \cdot \prod_{k=1}^{\infty} \left( 1 + \frac{t}{k} \right) \cdot e^{-\frac{t}{k}} \right)^{-1} \quad \text{mit } \gamma = 0.577216\ldots \quad (4.15) \]

gegeben ist.

Der unbekannte Parameter \( \rho \) stellt hier im konkreten Fall den zu bestimmenden Parameter in der zugrunde liegenden Verteilungsfunktion dar. Dies ist der Wert auf der x-Achse, wo der zugehörige y-Wert sein Maximum erreicht. Dieser Wert stellt dann für die untersuchten Daten das optimale Ergebnis dar. Sprich, mit diesem Wert des optimalen Parameters \( \rho \) können die Daten am besten erklärt werden.
4.5 Statistisches Analyse-Programm

Grundlage des Analyseprogramms ist das Bayes-Theorem. Es wird damit eine Wahrscheinlichkeitschätzung durchgeführt, um eine Aussage über die beiden vorab vermuteten Verteilungen zuzulassen. Dazu wird der in Abschnitt 4.6 beschriebene Rekursionsalgorithmus benötigt.


Abbildung 4.1: Statistisches Analyse-Programm

Je ausgeprägter, d.h. je höher und schmaler dieser Peak ist, desto besser passen die getroffenen Annahmen über die Verteilungen auf die zu analysierenden Daten. In diesem Fall gibt es genau einen Parameterwert $\rho$ mit maximalem Gewicht. Im Sinne der Bayes-Statistik wird dieser Parameterwert als „optimaler Parameter“ zur Beschreibung der Stichprobe herangezogen. Damit hat man einen guten Hinweis, daß die vorab getroffenen
Annahmen korrekt sind. Ergibt sich kein solcher Peak, so kann man daraus schließen, daß die zuvor getroffenen Annahmen nicht zutreffend sind und die Daten keiner der beiden Verteilungen entsprechen.

Im obigen Beispiel ergibt sich „nur“ für die Yule-Verteilung ein Peak. Bei kleinen Programm kann es vorkommen, daß sich für beide Verteilungen ein Peak ergibt. Dies kann damit erklärt werden, daß sich die beiden Verteilungen recht ähnlich sind und es erst bei einer größeren Datenmenge zu einem eindeutigen Ergebnis kommt. Der ausgeprägte der beiden Peaks stellt dann einen statistischen Hinweis dar, daß die zugehörige Verteilung die Daten besser erklärt als die andere Verteilung.

Dieses Merkmal ist auch bei den anderen untersuchten Programmen der Fall, wie später bei den Ergebnissen zu sehen sein wird. Es ergibt sich immer eine eindeutige Zuordnung zu einer der beiden Verteilungen. Falls sich kein solcher Peak einstellen würde, so muß man annehmen, daß die vorliegenden Daten auf keine der beiden Verteilungen passen. Die Folge daraus wäre dann, daß die Daten einer anderer Verteilung gehorchen, die die Daten besser erklärt. Es könnte aber auch sein, daß es sich um eine neue, unbekannte Verteilungsfunktion handelt.


Das komplette Listing des statistischen Analyseprogramms befindet sich im Anhang A.4.1.
KAPITEL 4. STATISTIK

4.6 Rekursionsalgorithmus

Hier folgt der in Moeschlin et al. [13] dargestellte Rekursionsalgorithmus. Grundlage für das Folgende ist die Kollektion

\[ W := \{ P(\gamma;.) \mid \gamma \in G \} \quad \text{(4.16)} \]

möglicher Stichprobenverteilungen, wobei im diskreten Fall \( P[.,.] \) durch eine \((r \times m)\)-Matrix \( p[.,.] \) gegeben ist. Nach der Grundannahme eines statistischen Experiments existiert in der vorgegebenen Kollektion \( W \) der möglichen Stichprobenverteilungen genau eine Verteilung \( p[w,.] \), die wahr ist, d.h. die Stichprobenwerte sind unabhängige Replikanten eben dieser Verteilung.

Für die Zug um Zug anfallende Stichprobenrealisation setzt man aus Gründen zweckmäßiger Schreibweise das Symbol „\( k \)“ anstelle von „\( x \)“. Zunächst wird ein gemäß \( p[w,.] \) verteilter Stichprobenwert \( k \in \mathbb{H} \) bestimmt, durch \( k \) in der folgenden Matrix \( p[.,.] \)

\[
\begin{array}{cccccc}
\text{Stichproben} & \text{Parameter} & 1 & 2 & \ldots & k & \ldots & m \\
q[1] & 1 & p[1,1] & p[1,2] & \ldots & p[1,k] & \ldots & p[1,m] \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
q[r] & r & p[r,1] & p[r,2] & \ldots & p[r,k] & \ldots & p[r,m] \\
\end{array}
\]

\[
\Gamma
\]

eine Spalte \( p[., k] \), als Kandidat für die Posteriori-Verteilung \( h[., k] \) (an der Stelle \( k \)) festgelegt. Zur Bestimmung von \( h[., k] \) wird jedes Element \( p[i, k] \) des Spaltenvektors \( p[., k] \) mit dem entsprechenden Element \( q[i] \) des Vektors \( q \) (Priori-Verteilung) multipliziert. Normalisiert man den so entstehenden Spaltenvektor derart, daß dessen Spaltensumme gleich 1 ist, so ist mit dem so bestimmten Vektor

\[
h[., k] := \left( \begin{array}{c} h[1, k] \\ \vdots \\ h[r, k] \end{array} \right) \quad \text{(4.17)}
\]

die Posteriori-Verteilung an der Stelle \( k \) gefunden. Schließlich wird im Sinne des Iterationsprinzips der Vektor \( q \) (bisherige Priori-Verteilung) durch den die Posteriori-Verteilung an der Stelle \( k \) beschreibenden Vektor \( h[., k] \) überschrieben:

\[
q[.] := h[., k] \quad \text{(4.18)}
\]
h[.,k] übernimmt nun die Funktion der Priori-Verteilung.

Dieser Algorithmus ist, wie in Abschnitt 4.5 bereits erwähnt, in dem statistischen Analyseprogramm umgesetzt.
KAPITEL 4. STATISTIK

4.7 Untersuchung der Beispielprogramme

Für die weitere Datenanalyse wurden einige repräsentative Programme bzw. deren Quellcode verwendet. Die Quellcodes wurden teilweise aus größeren firmeninternen Programmen der Entwicklungsabteilung CC/ESM der Robert Bosch GmbH in Stuttgart (Bosch) oder aus frei verfügbaren Quellen im Internet gewonnen.


4.7.1 Beispielprogramme

Für die Datenanalyse wurden folgende Java-Programme verwendet, die sich auch auf der beiliegenden DVD befinden (ausgenommen die Programme der Robert Bosch GmbH):

<table>
<thead>
<tr>
<th>Programm</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUnit 3.8.2</td>
<td>Test-Framework für Java-Programme</td>
</tr>
<tr>
<td>JHotDraw 6.0b1</td>
<td>GUI framework for Graphics</td>
</tr>
<tr>
<td>J2SDK 1.4.2</td>
<td>Java Software Development Kit</td>
</tr>
<tr>
<td>Sinaxe2</td>
<td>Code Generator</td>
</tr>
<tr>
<td>Apache Ant 1.7.0</td>
<td>Java-based build tool</td>
</tr>
<tr>
<td>Apache James 2.3.1</td>
<td>Plattformunabhängiger E-Mail-Server</td>
</tr>
<tr>
<td>Apache Tomcat 6.0.10</td>
<td>Web-Server</td>
</tr>
<tr>
<td>CoBrake</td>
<td>Tool zur Bremsenauslegung (Bosch)</td>
</tr>
<tr>
<td>Frinika</td>
<td>Music workstation software</td>
</tr>
<tr>
<td>JCommon 1.0.12</td>
<td>Basis von JFreeChart/JFreeReport</td>
</tr>
<tr>
<td>JFreeChart 1.0.6</td>
<td>Grafik-Bibliothek</td>
</tr>
<tr>
<td>JFreeReport 0.8.7</td>
<td>Reporting library for embedded solutions</td>
</tr>
<tr>
<td>Orson 0.5.0</td>
<td>Collection of charting components</td>
</tr>
<tr>
<td>JToolChain</td>
<td>API zur Fernsteuerung von Simulink</td>
</tr>
<tr>
<td>SimServ</td>
<td>Simulationsdatenbank-Server (Bosch)</td>
</tr>
<tr>
<td>SWT Swing-3-2-0004</td>
<td>Widget-Bibliothek</td>
</tr>
<tr>
<td>TAP 1.0.0</td>
<td>Typ-Analyse-Plug-in</td>
</tr>
<tr>
<td>Java Chess 0.1.0</td>
<td>Schachprogramm</td>
</tr>
<tr>
<td>Kaffe 1.1.7</td>
<td>Clean room impl. of the Java VM</td>
</tr>
<tr>
<td>Marathon 1.0.4</td>
<td>GUI development tool</td>
</tr>
</tbody>
</table>

Tabelle 4.1: Repräsentative Beispielprogramme
4.7.2 Untersuchung

Die Programme werden dazu innerhalb der Eclipse-IDE geladen, auf Fehlerfreiheit bezüglich der Syntax geprüft und anschließend der Typ-Analyse unterzogen. Die Quellen von Bosch wurden firmenintern der Typ-Analyse unterzogen und nur die Ergebnisse hier weiterverwendet. Dabei entsteht bei jeder Typ-Analyse eine Tabelle der nachfolgenden beispielhaften Form:

<table>
<thead>
<tr>
<th>Anzahl der Deklarationen</th>
<th>Typ</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>6</td>
<td>java.util.List&lt;java.io.File&gt;</td>
</tr>
<tr>
<td>4</td>
<td>void</td>
</tr>
<tr>
<td>3</td>
<td>java.io.File</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>3</td>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>2</td>
<td>int[]</td>
</tr>
<tr>
<td>1</td>
<td>FileFinder</td>
</tr>
<tr>
<td>1</td>
<td>char[]</td>
</tr>
<tr>
<td>1</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>1</td>
<td>java.util.Stack&lt;java.io.File&gt;</td>
</tr>
<tr>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>0</td>
<td>java.util.ArrayList&lt;java.io.File&gt;</td>
</tr>
<tr>
<td>0</td>
<td>IFileFinder</td>
</tr>
<tr>
<td>0</td>
<td>StartFileFinder</td>
</tr>
<tr>
<td>0</td>
<td>StartFileFinder.LocalClass</td>
</tr>
</tbody>
</table>

Tabelle 4.2: Beispieltabelle eines kleinen Beispielprogramms


In der ersten Zeile steht der Datentyp mit dem häufigsten Vorkommen (Deklarationen), in der zweiten Zeile der Datentyp mit dem zweithäufigsten Vorkommen usw. Am Ende der Liste kann es noch zu Einträgen mit der Anzahl von 0 Deklarationen geben. Diese deuten auf Datentypen im Programm hin, die dort zwar definiert werden, aber nie als Deklarationselement verwendet werden. In Java-Programmen kann dies z.B. die Klasse mit dem Einsprungspunkt zum Hauptprogramm „main()“ sein. Diese Klassen haben keinen Einfluß auf die weitere Analyse, sollen aber bei der Quellcodeanalyse nicht unberücksichtigt bleiben. Evtl. können auch hier weitere Optimierungen des Programms erfolgen, indem solche Klassen einfach weggelassen werden können. Für die weitere Analyse wird pro zu analysierendem Programm eine Text-Datei erzeugt, die nur die Anzahl
KAPITEL 4. STATISTIK

der aufgetretenen Typen auflistet. Die Namen der einzelnen Typen sind hier nicht von Bedeutung und wurden in den erzeugten TXT-Dateien einfach weggelassen. Der Inhalt der TXT-Datei ist nach der Anzahl der Deklarationen in absteigender Reihenfolge sortiert. Diese Datei kann grafisch als Histogramm in folgender Form dargestellt werden:

Abbildung 4.2: Histogramm

Die Daten des Histogramms (TXT-Datei) dienen im Weiteren als Eingabedaten für das statistische Auswerte-Programm.
5 Ergebnisse

5.1 Verteilungen

Die vorliegenden Beispielprogramme aus der Tabelle 4.1 wurden mit dem statistischen Auswerteprogramm untersucht. In sämtlichen Fällen ergibt sich ein eindeutiges Ergebnis zugunsten einer der beiden bekannten Verteilungen. Das heißt, die Daten der untersuchten Beispielprogramme fallen eindeutig in eine Klasse der beiden bekannten Verteilungen. Als Beispiel sei hier die Abbildung 5.1 des Programms „JUnit 3.8.2“ gegeben.

Abbildung 5.1: Posteriori-Verteilung (JUnit 3.8.2)

Damit lassen sich die Typ-Häufigkeiten des Programms „JUnit 3.8.2“ eindeutig der Yule-Verteilung zuordnen, wie man an dem roten Peak erkennen kann.
In diesem Beispiel der Grafikbibliothek „JFreeChart 1.0.6“ hat sich die Zipf/Estoup-Verteilung durchgesetzt, wie man an dem magenta Peak erkennen kann.
Abbildung 5.3: Posteriori-Verteilung (TAP 1.0.0)

In diesem Beispiel von „TAP 1.0.0“ hat sich in der Tendenz die Yule-Verteilung durchgesetzt, wie man an dem größeren, ausgeprägteren der beiden Peaks erkennen kann.
Für die weiteren Beispielprogramme ergeben sich die in Tabelle 5.1 dargestellten Klassifizierungen. Ein „X“ bedeutet hier, daß sich ein eindeutiges Ergebnis in Form eines Peaks bei der Posteriori-Verteilung ergibt. Damit ist das untersuchte Programm eindeutig der jeweiligen Verteilung zuzuordnen.

<table>
<thead>
<tr>
<th>Programm</th>
<th>Yule</th>
<th>Zipf/Estoup</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUnit 3.8.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JHotDraw 6.0b1</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>J2SDK 1.4.2</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Sinaxe2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Apache Ant 1.7.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache James 2.3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Tomcat 6.0.10</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CoBrake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frinika</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JCommon 1.0.12</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JFreeChart 1.0.6</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JFreeReport 0.8.7</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Orson 0.5.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JToolChain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimServ</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SWT Swing-3-2-0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAP 1.0.0</td>
<td>X</td>
<td>x</td>
</tr>
<tr>
<td>Java Chess 0.1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kaffe 1.1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marathon 1.0.4</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Tabelle 5.1: Klassifizierung der Verteilungen

5.2 Abweichungen


<table>
<thead>
<tr>
<th>Programm</th>
<th>Abweichung Yule</th>
<th>Abweichung Zipf/Estoup</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUnit 3.8.2</td>
<td>0.18635056631801772</td>
<td>0.2139252939932779</td>
</tr>
<tr>
<td>JHotDraw 6.0b1</td>
<td>0.22475110504797988</td>
<td>0.1901227629943218</td>
</tr>
<tr>
<td>J2SDK 1.4.2</td>
<td>1.185550352648667</td>
<td>0.784984972715766</td>
</tr>
<tr>
<td>Sinaxe2</td>
<td>0.36653506158059607</td>
<td>0.3774778781650896</td>
</tr>
<tr>
<td>Apache Ant 1.7.0</td>
<td>0.5533452634033342</td>
<td>0.6244172885230729</td>
</tr>
<tr>
<td>Apache James 2.3.1</td>
<td>0.15808007731559603</td>
<td>0.1903106975143099</td>
</tr>
<tr>
<td>Apache Tomcat 6.0.10</td>
<td>1.265925224811419</td>
<td>0.746983583426389</td>
</tr>
<tr>
<td>CoBrake</td>
<td>1.3943195295211803</td>
<td>0.6590067230093477</td>
</tr>
<tr>
<td>Frinika</td>
<td>0.4206676390460549</td>
<td>0.41466480798539784</td>
</tr>
<tr>
<td>JCommon 1.0.12</td>
<td>0.39069265595919156</td>
<td>0.40685801818957307</td>
</tr>
<tr>
<td>JFreeChart 1.0.6</td>
<td>0.441964038985785</td>
<td>0.4111447304536075</td>
</tr>
<tr>
<td>JFreeReport 0.8.7</td>
<td>0.3548829457760193</td>
<td>0.37578013951639283</td>
</tr>
<tr>
<td>Orson 0.5.0</td>
<td>0.47358322548335999</td>
<td>0.49268899326668157</td>
</tr>
<tr>
<td>JToolChain</td>
<td>0.38237538640069363</td>
<td>0.39626730822490724</td>
</tr>
<tr>
<td>SimServ</td>
<td>0.26056854371873894</td>
<td>0.2795288613168352</td>
</tr>
<tr>
<td>SWTSwing-3-2-0004</td>
<td>0.5170534440285449</td>
<td>0.50397098730439337</td>
</tr>
<tr>
<td>TAP 1.0.0</td>
<td>0.2852650188290128</td>
<td>0.31588253726842336</td>
</tr>
<tr>
<td>Java Chess 0.1.0</td>
<td>0.24328093517665073</td>
<td>0.2671864112061283</td>
</tr>
<tr>
<td>Kaffe 1.1.7</td>
<td>1.18427107356466</td>
<td>0.7899465773548795</td>
</tr>
<tr>
<td>Marathon 1.0.4</td>
<td>0.19592428024892475</td>
<td>0.2091552766634423</td>
</tr>
</tbody>
</table>

Tabelle 5.2: Abweichungen der Beispielprogramme

5.3 Fazit

Wie man aus den beiden Tabellen 5.1 und 5.2 sehen kann, ergibt sich keine eindeutige Mehrheit für eine der beiden Verteilungen. Man kann also feststellen, daß sich für die hier untersuchten Programme kein eindeutiges Ergebnis zugunsten einer der beiden Verteilungen ergeben hat. Um herauszufinden, welche Faktoren die Entscheidung zugunsten der einen oder der anderen Verteilung beeinflussen, sind weitere Arbeiten notwendig, die
wohl mit Unterstützung der Fachbereiche Mathematik bzw. Linguistik durchzuführen wären.
6 Typ-Analyse-Plug-in (TAP)

6.1 Anforderungen


Der Typ „java.lang.String“ sei insgesamt z.B. sechs mal in einem Programm enthalten, was die globale Typ-Analyse ergeben hat. Das Programm selbst besteht wiederum aus den drei Typen, hier z.B. aus den Klassen A, B und C. Der Typ „java.lang.String“ ist nun folgendermaßen in den drei Typen/Klassen verteilt:

<table>
<thead>
<tr>
<th>Typ</th>
<th>Anzahl der Vorkommen</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

Tabelle 6.1: Beispiel von Typ-Vorkommen

Für die beschriebenen Anforderungen werden hierfür die folgenden Ansichten (Views) innerhalb des Plug-ins realisiert:

- **Type Diagram View** - stellt die NOD- und NOI-Metrik grafisch dar
- **Type List View** - stellt die NOD- und NOI-Metrik tabellarisch dar
- **Detail Diagram View** - stellt das Vorkommen eines bestimmten Typs in einem anderen Typ grafisch dar
- **Detail List View** - stellt die Details der Vorkommen eines bestimmten Typs in einem anderen Typ tabellarisch dar
- **File List View** - stellt die untersuchten Java-Quellen tabellarisch dar

6.1.1 Type Diagram View für die grafische Ausgabe der NOD- und NOI-Metrik


Abbildung 6.1: Histogramm der Häufigkeit der Vorkommen von Typen innerhalb von JUnit 3.8.2

Durch das Umschalten, mittels „I“ in der Menüzeile dieser Ansicht, kann hier auf die Darstellung der Metrik NOI gewechselt werden. Diese gibt die Häufigkeit der Vorkommen von Instanziierungen von Typen an.

<table>
<thead>
<tr>
<th>Typ</th>
<th>NOD</th>
<th>NOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>617</td>
<td>0</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>264</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>187</td>
<td>0</td>
</tr>
<tr>
<td>junit.framework.Test</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>boolean</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>junit.framework.TestResult</td>
<td>69</td>
<td>22</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>51</td>
<td>7</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>junit.samples.money.IMoney</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>35</td>
<td>21</td>
</tr>
<tr>
<td>java.util Enumeration</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>junit.samples.money.Money</td>
<td>25</td>
<td>42</td>
</tr>
<tr>
<td>java.lang.String[]</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>long</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>junit.framework.TestSuite</td>
<td>23</td>
<td>32</td>
</tr>
<tr>
<td>junit.framework_TestCase</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>java.lang.Throwables</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>java.awt.Component</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>java.awt.GridBagConstraints</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>java.awt.event.ActionEvent</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>javax.swing.Icon</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>double</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>javax.swing.JButton</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>junit.framework.TestFailure</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>junit.samples.money.MoneyBag</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>javax.swing.JLabel</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

Tabelle 6.2: Tabelle der Häufigkeit der Vorkommen von Typen innerhalb von JUnit 3.8.2
6.1.2 Type List View für die tabellarische Ausgabe der NOD- und NOI-Metrik

Die Abbildung 6.2 stellt das oben gezeigte Histogramm der NOD-Metrik 6.1 in Form einer Tabelle dar. Die dargestellten Daten werden außerdem jeweils in einer Datei abgespeichert, um anschließend weiterverarbeitet werden zu können.

Abbildung 6.2: Listen-Ansicht der Deklarationen/Instanziierungen
6.1.3 Detail Diagram View für die grafische Ausgabe der Abhängigkeiten eines bestimmten Typs

Durch Auswahl einer Zeile, z.B. „junit.framework.TestResult“ in der vorherigen Ansicht, kann die Häufigkeit des Vorkommens dieses Typs in Deklarationselementen, die in anderen Typen vorkommen, analysiert werden.

Abbildung 6.3: Typ-Abhängigkeiten des Typs „junit.framework.TestResult“

Durch das Umschalten, mittels „I“ in der Menüzeile dieser Ansicht, kann auch hier auf die Vorkommen von Instanziierungen des selektierten Typs in anderen Typen dargestellt werden.
6.1.4 Detail List View für die tabellarische Ausgabe der Abhängigkeiten eines bestimmten Typs

Die detaillierten Daten dieser Analyse können in der folgenden Ansicht 6.4 in tabellarischer Form betrachtet werden. Mit Hilfe dieser Daten könnten nun ebenfalls Optimierungen eines bestimmten Typs vorgenommen werden. Es können zum einen häufig benutzte Typen bezüglich Laufzeit oder Speicherbedarf optimiert werden oder zum anderen können selten oder nicht benutzte Typen eliminiert bzw. wegoptimiert werden, z.B. durch Typen aus einer Standardbibliothek ersetzt werden. Außerdem kann man Abhängigkeiten zwischen einzelnen Typen erkennen und hat so frühzeitig die Möglichkeit abzuschätzen, welchen Einfluß eine Änderung eines Typs auf die von ihm abhängigen Typen mit sich bringt. Es kann damit auch der zeitliche Änderungsaufwand bei größeren Umstellungen in einem Programm abgeschätzt werden.

Abbildung 6.4: Detail-Ansicht der Vorkommen eines bestimmten Typs in anderen Typen
6.2 Praktische Umsetzung

Von dem aktuell selektierten Eclipse-Projekt müssen zunächst alle Java-Quellen, die sogenannten „Compilation-Units“ (CUs), identifiziert werden. Anschließend wird von jeder Compilation-Unit der zugehörige AST erzeugt. Dies kann durch das folgende beispielhafte Codefragment 6.1 realisiert werden:

```java
public class ASTHandler {
    // Set source file
    File file = new File("source.java");
    ...
    // Allocate buffer for file content
    char[] content = new char[file.length()];
    // Read in file
    file.read(content);
    ...
    // Create new ASTParser
    ASTParser astParser = ASTParser.newParser(AST.JLS3);
    // Set kind of source
    astParser.setKind(ASTParser.K_COMPILATION_UNIT);
    // Create fully qualified type names (e.g. "java.lang.String")
    astParser.setResolveBindings(true);
    // Set source contents
    astParser.setSource(content);
    // Create the AST
    CompilationUnit cu = (CompilationUnit)astParser.createAST(null);
    // Create type declaration visitor
    TypeDeclarationVisitor tdVisitor = new TypeDeclarationVisitor();
    // Set visitor
    cu.accept(tdVisitor);
    ...
}
```

Listing 6.1: AST-Erzeugung und Parsen einer Java-Quelle

Hier wird beispielhaft gezeigt, wie für die Quelle „source.java“ der AST erzeugt und anschließend der eigenen Besucherklasse zugeführt wird. Die Besucherklasse ist dabei eine durch den Anwender zur Verfügung gestellte Klasse, welche das bekannte „Visitor“-Entwurfsmuster realisiert. Durch den Aufruf der Funktion „createAST“ in Zeile 19 wird der AST der Java-Quelle erzeugt. Mit dem Aufruf der Funktion „accept()“ in Zeile 23 der Compilation-Unit, mit dem „TypeDeclarationVisitor“ als Argument, wird die „Analyse“ der Quelle angestoßen, um an die Informationen aus dem AST zu gelangen, an denen das eigene Programm interessiert ist. Die Ergebnisse dieses „Besuchervorgangs“ können dann zunächst in Tabellen gehalten werden, um nach erfolgreichem Abschluss weiter verarbeitet zu werden.
KAPITEL 6. TYP-ANALYSE-PLUG-IN (TAP)

Der Hauptaufwand steckt dabei in der Klasse „TypeDeclarationVisitor“, welche im Rahmen dieser Arbeit entstanden ist und die eigentliche Auswertung der Typen vornimmt. Dazu wird, wie oben bereits erwähnt, das „Visitor“-Entwurfsmuster verwendet, um den AST rekursiv zu durchlaufen. Die zu implementierende Besucherklasse muss nun alle AST-Knoten „abfangen“, an denen sie interessiert ist. Dazu müssen die Besucherfunktionen, sogenannte „Visitor“ implementiert werden. Die folgende Tabelle 6.3 zeigt die zugehörigen Knotentypen, für die Visitor-Funktionen zu realisieren sind:

<table>
<thead>
<tr>
<th>AST-Knotentyp</th>
<th>Verwendung</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableDeclarationFragment</td>
<td>Variablen-Deklarationen</td>
</tr>
<tr>
<td>ArrayCreation</td>
<td>Array-Deklarationen</td>
</tr>
<tr>
<td>MethodDeclaration</td>
<td>Rückgabetyp und Parametertyp(en) von Methoden und Konstrukturen</td>
</tr>
<tr>
<td>EnhancedForStatement</td>
<td>Deklarationen in „for“-Schleifen</td>
</tr>
<tr>
<td>TypeDeclaration</td>
<td>Klassen- bzw. Interface-Deklarationen sowie Deklarationen von Aufzählungstypen</td>
</tr>
<tr>
<td>WildcardType</td>
<td>Variable Anzahl von Parametern (z.B. String ...)</td>
</tr>
</tbody>
</table>

Tabelle 6.3: AST-Knoten die Typ-Deklarationen repräsentieren

Das folgende Listing 6.2 zeigt ein Codebeispiel für die relevanten Deklarationen anhand typischer Fälle.

```
public class publicType {   // TypeDeclaration
    private Stack<File> dirs;   // VariableDeclarationFragment
    private int[] array;         // ArrayCreation

    public boolean outputAllFiles(int num) {   // MethodDeclaration
        for (File file: dirs.pop().listFiles()) {   // Enh.ForStatement
            ...
        }
    }

    private class privateType {   // TypeDeclaration
        // Konstruktur
        private privateType(String... s) {   // WildcardType
            "String..." wird als ein Typ "java.lang.String[]" gezählt!
            ...
        }
        ...
        ...
    }

    private ...
    ...
    ...
}
```

Listing 6.2: Deklarationsbeispiele und deren Zuordnung zu obigen AST-Knotentypen.
Dieses Programmfragment hat somit als Ergebnis:

<table>
<thead>
<tr>
<th>Typ</th>
<th>NOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>int[]</td>
<td>1</td>
</tr>
<tr>
<td>java.io.File</td>
<td>1</td>
</tr>
<tr>
<td>java.lang.String[]</td>
<td>1</td>
</tr>
<tr>
<td>java.util.Stack&lt;java.io.File&gt;</td>
<td>1</td>
</tr>
<tr>
<td>void</td>
<td>1</td>
</tr>
<tr>
<td>publicType</td>
<td>0</td>
</tr>
<tr>
<td>publicType.privateType</td>
<td>0</td>
</tr>
</tbody>
</table>

Tabelle 6.4: NOD des Programmfragments

6.3 Architektur


6.4 Material und Methoden

Für die Erstellung des Plug-ins und des Analyse-Tools wurden die folgenden Hilfsmittel verwendet:

- Eclipse-IDE Version 3.3.0 (Europa)
- JFreeChart-Bibliothek Version 1.0.4
- JCommon-Bibliothek Version 1.0.8
- Launch4j Cross-Plattform für Java-Applikationswrapper Version 3.0.0-pre2

6.5 Eclipse-IDE

6.6 JFreeChart-Bibliothek


6.7 JCommon-Bibliothek

Diese Bibliothek stellt die Grundfunktionalität für die oben genannte Grafikbibliothek zur Verfügung. Weiterhin ist sie auch Grundlage für das Erstellen von Reports (JFree-Report).

6.8 Launch4j-Applikationswrapper

7 Schlußbetrachtungen

7.1 Zusammenfassung

Ziel dieser Arbeit war die Entwicklung eines Plug-ins, welches ein bestehendes Java-Programm auf die Häufigkeit der Verwendung der im Programm vorkommenden Typen hin untersucht. Mit dieser Analyse kann man sich einen Überblick über die verwendeten Typen verschaffen und ggf. Optimierungen vornehmen.


7.2 Einsatzmöglichkeiten von TAP


7.3 Ausblick

Neben der erfolgten statistischen Untersuchung bezüglich der Verteilung von Typen könnten noch detailliertere Fragen gestellt werden, die Raum für weiterführende Arbeiten bieten. Dazu gehört zum Beispiel die Frage, wie weit sich die „Grammatik“ von Java-Programmen an die Grammatik von „normalen“ Texten anlehnt bzw. damit übereinstimmt. Für den Vergleich der beiden Grammatiken könnte die folgende Zuordnung zugrunde gelegt werden:

7.4 Fazit

Abbildungsverzeichnis

3.1 AST-Workflow ......................................................... 7
3.2 AST-View .......................................................... 9
4.1 Statistisches Analyse-Programm .................................. 18
4.2 Histogramm .......................................................... 24
5.1 Posteriori-Verteilung (JUnit 3.8.2) .............................. 25
5.2 Posteriori-Verteilung (JFreeChart 1.0.6) ....................... 26
5.3 Posteriori-Verteilung (TAP 1.0.0) .............................. 27
6.1 Histogramm der Häufigkeit der Vorkommen von Typen innerhalb von JUnit 3.8.2 .............................................. 33
6.2 Listen-Ansicht der Deklarationen/Instanzierungen ........... 35
6.3 Typ-Abhängigkeiten des Typs „junit.framework.TestResult“ .............................................. 36
6.4 Detail-Ansicht der Vorkommen eines bestimmten Typs in anderen Typen .............................................. 37
A.1 Histogramm der Typ-Instanzierungen ............................ 51
A.2 Plug-in-Detail ......................................................... 52
A.3 Datei-Ansicht des ausgewählten Programms .................. 53
A.4 Selektion des Exports .............................................. 55
A.5 Ablage der JAR-Datei .............................................. 56
A.6 Optionen des Exports .............................................. 57
A.7 Start der Typ-Analyse .............................................. 59
A.8 JAR-Export .......................................................... 70
A.9 Ressourcen und JAR-Dateiname .................................. 71
A.10 JAR-Packet-Optionen ............................................. 72
A.11 JAR-Manifest-Spezifikation ...................................... 73
A.12 Grundeinstellungen .............................................. 74
A.13 Classpath .......................................................... 75
A.14 Minimum JRE ...................................................... 76
A.15 Start-Fehlermeldung .............................................. 77
Tabellenverzeichnis

3.1 Auswertung des Beispielcodes ........................................... 6
4.1 Repräsentative Beispielprogramme ...................................... 22
4.2 Beispieltable eine kleinen Beispielprogramms ....................... 23
5.1 Klassifizierung der Verteilungen ........................................ 28
5.2 Abweichungen der Beispielprogramme ................................ 29
6.1 Beispiel von Typ-Vorkommen .............................................. 31
6.2 Tabelle der Häufigkeit der Vorkommen von Typen innerhalb von JUnit
   3.8.2 .............................................................................. 34
6.3 AST-Knoten die Typ-Deklarationen repräsentieren .................... 39
6.4 NOD des Programmfragments .............................................. 40
7.1 Gegenüberstellung von Grammatiken .................................... 43
7.2 Inhalt der beiliegenden DVD .............................................. 48
A.1 AST-Knoten, der Typ-Instanzierungen repräsentiert .................. 50
A.2 Beispielprogramm von Typ-Instanzierungen ............................. 50
A.3 Pakete und Klassen des Typ-Analyse-Plug-ins ......................... 54
Listings

3.1 Algorithmus zur Berechnung der Metrik NOD .......................... 5
3.2 Beispielcode mit relevanten Deklarationselementen ................. 6
3.3 Beispiel für Besucherfunktionen zum Auswerten von AST-Daten .. 10

6.1 AST-Erzeugung und Parsen einer Java-Quelle ...................... 38
6.2 Deklarationsbeispiele und deren Zuordnung zu obigen AST-Knotentypen 39

A.1 Algorithmus zur Berechnung der Metrik NOI ....................... 49
A.2 Beispiel für Typ-Instanziierungen .................................. 50
A.3 Listing des statistischen Auswerteprogramms ..................... 60
Literaturverzeichnis


### Inhalt der beiliegenden DVD

<table>
<thead>
<tr>
<th>Verzeichnis</th>
<th>Inhalt</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdf</td>
<td>Master-Arbeit als PDF</td>
</tr>
<tr>
<td>latex</td>
<td>LaTeX-Quellen der Master-Arbeit</td>
</tr>
<tr>
<td>plugin</td>
<td>Typ-Analyse-Plug-in (TAP)</td>
</tr>
<tr>
<td>project/zip</td>
<td>Beispielprogramme</td>
</tr>
<tr>
<td>project/txt</td>
<td>Typ-Analyse-Ergebnisse</td>
</tr>
<tr>
<td>statistics</td>
<td>Statistisches Auswerteprogramm</td>
</tr>
<tr>
<td>eclipse</td>
<td>Eclipse-SDK 3.3 (Europa)</td>
</tr>
<tr>
<td>jfree</td>
<td>jcommon 1.0.8</td>
</tr>
<tr>
<td>jfree</td>
<td>jfreechart 1.0.4</td>
</tr>
<tr>
<td>launch4j</td>
<td>launch4j 3.0.0-pre2</td>
</tr>
</tbody>
</table>

Tabelle 7.2: Inhalt der beiliegenden DVD
A Anhang

A.1 Metrik - Number of Instantiations (NOI)

Diese Metrik wird für die aktuelle Arbeit nicht benötigt, kann aber durch einen relativ geringen Zusatzaufwand realisiert werden, da das Grundgerüst der Software dies erlaubt. Die Metrik mißt die Instanziierungen von Typen in Java-Programmen und kann folgendermaßen angegeben werden:

\[
NOI(A) = | \{ \text{new } A \} | \quad (A.1)
\]

wobei gilt: A = Typ

Für das Analyse-Plug-in ergibt sich der folgende Algorithmus in Form von Pseudocode:

```pseudocode
for each instantiationElement of actualProject do {
    Type := getTypeOfInstantiationElement;
    CountingArray[Type]++;
}
```

Listing A.1: Algorithmus zur Berechnung der Metrik NOI
A.2 Ermittlung der Typ-Instanziierungen


<table>
<thead>
<tr>
<th>AST-Knotentyp</th>
<th>Verwendung</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassInstanceCreation</td>
<td>Typ-Instanziierung</td>
</tr>
</tbody>
</table>

Tabelle A.1: AST-Knoten, der Typ-Instanziierungen repräsentiert

Das Listing A.2 zeigt ein Codebeispiel für den typischen Anwendungsfall von Typ-Instanziierungen.

```
public class publicType {
    ...
    public void publicType() {
        Integer i = new Integer(15);
        String s = new String("Test");
        List<String> l = new ArrayList<String>();
        ...
    }
}
```

Listing A.2: Beispiel für Typ-Instanziierungen


Im Beispiel aus A.2 werden 3 Typen jeweils einmal instanziert, sodaß das Analyseergebnis wie folgt aussieht:

<table>
<thead>
<tr>
<th>Typ</th>
<th>Anzahl der Typ-Instanziierungen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>String</td>
<td>1</td>
</tr>
<tr>
<td>List&lt;String&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

Tabelle A.2: Beispielprogramm von Typ-Instanziierungen

Abbildung A.1: Histogramm der Typ-Instanziierungen
A.3 Typ-Analyse-Plug-in (TAP)

Die folgenden Unterkapitel beschreiben die Installation, Anwendung, Erstellung und den Start des Typ-Analyse-Plug-ins.

A.3.1 Installation des Plug-ins


Abbildung A.2: Plug-in-Detail

Details zu aktuell installierten Plug-ins kann man durch Aktivierung des Menübefehls „Help > About Eclipse SDK“ und anschließender Selektion der Schaltfläche „Plug-in Details“ zur Anzeige bringen.

A.3.2 Anwendung des Typ-Analyse-Plug-ins

A.3.3 File List View für die Ausgabe aller analysierten Java-Quellen


Abbildung A.3: Datei-Ansicht des ausgewählten Programms

Die dargestellten Dateien wurden für die Analyse herangezogen, indem jeweils ihr zugehöriger AST erzeugt wurde und in diesem nach den relevanten Deklarationslementen gesucht wurde, die für die Bestimmung der Häufigkeit der Verwendung von Typen herangezogen wurden, siehe auch Kapitel 3.2. Die Informationen werden dabei in Listen gespeichert. Zum einen wird für jeden Typ die Anzahl seiner Verwendung aufsummiert und zum anderen wird festgehalten, in welchem anderen Typ der gefundene Typ vorkommt. Mit diesen Informationen sind dann die Ausgaben in den einzelnen Ansichten möglich, inklusive der Umschaltung zwischen den beiden Metriken NOD und NOI.
### A.3.4 Paketstruktur des Plug-ins

Die folgende Tabelle A.3 zeigt die Paketstruktur des Eclipse-Projekts.

<table>
<thead>
<tr>
<th>Pakete</th>
<th>Klassen</th>
</tr>
</thead>
<tbody>
<tr>
<td>de.feu.tap</td>
<td>Activator</td>
</tr>
<tr>
<td></td>
<td>DetailType</td>
</tr>
<tr>
<td></td>
<td>FileFinder</td>
</tr>
<tr>
<td></td>
<td>SearchResult</td>
</tr>
<tr>
<td></td>
<td>TypeDeclarationVisitor</td>
</tr>
<tr>
<td></td>
<td>TypeListElement</td>
</tr>
<tr>
<td></td>
<td>TypeNumber</td>
</tr>
<tr>
<td></td>
<td>XYBarToolTipGenerator</td>
</tr>
<tr>
<td>de.feu.tap.popup.actions</td>
<td>NewAction</td>
</tr>
<tr>
<td>de.feu.tap.views</td>
<td>DetailDiagramView</td>
</tr>
<tr>
<td></td>
<td>DetailListView</td>
</tr>
<tr>
<td></td>
<td>FileListView</td>
</tr>
<tr>
<td></td>
<td>TypeDiagramView</td>
</tr>
<tr>
<td></td>
<td>TypeListView</td>
</tr>
</tbody>
</table>

Tabelle A.3: Pakete und Klassen des Typ-Analyse-Plug-ins
A.3.5 Erstellen der JAR-Datei


Abbildung A.4: Selektion des Exports
In der nächsten Ansicht wird der Ort für die zu erzeugende JAR-Datei spezifiziert. Dies kann entweder ein Verzeichnis (Option „Directory“) sein, in dem dann ein Verzeichnis „plugins“ erzeugt wird, welches die JAR-Datei enthält. Eine andere Möglichkeit ist die Angabe einer Archivdatei (Option „Archive file“), welche dann anschließend das JAR-Archiv enthält.

![Abbildung A.5: Ablage der JAR-Datei](image)

In beiden Fällen wird das für die weitere Arbeit benötigte JAR-Archiv „de.feu.tap_1.0.0.jar“ erzeugt.
Im letzten Schritt sollten noch die Einstellungen auf der Registerkarte „Options“ überprüft werden und sichergestellt werden, daß für das Plug-in ein individuelles JAR-Archiv erzeugt wird.

Abbildung A.6: Optionen des Exports

A.3.6 Hinzufügen der JFreeChart-Bibliotheken

Um die JFreeChart-Bibliotheken hinzuzufügen zu können, muß das JAR-Archiv mit WinZip geöffnet werden. Anschließend müssen die entpackten JFreeChart-Bibliotheken manuell dem Archiv hinzugefügt werden. Hierbei handelt es sich um die beiden JARs „jcommon-1.0.8.jar“ und „jfreechart-1.0.4.jar“ Dies geschieht am einfachsten mittels Drag&Drop des entpackten Unterverzeichnisses „org“ der beiden Bibliotheken in das
geöffnete WinZip. Beim Entpacken der beiden JARs entsteht ein gemeinsames Unter-
verzeichnis, in dem die kompletten Bibliotheken enthalten sind. Wichtig ist hierbei, daß
die bereits im JAR-Archiv des Plug-ins enthaltene Datei „MANIFEST.MF“, im Unter-
verzeichnis „meta-inf“, **nicht** mit der gleichnamigen Datei der JFreeChart-Bibliothek
überschrieben wird. Im Normalfall warnt WinZip standardmäßig vor dem Überschrei-
ben. Die gleichnamige Datei „MANIFEST.MF“ der Grafik-Bibliothek kann einfach ge-
löscht werden, da sie nicht benötigt wird. Dies vermeidet den angesprochenen Konflikt.

A.3.7 Ausgaben des Typ-Analyse-Plug-ins

Zur Ausgabe der Analyseergebnisse werden in der aktuellen Version 1.0.0 die folgenden
Dateiformate verwendet:

- TXT - enthält die sortierten Typ-Vorkommen ohne Namen im Text-Format
- XLS - enthält die sortierten Typ-Vorkommen inklusive Namen im Excel-Format

Diese Dateien können für die Weiterverarbeitung in anderen Programmen dienen. Mit
der XLS-Datei besteht außerdem die Möglichkeit, die Histogramme in Excel darzustel-
len. Zusätzlich enthält sie auch die kumulierten und auf die Gesamtsumme aller Typ-
vorkommen normierten Summen der einzelnen Typ-Vorkommen, die für weitergehende
Darstellungen/Berechnungen dienen können.
A.3.8 Start der Typ-Analyse


Abbildung A.7: Start der Typ-Analyse
A.4 Statistisches Auswerteprogramm

A.4.1 Listing des Programms

```java
/* Java-Programm zur Bestimmung der Posteriori-Wahrscheinlichkeit und der Dichtefunktion */
package de.feu.bayeschart;

import java.awt.Color;
import java.awt.GridLayout;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JOptionPane;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.chart.title.TextTitle;
import org.jfree.chart.title.Title;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;

/* Applikation zur Anzeige von zwei Liniencharts basierend auf der Klasse XYDataset */
public class BayesChart extends ApplicationFrame {

    private static int anz = 0;
    private static final int Npar = 100;
    private static final int num = 6;
    private static String dataFile = null;
    static double[][] K = null;
    static double[] prior = new double[2 * Npar + 1];
    static double[] post = new double[2 * Npar + 1];
    static double[] rho = new double[2 * Npar + 1];
    static int[] sam = null;
    static double[] rh = null;
    static double[] cof = new double[num];
```
char ch;
int gammaoptYule = 0;
int gammaoptZipf = 0;
double diffYule = 0.0;
double diffZipf = 0.0;

private static final long serialVersionUID = 1L;
/* Erzeugt eine neue Applikation */
public BayesChart(final String title) {
  super(title + " - [" + dataFile.toString() + "]");
  getContentPane().setLayout(new GridLayout(1, 2));
  final XYDataset dataset1 = createDataset1();
  final XYDataset dataset2 = createDataset2();
  final JFreeChart chart1 = createChart1(dataset1);
  final JFreeChart chart2 = createChart2(dataset2);
  final ChartPanel chartPanel1 = new ChartPanel(chart1);
  final ChartPanel chartPanel2 = new ChartPanel(chart2);
  chartPanel1.setPreferredSize(new java.awt.Dimension(500, 350));
  chartPanel2.setPreferredSize(new java.awt.Dimension(500, 350));
  getContentPane().add(chartPanel1);
  getContentPane().add(chartPanel2);
}
/* Erzeugt ein neues Linienchart fuer die Dichtefunktionen
   beiden Verteilungen basierend auf einem XYDataset */
private JFreeChart createChart1(final XYDataset dataset) {
  final JFreeChart chart = ChartFactory.createXYLineChart("Yule-/Zipf-Estoup-Verteilung", "rho", "Wahrscheinlichkeit", dataset,
    PlotOrientation.VERTICAL, true, false, false);
  Title subTitle = new TextTitle("Posteriori");
  chart.addSubtitle(subTitle);
  // Einstellungen fuer das Chart vornehmen
  chart.setBackgroundPaint(Color.white);
final XYPlot plot = chart.getXYPlot();
plot.setBackgroundPaint(Color.lightGray);
plot.setDomainGridlinePaint(Color.white);
plot.setRangeGridlinePaint(Color.white);

final XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();
renderer.setSeriesShapesVisible(0, false);
renderer.setSeriesShapesVisible(1, false);
renderer.setSeriesPaint(0, Color.red);
renderer.setSeriesPaint(1, Color.magenta);
plot.setRenderer(renderer);

// Beide Achsen auf Integer-Anzeigewerte umstellen
final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());
final NumberAxis domainAxis = (NumberAxis) plot.getDomainAxis();
domainAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

return chart;

/* Ezeugt ein neues Liniенchart fuer die Anzeige der Posteriori-Verteilungen basierend auf einem XYDataset */
private JFreeChart createChart2(final XYDataset dataset) {
    // Neues Chart erzeugen
    final JFreeChart chart = ChartFactory.createXYLineChart(
        "Analysedaten - vs. Opt.-Verteilung", // chart title
        "Typen", // x axis label
        "Wahrscheinlichkeit", // y axis label
        dataset, // data
        PlotOrientation.VERTICAL, // vertical orient.
        true, // include legend
        false, // tooltips
        false); // urls

    Title subTitle = new TextTitle("Delta: [Yule]=" + diffYule + "
                                 [Zipf-Estoup]=" + diffZipf);
    chart.addSubtitle(subTitle);

    // Einstellungen fuer das Chart vornehmen
    chart.setBackgroundPaint(Color.white);
final XYPlot plot = chart.getXYPlot();
plot.setBackgroundPaint(Color.lightGray);
plot.setDomainGridlinePaint(Color.white);
plot.setRangeGridlinePaint(Color.white);

final XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();
renderer.setSeriesShapesVisible(0, false);
renderer.setSeriesShapesVisible(1, false);
renderer.setSeriesShapesVisible(2, false);
renderer.setSeriesPaint(0, Color.blue);
renderer.setSeriesPaint(1, Color.red);
renderer.setSeriesPaint(2, Color.magenta);
plot.setRenderer(renderer);

// Beide Achsen auf Integer-Anzeigewerte umstellen
final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());
final NumberAxis domainAxis = (NumberAxis) plot.getDomainAxis();
domainAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

return chart;
// Neue Charts erzeugen
final BayesChart demo = new BayesChart("Bayes Chart");
demo.pack();
RefineryUtilities.centerFrameOnScreen(demo);
demo.setVisible(true);
}

/* Berechnung der Gamma-Funktion. Es wird der Logarithmus der Gamma-Funktion berechnet */
public static double LNGamma(double xx) {
    double x, y, ser, tmp;
    cof[0] = 76.18009172947146;
    cof[1] = -86.50532032941677;
    cof[3] = -1.231739572450155;
    cof[4] = 0.1208650973866179E-2;
    cof[5] = -0.5395239384953E-5;
    x = xx;
    y = x;
    tmp = x + 5.5;
    tmp = tmp - (x + 0.5) * Math.log(tmp);
    ser = 1.0000000000190015;
    for (int j = 0; j < num; j++) {
        y = y + 1.0;
        ser = ser + cof[j] / y;
    }
    return -tmp + Math.log(2.506628274631005 * ser / x);
}

/* Berechnung der Beta-Funktion mit Hilfe der Gamma-Funktion.
Man beachte, dass der Logarithmus der Gamma-Funktion verwendet wird */
public static double Beta(double a, double b) {
    double expo;
    expo = LNGamma(a) + LNGamma(b) - LNGamma(a + b);
    return Math.exp(expo);
}

/* Einlesen der Daten aus der TXT-Datei in das Feld sam */
private static void readFile(String dataFile) {
    // Prüfen des Dateinamens der TXT-Datei
    if (dataFile != null) {
        // Erzeuge buffered reader

BufferedReader br = null;
try {
    // Oeffne buffered reader
    br = new BufferedReader(new FileReader(dataFile));
} catch (IOException e) {
    e.printStackTrace();
}

try {
    int lineNumber = 0;
    // Berechne Zeilenanzahl der TXT-Datei
    while (br.readLine() != null) {
        lineNumber++;
    }
    // Puffer alloken entsprechen der Zeilenanzahl;
    sam = new int[lineNumber];
    K = new double[2 * Npar + 1][lineNumber];
    rh = new double[lineNumber];
    anz = lineNumber;
    // Oeffne neuen buffered reader
    br = new BufferedReader(new FileReader(dataFile));
    // Lege Anzahl der Zeilen
    for (int i = 0; i < lineNumber; i++) {
        sam[i] = Integer.valueOf(br.readLine()).intValue();
    }
} catch (IOException e) {
    e.printStackTrace();
}
try {
    // Schliesse buffered reader
    br.close();
} catch (IOException e) {
    e.printStackTrace();
}
} catch (IOException e) {
    e.printStackTrace();
}

/* Berechnung der Normalverteilung */
private static void calcData() {
    int i, j;
    double sum;
    sum = 0.0;
    for (i = 0; i < anz; i++) {
        rh[i] = sam[i];
    }
sum = sum + rh[i];
}

for (i = 0; i < anz; i++) {
    rh[i] = rh[i] / sum;
}

for (i = -Npar; i <= Npar; i++) {
prior[i + Npar] = 1.0 / (2.0 * Npar + 1.0);
}

for (i = -Npar; i <= Npar; i++) {
rho[i + Npar] = Math.abs(i / (double)Npar * 5.0);
}

/* Berechnung der Koeffizienten der Matrix K 
Vergleiche auch Rekursionsalgorithmus */

// ZIPF/ESTOUP
for (i = 0; i <= Npar; i++) {
    sum = 0.0;
    for (j = 0; j < anz; j++) {
        K[i + Npar][j] = 1.0 / Math.exp(rho[i + Npar] * Math.log(j + 1));
        sum = sum + K[i + Npar][j];
    }
    for (j = 0; j < anz; j++) {
        K[i + Npar][j] = K[i + Npar][j] / sum;
    }
}

// YULE
for (i = -Npar; i <= -1; i++) {
    sum = 0.0;
    for (j = 0; j < anz; j++) {
        K[i + Npar][j] = Beta(j + 1.0, rho[i + Npar] + 1.0);
        sum = sum + K[i + Npar][j];
    }
    for (j = 0; j < anz; j++) {
        K[i + Npar][j] = K[i + Npar][j] / sum;
    }
}

/* Anwendung der Summenformel des Satzes von Bayes 
Bestimmung der Posteriori-Wahrscheinlichkeit */
static void estimateData() {

int i, j, gamma;
double sum;

for (i = anz - 1; i >= 0; i--) {
    for (j = 1; j < sam[i]; j++) {
        sum = 0.0;
        for (gamma = -Npar; gamma <= Npar; gamma++) {
            post[gamma + Npar] = K[gamma + Npar][i] * prior[gamma + Npar];
            sum = sum + post[gamma + Npar];
        }
        for (gamma = -Npar; gamma <= Npar; gamma++) {
            post[gamma + Npar] = post[gamma + Npar] / sum;
        }
prior = post;
    }
}

/* Erzeugen der Datensetzte fuer die Dichtefunktionen */
private XYDataset createDataset1() {
    int gamma;
    double xk, yk;
    double yscale = Npar;

    final XYSeries series1 = new XYSeries("Yule");
    for (gamma = -Npar; gamma <= -1; gamma++) {
        xk = rho[gamma + Npar];
        yk = Math.round(post[gamma + Npar] * yscale);
        series1.add(xk, yk);
    }

    final XYSeries series2 = new XYSeries("Zipf-Estoup");
    for (gamma = 0; gamma <= Npar; gamma++) {
        xk = rho[gamma + Npar];
        yk = Math.round(post[gamma + Npar] * yscale);
        series2.add(xk, yk);
    }

    final XYSeriesCollection dataset = new XYSeriesCollection();
    dataset.addSeries(series1);
    dataset.addSeries(series2);
    return dataset;
}

/* Erzeugen der Datensetzte fuer die Posteriori-Wahrscheinlich-
keiten, basierend auf den Analysedaten. Bestimmen des jeweiligen Optimums für ρ */

```java
private XYDataset createDataset2() {
    int gamma;
    int i;
    double xk, yk;
    double max;
    double scale = Npar * 20;

    max = 0.0;
    for (gamma = -Npar; gamma <= -1; gamma++) {
        if (post[gamma + Npar] > max) {
            max = post[gamma + Npar];
            gammaoptYule = gamma + Npar;
        }
    }

    max = 0.0;
    for (gamma = 0; gamma <= Npar; gamma++) {
        if (post[gamma + Npar] > max) {
            max = post[gamma + Npar];
            gammaoptZipf = gamma + Npar;
        }
    }


    scale = anz * 30.0;
    final XYSeries series1 = new XYSeries("Analysedaten");
    for (i = 0; i < anz; i++) {
        xk = i + 1;
        yk = Math.round(scale * rh[i]);
        series1.add(xk, yk);
    }

    final XYSeries series2 = new XYSeries("Opt. Yule");
    for (i = 0; i < anz; i++) {
        xk = i + 1;
        yk = Math.round(scale * K[gammaoptYule][i]);
        series2.add(xk, yk);
    }

    final XYSeries series3 = new XYSeries("Opt. Zipf/Estoup");
    for (i = 0; i < anz; i++) {
        xk = i + 1;
        yk = Math.round(scale * K[gammaoptYule][i]);
        series3.add(xk, yk);
    }
}
```
\begin{verbatim}
xk = i + 1;
yk = Math.round(scale * K[gammaoptZipf][i]);
series3.add(xk, yk);
}

/* Berechnung des Deltas zwischen opt. Verteilung und den Analysedaten */
for (i = 0; i < anz; i++) {
    diffYule += Math.abs(rh[i] - K[gammaoptYule][i]);
    diffZipf += Math.abs(rh[i] - K[gammaoptZipf][i]);
}

System.out.println("Delta[Yule]=" + diffYule);
System.out.println("Delta[Zipf]=" + diffZipf);

final XYSeriesCollection dataset = new XYSeriesCollection();
dataset.addSeries(series1);
dataset.addSeries(series2);
dataset.addSeries(series3);
return dataset;
}
\end{verbatim}

Listing A.3: Listing des statistischen Auswerteprogramms
A.4.2 Erstellen des Eclipse-Projekts


A.4.3 Erstellen der JAR-Datei


Abbildung A.8: JAR-Export
Hier muss die Option „JAR file“ ausgewählt werden. Im nächsten Schritt werden die Komponenten selektiert, die im Export enthalten sein sollen. Hier nur die auf der linken Seite vorhandene Ressource „de.feu.bayeschart“ selektieren. Auf der rechten Seite alle Dateien deselectieren, da hier für die Datensicherung des Programms bereits Dokumente und fertige JAR- bzw. EXE-Dateien gesichert wurden. Weiter unten im Fenster muß das Ausgabeverzeichnis inklusive JAR-Dateiname spezifiziert werden.

Abbildung A.9: Ressourcen und JAR-Dateiname
Im nächsten Schritt sollten keine Veränderungen vorgenommen werden.

Abbildung A.10: JAR-Packet-Optionen
Im letzten Schritt muß das bereits existierende Manifest „MANIFEST.MF“ ausgewählt werden, welches den Klassenamen enthält, der die Methode „main“ beinhaltet, die dann beim Start der EXE ausgeführt wird.

Abbildung A.11: JAR-Manifest-Spezifikation

A.4.4 Erstellen der EXE-Datei


Abbildung A.12: Grundeinstellungen

Hier die zu erstellende EXE-Datei spezifizieren und das vorhandene JAR auswählen.
Bei der folgenden Ansicht müssen keine Änderungen vorgenommen werden. Der so- genannte „Classpath“ wird automatisch aus der Manifest-Datei „MANIFEST.MF“ des JARs bezogen.

Abbildung A.13: Classpath
Bei dieser Ansicht muß die Minimalversion des „Java Runtime Environments“ (JRE) spezifiziert werden. Für die verwendete Grafik-Bibliothek „JFreeChart 1.0.8“ inklusive „JCommon 1.0.4“ muss die minimale JRE-Version 1.3.0 betragen. Bei zukünftigen Weiterentwicklungen der JFreeChart-Bibliothek kann sich dies zu einer höheren Version hin ändern.


Abbildung A.14: Minimum JRE

Abbildung A.15: Start-Fehlermeldung
Erklärung

Hiermit erkläre ich, daß ich die vorliegende Abschlussarbeit selbstständig verfaßt, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.


____________________________
Gerd Eichhorn