Inhalt

[1. Rollenverteilung 2](#_Toc104408040)

[2. Architektur 3](#_Toc104408041)

[I. Schnittstellen 3](#_Toc104408042)

[II. Klassendiagram Abstrakte Syntax 4](#_Toc104408043)

[III. Klassendiagram Getyped-Abstrakte-Syntax 5](#_Toc104408044)

[IV. Java Records 6](#_Toc104408045)

[V. Java Pattern Matching 7](#_Toc104408046)

[VI. Factory-Pattern 7](#_Toc104408047)

[3. Öffnen des Projektes in einer IDE 7](#_Toc104408048)

[4. Ausführen des Compilers 8](#_Toc104408049)

[5. Parser 9](#_Toc104408050)

[6. Type-Checker 9](#_Toc104408051)

[7. Bytecode-Generator 9](#_Toc104408052)

[8. Testing 10](#_Toc104408053)

[VII. Parser Tests 10](#_Toc104408054)

[VIII. Typ-Checker Tests 10](#_Toc104408055)

[IX. Bytecode-Generator-Tests 10](#_Toc104408056)

[X. Integration-Tests 10](#_Toc104408057)

[XI. Functional-Tests 10](#_Toc104408058)

# Rollenverteilung

* Projektleiter, Dokumentation und Test:

Name: Jonathan Schwab

GitHub-Alias: JDev2001

* Parser

Name: Felix Wochele

GitHub-Alias: FelixWochele

* Type-Check: Jan Perthel

Name: Jan Perthel

GitHub-Alias: jan510

* ASM

Name: Reinhold Jooß

GitHub-Alias: SuuyaaYeet

Repository: <https://github.com/JDev2001/JavaCompiler_TINF2020>

# Architektur

In diesem Abschnitt werden die Schnittstellen, Klassenstrukturen und speziellen verwendeten Java-Features erklärt.

## Schnittstellen

* Parser

getSyntaxTree(charStream:input):AbstractSyntax

* Type-Checker

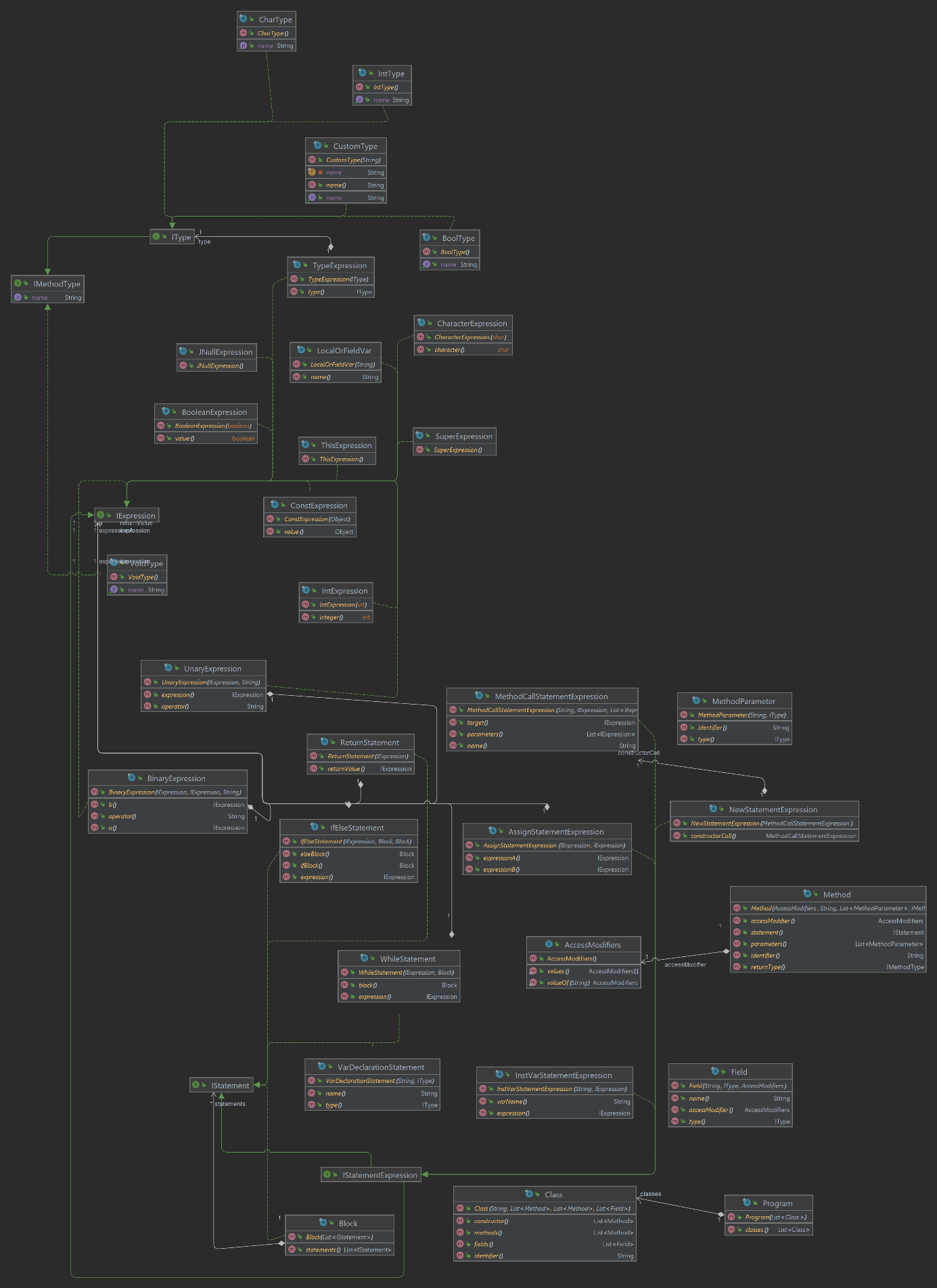
semantikCheck(input:AbstractSyntax):TypedAbstractSyntax

* Byte-Code-Generator:

generateByteCode(input:TypedAbstractSyntax):Hashmap<String, byte[]>

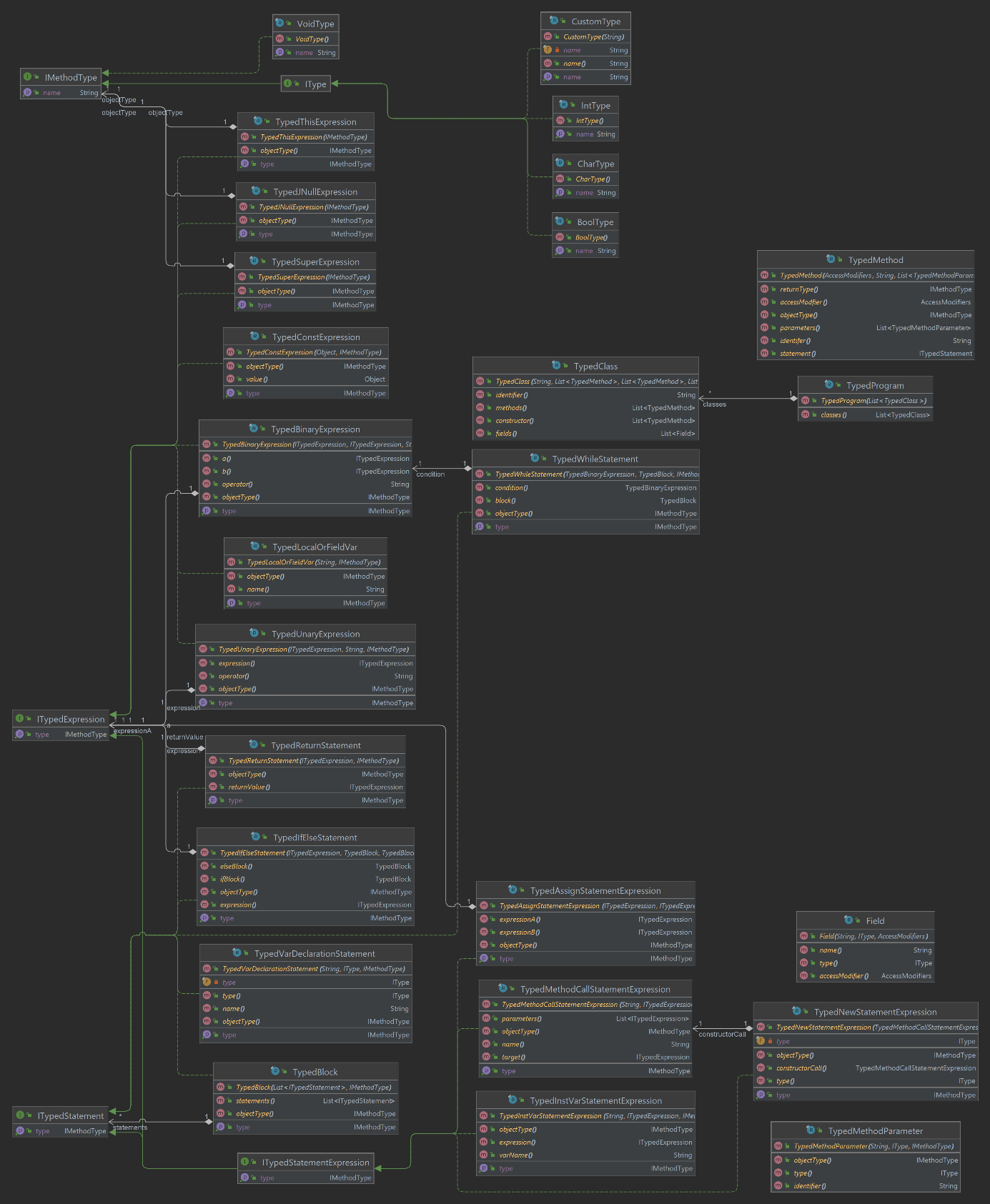
## Klassendiagram Abstrakte Syntax

Die abstrakte Syntax wird vom Parser erzeugt.



## Klassendiagram Getyped-Abstrakte-Syntax

Die getyped Abstrakte Syntax ist ähnlich aufgebaut wie die abstrakte Syntax. Jedoch werden alle Expressions und Statements um einen Typ erweitert.



## Java Records

Seit Version 17 werden von Java Record-Typen angeboten. Records dienen zur Speicherung von unveränderlichen Daten. Der Schreibaufwand von Code ist im Vergleich zu Klassen deutlich geringer. Des Weiteren besitzen Records Standardimplementierungen für die Methoden „hashCode“, „equals“ und „toString“. Diese Methoden können bei Bedarf aber zusätzlich implementiert werden. Records unterstützen keine Vererbung, erlauben es jedoch Schnittstellen zu implementieren. Ein Record kann wie Folgt definiert werden:

record Data (int Id, String name) {}

Zum Entwickeln eines JavaCompilers bieten sich diese Recordtypen an. Der Parser generiert eine abstrakte Syntax, die nach Generierung unveränderlich ist. Dasselbe gilt für den Type-Checker. Darüber hinaus müssen einige Tests geschrieben werden, welche die abstrakte Syntax und die getypte abstrakte Syntax nachbilden und mit den Ergebnissen des Parsers bzw. Type-Checks abgleichen. Werden die Record-Typen für diese Datenstruktur verwendet, muss anders als bei Klassen die Methode „equals“ nicht von Hand implementiert werden, um Assertions mit JUnit durchzuführen zu können.

## Java Pattern Matching

Seit Java 17 wird eine einfache Form von Pattern Matching angeboten. In einem Switch-Case-Statement wird ein instanceof-Check mit direktem anschließenden Typecast durchgeführt. So kann ein Basistyp auf verschiedene Subtypen geprüft werden.

void Check(Object o)

{

    switch (o) {

        case String s:

            // {...}

            break;

        case Integer i:

            //  {...}

            break;

        case Boolean i:

            //  {...}

            break;

    }

}

Durch die Verwendung von Pattern Matching im Type-Check und Bytecode-Generator kann auf das Visitor-Pattern verzichtet werden.

## Factory-Pattern

Um Abhängigkeiten zu entkoppeln ohne aufwendige Frameworklösungen wie Dependency Injection zu verwenden, wird das Factory Pattern genutzt. Eine Factory Klasse bietet einen IParser, einen ISemanticChecker und einen ICodeGenerator an. Die Factory instanziiert diese Klassen, sobald diese angefordert werden. Zurückgegeben werden nur Referenzen auf die Schnittstellen und nicht die Implementierungen. So wird Austauschbarkeit der einzelnen Komponenten gewährleistet.

# Öffnen des Projektes in einer IDE

Das erforderliche Sprachlevel zum Kompilieren des Java-Compilers ist die Preview-Version Java 17 oder Java 18 mit Unterstützung für Pattern-Matching for switch. Wird die neuste Version von IntelliJ verwendet, kann diese Einstellung in dem Fesnter „Project Structure“ vorgenommen werden: 

Bei anderen IDEs muss zustäzlich bei Gradle oder Maven das JVM Flag „--enable-preview“ gesetzt werden.

# Ausführen des Compilers

Der entwickelte Compiler kann mit „java -jar compiler.jar <<arg>>“ aufgerufen werden. Arg ist hierbei die Java Datei, die kompiliert werden soll. Der Compiler erzeugt anschließend eine .class Datei, welche den kompilierten Java-Bytecode enthält.

# Parser

Der Parser übersetzt Java-Quellcode in eine abstrakte Syntax. Hierfür wird der Parser Generator Antler verwendet. In einer separaten Datei (.g4-Datei) wird die Grammatik zum Parsen des Java-Quellcodes definiert. Aus der Grammatik werden Klassen generiert. Zur Übersetzung in die abstrakte Syntax werden verschiedene Adapter geschrieben, welche die abstrakte Syntax erzeugen und ggf. noch kleine Anpassungen vornehmen. So wird nach und nach ein Syntaxbaum aufgebaut.

Schnittstelle: getSyntaxTree(charStream:input):AbstractSyntax

# Type-Checker

Der Typchecker übersetzt die abstrakte Syntax in eine getyped abstrakte Syntax. Da es sich bei Java um eine statisch typisierte Programmiersprache handelt, müssen Typprüfungen zur Kompilierungszeit vorgenommen werden. Diese Überprüfungen sind nicht mit dem Parser-Generator möglich und müssen in einem separaten Schritt vorgenommen werden. Der Typ-Checker prüft mit Regeln aus der Vorlesung, ob der Code konsistent typisiert ist und leitet die Typen her.

Schnittstelle: semantikCheck(input:AbstractSyntax):TypedAbstractSyntax

# Bytecode-Generator

Der Bytecode-Generator übersetzt die typisierte abstrakte Syntax in Bytecode. Hierfür wird die Java-Bibliothek ASM verwendet. Der Abstrakte Syntaxbaum wird durchlaufen und mittels dem ASM ClassWriter daraus Bytecode generiert. Zurückgegeben wird eine Hashmap, wobei ein Eintrag aus dem Klassennamen und dem zugehörigen Bytecode besteht.

Schnittstelle: generateByteCode(input:TypedAbstractSyntax):Hashmap<String, byte[]>

# Testing

## Parser Tests

Ein Text, der Java-Quellcode enthält wird durch den Parser in eine abstrakte Syntax übersetzt. Das Ergebnis des Parsers wird anschließend mit einer händisch nachgebildeten Datenstruktur der erwarteten abstrakten Syntax auf inhaltliche Gleichheit überprüft. Getestet werden alle denkbaren Szenarios.

## Typ-Checker Tests

Eine händisch aufgebaute abstrakte Syntax wird dem Typ-Checker als Eingabe übergeben. Das Ergebnis ist eine getypte abstrakte Syntax. Diese getypte Abstrakte syntax wird mit einer händisch nachgebildeten Datenstruktur der erwarteten getyped abstrakten Syntax auf inhaltliche Gleichheit überprüft. Getestet werden alle denkbaren Szenarios.

## Bytecode-Generator-Tests

Über Java Reflection kann der generierte Bytecode getestet werden. Hierzu wird dieser über den ReflectionLoader eingelesen. Anschließend können die enthalten Klassen, deren Felder, Methoden und Konstruktoren zur Laufzeit untersucht werden. Außerdem können Methoden aufgerufen und das erwartete Ergebnis abgerufen werden. So kann der erwartete strukturelle einer Klasse geprüft werden. Außerdem ist die Auswertung von Methodenaufrufen, Statements und Expressions möglich.

## Integration-Tests

In den Integration-Tests werden der Parser, Type-Checker und ByteCode-Generator gemeinsam getestet. Die Eingabe ist somit ein String, der Java-Quellcode enthält und die Ausgabe ist Java-Bytecode. Über Reflection können dann Methoden im generierten Bytecode aufgerufen werden. Anschließend wird das Ergebnis validiert.

## Functional-Tests

In einem Functional-Test wird der gesamte Compiler als Java Anwendung überprüft. Hierzu wird die Jar-Datei mit einem Kommandozeilenargument (Pfad zu einer Java-Datei, die kompiliert werden soll) aufgerufen. Die Anwendung kompiliert anschließend ein .class-File aus dem Quellcode. Da von dem entwickelten Compiler keine statischen Methoden unterstützt werden (somit auch kein public static void main(String args[])), müssen die generierten Klassen von einer weiteren Java-Datei, die über JavaC kompiliert wird, eingebunden werden. Diese Hilfsdatei enthält den Einstiegspunkt und testet die kompilierten Klassen auf ihre Funktionalität.