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

Halbzeitvortrag zur Studienarbeit
von Dominik Schindler
am 20.12.2005

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
3. **ANTLR** („ANother Tool For Language Recognition“)
4. **Das Werkzeug**
    1. Statische Analyse
    2. Dynamische Analyse
    3. Messung der Überdeckung
5. **Stand der Arbeit**
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

Beispiel:

```
…
public static int countWords(String text) {
    int words = 0, chars = 0;
    char ch;
    text = text.trim();
    if (text.length() > 0) words++;
    for (int i = 0; i < text.length(); i++) {
            ch = text.charAt(i);
            if ((ch >= 'A') && (ch <= 'Z') || (ch >= 'a') && (ch <= 'z') || (ch ==
    ' ')) {
                    chars++;
            } else {
                    return -1;
            }
            if ((ch == ' ') || (ch == '\t') || (ch == '\n')) words++;
    }
    return words;
}
…
```

Testfall/Eingabe	Ergebnis
Dies ist ein Text	4
<leere Eingabe>	0
4711	-1

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 Auswertung“:** Unter unvollständiger Auswertung (engl. „short circuit evaluation“) versteht man das vorzeitige Beenden der Auswertung eines Ausdrucks, wenn das Endergebnis bereits feststeht.
  - Im Gegensatz dazu wird bei vollständiger Auswertung 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 (((u == 0) || (x > 5)) && ((y < 6) || (z == 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:

    (u = 1, x = 4, y = 5, **z = 0**) → „false“ und (u = 0, x = 6, y = 5, **z = 0**) → „true“
  - Trotzdem wurde die letzte Bedingung (z == 0) nicht ausreichend getestet, da im Falle von vollständiger Auswertung nur gegen „true“ getestet wurde.

1. Einleitung

Motivation (2):

- **Komplexe Entscheidung: (((u == 0) || (x > 5)) && ((y < 6) || (z == 0)))**
  - Testfall 1: (u = 1, x = 4, y = 5, **z = 0**) → „false“
  - Testfall 2: (u = 0, x = 6, y = 5, **z = 0**) → „true“
  - Im Falle von **unvollständiger Auswertung** – was gängige Praxis ist – wird die Bedingung **(z == 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 „true“ und „false“ 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 (x>-10)&&(x<10)||(y==0):

	(x>-10)	(x<10)	(y==0)	(x>-10)&&(x<10)	(x>-10)&&(x<10)||(y==0)
1	F	F	F	F	F
2	F	F	T	F	T
3	F	T	F	F	F
4	F	T	T	F	T
5	T	F	F	F	F
6	T	F	T	F	T
7	T	T	F	T	T
8	T	T	T	T	T

2.1. Grundlagen

Unvollständige Auswertung der Bedingung (x>-10)&&(x<10)||(y==0):

	(x>-10)	(x<10)	(y==0)	(x>-10)&&(x<10)	(x>-10)&&(x<10)||(y==0)
1,3	F	-	F	F	F
2,4	F	-	T	F	T
5	T	F	F	F	F
6	T	F	T	F	T
7,8	T	T	-	T	T

Bedingungs-/Entscheidungsüberdeckungstest

(„condition-/decition coverage“)

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

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

2.3. Grundlagen

Minimaler Mehrfach-Bedingungsüberdeckungstest

(„minimal multiple condition coverage“)

- Fordert, dass **jede** Bedingung, ob atomar oder nicht atomar, mindestens einmal gegen „true“ oder „false“ getestet wird.
- Dieser Test beinhaltet den Bedingungs-/Entscheidungsüberdeckungstest.
- D.h., die hierarchische/logische Struktur aller Bedingungen wird hierbei berücksichtigt. [Saglietti05]
- **Problem:** Invariante (Teil-)Bedingungen!

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

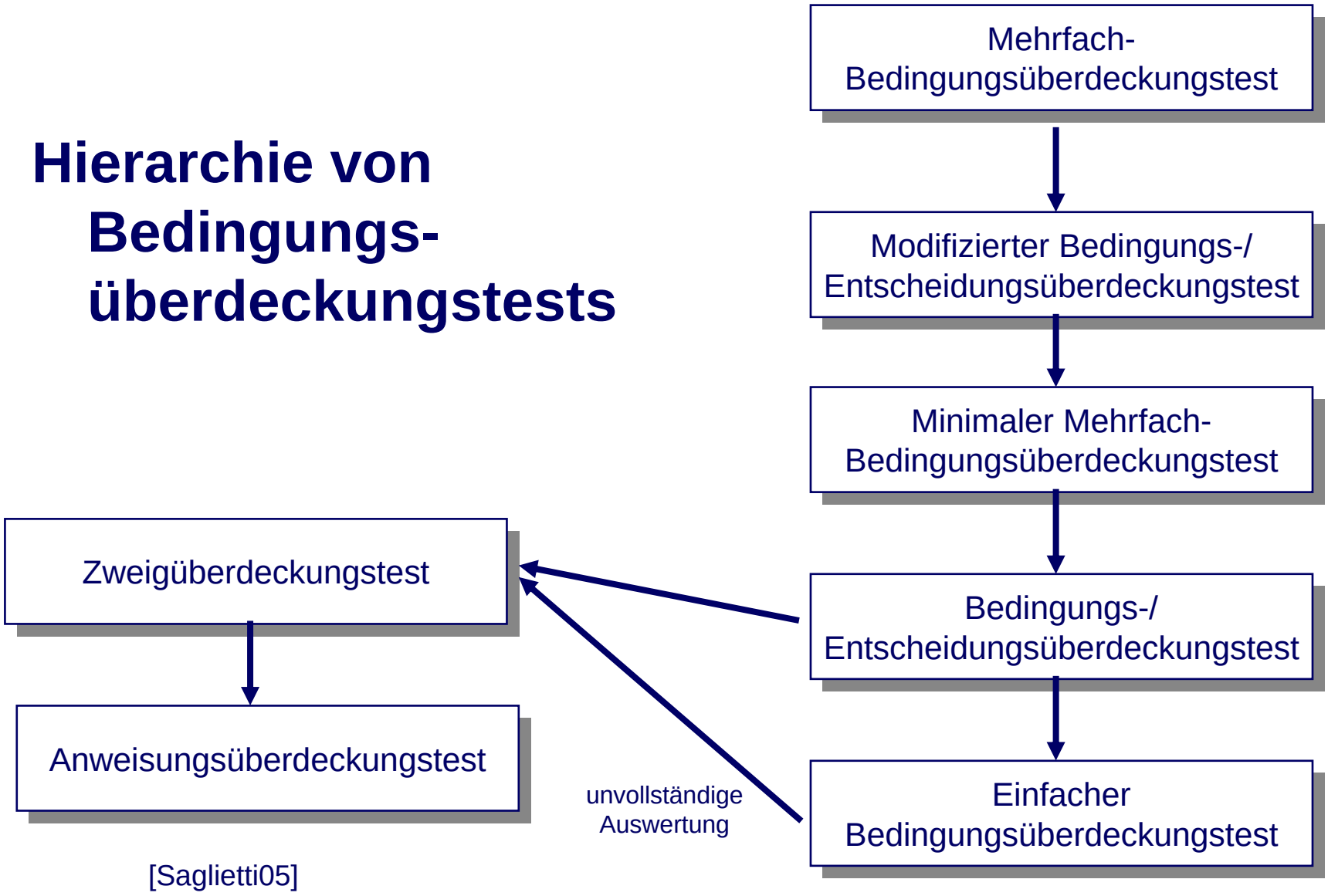
Mehrfach-Bedingungsüberdeckungstest

(„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 von Bedingungs-überdeckungstests



[Saglietti05]

2.6. Grundlagen

Übertragung der „klassischen“ Bedingungsüberdeckungstests auf Java

- Bei den Konstrukten **„if“**, **„while“**, **„do while“** und **„for“** ist keine besondere Behandlung der Bedingung nötig, da der Ausdruck so wie er ist analysiert werden kann.

- Trotzdem ist es schwierig bei o.g. Anweisungen 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

- Besondere Betrachtung benötigen der **ternäre Operator**, die **switch-case-Anweisung** sowie die Behandlung von **Exceptions**.

Ternärer Operator (?:)

- Ist der Ausdruck vor dem Fragezeichen „true“, 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 wie bei „if“ usw. als Bedingung angesehen.

Beispiel:

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

```
…
String s = „Datei kann gelesen
   werden:“;
s += (Logge (f.canRead(), 3 )) ?
   („ja“) : („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 zum Loggen der entsprechenden Bedingung verwendet. Das Loggen selbst erfolgt im entsprechenden Anweisungsblock.
- Um eine eindeutige Zuordnung zu ermöglichen werden alle Fälle („cases“) durch ein **`enterSwitch(x)`** und einem **`leaveSwitch(x)`**, 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) ) {
   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(x);
```

2.6. Grundlagen

Exceptions (1)

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

2.6. Grundlagen

Exceptions (2)

- Die Auswertung des Ausdrucks wird durch den Aufruf der Methode `startExpression(id)` eingeleitet und durch `endExpression(id)` abgeschlossen. Außerdem wird die **ganze Methode**, in der die entsprechende Anweisung vorkommt, 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 in den `finally`-Block verzweigt. In diesem Fall wird die Methode `handleException(„methodName“)` 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.

2.6. Grundlagen

Beispiel:

```
…
public static main () {
   if ( o.getValue() ) {
        Anweisungen;
   }
}
…
```

```
…
public static main () {
   try {
        …
        if (catchException(startExpression(5), Logge(o.getValue(), 5),
   endExpression(5)) {
                Anweisungen;
        …
        }
   } finally {
        handleException(„main“);
   }
}
…
```

2.6. Grundlagen

Polymorphie und dynamisches Binden

- Stellen Polymorphie und dynamisches Binden evtl. auch eine Art Bedingung dar?
- **Beispiel 1:** Abhängig von der an `obj` zugewiesenen Klassen-Instanz wird `A.toString()` bzw. `B.toString()` aufgerufen. D.h., dass der konkrete Typ der Variable `obj` zur Laufzeit ein Entscheidungskriterium darstellen könnte.
- **Beispiel 2:** Die unterschiedliche Belegung der Parameter von überladenen Methoden könnte ebenfalls als eine Art „Entscheidung“ betrachtet werden.
- Wie diese beiden Konzepte in diese Arbeit integriert werden können bedarf noch weiterer Nachforschungen.

Beispiel:

```
public class A {
    public String toString() {
          return "-A-";
    }
}
public class B extends A {
    public String toString() {
          return "-B-";
    }
}
public class C {
    private static void print(A obj) {
      System.out.println(obj.toString());
    }
    public static void main(String[] args) {
          A obj; // obj ist polymorph

          obj = new A();
          print(obj);   // „-A-“

          obj = new B();
          print(obj);   // „-B-“
    }
}
```

[IntWIKI]

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, wurde folgende Ansätze verglichen:

  - **Explizite Anreicherung**: Der Programmierer fügt Proben beim Schreiben des Quelltextes ein, **aber** keine Automatisierung und fehleranfällig. [IntHansel]

  - **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 („recognizers“), Compilern und Übersetzern („translators“) 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 „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 ASTs und das Durchlaufen sowie Transformieren von ASTs.

[IntANTLR]

3. ANTLR (3)

- Die verwendete Grammatik ähnelt der EBNF (=erweiterten Backus-Naur-Form), 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.

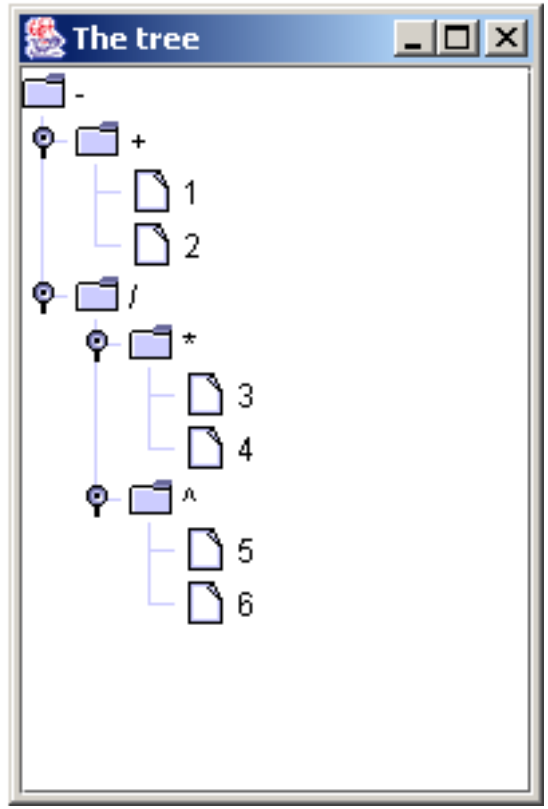
3. ANTLR (4)

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;
```

[IntMILLS]



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

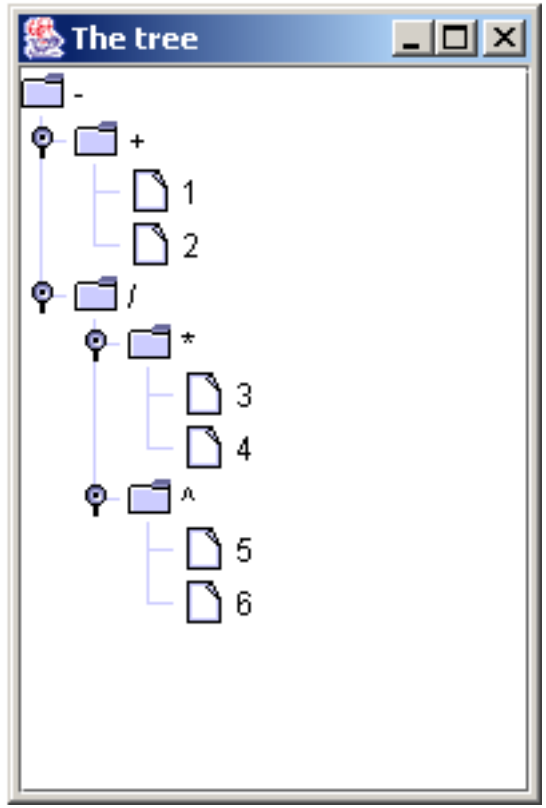
3. ANTLR (5)

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]

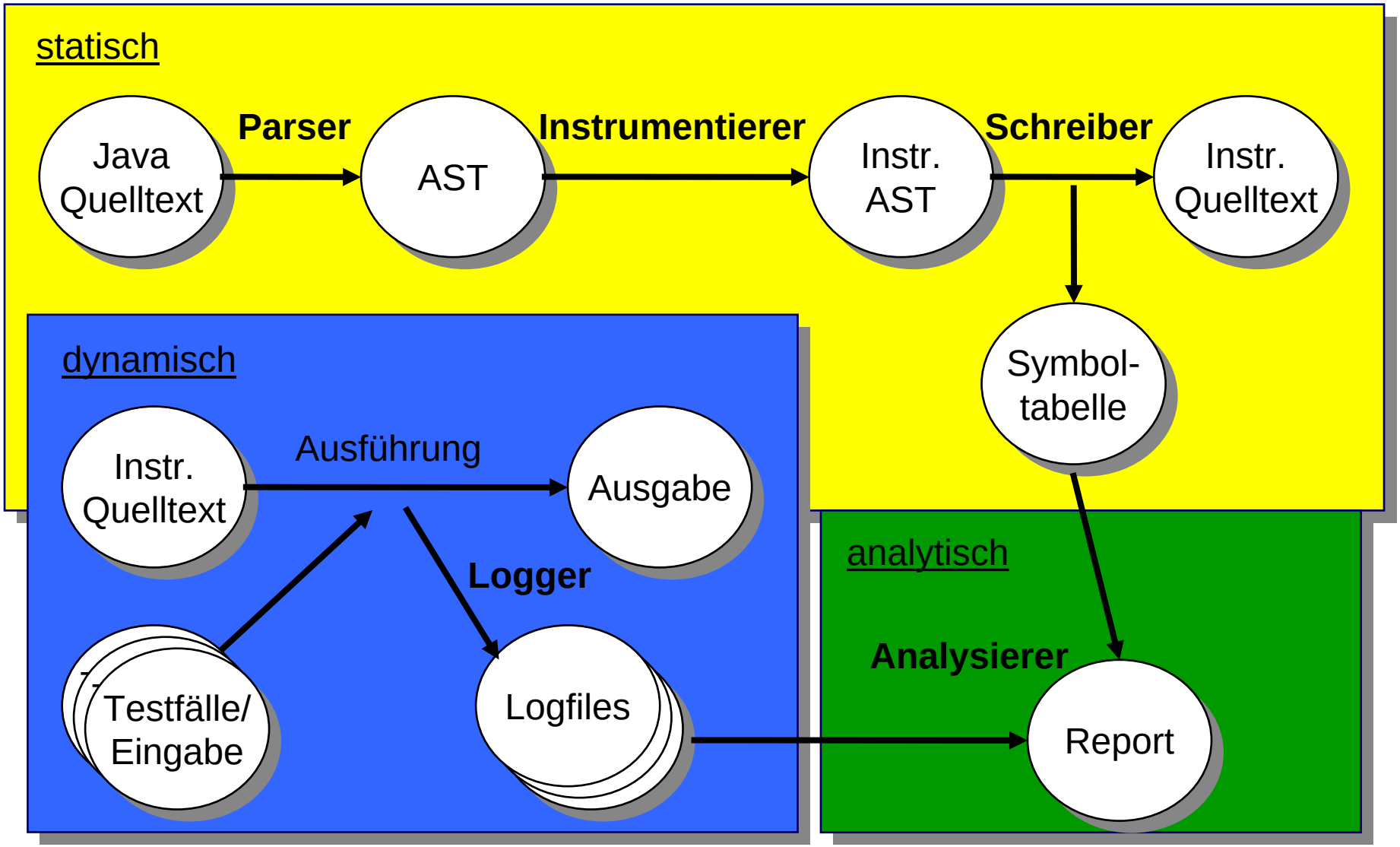


Beispiel-AST von Ausdruck (1+2-3*4/5^6)
Ergebnis: 2.999232

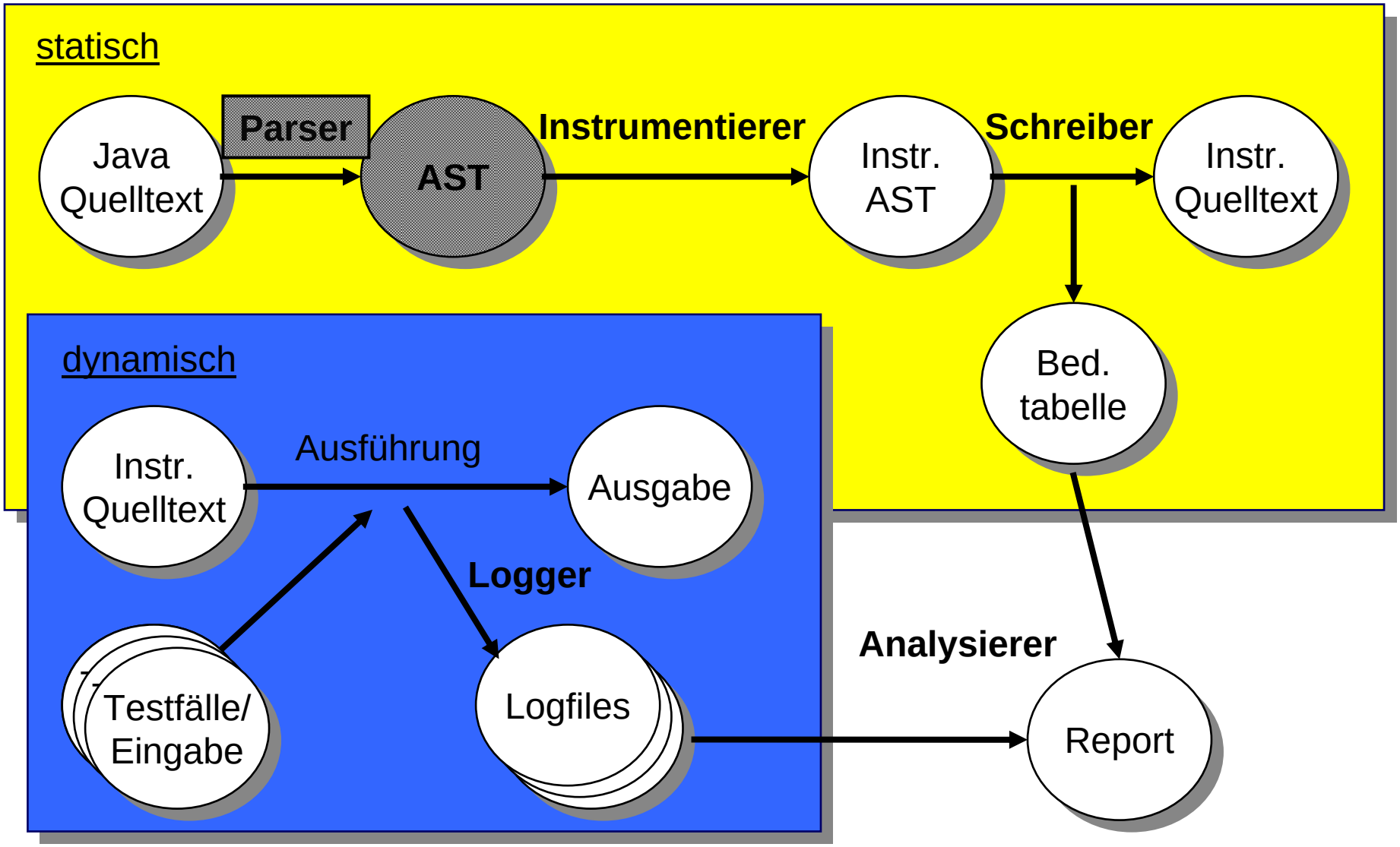
4. Das Werkzeug

Statische und dynamische
Analyse des Java-Quelltextes.

4. Das Werkzeug



4.1. Das Werkzeug

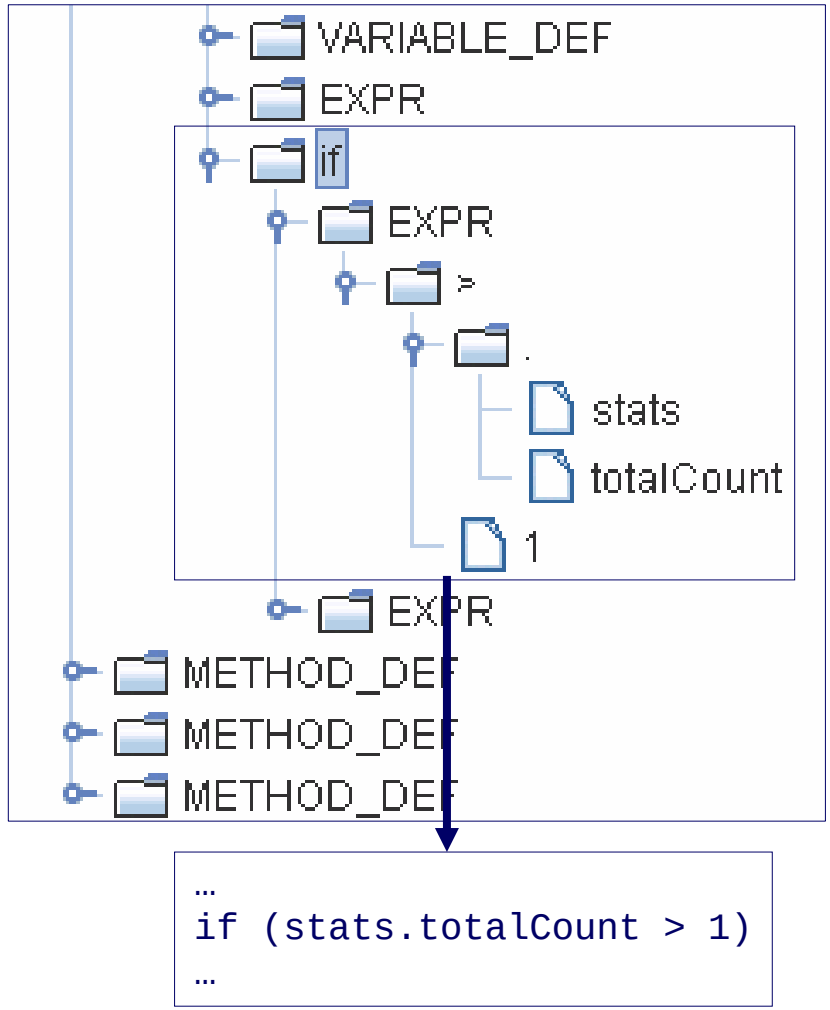


4.1. Das Werkzeug

Parser: JavaTokenizer + JavaRecognizer

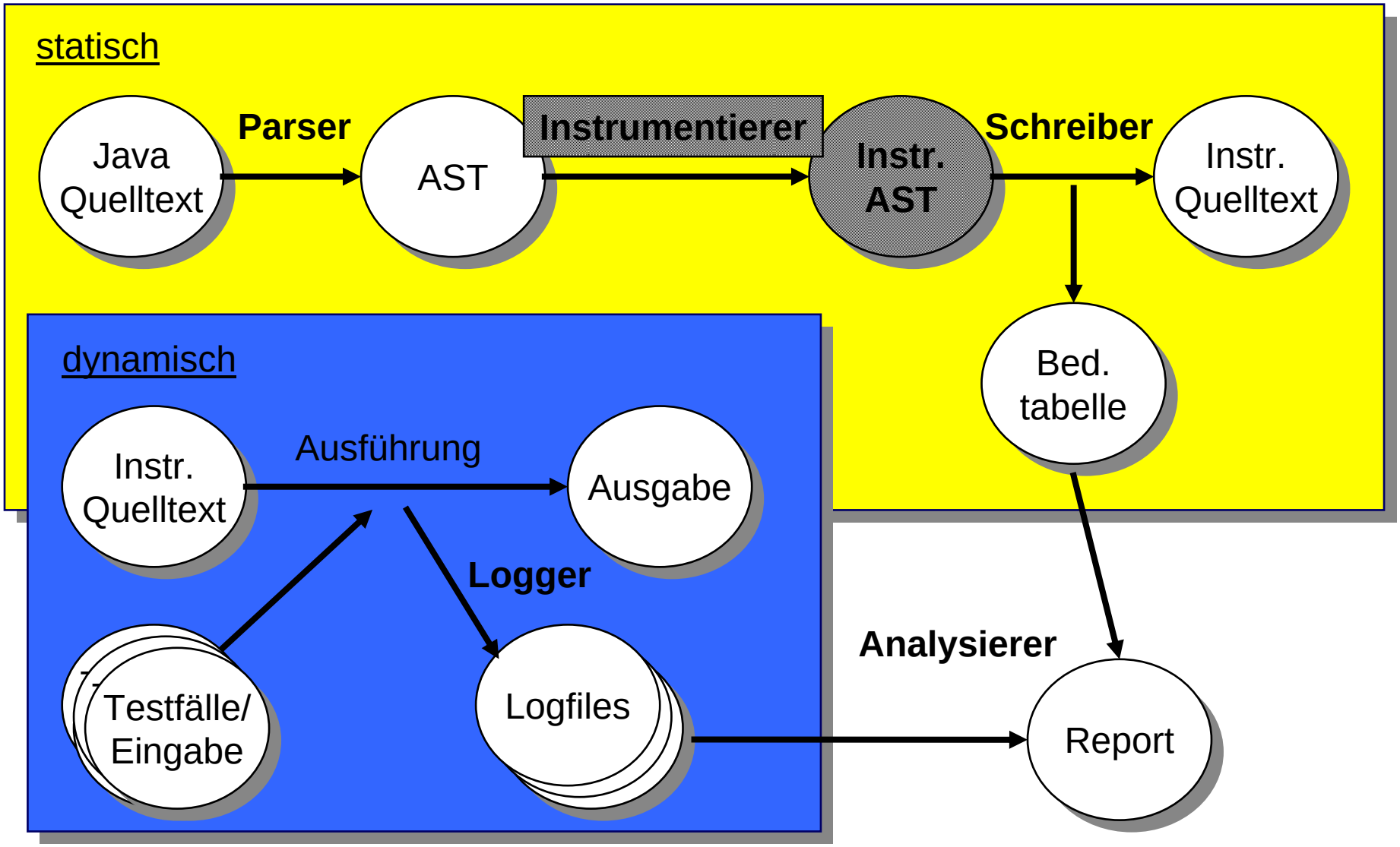
(erstellt aus Grammatik „java15.g“)

- 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 )?
…
```

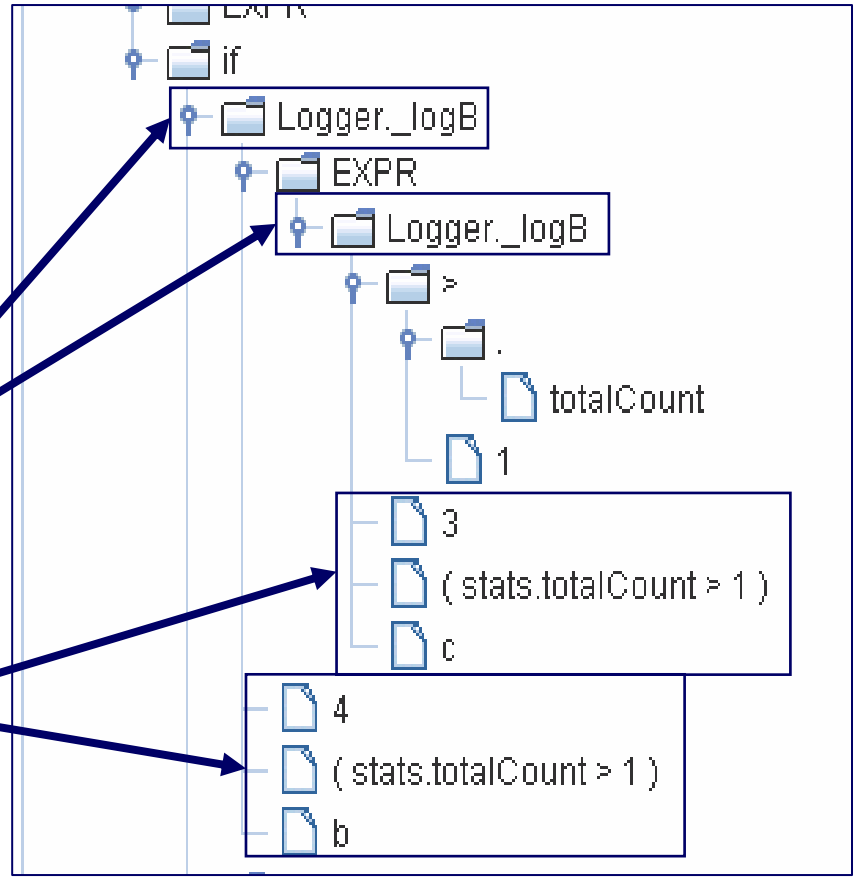
4.1. Das Werkzeug



4.1. Das Werkzeug

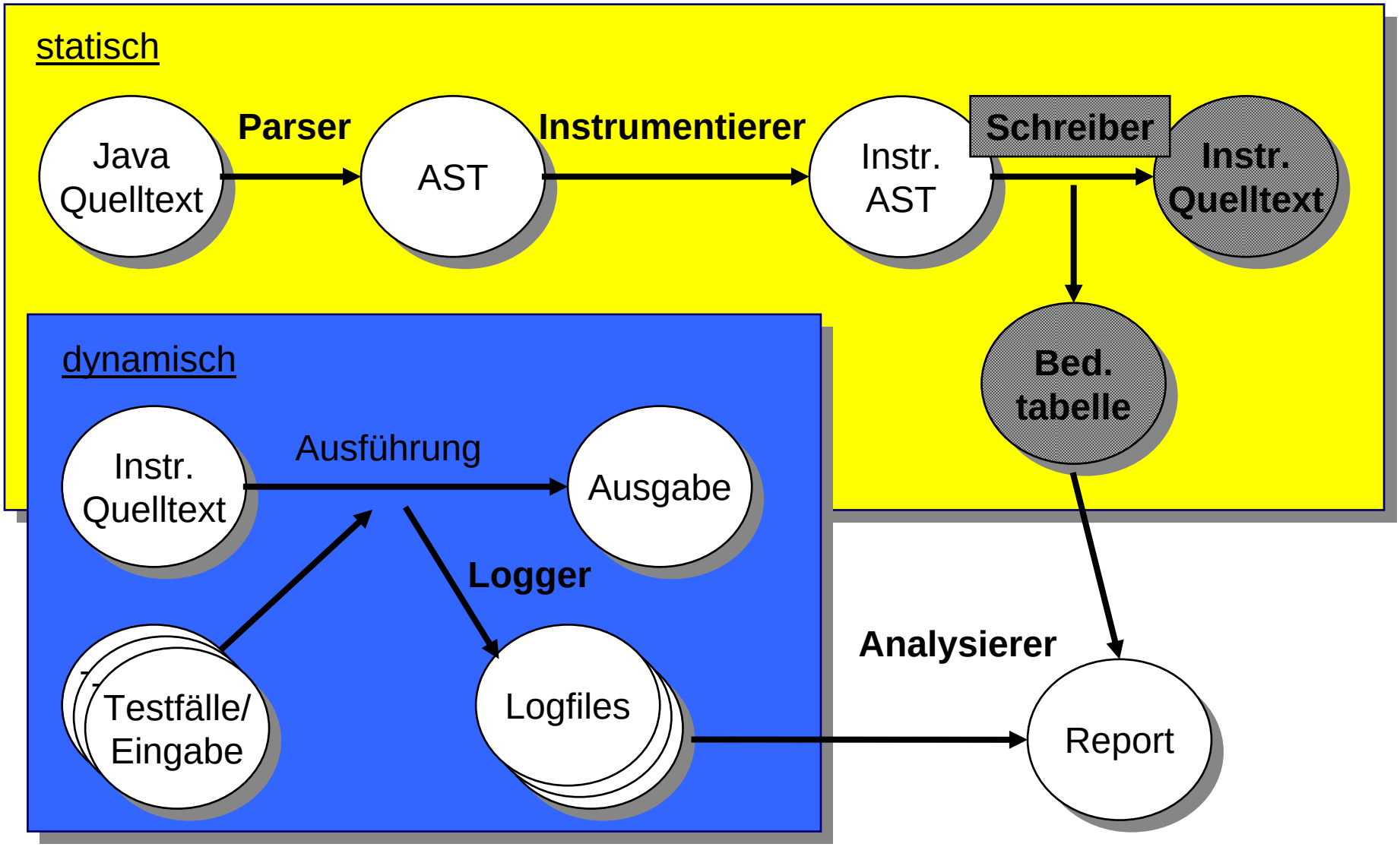
Instrumentierer:
JavaTreeParser
(erstellt aus Grammatik „java15.tree.instrumenter.g“)

- 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



4.1. Das Werkzeug

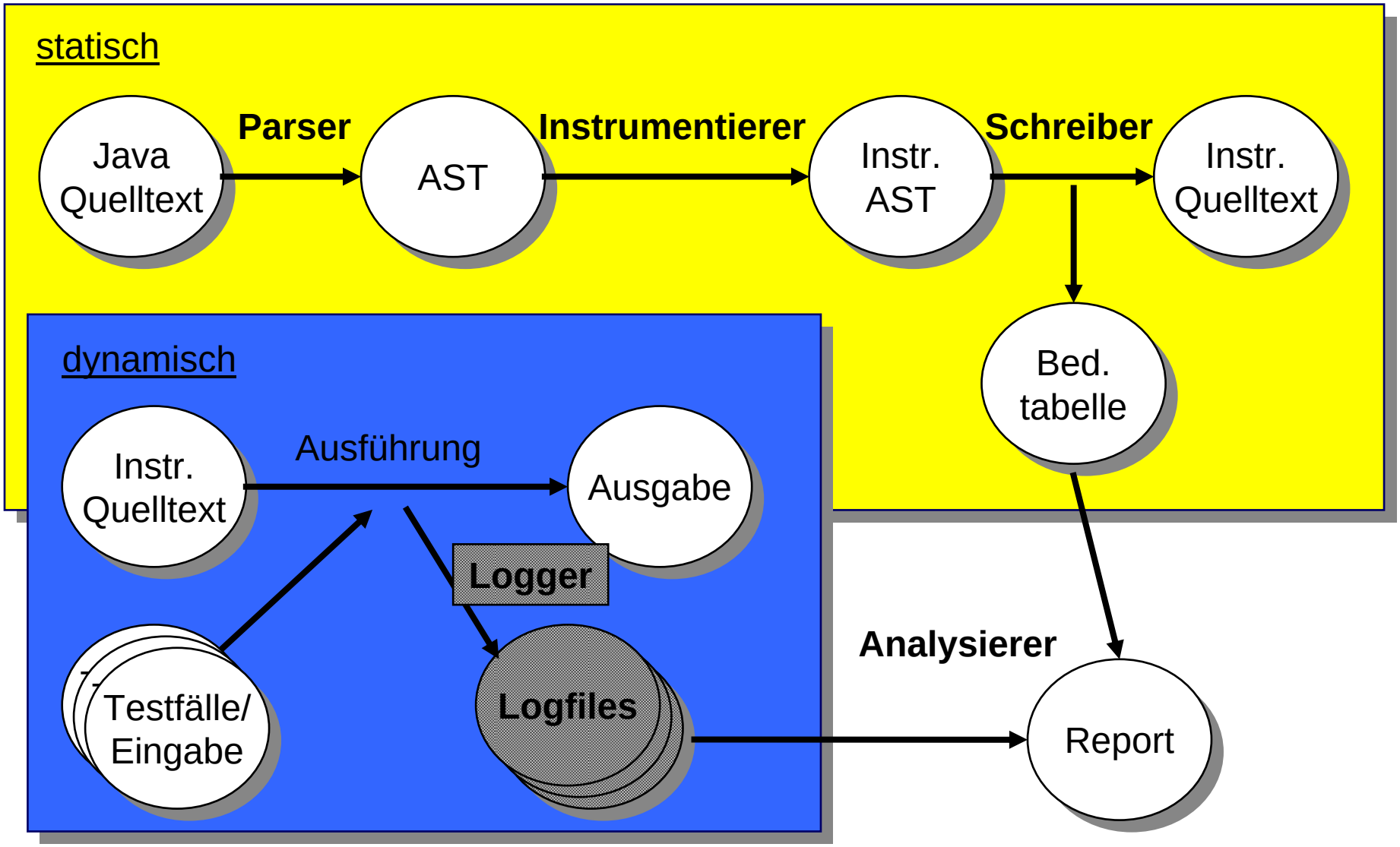
Schreiber: JavaTreeWriter (erstellt aus der Grammatik „java15.tree.writer.g“)

- Übernimmt vom Instrumentierer den instrumentierten AST und schreibt diesen als instrumentierten Java-Quelltext zurück.
- Außerdem erstellt die Schreiber-Komponente die Bedingungstabelle aus den instrumentierten AST.
- 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 („descition“/„branch“)
  - **„t“**: die Bedingung stammt vom ternären Operator
  - **„s“**: die Bedingung stammt von einem Fall („case“) der switch-case-Anweisung

4.1. Das Werkzeug

ID	Row	Type	Expression	Notes
1	8	a	(args.length > 0)	
2	8	b	(args.length > 0)	
3	24	a	(ch >= 'A')	
4	24	a	(ch <= 'Z')	
5	24	c	(ch >= 'A') && (ch <= 'Z')	
6	24	a	(ch >= 'a')	
7	24	a	(ch <= 'z')	
8	24	c	(ch >= 'a') && (ch <= 'z')	
9	24	c	(ch >= 'A') && (ch <= 'Z') || (ch >= 'a') && (ch <= 'z')	
10	24	a	(ch == ' ')	
11	24	c	(ch >= 'A') && (ch <= 'Z') || (ch >= 'a') && (ch <= 'z')|| (ch == ' ')	
12	24	b	(ch >= 'A') && (ch <= 'Z') || (ch >= 'a') && (ch <= 'z') || (ch == ' ')	
13	31	a	(ch == ' ')	
14	31	a	(ch == '\t')	
15	31	c	(ch == ' ') || (ch == '\t')	
16	31	a	(ch == '\n')	
17	31	c	(ch == ' ') || (ch == '\t') || (ch == '\n')	
18	31	b	(ch == ' ') || (ch == '\t') || (ch == '\n')	

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, den Namen des Threads sowie die Nummer des Threads (evtl. später noch mehr) in ein Logfile.
- Dabei legt er bei jeder Ausführung ein separates Logfile an.

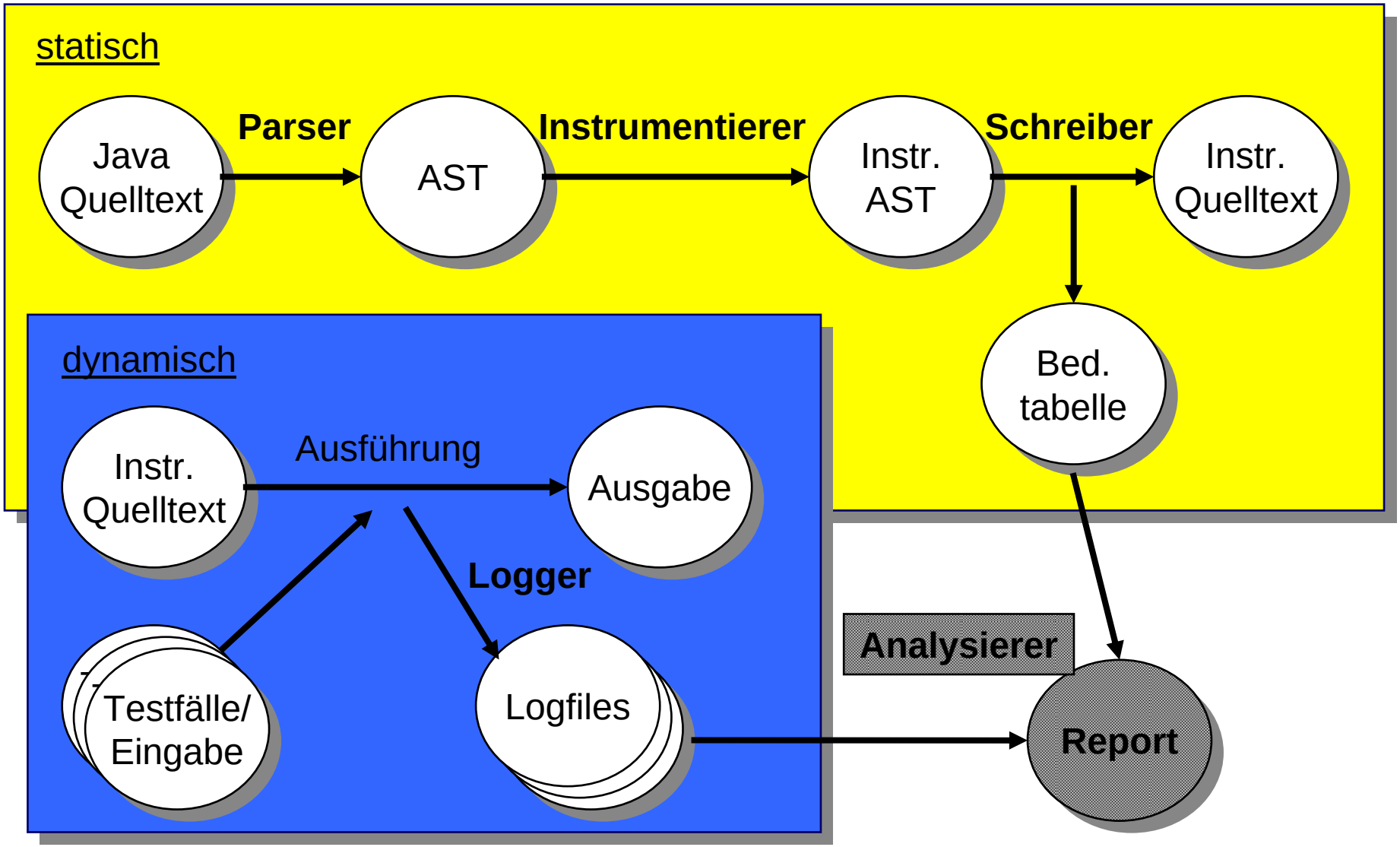
4.2. Das Werkzeug

ID	Value	ThreadN	ThreadId
1	True	main	1
2	True	main	1
3	True	main	1
4	False	main	1
5	False	main	1
6	True	main	1
7	True	main	1
8	True	main	1
9	True	main	1
11	True	main	1
12	True	main	1
13	False	main	1
14	False	main	1
15	False	main	1
16	False	main	1
17	False	main	1
18	False	main	1
3	True	main	1
4	False	main	1
5	False	main	1
6	True	main	1
7	True	main	1
8	True	main	1
9	True	main	1
11	True	main	1
12	True	main	1

13	False	main	1
14	False	main	1
15	False	main	1
16	False	main	1
17	False	main	1
18	False	main	1
3	True	main	1
4	False	main	1
5	False	main	1
6	True	main	1
7	True	main	1
8	True	main	1
9	True	main	1
11	True	main	1
12	True	main	1
13	False	main	1
14	False	main	1
15	False	main	1
16	False	main	1
17	False	main	1
18	False	main	1
3	True	main	1
4	False	main	1
5	False	main	1
6	True	main	1
7	True	main	1

8	True	main	1
9	True	main	1
11	True	main	1
12	True	main	1
13	False	main	1
14	False	main	1
15	False	main	1
16	False	main	1
17	False	main	1
18	False	main	1
3	True	main	1
4	False	main	1
5	False	main	1
6	True	main	1
7	True	main	1
8	True	main	1
9	True	main	1
11	True	main	1
12	True	main	1
13	False	main	1
14	False	main	1
15	False	main	1
16	False	main	1
17	False	main	1
18	False	main	1

4.3. Das Werkzeug



4.3. Das Werkzeug

Analyzer

- Bestimmt aus der Bedingungstabelle und den Logfiles, welche Bedingungsüberdeckungskriterien erfüllt sind und leitet daraus die Bedingungsüberdeckungsmetriken ab.
- Evtl. wird in dieser Komponente statisch die Erfüllbarkeit der Bedingungen getestet. Dazu muss noch eine passende Vorgehensweise entwickelt werden.
- Als Ausgabeformat wurde XML gewählt (in Arbeit), da es ein universelles Format ist und es sich außerdem leicht in andere Formate umwandeln lässt (mittels XSLT).

5. Stand der Arbeit

5. Stand der Arbeit

- **JavaTokenizer + JavaRecognizer:** ~100% fertig
- **JavaTreeInstrumenter:** ca. 70% fertig (externe Verweise werden noch nicht aufgelöst)
- **JavaTreeWriter:** ca. 80% fertig (es fehlt noch die Behandlung besonderer Konstrukte)
- **Logger:** ca. 70% fertig
- **Analyzer:** ca. 20% fertig (es fehlen noch 3 der 5 Bedingungsüberdeckungskriterien sowie die eventuelle statische Bestimmung der Erfüllbarkeit der Bedingungen)

5. Stand der Arbeit

„What to do?“

- Momentan kann nur eine Klasse, nämlich die Hauptklasse instrumentiert werden. Am Ende sollen alle Klassen in einem Verzeichnis instrumentiert werden können.
- Zur Zeit werden spezielle und nicht oft verwendete Konstrukte (z.B. Annotations) noch nicht geschrieben → es können nur „ausgewählte“ Java-Programme getestet werden.
- Momentan erfolgt noch keine Auflösung externer Referenzen (in Arbeit). Geplant ist mittels Reflection die Klasse dynamisch zu laden um den Typ der öffentlichen Variable bzw. Methode bestimmen zu können.
- Einbinden von Polymorphie und dynamisches Binden. Dazu müssen noch passende Konzepte ausgearbeitet werden.
- Eine funktionale und ansprechende GUI

6. Demonstration

Literaturverzeichnis

- **[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