BERUFSAKADEMIE
STUTTGART

University of Cooperative Education

Konsolidierung und Vorbereitung des Java-Typinferenz Projekts

durch die Erstellung einer Testsuite für

ein Open–Source-Projekt

Studienarbeit

Eine Projekt-Dokumentation eingereicht für den Abschluss zum

Diplom-Informatiker (Berufsakademie)

im Studiengang Angewandte Informatik
an der Berufsakademie Stuttgart

von

Thorsten Hake und Christian Stresing

Juni 2008

Zeitraum	5. und 6. Semester
Kurs	TIT05AIA
Firma	IBM Deutschland GmbH Stuttgart
Betreuer	Timo Holzherr nero AG
Projektverantwortlicher	Prof. Dr. Plümicke Berufsakademie Horb

Selbständigkeitserklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Thema

Konsolidierung und Vorbereitung des Java-Typinferenz Projekts durch die Erstellung einer Testsuite für ein Open–Source-Projekt

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Ort: _____

Datum: _____

Unterschriften: _____

Zusammenfassung

Das Projekt des Java-Typinferenz-Compilers ist über die Jahre stark gewachsen. Der verhältnismäßig aufwändige Algorithmus von Prof. Dr. Martin Plümicke zur Berechnung der Variablentypen in der Programmiersprache Java ist durch eine große Anzahl von Studentenprojekten in einem Compiler implementiert worden.

Um den Status des Projekts zu ermitteln und der Weiterentwicklung Unterstützung zu bieten, ist es höchste Zeit Softwaretests durchzuführen. Im Rahmen dieser Arbeit wurde ein Testframework mit samt umfangreicher Funktionstestssammlung erstellt. Diese Tests verifizieren die Funktionsweise bzw. decken einige noch enthaltene Fehler des Compilers auf. Bestandteil der Arbeit ist zudem die Feststellung und Analyse der Fehler.

Der erfasste Status gibt Ausschluss über benötigte Verbesserungen und die geplante Überführung in ein Open Source Projekt.

Abstract

The Java-Type-Inference-Compiler project has grown during its years of existence. Many student projects have since helped implementing the complex type inference algorithm developed by Prof. Dr. Martin Plümicke leaving a nebulous touch on the project.

Thus, it is about time to introduce some extensive software testing to analyze the project's state and allow for further development. This paper deals with the ideas, introduction and implementation of a test framework whose tests focus on verifying the correct type inference and discover bugs. These bugs are documented in this paper as well.

The paper also aims at showing what is currently implemented and what needs to be accomplished in the future in order to address the needs of an open source project.

Danksagung

An dieser Stelle möchten wir uns bei Prof. Dr. Martin Plümicke und Timo Holzherr bedanken, die uns bei Fragestellungen bezüglich entdeckter Fehler beim Testen des Typinferenzalgorithmus unterstützt und Antwort gegeben haben. Auch wenn aufgrund der größeren Distanz zwischen den Aufenthaltsorten der Kommunikation oftmals Schwierigkeiten im Weg standen, konnten wir das Projekt dennoch erfolgreich abschließen.

Inhaltsverzeichnis

1 Einführung

1.1 Typinferenz

Der Begriff Typinferenz bezeichnet das Bestimmen eines Typs, der noch nicht bestimmt ist durch die Analyse von beeinflussende Faktoren. Der in [PLU05] definierte Typinferenz Algorithmus kann dazu benutzt werden in der Programmiersprache JAVA unbekannte Typen zu bestimmen. Hierdurch muss bei der Programmierung nicht jede Variable typisiert werden, da der Typ bei einigen Variablen vom Compiler selber inferiert werden kann. Wo es nicht eindeutig bestimmbar ist, ermittelt der Compiler eine Auswahl an möglichen Typen.

1.2 Projektgeschichte

Der in dieser Studienarbeit im Mittelpunkt stehende Compiler wurde im Rahmen der Vorlesungen „Informatik 4“ und „Software Engineering“ vom Jahrgang TIT2000 an der Berufsakademie Horb entwickelt.

Im Rahmen von darauf aufsetzenden Studienarbeiten wurde dann mit den Studienarbeiten [REI05] und [HAA04] der Compiler um generische Typen erweitert. Der in [PLU05] beschriebene Typinferenzalgorithmus wurde dann in der Studienarbeit [OTT04] und in [BAE05] im Compiler integriert. Gebundene generische Typen, Wildcards, Interfaces und Packages wurden in [HOL06] und [LUE07] dem Algorithmus hinzugefügt. Die Studienarbeiten [BUR07] und [SCH07] hatten zudem das Ziel den Compiler Java 5.0 kompatibel zu machen.

Somit ist der Compiler über die Jahre stark gewachsen. Dabei wurde bisher keine ausreichende bzw. vor allem beständige Verifikation der Funktion während der Entwicklung getätigt.

1.3 Aufgabenstellung

Die Aufgabenstellung hat sich leider im Laufe der Arbeit geändert. Aufgrund eines Missverständnisses in der Absprache der beteiligten Personen wurden anfangs mit großem Engagement Unit-Tests entwickelt, die eine intensive Einarbeitung in den

bestehenden Code und die Grundlagen der Typinferenz erforderten. Letztlich war dieser Teil jedoch nicht Bestandteil der danach folgenden Arbeit.

Aufgabenstellung dieser Arbeit ist es, eine Ansammlung von Funktionstests zu implementieren um die Funktionalität des Compilers bzw. des Typinferenzalgorithmus zu überprüfen. Darüber hinaus sind umfangreichere Testfälle auch für eine Weiterentwicklung des Algorithmus von Nutzen.

Außerdem sollen erste Überlegungen angestellt werden um das Projekt anschließend in ein Open-Source-Projekt umzuwandeln.

2 Grundlagen

2.1 Testen

Beim Testen allgemein geht es um die Feststellung und um die Sicherung bzw. um die Methode zur Feststellung und Sicherung der Qualität eines bestimmten Produkts. Da es sich in unserem Fall um eine Software handelt, beziehen sich die weiteren Erklärungen demnach auf Software-Tests.

Software-Tests gibt es in sehr vielen verschiedenen Ausprägungen. Jede dieser Ausprägungen hat einen eigenen ganz bestimmten Fokus in Bezug auf die Qualitätssicherung von Software. Im folgenden sind zusammenfassend ein paar Kategorien gelistet:

Modul-/Komponententests konzentrieren auf eine einzelne Einheit, eine Unit. Modultests sind die zentrale Testaufgabe der Programmierer

Integrationstest/Interaktionstest sind Tests, die sich auf das Zusammenspiel mehrerer zuvor einzeln getesteter Komponenten beziehen. Eine klare Trennung zwischen Modultests und Integrationstests ist allerdings in der Regel bei objektorientierter Programmierung sehr schwer.

Performanz- und Lasttests dienen der Sicherung zur Erfüllung von meist nicht funktionalen Anforderungen wie Antwortzeiten und erwartete Nutzerzahlen.

Funktionstests/Akzeptanztests richten sich an den späteren Anwender und stellen die Anforderungen an den Funktionsumfang sicher. Hier erstellt, im Gegensatz zu den vorhergehenden Kategorien, der Auftraggeber meist selbst die Testszenarien nach den vereinbarten, von der Software zu lösenden Aufgaben.

Das Testen von Software ist einer der wichtigsten Bestandteile im Software-Entwicklungsprozess. Alle Softwareentwickler wissen um die Bedeutung und trotzdem: nur selten wird ausreichend getestet. Die Ursachen hierfür sind vielfältig. Dabei handelt es sich seltener um valide Gründe, als viel mehr um persönliche Ausreden. Nach [LIN02] gehören zu den meisten Ausreden der Softwareentwickler:

1. Zeitmangel

2. Testen ist uninteressant und langweilig
3. Glaube an die Korrektheit des eigenen Codes
4. Testen ist Aufgabe der Testabteilung

Dabei lassen sich meist all diese Punkte entkräften.

An Zeit mangelt es einem Softwareentwickler bei heutigem Feature-Druck und den sehr kurzen Release-Zyklen in der Regel generell. Nur beim Testen die Zeit wegzunehmen bringt den Entwickler in einen Teufelskreis. Denn je weniger Tests, desto mehr Fehler schleichen sich in den Code ein und desto mehr Fehler auftreten, desto mehr Zeit muss in die Fehlerbehebung gesteckt werden. Allgemein kann man sogar soweit gehen und sagen, dass Testen Zeit spart. Denn je mehr man testet, desto zeitiger lassen sich Fehler finden.

Testcode zu schreiben, und damit das Aufdecken von Fehlern, und das durchdachte Testen verlangen vom Ersteller ebenso viel wie das Schreiben von Programmcode an sich.

Auch wenn man sich sicher ist, jedes Detail seines großen Projekts zu kennen und auch fest von dessen Korrektheit überzeugt ist, so kommen einem spätestens beim Abändern eines Details in einer ganz bestimmten Klasse Zweifel, welche Auswirkungen diese Änderung mit sich bringt. An dieser Stelle haben bewährte Testfälle den Vorteil, dass sie den Status festhalten und gegen eine Änderung am Code immun sind, also weiterhin die Richtigkeit aller Szenarien sicherstellen.

Ebenso wie die ersten 3 Punkte lässt sich auch der letzte widerlegen. Die Testabteilung oder das Team, das mit dem Testen beauftragt ist, hat die Aufgabe die Korrektheit und die Vollständigkeit des Gesamtprodukts zu verifizieren. Gemäß dem von [LIN02] erwähnten „Antidecomposition-Axiom“ gilt aber: „Das Testen eines zusammengesetzten Systems reicht nicht aus, um die Fehler seiner Komponenten aufzudecken“. Demnach ist der Programmierer selbst dafür verantwortlich, dass seine Komponenten auch in isolierter Umgebung getestet werden.

2.2 Unit-Test vs. Funktionstest

Die Unterscheidung zwischen einem Unit-Test oder einem Funktionstest ist allgemein die wohl verbreitetste Methode zur Klassifikation von Software-Tests.

Unit-Tests dienen in erster Linie der Absicherung des Entwicklers gegenüber seinem Code. So kann hier auf niedrigster Stufe die Fehlerentdeckung sehr stark vereinfacht

werden und die Vermeidung von Fehlern, die sich gar nicht bzw. erst sehr spät im Entwicklungsprozess herausstellen erhöht werden. Der Entwickler ist selbst verantwortlich für die Erstellung des Testcodes. Idealerweise sollte die Erstellung gleichzeitig mit der des Programmcodes geschehen. Das kann sogar soweit gehen, dass, aus dem XP-Ansatz hervorgehend, der Testcode vor dem Programmcode geschrieben wird und somit die Anforderungen an das einzelne Modul bzw. die zu testende Klasse (Class-Under-Test: CUT) definiert. Dieses Konzept wird als Test-First-Ansatz bezeichnet, im Gegensatz zu *Design-First-Programmierung*. Das Schreiben von Testcode nach der Erstellung des Programmcodes ist zumeist sehr schwierig, da in diesem Fall bei der Entwicklung selten Rücksicht auf die Testbarkeit genommen wird. Testbarkeit geht aber zumeist einher mit der Wartbarkeit von Code. Eine vom Unit-Test verlangte Austauschbarkeit von Komponenten, ist ein Aspekt, der in der normalen Programmierung untergehen kann. Beispielsweise werden andere Module oftmals fest im Programmcode verwendet. Unit-Tests bedingen aber das ausschließliche Testen eines einzigen Moduls, wozu verwendete, noch nicht getestete Module, durch Dummy-Module ersetzt werden müssen („Stubbing“). Diese Forderung hilft, den eigenen Programmcode modularer aufzubauen und so die Austauschbarkeit von referenzierten Klassen zu ermöglichen. Ein Aspekt, der die Wartbarkeit stark erhöht. Wichtig bei Unit-Tests ist vor allem der White-Box-Ansatz. Da jedem Entwickler die Implementierung seiner zu testenden Module bekannt ist, ist es hier ein Leichtes implementierungsabhängig zu testen. Auf diese Weise können Grenzfälle getestet werden und die Testabdeckung extrem erhöht werden.

Entgegen den Unit-Tests werden in der professionellen Softwareentwicklung die Funktionstests in der Regel von eigenen Test-Abteilungen oder aber am Ende eines Entwicklungsprozesses von den Entwicklern zur Verifikation der Anforderungen des Kunden an die Software durchgeführt. Darüber hinaus kann ein Funktionstest aber auch als Messinstrument für den Status des Projekts eingesetzt werden. Man kann somit jederzeit erkennen wie weit ein Projekt gekommen ist, was funktioniert und was noch nicht. Dabei wird in aller Regel der Backbox-Ansatz, also das reine Input-Output Testen, benutzt.

Beiden Testformen ist der Ruf nach einer Automatisierung gemein. So möchte man nicht jedes Mal die Testfälle einzeln von Hand ablaufen lassen und deren Ergebnis überprüfen.

Gerade bei Veränderungen am Code, etwa Bugfixes oder Funktionserweiterungen, ist es wichtig sicherzustellen, dass der bisherige Status erhalten bleibt. Das heißt, dass eine Verbesserung an einer Stelle, nicht durch eine unbedachte Abhängigkeit an anderer Stelle einen Fehler hervorruft. Automatisierte Tests, sowohl im Unitkontext, als auch im Funktionskontext, sind hierbei von großem Vorteil. Sie helfen den Entwicklern innerhalb kürzester Zeit festzustellen, ob neue Probleme durch eine Veränderung entstanden sind. Auf diese Weise sind auch Codeoptimierungen leicht möglich. So kann jederzeit überprüft werden, ob die Korrektheit weiterhin gegeben ist.

Während beim Unit-Test in der Regel einfache Parameter bzw. lediglich der Rückgabewert einer einzelnen Methode getestet werden/wird, reichen beim Funktionstest hier nicht nur einfache Asserts. So handelt es sich in der Regel um ein umfassendes, zumeist auch komplexes Gesamtergebnis. Das erfordert in aller Regel die Entwicklung eines kleinen Frameworks, welches dem Testschreiber zumeist den Verifizierungsprozess seiner Erwartungen an den Testfall abnimmt und eventuell Utility-Funktionen - wie immer wiederkehrende Testvoraussetzungen - abnimmt. Ein solches Framework kann dennoch auf ein bestehendes Testframework zurückgreifen, welches einem die automatisierte Ausführung und das Reporting der Testfälle abnehmen könnte. Im nächsten Abschnitt soll kurz ein solches Basisframework vorgestellt werden.

2.3 JUnit

JUnit ist ein Test-Framework, das das automatisierte Ablaufen von Tests ermöglicht. Es ist mittlerweile zum „de facto“ Standard der Java-Test-Werkzeuge geworden und das Vorbild vieler portierter Frameworks für andere Sprachen (z. B. CppUnit für C++, PHPUnit für PHP, PyUnit für Python,...). JUnit und die Erfahrungen, die damit gemacht wurden, sind zu einer Grundlage für die Entwicklung der „Test-Driven-Development“-Strategie geworden, dessen neuste Erkenntnis der Test-First-Ansatz, wie im letzten Kapitel kurz erwähnt, darstellt. Auch Alex Garrett [GAR05] bescheinigt dem JUnit-Framework die starke Vereinfachung des Testens: „Testing before JUnit came along wasn't impossible, but it was difficult. In fact, it was so difficult that it often didn't get done“. JUnit wurde von Kent Beck und Erich Gamma entwickelt und wird seit einigen Jahren als Open-Source-Projekt von der JUnit.org – Community weiterentwickelt. Aktuell ist die Version 4.4 (18. Juli 2007) erhältlich.

2.3.1 Merkmale

Die wesentlichen Merkmale des JUnit-Frameworks lassen sich wie folgt zusammenfassen. JUnit...

- bietet ein Template für Tests mit Test-Fixtures (setup, teardown) und Testausführung
- erlaubt das hierarchische Gliedern von Tests und Schachteln in Testsuites
- erlaubt das automatische und einfache Ausführen von Tests
- bietet verschiedene, sogenannte „Test Runners“ für die Darstellung der Testergebnisse.

Obgleich zahlreicher erhältlicher Erweiterungen, ist das JUnit-Basisframework verhältnismäßig einfach gehalten. Ein einfacher Testfall, der zwei verschiedene Szenarien eines Moduls „EinModul“ testet, könnte etwa so aussehen:

```
public class TestEinModul extends TestCase{
      EinModul test = null;
      public void setUp() {
            this.test= new EinModul();
      {
      public void testScenario1() {
            //run test
            //assertTrue
      }
      public void testScenario2() {
            //run test
            //assertTrue
      }
      public void tearDown() {
            //do cleanup
      }
```

```
}
```

Hierbei wird nun vom Framework jeweils die `setUp()` - Methode, eine `testMethode()` und dann die `tearDown()`-Methode aufgerufen. Die Darstellung der Ergebnisse kann auf unterschiedliche Weise geschehen. So gibt es die Möglichkeit einer einfachen Konsolenausgabe, die lediglich eine Zusammenfassung über die Anzahl erfolgreicher und fehlgeschlagener Tests gibt (`junit.textui.TestRunner`) oder auch grafisch aufbereitete Reporting-Schnittstellen wie den `junit.swingui.TestRunner`. Letztere bieten übersichtlichere und farblich gekennzeichnete Darstellungen mit Debug-Ausgaben über die Ursache und Stelle des Fehlers bei einem fehlerhaften Test.

Obwohl JUnit viele Stärken hat, die auch zumeist in seiner relativ einfachen Struktur begründet sind, gibt es auch Aspekte, die JUnit nicht adressiert. Nach [GAR05] sind das:

- Automatische Generierung von Tests für ein bestimmtes zu testendes Modul
- Messinstrumente zur Testabdeckung
- Aufzeigen von schwachen Testfällen.

An dieser Stelle muss der Entwickler bzw. der Tester eingreifen und dies entweder händisch oder unter Zuhilfenahme weiterer Werkzeuge erledigen.

Installation

JUnit bedarf keiner großen Installation. Nach dem Herunterladen von sourceforge.net muss die .jar-Datei lediglich in den Klassenpfad eingebunden werden. In Eclipse geschieht das durch das Referenzieren der JUnit-Bibliothek im Build-Path.

3 Funktionstests für den Typinferenzalgorithmus

3.1 Einführung und bisheriger Status

Bisher waren systematische Tests nicht Bestandteil der Entwicklung. Jeder Entwickler hat seine eigenen Tests gefahren, diese aber häufig nicht dokumentiert und wieder gelöscht. Während der Arbeit von [SCH07] wurde ein Versuch unternommen bestehende, einzelne Testfälle in eine auf JUnit basierende Umgebung aufzunehmen. Dieses Framework überprüfte jedoch lediglich, dass beim Parsen, bei der Typrekonstruktion und beim anschließenden Kompilieren keine Exceptions geworfen werden. Für eine umfangreichere Überprüfung der richtigen Funktionsweise des Typinferenzalgorithmus ist dies nicht ausreichend.

Genau an diesem Punkt setzt diese Arbeit an. Das erstellte Testframework sowie die dazugehörenden Testfälle fokussieren einen möglichst vollständigen Test der korrekten Funktionsweise der Typinferenz. Es werden vollständige Testklassen getestet. Klassen, wie sie auch später Verwendung finden könnten. Das heißt, die Tests beziehen sich auf vollständige *.jav-Dateien. Dateien, die vom Compiler zunächst geparst, dann inferiert und letztendlich auch in Java-Bytecode umgesetzt werden sollen. Dabei wird allerdings weder die Vollständigkeit des Parsers, noch die Ausführbarkeit des erzeugten Bytecodes betrachtet. Letztere führen jedoch wie bisher im Falle eines auftretenden Fehlers, also einer geworfenen Exception während dem Kompiliervorgang, zum negativen Ausgang des jeweiligen Testfalls.

3.2 Test Framework

Wie bereits im Kapitel über Funktionstests erläutert (siehe Kapitel 2.2 S. 4) bedarf es, über eine grundlegende Testumgebung hinaus, auch einer gewissen Implementierung eines Test-Frameworks, das den Testschreiber in seiner Aufgabe möglichst viel Unterstützung bietet um die komplexen Testergebnisse zu überprüfen. Damit wird das Schreiben der Testfälle erheblich erleichtert.

Das im Weiteren erläuterte Test-Framework basiert auf der JUnit-Testumgebung (siehe Kapitel 2.3 S. 6). Da JUnit allerdings in erster Linie für Unit-Tests gedacht ist, werden

nur grundlegende Bestandteile wie die automatisierte Ausführung und die Möglichkeit der Schachtelung von Testcases in Testsuites genutzt. Auch ein paar Assert-Funktionen von JUnit finden Verwendung um die Typinferierung zu testen. Demnach werden einige, im nächsten Kapitel erläuterte, Designentscheidungen der Auslegung des JUnit-Frameworks unter Umständen etwas widersprechen; für unsere Anforderung, dem einfachen Erstellen und Ausführen der Testfällen, eignet sich JUnit aber dennoch! Ein weitere Grund für die Benutzung von JUnit ist die einfache Integration in die Entwicklungsumgebung Eclipse.

3.2.1 Allgemeine Struktur

Aus Gründen der Übersichtlichkeit sowie der einfacheren Wartbarkeit der Testfälle lag der wesentliche Fokus beim Design des Test Frameworks auf die möglichst einfache Erstellung einzelner Testfälle. Dies führte dazu, dass jeder Testfall genau eine .jav-Datei testet. Da der Compiler immer eine .jav-Datei als Ganzes verarbeitet, gibt es pro Testfall auch immer genau eine Testfunktion, die `testSetup()`. Diese Funktion entspricht der JUnit-Konvention und wird beim Starten eines jeden Tests aufgerufen. Entgegen dem Design von JUnit wird also pro Testfall lediglich diese eine Methode aufgerufen, die alle Compilerschritte durchläuft und das Ergebnis anhand der Erwartungen überprüft. Eine Zerlegung der CUT (Class-Under-Test) in einzelne Tests ergibt sich beim Funktionstest nicht. Ebenso widerspricht die Möglichkeit mehrere .jav-Dateien in einen Testfall zu integrieren und über separate Testmethoden zu testen unserer Vorstellung eines übersichtlichen Testfalls. Der Aufbau eines solchen Testfalls wird in Kapitel 3.2.3 S. 14 erläutert.

Als Konsequenz aus dieser Entscheidung wird ein einzelner Test nun nicht anhand der aufgerufenen Testmethode unterschieden, sondern anhand eines Testfalls - einer Klasse. In dem in Eclipse integrierten JUnit-Testrunner ist dies ein wenig unvorteilhaft dargestellt, da ein möglicher fehlgeschlagener Test nun immer in der gleichen Testmethode auftritt. Da die hierarchische Zuordnung zum jeweiligen Testfall allerdings weiterhin gegeben ist, fällt dies nicht weiter ins Gewicht.

3.2.2 Die Datenstruktur Expectation

Die Datenstruktur der Expectation-Klasse dient der Definition der Erwartungen an einen Testfall. Sie wurde erstellt, um ebenfalls möglichst einfach und logisch die Erwartungen in einem Testfall zu definieren. Die vom Compiler verwendeten Klassen für die Erzeugung eines CTypeReconstructionResult sind, wenn man sie von Hand für jeden Testfall erzeugt, unpraktisch. Diese Klassen sehen keine hierarchische Zuordnung der Bestandteile einer Test-Klasse und deren Methoden, Attributen und lokalen Variablen vor. Zudem wird für jede mögliche Typannahme eine neues CTypeReconstructionResult-Object erstellt, was bei nur einer möglichen Typauswahl enormen Aufwand für den Testersteller bedeutet. Dieses Problem wurde mit der neuen Datenstruktur umgangen. Sie erlaubt für jeden zu bestimmenden Typen einer Variable oder Methode mehrere Ergebnisse zu definieren - gleichbedeutend mit einer vom Benutzer zu treffenden Typauswahl.

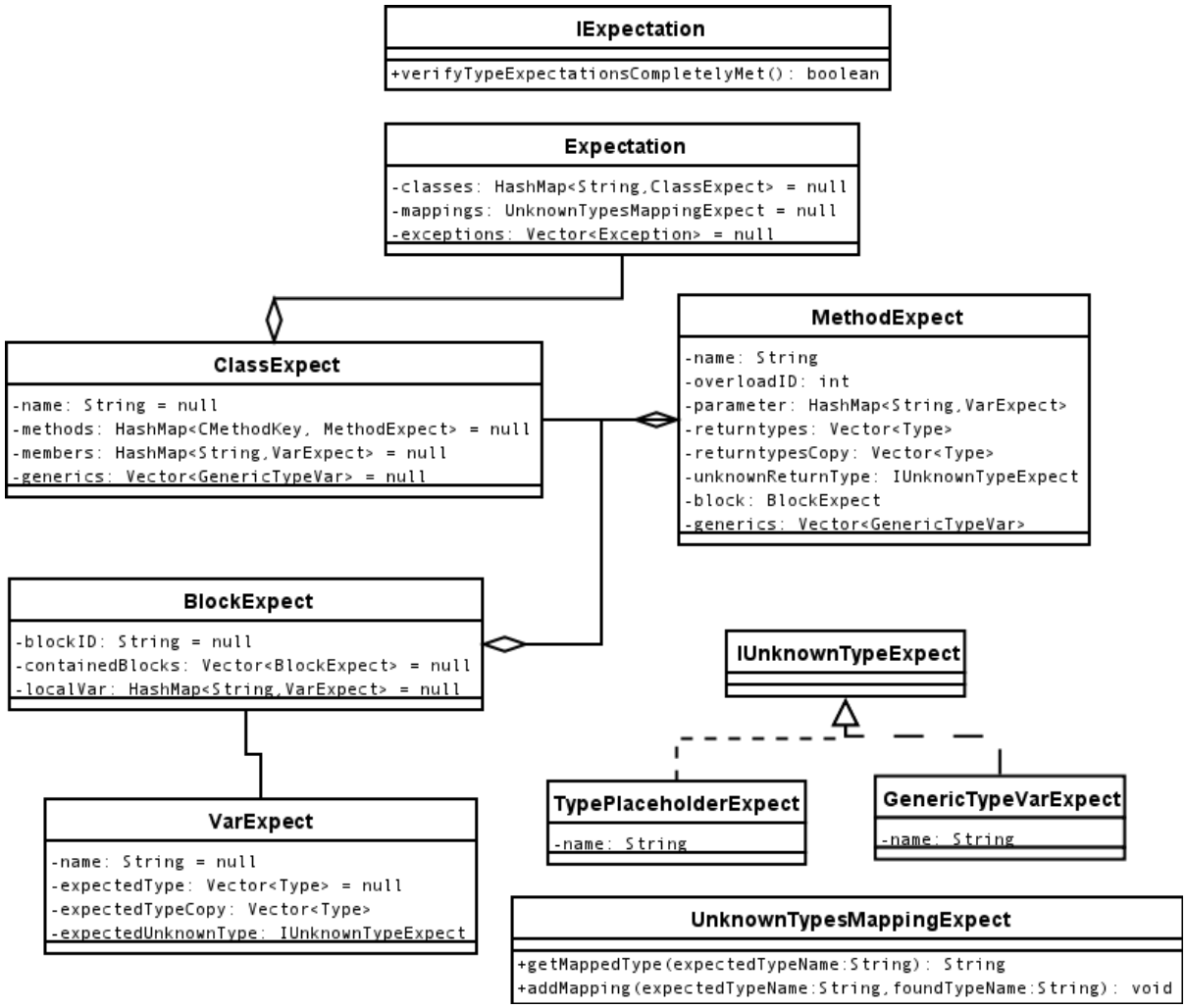

Illustration 1: Klassendiagramm der Expectation Datenstruktur

Wie in dem Klassendiagramm (Illustration 1) verdeutlich wird versucht mit dieser Struktur eine Hierarchie aufzubauen. Eine Erwartung `Expectation` besitzt bestimmte Klassen `ClassExpect`, welche wiederum in Methoden `MethodExpect` unterteilt sind. Methoden beinhalten Blöcke `BlockExpect`, welche Variablen `VarExpect` besitzen. In jeden dieser Bereiche werden andere Eigenschaften wie vorhandene Membervariablen, Methodenparameter oder Generics eingeordnet. Zusätzlich zu den normalen Klassenstrukturen ist auch die Möglichkeit eine zu erwartende Exception zu definieren. Dies geschieht in der `Expectation`-Klasse, da es nur nötig ist global auf eine Exception zu lauschen. Eine einzelne Exception bringt den Testfall zum Abbruch; im Falle einer erwarteten Exception auch mit positivem Ausgang.

Interessant für das Testen des Typinferenzalgorithmus sind in erster Linie natürlich die Typen von Variablen oder Methoden. Hierfür werden bei sämtlichen Variablen bzw. Rückgabetypen von Methoden jeweils eine Liste der möglichen Typen geführt. Diese sind in jedem Fall vom Typ `Type`, wie er bereits vom bestehenden Compiler benutzt wird. Demzufolge ist es hier möglich neben den normalen Objekten `RefType` auch klassifizierte Typen bzw. parametrisierte Typen wie beispielsweise `Vector<String>` zu definieren. Zudem existiert zu jedem Vector der Typerwartungen auch eine Kopie. Die Bedeutung der Kopie wird später im Kapitel 3.2.4 auf Seite 17 näher beleuchtet.

Als Besonderheit an dieser Stelle sei das Interface `IUnknownTypeExpect` erwähnt. Dieses dient dazu einen erwarteten Generic bzw. Typeplaceholder im Testfall zu definieren. Nicht immer lässt sich der zu verwendende Typ einer Variable einschränken, sodass entweder Generics benötigt werden oder eben keine Typannahme gemacht werden kann (TypePlaceholder). Da hier vom Compiler eigenständig Typnamen erzeugt und vergeben werden, ist es nur mit erhöhtem Aufwand möglich, genau den gleichen Typnamen im Testfall zu definieren um eine anschließende Verifikation durchzuführen. Aus diesem Grund wurden für diese beiden Fälle die Klassen `TypePlaceholderExpect` sowie `GenericTypeVarExpect` eingeführt. Mit ihnen lässt sich bei der Definition der Erwartung an ein Testfall ein beliebiger Name wählen, welcher natürlich für gleiche Typen gleich ausfallen sollte. Das entsprechende Ersetzen der selbst definierten Namen beim Vergleich mit dem vom Compiler generierten Namen, übernimmt das Testframework unter Zuhilfenahme der Klasse `UnknownTypesMappingExpect`, welche über die `Expectation`-Klasse referenziert wird. Allerdings ist es nötig für diesen Sonderfall eine eigenen Feldvariable in den Klassen `MethodExpect` und `VarExpect` zu verwenden, da die Klassen `TypePlaceholderExpect` und `GenericTypeVarExpect` nichts mit dem Typ `Type` gemeinsam haben und daher nicht von diesem abgeleitet wurden. Ein einheitliches Interface hätte Abhilfe geschaffen, aber auch einen Eingriff in den Quellcode vom Compiler erfordert. Das sollte jedoch nicht vom Tester geschehen, sondern schon bei der Entwicklung berücksichtigt werden. Außerdem bringt aus Sicht des Compilers ein Interface, dass sowohl von `Type` als auch von den beiden Typen für das Test-Framework implementiert wird, keinen Nutzen für den Compiler selbst. Eine solche Änderung wäre aber für die Zukunft bei einem weiteren Verwendungszweck von Vorteil um die Klassen `MethodExpect` und `VarExpect` logischer zu strukturieren. Auf ein

Aufteilen der zuletzt genannten beiden Klassen in Klassen, welche „normale“ Typannahmen definieren und in welche, die ungewisse Typen definieren, wurde aufgrund der entstehenden Komplexität für die Testerstellung verzichtet.

3.2.3 Aufbau eines Testfalls

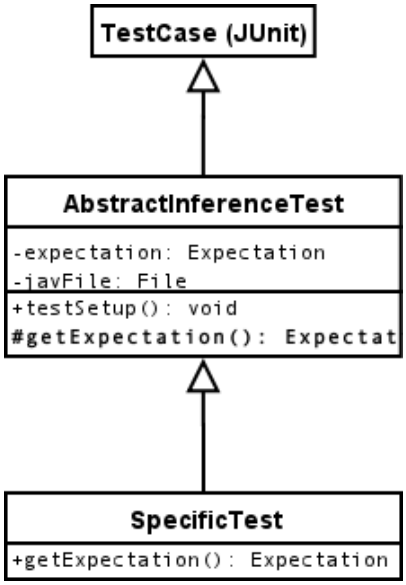

Illustration 2: Aufbau eines Testfalls

Wie in Illustration 2 zu sehen, erbt ein jeder Testfall von der JUnit-Klasse Testcase. Allerdings nur transitiv. Aus den bereits genannten Gründen für die einfache Testerstellung wurde die wesentliche Funktionalität zur Überprüfung eines erfolgreichen Tests in die abstrakte Klasse `AbstractInferenceTest` verlagert und ein einzelner Testfall (Beispiel: „`SpecificTest`“) erbt eben diese Funktionalität. Der genaue Umfang der Funktionalität der abstrakten Klasse wird im folgenden Abschnitt erläutert.

Aus dem obenstehenden Klassendiagramm geht hervor, dass ein jeder Test die Methode `getExpectation()` haben muss. Diese wird vom Framework beim Starten des Tests durch die Methode `testSetup()` in der `AbstractInferenceTest`-Klasse aufgerufen. Somit werden die Erwartungen geladen. Um einen neuen Testcase zu schreiben genügt es demnach, neben der *.jav-Datei, eine Klasse anzulegen, die die abstrakte Klasse `AbstractInferenceTest` erweitert und die Erwartungshaltung an

den Testfall in der Methode `getExpectation()` definiert. Im Folgenden soll die Ausgestaltung dieser Methode mit Hilfe der im letzten Abschnitt erläuterten Datenstruktur Expectation aufgezeigt werden.

Zunächst die *.jav-Datei: Diese könnte wie im Nachfolgenden zu sehen aus einer Klasse `TestSimpleTest` (Testklassen sollten gemäß Konvention immer mit dem Wort „Test“ beginnen) mit einer Methode `m()`, welche wiederum eine Variable `c` enthält, bestehen. Die Methode `m()` hat darüberhinaus noch einen Parameter `b` vom Typ `java.lang.Integer`.

```
public class TestSimpleTest<T> {
      public m(Integer b) {
            c;
            c=b;
            //...
      }
}
```

Die dazugehörige Testklasse `TestSimpleTest.java` muss nun in ihrer Erwartung an den Testfall die Klasse `TestSimpleTest` definieren und darüber hinaus beschreiben, dass diese Klasse eine Methode vom Typ `void` mit dem Parameter `b` besitzt. Die Methode besitzt im 1. Block eine lokale Variable `c`, welche aufgrund der Zuweisung vom Typ `java.lang.Integer` sein sollte und auch so inferiert werden sollte.

```
public class TestSimpleTest extends AbstractInferenceTest {
      public TestSimpleTest(String name) {
            super(name, „TestSimpleTest.jav");
      }
      @Override
      protected Expectation getExpectations() {
            //Class
            ClassExpect testSimpletest = new ClassExpect(„TestSimpleTest“);
```

```
            testSimpletest.addGeneric(new GenericTypeVar(„T“,-1));
            //Method
            MethodExpect m = new MethodExpect("m", new RefType("void",-1));
            m.addParameter(new VarExpect(„b“, new RefType („java.lang.Integer“,
-1);
            //Variables
            VarExpect var = new VarExpect("c", new RefType(„java.lang.Integer“,
-1);
            m.getBlock().addLocalVar(var);
            //add method to class
            testSimpleVariable.addMethod(m);
            //return Expectation
            return new Expectation(testSimpleVariable);
      }
}
```

Zunächst also definiert man die Klassen für den Testfall. In unserem Beispiel ist dies nur die Klasse `TestSimpleTest`. Das geschieht über die Instanzierung der `ClassExpect`-Klasse. An dieser Stelle wird auch der Generic `T` hinzugefügt. Nun folgt die Methode, wobei neben ihrem Namen auch der Rückgabetyp an das MethodExpect-Objekt übergeben wird. In unserem Fall ist nur ein Rückgabetyp – nämlich `void`- möglich. Für den Fall, dass weitere Typen möglich wären, schafft die Methode `addReturntype()` der MethodExpect-Klasse die Möglichkeit solche hinzuzufügen. Da unsere Methode im Testfall auch einen Parameter besitzt, ist das Hinzufügen dieser zum MethodExpect-Objekt `m` nötig. Das geschieht über die Methode `addParameter()`, welche eben eine Variable mit einem Typen erwartet. Zuletzt definieren wir noch die auftretende lokale Variable c. Wie oben zu sehen, wird hier wie auch beim Parameter die VarExpect-Klasse verwendet, die wiederum neben dem Namen der Variablen auch den Typen der Variablen erwartet. Auch hier können weitere Typmöglichkeiten über die Klassenmethode `addExpectedType()` hinzugefügt werden. Nachdem alle Bestandteile des Testfalls definiert wurden, müssen diese noch hierarchisch eingeordnet werden. So wird die Variable dem Hauptblock der Methode m hinzugefügt. Die Methode wiederum wird der

Klasse zugeordnet und die definierte `Expectation` mit der Klasse als Parameter an die aufrufende Instanz zurückgegeben.

Was an dieser Stelle bereits auffällt, sind die vielen `-1` im Testfall. Da wir uns für die Verwendung der vom Compiler definierten Typen entschieden haben, muss das Offset dem Konstruktor eines solchen Typen mitgegeben werden. Das Offset interessiert aber nicht für die Überprüfung der korrekten Funktionsweise des Typinferenzalgorithmus und wäre vom Testschreiber auch nur mit erhöhtem Aufwand zu bestimmen. In diesem Fall fiel die Entscheidung jedoch gegen Wrapper-Klassen, da dies die Testerstellung für andere Entwickler verkompliziert hätte. Jene hätten sich zunehmend mehr in die Klassenstruktur des Frameworks einarbeiten müssen. Es wurde also unter Abwägung der Argumente ein Kompromiss zwischen einfachen Testfällen und einem zugleich simplen Test-Framework gemacht.

3.2.4 Überprüfung der korrekten Typinferierung: Die Klasse AbstractInferenceTest

Das Kernstück des Testframeworks ist die Klasse `AbstractInferenceTest`. Diese Klasse wird von allen Testfällen geerbt und beinhaltet die Logik für die Überprüfung der korrekten Funktionsweise des Typinferenzalgorithmus. Im folgenden soll kurz der Ablauf eines Testfalls sowie die Überpüfung der Ergebnisse erläutert werden.

Über die bereits erwähnte Methode `testSetup()` wird ein jeder Testfall gestartet. Zuerst wird das Logging aktiviert. Das Logging der Testfälle soll getrennt von den Logs des Compilers gehalten werden. Hierzu wurde das bereits verwendete Log4j um eine weitere Instanz erweitert. Die im Package `mycompiler.test` liegende log4jTesting.xml dient der Konfiguration der Loglevel für die Ausführung der Testfälle und kann somit auch getrennt von den Logs des Compiler ausgegeben werden. Um sowohl beim Starten eines einzelnen Tests, als auch beim Starten mehrerer Tests über eine TestSuite ein Logging aktivieren zu können, wurde ein entsprechender Log4j-Wrapper benötigt. Dieses Singleton verhindert, dass das Konfigurieren des Loggers mehr als einmal erfolgt. Hiernach folgen die Compiler-API-Calls zum Parsen, für die Typrekonstruktion und für die Code-Generation.

Die Funktion des Test-Frameworks, die Überprüfung der Ergebnisse der Typrekonstruktion, setzt direkt nach der Typrekonstruktion des Compilers ein. Der

Aufruf von `typeReconstruction()` der Compiler-API liefert das Resultset in Form eines Vectors von `CTypeReconstructionResult`. Wenn sowohl das Resultset wie auch die im Testfall definierte Erwartung an den Testfall regulär, d.h. nicht `null` sind, wird die Subroutine `checkResults()` aufgerufen. Diese Methode ist der eigentliche Einstieg für die Überprüfung. Im folgenden wird der Ablauf der Methode anhand Illustration 3 kurz geschildert.

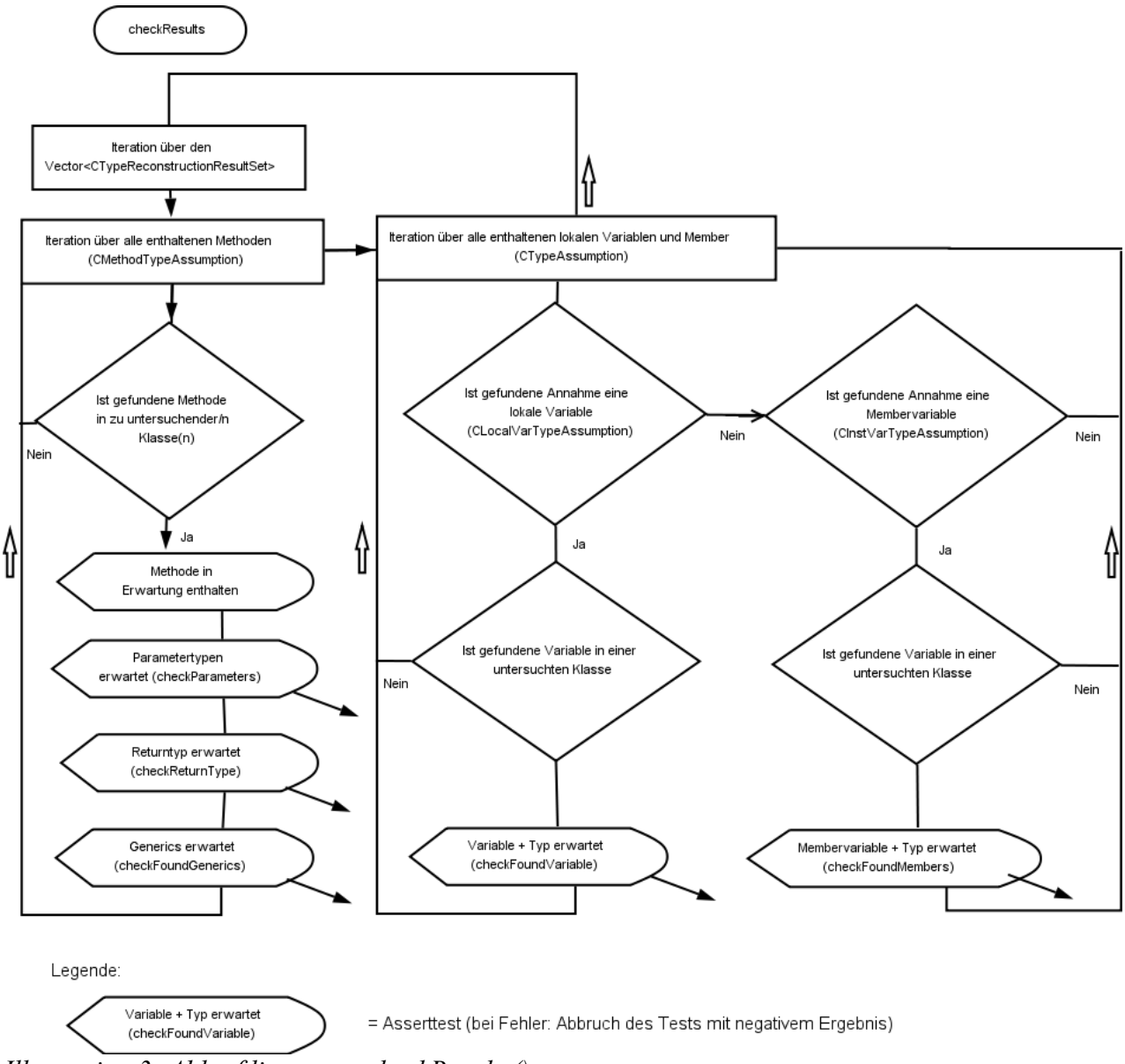

Illustration 3: Ablaufdiagramm checkResults()

Ein jedes `CTypeReconstructionResult`-Objekt besitzt eine komplette, valide Typrekonstruktion. Es wird zunächst über diesen Vector iteriert. Als erster Test wird

überprüft, ob alle erwarteten Klassen im Compiler-Ergebnis vorhanden sind (`checkClasses()`) und sämtliche Klassengenerics den Erwartungen entsprechen. Wie bereits bekannt, kennt das `CTypeReconstructionResult` keine Hierarchie. Im nächsten Schritt wird über alle Methoden iteriert, welche gefiltert nach der zugehörigen Klasse dann einzeln überprüft werden. Es erfolgt zunächst wieder die Überprüfung, ob eine Methode auch Bestandteil der Erwartung ist. Ist dies der Fall, werden die Parameter überprüft, das heißt deren Typinferierung mit den Erwartungen verglichen. Beinhaltet dieses `CTypeReconstructionResult` ein erwartetes Szenario, werden nun der Rückgabetyp und anschließend die Methodengenerics einer Methode überprüft. Der Grund, warum Generics erst am Ende überprüft werden, resultiert aus dem besonderen Fall von unbestimmten Typen von Parametern. Kann der Typinferenzalgorithmus keine Einschränkung für einen Methodenparameter machen, so erzeugt er einen Methodengeneric hierfür. In den Erwartungen ist der Testschreiber aber nicht in der Lage vorher zusehen, wie dieser Generic heißt. Beim Überprüfen der Parameter wird für den Fall, dass ein Parameter generisch ist, der Typ, der vom Compiler inferiert wurde, auch in die erwartete Generic-Liste aufgenommen um anschließend eine erfolgreiche Überprüfung der Generics durchzuführen. Hierzu wird die bereits erwähnte Klasse `UnknownTypesMappingExpect` verwendet, die die Beziehung zwischen dem automatisch generierten Namen eines generischen Methodenparametertypen mit dem der Erwartung kennzeichnet.

Nach den Methoden werden alle lokalen Variablen betrachtet. Wiederum wird über das eine `CTypeReconstructionResult` iteriert und relevante Variablen für die getesteten Klassen gefiltert. Da eine lokale Variable immer nur eindeutig pro Block ist und Blöcke wie bekannt beliebig in einer Methode geschachtelt werden können, werden die Variablen über einen rekursiven Algorithmus in den gestellten Erwartungen gesucht. Wird die Variable gefunden und ihr Typ verifiziert, wird die Iteration inkrementiert und die nächste Variable überprüft.

Zuletzt werden die Membervariablen zur Kontrolle herangezogen.

Bei allen Typüberprüfungen wird, falls der gefundene Typ auch erwartet wurde, dieser aus einer Kopie der Liste mit den erwarteten Typen gelöscht. Auf diese Weise kann überprüft werden, ob nicht nur alle inferierten Typen erwartet, sondern auch alle erwarteten Typen inferiert wurden. Dies geschieht mit der Methode

`verifyTypeExpectationsCompletelyMet()` der Expectation-Datenstruktur. Sie überprüft, ob sämtliche `expectedTypesCopy`-Vektoren leer sind und somit vollständig inferiert wurden. Dass gefundene Typen nicht aus der eigentlichen Liste mit den erwarteten Typen gelöscht werden konnten, liegt daran, dass es für jede mögliche Kombination einer Typinferierung ein neues `CTypeReconstructionResult`-Objekt gibt und somit auch beispielsweise mehrmals die gleiche Variable mit dem gleichen Typen überprüft werden muss.

Nachdem die Typinferierung erfolgreich überprüft wurde, wird das erste `CTypeReconstructionResult` benutzt um eine Typsubstitution durchzuführen. Dies ist erforderlich um orginalen Java-Code zu erhalten, welcher dann in Java-Bytecode umgesetzt wird. Das Konzept hierfür ist der alten Testumgebung in `mycompiler.mytest` von [SCH07] entnommen.

Tritt nun während der Ausführung der `testSetup()` eine Exception auf, so wird diese behandelt und untersucht, ob hierfür eine Erwartung definiert wurde. Dies ermöglicht zudem Negativtests zu definieren. Wurde keine Erwartung für eine Exception definiert, wird die auftretende Exception an die JUnit-Instanz weitergeleitet und ein negativer Testausgang ist die Folge.

3.3 Struktur der Testfälle

Um einen möglichst weitreichenden und umfassenden Funktionstest des Typinferenzalgorithmus zu ermöglichen wurde eine sinnvolle Strukturierung der Testfälle gesucht. Diese Struktur ist in Form von java-packages abgebildet. Außerdem wurde versucht, möglichst beschreibene Datei-/Klassennamen zu finden.

Zur Ausführung der Testfälle wird die Struktur in Form von JUnit-Testsuites umgesetzt. So besitzt jede Kategorie eine TestSuite, die die in der Kategorie enthaltenen Testfälle listet. Die einzelnen Kategorien werden wiederum in einer Ober-Testsuite „AllTests.java“ im Paket mycompiler.test zusammengefasst. Auf diese Weise lassen sich alle Tests auf einmal oder auch nach Kategorie getrennt ausführen. Natürlich kann auch ein jeder Test einzeln ausgeführt werden, indem er beispielsweise im Eclipse als „JUnit-Test“ ausgeführt wird.

Im folgenden sollen die gewählten Kategorien ein wenig näher erläutert werden.

mycompiler.test.trivial

Hier liegen Testfälle, die sich mit den einfachsten Strukturen von Java-Programmen beschäftigen. So werden leere Klassen, leere Methoden, Konstruktoren und Interfaces auf ihre korrekte Erfassung getestet. Dabei wurde kein Wert auf vollständige Parsertests gelegt, sondern der Aufgabe entsprechend die für den Typinferenzalgorithmus entscheidenen Strukturen untersucht.

mycompiler.test.primitiveTypes

Diese Kategorie hat den Anspruch alle möglichen, einfachen Typen zu testen. Das heißt, Tests in dieser Rubrik überprüfen die richtige Inferierung bzw. Erkennung der in Java bekannten primitiven Typen. Da sich der Typinferenzalgorithmus hauptsächlich auf Objekte beschränkt, geht es in erster Linie um die Wrapper-Klassen der primitiven Typen.

mycompiler.test.complexTypes

Als Gegenstück zu den primitiven Typen wird hier versucht, komplexe Typen, d.h. eigene Objekte und auch von Java bekannte Standard-Typen, zu examinieren. Das geschieht aufgrund der Fülle an Typen aus der Standardbibliothek nur exemplarisch. So wird überprüft, ob importierte Typen richtig erkannt und inferiert werden und etwaige Vererbungsstrukturen der Typen der Standardbibliothek eingehalten werden.

mycompiler.test.operators

Die in Java bekannten Operatoren sind fest definiert. Java erlaubt kein Überschreiben von Operatoren. Somit können hier relativ umfassend sämtliche bekannte Operatoren getestet werden. Dabei geht es in erster Linie darum, den Algorithmus dahingehend zu testen, das richtige Typen zu richtigen Operatoren inferiert werden. So kann es sich beispielsweise beim „+“-Operator sowohl um die arithmetische Operation auf Zahlen, als auch um die Operation auf die String-Klasse zur Konkatenation handeln.

mycompiler.test.blocks

Wie schon im package `mycompiler.test.trivial` handelt es sich hier zu gewissen Teilen um Parsertests. Neben Elementen wie If- oder While-Statements werden aber auch

die korrekte Behandlung von Verschachtelungen von Blöcken und den zugehörigen Scopes lokaler Variablen getestet. Wiederum liegt der Fokus nicht auf vollständigen Parser-Tests, sondern um prinzipielle Tests für die Typinferierung. Für die Typinferenz ist es relativ unwichtig, ob es sich um einen Bedingungsblock oder um eine vorprüfende Schleife handelt.

mycompiler.test.generics

Ein wichtiges, weil auch komplexes, Thema für die Typinferierung ist das Gebiet der generischen Typen. Hier finden sich sämtliche in Verbindung mit Generics interessante Tests, die auch entsprechende Bounds und Wildcards testen. Zudem können Generics geschachtelt werden. Die korrekte Benutzung und die Beachtung der vorhandenen sowie berechneten Einschränkungen sind von großem Interesse. Aufgrund der Fülle von möglichen Kombinationen und vorstellbaren Konstellationen wurde versucht, exemplarisch Konzepte zu testen.

mycompiler.test.staticAccess

Eine bedeutende Eigenschaft von objektorientierten Sprachen ist der Unterschied zwischen einem Zugriff auf ein Objekte und einem prozeduralen Zugriff auf eine Klasse. Wie der Name impliziert geht es hierbei um Zugriffskonzepte, welche auch auf den Bereich der Modifier erweitert werden könnte. Schließlich wird hierüber auch ein Scope für die Typinferierung definiert.

mycompiler.test.javaConcepts.inheritance

Eine große Bedeutung objektorientierter Sprachen wie Java hat die Vererbung. Eine vollständige Testsuite ist aufgrund der Fülle von Szenarien wiederum nur schwer möglich. Somit liegt der Schwerpunkt auf bestimmten Konzepten der Vererbung. Inferenz über Interfaces, mehrere Vererbungshierarchien sowie das Konzept des Überschreibens und dergleichen werden getestet.

mycompiler.test.javaConcepts.overloading

Das Thema Überladen wird in dieser Kategorie ausgiebig getestet. So unterliegt das Überladen einigen Regeln, die bei gleicher Anzahl von Parametern beispielsweise für den Typinferenzalgorithmus von Interesse sind. Auch ist es nicht möglich Methoden zu überladen, die sich nur im Rückgabetyp unterscheiden.

mycompiler.test.inferenceByCharacteristic

Hier wird der Typinferenzalgorithmus weitestgehend in Abhängigkeit von den möglichen Wegen einer Typinferierung getestet. Das heißt, es werden die Elemente untersucht, die eine Aussagekraft für die Typerkennung haben.

3.4 Ergebnisse der Testfälle

Zum Zeitpunkt der Erstellung dieser Arbeit befinden sich 87 Testfälle, die allerdings öfter mehrere Testszenarien gleichzeitig testen, in der Testsuite. Bei aktivierter Codegenerierung schlagen 42 Tests fehl, wobei 17 die Erwartungen nicht erfüllen. Schaltet man die Codegenierung ab, so haben 30 Tests einen negativen Ausgang, wovon 18 die Erwartungen nicht erfüllen. Ein negativer Ausgang eines Tests muss nicht unbedingt einhergehen mit einer unerfüllten Erwartung. Manche Testszenarien schlagen schlicht und einfach beim Kompilieren fehl. Der zusätzliche, die Erwartungen nicht erfüllende Test beim Abschalten der Codeerzeugung entsteht dadurch, dass eine erwartete Exception erst in diesem letzten Compilerschritt geworfen wurde.

Als ein Ergebnis der Arbeit werden im folgenden die Ausgänge der Testfälle erläutert und auftretende Fehler näher untersucht. Um eine Übersichtlichkeit zu wahren, unterteilen wir wiederum nach den Kategorien.

Als Erläuterung zu den Grafiken sei gesagt, dass jeweils mit und ohne Codeerzeugung getestet wurde. Die Farben erklären sich wie folgt:

- grün**:** Testfall erfolgreich verlaufen
- rot: Es wurde eine Exception beim Testen geworfen
- braun: Die Erwartung wurde nicht erfüllt (ein Assert-Statement schlug fehl).

mycompiler.test.trivial

Testcase-Name	mit Codegen	ohne Codegen
TestClassEmpty		
TestClassEmptyGenerics		
TestClassMember		
TestClassMemberAssignment		
TestConstants		
TestConstructor		
TestConstructorNegative		
TestInterfaceMember		
TestInterfaceMethod		
TestInterfaceEmpty		
TestMethodEmpty		
TestMethodEmptyGeneric		
TestMethodEmptyParameter		
TestMethodEmptyParameterGenericExtends		
TestInterfaceNotInferenced		

Illustration 4: mycompiler.test.trivial

TestClassMemberAssignment

Dieser Testfall schlägt mit einer Parser-Exception fehl. Dabei liegt das Problem in dem `this`-Descriptor, welcher in Bezug auf eine Member-Referenzierung vom Parser nicht erkannt wird:

```
public class TestClassMemberAssignment {
      Integer member;
      public void m(a) {
            this.a=a; //error
      }
}.
```

TestConstructorNegative

Die Erwartung an den Testfall war, dass eine Exception geworfen wird, weil ein Konstruktor keinen benutzerdefinierten Rückgabewert haben kann. Der Compiler erlaubt dies zur Zeit dennoch.

```
public class TestConstructorNegative {
      public TestConstructorNegative(java.lang.Integer a, String b) {
            return a; //error
      }
}
```

TestInterfaceMember

Der Parser erlaubt das Initialisieren von Attributen in Interfaces nicht.

```
interface TestInterfaceMember {

      Integer member=3; //error

}
```

TestMethodEmptyParameter

Bestandteil des Testfalls ist eine Methode mit frei wählbaren Typen für die beiden Methodenparameter.

```
public class TestMethodEmptyParameter{
      emptyMethodParameter3(a,b) {
      }
}
```

Beide Methodenparameter können demzufolge generische Typen besitzen, die auch richtig als Methodengenerics inferiert werden. Allerdings liefert der Compiler auch den Fall, dass beide Parameter den gleichen generischen Typen besitzen. In diesem Fall ist nur ein Methoden-Generic von Nöten. Das Resultset vom Compiler beinhaltet jedoch auch für diesen Fall zwei Methoden-Generics, was falsch ist und vom Eclipse-Plugin entsprechend behandelt werden muss:

```
public class TestMethodEmptyParameter{
      void <T,S> emptyMethodParameter3(T a, T b) {
      }
}
```

Allein an dieser Stelle stellt sich die Frage, ob der Fall eigentlich gar nicht erst inferiert werden sollte, da es sich quasi um einen Spezialfall handelt. Dieser Speziafall sollte demnach erst durch Einsetzen auftreten. Der Fall lässt Raum für Diskussionen.

TestInterfaceNotInferenced

Nach gemeinsamer Auffassung sind wir in Rücksprache mit Herrn Plümicke und Herrn Holzherr zu dem Schluss gekommen, dass es wenig Sinn macht, ein Interface zu inferieren. Schließlich dient ein Interface der Vereinbarung einer vorgegebenen Struktur. Ein Interface besitzt zu dem keine Implementierung, weshalb es für den Typinferenzalgorithmus nicht möglich ist, qualifizierte Aussagen über mögliche Typen zu

machen. Der Algorithmus würde sie in jedem Fall frei wählbar, also als Generics inferieren. Das kann aber nicht Sinn eines Interfaces sein. Insofern testet dieser Testfall, dass der Compiler einen Fehler werfen muss, wenn Deklarationen in einem Interface nicht der normalen Java-Grammatik entsprechen. Der Testfall schlägt fehl, da diese Entscheidung erst kürzlich gefällt wurde und noch nicht im Projekt verwirklicht wurde.

mycompiler.test.primitiveTypes

Testcase-Name	mit Codegen	ohne Codegen
BooleanTest		
ByteTest		
CharTest		
DoubleTest		
FloatTest		
IntegerTest		
LongTest		
StringTest		
TestSimpleTypes		

Illustration 5: mycompiler.test.primitiveTypes

DoubleTest/FloatTest/LongTest

Bei den auftretenden Fehlern handelt es sich durchweg um Parserfehler. So wird die Schreibweise, die den jeweiligen Typen identifiziert, nicht vom Parser akzeptiert.

```
void longTest(l){
            l = 4L;//error
      }
void floatingPoint(f){
            f = 5f;//error
      }
void doublePrecision(d){
            d = 1.0;//error
      }
```

TestSimpleTypes

Primitive Typen wie „int, short, char, …“ wurden nicht vollständig implementiert. Intensiveres Testen wurde daher nicht betrieben und wäre beim Ausbau der Funktionalität für (Un-)Boxing und die Verwendung vom Typinferenzalgorithmus

notwendig und nachträglich zu erweitern.

mycompiler.test.complexTypes

Testcase-Name	mit Codegen	ohne Codegen
TestOwnClassMember		
TestOwnClassMethod		
TestStandardLibMember		
TestStandardLibMethod		
TestStandardLibInheritanceInference		

Illustration 6: mycompiler.test.complexTypes

Hier schlagen sämtliche Tests bei der Codegenerierung fehl. Lässt man diese weg, so tritt nur noch der folgende Fehler auf:

TestStandardLibMember

Der Grund für den negativen Ausgang des Testfalls ist die Inferierung eines Standardtypen anhand des Typs einer Membervariable. Im Testfall wurde dies beispielsweise mit der Feldvariable `out` der Klasse `System` getestet.

```
import java.lang.System;
import java.io.PrintStream;

public class TestStandardLibMember {

	public m1() {
		a;
		a=System.out; //error
		a.println();
	}
}
```

`a` kann nicht nicht als `PrintStream` inferiert werden.

TestStandardLibInheritanceInference

Im folgenden Testfall wird die Inferierung einer Variablen anhand der gerufenen Methode getestet. Hierbei handelt es sich um eine Methode aus der Java-Standard-Bibliothek. Gemäß dem Typinferenzalgorithmus müssten alle importierten Typen, die diese Methode besitzen, inferiert werden. Im folgenden Beispiel werden alle Typen, bis

auf den `java.io.BufferedOutputStream`, richtig inferiert. Letzterer taucht nicht im Ergebnis des Compilers auf, was zum negativen Ausgang des Testfalls führt.

```
import java.io.OutputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.BufferedOutputStream;
import java.sql.Connection;

class TestStandardLibInheritanceInference{
      foo(x){
            x.close();
      }
}
```

mycompiler.test.operators

Die TestSuite „AllTestsOperators“ ist maßgeblich durch die vom Compiler bzw. Parser akzeptierten Operatoren bestimmt. So sind derzeit Operatoren für Bit-Shifting nicht implementiert und im Parser auskommentiert. Zudem sind ebenso nicht alle in Java bekannten primitiven Typen implementiert, sodass auch hier Einschränkungen hinzunehmen sind.

Testcase-Name	mit Codegen	ohne Codegen
TestOperatorArithmetic		
TestOperatorBitwise		
TestOperatorBool		
TestOperatorComparison		
TestOperatorIncrement		
TestOperatorObjects		
TestOperatorString		

Illustration 7: mycompiler.test.operators

TestOperatorArithmetic

Der positive Ausgang des Testfalls, wenn man die Codeerzeugung außen vor lässt, ist nur temporär. Typen, die bisher im Compiler nicht implementiert sind, wurden zunächst auskommentiert. So werden bei den Operationen lediglich die Typen `java.lang.Integer`, `java.lang.Long`, `java.lang.Float` und

`java.lang.Double` betrachtet.

TestOperatorBitwise

Da die Bitshift-Operatoren vom Parser nicht akzeptiert werden, beschränken wir uns auf die Operatoren &, ^ und | . Laut [SUN08] sind hier nur ganzzahlige Typen erlaubt. Dies schränkt mögliche Operandentypen der Operatoren auf `java.lang.Byte`, `java.lang.Short`, `java.lang.Character`, `java.lang.Integer` und `java.lang.Long` ein.

Im folgenden Codebeispiel wird vom Compiler jedoch ein generischer Typ angenommen. Auch der Rückgabetyp wird falsch inferiert. Dieser ist entweder `java.lang.Long` oder `java.lang.Integer`. Für den Fall, dass einer der Operanden vom Typ `java.lang.Long` ist, muss der Rückgabetyp ebenfalls `java.lang.Long` sein.

```
public class TestOperatorBitwise {

      public m1(a,b) {
            return a & b;
      }
//...

}
```

Die Ursache für diesen Fehler liegt bei näherer Betrachtung nicht in einer falschen Implementierung, sondern darin, dass die Operatoren vom Parser erkannt, aber nicht gesondert behandelt werden. Es handelt sich also in erster Linie um ein noch zu implementierendes Feature.

TestOperatorBool

Dieser Testfall funktioniert prinzipiell sehr wohl. Der Grund für das Fehlschlagen des Testfalls ist die unter **mycompiler.test.trivial -> TestMethodEmptyParameter** beschriebene Problematik mit den Methodengenerics.

TestOperatorComparison

Hier tritt ein unerwartetes Problem auf. Im folgenden Code tritt eine Abhängigkeit zwischen den verschiedenen Methoden auf, die so überhaupt nicht vorhanden ist:

```
public class TestOperatorComparison {

        public m1(a,b) {
                return a < b;
        }

        public m2(a, b) {
                return a >= b;
        }

        public m3(a, b) {
                return a==b;
        }
...
}
```

Der eigentliche Fehler tritt in der Methode `m3()` auf. Hier müsste eigentlich für `a` und für `b` je ein Generic inferiert werden. Jedoch anstelle von 2 Generics werden unterschiedlich viele Generics erzeugt und zwar in Abhängigkeit der Anzahl der Vergleiche in den Methoden zuvor. So werden im aktuellen Fall rund 30 Generics inferiert. Auch nachfolgende Vergleiche haben einen Einfluss auf die Anzahl der möglichen generischen Typen für `a`, `b` und den Rückgabetyp. Wird nur ein Vergleich gemacht, so werden 2 generische Typen inferiert, was korrekt wäre.

TestOperatorString

Der Operator „+“ ist für die Konkatenation von Strings leider nicht implementiert, weshalb der Testfall fehlschlägt.

mycompiler.test.blocks

Testcase-Name	mit Codegen	ohne Codegen
TestForStmt		
TestIfStmt		
TestInferenceAcrossBlocks		
TestSimpleBlocks		
TestSimpleVariable		
TestWhileStmt		
TestUndeterminedReturnNegative		
TestSwitchStmt	nicht implementiert	
TestTryCatchBlock	nicht implementiert	
TestUnitializedVariable		

Illustration 8: mycompiler.test.generics

Der Compiler/Parser versteht nicht alle Sprachelemente von Java. So ist ein Switch-Statement oder ein Try-Catch-Block nicht definiert. Abgesehen von den Parserproblemen und mit Fokus auf die Typinferierung treten nur 2 Fehler auf:

TestUndeterminedReturnNegative

Im Testfall wird ein Szenario getestet, das so in Java nicht erlaubt ist. So wird nur in Abhängigkeit des Parameters `a` auch ein Wert zurückgegeben. Gilt beispielsweise `a=true`, so wird nichts an die aufrufende Instanz zurückgegeben. Der Rückgabetyp der Methode ist also nicht klar:

```
public class TestUndeterminedReturnNegative {

	public m1(a) {
		if (a) {
			a=false;
		}
		else
			return false;
	}
}
```

Die Erwartung definiert demzufolge eine Exception, die vom Compiler geworfen werden müsste. Dies geschieht jedoch nicht!

TestUninitializedVariable

Das Lesen einer nicht initialisierten Variable sollte vom Compiler verhindert werden.

Zwar wird in Java standardmäßig eine Integer-Variable mit 0 initialisiert, dennoch ist dies ein vom Programmierer unkontrolliertes Verhalten, dass als Fehler markiert werden muss. Folgender Code wirft jedoch keine Exception, weshalb der Test fehl schlägt:

```
public class TestUninitializedVariable {

	public void m1() {
		a;
		a++; //error
	}

}.
```

mycompiler.test.generics

Testcase-Name	mit Codegen	ohne Codegen
TestClassesWithBoundedGenericsOfTwoTypes		
TestClassesWithBoundedGenericsUsedInMethods		
TestExtendedClassesWithBoundedGenerics		
TestSimpleClassesWithBoundedGenerics		
TestSimpleClassesWithBoundedGenericsNegative		
TestSimpleClassesWithGenerics		
TestSimpleClassesWithGenericsNegative		
TestAssignmentTwoGenericTypesNegative		
TestNestedGenerics		
TestNestedGenericsNonExistingType		

Illustration 9: mycompiler.test.generics

TestClassesWithBoundedGenericsOfTwoTypes

Dieser Test schlägt fehl, weil eine erwartete Exception nicht geworfen wird. So ist in der letzten Klassendefinition in dem folgenden Beispiel der Generic `T` daran gebunden, dass ein Typ zwei Klassen erweitert. In Java ist Mehrfachvererbung aber nicht möglich.

```
interface I1 {}
interface I2 {}
class Implementation implements I1 {}
class Implementation2 implements I2 {}
class E<T extends Implementation & Implementation2>{} //error
```

TestClassesWithBoundedGenericsUsedInMethods/TestExtendedClassesWithBoundedGenerics

In diesen beiden Testfällen tritt der gleiche Fehler auf. So werden die Bounds eines Klassen-Generics für die spätere Verwendung des Typs nicht beachtet. Im folgenden Beispiel etwa, wird in der Klasse `Tester` ein Generic `T` eingeführt, dessen Bounds klarstellen, dass `T` nur einen Typen haben kann, der beide Interfaces `I1` und `I2` implementiert. Dieses Szenario impliziert, dass der Typ `T` beide Methoden `m1()` und `m2()` haben muss und ein Aufruf dieser demnach valide ist. Jedoch ist bei Verwendung des Typs `T` in der Methode `m3()` nicht mehr bekannt, dass `T` beide Interfaces implementiert:

```
interface I1 {
	public m1();
}

interface I2 {
	public m2();
}

class Implementation implements I1, I2 {
	public m1() {}

	public m2() {}
}
class Tester<T extends I1 & I2> {

	public m3(T x) {
		x.m1(); //error
		x.m2();
	}
}
```

TestSimpleClassesWithBoundedGenericsNegative

Wie im folgenden Beispiel ersichtlich, wird in der Methode `m1` die Klasse `B` mit dem Parameter `String` instanziert. Die Klasse `B` hat jedoch einen gebundenen Generic, der einen von `java.lang.Number` abgeleiteten Typen erwartet. Dies ist invalider Code der eine Exception werfen müsste, was allerdings nicht passiert. Aus diesem Grund ist der Testausgang negativ.

```
public class B<T extends java.lang.Number> {
	public m(T x) {
		return x;
	}
}
```

```
class A {
      public m1() {
            x;
            x = new B<String>();
            return x.m("abc");

      }
}
```

TestNestedGenerics

In diesem Testfall wird die mögliche Verschachtelung der Bounds von Generics getestet. So wird definiert, dass der Generic `T` der Methode `foo()` entweder eine Instanz der Klasse `A<T>` oder einer davon abgeleiteten Klasse sein muss. Zusätzlich ist der zweite Parameter der Methode vom Typ des Parameters der Klasse A. Die Methode `m()` testet dieses Verhalten entsprechend, indem Parameter a vom Typ `A<String>` und der Parameter `b` vom Typ `String` ist.

```
class A<T>{

}

public class TestNestedGenerics{
 <T extends A<C>,C> void foo(T a,C b){}

 void m(){
  TestNestedGenerics t = new TestNestedGenerics();
  String str = new String();
  A<String> a = new A<String>();
  t.foo(a,str);
 }
}
```

In Java ist dies valider Code, der allerdings in unserem Typinferenzalgorithmus fehl schlägt, da keine Methode gefunden werden kann, die mit diesen Parametern aufgerufen werden könnte.

TestNestedGenericsNonExistingType

Dieser Testfall ist ähnlich dem vorher aufgeführten Fall, außer das dieses Mal die Klasse `A<T>` nicht definiert wurde. Demzufolge müsste folgende Codezeile zu einem Fehler führen:

```
public class TestNestedGenerics{
 <T extends A<C>,C> void foo(T a,C b){}
}
```

Allerdings akzeptiert der Compiler diese Zeile, weshalb der Test negativ ausfällt.

mycompiler.test.staticAccess

Testcase-Name	mit Codegen	ohne Codegen
TestNonStaticAccess		
TestStaticAccess		
TestStaticAccessError		

Illustration 10: mycompiler.test.staticAccess

Bei eingeschalteter Codegenerierung schlagen hier eigentlich alle Testfälle fehl. Da jedoch im Fall von `TestStaticAccessError` eine Exception erwartet wird, ist der Ausgang positiv. Das ist natürlich nicht korrekt, denn der Fehler sollte nicht erst bei der Codegenerierung auftreten. Die ohnehin fehlerhafte Codeerzeugung verfälscht hier also das Ergebnis. Die Ursache für die Verfälschung liegt nun darin, dass das Test-Framework derzeit noch nicht genau überprüft, welche Art von Exception geworfen wird. Das Ausschalten der Coderzeugung zeigt letztendlich, dass folgender Testfall fehl schlägt.

TestStaticAccessError

Das Konzept des statischen bzw. nicht-statischen Zugriffs auf Klassen bzw. Objekte wurde im vorliegenden Compiler nicht behandelt. Der Test bei dem auf nicht statische Elemente auf statischem Weg zugegriffen wird, schlägt also demnach nicht wie erwartet fehl.

```
public class Access{
	public String nonStaticMember="abc";
	public nonStaticMethod() {
		return "def";
	}
}
class TestStaticAccessError {
	public methodError1(a,b) {
```

```
            x;
            a=x.nonStaticMember; //error
            b=x.nonStaticMethod(); //error
      }
}
```

Generell ist an dieser Stelle zu bemerken, dass verschiedene Java Schlüsselwörter keine Beachtung finden. Dazu gehören die Modifier `private`, `public`, `protected`, aber auch `abstract`.

mycompiler.test.javaConcepts.inheritance

Testcase-Name	mit Codegen	ohne Codegen
TestInheritanceAcrossLevel		
TestInheritanceCircle		
TestIneritanceConstructor		
TestInheritanceMultiple		
TestInheritanceMultipleClasses		
TestInheritanceOverriding		
TestInheritanceTwoHierarchies		
TestSimpleInheritance		

Illustration 11: mycompiler.test.javaConcepts.inheritance

Wiederum lassen wir in unseren näheren Betrachtungen die Fehler, die bei der Codeerzeugung auftreten außen vor. Dann verlaufen die Testfälle bis auf einen korrekt.

TestInheritanceConstructor

Der Testfall benutzt den `super()`-Konstruktor, welcher schlicht vom Parser nicht akzeptiert wird.

```
class A {
      public Integer member;

      public A(i) {
            member=i;
      }
```

```
}

class B extends A {
	public String memberString;
	public B(i,s) {
		super(i); //error
		memberString=s;
	}
}
//...
```

mycompiler.test.javaConcepts.overloading

Testcase-Name	mit Codegen	ohne Codegen
OverloadingDifferentNumberOfParameters		
OverloadingDifferentNumberOfParametersAndDifferentTypes		
OverloadingGenericNotSameHierarchy		
OverloadingGenericSameHierarchy		
OverloadingGenericTypeInferenceNotSameHierarchy		
OverloadingGenericTypeInferenceSameHierarchy		
OverloadingNotSameHierarchy		
OverloadingSameSignature		
OverloadingSameSignatureDifferentReturnTypes		
OverloadingSameSignatureGenerics		
OverloadingSameHierarchy		
OverloadingTypeInferenceNotSameHierarchy		
OverloadingTypeInferenceSameHierarchy		

Illustration 12: mycompiler.test.javaConcepts.overloading

OverloadingSameSignature

Hier werden zwei vollständig identische Methoden in einer Klasse definiert, was natürlich nicht erlaubt ist. Der Compiler wirft jedoch die Exception erst bei der Bytecodeerzeugung. Da dies aber auch im Plugin dem Benutzer erkenntlich gemacht werden sollte, ist das Verhalten nicht richtig. Eine entsprechende Exception sollte bereits während der Typinferierung geworfen werden. Deshalb schlägt dieser Testfall, wenn die Bytecodeerzeugung deaktiviert ist, fehl.

```
class OverloadingSameSignature{
	void foo(){
	}
	void foo(){ //error
	}
}
```

OverloadingSameSignatureDifferentReturnTypes/OverloadingSameSignatureGenerics

Hier tritt beide Male der gleiche Fehler auf. Es wird jeweils eine Methode überladen,

wobei die zweite Methode sich nur im Rückgabetyp unterscheidet. Dies ist nach Java nicht möglich, da beim Aufruf der Methode vom Compiler/JVM nicht unterschieden werden kann, welche der beiden Methoden aufgerufen werden muss. Die erwartete Exception wird von unserem Compiler jedoch nicht geworfen, was zum negativen Ausgang beider Testfälle führt.

```
class OverloadingSameSignatureDifferentReturnTypes{
      void foo(){
      }
      String foo(){
            return "abcd";
      }
}
```

mycompiler.test.inferenceByCharacteristic

Testcase-Name	mit Codegen	ohne Codegen
TestInferenceOwnTypeByMember		
TestInferenceOwnTypeByMemberAcrossClasses		
TestInferenceOwnTypeByMethodCall		
TestInferenceOwnTypeByMethodCallAcrossClasses		
TestInferenceOwnTypeByMethodParameter		
TestInferenceOwnTypeByMethodReturnTypeMixed1		
TestInferenceOwnTypeByReturnType		
TestInferenceOwnTypeByReturnTypeAcrossClasses		
TestInferenceStdTypeByOperation		
TestInferenceStdTypeByReturnType		

Illustration 13: mycompiler.test.javaConcepts.

TestInferenceOwnTypeByMember

Hier tritt ein bereits vorher beschriebener Fehler auf. So wird der `this`-Operator vom Parser nicht verstanden, was zu diesem negativen Ergebnis führt (siehe mycompiler.test.trivial -> TestClassMemberAssignment S. 38)

TestInferenceOwnTypeByMethodCall

Folgender Testcase deckt einen interessanten Fehler im Compiler auf:

```
public class TestInferenceOwnTypeByMethodCall{

  public m1(a) {
    return a;
```

```
  }
  public void m01(d) { //error
      e;
    e=m1(d);
  }
```

So wird für `a` ein Generic inferiert, der auch als Methodengeneric erkannt wird. Dies sollte auch bei `d` passieren und `e` sollte den gleichen Typen wie `d` bekommen. Allerdings wird bei `m01()` kein Methodengeneric erzeugt, sondern der exakt gleiche Typ wie für `a` angenommen. Dies ist falsch, da ein Generic immer einen Scope hat, in diesem Fall den Scope einer Methode. Da für a der Typ ebenso egal ist wie für `d`, ist es nicht wichtig, festzulegen, dass diese Typen immer die gleichen sind. Hier muss grundlegend nachgebessert werden und die Reichweite eines Generics genauer definiert werden.

TestInferenceStdTypeByOperation

Hier wird versucht, den Typ eines Methodenparameters anhand einer Methode der Standardlibrary zu bestimmen. Obwohl entsprechend erwarteter Typ importiert ist, inferiert der Compiler im folgenden Codebeispiel für a nicht `java.util.HashMap<String,String>`.

```
import java.util.HashMap;
import java.lang.Object;

public class TestInferenceStdTypeByOperation {

      public m1(a) {
            a.put("1","1"); //error
      }

      public m2(b) {
            b.toString(); //error
      }
}
```

Zudem wird in der Methode `m2()` für `b` außer `java.lang.Object` auch `java.lang.Integer/Long/Float/Double` inferiert. Das ist an dieser Stelle fraglich, denn obwohl diese Ersetzungen sicherlich möglich sind, so sind sie nicht zwingend erforderlich und somit falsch!

TestInferenceStdTypeByReturnType

Folgender Testcode

```
import java.util.Vector;
import java.util.ArrayList;
```

```
public class TestInferenceStdTypeByReturnType {

	public void m1(a) {
		b;
		b= new Vector<String>();
		a = b.size();
	}

	public void m2(a) {
		b;
		b= new ArrayList<String>();
		a = b.get(0);

	}
}
```

schlägt fehl, da der Compiler andere Typen inferiert, als die Erwartung definiert. Während die Methode `m1()` noch richtig inferiert wird, treten bei `m2()` Probleme auf. So inferiert der Compiler für `a`

- `? super java.lang.String`
- `java.lang.String`
- `? extends java.lang.String`

Der 1. und der 3. Typ sind schlicht falsch. Solche Typen kennt Java gar nicht. Aber auch Typ 2 ist falsch inferiert. Denn in Abhängigkeit vom Typ von `b` ist a höchstens ein `java.lang.Object`. Für `b` werden die folgenden Typen erkannt, die auch der Erwartung entsprechen:

- `java.util.ArrayList<java.lang.String>`
- `java.util.ArrayList<? extends java.lang.String>`
- `java.util.ArrayList<? super java.lang.String>`

.

Für die ersten beiden Typen wäre `a` vom Typ `java.lang.String`. Die dritte Möglichkeit für `b` erlaubt jedoch auch Objekte vom Typ `java.lang.Object` in der ArrayList, weshalb `a` höchstens vom Typ `java.lang.Object` sein kann.

3.5 Code Coverage – Analyse der Funktionstests

Für die vorgestellte Code-Coverage Messung wird das Eclipse-Plugin EclEmma [EMA08] verwendet, welches eine sehr einfach durchzuführende Messung erlaubt, die bis auf

Methodenebene heruntergebrochen werden kann.

Nun ist eine solche Messung ein guter Indikator für die Codeabdeckung der Testsuite, nicht zuletzt zeigt sie aber auch wie gut der Code gewartet wurde. Somit sind die Ergebnisse nicht immer leicht zu interpretieren. Im folgenden soll dies dennoch versucht werden.

Die durchgeführte Messung wurde mit der `AllTests.java` gemacht. Diese Klasse ist der Einstiegspunkt für alle verfügbaren Testfälle und schachtelt wie bereits beschrieben die Testsuites der Testgruppen.

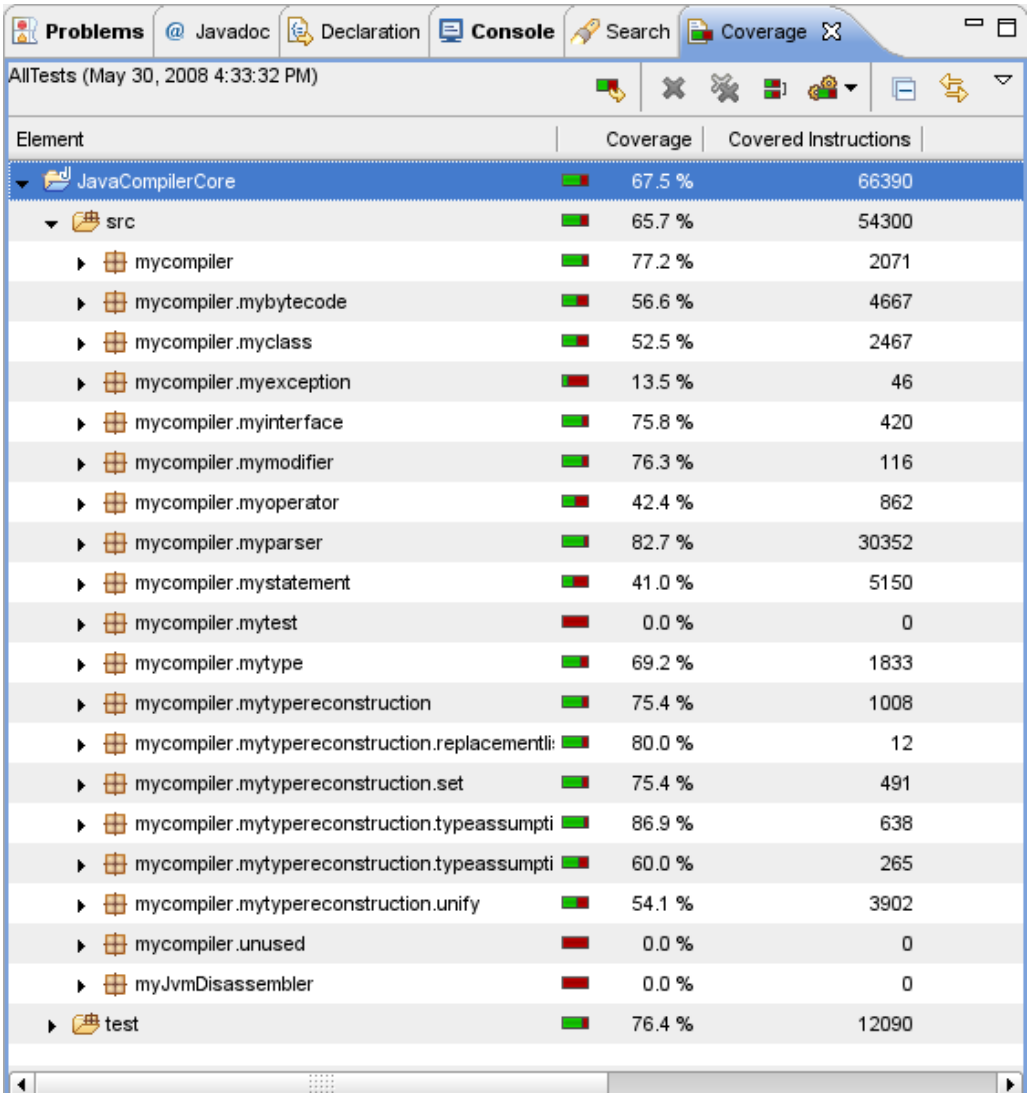

Illustration 14: Code Coverage Messung bei Ausführung aller Tests (AllTests.java)

Die angegebene Code Coverage ist zunächst durchaus sehr positiv für Projekte dieser

Größenordnung. 65,7%, wobei man von vorn herein nicht getesteten Code wie `mycompiler.mytest`, `mycompiler.unused`, `mycompiler.myJvmDisassembler` noch herausrechnen müsste, sprechen für eine relativ gute Testabdeckung. Bei näherer Betrachtung lässt sich das Ergebnis nochmal aufwerten. So lag der Fokus der Funktionstests in erster Linie auf dem Testen der Typrekonstruktion. Hiervon betroffene Codestücke erzielen ein durchaus höheren Messwert. Sich an dieser Stelle auf konkrete Zahlenwerte zu beziehen erscheint wenig Sinn zu machen. Gerade Klassen wie die `Unify.java` enthalten Methoden, die als deprecated zu markieren wären und somit das Ergebnis in Einzelfällen nach unten ziehen. Oftmals werden ganze Klassen gar nicht angefasst. Eine Code Coverage von 0% ist ein Indikator dafür, dass die Klassen eventuell in das mycompiler.unused Package verschoben werden sollten. Somit ist die Messung deutlich nur als Durchschnittsindikator zu sehen, die eben neben den Testcases selbst auch helfen kann, toten Code zu finden und Codeverbesserungen vorzunehmen.

Ein weitere Punkt ist die bereits mehrmals erwähnt Codegenerierung. Die große Anzahl an anzutreffenden Fehlern führen oftmals zu frühzeitigem Abbruch der Bytecodeerzeugung, wodurch viele Programmteile niemals angefasst werden.

Das Ergebnis der Code-Coverage Messung kann als erster Anhaltspunkt für die Testabdeckung gewählt werden. Nichtsdestotrotz ist eine hohe Codeabdeckung bei Funktionstests kein alleiniges Merkmal für einen nahezu vollständigen Test, wie es etwa bei Unit-Tests der Fall ist.

4 Überführung in ein Open-Source-Projekt

Nach dem Testen und der damit verbundenen Fehlerbehebung wäre der nächste vorstellbare Schritt die Umwandlung und Überführung der Projekte in ein Open-Source-Projekt um die Weiterentwicklung dynamischer zu machen und die Bekanntheit der Idee zu verbreiten. Um die Projekte rund um den Typinferenzalgorithmus dieser Verwandlung zu unterziehen, müssen verschiedene Schritte unternommen werden. Zuerst muss analysiert werden, welche formalen Bedingungen die Projekte erfüllen müssten um als Open Source Projekte akzeptiert zu werden. Darauf aufbauend sollten die Aspekte betrachtet werden, die ein erfolgreiches Open Source Projekt ausmachen. Unter Beachtung dieser Erkenntnisse sollten dann der aktuelle Stand der Projekte analysiert werden und konkrete Maßnahmen zur Transformation in ein Open Source Projekt getroffen werden.

4.1 Formale Bedingungen

Um als ein „echtes“ Open Source Projekt zu gelten muss das Projekt einer von der Open Source Initiative zertifizierten Open Source Lizenz unterliegen. Hierbei ist es egal um welche der vielen OSI-Lizenzen wie z.B. GPL, LGPL, BSD oder der Mozilla Public License es sich handelt. Diese Lizenzen unterscheiden sich nur in geringen Punkten. Sie stimmen in den wesentlichen Aussagen über die Verwendung und Weitergabe der lizenzierten Software überein. Hieraus ergeben sich auch die wichtigsten Vorgaben, die für eine OSI Lizenz gelten[1]:

1. Freie Wahl der Weitergabe
2. Freie Verfügbarkeit des Quelltextes
3. Modifikationen des Quelltextes müssen erlaubt sein.

Im Gegensatz zur engeren Definition der Free Software Foundation (FSF) ist es nicht zwingend notwendig, dass abgeleitete Werke unter die gleiche Lizenz gestellt werden müssen (z.B. wie bei der GPL). Sie erlaubt es allerdings. Deswegen sind die Lizenzen, die von der FSF zertifiziert sind, eine Teilmenge der OSI Lizenzen.

1 The Open Source Definition | Open Source Initiative http://opensource.org/docs/osd 17.05.2008

Darüber hinaus ist es entgegen der weitläufigen Meinung möglich, für die Weitergabe Geld zu verlangen. Dies schließt allerdings nicht den Quelltext des Programms ein. Dieser muss immer frei zur Verfügung stehen. Der Quelltext sollte zudem auch verständlich sein. Ein freier obfuskierter Quelltext würde nicht als Open Source bezeichnet werden.

Die formalen Bedingungen begrenzen sich also eher auf die Lizenz, unter die das Projekt gestellt wird. Es sollten allerdings auch die Gepflogenheiten der Open Source Community in Betracht gezogen werden. Diese Gepflogenheiten werden im nächsten Kapitel besprochen.

4.2 Eigenschaften erfolgreicher Open Source Projekte

Es gibt immer wieder Diskussionen darüber welche Eigenschaften ein Open Source Projekt besonders erfolgreich machen. Hierüber kann man wohl keine allgemeine Aussage treffen, da diese auf die Komplexität und Verschiedenheit von Open Source Projekten des Öfteren nicht zutreffen würde. In diesem Abschnitt wird versucht eine möglichst allgemeine Aussage zu treffen. Es wird allerdings kein Anspruch darauf erhoben, dass jene Aussage in jedem Fall zutrifft.

Um die These zu erarbeiten muss zunächst auf die übliche Entwicklerstruktur in einem Open Source Projekt eingegangen werden.

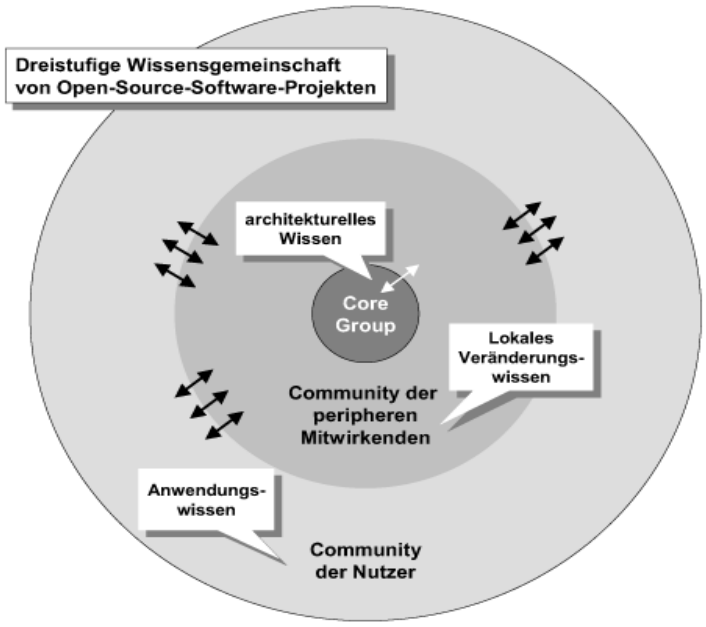

Illustration 15: Darstellung der verschiedenen Ringe von Entwicklern eines Open Source Projektes

Zum einen wäre da der Kern der Entwickler. Diese Gruppe der Entwickler treibt aktiv die Entwicklung des Open Source Projektes. Sie besteht im Normalfall nur aus wenigen Entwicklern, welche sich fast tagtäglich mit dem Projekt beschäftigen. Als nächste Gruppe kommen die Entwickler, welche sich sporadisch mit der Fortführung des Projektes beschäftigen und Patches an die Gruppe der Kernentwickler weitergeben. Diese Gruppe ist in einem erfolgreichen Projekt normalerweise um ein vielfaches Größer als die Kerngruppe. Die letzte, für den Entwicklungsprozess auf den ersten Blick unscheinbare, Gruppe ist die Gruppe der Benutzer. Diese Gruppe spielt im Open Source Entwicklungsprozess eine große Rolle, denn die Mitglieder dieser Gruppe benutzen die Software im besten Fall tagtäglich und können Rückmeldungen über Fehler oder fehlende Funktionen an die Entwickler geben.

Aus diesem Entwicklungsmodell ergeben sich schon einige Herausforderungen für ein Open Source Projekt.

Das Projekt sollte das Ziel haben eine möglichst große Benutzerzahl in seiner Zielgruppe zu erreichen. Nur mit der Unterstützung einer Community wird es möglich sein die Vorteile einer Open Source gestützten Entwicklung zu erreichen. Damit möglichst große Teile der Zielgruppe über das Open Source Projekt erfahren, bedarf es einiger weiterer Schritte. Das Projekt sollte zum Zeitpunkt der Veröffentlichung schon einen Status erreicht haben, der es den Nutzern erlaubt ein wenig mit dem Produkt zu experimentieren. Darüber hinaus muss sichergestellt werden, dass mit einem guten Marketing auch die Zielgruppe erreicht wird und über die Vorteile der Verwendung des neuen Produktes aufgeklärt wird.

Den verschiedenen Entwicklergruppen sollte es möglichst einfach gemacht werden Anregungen und Rückmeldungen wie z.B. Feature-Requests, Bug Reports an die richtigen Positionen weiterzuleiten. Die Voraussetzung für die effiziente Verarbeitung dieser Informationen ist eine ordentliche Organisationsstruktur des Projektes. In vielen Open Source Projekten wird dies durch eine kleine Gruppe von Entwicklern als Entscheidungsträger und dem Einsatz von Kollaborationssoftware erreicht. Bei der Verwendung dieser Kommunikationsstruktur sollte der Respekt gegenüber den Community Mitgliedern gewahrt werden. Rückmeldungen sollten ernst genommen werden und ausführlich diskutiert werden bevor sie in ein Projekt eingehen oder eben nicht. Durch ein falsches Kommunikationsverhalten kann einem Open Source Projekt die

sorgfältig aufgebaute Benutzerstruktur schnell wegfallen. Dies würde jedoch zu einer langsameren Entwicklung des Projektes führen.

Zusammenfassend kann man sagen, dass ein Open Source Projekt erfolgreich sein kann wenn folgende Tatsachen beachtet werden:

- Die Anzahl der Benutzer ist entscheidend für den Erfolg eines Projektes.
- Die Kommunikation zwischen den verschiedenen Entwicklergruppen sollte geregelt sein.
- Das Projekt sollte bei Einführung schon einen angemessenen Funktionumfang besitzen um Benutzer an Experimenten zu interessieren.

4.3 Maßnahmen zur Transformation

Diese Eigenschaften geben einige Schritte vor, die für eine erfolgreiche Transformation in ein Open Source Projekt hilfreich sein könnten.

Der erste Schritt wäre das Erstellen einer stabilen Version der Projekte rund um den Typinferenzalgorithmus. Diese stabile Version sollte in weiten Teilen über Tests abgedeckt sein. Außerdem sollte sie sich einfach benutzen lassen, damit es neuen Benutzern möglichst einfach wird, sich mit dem Projekt auseinander zusetzen. Hierfür wäre zum Beispiel noch eine stabile und relativ fehlerfreie Implementierung der Bytecode-Erzeugung notwendig. Um einen Benutzer zu ermutigen aktiv am Projekt mitzuentwickeln, sollten zudem auch die Hürden zum Entwickeln neuen Codes herabgesetzt werden. Die Projekte haben wie bereits bei der Projektgeschichte beschrieben eine lange Geschichte und durch die vielen mitwirkenden Studenten einen für Neueinsteiger schwer lesbaren Quelltext. Dieser sollte noch einmal analysiert und überarbeitet werden um Interessierten diesen leichten Einstieg zu ermöglichen.

Nach oder während der Stabilisierung sollte man sich Gedanken über die Open Source Lizenz machen unter der man den Quelltext zur Verfügung stellt. Die Lizenzen unterscheiden sich untereinander in einigen Bereichen doch etwas und man sollte die für seine Ziele geeignete Lizenz auswählen.

Nachdem der Quelltext und damit die Projekte unter einer entsprechenden Lizenz gestellt wurden, kann über eine geeignete Organisationsstruktur entschieden werden. Am Anfang wird diese sicherlich sehr informal sein. Eine einfache Organisationsstruktur genügt

einem Open Source Projekt im Anfangsstadium ohne große Benutzergruppen sicherlich. Im Laufe der Projektentwicklung sollte aber darauf geachtet werden, dass diese sich mit dem Projekt und dessen Bedürfnisse weiterentwickelt.

Der letzte Schritt ist wohl einer der wichtigsten für die Verbreitung und Präsenz des Projektes: Es müssen geeignete Kommunikationsplattformen für die Community hergestellt und aufgesetzt werden. Dies beinhaltet Systeme zum Zugriff auf aktuelle Quellcodeversionen (Code Versionsverwaltungssystems) als auch geeignete Lösungen zum Erhalten von Rückmeldungen der Nutzer. Hier kann auf Lösungen wie z.B. Sourceforge[2] oder Launchpad[3] zurückgegriffen werden, welche Open Source Projekten eine umfassende Suite an Werkzeugen an die Hand geben. Dazu zählen CVS/Subversion-Server (Code Hosting), Dokumentationsplatform, Bug Reporting, Mailing Lists und vieles mehr. Zu der Veröffentlichung und Publikation des Projektes zählt allerdings auch die Vermarktung des Projektes in der Zielgruppe. Die Zielgruppe muss auf das Projekt aufmerksam gemacht werden. Die Vorteile der Projekte müssen ausgearbeitet und kommuniziert werden. Ohne eine solide Marketingstrategie wird es ein Projekt schwer haben, genügend Nutzer begeistern zu können um die Entwicklung schneller voranzutreiben.

Ein weiterer Schritt zur Erweiterung der Zielgruppe wäre die Internationalisierung des Projektes. Bisher sind viele Teile der Projekte in Deutsch dokumentiert. Dies reduziert die möglichen Entwickler auf die deutsch-sprechenden Menschen ein. Um auch andere Entwickler zu erreichen, wäre eine Internationalisierung der Dokumentation und des Quelltextes sinnvoll.

2 http://sourceforge.net
3 https://launchpad.net

5 Fazit und Ausblick

> „Program testing can be quite effective for showing the presence of bugs, but it is hopelessly inadequate for showing their absence“ - Edsgar W. Dijkstra

Schon sehr einfache Programme benötigen für einen vollständigen bzw. erschöpfenden Test bereits eine immens hohe Anzahl an Testfällen. Ein angemessener Test ist immer nur eine Stichprobe.

Der in dieser Arbeit beschriebene Ansatz zum umfangreichen Testen der Funktionsweise des Typinferenzalgorithmus ist eine gute Grundlage um aufbauend einen beständigen Entwicklungsprozess zu fördern. Die Funktionsweise kann jederzeit anhand der Testsuite verifiziert werden.

Die Tests haben gezeigt, dass der Algorithmus bis auf wenige Ausnahmen sehr solide funktioniert. Größere Probleme traten zumeist im Parser bzw. vor allem bei der Codegenerierung auf. Hier muss ausführlich nachgebessert werden. Die Java-Zugriffskonzepte sind noch nicht implementiert. Diese sind jedoch auch in Hinsicht auf den Typinferenzalgorithmus interessant. Zudem ist die Implementierung sämtlicher bekannter Grundtypen aus der Java-Programmiersprache von Interesse für eine tatsächliche Verwendung eines solchen Compilers, obwohl sie vielleicht in dem hier vorliegenden Prototypen eher eine untergeordnete Rolle spielen.

Das Testframework kann als Grundlage für weitere Tests herhalten und die Kategorien können jederzeit weitere Testfälle aufnehmen um die Vollständigkeit der Funktionstests noch zu erhöhen. Dies ist insbesondere in Hinblick auf die Weiterentwicklung wichtig, da nun zeitgleich auch Funktionstests ohne größeren Aufwand geschrieben werden und so die Testsuite beständig wachsen kann. Auch ist dies ein Schritt hin zu einem Open-Source-Projekt.

Die zahlreichen Probleme beim Parser bzw. insbesondere bei der Coderzeugung müssen jedoch noch einmal gezielt adressiert werden. Hier lässt sich auch die Idee der Verwendung von Standardtools als Rahmen für den Typinferenzalgorithmus einbringen, die eine Weiterentwicklung als Open Source Projekt allein auf die Typinferenz konzentrieren lassen.

6 Literaturverzeichnis

[PLU05] Martin Plümicke, Type Inference in Generic Java, 2005

[LIN02] Johannes Link, Unit Tests mit Java, dpunkt.verlag 2002

[NEA03] David Neary, http://www.linux.ie/articles/tutorials/junit.php, 2003

[GAR05] JUnit antipatterns, Alex Garret, http://www.ibm.com/developerworks/opensource/library/os-junit/, 2003

[KOC03] Karola Koch, Success Factors of Open Source Projects, http://www.ilias.de/conference/2003/pdf/20-KOCH.pdf, 2003

[BOE02] Tilo Böhmann, Was Wissensmanagement von Open-Source-Software lernen kann, http://www.symposion.de/wm-ph/wm-ph_50.htm, 2002

[SUN08] Bitwise and Bit Shift Operators, Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/java/nutsandbolts/op3.html, 2008

[EMA08] Java Code Coverage for Eclipse, Marc R. Hoffmann, http://www.eclemma.org, 2008

[WIKI08] Funktionaler Systemtest, http://de.wikipedia.org/wiki/Softwaretest#Funktionaler_Systemtest, 16. 05. 2008

[REI05] Felix Reichenbach, Erweiterung der semantischen Analyse des Java-Compilers, 2005

[HAA04] Markus Haas, Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen., 2004

[OTT04] Thomas Ott, Typinferenz in Java, 2004

[BAE05] Joerg Baeuerle, Typinfernz in Java, 2005

[HOL06] Timo Holzherr, Typinferenz in Java, 2006

[LUE07] Arne Lüdtke, Java Typinferenz mit Wildcards, 2007

[BUR07] Achim Burger, Erweiterte Typinferenz in Java 5.0, 2007

[SCH07] Jürgen Schmiing, Migration einer Codegenerierungskomponente für einen Java-Compiler, 2007

[MEL05] Markus Melzer, Integration der Java-Typinferenz in eine Programmierumgebung, 2005

[HOM06] Thomas Homberger, Einbindung von Java – Typinferenz in eine integrierte Entwicklungsumgebung, 2006

7 Abbildungsverzeichnis

A Anhang

A.1 Unit-Tests wichtiger Bestandteile des Typinferenzalgorithmus

Ein funktionaler Test wie im Kapitel 2.2 beschrieben ist nicht ausreichend für eine große Testabdeckung und kann bei einer nachträglichen Bearbeitung des Codes nur in beschränktem Maße die Erhaltung der Korrektheit gewährleisten. Daher empfiehlt es sich Unit-Tests zu entwickeln. Der vorhandene Code ist durch die große Anzahl an Studienarbeiten, die sich mit dem Thema beschäftigt haben, sehr gewachsen und nicht immer im vorhandenen Umfang verwendet und benötigt. So sind die Algorithmen der Typinferenz, sicherlich in Folge von zahlreichen Veränderungen wie durch das Hinzufügen von Wildcards und Generics, in extrem große und damit unübersichtliche Funktionen verteilt. Unit-Tests können an dieser Stelle helfen, die zahlreichen Verschachtelungen innerhalb der Algorithmen zu verringern, indem sie den Status quo festhalten. Sie sichern damit die Korrektheit eines Moduls während der Veränderungsprozesse.

Obgleich dieser Vorteile stellt das Schreiben von Unit-Tests sehr hohe Anforderungen an den Testschreiber. Nicht nur muss er vollstes Code-Verständnis besitzen, er muss auch noch damit umgehen, dass der Code nicht von vornherein, wie es das Test-Driven-Development vorschreibt, als testbarer Code entwickelt wurde. Unit-Tests verlangen beispielsweise das isolierte Testen eines Moduls unabhängig von den benutzten Objekten. Die benutzten Datenstrukturen sind jedoch in der Regel – auch aufgrund der Komplexität des Problems – meist nur schwer von Hand erzeugbar. Ohne Veränderungen am Code ist die Benutzung von Mockobjekten und Stubs ebenso nicht möglich, da sie sich nicht dem existierenden Code unterschieben lassen. Auf diese Weise können Fehler bei der Erzeugung der benutzten Datenstrukturen das Testergebnis des eigentlich zu testenden Moduls verfälschen.

Verfolgt man nicht diese strikten Regeln der Unit-Tests, lassen sich dennoch Testfälle erzeugen, die ein Problem wesentlich dichter eingrenzen, als es die im Rahmen dieser Studienarbeit erstellten Funktionstests können. Im folgenden sollen einmal zwei Testfälle

vorgestellt werden um zum einen den Testnutzen, aber auch die Testprobleme anzureißen.

A.1.1 TestTrMakeFC

Dieser Testfall hat den Anspruch, die für den Typinferenzalgorithmus wichtige Funktion „makeFC()“ zu testen. Der grundsätzliche Funktionsweise wird an dieser Stelle allerdings keine Aufmerksamkeit geschenkt. Sie ist bereits in vorhergehenden Studienarbeiten beschrieben [OTT04]. Vielmehr soll es um eine beispielshafte Unit-Test-Implementierung gehen. In Anlehnung an die von [OTT04] beschriebene Funktionsweise sowie das vorhandene Ablaufdiagramm konnten zwei Testszenarien identifiziert werden.

In der `testTrivialMakeFC()` geht es um den einfachsten Fall: Wir testen einfache Klassen und Interfaces, die weder miteinander verwandt sind, noch Klassengenerics besitzen. Die Finite Closure (FC) ist dem Algorithmus entsprechend in diesem Fall eine leere Menge. Die testAllMakeFC() hingegen beschäftigt sich mit einem Szenario, das den Anspruch auf annähernd vollständige Testabdeckung hat. So wurde ein Beispiel gewählt, welches jede mögliche Abzweigung passiert und somit die komplette Implementierung testet.

```
(1) class AbstractList<A>{}
(2) class Vektor<A> extends AbstractList<A>{}
(3) class Matrix<A> extends Vektor<Vektor<A>>{}
(4) class ExMatrix<A> extends Matrix<A>{}
```

Auffällig beim Testen dieser Methode ist, dass zahlreiche Objekte verwendet werden, die sich jedoch nicht mocken lassen. Das geschieht dadurch, dass die Bestimmung der zu verwendenen Objekte nicht über Methodenparameter geschieht. D.h. man kann an dieser Stelle keine Mockobjekte unterschieben. So ist man beim Unit-Testen darauf angewiesen, dass der Aufruf von `Unify.varSubst(L1, substHash1)` beispielsweise funktioniert. Tritt hier ein Fehler auf, so wird dieser auf die getestete Methode `makeFC()` zurückgeführt, trat aber an einer ganz anderen Stelle auf. Abhilfe gegenüber Code-Veränderungen schafft hier nur das Testen der benutzten Klassen bevor dieser Testcase ausgeführt wird. Man bewegt sich dann allerdings weg vom reinen Unit-Test und schafft mit diesem Test einen Integrationstest. Dies ist ein Problem beim Unit-Testen von Code, der nicht nach der Test-Driven-Development-Idee geschrieben wurde.

Nutzt man hier eine Code-Coverage Messung, so erreichen wir einen Wert von 94,4%. Da gerade Unit-Tests ja in der Regel White-Box-Tests sind und auch dieser Test danach strebte, möglichst 100% abzudecken, liegen wir hier dennoch knapp 6% entfernt vom Ziel. Das ist dadurch zu erklären, dass der Code von `makeFC()` auch Fehlerbehandlung beinhaltet. Jedoch wurde mit dieser Testsuite die Fehlerbehandlung überhaupt nicht getestet. Darüber hinaus zeigt die Messung aber auch, wie stark andere Komponenten des Compilers benutzt wurden um eine eigentlich isoliert zu testende Methode zu überprüfen. Damit lässt sich die zuvor aufgestellte Beobachtung bestätigen, dass die Methode `makeFC()` aufgrund ihrer nicht „mock"-baren Struktur nicht zu 100% nach dem Konzept eines Unit-Tests getestet werden konnte.

A.1.2 TestTrSubUnify

Der Algorithmus der `subUnify()`-Methode wie sie hauptsächlich von Arne Lüdke nach dem Algorithmus von Martin Plümicke implementiert wurde, wurde zwar nach [LUE07] getestet, ein dauerhafter Test war jedoch nicht vorhanden. Gerade die Komplexität dieses Algorithmus sowie seiner Implementierung macht einen Unit-Test erforderlich. So wurde wiederum ein White-Box Test implementiert, der jede der 13 Regeln [PLU05] über der Methode testen soll. Im folgenden ist eine Zusammenstellung der getesteten Regeln mit dem dazugehörigen Testbeispiel aufgelistet. Es werden dabei 2 unterschiedliche Klassenstrukturen für das Testen verwendet:

(a)

```
interface A{}

class B implements A {}

class C extends B{}
```

(b)

```
interface AbstractList<A>{}

class Vektor<A> extends AbstractList<A>{}

class Matrix<A> extends Vektor<Vektor<A>>{}

class ExMatrix<A> extends Matrix<A>{}
```

Regel	Testfälle
Erase1	Nach (a): (1){ B <* A} -> {} (2){ C <* A} -> {} (3){ C <* B} -> {}
Erase2	Nach (a): (1){ B <*? extends A} -> {} (2){ C <*? super A} -> {}
Erase3	Nach (a): (1){ A = B} -> {}
Swap	Nach (a): (1){ A = TypePlaceholder a} -> {a = A} A: RefType, GenericTypeVar, any WildcardType
ReduceLow	Nach (a): (1){?extends A < B} -> {A < B}
ReduceUp	Nach (a): (1){A <? super B} -> {A < B}
ReduceUpLow	Nach (a): (1){?extends A <? super B} -> {A < B}
Reduce1	Nach (b): (1){Vektor<a> < AbstractList<b> -> {a<?b}
Reduce2*	(1){A<a,b,c> = A<d,e,f> -> {a=d,b=e,c=f}
ReduceExt	Nach (b): (1){Vektor<a> <? ?extends AbstractList<b> -> {a<?b} (2){?extends Vektor<a> <? ?extends AbstractList<b> -> {a<?b}
ReduceSup	Nach (b): (1){Vektor<a> <? ?super AbstractList<b> -> {a<?b} (2){?super Vektor<a> <? ?super AbstractList<b> -> {a<?b}
ReduceEq	(1){Vektor<a,b,c> <? Vektor<d,e,f> -> {a=d, b=e, c=f}

Anhand dieses Tests lässt sich ein weiterer Vorteil von Unit-Tests zeigen. Wie auch die Code-Coverage Messung bestätigt, wird für die Regel Reduce2, für die es im Code auch einen ganzen Zweig gibt, niemals angesprungen. Somit zeigt der Unit-Tests eventuelle Fehler wie auch in erster Linie Codeduplikationen bzw. unnötigen Code auf. Reduce2 hat die Voraussetzung, dass es sich bei der Unifikation um den gleichen Typen mit unterschiedlichen Parametern und einem „=“-Vergleich handelt. Genau dieser Fall wird auch bei Erase3 angesprungen

```
if( P.isEqual() && P.OperatorEqual() ) //Erase3
                {
                  ...
```

```
}
```

Der Operator ist „=“ und `P.isEqual()` liefert `true`, da beide Typen sowohl den gleichen Namen, als auch die gleiche Anzahl an generischen Parametern besitzen. Die Bedingung, für Reduce2, die in der `subUnfiy()`-Methode erst später getestet wird, wird somit niemals angesprungen.

Die Code-Coverage Messung zeigt auch hier ein gutes Ergebnis von 80,6%. Wiederum sind es in erster Linie die Fehlerfälle, die nicht getestet wurden.

A.1.3 Fazit

Natürlich handelt es sich bei dem zugrunde liegenden Compiler mit Typinferenzalgorithmus um einen ersten Prototypen. Für eine eventuellen Überführung in ein echtes Produkt bedarf dieser jedoch einiger gravierender Codeveränderungen. So sollte gerade wegen der Komplexität des Themas, durchaus auch aus Performanzgründen, ein leicht wartbarer und vor allem testbarer Code geschrieben werden, der womöglich sogar nach dem Test-First-Ansatz entwickelt werden könnte. Die getätigten Unit-Tests erheben keinen Anspruch auf Vollständigkeit, zumal die Fehlerfälle nicht getestet wurden. Dies hat vor allem damit zu tun, dass der Fokus dieser Arbeit letztendendes nicht auf den Unit-Tests, sondern auf Funktionstests lag. Der Abschnitt zu den Unit-Tests stellt lediglich eine kurze Vorstellung der Möglichkeiten bzw. der Vorteile und Probleme von Unit-Tests für den bestehenden Compilers dar.