“Statische und dynamische Analyse der Bedingungsüberdeckung objektorientierter Java-Programme”

Abschlussvortrag zur Studienarbeit
von Dominik Schindler
am 25.04.2006

Übersicht

1. **Einleitung**
    1. Aufgabe
    2. Motivation
2. **Grundlagen**
    1. Einfacher Bedingungsüberdeckungstest
    2. Bedingungs-/Entscheidungsüberdeckungstest
    3. Minimaler Mehrfach-Bedingungsüberdeckungstest
    4. Modifizierter Bedingungs-/Entscheidungsüberdeckungstest
    5. Mehrfach-Bedingungsüberdeckungstest
    6. Übertragung der „klassischen“ Bedingungsüberdeckungstests auf Java
    7. Definition der Überdeckungsmetriken
3. **ANTLR** („ANother Tool For Language Recognition“)
4. **Das Werkzeug**
    1. Statische Analyse
    2. Dynamische Analyse
    3. Messung der Überdeckung
5. **Ausblick**
6. **Demonstration**

1. Einleitung

Aufgabenstellung
und Motivation

1. Einleitung

- **Aufgabe:** Messung der Bedingungsüberdeckung objektorientierter Java-Programme.

- Dazu sollen bereits „…existierende Ansätze zur Übertragung der klassischen Bedingungsüberdeckungskriterien auf objekt-orientierte Software vergleichend bewertet und mit eigenen Konzepten geeignet ergänzt werden“.

- Außerdem „…ist für die Sprache Java ein Werkzeug zu entwickeln, welches die Programme bezüglich der bereits ermittelten Kriterien statisch analysiert, sowie, darauf aufbauend, die während der Testausführung erzielten Bedingungsüberdeckungen ermittelt“.

1. Einleitung

- **Definition „atomare (Teil-)Bedingung“:** Eine atomare (Teil-)Bedingung ist eine Bedingung, die nicht in weitere Unterbedingungen zerlegt werden kann.
  - D.h., dass atomare Bedingungen keine bool‘schen Operatoren wie AND, OR und NOT enthalten.
  - **Beispiel:** (i >= 0) && (i <= 10) ist keine atomare Bedingung, aber die Teil-bedingungen (i >= 0) und (i <= 10) sind atomar
- **Definition „unvollständige Evaluation“:** Unter unvollständiger Evaluation (engl. „Short circuit evaluation“) versteht man das vorzeitige Beenden der Auswertung eines Ausdrucks, wenn das Endergebnis bereits feststeht.
  - Im Gegensatz dazu wird bei vollständiger Evaluation immer der ganze Ausdruck ausgewertet.
  - **Beispiel:** (i >= 0) && (i <= 10): Sei (i < 0), dann ist die erste Bedingung bereits „false“ und die Auswertung wird vorzeitig beendet.

1. Einleitung

Motivation (1):

- **Problem:** Der Zweigüberdeckungstest ist nicht adäquat für den Test komplexer Bedingungen!
- **Einfache Bedingung: if (x > 5)** … kann als „ausreichend getestet“ [Ligge03] betrachtet werden, wenn gegen die beiden Wahrheitswerte „true“ und „false“ getestet wurde.
- **Komplexe Bedingung: if ((a < 10) && (b > 0) || (c > 10) && (d == 0))** ... kann beim Test gegen die beiden Wahrheitswerte „true“ und „false“ als nicht ausreichend betrachtet werden, da die Struktur nicht beachtet wird.
  - Z.B. erreichen folgende zwei Testfälle bereits vollständige Zweigüberdeckung:

    a = 5, b = 0, c = 7, d = 1 → Falsch und a = 5, b = 1, c = 11, d = 1 → Wahr
  - Trotzdem wurde die letzte Bedingung (d == 0) nicht ausreichend getestet, da im Falle von vollständiger Evaluation nur gegen „true“ getestet wurde.

1. Einleitung

Motivation (2):

- **Komplexe Entscheidung: ((a < 10) && (b > 0) || (c > 10) && (d == 0))**
  - **Testfall 1:** a = 5, b = 0, c = 7, d = 1 → Falsch
  - **Testfall 2:** und a = 5, b = 1, c = 11, d = 1 → Wahr
  - Im Falle von **unvollständiger Evaluation** – was gängige Praxis ist – wird die Bedingung **(d == 0)** überhaupt nicht getestet und dadurch ein eventuell vorhandener Fehler maskiert!
  - „Grundsätzlich gilt bei einer Evaluation von Entscheidungen von links nach rechts, dass Teilentscheidungen um so schlechter geprüft werden, je weiter rechts sie in einer zusammengesetzten Entscheidung stehen.“ [Ligge03]

2. Grundlagen

Die klassische Bedingungsüberdeckungstests
und deren Übertragung auf Java-Programme.

Einfacher Bedingungsüberdeckungstest

(„Simple condition coverage“)

- Fordert, dass **jede atomare** Bedingung mindestens einmal gegen Wahr und Falsch getestet wird.
- Im Falle von unvollständiger Auswertung schließt der einfache Bedingungsüberdeckungstest den Zweigüberdeckungstest (*„Branch coverage“*) mit ein.
- Bei vollständiger Auswertung ist das nicht unbedingt der Fall.

2.1. Grundlagen

Vollständige Auswertung der Bedingung ((a < 10) && (b > 0) || (c > 10) && (d == 0)):

	(a<10)	(b>0)	(c>10)	(d==0)	(a<10 && b>0)	(c>10 && d==0)	Gesamt
1	F	F	F	F	F	F	F
2	F	F	F	W	F	F	F
3	F	F	W	F	F	F	F
4	F	F	W	W	F	W	F
5	F	W	F	F	F	F	F
6	F	W	F	W	F	F	F
7	F	W	W	F	F	F	F
8	F	W	W	W	F	W	W
9	W	F	F	F	F	F	F
10	W	F	F	W	F	F	F
11	W	F	W	F	F	F	F
12	W	F	W	W	F	W	W
13	W	W	F	F	W	F	W
14	W	W	F	W	W	F	W
15	W	W	W	F	W	F	W
16	w	w	W	W	W	W	W

2.1. Grundlagen

Unvollständige Auswertung der Bedingung ((a < 10) && (b > 0) || (c > 10) && (d == 0)):

Klasse	Testfälle	(a<10)	(b>0)	(c>10)	(d==0)	(a<10 && b>0)	(c>10 && d ==0)	Gesamt
I	1, 2, 5, 6	F	-	F	-	F	F	F
II	3, 7	F	-	W	F	F	F	F
III	4, 8	F	-	W	W	F	W	W
IV	9, 10	W	F	F	F	F	F	F
V	11	W	F	W	F	F	F	F
VI	12	W	F	W	W	F	W	W
VII	13, 14	W	W	F	-	W	-	W
VIII	15	W	W	W	-	W	-	W
IX	16	W	W	W	-	W	-	W

Bedingungs-/Entscheidungsüberdeckungstest

(„Condition-/decition coverage“)

- Fordert zusätzlich zum einfachen Bedingungsüberdeckungstest, dass außerdem alle Zweige überdeckt werden (*“Branch coverage“* bzw. *„decition coverage“*).

- D.h., dieser Test beinhaltet den einfachen Bedingungsüberdeckungstest und den Zweigüberdeckungstest.

Minimaler Mehrfach-Bedingungsüberdeckungstest

(„Minimal multiple condition coverage“)

- Fordert, dass **jede** Bedingung, ob atomar oder nicht atomar, mindestens einmal gegen Wahr oder Falsch getestet wird.
- Dieser Test beinhaltet den Bedingungs-/Entscheidungsüberdeckungstest.
- D.h., die hierarchische/logische Struktur aller Bedingungen wird hierbei berücksichtigt. [Ligge03]
- **Problem:** Invariante (Teil-)Bedingungen!

2.4. Grundlagen

Modifizierter Bedingungs-/Entscheidungsüberdeckungstest

(MCDC, „Modified condition/decition coverage“)

- Fordert Testfälle die belegen, dass **jede atomare** Bedingung einen Einfluss auf den Wahrheitswert der gesamten Bedingung hat, unabhängig von den anderen Bedingungen.

- Dieser Test benötigt linearen Testaufwand, da zum Testen einer Bedingung mit n atomaren Bedingungen höchstens n+1 Testfälle benötigt werden.

- **Anmerkung:** Dieser Test wird vom RTCA DO-178B Standard für Luftfahrt-Software gefordert. [RCTA92]

2.5. Grundlagen

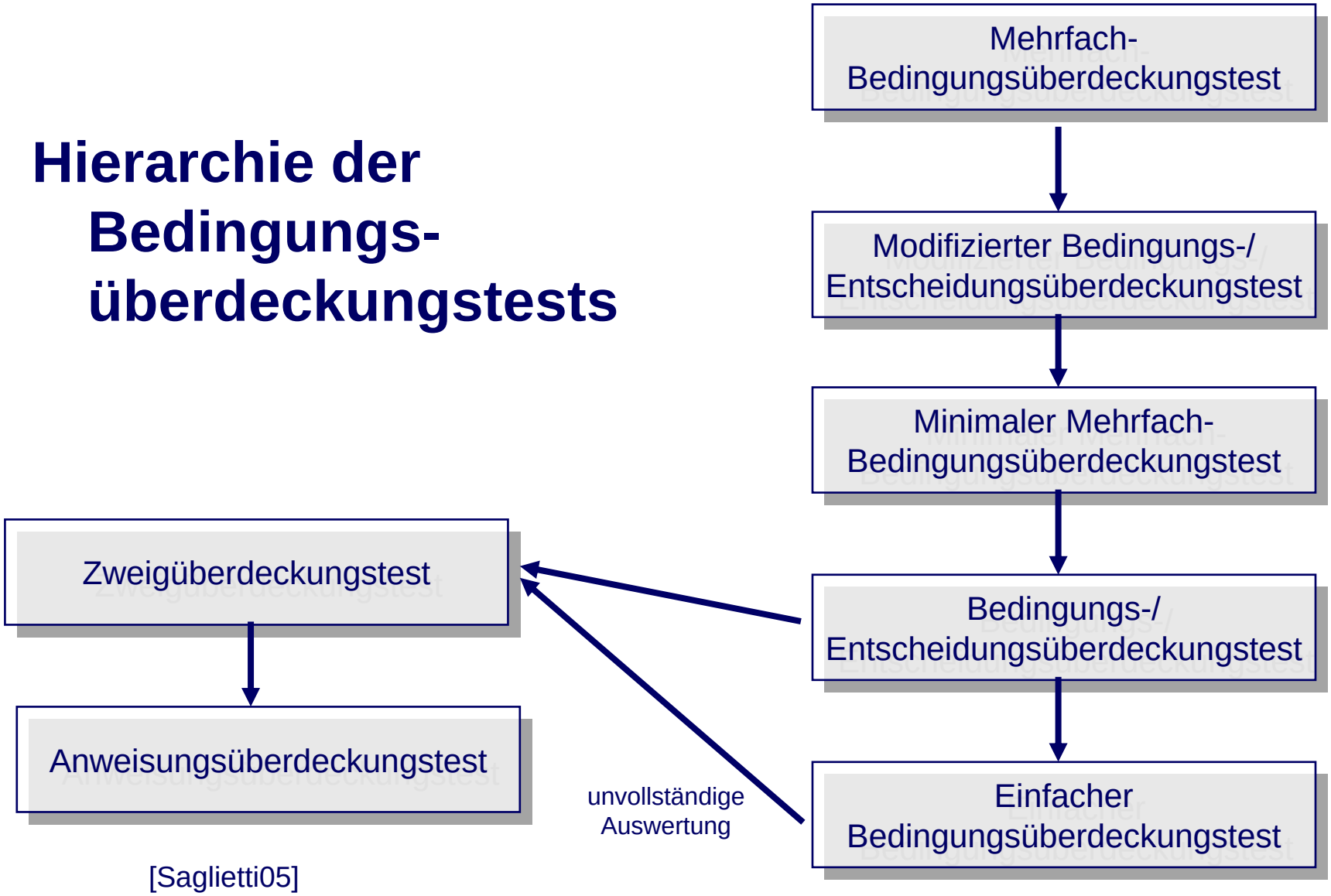
Mehrfach-Bedingungsüberdeckungstest

(„Multiple condition coverage“)

- Fordert, dass alle möglichen Wahrheitswert-Kombinationen der Bedingungen getestet werden.
- **Problem:** Dieser Test ist der aufwändigste Bedingungs-überdeckungstest, da für n atomare Bedingungen 2^n Testfälle benötigt werden.
- **Weiteres Problem:** Invariante (Teil)-Bedingungen, für die keine Testfälle gefunden werden können.

2.5. Grundlagen

Hierarchie der Bedingungs-überdeckungstests



[Saglietti05]

2.6. Grundlagen

Übertragung der „klassischen“ Bedingungsüberdeckungstests auf Java

- Bei den Anweisungen **„if“**, **„while“**, **„do while“** und **„for“** ist keine besondere Behandlung der Bedingung nötig, da der Ausdruck explizit mit der Anweisung angegeben wird.

- **Schwierigkeit:** Statisch festzustellen, ob die atomaren Bedingungen „boolean“ sind oder nicht (z.B. bei „if (a | b) …“)

- Deshalb wird beim Matchen einer Variablen der Typ bestimmt um festzustellen, ob diese Variable „boolean“ ist oder nicht (bei lokalen Variablen durch eine **Variablentabelle**, externe Referenzen durch **Reflection**).

2.6. Grundlagen

- Bedingungen müssen nicht immer explizit sein!
- So stellt z.B. die Auswahl eines Falles in der Switch-Case-Anweisung ebenfalls eine Bedingung dar
- Für solche implizite Bedingungen werden explizite Bedingungen erzeugt, die dann mit den bereits beschriebenen Kriterien analysiert werden können
- **Betroffene Anweisungen/Konzepte:** ternäre Operator (:?), Switch-Case-Anweisung, Polymorphie und dynamisches Binden sowie Überladen von Methoden
- Außerdem müssen Vorkehrungen für evtl. bei der Auswertung auftretende Ausnahmen (Exceptions) getroffen werden

2.6. Grundlagen

Ternärer Operator (?:)

- Ist der Ausdruck vor dem Fragezeichen wahr, so wird der Teilausdruck vor dem Doppelpunkt, sonst derjenige nach dem Doppelpunkt zurückgeliefert. Der resultierende Typ der beiden alternativen Ausdrücke muss deshalb gleich sein.

- Der Ausdruck vor dem Fragezeichen wird als zu analysierende Bedingung angesehen, da der ternäre Operator leicht in eine If-Else-Anweisung umgewandelt werden könnte.

Beispiel:

```
…
String s = „Datei kann gelesen
   werden:“;
s += (f.canRead()) ? („ja“) :
   („nein“);
…
```

```
…
if (f.canRead()) {
   s += „ja“;
} else {
   s += „nein“;
}
…
```

2.6. Grundlagen

Switch-Case-Anweisung

- Die Switch-Case-Anweisung vergleicht nacheinander den Ausdruck hinter dem „switch" (dieser muss ein primitiver Typ wie „byte“, „char“, „short“ oder „int“ sein) mit jedem einzelnen Fallwert („case“). Stimmt dieser Ausdruck mit der Konstanten überein, so wird der Anweisungsblock hinter der Sprungmarke ausgeführt.

- Dieser Vergleich wird zur Konstruktion der expliziten Bedingung verwendet. Das Loggen dieser Bedingung erfolgt im entsprechenden Anweisungsblock.

- Um eine eindeutige Zuordnung zu ermöglichen werden alle Fälle („cases“) durch ein **`enterSwitch( x, Id)`** und einem **`leaveSwitch( Id )`**, dass umgehend nach der Switch-Case-Anweisung eingefügt wird, eingeschlossen.

Beispiel:

```
…
switch (x) {
   case y : Anweisung(en)1;
   case z : Anweisung(en)2; break;
   default: Anweisung(en)3;
}
…
```

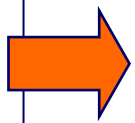
```
…
switch ( enterSwitch( x, Id ) ) {
   case y : Logge (x == y);
            Anweisung(en)1;
   case z : Logge (x == z);
            Anweisung(en)2; break;
   default: Logge (x == default);
             Anweisung(en)3;
}
leaveSwitch( Id );
…
```

Polymorphie und dynamisches Binden

- **Polymorphie** (griechisch, „Vielgestaltigkeit“): Eigenschaft einer Variablen, Instanzen unterschiedlicher Klassen (also Objekte) aufnehmen zu können
- Es gilt bei polymorphen Variablen, dass das Objekt eine Instanz des deklarierten Variablentyps, oder einer davon abgeleiteten Klasse sein muss.
- **Dynamisches Binden:** Die Wahl der richtigen Methode wird zur Laufzeit getroffen.
- Das dynamische Binden kann durch einen If-Else-Anweisung und dem Operator „instanceof“ ausgedrückt werden.

2.6. Grundlagen

```
publ ic  class A {
    public  String  toString() {
           return "Klasse A" ;
    }
}
public class B extends A {
    public String toString() {
           return "Klasse B" ;
    }
}
public class C extends A {
    public String toString() {
           return "Klasse C" ;
    }
}
public class TestClass {
    public static void main (String[] args) {
           A obj; // obj ist polymorph
           obj = new A();
           println(obj.toString()); //"Klasse A"
           obj = new B();
           println(obj.toString()); //"Klasse B"
           obj = new C();
           println(obj.toString()); //"Klasse C"
    }
}
```



```
public class TestClass {
    public static void main (String[] args) {
           A obj; // obj ist polymorph
           …
           obj = new B();
           if (obj instanceOf A) {
                      println(A.toString());
           } else if (obj instanceOf B) {
                      println(B.toString());
           } else if (obj instanceOb C) {
                      println(C.toString());
           }
           …
    }
}
```

2.6. Grundlagen

Überladen von Methoden

- **Überladen einer Methode:** Definition mehrerer Methoden mit dem selben Namen, aber unterschiedlichen Signaturen
- Die Wahl der richtigen Methode wird anhand der übergebenen Parameter zur Kompilationszeit entschieden (= **statisches Binden**) außer beim dynamischen Binden
- D.h., die übergebenen Parameter stellen ebenfalls eine Bedingung dar

2.6. Grundlagen

Ausnahmen (Exceptions) (1)

- Der Zugriff auf eine nicht initialisierte Variable bzw. der Aufruf einer Methode in einem Ausdruck kann u.U. eine Exception werfen
- Wird eine solche Exception geworfen, ist der Wert des Ausdrucks unbekannt und darf deshalb nicht ins Ergebnis mit einfließen
- Auch die Weiterverarbeitung der Exception soll nicht behindert werden
- Um solche Exceptions erkennen zu können, wird eine Methode `catchException(int, boolean, int)` hinzugefügt, deren erster und dritter Parameter als Platzhalter für die beiden Methoden `startExpression(id)` und `endExpression(id)` dienen
- Der überwachte Ausdruck steht bei dieser Methode an zweiter Stelle
- Diese „Kapselung“ des überwachten Ausdrucks ist nötig, da dort wo ein Ausdruck erwartet wird, keine Anweisungsfolge stehen darf
- `catchException()`muss außerdem den überwachten Ausdruck wieder zurückliefern, da das Ergebnis von `catchException()` als Entscheidung für die entsprechende Anweisung („if“, „while“, …) dient.

2.6. Grundlagen

Ausnahmen (Exceptions) (2)

- Die Auswertung des Ausdrucks wird durch den Aufruf der Methode `startExpression(id)` eingeleitet und durch `endExpression(id)` abgeschlossen.
- Dabei bezeichnet ID die Identifikation des überwachten Ausdrucks
- Außerdem werden allen Methoden in allen Klassen durch einen allumfassenden `try/finally`-Block eingeschlossen
- Tritt jetzt eine Exception auf, wird zwar die Methode `startExpression(id)` aufgerufen, aber nicht die `endExpression(id)`-Methode, sondern es wird umgehend in den `finally`-Block verzweigt
- In diesem Fall wird die Methode `handleException(methodId)` ausgeführt
- Somit wird erkannt, wo genau eine Exception aufgetreten ist, ohne die Weiterverarbeitung der Exception durch evtl. bereits vorhandene `try/finally`-Blöcke zu beeinflussen.
- D.h. also, das Ergebnis einer (Teil-)Bedingung kann nicht nur „true“ oder „false“, sondern auch „unknown“ sein.

Definition der Überdeckungsmetriken

- Der Überdeckungsgrad gibt die Vollständigkeit eines Tests bezogen auf ein bestimmtes Testkriterium an. [Ligge03]
- Bei strukturorientierten Tests ist der Überdeckungsgrad wie folgt definiert:

$$UG = \frac{\text{Anzahl überdeckter Elemente}}{\text{Anzahl von Kriterium geforderte Elemente}}$$

- Bei den Bedingungsüberdeckungstests entsprechen die Elemente den Bedingungen
- Eine Bedingung gilt als überdeckt, wenn sie mindestens einmal zu Wahr und einmal zu Falsch ausgewertet wurde

Verschiedene Ausprägungen des Überdeckungsgrads (1)

- **Grobgranular:** Alle Bedingungen im gesamten Programm und alle Testfälle

$$UG_{grobgranular} = \frac{\text{Anzahl überdeckter Bedingungen aller Testfälle}}{\text{Anzahl vom Kriterium geforderten Bedingungen des ganzen Programms}}$$

- **Feingranular:** Überdeckungsgrad für jede Gesamtbedingung b und für jeden Testfall t

$$UG_{feingranular} = \frac{\text{Anzahl vom Testfall t ueberdeckter Teilbedingungen einer Bedingung b}}{\text{Anzahl der vom Kritierium k geforderten Teilbedingungen der Bedingung b}}$$

- Zwischen den beiden Extremen grobgranular und feingranular können weitere Abstufungen definiert werden, z.B. pro Methode, pro Klasse, …

Verschiedene Ausprägungen des Überdeckungsgrads (2)

- **Switch-Case-Coverage Überdeckungsgrad:** *grobgranular* (alle Switch-Case-Anweisungen und alle Testfälle) und *feingranular* (pro Switch-Case und pro Testfall)

$$UG_{\text{Switch-Case}} = \frac{\text{Anzahl überdeckter Switch-Case-Fälle}}{\text{Anzahl in einem Switch-Case vorhandene Fälle}}$$

- **Method Coverage Überdeckungsgrad:** *grobgranular* (alle Methoden der Programms und alle Testfälle) und *feingranular* (pro (überladene) Methode und pro Testfall):

$$UG_{\text{Methods}} = \frac{\text{Anzahl überdeckter Methoden}}{\text{Anzahl geforderter Methoden}}$$

3. ANTLR

Parser-Generator

3. ANTLR (1)

- Um die Bedingungsüberdeckung messen zu können, werden zusätzliche Informationen über das PUT („*Program under test*“) benötigt.

- Um an diese Informationen zu kommen, wurden folgende Ansätze verglichen:

  - **Explizite Anreicherung**: Der Programmierer fügt selbstständig Proben beim Schreiben des Quelltextes ein, **aber** umständlich bei verschachtelten Bedingungen und fehleranfällig.

  - **Anpassung der JVM**: Die JVM bietet eine Schnittstelle, um durch externe Programme den Programmstatus erfragen und die Ausführung der Anweisungen kontrollieren zu können. **Aber:** Methoden zur Analyse der Bedingungsauswertung werden vom JVM TI („Tool Interface“) nicht angeboten [INTSUN]

  - **Instrumentierung des Quelltextes**: Automatische Instrumentierung des Quelltextes durch Überführung in einen AST („Abstract Syntax Tree“, abstrakter Syntaxbaum) und anschließende Instrumentierung → sehr flexibel und deshalb gewählter Ansatz

3. ANTLR (2)

- Akronym für „ANother Tool for Language Recognition“
- Stellt ein Framework zum Erzeugen von Parsern , Compilern und Übersetzern aus Grammatiken zur Verfügung.
- ANTLR erstellt einen rekursiven LL(k)-Parser in Java oder C/C++ für die übergebene Grammatik.
- LL(k) bedeutet, dass jeder Ableitungsschritt von links nach rechts und mit einem Vorausblick („Lookahead“) von k Tokens erfolgt.
- ANTLR ist sehr beliebt aufgrund seiner Einfachheit, Mächtigkeit, Flexibilität, weil es für den Menschen lesbaren Code generiert und außerdem kostenlos ist.
- ANTLR unterstützt außerdem hervorragend das Erzeugen von abstrakten Syntaxbäumen (AST) und das Durchlaufen sowie Transformieren solcher Syntaxbäume mittels Grammatiken.

[IntANTLR]

3. ANTLR (3)

- Die verwendete Grammatik ähnelt der erweiterten Backus-Naur-Form (EBNF), ist also kontextfrei.
- Die Regeln der Grammatik können durch Aktionen angereichert werden, die beim Matchen einer Regel ausgeführt werden.
- ANTLR erzeugt aus einer Grammatik folgende Komponenten:
  - **Lexer/Tokenizer:** Erstellt aus einer Folge von Zeichen logisch zusammengehörige Einheiten, sog. Tokens
  - **Parser/Recognizer:** Entscheidet, ob eine Folge von Tokens zur Sprache einer bestimmten Grammatik gehört. Außerdem führt der Parser evtl. vorhandene Aktionen aus, wenn für die entsprechende Regel eine Übereinstimmung gefunden wurde.

  bzw.

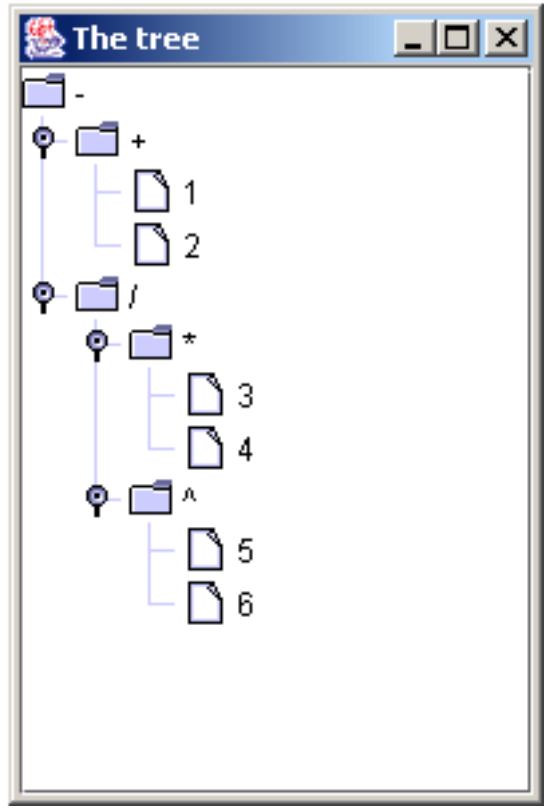
  - **TreeParser/TreeWalker:** Es gilt das Gleiche wie beim o.g. Parser nur wird als Eingabe keine Folge von Tokens verwendet, sondern ein AST, der nach den Regeln der Grammatik durchlaufen wird.

3. ANTLR (4)

Beispiel Grammatik für das Parsen einer Sprache:

```
class ExpressionLexer extends Lexer;
   PLUS  : '+' ;
   MINUS: '-' ;
   MUL   : '*' ;
   DIV   : '/' ;
   MOD   : '%' ;
   POW   : '^' ;
   SEMI  : ';' ;
   protected DIGIT : '0'..'9' ;
   INT : (DIGIT)+ ;


class ExpressionParser extends Parser;
options { buildAST=true; }
   expr          : sumExpr SEMI!;
   sumExpr       : prodExpr ((PLUS^|MINUS^) prodExpr)*;
   prodExpr      : powExpr ((MUL^|DIV^|MOD^) powExpr)*;
   powExpr       : atom (POW^ atom)?;
   atom          : INT;
```



Beispiel-AST von Ausdruck (1+2-3*4/5^6)

[IntMILLS]

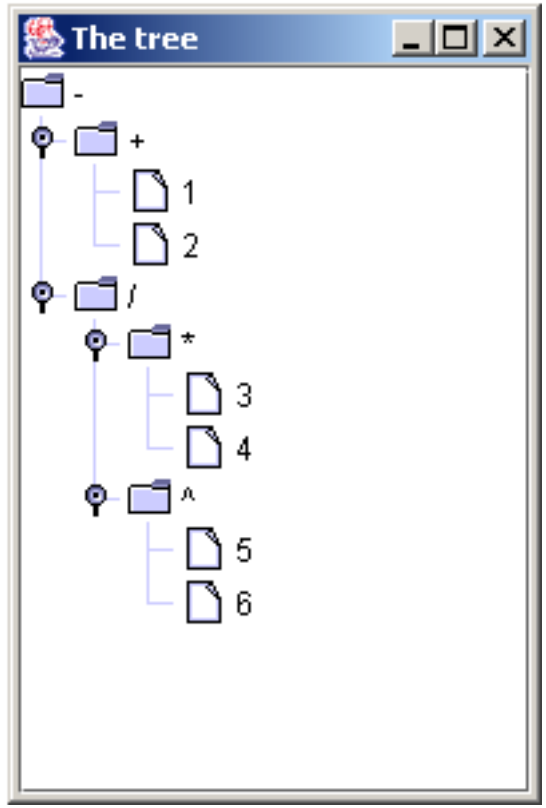
3. ANTLR (5)

Beispiel Grammatik für das Parsen eines ASTs:

```
class ExpressionTreeWalker extends TreeParser;

expr returns [double r]
{ double a,b; r=0; }
    : #(PLUS a=expr b=expr)  { r=a+b; }
    | #(MINUS a=expr b=expr) { r=a-b; }
    | #(MUL a=expr b=expr)   { r=a*b; }
    | #(DIV a=expr b=expr)   { r=a/b; }
    | #(MOD a=expr b=expr)   { r=a%b; }
    | #(POW a=expr b=expr)   { r=Math.pow(a,b); }
    | i:INT { r=(double)Integer.parseInt(i.getText()); }
    }
;
```

[IntMILLS]

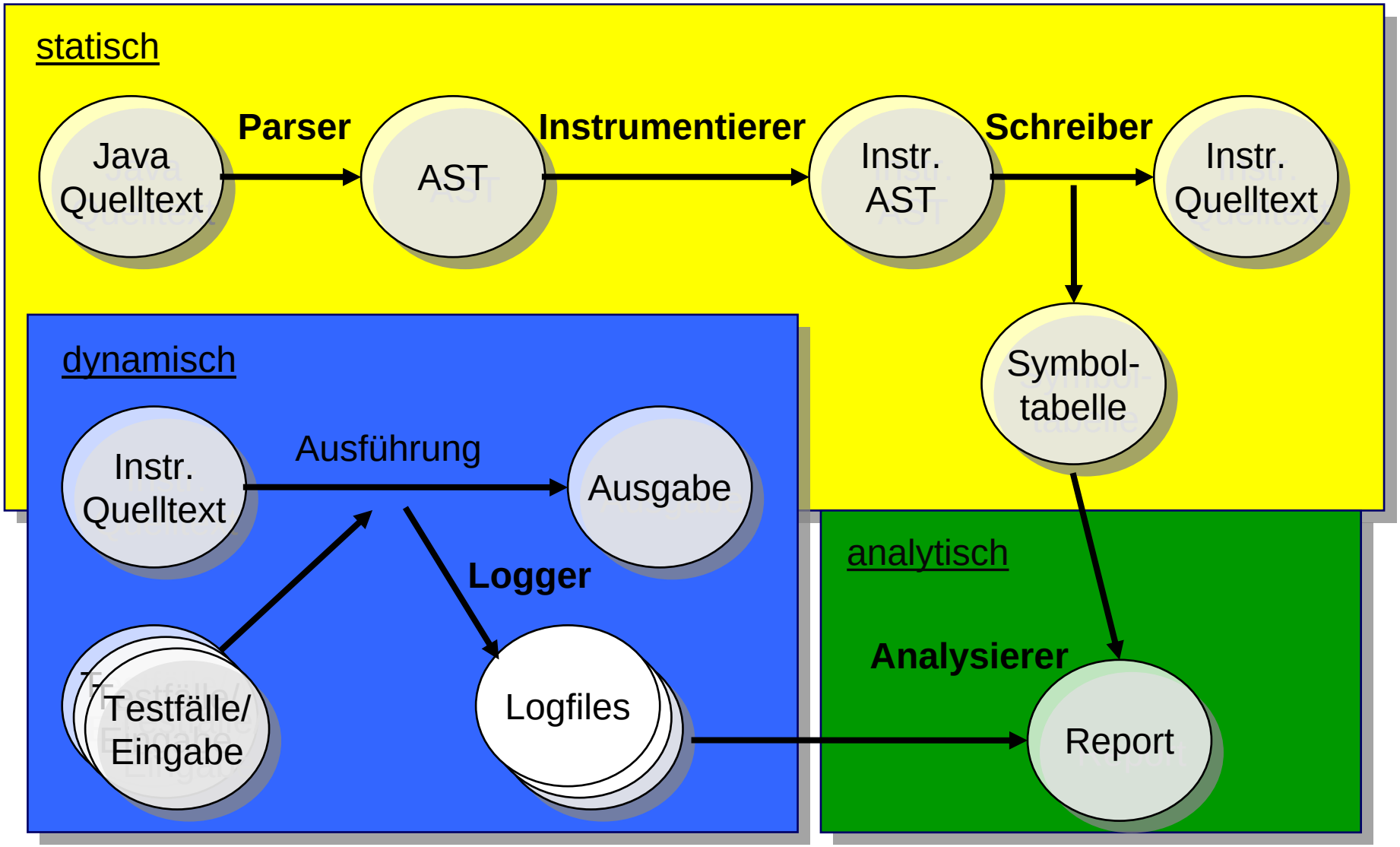


Beispiel-AST von Ausdruck (1+2-3*4/5^6)
Ergebnis: 2.999232

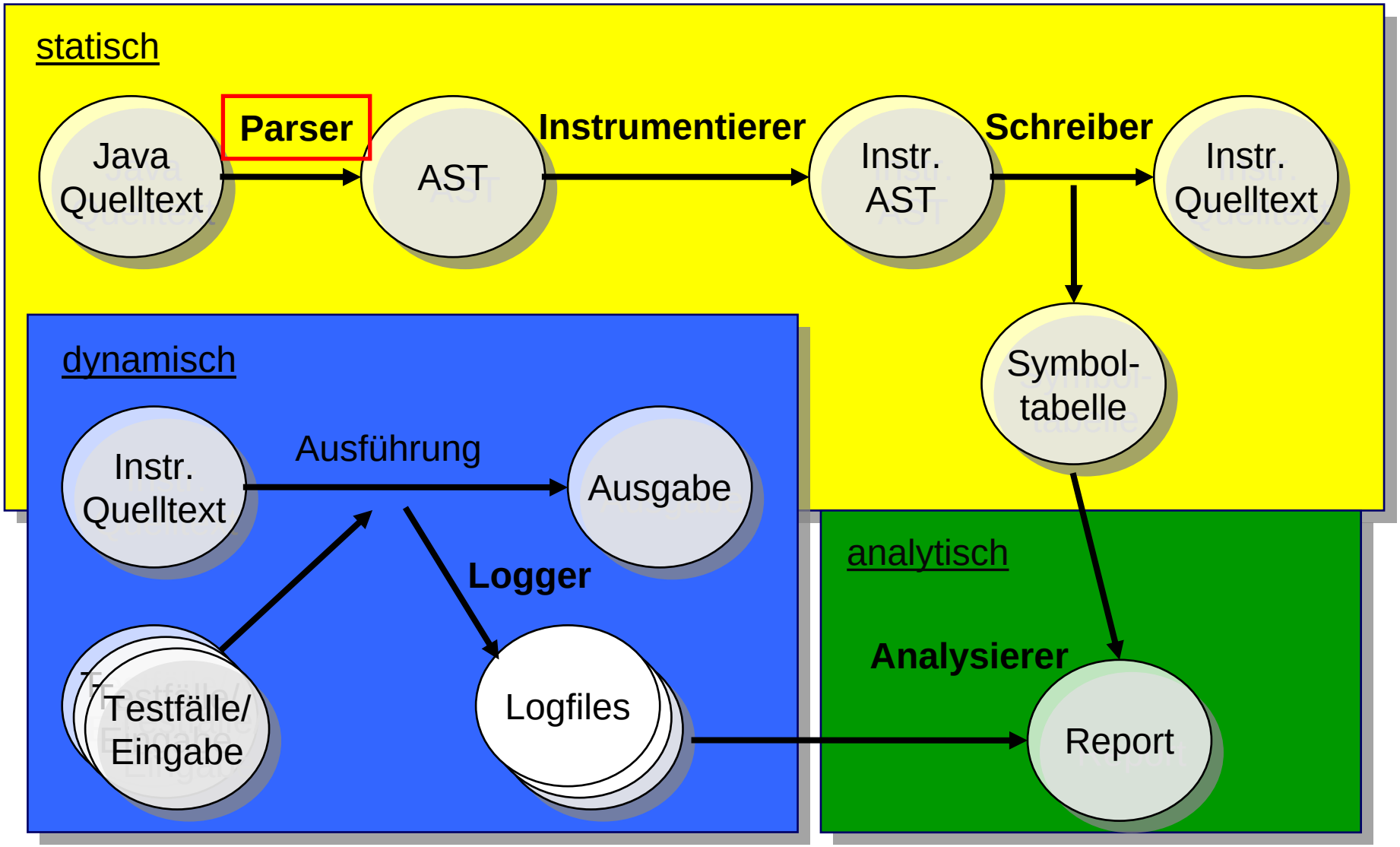
4. Das Werkzeug

Statische und dynamische
Analyse des Java-Quelltextes.

4. Das Werkzeug



4.1. Das Werkzeug

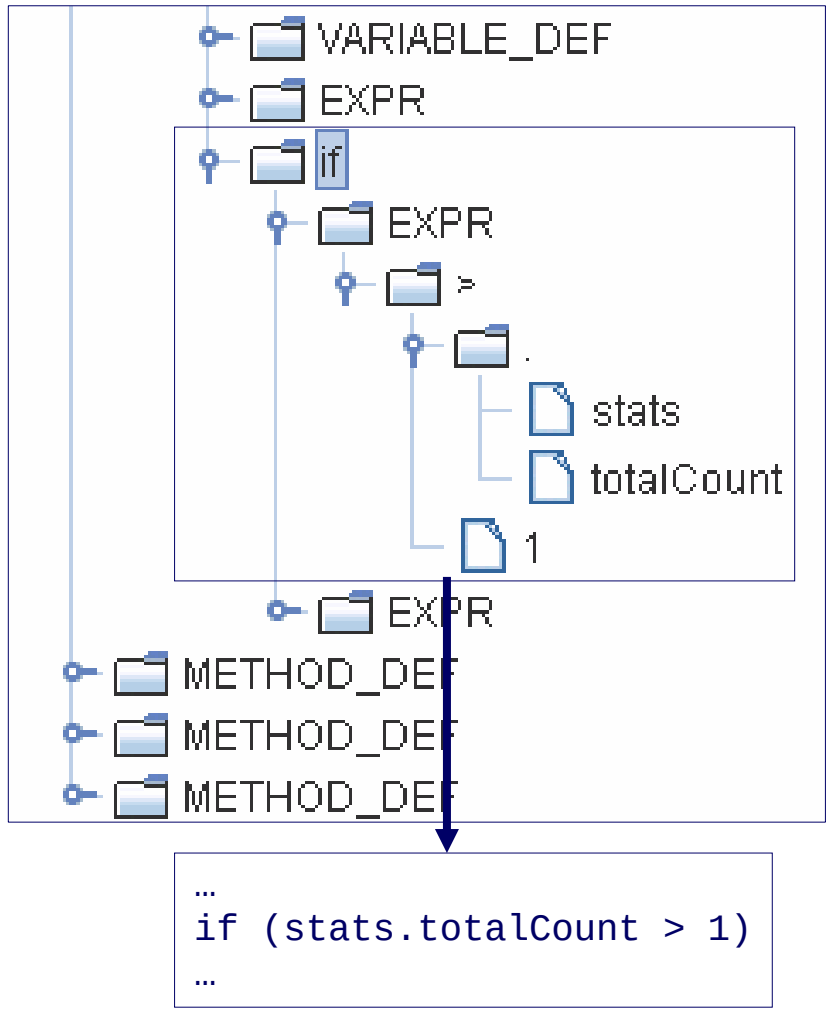


4.1. Das Werkzeug

Parser: JavaTokenizer + JavaRecognizer

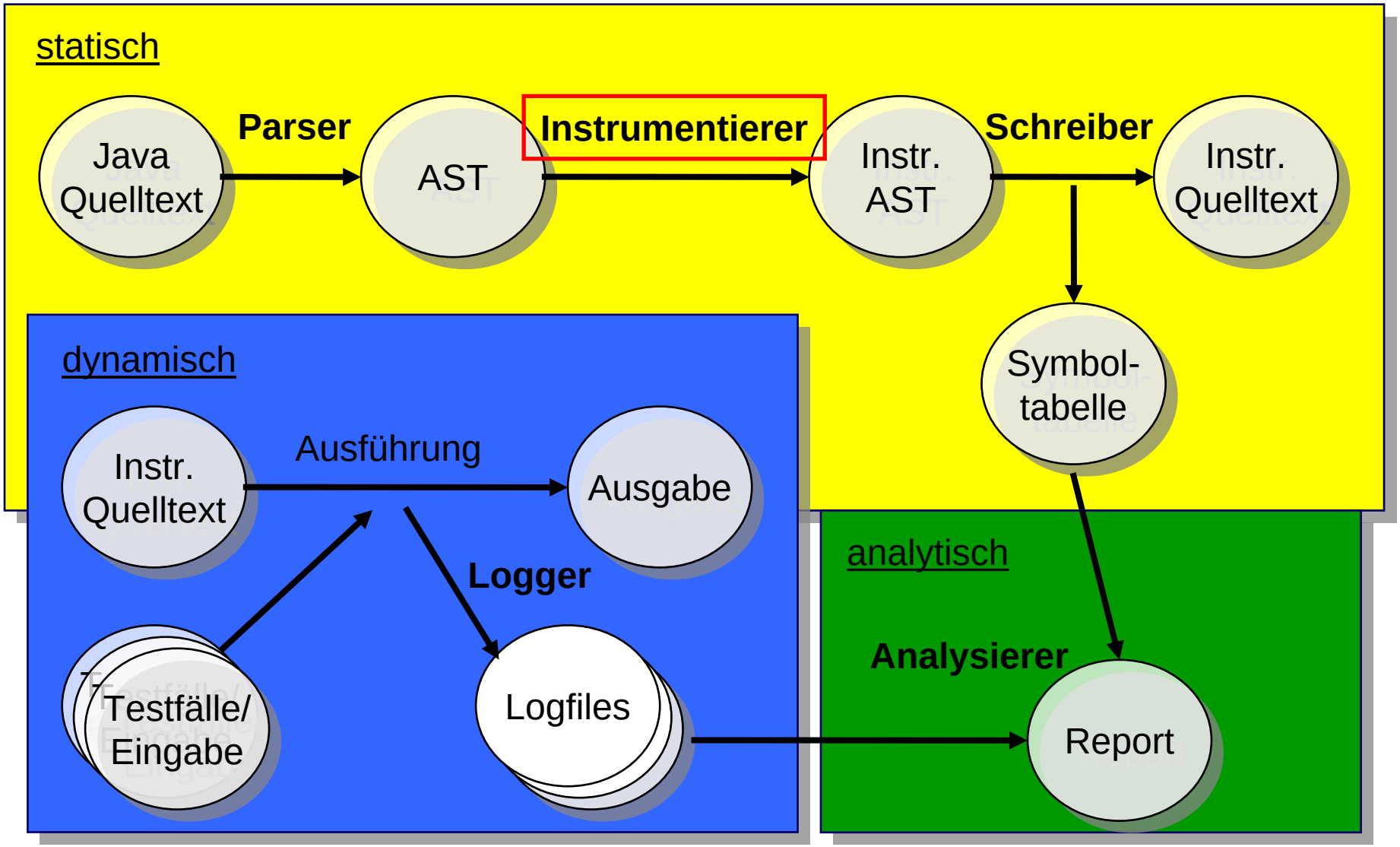
(erstellt aus Grammatik „java15.g“)

- Liest den Java-Quelltext ein und bestimmt die relevanten Token
- Danach wird aus den Token ein AST nach den Regeln der Grammatik gebildet (siehe rechts)




```
…
// If-else statement
| "if"^ LPAREN! expression RPAREN! statement
( : "else"! statement )?
…
```

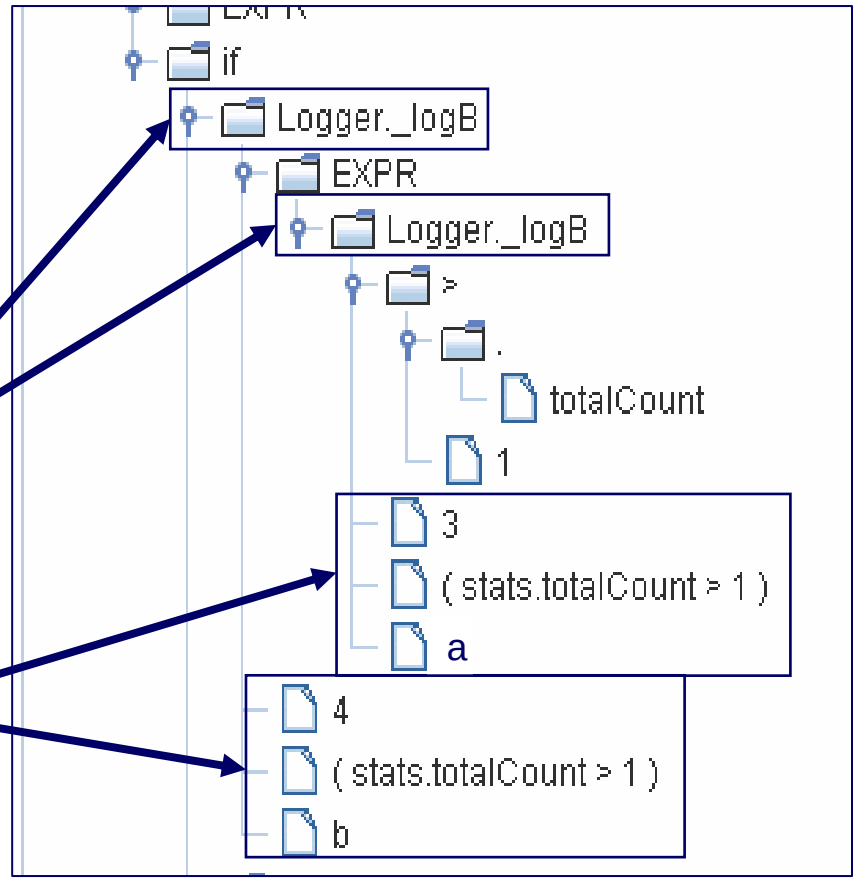
4.1. Das Werkzeug



4.1. Das Werkzeug

Instrumentierer:
JavaTreeParser
(erstellt aus Grammatik „java15.tree.instrumenter.g“)

- Übernimmt vom Parser den AST und instrumentiert diesen
- Dazu werden an den relevanten Stellen zusätzliche Knoten eingefügt
- Diese Knoten besitzen weitere Informationen wie z.B. die laufend durchnummerierte ID, der bis dahin zusammengesetzte Ausdruck und die Art des Ausdrucks



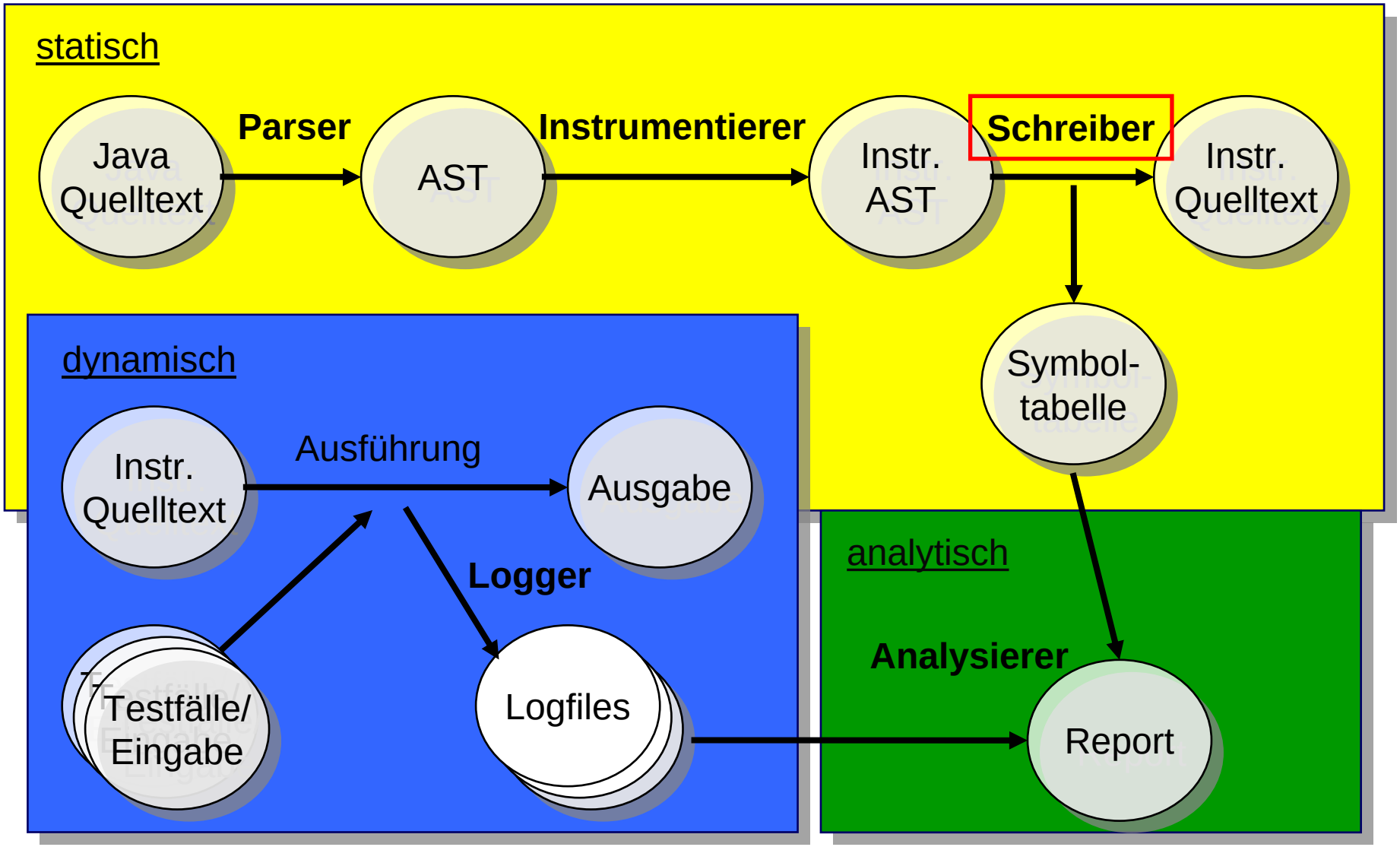

```
…
if ( Logger._logB( Logger._logB( stats.totalCount > 1, 3 ), 4 ) ) {
…
```

4.1. Das Werkzeug

Instrumentierer (2)

- Instrumentiert werden wie in den Grundlagen beschrieben:
  - Bedingungen
  - Switch-Case Anweisungen
  - Ternäre Operatoren
  - Methoden / Konstruktoren
  - Klassen
- Außerdem wird in dieser Klasse der Typ einer Variable bzw. Methode bestimmt und entsprechend instrumentiert

4.1. Das Werkzeug



4.1. Das Werkzeug

Schreiber: JavaTreeWriter (erstellt aus der Grammatik „java15.tree.writer.g“)

- Übernimmt vom Instrumentierer den instrumentierten AST und schreibt diesen als instrumentierten Java-Quelltext zurück
- Beim Zurückschreiben fügt diese Komponente außerdem die Methoden für die Behandlung der Ausnahmen und der Switch-Case Anweisungen ein
- Weiterhin erstellt die Schreiber-Komponente die Symboltabelle aus den instrumentierten AST
- Inhalt einer Symboltabelle:
  - Bedingungen (explizit und implizit)
  - die Bezeichner der definierten Methoden, Konstruktoren und Klassen

4.1. Das Werkzeug

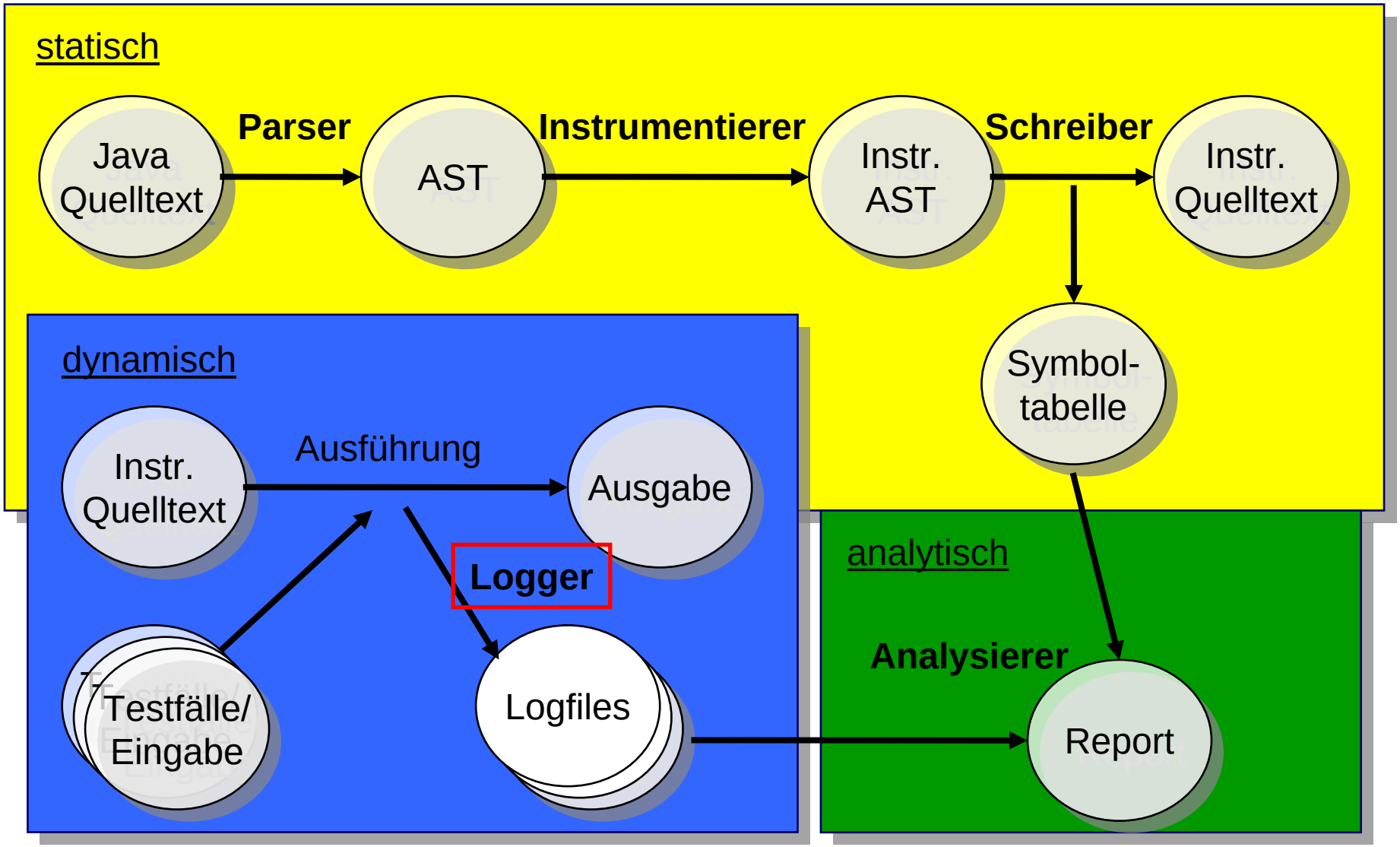
Schreiber: JavaTreeWriter

- Eine Bedingung kann von folgendem Typ sein:
  - **„a“**: die Bedingung ist atomar
  - **„p“**: die Bedingung ist primär, also eine bool‘sche Variable („Primary“)
  - **„c“**: die Bedingung ist zusammengesetzt („Combined“)
  - **„b“**: die Bedingung ist eine Entscheidung („Decition“/„Branch“)
  - **„t“**: die Bedingung stammt vom ternären Operator
  - **„s“**: die Bedingung stammt von einem Fall („Case“) einer Switch-Case-Anweisung
- weiteren möglichen Typen eines Eintrags in der Symboltabelle:
  - **„m“:** bezeichnet eine Methode
  - **„o“:** bezeichnet einen Konstruktor
  - **„k“:** bezeichnet eine Klasse
  - **„w“**: bezeichnet eine Switch-Case-Anweisung

4.1. Das Werkzeug

ID	Row	Type	Expression	Notes
1	5	ke	BigFib	
2	14	o	BigFib ( )	
3	27	m	getFib (int c, PrintStream printTo)	
4	29	a	(c > 0)	
5	29	b	(c > 0)	
6	32	a	null != printTo	
7	32	b	null != printTo	
8	38	a	printTo != null	
9	38	b	printTo != null	
10	42	p	qqqq	
11	42	c	qqqq || (!qqqq)	
12	42	b	qqqq || (!qqqq)	
13	48	a	(c == 0)	
14	48	b	(c == 0)	
15	63	m	main (String[] args)	
16	70	a	(args.length > 0)	
17	70	b	(args.length > 0)	
18	77	a	(limit < 1)	
19	77	b	(limit < 1)	
1	92	kl	BigFib	

4.2. Das Werkzeug



4.2. Das Werkzeug

Logger

- Beim Schreiben des instrumentierten Quelltextes durch die Schreiber-Komponente wurde außerdem Code für das Loggen hinzugefügt.
- Die Ausführung des instrumentierten Quelltextes mit den Eingabedaten (=Testfälle) führt ebenfalls Methoden des Loggers aus.
- Der Logger schreibt für jede (Teil-)Bedingung die ID, den Wert und die eindeutige ID des Threads in ein Logdatei.
- Außerdem wird jedes Betreten und Verlassen der Methoden bzw. Konstruktoren ebenfalls mit der ID in der Logdatei vermerkt.
- Dabei legt er bei jeder Ausführung ein separates Logfile an.

4.2. Das Werkzeug

Aufbau einer Logdatei (1)

- Ein Eintrag in der Logdatei besteht aus den 3 Long-Werten *ID*, *Value* und *ThreadID*
- Dabei bezeichnet die *ID* entweder eine Bedingung in der Symboltabelle (*ID* > 0), oder ein „besonderes“ Ereignis:
  - -2: Switch-Case Anweisung wurde betreten (switchEnter())
  - -3: Switch-Case Anweisung wurde verlassen (switchLeft())
  - -4: catchException wurde aufgerufen
  - -5: Evaluation einer Entscheidung wurde gestartet (startExpression())
  - -6: Evaluation einer Entscheidung wurde erfolgreich beendet (endExpression())
  - -7: Ein finally-Block wurde betreten (handleException(()
  - -8: Eine Methode bzw. ein Konstruktor wurde betreten (enterMethod())
  - -9: Eine Methode bzw. ein Konstruktor wurde verlassen (leftMethod())

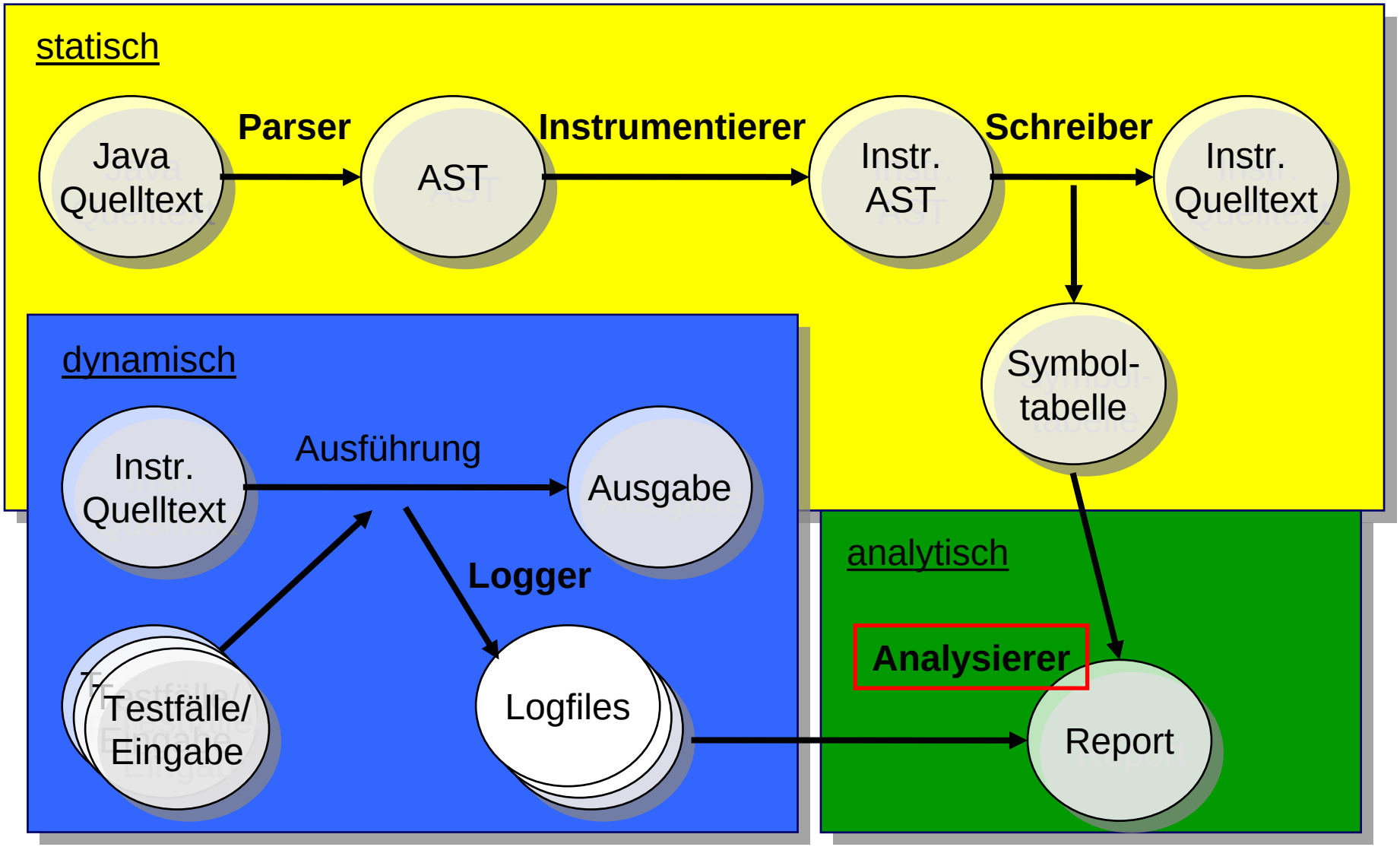
4.2. Das Werkzeug

Aufbau einer Logdatei (2)

- Bei einer Bedingung steht *Value* „0“ für Falsch, „1“ für Wahr und „2“ für unbekannt
- Bei einem „besonderen“ Ereignis bezeichnet *Value* die ID der Methode bzw. des Konstruktors in der Symboltabelle

```
0000h: FF FF FF FF FF FF FF F8 00 00 00 00 00 00 00 04
0010h: 00 00 00 00 00 00 00 01 FF FF FF FF FF FF FF FB
0020h: 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 01
0030h: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01
0040h: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 02
0050h: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01
0060h: FF FF FF FF FF FF FF FA FF FF FF FF FF FF FF FF
```

4.3. Das Werkzeug



4.3. Das Werkzeug

Analyzer (1)

- Bestimmt aus der Symboltabelle und den Logdateien, welche Bedingungsüberdeckungskriterien erfüllt sind und leitet daraus die Bedingungsüberdeckungsmetriken ab.

- Bei der einfachen Bedingungsüberdeckung, der Bedingungs-/Entscheidungsüberdeckung und der minimalen Mehrfachbedingungsüberdeckung wird wie folgt vorgegangen:

  - Es werden die Bedingungen mit den Typ „a“ und „p“, bzw. „a“, „p“, „b“ bzw. „a“, „p“, „b“, „c“ aus der Symboltabelle ausgewählt
  - Anschließend wird mit den in den Logdateien gespeicherten Informationen bestimmt, welche Bedingungen überdeckt wurden, und welche nur zu Falsch, nur zu Wahr oder gar nicht ausgewertet wurden

4.3. Das Werkzeug

Analyzer (2)

- Bei der modifizierten Bedingungs-/Entscheidungsüberdeckung wird wie folgt vorgegangen:
  - Der Analysierer betrachtet eine Auswertung der Gesamtbedingung als fest und vergleicht diesen mit den verbleibenden Auswertungen in den Logdateien
  - Wird eine Übereinstimmung mit den nicht betrachteten atomaren Bedingungen gefunden, wird das Gesamtergebnis der beiden Bedingungen verglichen
  - Sind beide Ergebnisse verschieden, dann wurde die betrachtete atomare Bedingung überdeckt
  - Sind beide Ergebnisse gleich, wird der nächste Testfall als fest angenommen und von vorne begonnen
  - **Beispiel 1 (a, b, c)**: (0**0**0) = 0, …, (0**1**0) = 1 → b hat einen Einfluss auf die Gesamtbedingung (vollständige Evaluation)
  - **Beispiel 2 (a, b, c):** (**0**00) = 0, …, (**1**x0) = 1 → a hat einen Einfluss auf die Gesamtbedingung (unvollständige Evaluation)

4.3. Das Werkzeug

Analyzer (3)

- Bei der Mehrfachbedingungsüberdeckung wird wie folgt vorgegangen:
  - Es wird eine Liste erstellt, in der alle möglichen Kombinationen atomarer Bedingungen einer Gesamtbedingung gespeichert werden
  - Anschließend werden die Auswertungen der Gesamtbedingung aus den Logdateien mit den Testfällen in der Liste verglichen
  - Bei einer Übereinstimmung wird der entsprechende Eintrag als überdeckt markiert

4.3. Das Werkzeug

Analyzer (4)

- Beispiel: Es existieren Logdateien mit den folgenden Auswertungen der Gesamtbedingung (a, b, c):
  - (0, 0, 0,), (0,1,0), (1, 0, 1) bei vollständiger Evaluation
  - Bei unvollständiger Evaluation werden nicht evaluierte Bedingungen als Wahr und Falsch angenommen (Wildcard)
  - (0, x, 0), (0, 1, x) … bei unvollständiger Evaluation

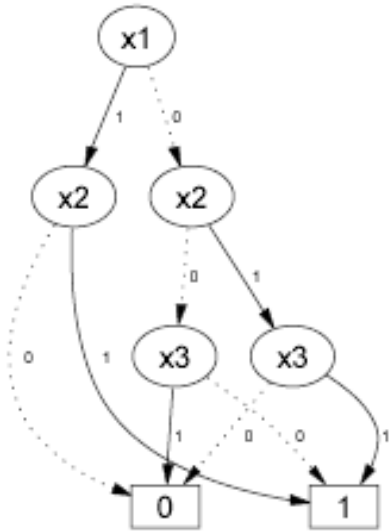
Über-deckt	a	b	c
V,U	0	0	0
	0	0	1
V,U	0	1	0
U	0	1	1
	1	0	0
V	1	0	1
	1	1	0
	1	1	1

5. Ausblick

5. Ausblick (1)

Statische Überprüfung der Erfüllbarkeit

- **Naiver Ansatz:** „Brute Force“
  - alle möglichen Wahrheitswertkombinationen durchprobieren
  - wenn das Gesamtergebnis immer Falsch oder Wahr ist, liegt eine Tautologie bzw. Kontradiktion vor
- **Verwendung eines Entscheidungsdiagramms** (ROBBD, „Reduced Ordered Binary Decition Diagram“)
  - **ROBBD:** geordneter, azyklischer Graph mit den beiden Senken „0“ und „1“
  - Die Senken entsprechen der Auswertung der Bedingung zu Falsch und Wahr
  - **Kontradiktion:** alle Kanten zeigen auf die Senke „0“
  - **Tautologie:** alle Kanten zeigen auf die Senke „1“



5. Ausblick (2)

Weitere mögliche Erweitungen

- Evtl. weitere Überdeckungsgrade (z.B. pro Package)
- Evtl. Vorauswertung der Bedingungen vornehmen

```
Boolean a, b, c, d;
Boolean e = (a || b) && (c || d);
If (e) { …}
```

  - **Naiver Ansatz:** Variable e durch den zugewiesenen Ausdruck ersetzen

```
Boolean a, b, c, d;
If ((a || b) && (c || d)) { …}
```

  - **Problem:** evtl. nicht im Sinne des Programmierers

6. Demonstration

Literaturverzeichnis

- **[Balz98]** Helmut Balzert, Lehrbuch der Softwaretechnik 2, Spektrum Akademischer Verlag, 1998
- **[Saglietti05]** „Verification & Validation“, Francesca Saglietti, WS 2005/06, Teil 10, S. 1-33
- **[RTCA92]** RCTA, "Software Considerations in Airborne Systems and Equipment Certification", 1992, S. 31, 74
- **[Ligge03]** „Software-Basistechnologie III / Software-Konstruktion II“, Peter Liggesmeyer, 2003
- **[IntHansel]** http://hansel.sourceforge.net/, Stand 05.12.2005
- **[IntANTLR]** http://www.antlr.org, Stand 06.12.2005
- **[IntMILLS]** http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/antlr/antlr.html, Stand 06.12.2005
- **[IntWIKI]** http://www.wikipedia.de, Stand 15.12.2005
- **[INTSUN]** http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html