Compilezeit-Prüfung von Spring-Konfigurationen

Vincent von Hof*
von-hof@wi.uni-muenster.de

Konrad Fögen**
foegen@swc.rwth-aachen.de

Herbert Kuchen*
kuchen@uni-muenster.de

* Universität Münster, ERCIS
** RWTH Aachen, SWC group

Zusammenfassung

Dependency Injection Frameworks wie das Spring Framework verlassen sich auf dynamische Sprachfähigkeiten von Java. Sofern diese Fähigkeiten auf unvorhergesehene Art und Weise eingesetzt werden, können Fehler auftreten, die zur Übersetzungszeit nicht vom Java-Compiler erkannt werden. Diese Arbeit diskutiert die Anwendung von statischer Programmcode-Analyse als Mittel, besagte Übersetzungszeit-Prüfungen wiederherzustellen. Zuerst werden mögliche Fehler in der Konfiguration von Spring identifiziert und klassifiziert. Attributierte Grammatiken werden benutzt, um auf formale Art und Weise Fehler festzustellen. Anschließend wird eine prototypische Compiler-Erweiterung basierend auf der Java Pluggable Annotation Processing API vorgestellt.

1 Einführung

Java ist eine der beliebtesten Programmiersprachen, insbesondere im Hinblick auf Enterprise Applications, wie verschiedene Indizes belegen[1]. Ferner ist *Dependency Injection* (DI) eine häufig eingesetzte Technik, um die Entwicklung von Java-Anwendungen zu unterstützen und zu vereinfachen. DI gehört zur Gruppe der *Erzeugungsmuster*, welche den Prozess der Objekt-Erzeugung und -Komposition abstrahieren [8, p.94]. Typischerweise sind DI-Implementierungen generisch und treffen keinerlei Annahmen über die Objekte, welche sie verwalten. Stattdessen verlassen sie sich auf eine externe Konfiguration [19, p.17].

Allerdings macht es die Natur der generischen Implementierung erforderlich, dass dynamische Sprach-Features wie die *Java Reflection API* [15] eingesetzt werden. Trotz der Nützlichkeit der Java Reflection API bringt deren Verwendung das Problem mit sich, dass Fehler, welche durch eine unvorhergesehene Verwendung auftreten, sich nicht allein durch die statische Programmcode-Analyse des Java-Compilers aufspüren lassen. Daraus folgt, dass sich die Erkennung von Fehlern von der Übersetzungszeit auf die Installationszeit, d.h. die Zeit, zu welcher das Programm auf dem Anwendungsserver installiert (*deployed*) wird, oder die Laufzeit verschiebt.

Da die Fehler nicht automatisch zur Übersetzungszeit erkannt werden und eine händische Überprüfung zeitlich aufwendig ist, haben speziell Entwickler ein Interesse an automatisierten Lösungen, die eine Erkennung so früh wie möglich zulassen, d.h. im besten Falle schon zur Übersetzungszeit während der Entwicklung.

Der im Folgenden präsentierte Ansatz beschäftigt sich mit einer Compiler-Erweiterung basierend auf attributierten Grammatiken [11] mit dem Ziel, zur Übersetzungszeit auftretende Fehlerzustände zu entdecken, welche sich durch unvorhergesehene Nutzung des *Spring Frameworks* [17] ergeben.

Spring wurde auf Grund seiner Popularität und praktischen Relevanz aus der Menge der verschiedene DI-Implementierungen für Java als Repräsentant gewählt. Des Weiteren basiert die Konfiguration von Spring auf Java-Annotationen, was es für die Bearbeitung durch die Pluggable Annotation API prädestiniert. Prinzipiell ist es möglich, diesen Ansatz auch für andere DI-Frameworks umzusetzen.

[1] Siehe z.B. http://lang-index.sourceforge.net

		Spring Context	
		Abhängig	**Unabhängig**
Analyse Level	**Oberhalb des Methoden-Levels**	Gruppe I (13 Fehler)	Gruppe II (13 Fehler)
	Unterhalb des Methoden-Levels	Gruppe III (8 Fehler)	Gruppe IV (4 Fehler)

Tabelle 1: Klassifikation von Konfigurationsfehlern.

Es existieren bereits einige Tools für *Statische Codeanalyse* von Java-Programmen wie z.B. FindBugs [2], Checkstyle [4], PMD [18], SonarQube [21], Java Language Extender [23], ESC/Java2 [5], JastAddJ [7], JavaCOP [13], JQual [9] und das Checker framework [16]. Alle diese Tools sind als generelle Inspektionswerkzeuge konzipiert. Nach unserem Wissen existiert kein Tool, welches speziell die statische Überprüfung von Fehlern bei der Konfiguration von Spring ermöglicht.

Diese Arbeit ist wie folgt strukturiert. In Abschnitt 2 werden mögliche Fehler bei der Spring-Konfiguration aufgezeigt und klassifiziert. In Abschnitt 3 werden daraufhin attributierte Grammatiken vorgestellt, welche formal die Erkennung der Spring-Konfigurationsfehler beschreiben. Ausgehend von diesen attributierten Grammatiken wird in Abschnitt 4 ein Prototyp einer Compiler-Erweiterung basierend auf Java's Pluggable Annotation Processing API beschrieben. Die durch die Entwicklung des Prototypen gewonnenen Erkenntnisse werden in Abschnitt 5 genutzt, um den Ansatz zu bewerten. In Abschnitt 6 wird ein Fazit gezogen und es werden zukünftige Betätigungsfelder aufgezeigt.

2 Spring Konfigurationsfehler

Das Spring Framework ist ein Open Source Framework, welches das Dependency-Injection-Entwurfsmuster implementiert. Im Kern stellt die *Spring Context* Komponente nachgefragte und abhängige Objekte bereit. Diese Objekte, so genannte *Spring Beans*, können jegliche Art von einfachen Plain Old Java Objects (POJOs) sein [24, p.4].

Verschiedene Implementierungen für den *Spring*-Kontext existieren, welche sich primär in der Art ihrer Konfiguration unterscheiden, basierend auf z.B. Java oder Extensible Markup Language (XML). Im Allgemeinen bestehen Konfigurationen sowohl aus Merkmalen, die sich direkt auf den *Spring*-Kontext beziehen, als auch aus Definitionen für die *Spring Beans*. Spring bietet dabei drei verschiedene Möglichkeiten, *Spring Beans* zu definieren: Explizite Konfiguration via Java und XML und/oder implizite Konfigurationen via Java Annotationen. Diese Arbeit konzentriert sich auf annotations-basierte Konfigurationen. Für weitere Informationen zu Spring siehe [24], [19] oder [10].

Um mögliche Fehlertypen zu identifizieren, wurde zuerst ein Literaturreview der Spring-Referenzdokumentation [10] durchgeführt, gefolgt von Experteninterviews mit Entwicklern eines Software-Unternehmens. In dieser Arbeit werden *Core Container*, für DI, und Datenzugriffs- und Integrations-Module betrachtet, da diese in jeder Spring-basierten Java-Anwendung zur Verfügung stehen.

Schlussendlich ergaben sich 38 unterscheidbare Fehlertypen. Diese Fehlertypen sind in Tabelle 1 in vier Gruppen klassifiziert. Die vier Gruppen ergeben sich aus zwei Dimensionen, welche je zwei Ausprägungen annehmen können. Die Klassifizierung hängt dabei einerseits von den wichtigsten Features des Spring-Frameworks ab. Die Verwendung bestimmter Features bestimmt die Zuordnung eines Fehlers zu einer bestimmten Dimension. Eine *Spring Context* benannte Dimension bestimmt, ob die Analyse Informationen benötigt, die aus dem *Spring Context* abgeleitet werden. *Analysis Level* ist die zweite Dimension und sie bestimmt, ob für eine Analyse Informationen bezüglich des Kontrollflusses von Sprachkonstrukten unterhalb des Methoden-Levels benötigt werden, z.B. Methodenaufrufe oder Variablen-Zuweisungen.

Im Folgenden werden mögliche Fehler beschrieben, um die Fehlertypen zu veranschaulichen.

Gruppe I

In diese Gruppe lassen sich Fehler einordnen, die vom *Spring*-Kontext und mindestens einer weiteren Komponente, welche eine Spring-spezifische Annotation benutzt, abhängen. Diese Fehler treten *oberhalb* des Methoden-Levels auf, z.B. bei Deklarationen von Klassen, Methoden oder Attributen. Für sich betrachtet sind dabei die *Spring Context* Konfiguration und die weitere Komponente jeweils sogar korrekt, allerdings ist zumindest die Interaktion der beiden Elemente fehlerbehaftet.

Als Beispiel sei Spring's Transaktions-Infrastruktur genannt. Sie kapselt spezifische Transaktions-Management APIs und bietet ein deklaratives Modell zur Integration in Anwendungen [10, chap.12.3]. Sowohl ein *Spring*

Context, der einen Transaktions-Manager deklariert, als auch eine *@EnableTransactionManagement* Annotation müssen vorhanden sein, um das Transaktions-Management benutzen zu können. Sobald dies geschehen ist, lassen sich Methoden per *@Transactional* annotieren, um Transaktions-Management-Funktionalität für diese Methode zu aktivieren. Die korrekte Verwendung ist in Listing 1 dargestellt.

```
1  @Configuration
2  @EnableTransactionManagement
3  public class SpringConfig {
4    @Bean
5    public PlatformTransactionManager transactionManager() {...}
6  }
7
8  @Component
9  public class PrintService {
10   @Transactional
11   public void print() {...}
12 }
```

Listing 1: Darstellung von Spring's Transaktionsmanagement.

In dieser Situation kann dann ein Fehler auftreten, wenn das Transaktionsmanagement aktiviert ist, jedoch nicht verwendet wird, da keinerlei Methoden mit *@Transactional* versehen sind, z.B. wenn in Listing 1 die Zeile 10 fehlen würde. Auch kann die umgekehrte Situation eintreten und zu einem Fehlerzustand führen, wenn per *@Transactional* annotierte Methoden existieren und gleichzeitig das Transaktionsmanagement deaktiviert ist, weil z.B. Zeile 2 vergessen wurde.

Gruppe II

Fehler in Gruppe II sind unabhängig vom Spring Context und treten oberhalb des Methoden-Levels auf. Sie umfassen Sprachkonstrukte mit Annotation, die gleichzeitig weitere Attribute oder Annotationen aufweisen, die inkompatibel mit der ersten Annotation sind.

Zum Beispiel benutzt das Spring Framework die *@Autowired* Annotation, um Methoden oder Felder zu markieren, die als *Injection Point* in Frage kommen [24, p.39]. Zum Teil können Abhängigkeiten auch mehrdeutig sein, sodass der Spring Context mehrere Bean Definitionen vorfindet, die einem Injection Point genügen. In dem Falle wird eine der passenden Beans ausgewählt [24, p.75]. Die *@Qualifier* Annotation kann im Zusammenspiel mit *@Autowired* genutzt werden, um die Ergebnismenge einzuschränken. Allerdings resultiert es in einem Fehler, *@Qualifier* ohne eine entsprechende *@Autowired* Annotation zu verwenden.

Des Weiteren kann die *@Qualifier* Annotation auch indirekt genutzt werden. Es macht keinen Unterschied, ob *@Qualifier* direkt oder aber eine Annotations-Klasse verwendet wird, welche wiederum mit *@Qualifier* versehen ist. Sowohl ein entsprechender Fehler als auch die indirekte Nutzung von *@Qualifier* sind in Listing 2 dargestellt.

```
1  @Qualifier
2  @interface DinA4Format {...}
3
4  @Component
5  @DinA4Format
6  class DinA4DocFormatter implements DocFormatter {...}
7
8  @Component
9  class PrintService {
10   //@Autowired  fehlt
11   @DinA4Format
12   public PrintService(DocFormatter f) {...}
13 }
```

Listing 2: Verwendung von @Qualifier ohne @Autowired.

Gruppe III

Wie Fehler aus Gruppe I beruhen auch Fehler der Gruppe III auf speziellen Spring Context Konfigurationen. Allerdings hängen sie zusätzlich von Sprachkonstrukten *unterhalb* des Methoden-Levels ab. Zum Beispiel wird der Lebenszyklus einer Bean über eine *Bean Definition* geregelt. Er beginnt, nachdem der zugehörige Spring-Kontext initialisiert wurde, und endet, wenn besagter Kontext abgeschaltet wird. Innerhalb dieses Zeitraumes wird der

Lebenszyklus einer Bean über einen *Scope*, d.h. einen Gültigkeitsbereich, definiert [24, p.81]. Standardmäßig, d.h. ohne weitere Konfiguration, befindet sich eine Bean im *Singleton* Scope. In diesem Scope existiert eine einzige, gemeinsam genutzte Instanz der Bean (pro Spring-Kontext) und diese existiert, bis der Spring-Kontext beendet wird [10, chap. 5.5.1]. Im Gegensatz dazu werden Beans, die mit dem *Prototype* Scope definiert sind, bei jeder Anfrage neu erstellt.

Ein wichtiger Aspekt bezüglich des Prototype Scope ist, dass der Spring-Kontext den Lebenszyklus solcher Beans nicht überwacht. Obwohl Prototype und Singleton Beans auf gleiche Art und Weise initialisiert werden, ist ihre Zerstörung unterschiedlich. Da der Spring Context keine Referenzen auf Prototype Beans speichert, kann er trivialerweise auch ihre Zerstörung nicht steuern.

Eine Spring-Bean-Definition gilt als entweder explizit definiert, wenn eine Methode direkt per *@Bean* Annotation versehen wird, oder sie ist impliziert definiert, wenn eine Klasse direkt per *@Component* annotiert wird. Des Weiteren erlaubt es Spring sogenannte *Lifecycle Callbacks* zu definieren, wodurch Methodenaufrufe durch den Spring-Kontext zu bestimmten Phasen des Lebenszyklus einer Bean ausgelöst werden [24, p.33]. Unter anderem existieren hierzu Methoden-Annotationen für die Lebenszyklus-Events *@PostConstruct* und *@PreDestroy*, d.h. Methoden, die vor der Erstellung respektive Zerstörung einer Bean ausgeführt werden sollen.

Da der Spring-Kontext allerdings, wie beschrieben, nicht über die Zerstörung von Prototype Beans wacht, müssen Annotationen, welche mit dem Lifecycle Event der Zerstörung in Verbindung stehen, für Prototype Beans als fehlerhaft angesehen werden. Listing 3 illustriert diesen Fehler, da dort eine Komponente eine *close* Methode besitzt, die zum Freisetzen von Ressourcen dient, allerdings auf Grund des gesetzten Prototype Scopes niemals zur Ausführung kommen wird.

```
1  @Configuration
2  public class SpringConfig {
3    @Bean
4    @Scope("prototype")
5    public PrintService printService() {
6      return new PrintService();
7    }
8  }
9
10 public class PrintService {
11   @PreDestroy
12   public void close() {  // nicht aufgerufen!
13     this.usbConnection.close();
14   }
15 }
```

Listing 3: Lebenszyklus von Beans mit *prototype scope*.

Gruppe IV

Gruppe IV beinhaltet Fehler, die unterhalb des Methoden-Levels auftreten und nicht vom Spring Context abhängen. Ein Beispiel lässt sich bei Spring's *JdbcTemplate* Komponente finden, welche eine Abstraktionsebene über Java's JDBC API darstellt. In diesem Beispiel wird SQL genutzt, um mit einer Datenbank zu interagieren. Der korrespondierende Code ist sprachlich gesehen von dem ihn umgebenden Java-Code getrennt. Der Java-Compiler kann nicht überprüfen, inwiefern der verwendete SQL-Code den Sprachdefinitionen von SQL genügt. Der klassische Anwendungsfall hierfür involviert einen Entwickler, der seine SQL-Statements händisch mit einem Tool gegen eine Datenbank testet, um den SQL-Code schlussendlich in Java als String zu hinterlegen. Eine einfache Restriktion in diesem Kontext ist die Notwendigkeit, das SQL-Statement im Tool mit einem Semikolon zu beenden. Ein Semikolon am Ende eines SQL-Strings hat in Java allerdings einen Laufzeitfehler zur Folge.

Diese Art von Problemen lässt sich erkennen, indem *Pluggable Type Systems* ähnlich dem Checker-Framework für reguläre Ausdrücke verwendet werden [22]. Die Diskussion dieses Fehlertyps ist nicht Bestandteil dieser Arbeit.

3 Fehlererkennung mit Hilfe von attributierten Grammatiken

Knuth führte 1968 mit *Attributierten Grammatiken* eine formale Herangehensweise zur Beschreibung und Handhabung von semantischen Aspekten von Programmiersprachen ein [11]. Attributierte Grammatiken sind kontextfreie Grammatiken, die mit Attributen und semantischen Regel erweitert werden [20, pp.66-67]. Jedes Nichtterminal einer kontextfreien Grammatik kann mehrere Attribute besitzen. Jedes Attribut kann entweder *synthetisiert*

(synthesized) oder *vererbt* (inherited) sein und besitzt einen Wert, der durch eine semantische Regel definiert wird, die einer Regel einer kontextfreien Grammatik zugeordnet wird.

Sei $A ::= B_1 \dots B_n$ (für $n \in \mathbb{N}$) eine kontextfreie Regel mit den Nichtterminalen $A, B_1, \dots, B_n$, welche jeweils ein synthetisiertes Attribut s und ein vererbtes Attribut i besitzen, so gilt, dass die korrespondieren semantischen Regeln die Werte der Attribute $A.s, B_1.i, \dots, B_n.i$ wie folgt definieren:

$$A.s \leftarrow f(A.i, B_1.s, \dots, B_n.s) \quad (1)$$

$$B_j.i \leftarrow g(A.i, B_1.s, \dots, B_{j-1}.s, B_{j+1}.s, \dots, B_n.s) \quad (2)$$

wobei f und g Funktionen sind, die Attributwerte auf andere Attributwerte abbilden, und $j \in \{1, \dots, n\}$. Wenn die kontextfreien Regeln Terminale enthalten und/oder wenn Nichtterminale mehrere synthetisierte und vererbte Attribute besitzen, so müssen die Formeln (1) und (2) entsprechend verallgemeinert werden (s. [1] für eine vollständige Beschreibung von attributierten Grammatiken).

Die folgende Notation wird in dieser Arbeit genutzt, um Attribute und semantische Regeln darzustellen. Semantische Regeln werden in *geschweifte Klammern* eingefasst und hinter den Rumpf der zugehörigen kontextfreien Regel geschrieben. $\$0.a$ referenziert das Attribute a des Symbols auf der linken Seite der kontextfreien Regel, $\$1.a$ bezeichnet das Attribut a des ganz links stehenden Symbols auf der rechten Seite der kontextfreien Regel und so weiter. Jede semantische Regel $a \leftarrow e$ besteht aus einem Attribut a auf der linken Seite und einem Ausdruck e bestehend aus Attributen, Operationssymbolen und Konstanten auf der rechten Seite. Sie repräsentiert eine Wertzuweisung des Werts von e an das Attribut a auf der linken Seite. Für den Ausdruck auf der rechten Seite benutzen wir eine Syntax ähnlich wie in C oder Java.

Nachdem ein Syntaxbaum (nach lexikalischer und syntaktischer Analyse) aufgebaut wurde, können die Werte der Attribute der Nichtterminalsymbole des Baumes durch das Anwenden der semantischen Regeln abgeleitet werden [1, p.54]. Die Reihenfolge, in welcher die Attribute ausgewertet werden, muss dabei den durch die semantische Regeln induzierten Abhängigkeiten folgen. Im Allgemeinen gibt es allerdings keine Garantie, dass eine Reihenfolge existiert, durch die alle Attribute von allen Knoten evaluiert werden können. Es existieren aber Klassen von attributierten Grammatiken, welche die Benutzung der Attribute und semantischen Regeln einschränken, um die Existenz einer Auswertungsreihenfolge zu garantieren [1, p.313].

Für diese Arbeit relevant sind insbesondere *S-* und *L-attributierte Grammatiken*. S-attributierte Grammatiken verwenden ausschließlich synthetisierte und keine vererbten Attribute [1, p.313]. Sie erlauben eine von unten-nach-oben (bottom-up) verlaufende Evaluation von Attributen. L-attributierte Grammatiken erlauben eine Auswertung der Attribute von links nach rechts; siehe [1] für Details. Bereits mit S-attributierten Grammatiken lassen sich alle in Programmiersprachen erforderlichen semantischen Überprüfungen durchführen. Die relevanten Informationen müssen dann ggf. bis zur Wurzel durchgereicht und dort überprüft werden. L-attributierte Grammatiken erlauben dies durch ihre größere Flexibilität manchmal etwas eleganter zu bewerkstelligen.

Im Folgenden wird eine LL(1)-konforme kontextfreie Grammatik vorgestellt, welche die Untermenge von Java Sprachkonstrukten beschreibt, die für das Erkennen von Spring-Konfigurationsfehlern relevant ist. Abbildung 1 gibt einen Überblick über die Produktionen, welche für eine erfolgreiche Analyse benötigt werden.

Die semantischen Regeln, die für die attributierten Grammatiken genutzt werden, basieren auf den folgenden Konstanten und Operationen: *error* wird genutzt, um anzuzeigen, dass ein Fehler festgestellt wurde, *emptySet* erstellt eine leere Menge, *newSet* erstellt ein Menge mit einem einzelnen Element, *intersects* gibt an, ob die Schnittmenge zweier Mengen nicht leer ist, und *union* errechnet die Vereinigungsmenge zweier Mengen. Die Funktion *value* gibt für *identifier* den korrespondierenden String zurück.
Im Folgenden wird beispielhaft eine attributierte Grammatik vorgestellt, die es erlaubt, den Fehler zu erkennen, der in Listing 1 demonstriert wurde.

Spring-Kontext-abhängige Fehler, die oberhalb des Methoden-Levels auftreten, zeichnen sich typischerweise durch die An- oder Abwesenheit spezifischer Annotationen aus (Gruppe I). Die An- oder Abwesenheit lässt sich durch zwei synthetisierte boolesche Attribute *enabled* und *used* ausdrücken.
Der Fehler, der in Listing 1 demonstriert wurde, kann wie folgt erkannt werden. Das Attribut *enabled* evaluiert zu true, sofern eine Spring-Konfiguration existiert und diese mit *@EnableTransactionManagement* annotiert ist. Das Attribut *used* evaluiert zu true, sofern eine Methode existiert, die mit *@Transactional* annotiert ist. Ein Fehler zeigt sich, wenn $enabled \wedge \neg used$ wahr ist.
Die Erkennung lässt sich durch eine S-attributierte Grammatik mit vier synthetisierten Attributen *enabled*, *used*,

⟨*root*⟩ ::= ⟨*typedecllist*⟩
  | $

⟨*typedecllist*⟩ ::= ⟨*typedecl*⟩ ⟨*typedecllist*⟩
  | ε

⟨*typedecl*⟩ ::= ⟨*modifiers*⟩ ⟨*typedecltype*⟩

⟨*modifiers*⟩ ::= '`public`'
  | '`private`'
  | ⟨*annotation*⟩ ⟨*modifiers*⟩
  | ε

⟨*annotation*⟩ ::= '`@`' ⟨*identifier*⟩ ⟨*annoarguments*⟩

⟨*typedecltype*⟩ ::= ⟨*annotypedecl*⟩
  | ⟨*classdecl*⟩
  | ⟨*interfacedecl*⟩

⟨*annotypedecl*⟩ ::= '`@interface`' ⟨*identifier*⟩ '`{`' ⟨*annotypebody*⟩ '`}`'

⟨*classdecl*⟩ ::= '`class`' ⟨*identifier*⟩ ⟨*superclass*⟩ ⟨*interfaces*⟩ '`{`' ⟨*classbody*⟩ '`}`'

⟨*interfacedecl*⟩ ::= '`interface`' ⟨*identifier*⟩ ⟨*superinterface*⟩ '`{`' ⟨*interfacebody*⟩ '`}`'

⟨*classbody*⟩ ::= ⟨*modifiers*⟩ ⟨*type*⟩ ⟨*identifier*⟩ ⟨*classbodytype*⟩ ⟨*classbody*⟩
  | ε

Abbildung 1: (Vereinfachte) Java-Grammatik in BNF-Notation (Ausschnitt).

name und *names* ausdrücken, wobei *enabled* auf das Vorhandensein von *EnableTransactionManagement* und analog *used* auf *@Transactional* hindeutet. *name* bezeichnet einen *identifier* einer Annotation und *names* eine Menge von Annotationsnamen.

Im Folgenden werden die semantischen Regeln detailliert beschrieben. Tabelle 2 gibt einen Überblick über die relevanten Nichtterminale und ihre synthetisierten Attribute. Abbildung 2 und 3 zeigen eine korrespondierende S-attributierte Grammatik. Die Auswertung startet mit dem Sammeln von ⟨*annotation*⟩-Namen. Jedes ⟨*annotation*⟩-Element gibt seinen *name* an das umschließende ⟨$modifiers$⟩-Element weiter, welches die Namen sammelt. Für ⟨*classbody*⟩ und ⟨$interfacebody$⟩ wird das Attribut *used* auf true gesetzt, wenn die Menge der gesammelten *modifiers* die *@Transactional* Annotation beinhalten.

Danach wird der Wert der *used*-Attribute über ⟨*classdecl*⟩ und ⟨$interfacedecl$⟩ an ⟨*typedecltype*⟩ weitergereicht (s. Abbildung 3). Der Wert des *used*-Attributs für Annotationstypen ist immer false, da sie die *@Transactional* Semantik nicht nutzen können. ⟨*typedecltype*⟩ propagiert den *used*-Wert an die einschließende ⟨*typedecl*⟩.

Nichtterminale Symbole	**Synthetisierte Attribute**
⟨*root*⟩	-
⟨*typedecllist*⟩	enabled, used
⟨*typedecl*⟩	enabled, used
⟨*typedecltype*⟩	used
⟨*classdecl*⟩	used
⟨*classbody*⟩	used
⟨$interfacedecl$⟩	used
⟨$interfacebody$⟩	used
⟨$modifiers$⟩	names
⟨*annotation*⟩	name

Tabelle 2: Nichtterminale und ihre Attribute.

$\langle annotation \rangle$ ::= '`@`' $\langle identifier \rangle$
  {$0.*name* ← *value*($1); }

$\langle modifiers \rangle$ ::= '`public`'
  {$0.*names* ← *emptySet*(); }
| '`private`'
  {$0.*names* ← *emptySet*(); }
| $\langle annotation \rangle$ $\langle modifiers \rangle$
  {$0.*names* ← *union*(*newSet*($1.*name*), $2.*names*); }
| ε
  {$0.*names* ← *emptySet*(); }

$\langle classbody \rangle$ ::= $\langle modifiers \rangle$ $\langle type \rangle$ $\langle identifier \rangle$ $\langle classbodytype \rangle$ $\langle classbody \rangle$
  {$0.*used* ← *intersects*(*newSet*('`Transactional`'), $1.*names*) || $5.*used*; }
| ε
  {$0.*used* ← *false*; }

$\langle interfacebody \rangle$ ::= $\langle modifiers \rangle$ $\langle type \rangle$ $\langle identifier \rangle$ $\langle methoddecl \rangle$ $\langle interfacebody \rangle$
  {$0.*used* ← *intersects*(*newSet*('`Transactional`'), $1.*names*) || $5.*used*; }
| ε
  {$0.*used* ← *false*; }

Abbildung 2: S-attributierte Grammatik zur Detektierung von `Transactional`-Fehlern (Ausschnitt 1).

Zusätzlich wird überprüft, ob die Typdeklaration selbst die *@Transactional* Annotation benutzt. Auch wird überprüft, ob die Typdeklaration tatsächlich eine Spring-Kontext-Konfigurationsklasse ist und, falls dem so ist, ob das Transaktionsmanagement aktiviert ist. Das *enabled*-Attribut repräsentiert das Ergebnis dieser Überprüfung.

Für jede $\langle typedecllist \rangle$ werden die Attribute jeder eingeschlossenen Typdeklarationen zusammengefasst. Schließlich findet die finale Überprüfung an der Wurzel des Syntaxbaums statt. Ein Fehler wird erkannt, wenn das Transaktionsmanagement aktiviert ist, jedoch keine vorhandene Methode die *@Transactional* Annotation verwendet (also: $enabled \wedge \neg used$).

Mit diesen semantischen Regeln ergibt sich für das Beispiel in Listing 1 eine Atributierung des abstrakten Syntaxbaums (ASB), wie sie ausschnittsweise in Abbildung 4 zu sehen ist.

Weitere Fehler oberhalb des Methoden-Levels lassen sich durch ähnliche attributierte Grammatiken erkennen. Für Fehler der Gruppe II verwenden wir L-attributierte Grammatiken. Für Fehler, die *unterhalb* des Methoden-Levels (Gruppe III & IV) auftreten, erzeugt eine L-attributierte Grammatik zunächst einen Kontrollfluss-Graphen (KFG) [1], anhand dessen dann eine Analyse erreichender Definitionen (*reaching definitions*)[14, p.218] durchgeführt wird. Hiermit lassen sich dann z.B. Fehler bezüglich Objekt-zestörender Methoden bei Beans mit Gültigkeitsbereich `prototype` feststellen, wie sie in Listing 3 auftreten.

Der KFG mit den Ergebnissen der Analyse erreichender Definitionen für das Beispiel in Listing 4 findet sich in Abbildung 5. Anzumerken ist, dass bei diesem konstruierten Beispiel die Analyse auf einen Konfigurationsfehler hinweist, der in der Praxis nicht auftreten kann, da die korrespondierende Bedingung nie erfüllt wird (`if (false)`), was aber von der Analyse ignoriert wird.

```
1  @Configuration
2  public class SpringConfig{
3    @Bean
4    @Scope("prototype")
5    public PrintService printService(){
6      PrintService p = new PrintService();
7      if(false)
8        p = new LaserPrinterService();
9      return p;}
10 }
11
12 public class PrintService {...}
13
14 public class LaserPrinterService
15     extends PrintService {
```

⟨*classdecl*⟩ ::= '`class`' ⟨*identifier*⟩ ⟨*superclass*⟩ ⟨*interfaces*⟩ '`{`' ⟨*classbody*⟩ '`}`'
{$0.*used* ← $4.*used*; }

⟨*interfacedecl*⟩ ::= '`interface`' ⟨*identifier*⟩ ⟨*superinterface*⟩ '`{`' ⟨*interfacebody*⟩ '`}`'
{$0.*used* ← $3.*used*; }

⟨*typedecltype*⟩ ::= ⟨*annotypedecl*⟩
{$0.*used* ← *false*; }
| ⟨*classdecl*⟩
{$0.*used* ← $1.*used*; }
| ⟨*interfacedecl*⟩
{$0.*used* ← $1.*used*; }

⟨*typedecl*⟩ ::= ⟨*modifiers*⟩ ⟨*typedecltype*⟩
{$0.*used* ← $2.*used* || *intersects*(*newSet*('`Transactional`'), $1.*names*);
$0.*enabled* ← *intersects*(*newSet*('`Configuration`'), $1.*names*)
&& *intersects*(*newSet*('`EnableTransactionManagement`'), $1.*names*); }

⟨*typedecllist*⟩ ::= ⟨*typedecl*⟩ ⟨*typedecllist*⟩
{$0.*enabled* ← $1.*enabled* || $2.*enabled*;
$0.*used* ← $1.*used* || $2.*used*; }
| *ε*
{$0.*enabled* ← *false*;
$0.*used* ← *false*; }

⟨*root*⟩ ::= ⟨*typedecllist*⟩
{*if*($1.*enabled* && !$1.*used*){*error*(); }}
| $

Abbildung 3: S-attributierte Grammatik zur Detektierung von `Transactional`-Fehlern (Ausschnitt 2).

```
16    @PreDestroy
17    public void close() {...}
18  }
```

Listing 4: Bean mit Gültigkeitsbereich `prototype`.

4 Prototypische Implementierung

Ausgehend von den oben erläuterten attributierten Grammatiken wurde in Java eine prototypische Implementierungen der Annotations-Analyse mit Hilfe der *Java Pluggable Annotation Processing API* umgesetzt. Diese API ist durch die Java Specification Request (JSR) 269 spezifiziert und erlaubt die Verarbeitung von Annotationen während der Compilezeit [6]. Sie definiert ein Sprachmodell des verarbeiteten Java-Codes, das auf dem Kompositum-Entwurfsmuster [8, p.183] beruht. Basierend auf einem Ansatz, wie er in [3] vorgestellt wurde, wird eine Art abstrakter Syntaxbaum zur Verfügung gestellt. Außerdem definiert diese API, wie Compiler-Erweiterungen zu deklarieren und auszuführen sind. Abbildung 6 illustriert die Architektur des Java-Compilers. Sie basiert auf dem Pipe- und Filter-Architekturmuster [12, p.432]. Die Abbildung zeigt, wie unser Prototyp sich in den Übersetzungsprozess über die *Pluggable Annotation Processing API* einfügt. Sobald in diesem Prozess die lexikalische und syntaktische Analyse des Quellcodes abgeschlossen ist, wird der Prototyp durch das Plugin-Interface aufgerufen. Die Transformation der Gesamtheit der erstellten attributierten Grammatiken in Compiler-Plugins erfolgte von Hand, lässt sich aber in zukünftigen Arbeiten automatisieren. Gefundene Fehler werden durch die *Messager*-Komponente zu den Fehlermeldungen des Compilers hinzugefügt.

Die Pluggable Annotation Processing API stellt lediglich eine Untermenge von Java bereit. Sprachkonstrukte, die sich innerhalb von Methodenrümpfen finden, wie z.B. Zuweisungen oder Methodenaufrufe, sind nicht enthalten. Um Letztere zu bearbeiten, wird eine weitere API verwendet. Oracle's javac-Compiler stellt hierfür eine Compiler-spezifische API namens *Compiler Tree API* bereit. Diese Low-Level API stellt eine Struktur bereit,

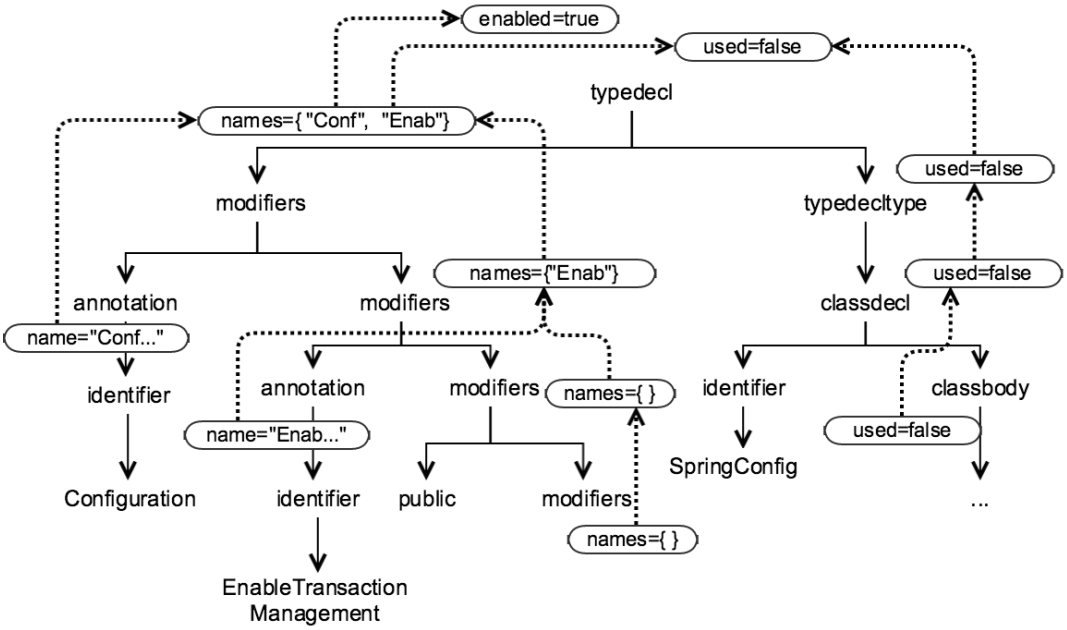

Abbildung 4: Annotierter Syntax-Baum (Ausschnitt).

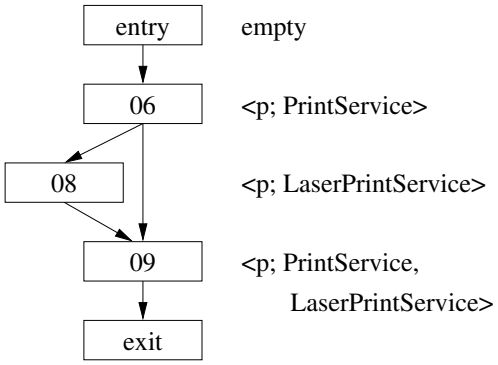

Abbildung 5: KFG für eine prototypische Bean Definition.

die dem benötigten *vollständigen* ASB entspricht. Hierauf aufbauend lassen sich der KFG erstellen und die in Abschnitt 3 erwähnte Datenfluss-Analyse durchführen.

5 Evaluierung

Der entwickelte Prototyp wurde in mehreren Java-Projekten eingesetzt, um seine Funktionalität zu demonstrieren und die Performance zu evaluieren.

Alle Beispielanwendungen basieren auf dem Spring Framework in Version 3.11: Die *Spring Pet Clinic*[2] ist eine Beispielanwendung, die durch das Spring Framework selbst bereitgestellt wird. Sie wurde ausgewählt, da hier jegliche Spring Features vorkommen, die durch den Annotations-Prozessor berücksichtigt werden können, wie Dependency Injection, Transaktionsmanagement und Caching. *Broadleaf Commerce*[3] ist ein Open-Source E-Commerce Framework basierend auf Java und Spring und besteht aus $\sim$ 115.000 Zeilen Quellcode. Es repräsentiert eine realitätsnahe Anwendung. Zusätzlich werden 12 Beispiele[4] berücksichtigt, die jeweils genau eine Instanz der zu identifizierenden Spring-Konfigurations-Fehlertypen beinhalten. Sie repräsentieren zwar keine rea-

Siehe https://github.com/spring-projects/spring-petclinic

Siehe https://github.com/BroadleafCommerce/BroadleafCommerce

Siehe https://github.com/vvhof/DetectingSpringConfigurationErrorsExamples

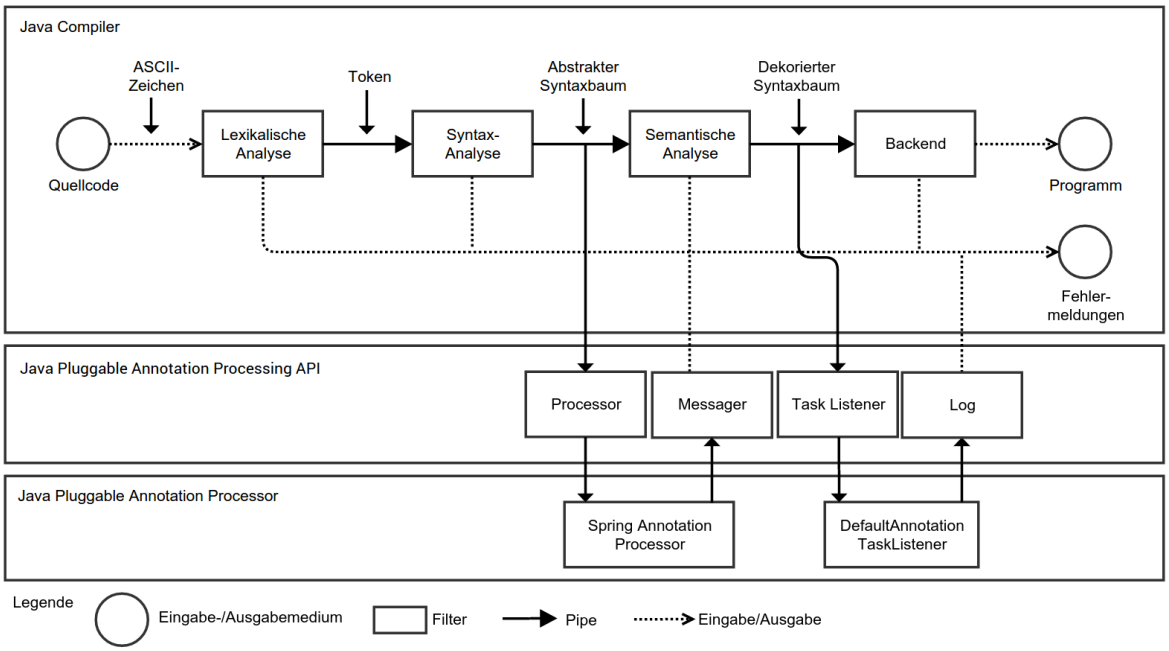

Abbildung 6: Pipe- und Filter-Architektur des Java-Compilers.

litätsnahen Anwendungsfälle, sind allerdings hilfreich, um zeigen zu können, dass der Prototyp die verschiedenen Fehlertypen korrekt erkennen kann.

Um die Performance zu messen, werden die Build-Zeiten der Projekte jeweils mit und ohne Prototyp verglichen. Da alle in Frage kommenden Projekte auf Maven basierend, können die Zeiten direkt in Maven selbst gemessen werden. Um sicherzustellen, dass die Zeiten vergleichbar sind, werden alle Builds auf derselben Maschine mit identischer Konfiguration durchgeführt. Zusätzlich werden die Builds mehrfach ausgeführt, um den Einfluss externen Faktoren, wie z.B. anderer Systemprozesse, zu minimieren.

Für jedes Projekt werden 41 Builds ausgeführt. Der erste Build wird nicht mitgemessen, da Maven beim ersten Durchlauf ggf. Pakete herunterladen muss, was die Build-Zeit verfälscht. Des Weiteren benötigt die Java Virtual Machine einige Zeit zur Initialisierung, was wiederum die Resultate verfälschen kann. 40 zusätzliche Builds werden daraufhin durchgeführt, um daraus die tatsächlichen Build-Zeiten zu errechnen. 20 dieser Builds werden mit dem Prototypen und 20 ohne den Prototypen durchführt.

Da es sich bei Broadleaf Commerce (in Version 3.1.0-ALPHA3) und der Spring Pet Clinic um gut ausgetestete Anwendungen handelt, zeigen sich hier natürlich keine Spring-spezifischen Fehler. Hier interessieren daher nur die ermittelten Build-Zeiten. Alle Fehler in den konstruierten Beispielen werden problemlos gefunden.

Die Laufzeiten der Build-Prozesse mit aktiviertem und deaktiviertem Prototypen unterscheiden sich nicht signifikant (siehe Tabelle 3). Die Unterschiede betragen weniger als eine Sekunde für kleine Projekte. Misst man die Zeiten für einen Annotations-Prozessor, der keinerlei Überprüfungen durchführen muss, ergeben sich dabei vergleichbare Werte. Daher liegt der Schluss nahe, dass ein Großteil der Differenz auf den Annotations-Prozessor selbst und nicht auf den Algorithmus für die Analyse entfällt. Die größte absolute Differenz tritt beim Broadleaf Commerce auf und äußert sich bei diesem Projekt mit 115.000 LoC in einer Steigerung der Build-Zeit um ~ 2 Sekunden und damit $\sim 3\%$. Es sei angemerkt, dass dieses Projekt aus sieben modularen Maven-Projekten besteht und das Java-Plugin für jedes dieser Projekte gestartet werden muss. Daher wird auch der Prototyp sieben Mal lokalisiert und initialisiert.

Bisher kann unser Prototyp mit 12 verschiedenen Fehlertypen umgehen und er kann, wie unsere Experimente zeigen, Fehlerinstanzen der Fehlertypen erfolgreich aufspüren. Die Implementierung weiterer Fehlertypen steht noch aus.

Folgendes lässt sich in Bezug auf die Korrektheit und Vollständigkeit unseres Tools anmerken. Oberhalb des Methoden-Levels werden alle Fehler korrekt erkannt und es werden keine Fehler irrtümlich gemeldet. Unterhalb des Methoden-Levels benutzen wir die erwähnte statische Analyse basierend auf dem Kontrollfluss-Graphen. Durch den damit verbundenen (und unvermeidbaren) Präzisionsverlust kann es vorkommen, dass Fehler gemeldet werden, die auf Grund von Datenabhängigkeiten allerdings niemals auftreten können. Das Beispiel in Listing 4

		Durchschn. Build-Zeiten in ms		
Projekt	**LoC**	**Deaktiviert**	**Aktiviert**	**Diff.**
Fehlertyp 1	29	2 568	2 971	16%
Fehlertyp 2	156	2 675	2 932	10%
Fehlertyp 3	36	2 544	2 741	8%
Fehlertyp 4	37	2 535	2 825	11%
Fehlertyp 5	37	2 524	2 870	14%
Fehlertyp 6	69	2 524	2 915	15%
Fehlertyp 7	56	2 558	2 756	8%
Fehlertyp 8	53	2 501	2 629	5%
Fehlertyp 9	39	2 463	2 725	11%
Fehlertyp 10	39	2 469	2 668	8%
Fehlertyp 11	54	2 502	2 764	11%
Fehlertyp 12	54	2 516	2 748	9%
Spring PetClinic	1 390	9 912	11 448	15%
Broadleaf Commerce	115 902	55 108	56 970	3%

Tabelle 3: Build-Zeiten mit aktiviertem und deaktiviertem Prototypen.

hat dies veranschaulicht. Erfreulicherweise treten solche Probleme in der Praxis selten auf, da es schlechter Programmierstil wäre, die Korrektheit einer Konfiguration von Kontroll- und Datenfluss abhängig zu machen. Man könnte sogar einen Schritt weiter gehen und argumentieren, dass dies tatsächlich auch als Fehler gemeldet werden sollte, damit solche stilistischen Vergehen behoben werden können.

Unsere aktuelle Implementierung unterstützt bisher noch keine Analyse über Methodengrenzen hinweg. Dementsprechend werden Fehler, die ausschließlich durch solch eine Analyse erkannt werden können, zur Zeit noch nicht entdeckt.

Abschließend seien zwei Einschränkungen für den durch uns gewählten Ansatz genannt. Die verwendete *Pluggable Annotation Processing API* macht es zwingend erforderlich, einen Java-Compiler zu verwenden, der diese unterstützt. Dadurch dass die *Compiler Tree API* verwendet wird, muss es sich hierbei weiterhin auf den Oracle-spezifischen Java-Compiler *javac* handeln.

6 Fazit und zukünftige Betätigungsfelder

Dependency Injection ist ein elegantes Entwurfsmuster. Bei seinem Einsatz können allerdings Konfigurationsfehler auftreten, welche nicht durch den Java-Compiler erkannt werden. Diese fehlerhaften Konfigurationen sind daher erst zur Laufzeit erkennbar, weswegen ein kompliziertes Debugging erforderlich wird, was zu Verzögerungen bei der Softwareentwicklung führt. In dieser Arbeit wurde ein selbst entwickeltes Compiler-Plugin für den *javac*-Compiler vorgestellt, welches es erlaubt, fehlerhafte Konfigurationen bereits zur Compilezeit zu erkennen. Konzeptionell basiert das Plugin auf attributierten Grammatiken und verschiedenen APIs des Java-Compilers.

In einem ersten Schritt wurde für das weit verbreitete Framework Spring basierend auf durchgeführten Literaturreviews und Experteninterviews eine Menge von 38 verschiedenen Fehlertypen zusammengestellt. Anschließend wurden diese Fehlertypen in vier Kategorien unterteilt. Die Klassifikation basiert auf zwei Dimensionen. Die erste Dimension beschäftigt sich mit der Frage, ob eine Überprüfung des Fehlers oberhalb oder unterhalb des Methoden-Levels stattfinden muss. Die Zweite unterscheidet, ob der Fehlertyp vom gewählten Spring-Kontext abhängig ist oder nicht. Für jede dieser vier Fehlerklassen wurde ein Schema für eine S- oder L-attributierte Grammatik entwickelt und anschließend für jeden möglichen Fehler instanziiert. Durch die Kombination der attributierten Grammatiken aller Fehlertypen wurde eine übergreifende L-attributierte Grammatik zur Fehlererkennung erzeugt. Für Fehler, die vom Kontroll- und Datenfluss abhängen, wurde eine Analyse erreichender Definitionen (reaching definitions) basierend auf dem Kontrollfluss-Graphen durchgeführt.

In Experimenten mit zwei großen und verschiedenen kleineren Spring-Applikationen konnte nachgewiesen werden, dass das entwickelte Plugin lediglich einen geringen Mehraufwand in der Build-Phase induziert. Außerdem konnte das Plugin alle vorhandenen Konfigurationsfehler erkennen. Im Prinzip können bei aufgrund von Datenabhängigkeiten unerreichbaren Code-Teilen irrtümliche Fehlermeldungen (*false-positives*) auftreten. Diese haben sich im praktischen Einsatz des Werkzeugs bisher allerdings noch nicht bemerkbar gemacht.

Unser Plugin stellt ein hilfreiches Tool für die Spring-Entwicklung dar und wird vom Projektpartner aktuell erfolgreich in der Praxis eingesetzt. Dort unterstützt es das Softwareentwicklungs-Team dabei, Spring-basierte Projekte schneller zu realisieren.

Die aktuelle Implementierung kann 12 der 38 identifizierten Fehlertypen erkennen. In zukünftigen Arbeiten soll das Plugin erweitert werden, sodass zusätzlich die noch fehlenden Fehlertypen erkannt werden. Für alle bis auf 5 Fehlertypen ist dies nach dem existierenden Schema ein einfaches Unterfangen. Die verbleibenden 5 Fehlertypen werden sich nur durch eine Analyse über Methodengrenzen hinweg erkennen lassen.

7 Danksagung

Wir danken der viadee GmbH für ihre Zusammenarbeit.

Literatur

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[3] G. Bracha and D. Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages. *SIGPLAN Not.*, 39(10):331–344, Oct. 2004.

[4] O. Burn. Checkstyle, 2003. `https://checkstyle.sourceforge.net`.

[5] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[6] J. D. Darcy, 2006. `https://www.jcp.org/en/jsr/detail?id=269`.

[7] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *ACM SIGPLAN Notices*, volume 42, pages 321–336. ACM, 2007.

[10] R. e. a. Johnson. Spring Framework Reference Documentation, 2015. `http://docs.spring.io/spring/docs/3.2.11.RELEASE/spring-framework-reference/htmlsingle/`.

[11] D. E. Knuth. Semantics of Context-Free Languages. In *Mathematical Systems Theory*, pages 127–145, 1968.

[12] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2012.

[13] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4, 2010.

[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[15] Oracle Corporation. Package java.lang.reflect, 2015. `https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html`.

[16] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[17] Pivotal Sofware, Inc. Spring Framework, 2015. `http://openjdk.java.net/projects/compiler-grammar/`.

[18] PMD. Pmd, 2015. `http://pmd.sourceforge.net`.

[19] D. R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[20] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[21] SonarSource S.A. SonarQube, 2015. `http://www.sonarqube.org`.

[22] E. Spishak, W. Dietl, and M. D. Ernst. A Type System for Regular Expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[23] E. Van Wyk, L. Krishnan, D. Bodin, and E. Johnson. Adding Domain-specific and General Purpose Language Features to Java with the Java Language Extender. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 728–729, New York, NY, USA, 2006. ACM.

[24] C. Walls. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 4th edition, 2014.