Automatic Refactoring with the Null Object Pattern using PARROT

Bachelorarbeit

Tim-Niklas Reck

28. September 2017

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 28. September 2017

Zusammenfassung

Durch den technologischen Fortschritt sind Mehrkernprozessoren heutzutage ein weit verbreiteter Standard. Die meisten Programme werden jedoch aufgrund der Komplexität und Fehleranfälligkeit von nebenläufigen Programmen noch sequentiell programmiert.

Im Kontext eines halbautomatischen Ansatzes zur Parallelisierung von Programmen wird in dieser Bachelorarbeit ein Ansatz vorgestellt, der eine automatische Umwandlung von Klassen nach dem Nullobjekt-Muster durchführt. Dadurch können mögliche Stellen zur Parallelisierung einfacher erkannt werden. Für unseren Ansatz gehen wir davon aus, dass Java-Programme als Systemabhängigkeitsgraph in einer *Neo4J*-Datenbank vorliegen. Wir entwickeln anhand eines Prototypen eine Abfrage, die mit Hilfe eines Musterabgleichs mögliche Kandidaten für das Nullobjekt-Muster in der Datenbank identifiziert und anschließend entsprechend transformiert.

Für die Evaluation schauen wir uns an, wie unsere Abfrage zur Erkennung anhand verschiedener Beispiele sowie einer bekannten Java-Anwendung funktioniert. Die Transformation testen wir durch einen Vorher-Nachher-Vergleich eines selbst geschriebenen Prototypen. Wir kommen zu dem Ergebnis, dass sowohl die Erkennung als auch die Transformation korrekt durchführbar sind.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1. Motivation

Die stets voranschreitende Weiterentwicklung von Software und ihre zunehmende Komplexität machen es fast unmöglich, diese noch manuell zu analysieren und in ihrer Gesamtheit zu erfassen. Dies erschwert es auch, ein Programm für die Einbindung moderner Hardwarestandards komplett umzustrukturieren, wie es z.B. bei der Parallelisierung von Programmen zur Benutzung von Mehrkernprozessoren nötig wäre.

Die Frameworks Soot[1] und Kieker[2] ermöglichen es, viele dieser Analysen bei Java-Programmen automatisch durchzuführen und Systemabhängigkeitsgraphen (englisch: system dependence graph, SDG) zu erstellen, auf deren Basis Änderungen an der Softwarestruktur vorgenommen werden können. Der in Entwicklung befindliche Ansatz PARROT [Wulf 2014] zielt darauf ab, mit Hilfe von diesen SDGs eine halbautomatische Parallelisierung von Programmen zu ermöglichen.

Im Quellcode eines Programms können allerdings Stellen vorkommen, die die Parallelisierung durch ihre Komplexität erschweren. Ein mögliches dieser Probleme ist z.B. die Abfrage, ob ein Objekt bereits initialisiert wurde oder noch einen `null`-Wert hat. Durch die wiederholten Abfragen und die durch `if`-Anweisungen entstehenden Verzweigungen im Systemabhängigkeitsgraphen ist es nicht möglich, eine einheitliche Parallelisierungsvorschrift zu erstellen. Werden diese Objekte jedoch mit sogenannten Nullobjekten initialisiert, ist es möglich, die `if`-Anweisungen wegzulassen. Das Aufrufen von Funktionen eines Nullobjektes hat zwar keinen Effekt, kann aber auch keine Laufzeitfehler generieren. Somit fallen die zusätzlichen Verzweigungen im SDG weg, sodass sich solche Stellen einfacher parallelisieren lassen.

1.2. Kontext

Diese Arbeit ist ein Beitrag zum Ansatz PARROT von Wulf [2014], welcher in Abbildung 1.1 illustriert wird. PARROT ist ein hybrider Ansatz zur halbautomatischen Parallelisierung von objektorientierten Programmiersprachen, d.h. es werden sowohl statisch, als auch dynamisch gewonnene Informationen in die Analyse einbezogen.

[1] `https://sable.github.io/soot/`
[2] `http://kieker-monitoring.net/`

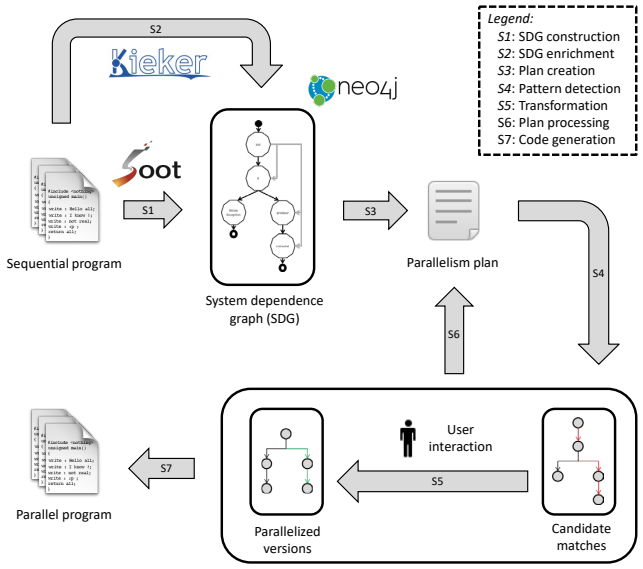

Abbildung 1.1. Der halbautomatische Ansatz zur Parallelisierung PARROT (In Anlehnung an Wulf [2014])

Im ersten Schritt des Ansatzes (S1) wird aus dem Quellcode einer Anwendung unter Verwendung von *Soot* ein SDG gewonnen, der in einer Graphendatenbank gespeichert wird. Dieser SDG wird im zweiten Schritt (S2) durch während der Laufzeit gewonnene Informationen erweitert. Der dritte Schritt (S3) dient der Vorfilterung und Erstellung eines Parallelisierungsplans, damit die Codesegmente mit dem höchsten Parallelisierungspotential priorisiert werden können. Auf Basis dieses Plans kann der Benutzer nun ein Pattern Matching starten (S4). Dieses Pattern Matching dient dazu, vordefinierte sowie potentiell parallelisierbare Codesegmente zu finden. Anschließend kann der Nutzer während des fünften Schrittes (S5) potentielle Übereinstimmungen annehmen oder ablehnen. Akzeptierte und in der Graphendatenbank vordefinierte Segmente werden nun automatisch entsprechend der dazugehörigen Transformation transformiert. So können solche Segmente umstrukturiert (Refactoring) oder auch parallelisiert werden. Nun kann der Benutzer entweder mit weiteren Codeabschnitten fortfahren (S6) oder aber den SDG automatisch wieder in die originale Programmiersyntax übertragen (S7).

1.3. Ziele

Das Ziel dieser Arbeit ist die Entwicklung eines Tools, das unter Verwendung von Cypher-Abfragen geeignete Kandidaten für das in Abschnitt 2.3 beschriebene Nullobjekt-Muster findet und transformiert. Dieses Tool soll in dem in Abschnitt 1.2 beschriebenen Ansatz

von C. Wulf verwendet werden. Die dafür benötigten Schritte werden wir im Folgenden vorstellen und näher erläutern.

1.3.1. Z1: Erkennung von Nullobjekt-Kandidaten

Das erste Ziel dieser Arbeit ist es, die für eine Transformation relevanten Kandidaten zu identifizieren. Dafür soll eine Cypher-Abfrage (siehe Abschnitt 2.2.2) entwickelt werden, mit deren Hilfe wir die Muster dieser Kandidaten im Systemabhängigkeitsgraphen (siehe Abschnitt 2.1) finden können. Die Kriterien, die zur Identifikation von Kandidaten notwendig sind, müssen zunächst anhand eines Beispiels festgestellt und überprüft werden.

1.3.2. Z2: Transformation von Nullobjekt-Kandidaten

Die in Z1 erkannten Kandidaten müssen nun nach dem Nullobjekt-Muster (siehe Abschnitt 2.3) transformiert werden. Dafür soll ein Verfahren entwickelt werden, das die Kandidaten in der *Neo4J*-Datenbank automatisch umwandelt. Die Transformation soll korrekt ausgeführt werden, sodass eine Quellcode-Generierung aus dem SDG ein sinnvolles Programm ausgibt, das sich in einer Java-Umgebung ausführen lässt.

1.3.3. Z3: Evaluation

Die Funktionalität unseres Ansatzes wird ausführlich überprüft. Die Erkennung der Kandidaten (Z1) wird anhand eines eigenen Codebeispiels und einer Auswahl von relevanten Klassen der Java-Anwendung *Apache Ant* überprüft. Dabei wird ein besonderes Augenmerk auf die Korrektheit und die Vollständigkeit der Rückgabe unserer Cypher-Match-Abfrage gelegt. Die korrekte Funktionsweise der Transformation (Z2) soll durch einen Vorher-Nachher-Vergleich von Quellcode und SDG mit dem Ursprungsprogramm sowie einen Vergleich mit einer manuell transformierten Version des Programms evaluiert werden.

1.4. Aufbau

In dem folgenden Kapitel 2 stellen wir eine kurze Übersicht der für das Verständnis dieser Arbeit wichtigen Grundlagen vor. Anschließend präsentieren wir in Kapitel 3 kurz den gesamten von uns verfolgten Ansatz. Daraufhin erläutern wir in Abschnitt 3.1 unsere Vorgehensweise bei der Erkennung von möglichen Kandidaten, die eine Grundlage für die darauf folgende Transformation in Abschnitt 3.2 ist. Die Evaluation unseres Ansatzes erfolgt in Kapitel 4. Das Kapitel ist dabei unterteilt in die Evaluation der Erkennung in Abschnitt 4.1 und die Evaluation der Transformation in Abschnitt 4.2. In Kapitel 5 stellen wir einige ähnliche Arbeiten vor und ziehen in Kapitel 6 ein Fazit zu unserer Arbeit.

Kapitel 2

Grundlagen und Technologien

2.1. Graphentheorie

Das Verständnis der Graphentheorie bildet eine wichtige Grundlage unseres Ansatzes. In diesem Kapitel werden wir zunächst in Abschnitt 2.1.1 allgemeine Graphen einführen, die keinem Programmierparadigma unterliegen. Anschließend werden wir den von uns verwendeten Java-spezifischen Systemabhängigkeitsgraphen in Abschnitt 2.1.2 erläutern.

2.1.1. Allgemeine Graphen

Ein *Graph* G ist das Tupel $G = (V, E)$ aus einer Menge von Knoten V und einer Menge von Kanten E, die je nach Typ des Graphen zwei oder mehr Knoten verbinden. Die Kanten können gerichtet oder ungerichtet sein. Außerdem besteht die Möglichkeit, eine Gewichtung der Kanten einzuführen. Wenn zwischen zwei Knoten mehrere Kanten in der selben Richtung existieren, so nennen wir den Graphen auch *Multigraph* [Golumbic 2004].

Ein *Eigenschaftsgraph* (engl. property graph) ist ein gerichteter Multigraph, dessen Knoten und Kanten durch multiple Eigenschaften in Form von Schlüssel-Wert-Paaren ergänzt werden können. Bei Eigenschaftsgraphen ist es zusätzlich möglich, Knoten und Kanten mit Labels zu versehen. Bei Knoten können diese Labels z.B. als Identifier benutzt werden. Die Labels der Kanten dienen der Darstellung der Art der Beziehung zwischen zwei Knoten. Aus den Eigenschaftsgraphen kann, wie in Abbildung 2.1 in grau dargestellt, durch das Hinzufügen oder Entfernen zusätzlicher Informationen eine Vielzahl anderer Graphen abgeleitet werden [Rodriguez und Neubauer 2010].

Systemabhängigkeitsgraphen (engl. system dependence graph, SDG) werden benutzt um *Programmabhängigkeitsgraphen* (engl. program dependence graph, PDG) zu einem Modell der interprozeduralen Abhängigkeiten zu erweitern. Der PDG dient dazu, die Kontroll- und Datenflussabhängigkeiten von Methoden darzustellen. [Ferrante u. a. 1987] Im SDG werden die Zusammenhänge der aufrufenden und aufgerufenen Methoden in einem PDG durch zusätzliche Caller- und Callee-Knoten ergänzt. Transitive Datenabhängigkeiten werden ebenfalls durch zusätzliche Kanten markiert [Horwitz u. a. 1988].

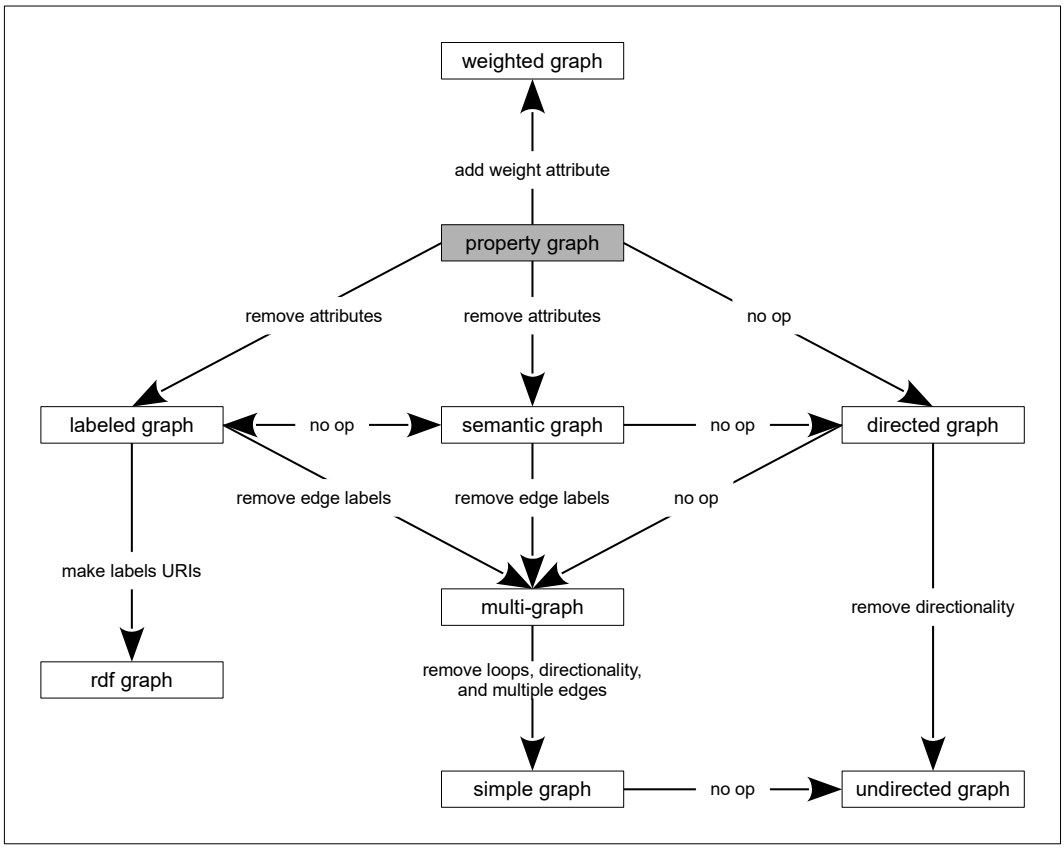

Abbildung 2.1. Zusammenhang verschiedener Graphmodelle mit dem Eigenschaftsgraphen als Zentrum. (In Anlehnung an Rodriguez und Neubauer [2010])

2.1.2. Java-spezifische Systemabhängigkeitsgraphen

Die bisher vorgestellten Graphentypen unterliegen keiner Programmiersprache und können somit für die Darstellung aller Programme verwendet werden. Um die Besonderheiten einzelner Sprachen besser darstellen zu können, gibt es Modelle, die den SDG erweitern und somit für verschiedene Sprachen anpassen [Finkes 2016]. In diesem Kapitel werden der Java-spezifische SDG (JSysDG) von Walkinshaw u. a. [2003] und die Abweichungen des in Abschnitt 1.2 aufgeführten Ansatzes *PARROT* von Wulf [2014] aufgeführt.

Der JSysDG stellt als Multigraph die Kontroll- und Datenabhängigkeiten in einem Java-Programm dar. Dabei werden einzelne Statements eines Programmes je nach ihrer Rolle in unterschiedlichen Schichten dargestellt, die zusammengesetzt das ganze Programm repräsentieren. Durch die komplexe Struktur dieses Graphen ist er jedoch sehr unübersichtlich und somit nicht zur Visualisierung geeignet. Dies lässt sich durch das

Ausblenden aller Knoten in nicht benachbarten Schichten verbessern. Insgesamt gibt es fünf dieser Schichten.

Der **Statement Graph** bildet die unterste Schicht und ist ein atomares Konstrukt, das einen einzelnen Ausdruck im Quellcode des Programms repräsentiert.

Die nächste Schicht bildet der **Method Dependence Graph**. Durch diesen werden einzelne Methoden oder Prozeduren des Programms dargestellt. In dieser Schicht gibt es drei verschiedene Knotentypen: Der *Method-Entry*-Knoten repräsentiert den Einstiegspunkt der Funktion und ist über *Control-Dependence*-Kanten mit den anderen Knoten der Methode verbunden. Die Parameterübergabe wird durch *Actual-In/Out*- und *Formal-In/Out*-Knoten und die sie verbindenden *Parameter-In/Out*-Kanten dargestellt. Außerdem gibt es in dieser Schicht die *Call-Dependence*-Kanten, die aufrufende und aufgerufene Instanzen verbinden.

Der **Class Dependence Graph** repräsentiert die Klassen eines Programms. Er besitzt einen *Class-Entry*-Knoten, der über *Class-Membership*-Kanten, die mit *package (default)*, *public* oder *protected* markiert werden, mit den Methoden seiner Klasse verbunden ist. Vererbung von anderen Klassen wird durch *Class-Dependence*-Kanten hergestellt. Die Daten einer Klasse stehen durch *Data-Member*-Edges mit ihrem Class-Entry-Knoten in Verbindung.

Die vierte Schicht besteht aus dem **Interface Dependence Graphen**. Dieser besitzt einen *Interface-Entry*-Knoten, der mit *Abstract-Member*-Kanten mit den abstrakten Klassen verbunden ist. Diese wiederum sind mit *Implements-Abstract-Method*-Kanten mit ihren implementierenden Methoden verbunden. Klassen, die das Interface implementieren, sind durch *Implements*-Kanten mit dem Entry-Knoten verbunden.

Die letzte Schicht ist der **Package Dependency Graph**. Dieser repräsentiert die einzelnen Pakete eines Programms, deren Klassen und Interfaces jeweils über *Package-Member*-Kanten mit ihrem *Package-Entry*-Knoten verbunden sind.

Beim Ansatz *PARROT* wird für die Erstellung eines SDG aus dem Quellcode eines Programms das Programm *Java2Neo4J*[1] verwendet. Dieses bedient sich einiger unterschiedlicher Knoten und Kantentypen, die in Tabelle 2.1 aufgelistet werden.

Im **Statement-Layer** fallen bei Java2Neo4J die Declaration-Knoten weg. Die Invocation-Knoten werden unterschieden zwischen MethodCall-, MethodCallWithReturnValue- und ConstructorCall-Knoten. Loop- und Noop-Knoten werden bei Java2Neo4J zu NopStmt-Knoten kombiniert. Diese Knoten haben eine zusätzliche Eigenschaft, die der Unterscheidung der Schleifentypen dient. Außerdem gibt es in diesem Layer drei weitere Kantentypen. Die AGGR_CTRL_FLOW-Kante ist für den Kontrollfluss zwischen `for`-Schleifen zuständig. CALLER_OF-Kanten verbinden Assignment- mit MethodCallWithReturnValue-Knoten und LAST_UNIT-Kanten verbinden den letzten Knoten einer Bedingung mit dem ersten.

Das **Method-Layer** besitzt nur einen Knotentypen Method, da Actual-In/Out und Formal-In/Out noch nicht unterstützt werden. Die CALLS-Kante führt von einem Assignment zur Implementierung einer Methode. Die AGGREGATED_CALLS-Kante verbindet zwei Methoden. Parameter-In/Out werden noch nicht unterstützt. Die CONTAINS_UNIT-Kante dient dazu, lokale Felder mit den sie beinhaltenden Methoden zu verbinden. Die

[1] `https://build.se.informatik.uni-kiel.de/chw/sootexample`

LAST_UNIT-Kante verbindet den letzten Knoten einer Methode mit dem ersten und die THROWS-Kante führt zum Class-Knoten von einer geworfenen Exception.

Das **Field-Layer** ist nur in Java2Neo4J vorhanden. Es enthält die Field-Knoten, die über AGGREGATED_FIELD_READ- und AGGREGATED_FIELD_WRITE-Kanten mit aufrufenden Methoden und Konstruktoren verbunden sind.

Beim **Class-Layer** wurde die Class-Dependence-Kante entfernt. Dafür werden die Class-Member-Kanten nun unterschieden in CONTAINS_METHOD- und CONTAINS_-CONSTRUCTOR-Kanten. Die EXTENDS-Kante wird eingesetzt, um Vererbung zwischen Klassen anzuzeigen. Die Abstract-Member- und Implements-Abstract-Member-Kanten des **Interface-Layer** wurden entfernt und nur die IMPLEMENTS-Kante bleibt übrig.

Im **Package-Layer** wurden nur Umbenennungen durchgeführt. [Koppenhagen 2016]

2.2. Die Graphdatenbank Neo4J

Neo4J[2] ist die für unseren Ansatz benutzte Graphdatenbank. Um diese zu verstehen, werden wir zunächst in Abschnitt 2.2.1 das zugrunde liegende Graphmodell einführen. Anschließend werden wir in Abschnitt 2.2.2 die Abfragesprache Cypher, die für die Benutzung von Neo4J essentiell ist, erläutern.

2.2.1. Das Neo4J-Graphmodell

Neo4J ist ein Datenbank-Verwaltungssystem (engl. database management system, DBMS), das zu den NoSQL-Systemen (engl. Not only SQL) gezählt wird [Sullivan 2015]. Die Daten liegen anders als bei relationalen Datenbanken nicht in Form von Tabellen vor, sondern werden durch Knoten und Kanten beschrieben. Dabei besitzt jedes Objekt noch eine eindeutige ID. Konkrete Entitäten wie Methoden und Klassen des SDG werden durch Knoten dargestellt. Die Kanten zwischen den Knoten stehen für Beziehungen, wie beispielsweise Abhängigkeiten oder Zusammenhänge. Den verschiedenen Knoten und Kanten können zusätzlich noch beschreibende Eigenschaften in der Form von Schlüssel-Wert-Paaren hinzugefügt werden [Finkes 2016]. Zusätzlich lassen sich die Knoten noch durch Labels gruppieren, wodurch das Schreiben und Auswerten von Abfragen erleichtert und effizienter wird. Labels dienen u.a. auch während der Laufzeit dazu, temporäre Zustände zu markieren [Neo Technology 2015b].

Abbildung 2.2 zeigt eine Darstellung einer beispielhaften Datenstruktur. Es gibt drei Knoten, von denen zwei das Label *Person* und einer das Label *Movie* haben. Von den *Person*-Knoten geht jeweils eine gerichtete Kante zum *Movie*-Knoten. Die *ACTED_IN*-Kante und die Knoten werden jeweils mit Schlüssel-Wert-Paaren näher beschrieben.

[2] https://neo4j.com/

Tabelle 2.1. Vergleich der Knoten und Kanten von JSysDG und Java2Neo4J in Anlehnung an Koppenhagen [2016]

Layer	Type	JSysDG	Java2Neo4J
Statement	Node	Assignment	Assignment
		Condition	Condition
		Declaration	-
		Invocation	MethodCall MethodCallWithReturnValue ConstructorCall
		Loop	NopStmt
		Noop	
		Return	ReturnStmt
	Edge	Data Dependence	DATA_FLOW
		Control Dependence	CONTROL_FLOW AGGR_CTRL_FLOW
		-	CALLER_OF
		-	LAST_UNIT
Method	Node	Method Entry	Method
		Actual In	-
		Actual Out	-
		Formal In	-
		Formal Out	-
	Edge	Call Dependence	CALLS AGGREGATED_CALLS
		Parameter In	-
		Parameter Out	-
		-	CONTAINS_UNIT
		-	LAST_UNIT
		-	THROWS
Field	Node	-	Field
	Edge	-	AGGREGATED_FIELD_READ
		-	AGGREGATED_FIELD_WRITE
Class	Node	Class Entry	Class
	Edge	Class Dependence	-
		Class Member	CONTAINS_CONSTRUCTOR CONTAINS_METHOD
		Data Member	CONTAINS_FIELD
		-	EXTENDS
Interface	Node	Interface Entry	Interface
	Edge	Abstract Member	-
		Implements	IMPLEMENTS
		Implements Abstract Method	-
Package	Node	Package Entry	Package
	Edge	Package Member	CONTAINS_TYPE

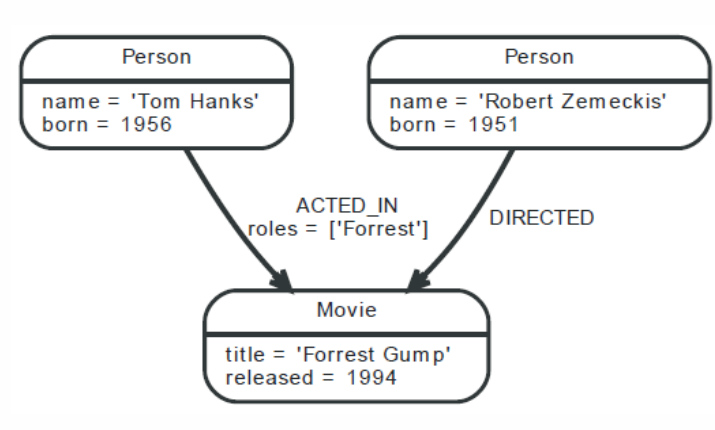

Abbildung 2.2. Beispiel eines Graphen in Neo4J [Neo Technology 2015b]

2.2.2. Die Abfragesprache Cypher

Cypher ist die für Neo4J zu verwendende deklarative Abfragesprache, mit welcher der Inhalt der Datenbank abgefragt und aktualisiert werden kann. Dabei orientiert sich die Struktur der Klauseln an der von der SQL (Structured Query Language) bekannten Struktur [Neo Technology 2015a]. Die Auswertung der Abfragen erfolgt mit einem Pattern-Matching, sodass der Graph nur entlang jener Knoten und Kannten, die das jeweilige Muster erfüllen, traversiert wird. Der Einfachheit halber beginnen Cypher-Abfragen mit einer oder mehreren Klauseln, die die Art der Abfrage spezifiziert, auf die eine Beschreibung eines Musters oder anderer Angaben folgt, nach denen gehandelt werden soll. In den Mustern werden Knoten mit Klammern () und Kanten mit Pfeilsymbolen -> dargestellt [Finkes 2016].

Für Abbildung 2.2 ist eine einfache Abfrage nach diesem Schema in Listing 2.1 aufgeführt. Diese Abfrage listet die Namen aller Filme auf, in denen 'Tom Hanks' laut unserer Datenbank als Schauspieler mitgewirkt hat. Da für die Abfrage kein expliziter Startknoten angegeben wurde, werden alle Knoten als potentielle Startknoten angesehen. Es werden also alle Knoten und Kanten selektiert, die dem Muster in der *MATCH*-Klausel entsprechen. Aus diesen Knoten definiert die *RETURN*-Klausel die gewünschte Projektion der Rückgabewerte unserer Abfrage. Neben den in Listing 2.1 verwendeten Klauseln gibt es noch weitere, die der Sortierung und Limitierung der Ergebnisse oder der Veränderung der Datenbank dienen.

Listing 2.1. Abfrage in Cypher: In welchen Filmen spielte Tom Hanks mit?

```
1  MATCH (tom {name: 'Tom Hanks'})-[:ACTED_IN]->(movie)
2  RETURN tom.name, movie.name
```

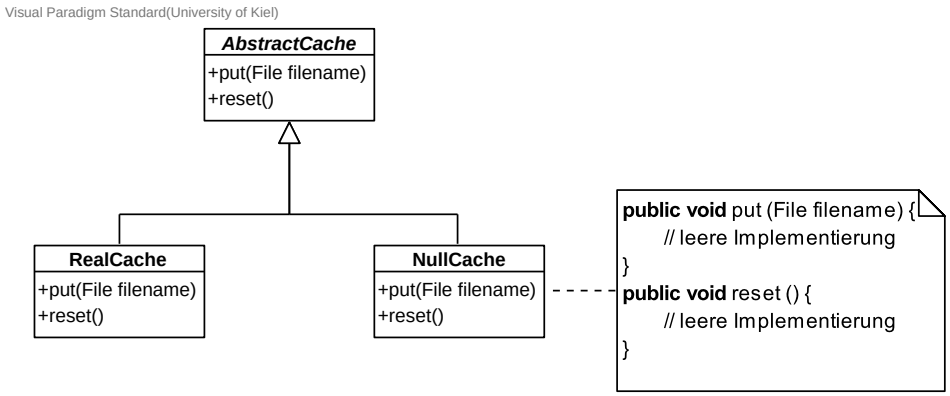

Abbildung 2.3. Umsetzung der Klasse *Cache* aus Listing 2.2 nach dem Nullobjekt-Muster.

2.3. Das Nullobjekt-Muster

Das Nullobjekt-Muster (engl. Null Object Pattern) ist ein Strukturmuster, das bei Klassen wie jener in Listing 2.2 verwendet werden kann, die ein Feld enthalten, das nicht automatisch initialisiert wird. Dieses Feld kann somit einen Wert von *null* annehmen. Um nicht zur Vermeidung von Laufzeitfehlern bei jedem Zugriff auf ein Feld seine Initialisierung überprüfen zu müssen (siehe Listing 2.2, Zeile 12), kann man das Feld mit einem dazugehörigen Nullobjekt initialisieren.

Abbildung 2.3 zeigt eine mögliche Struktur des Nullobjekt-Musters für das gegebene Beispiel aus Listing 2.2. Es wird eine abstrakte Klasse mit zwei Subklassen implementiert. Dies ist in unserem Beispiel *AbstractCache*. Die Funktionen dieser Klasse sind abstrakt und haben die gleichen Namen wie die der ursprünglichen Klasse des Feldes. Eine der Subklassen ist die ursprüngliche Klasse, die bei einer Initialisierung des Feldes in unserer Klasse instanziiert wird. Diese Klasse wird in Abbildung 2.3 durch *RealCache* repräsentiert. Die andere Subklasse ist das Nullobjekt, bei uns der *NullCache*, welches die gleichen Funktionen besitzt, die allerdings einen leeren Körper haben und somit bei einem Aufruf auch keine Fehler erzeugen. Für das Nullobjekt bietet sich oft eine Implementierung als Singleton an.

In Listing 2.3 wird das modifizierte Objekt zur Initialisierung des Feldes verwendet (siehe Z.2) und die Abfragen, ob ein Feld initialisiert wurde, fallen weg. Die Funktionen können somit unabhängig von der Initialisierung aufgerufen werden (vgl. Zeile 12). Umfassendere Informationen zum Nullobjekt-Muster sind z.B. in Fowler [2000] nachzulesen.

Listing 2.2. Ursrpüngliche Klasse

```
1  public class RandomClass {
2     private Cache cache;
3
4     public RandomClass(){
5     }
6
7     public void createCache(){
8        this.cache = new Cache();
9     }
10
11    public void cacheFile(File filename)
          {
12       if (cache != null) {
13          cache.put(filename);
14       }
15    }
16
17    public void clearCache(){
18          if (cache != null) {
19             cache.reset();
20       }
21    }
22 }
```

Listing 2.3. Klasse mit Nullobjekt.

```
1  public class RandomClass {
2     private AbstractCache cache = new
          NullCache();
3
4     public RandomClass(){
5     }
6
7     public void createCache(){
8        this.cache = new RealCache();
9     }
10
11    public void cacheFile(File filename)
          {
12       cache.put(filename);
13    }
14
15    public void clearCache(){
16          cache.reset();
17    }
18 }
```

Kapitel 3

Ansatz

Eine Übersicht des gesamten Ansatzes dieser Arbeit ist in Abbildung 3.1 dargestellt. Zunächst muss das zu analysierende Programm in seine SDG-Repräsentation übersetzt werden. Dafür wird das Tool *Java2Neo4J* benutzt (S1). In diesem SDG können nun mit Hilfe von Cypher-Match-Abfragen (CMQs) die Muster möglicher Kandidaten gefunden werden (S2). Die identifizierten Kandidaten werden im SDG anschließend mit Cypher-Update-Abfragen (CUQs) in das Nullobjekt-Muster transformiert (S3). Der transformierte SDG wird abschließend mit dem Tool *Sdg2Java* wieder in Java-Quellcode übersetzt (S4). Die Evaluation der Durchführbarkeit und Korrektheit unseres Ansatzes wird auf Basis eigener und Open-Source-Codebeispiele (S5).

Für die Schritte S1 und S4 greifen wir auf die bestehenden Tools *Java2Neo4J* und *Sdg2Java* zurück. Diese wurden im Rahmen des Ansatzes *PARROT* (siehe Abschnitt 1.2) zur Generierung eines SDG aus Java-Quellcode und umgekehrt entwickelt. Die grau markierten Schritte stellen den Kernpunkt unserer Arbeit dar. Die Entwicklung der CMQ thematisieren wir ausführlich in Abschnitt 3.1. Zur Transformation der identifizierten Kandidaten sind mehrere Zwischenschritte notwendig. Diese erläutern wir in Abschnitt 3.2. Die Evaluation unseres Ansatzes führen wir in Kapitel 4 durch.

3.1. Erkennung von Kandidaten

Zur Erkennung der Nullobjekt-Kandidaten ist es zunächst notwendig, den Quellcode in einen Java-spezifischen Systemabhängigkeitsgraphen umzuwandeln. Dafür verwenden wir das in Abschnitt 2.1.2 vorgestellte Programm *Java2Neo4J*. Dieses liefert uns eine Neo4J-Datenbank, die einen JSysDG enthält. Der mit *Neo4J* mitgelieferte Browser ermöglicht es uns eine grafische Darstellung von Teilen des Graphen zu generieren und somit herauszufinden, wie mögliche Kandidaten in der Datenbank repräsentiert werden. In Abschnitt 3.1.1 werden wir anhand eines Beispielprogramms Kandidaten identifizieren und die Muster ihrer Repräsentation in der Datenbank näher betrachten. Darauf aufbauend stellen wir in Abschnitt 3.1.2 eine Cypher-Abfrage vor, die automatisch Stellen mit genau diesem Muster im Graphen findet.

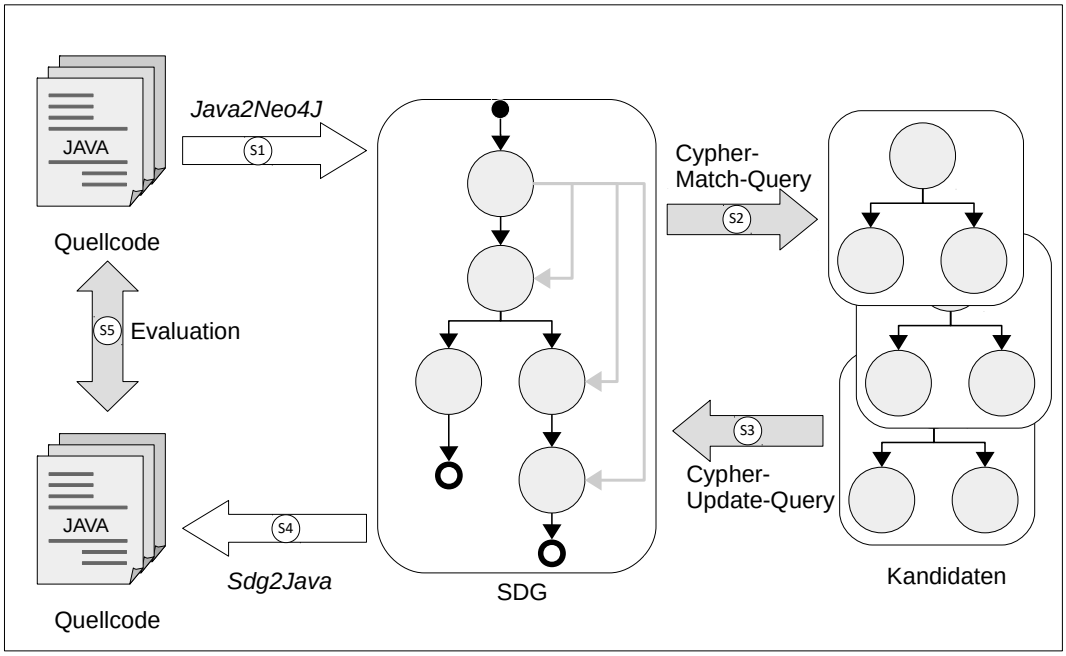

Abbildung 3.1. Der Ansatz dieser Arbeit

3.1.1. Repräsentation des Datenmodells durch Neo4J

Eine Klasse, die als Kandidat gewertet wird, muss notwendigerweise in einer anderen Klasse, der *Aufruferklasse*, als Feld deklariert werden und darf nicht den Modifizierer `final` besitzen. Sollte es letzteren besitzen, muss es initialisiert werden und kann danach nicht mehr mit `null` überschrieben werden. Ist das Feld nicht `final`, so kann auch eine Initialisierung bei der Deklaration oder im Konstruktor später wieder überschrieben werden. Nach der Umwandlung des Feldes unserer Beispielklasse (siehe Listing 3.1, Zeile 3) zum Graphen können wir das entstehende Muster erkennen (siehe Abbildung 3.2). In der *Neo4J*-Datenbank wird dieses durch eine `CONTAINS_FIELD`-Beziehung von einem `Class`-Knoten zu einem `Field`-Knoten und dem `isfinal`-Property repräsentiert, welches entweder den Wert `false` oder `true` annimmt. Die Klasse des Feldes hat keine direkte Beziehung zu diesem, kann aber durch das `vartype`-Property des Feldes ermittelt werden, welches ihrem Fully-Qualified-Name (FQN) entspricht.

Weiterhin ist es für die Erkennung als Kandidat notwendig, dass eine Bedingung existiert, in der überprüft wird, ob das Feld bereits initialisiert ist. In unserer Beispielklasse in Listing 3.1 geschieht dies in der sechsten Zeile. Eine `if`-Anweisung wird in der *Neo4J*-Darstellung des Kontrollflusses einer Methode durch eine Abfolge von `Nop`-Statements (`NopStmt`) repräsentiert.

`Nop`-Statements werden durch den Wert des Properties `nopkind` unterschieden. Der erste Knoten einer `if`-Anweisung hat immer den `nopkind`-Wert `IF_COND`. Dieser bezeichnet den Beginn der `if`-Anweisung. Diese wird durch einen `NopStmt`-Knoten beendet, dessen `nopkind` gleich `IF_END` ist. Zwischen diesen beiden Knoten befinden sich noch die Nop-Statements mit den Werten `IF_THEN` und `IF_ELSE`. Diese leiten den Kontrollfluss ein, der im Falle einer erfüllten bzw. nicht erfüllten `if`-Bedingung eintritt. Der Kontrollfluss endet in beiden Fällen im Endknoten. Sollte eine `if`-Anweisung mehrere Bedingungen besitzen, die durch `&` oder | getrennt werden, so werden diese als eigene `if`-Anweisungen repräsentiert, deren `nopkind`-Werte durch das Suffix `_X` ergänzt werden.

Den Kontrollfluss der Methode `clearCache` haben wir in Abbildung 3.3 vereinfacht dargestellt. Die mit '[*]' markierten Kontrollflusskanten können für eine beliebig lange Folge von weiteren Operationen, Zuweisungen und Methodenaufrufen stehen. Da diese jedoch für die Erkennung nicht relevant sind, zeigen wir nur die gekürzte Variante. Die für uns kritische Abfrage, ob das Feld `null` ist, repräsentieren ein `Assignment`-Knoten und ein `Condition`-Knoten. Der Assignment-Knoten steht für eine temporäre Zuweisung des Feldes als erster Operand der Bedingung des `Condition`-Knotens. Der zweite Operand ist der Wert `null` und die Operation ist der Ungleich-Operator. Semantisch ist es jedoch nicht wichtig, ob der Wert des Feldes als erster oder zweiter Operand zugewiesen wird.

Sollte die Operation des `Condition`-Knotens etwas anderes als den Ungleich-Operator enthalten oder einer der Operanden nicht `null` sein, so können wir die Klasse des Feldes vorerst ausschließen. Sollten noch von anderen Methoden der Klasse relevante Aufrufe ausgehen, so kann die Klasse dennoch als Kandidat bestätigt werden.

Listing 3.1. Beispielklasse mit Nullobjekt-Kandidat

```
1  public class MainClass {
2
3      private Cache cache;
4
5      public void clearCache(){
6          if (cache != null) {
7              cache.reset();
8          }
9      }
10
11     public void createCache(){
12         if (cache == null) {
13             this.cache = new Cache();
14         }
15     }
16 }
```

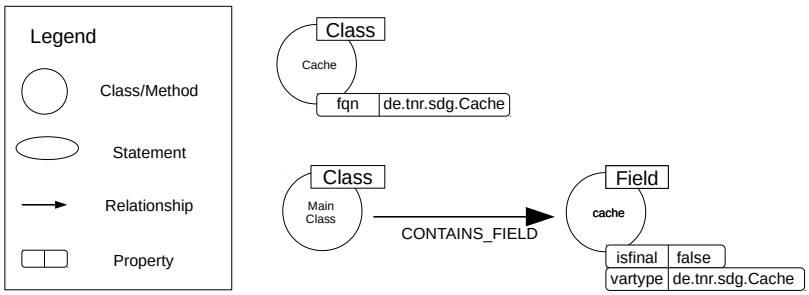

Abbildung 3.2. Repräsentation des Beispielfeldes in Neo4j

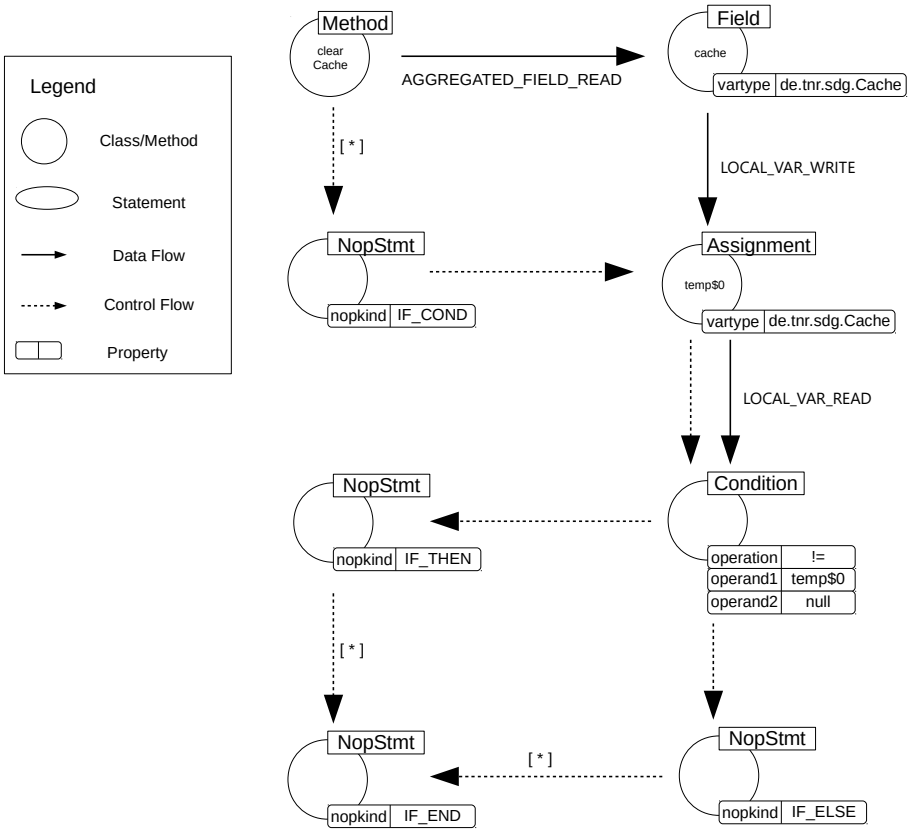

Abbildung 3.3. Repräsentation des Aufrufs durch eine Methode

3.1.2. Definition der Cypher-Match-Abfrage

Nachdem wir in Abschnitt 3.1.1 die Muster der Kandidaten erläutert haben, können wir diese nun in einer Cypher-Match-Abfrage modellieren. Listing 3.2 zeigt diese Abfrage. Im Folgenden werden wir sie schrittweise erläutern.

Listing 3.2. MATCH-Abfrage

```
1 MATCH (mainClass:Class)-[:CONTAINS_FIELD]->(candidateField:Field {isfinal:false})<-[:
      AGGREGATED_FIELD_READ]-(method:Method)
2   USING INDEX candidateField:Field(isfinal)
3 MATCH (candidateField)-[:DATA_FLOW]->(condVariable:Assignment)-[:DATA_FLOW]->(condition:
      Condition {operation:"!="})
4   WHERE condition.operand1 = "null" OR condition.operand2 = "null"
5 MATCH (condVariable)<-[:CONTROL_FLOW]-(ifStmt:NopStmt)
6   WHERE ifStmt.nopkind = "IF_COND" OR ifStmt.nopkind = "IF_COND_X" OR (ifStmt) <-[:
        CONTROL_FLOW]- (:Condition)
7 MATCH (candidate:Class)
8   WHERE candidate.fqn = candidateField.vartype AND candidate.isabstract = false
9 RETURN DISTINCT candidateField, condVariable, candidate
```

Durch aufeinander aufbauende `MATCH`-Klauseln innerhalb der Abfrage wird die Auswahl an Knoten nach und nach eingegrenzt. Um dies zu beschleunigen bietet *Neo4J* die Möglichkeit, Indizes zu erstellen, welche die Geschwindigkeit von Suchoperationen auf Kosten der Schreibgeschwindigkeit erhöhen. Diese Indizes bestehen immer aus einer Kombination von einem Label und einem oder mehreren Properties. In Listing 3.3 erstellen wir die für diese Match-Abfrage benötigten Indizes. In den meisten Fällen greift Neo4J bei der Ausführung von Abfragen automatisch auf diese zu. In einigen Situationen erkennt *Neo4J* die Möglichkeit jedoch nicht. Dann können wir dies wie in Zeile 2 von Listing 3.2 durch Benutzung der `USING INDEX`-Klausel erzwingen.

Listing 3.3. Erstellung der für die MATCH-Abfrage benötigten Indizes

```
1   CREATE INDEX ON :Field(isfinal)
2   CREATE INDEX ON :Condition(operation)
3   CREATE INDEX ON :Class(fqn)
```

Zunächst wenden wir in der ersten Zeile unser Wissen über die Beziehung zwischen Aufruferklasse und Feld an und fordern diese als Bedingung. Das Feld muss im `isfinal`-Property einen Wert von `false` haben. Da in den Mustern der Cypher-Abfragen Relationen von zwei Seiten an einen Knoten modelliert werden können, nutzen wir dies um alle `Field`-Knoten auszuschließen, die von keiner Methode gelesen werden.

In der dritten Zeile benutzen wir die gerade gefundenen `Field`-Knoten und überprüfen, ob sie einen Datenfluss zu einem `Assignment`-Knoten haben, der wiederum direkt zu einem `Condition`-Knoten weiterfließt. Dabei wird die Auswahl gültiger `Condition`-Knoten eingeschränkt durch die Forderung, dass ihre Operation dem Operator '! =' entsprechen muss. Eine weitere Einschränkung wird durch die `WHERE`-Klausel in Zeile 4 festgelegt, wodurch entweder der erste oder der zweite Operand einen Wert von `null` haben muss. Die fünfte und sechste Zeile dienen dazu, die `Condition`-Knoten weiterführend nur auf jene zu begrenzen, die nur in einer `if`-Anweisung auftreten. Dabei beachten wir sowohl

den Aufruf als einzige Bedingung, als auch den mit mehreren Bedingungen.

Die Zeilen 7 und 8 dienen dazu, die Klassen der gefundenen Felder durch Abgleich von `fqn` und `vartype` zu identifizieren und abstrakte Klassen auszuschließen, da wir diese vorerst nicht transformieren können. Auch Interfaces, Primitive Klassen und Klassen, die nicht in der Datenbank vorhanden sind, weil sie aus externen Bibliotheken stammen, werden durch diese Zeile als Kandidaten ausgeschlossen.

Anschließend werden in der letzten Zeile die gefundenen Klassen, Felder und Zuweisungen zurückgegeben. Das Schlüsselwort '`DISTINCT`' dient dazu, doppelte Ergebnisse aus der Ergebnismenge herauszufiltern.

Für unseren Ansatz wird diese `MATCH`-Abfrage mit Hilfe der *Neo4J*-Java-API in einer Java-Umgebung ausgeführt und die einzelnen Ergebnismengen `candidateField`, `condVariable` und `candidate` zur späteren Weiterverarbeitung als Listen zwischengespeichert. Es ist jedoch auch möglich, die Abfrage direkt mit *Neo4J* zu benutzen (z.B. im *Neo4J*-Browser), wenn die Indizes vorher initialisiert werden.

3.2. Transformation der Kandidaten

Nachdem wir in Abschnitt 3.1 eine Cypher-Abfrage zur Identifikation von Kandidaten vorgestellt haben, widmen wir uns in diesem Kapitel der zugehörigen Transformation. Dafür werden wir in Abschnitt 3.2.1 zunächst unser Vorgehen bei der Bearbeitung der Graphdatenbank erläutern. Anschließend stellen wir die Funktionsweise der Transformation der Kandidaten in Abschnitt 3.2.2 und der Aufruferklassen, die auf entsprechende Felder der Kandidaten zugreifen, in Abschnitt 3.2.3 vor. Da die gesamte Transformation sehr umfangreich ist, geben wir in diesem Abschnitt nur einige Beispiele für Abfragen an und verweisen auf den Quellcode des Programms, der im Github[1] verfügbar ist.

3.2.1. Graphtransformation in Neo4J

Wie auch schon Krause [2015] stoßen wir bei der Transformation auf einige Hürden, die es zu überwinden gilt. Denn bei der Transformation des SDG in der *Neo4J*-Datenbank wollen wir nicht nur Änderungen an bekannten Klassen durchführen, sondern benötigen einen Algorithmus, der sich auf alle Klassen anwenden lässt, die von der in Abschnitt 3.1 vorgestellten Cypher-Match-Abfrage (CMQ) identifiziert werden. Daher müssen Knoten und Beziehungen verändert, gelöscht oder hinzugefügt werden können, wenn dies erforderlich ist. *Neo4J* bietet dafür die Möglichkeiten der direkten Benutzung von Cypher-Update-Abfragen (CUQ) im *Neo4J*-Browser, der Nutzung von *REST*-Abfragen oder der Verwendung der Java-API von *Neo4J*.

Die Ergebnismenge der CMQ kann bezüglich enthaltener Methoden, Felder und Konstruktoren sehr unterschiedlich aufgebaute Klassen enthalten. Eine flexible CUQ, die alle

[1] `https://github.com/Invice/NullObjectPattern/tree/master/Neo4J-Query`

diese Möglichkeiten einbezieht, müsste viele Schleifen und Bedingungen enthalten. Programmatisch würde diese CUQ als ein sehr langer und unübersichtlicher String dargestellt werden. Notwendige Änderungen der Transformation können somit leicht semantische oder syntaktische Fehler enthalten, deren Debugging nur durch das Betrachten dieses langen Strings möglich wäre. Weiterhin gestaltet es sich schwierig, alle Informationen, die für die Transformation wichtig sind, gezielt abzufragen, da innerhalb von `FOREACH`-Schleifen einer Abfrage keine `MATCH`-Klauseln erlaubt sind. Auch die Erstellung beziehungsweise Umwandlung einzelner Strings, die als Property-Werte der Knoten vorkommen, ist unter Verwendung von Cypher sehr umständlich.

Andererseits ist es mit Cypher einfach, mit kurzen und übersichtlichen Abfragen nach bestimmten Mustern zu suchen. Anders als wir in Kapitel 3 beschreiben, arbeiten wir daher nicht mit einer einzigen CUQ, sondern mit mehreren CUQs, deren Ergebnisse wir mit der Java-API kombinieren. Dieses Vorgehen erleichtert das Erstellen, Zwischenspeichern und Ändern von Knoten und Beziehungen erheblich. Insgesamt kombinieren wir somit die Stärken beider Ansätze und können nach Bedarf auf beide Funktionen zurückgreifen.

Im Folgenden erläutern wir kurz die wichtigsten Funktionalitäten der Java-API, die für die Transformation benötigt werden.

Einfügen neuer Neo4J-Knoten

Für die Transformation eines Kandidaten ist es notwendig, neue Knoten zu erstellen und in die Datenbank einzufügen. Die Schnittstelle zur *Neo4J*-Datenbank wird in der Java-API vom `GraphDatabaseService` gestellt. Wie alle Zugriffe auf die Datenbank muss dies innerhalb einer `Transaction` passieren. Diese stellt sicher, dass Änderungen nur dann in der Datenbank übernommen werden, wenn jeder Schritt der `Transaction` erfolgreich ist. Knoten werden ausschließlich durch die Methode `GraphDataBase.createNode()` erstellt. In Listing 3.4 sehen wir dies in Form eines kurzen Ausschnitts der Transformation. In Zeile 1 wird der neue Knoten erstellt und in die Datenbank eingefügt. Das übergebene Argument ist das `Label` des Knotens. In dem *Java2Neo4J*-Modell wird jedem Knoten nur ein Label zugeordnet. Sollte es dennoch erforderlich sein, weitere Labels hinzuzufügen, kann dies durch die Methode `Node.addLabel(Label label)` geschehen. In der zweiten Zeile fügen wir dem Knoten das Property `fqn` hinzu. Mit der selben Methode kann der Knoten um weitere Properties erweitert werden.

Listing 3.4. Erstellung neuer Knoten

```
1  Node realNode = dbService.createNode(SDGLabel.CLASS);
2  realNode.setProperty(SDGPropertyKey.FQN, "de.tnr.sdg.MainClass");
```

Ähnlich wie in der Arbeit von Krause [2015] benutzen wir konstante Labels und Property-Keys, die wie in Listing 3.5 in den gesonderten Klassen `SDGLabel` und `SDGPropertyKeys` gespeichert werden. Dadurch ist es möglich Namensänderungen mit geringem Aufwand durchzuführen.

Listing 3.5. Konstante Namen von Labels und Properties

```
1 public class SDGLabel{
2    public static final Label ASSIGNMENT = Label.label("Assignment");
3    ...}
4
5 public class SDGPropertyKeys{
6    public static final String ARGS = "args";
7    ...}
```

Einfügen neuer Neo4J-Beziehungen

Bei der Erstellung neuer Knoten oder der Verschiebung alter Knoten ist es nötig, neue Beziehungen in der Datenbank zu erstellen. Dafür wird in der Java-API von *Neo4J* die in Listing 3.6 folgende Methode zur Verfügung gestellt:

Listing 3.6. Erstellung einer neuen Beziehung zwischen zwei Knoten

```
1    nullNode.createRelationshipTo(candidateNode, RelTypes.EXTENDS);
```

In diesem Fall wird eine `EXTENDS`-Beziehung vom Knoten `nullNode` zum Knoten `candidateNode` hinzugefügt.

Wenn wir einen Subgraph in die Datenbank einfügen wollen, so müssen wir nach dessen Erstellung eine Verbindung mit dem bestehenden Graphen herstellen. Dies geschieht durch das Hinzufügen von Beziehungen zwischen den Start- und Endknoten des Subgraphen und den entsprechenden Knoten des Hauptgraphen. Beziehungen, die durch das Einfügen des Subgraphen überflüssig geworden sind, müssen entfernt werden. Dies geschieht mit der `delete()`-Methode der Klasse `Relationship`.

3.2.2. Transformation der Kandidaten-Klasse

Zu Beginn der Transformation werden die neuen Varianten der Kandidaten-Klasse erstellt. Dafür lesen wir zunächst den FQN des Kandidaten aus und passen ihn für jede der Variationen nach dem Nullobjekt-Muster mit den Präfixen `Real`, `Null` oder `Abstract` an. Durch eine kurze Abfrage, die Klassen mit diesen FQNs sucht (siehe Listing 3.7), können wir überprüfen, ob die Namen schon verwendet werden. In diesem Fall fügen wir so lange eine wachsende Zahl hinzu, bis die neuen Namen verfügbar sind. So könnte sich z.B. der Name `Real2Cache` ergeben.

Listing 3.7. Abfrage zur Erkennung doppelter FQNs

```
1 MATCH (class:Class)
2    WHERE class.fqn = $realFqn
3       OR class.fqn = $abstractFqn
4       OR class.fqn = $nullFqn
5 RETURN class
```

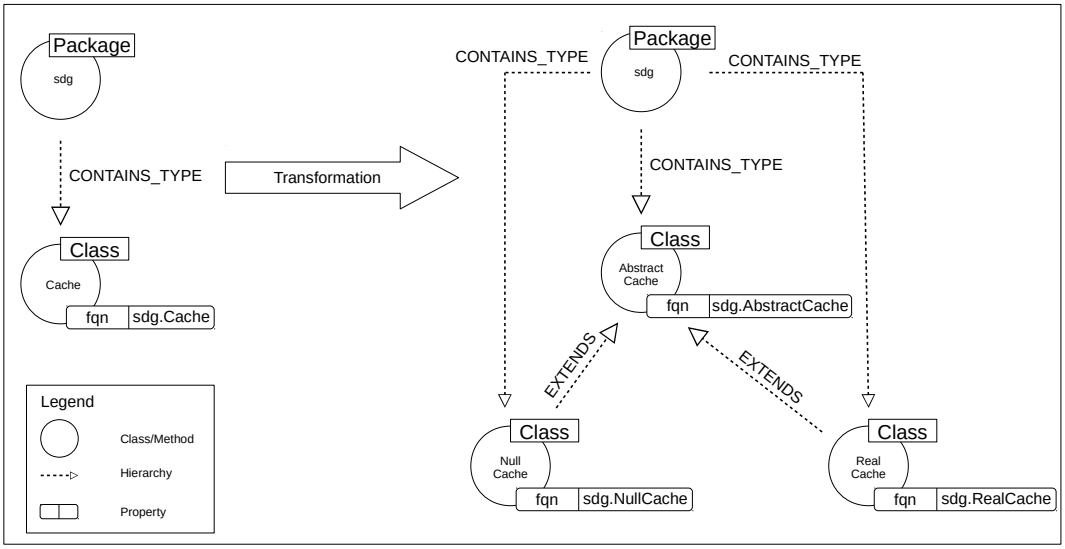

Abbildung 3.4. Erstellung neuer Klassenknoten

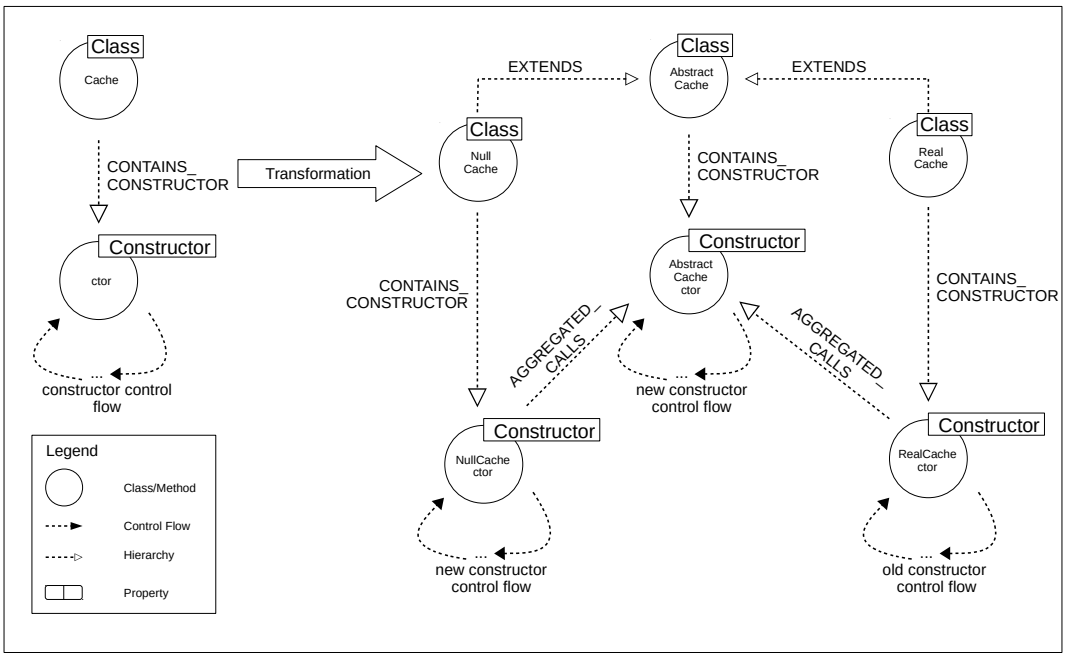

Abbildung 3.5. Erstellung neuer Konstruktoren

Zur Erstellung der neuen Klassen fügen wir in der Datenbank, wie in Abbildung 3.4 zu sehen, zwei neue Knoten hinzu, den `Real`- und den `Null`-Knoten, die beide den `Abstract`-Knoten erweitern. Letzterer wird durch Umbenennung des Kandidaten gewonnen. Die neuen Knoten werden durch eine `CONTAINS_TYPE`-Beziehung dem Paket zugeordnet und erhalten bis auf Namen die selben Eigenschaften der Kandidaten-Klasse.

Damit die neuen Klassen auch instanziiert werden können, müssen sie Konstruktoren besitzen. Der `Abstract`-Knoten ist nur durch Umbenennung des Kandidaten-Knotens entstanden und besitzt somit noch dessen Konstruktoren. Die abstrakte Klasse soll jedoch nicht instanziiert werden können. Daher erstellen wir, wie in Abbildung 3.5 gezeigt, für jeden Konstruktor des Kandidaten je einen Konstruktor für die `Real`- und `Null`-Klasse. Der Kontrollfluss und alle Konstruktoraufrufe des Kandidaten-Konstruktors werden auf den `Real`-Konstruktor übertragen, um sicherstellen zu können, dass alle Felder, die vor der Transformation mit einem Kandidaten initialisiert wurden, jetzt mit einer Instanz der `Real`-Klasse initialisiert werden. Die Konstruktoren der `Abstract`- und der der `Null`-Klasse werden durch Konstruktoren ohne spezielle Funktion ersetzt. Da jeder Konstruktor seinen Superkonstruktor aufruft, müssen anschließend noch die `CALLS`- und `AGGREGATED_CALLS`-Beziehungen korrekt hinzugefügt werden. Zur korrekten Zuordnung der Mutterklasse des Konstruktors müssen die entsprechenden Properties (`fqn`, `displayname`, etc.) der Konstruktorknoten angepasst werden. Dies geschieht durch einfache String-Transformationen.

Für den Transfer der Methoden nutzen wir eine ähnliche Vorgehensweise, nachdem wir sie mit der Abfrage aus Listing 3.8 identifiziert haben. Da nur die Methoden der `Real`-Klasse eine Funktion besitzen, übernehmen wir die `Method`-Knoten der Kandidaten-Klasse mitsamt Kontrollfluss und übertragen sie dem `Real`-Knoten. Dafür werden Beziehungen und Properties angepasst. Für weitere Schritte ist es wichtig, die Sichtbarkeit der Methode, also ihr `visibility`-Property zu überprüfen. Eine private Methode kann nur innerhalb der selben Methode aufgerufen werden. Weder die `Abstract`- noch die `Null`-Klasse benötigen eine solche Methode. Damit sie aus Methoden anderer Klassen aufgerufen werden können, müssen `public`-Methoden für beide Klassen ebenfalls erstellt werden. Die Properties der Methode des `Real`-Knotens werden für die beiden neuen Knoten kopiert. Lediglich das `isabstract`-Property für die abstrakte Methode und die für die Zuordnung relevanten Properties wie `fqn` und `displayname` werden angepasst. Für die Methode des `Null`-Knotens erstellen wir einen Kontrollfluss, damit Parameter erkannt und ein Rückgabewert erzeugt werden können. Den Typ des Rückgabewerts können wir dem `returntype`-Property der Methode entnehmen. Die eingehenden `MethodCall`- und `MethodCallWithReturnValue`- Beziehungen der `Real`-Methode werden abschließend an die `Abstract`-Methode übertragen, da diese ausgehend vom initialisierten Typ des Feldes entweder die `Null`-Methode oder die `Real`-Methode aufruft.

Listing 3.8. Abfrage zur Identifizierung der Methoden eines Kandidaten

```
1 MATCH (candidate:Class)-[:CONTAINS_METHOD]->(method:Method)
2     WHERE id(candidate) = $candidateId
3 RETURN method
```

Zum Abschluss der Transformation der Kandidaten-Klasse werden noch alle in der Kandidatenklasse deklarierten Felder an die `Real`-Klasse übertragen. Die beiden anderen Klassen benötigen die Felder nicht, da sie entsprechend des Nullobjekt-Musters nicht initialisiert werden oder nicht darauf zugreifen.

3.2.3. Transformation der Aufruferklassen

Bei der Transformation nach dem Nullobjekt-Muster muss nicht nur die Kandidatenklasse, sondern auch die Aufruferklasse angepasst werden, die ein entsprechendes Feld enthält. Die Felder werden mit der `MATCH`-Abfrage aus Abschnitt 3.1 mitsamt ihrer Aufrufe in `if`-Anweisungen identifiziert und als Liste zwischengespeichert. Diese Informationen nutzen wir, um den Kontrollfluss der `if`-Anweisungen dahingehend abzuändern, dass die Abfrage, ob das entsprechende Feld `null` ist, entfernt wird. Ist diese Abfrage die einzige Bedingung, so folgt der `Condition`-Knoten der Bedingung direkt auf einen `NopStmt`-Knoten mit dem Property `nopkind = IF_COND`. Ist dies der Fall, so können wir alle Knoten, die zur `if`-Anweisung und dem `else`-Block gehören, löschen und den Kontrollfluss des `then`-Blocks an der entsprechenden Stelle der Methode einsetzen. Sind noch andere Bedingungen in der `if`-Anweisung vorhanden, erkennen wir dies dadurch, dass der `NopStmt`-Knoten entweder kein `nopkind`-Property oder dieses einen Wert von `IF_COND_X` hat. In diesem Fall löschen wir nur die Bedingung aus dem Kontrollfluss der `if`-Anweisung.

Zum Abschluss muss noch der Typ aller Felder der gerade transformierten Kandidaten angepasst werden, was wir durch eine Änderung des `vartype`-Properties erreichen. In Abbildung Abbildung 3.6 demonstrieren wir dies anhand der Felder `cache` und `cache2`. Der neue Typ des Feldes entspricht dem FQN der `Abstract`-Klasse. Die Initialisierung der Felder geschieht entweder mit dem Typ der `Real`-Klasse oder der `Null`-Klasse. Dies wird jedoch bereits während der Konstruktor-Transformation dieser Klassen durchgeführt.

Alle Felder, die bisher nicht beim Aufruf des Konstruktors der Aufruferklasse initialisiert werden, können anhand einer fehlenden `AGGREGATED_FIELD_WRITE`-Beziehung erkannt werden (siehe Listing 3.9, Z. 3). In der Abbildung 3.6 fehlt beispielsweise dem Feld `cache2` vor der Transformation die entsprechende Beziehung. Alle Felder, denen diese Beziehung fehlt, initialisieren wir durch Hinzufügen des entsprechenden Konstruktoraufrufs und der entsprechenden Zuweisung im Graphmodell mit einer Instanz der Null-Klasse.

Listing 3.9. Abfrage zur Erkennung nicht initialisierter Felder einer Aufruferklasse

```
1 MATCH (field:Field {vartype: $vartype})<-[:CONTAINS_FIELD]-(callerClass:Class)
2     WHERE id(callerClass) = $callerClassId
3         AND NOT (field)<-[:AGGREGATED_FIELD_WRITE]-(:Constructor)<-[:CONTAINS_CONSTRUCTOR]-(
              callerClass)
4 RETURN field
```

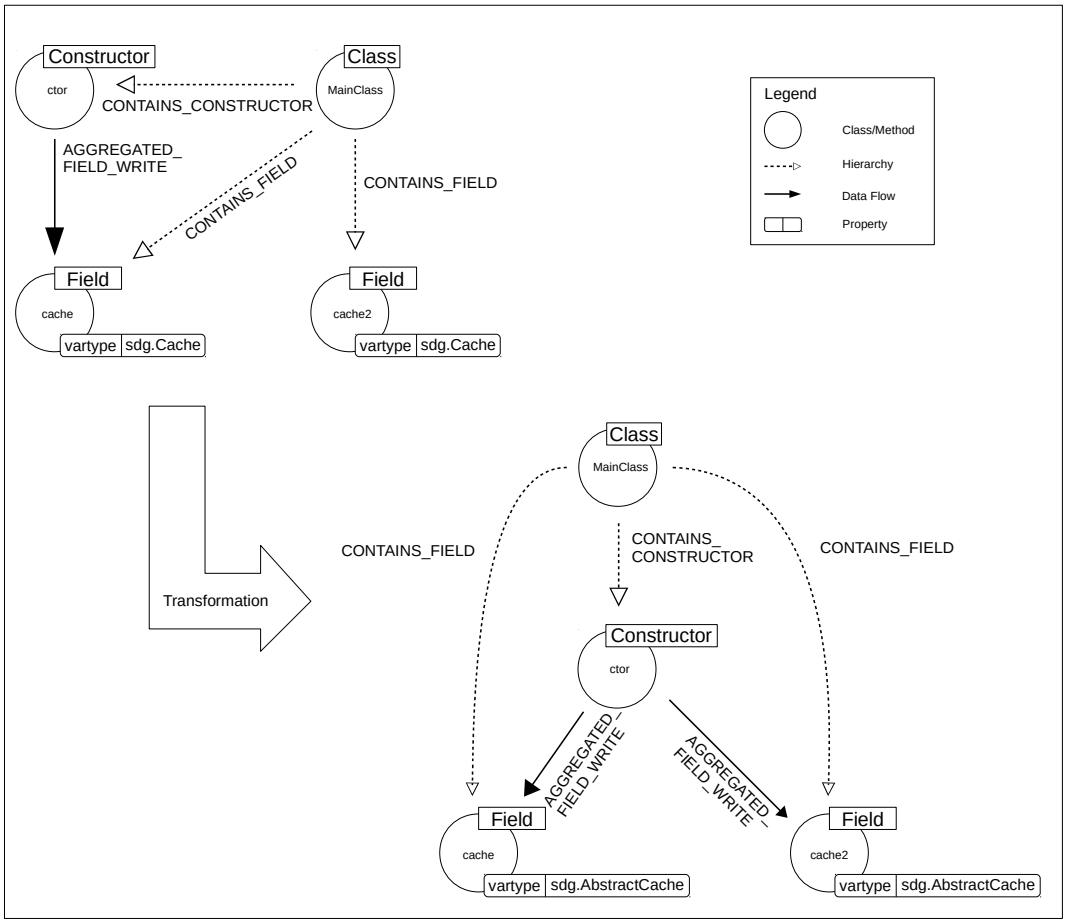

Abbildung 3.6. Aktualisierung und Initialisierung der Felder

Kapitel 4

Evaluation

4.1. Evaluation der Erkennung

Dieses Kapitel dient der Evaluation der Vollständigkeit und Korrektheit der von uns entwickelten `MATCH`-Abfrage zur Erkennung von Kandidaten für das Nullobjekt-Muster. Das Experiment und die bei unseren Tests verwendete Hardware beschreiben wir in Abschnitt 4.1.1. In Abschnitt 4.1.2 präsentieren und diskutieren wir die Ergebnisse des Experiments. Anschließend beschäftigen wir uns in Abschnitt 4.1.3 mit möglichen Beeinträchtigungen der Validität unserer Vorgehensweise.

4.1.1. Methodik

Beschreibung des Experiments

Für die Evaluation der Erkennung wandeln wir den Quellcode eines Programms mit *Java2Neo4J* in die von *PARROT* verwendete Form des JSysDG um (siehe Abschnitt 2.1.2). Auf die dadurch erstellte Datenbank wenden wir eine abgewandelte Form unserer `MATCH`-Abfrage (siehe Listing 4.1) an, die es uns erlaubt, in der Datenbank gezielt nach einzelnen Feldern zu suchen. Die zusätzliche Zeile 3 beschränkt die Felder der potentiellen Kandidaten auf eine bestimmte Klasse, welche durch die `customFqn` identifiziert wird. Wollen wir nur nach einem bestimmten Feld suchen, so können wir dies durch das Property `candidateField.name` erreichen. In Zeile 10 ergänzt eine zusätzliche `MATCH`-Abfrage die gesamte Abfrage. Diese identifiziert zu jeder `if`-Anweisung einer Methode, die das Feld auf seine Initialisierung überprüft, den Pfad zum `Method`-Knoten der Anweisung. Der Rückgabewert der Abfrage enthält im Falle einer Übereinstimmung das Feld, seine Klasse und einen Verweis auf die `if`-Anweisung mitsamt dem Pfad zur enthaltenden Methode.

In den zwei in Abschnitt 4.1.1 vorgestellten Szenarien werden wir ausgewählte Felder mit dieser abgeänderten Abfrage in der Datenbank suchen und die Ergebnisse auf ihre Korrektheit überprüfen. Dafür werden wir durch textuelle Analyse des Quellcodes und Betrachtung des SDG nachvollziehen, ob die gefundenen Felder tatsächlich in unser Schema passen und ob alle der zurückgegebenen Verweise zu Methoden korrekt und vollständig sind.

Listing 4.1. Abgeänderte `MATCH`-Query

```
1  MATCH (mainClass:Class)-[:CONTAINS_FIELD]->(candidateField:Field {isfinal:false})<-[:
       AGGREGATED_FIELD_READ]-(method:Method)
2      USING INDEX candidateField:Field(isfinal)
3      WHERE candidateField.vartype='customFqn' AND candidateField.name='customName'
4  MATCH (candidateField)-[:DATA_FLOW]->(condVariable:Assignment)-[:DATA_FLOW]->(condition:
       Condition {operation:"!="})
5      WHERE condition.operand1 = "null" OR condition.operand2 = "null"
6  MATCH (condVariable)<-[:CONTROL_FLOW]-(ifStmt:NopStmt)
7      WHERE ifStmt.nopkind = "IF_COND" OR ifStmt.nopkind = "IF_COND_X" OR (ifStmt) <-[:
           CONTROL_FLOW]- (:Condition)
8  MATCH (candidate:Class)
9      WHERE candidate.fqn = candidateField.vartype AND candidate.isabstract = false
10 MATCH p = ((condVariable) <-[:DATA_FLOW]- (:Assignment {operation:'thisdeclaration'}) <-[:
       CONTAINS_UNIT]- (:Method))
11 RETURN DISTINCT candidateField, condVariable, candidate, p
```

Szenarien

Das Experiment besteht aus zwei Szenarien, die sich auf unterschiedliche Ursprünge der verwendeten Quellcodes beziehen. Das erste Szenario (Szenario 1) wird auf einer *Neo4J*-Datenbank ausgeführt, die aus von uns selbst entwickeltem Quellcode generiert wird. Dieser besteht aus der Aufruferklasse `MainClass` (siehe Listing 4.2) und den Kandidaten-Klassen `Cache` (siehe Listing 4.3), `Text` (siehe Listing 4.4) und `AbstractObject` (siehe Listing 4.5). Die letzte dieser Klassen ist abstrakt und enthält nur die abstrakte Methode `getAbstractObjectName()`. Die Aufruferklasse besitzt mehrere Felder aller Kandidatenklassen. Diese Felder werden in den Methoden der Aufruferklasse überprüft und aufgerufen.

Das zweite Szenario (Szenario 2) wird auf einer *Neo4J*-Datenbank ausgeführt, die mit *Java2Neo4J* aus dem Quelltext des Analysetools *Apache Ant*[1] in der Version 1.9.9 generiert wird. Wir testen vier ausgewählte Methoden aus dem Quellcode von *Ant*, da eine vollständige Evaluation der 40 gefundenen Kandidaten im Umfang dieser Arbeit nicht möglich ist. Die von uns gewählten Klassen erfüllen verschiedene Kriterien, die wir im Umfeld einer realen Anwendung überprüfen wollen. Für die Evaluation der Vollständigkeit unserer CMQ führen wir eine textuelle Suche nach Kandidaten im Quellcode von *Ant* durch. Da auch diese sehr umfangreich ist und den Rahmen dieser Arbeit sprengen würde, halten wir unsere Ergebnisse rein textuell fest.

Szenario 1: Evaluation mit selbst erstelltem Quellcode

Mit Hilfe der drei Kandidaten-Klassen testen wir, ob unsere CMQ mehrere Kandidaten finden kann. Dafür erstellen wir für die Klassen `Cache` und `Text` Felder in der Aufruferklasse, die die Kriterien zur Erkennung als Kandidaten erfüllen. Da wir abstrakte Klassen nicht als Kandidaten identifizieren wollen, testen wir anhand der Klasse `AbstractObject`, ob sie als Kandidat ausgeschlossen wird, obwohl sie die anderen Kriterien erfüllt. Die Methoden

[1] `http://ant.apache.org/`

und Felder der Kandidaten sollen den SDG einer größeren Klasse simulieren. Sie dienen dem späteren Schritt der Evaluation der Transformation in Abschnitt 4.2.

Die in Listing 4.2 vorgestellte Aufruferklasse besitzt vier in den Zeilen 3-8 deklarierte Felder. Das erste Feld ist das `cache`-Feld, welches in den Methoden `createCache()` (Z. 14-18), `clearCache()` (Z. 20-24), `cacheFile()` (Z. 42-46), `postText()` (Z. 48-59) und `checkDifference()` (Z. 61-68) auf seine Initialisierung überprüft wird. Das Feld ist nicht `final` und die Aufrufe in den Methoden `clearCache()`, `cacheFile()` und `postText()` enthalten das für uns wichtige Schema '`cache != null`'. Diese Aufrufe dienen dem Test, ob ein Feld, das relevante und irrelevante Aufrufe besitzt, trotzdem erkannt wird. Wir erwarten daher, dass dieses Feld als Kandidat mit genau diesen drei Methoden zurückgegeben wird.

Das im zweiten Beispiel getestete Feld `cache2` ist zwar auch nicht `final`, wird aber nur von der Methode `checkDifference()` verwendet und dort nur auf Gleichheit mit dem Feld `cache` getestet. Daher erwarten wir, dass das Feld nicht identifiziert wird und somit auch keine übereinstimmenden Methodenaufrufe ergibt.

Das Feld `text` wird in den Methoden `createText()` (Z. 26-30) und `postText()` aufgerufen. Davon sind nur die beiden Aufrufe in der `postText()` Methode relevant. In diesem Beispiel wird getestet, ob mehrfache Aufrufe innerhalb einer Methode und mehrere Bedingungen in einer `if`-Anweisung erkannt werden. Da das Feld nicht `final` ist, erwarten wir, dass dieses Feld erkannt wird und zwei Aufrufe in der selben Methode gefunden werden.

Das vierte Feld namens `title` ist nicht `final`, besitzt aber auch keine relevanten Aufrufe. Daher sollte es nicht erkannt werden. Das gleiche erwarten wir auch von dem Feld `subtitle`, das zwar in der Methode `postSubtitle()` (Z. 36-40) einen relevanten Aufruf besitzt, aber als `final` deklariert ist und somit von uns als Kandidat ausgeschlossen wird.

Das Feld `abstractObject` besitzt einen relevanten Aufruf durch eine Methode, ist aber vom Typ `AbstractObject`. Dies ist eine abstrakte Klasse und sollte somit von der CMQ ausgeschlossen werden. Daher erwarten wir auch, dass dieses Feld nicht identifiziert wird.

Listing 4.2. Die Aufruferklasse MainClass.java

```
1  public class MainClass {
2
3      private Cache cache;
4      private Cache cache2 = new Cache();
5      private Text text;
6      private Text title;
7      private final Text subtitle;
8      private AbstractObject abstractObject;
9
10     public MainClass(){
11         subtitle = new Text();
12     }
13
14     public void createCache(){
15         if (cache == null) {
16             this.cache = new Cache();
17         }
18     }
19
20     public void clearCache(){
21         if (cache != null) {
```

```
22            cache.reset();
23        }
24    }
25
26    public void createText(){
27        if (text == null) {
28            text = new Text();
29        }
30    }
31
32    public void postTitle(){
33        System.out.println(title.getText());
34    }
35
36    public void postSubTitle(){
37        if (subtitle != null) {
38            System.out.println(subtitle.getText());
39        }
40    }
41
42    public void cacheFile(String filename){
43        if (cache != null) {
44            cache.put(filename, 0);
45        }
46    }
47
48    public void postText(){
49        if (text != null){
50            text.getText();
51        }
52        if (cache != null && text != null){
53            cache.put(text.getText(), 1);
54            text.setText("second Text");
55        }
56        createCache();
57    }
58
59    public boolean checkDifference() {
60        if (cache!=cache2) {
61            cache.reset();
62            return true;
63        } else {
64            return false;
65        }
66    }
67
68    public String getAbstractObjectName(){
69        String tmp = null;
70        if (abstractObject != null){
71            tmp = abstractObject.getName();
72        }
73        return tmp;
74    }
75
76    public static void main(String[] args) {
77        ...
78    }
79 }
```

Listing 4.3. Kandidatenklasse Cache.java

```
1  public class Cache {
2
3      public void put(String filename, int
           size) {
4          size();
5      }
6
7      public void reset() {
8          size();
9      }
10
11     private int size(){
12         return 0;
13     }
14 }
```

Listing 4.4. Kandidatenklasse Text.java

```
1  public class Text {
2
3      private String text = "Some text";
4
5      public String getText() {
6          return text;
7      }
8
9      public void setText(String text) {
10         this.text = text;
11     }
12 }
```

Listing 4.5. Kandidatenklasse AbstractObject.java

```
1 public abstract class AbstractObject {
2     public abstract String getName();
3 }
```

Szenario 2: Evaluation des SDG von Apache Ant

Beispiel 1 (Evaluation des Matchings der Ant-Klasse Union.java) Die in Listing 4.6 dargestellte Ant-Klasse `Path` enthält ein Feld der nicht abstrakten Klasse `Union` (Z. 5). Dieses wird mit `null` initialisiert und in einer Vielzahl von Methoden der Klasse `Path` aufgerufen. Jedoch enthalten nur die drei im Listing gezeigten Methoden (Z. 7, Z. 14, Z. 22) die relevante Abfrage, ob das Feld initialisiert ist. Der Aufruf in der Klasse `dieOnCircularReference(Stack-<Object >, Project)` ist dabei im `else`-Zweig einer `if`-Anweisung verschachtelt. Die CMQ sollte in der Lage sein, dies zu erkennen. Daher erwarten wir, dass das Feld und alle drei Methoden von der CMQ zurückgegeben werden.

Beispiel 2 (Evaluation des Matchings der Ant-Klasse ProjectComponent.java) Die in Listing 4.7 dargestellte Ant-Klasse `ConcatFileInputStream` enthält ein Feld der abstrakten Klasse `ProjectComponent` (Z. 5). Dessen Wert wird in der Methode `log(String, int)` (Z. 7) innerhalb einer `if`-Anweisung abgefragt, wodurch es für uns eine relevante Abfrage ist. Mit diesem Beispiel können wir überprüfen, ob ein Feld einer abstrakten Klasse auch in der realen Anwendung erfolgreich ausgeschlossen wird. Wir erwarten somit, dass diese Klasse nicht erkannt wird.

Listing 4.6. Ausschnitt der Ant-Klasse Path.java

```
1  package org.apache.tools.ant.types;
2  ...
3  public class Path extends DataType implements Cloneable, ResourceCollection {
4      ...
5      private Union union = null;
6      ...
7       public void setRefid(Reference r) throws BuildException {
8          if (union != null) {
9              throw tooManyAttributes();
10         }
11         super.setRefid(r);
12     }...
13      public void setCache(boolean b) {
14         checkAttributesAllowed();
15         cache = b;
16         if (union != null) {
17             union.setCache(b);
18         }
19     }...
20     protected synchronized void dieOnCircularReference(Stack<Object> stk, Project p)
21         throws BuildException {
22         if (isChecked()) {
23             return;
24         }
25         if (isReference()) {
26             super.dieOnCircularReference(stk, p);
27         } else {
28             if (union != null) {
29                 pushAndInvokeCircularReferenceCheck(union, stk, p);
30             }
31             setChecked(true);
32         }
33     }
34 }
```

Listing 4.7. Ausschnitt der Ant-Klasse ConcatFileInputStream.java

```
1  package org.apache.tools.ant.util;
2  ...
3  public class ConcatFileInputStream extends InputStream {
4      ...
5     private ProjectComponent managingPc;
6     ...
7      public void log(String message, int loglevel) {
8          if (managingPc != null) {
9              managingPc.log(message, loglevel);
10         } else {
11             if (loglevel > Project.MSG_WARN) {
12                 System.out.println(message);
13             } else {
14                 System.err.println(message);
15             }
16         }
17     }    ...
18 }
```

Beispiel 3 (Evaluation des Matchings der Ant-Klasse PropertySet.java) Die in Listing 4.8 dargestellte Ant-Klasse `ExpandProperties` enthält ein Feld der nicht abstrakten Klasse `PropertySet` (Z. 7). Der Wert des Feldes wird in der Methode `add(PropertySet)` (Z. 9) überprüft. Da in dieser Methode Parameter und Feld den gleichen Namen haben, findet der Feldzugriff mit `this.propertySet` statt. Die Klasse eignet sich somit für den Test, ob die CMQ in der Lage ist, diesen Feldzugriff trotzdem richtig zu erfassen. Daher erwarten wir, dass das Feld und die Klasse `PropertySet` von der CMQ identifiziert werden.

Beispiel 4 (Evaluation des Matchings der Ant-Klasse Mapper.java) Die in Listing 4.9 dargestellte Ant-Klasse `PropertySet` enthält ein Feld der abstrakten Klasse `Mapper` (Z.5). Gleiches gilt für die Klassen `UpToDate`, `PresentSelector` und `MappingSelector`. Diese sind in Listing 6.1, Listing 6.2 und Listing 6.3 dargestellt. Dies eignet sich zur Überprüfung, ob die CMQ solche Klassen als Kandidaten identifiziert, die in mehr als einer Klasse als Feld vorkommen. Für die Auswertung fassen wir die Ergebnisse zusammen und erwarten somit von der CMQ, vier Felder und die acht zugehörigen Funktionen zu identifizieren.

Experimentierumgebung

Hardware Das für die Evaluation genutzte System verfügt über einen Intel Core i7-Prozessor (2600k) mit 3,40GHz. Der Arbeitsspeicher besteht aus 16 GB DDR3 RAM und wird von einer SSD mit 250 GB Hauptspeicher ergänzt.

Software Unser System wird über die Windows 10 Home 64-Bit-Variante der Version 1703 betrieben. Die Ausführung unserer Transformationen erfolgt unter Benutzung der Eclipse IDE in der Version Neon.3 Release (4.6.3). Für die Transformation von Quellcode zum SDG verwenden wir Java2Neo4J, welches mit einer Java-Umgebung der Version 7 (Oracle jdk1.7.0_80) und der Community-Edition der Neo4J-Datenbank in Version 2.3.5 ausgeführt wird. Die Erkennung der Kandidaten erfolgt unter der Java-Version 8 (Oracle jre1.8.0_-112) und der Neo4J-Community-Edition in Version 3.2.3. Der Verwendete Neo4J-Browser entstammt ebenfalls dieser Version.

Listing 4.8. Ausschnitt der Ant-Klasse ExpandProperties.java

```
1  package org.apache.tools.ant.filters;
2  ...
3  public final class ExpandProperties extends BaseFilterReader implements ChainableReader {
4      ...
5      private PropertySet propertySet;
6      ...
7      public void add(PropertySet propertySet) {
8          if (this.propertySet != null) {
9              throw new BuildException("expandproperties filter accepts only one propertyset");
10         }
11         this.propertySet = propertySet;
12     }...
13 }
```

Listing 4.9. Ausschnitt der Ant-Klasse PropertySet.java

```
1  package org.apache.tools.ant.types;
2  ...
3  public class PropertySet extends DataType implements ResourceCollection {
4     private Mapper mapper;
5     ...
6     public Mapper createMapper() {
7           assertNotReference();
8           if (mapper != null) {
9               throw new BuildException("Too many <mapper>s!");
10          }
11          mapper = new Mapper(getProject());
12          setChecked(false);
13          return mapper;
14      }...
15      protected synchronized void dieOnCircularReference(Stack<Object> stk, Project p)
16          throws BuildException {
17          if (isChecked()) {
18              return;
19          }
20          if (isReference()) {
21              super.dieOnCircularReference(stk, p);
22          } else {
23              if (mapper != null) {
24                 pushAndInvokeCircularReferenceCheck(mapper, stk, p);}
25              for (PropertySet propertySet : setRefs) {
26                  pushAndInvokeCircularReferenceCheck(propertySet, stk, p);
27              }
28              setChecked(true);
29          }
30      }...
31 }
```

4.1.2. Ergebnisse und Diskussion

Szenario 1: Evaluation mit selbst erstelltem Quellcode

Tabelle 4.1 zeigt die Ergebnisse des Musterabgleichs der in Listing 4.1 formulierten CMQ. In allen vorgestellten Beispielen wurden die korrekten Felder gefunden. Wenn kein Feld identifiziert wird, so wird von unserer CMQ auch keine Methode ausgegeben. In diesem Fall können wir also keine Aussage über die Korrektheit der gefundenen Methoden treffen. Für unseren Ansatz bedeutet dieses Ergebnis, dass die beiden Klassen `Cache` und `Text` zur Transformation ausgewählt werden, da von beiden mindestens ein Feld als Kandidat identifiziert wurde. Die Klasse `AbstractObject` erfüllt die Voraussetzungen zur Transformation nicht.

Szenario 2: Evaluation des SDG von Apache Ant

Tabelle 4.2 zeigt die Ergebnisse der Tests. Für die ersten drei Tests fallen die Ergebnisse wie erwartet aus. Im Fall der `Mapper`-Klasse findet die CMQ jedoch eine zusätzliche Methode,

Tabelle 4.1. Evaluation anhand unseres eigenen Quellcodes aus Szenario 1

Klasse	Feldname	Übereinstimmung erwartet	Korrektes Matching	
			des Feldes	der Methoden
`Cache`	`cache`	ja	✓	✓
	`cache2`	nein	✓	-
`Text`	`text`	ja	✓	✓
	`title`	nein	✓	-
	`subtitle`	nein	✓	-
`AbstractObject`	`abstractObject`	nein	✓	-

die auf eines der Felder zugreift. Das liegt daran, dass das entsprechende Feld der Klasse `MappingSelector` als `protected` deklariert ist. Somit kann eine Methode der Subklasse `DependentSelector` auf das Feld zugreifen. Genauer betrachtet ist die Rückgabe der Methode sinnvoll für unseren Ansatz. Wenn die Klasse `Mapper` transformiert wird, so sollen alle überflüssigen Aufrufe in `if`-Anweisungen entfernt werden, die auf besagte Felder zugreifen. Da die Transformation dafür mit der Ergebnismenge der Methodenaufrufe arbeitet, wird die Bedingung somit trotz der Tatsache, dass sie in einer anderen Klasse ausgeführt wird, entfernt. Für Felder, die als `public` deklariert sind, gilt dies ebenfalls.

Bei der textuellen Analyse des Quellcodes finden wir weitaus mehr Kandidaten als unsere CMQ. Grundlegend dafür ist die Erstellung des SDG durch *Java2Neo4J*. Das Tool erstellt nur aus tatsächlich aufgerufenen Teilen des Programms den entsprechenden SDG. Schränken wir unsere manuelle Analyse auf die Teile von *Ant* ein, die auch in der Datenbank vorhanden sind, so finden wir neben den Feldern der 40 Kandidaten unserer CMQ nur noch wenige weitere Kandidatenfelder. Diese lassen sich mit ihrem Typ in drei Gruppen unterteilen: Abstrakte Klassen, Interfaces und Arrays. Die abstrakten Klassen und Interfaces schließen wir bewusst in unserer CMQ aus. Die abstrakten Klassen können wir noch nicht sinnvoll transformieren. Interfaces sind in unserer Datenbankdarstellung nicht implementiert. Daher haben entsprechende Felder in der Datenbank keine zugehörige Klasse. Die Felder vom Typ eines Arrays werden aus diesem Grund ebenfalls nicht erkannt, da ein Feld vom Typ `String[]` auf eine nicht vorhandene Klasse des selben Typs verweist. Aus den genannten Gründen sind Änderungen am Datenbankmodell von *Java2Neo4J* erforderlich, um eine korrekte Identifikation zu ermöglichen. Da wir während der textuellen Analyse auf die 40 von der CMQ gefundenen Kandidaten stoßen, können wir deren Korrektheit annehmen.

4.1.3. Beeinträchtigungen der Validität

Interne Validität Die von uns verwendete Version von *Java2Neo4J* unterstützt aufgrund der Abhängigkeit von *Soot*[2] noch kein *Java 8* und folgende Versionen. Somit können wir bei

[2] `https://sable.github.io/soot/`

Tabelle 4.2. Evaluation anhand des Ant-Quellcodes

Klasse	Feldname	Übereinstimmung erwartet	Korrektes Matching	
			des Feldes	der Methoden
`Union`	`union`	ja	✓	✓
`ProjectComponent`	`managingPc`	nein	✓	-
`PropertySet`	`propertySet`	ja	✓	✓
`Mapper`	mapper	ja	✓	+1

unseren CMQ nur Programme berücksichtigen, die bis inklusive *Java 7* entwickelt wurden. Somit können wir nicht garantieren, dass die Graphen zukünftig mit gleichen Knoten und Kantentypen und auf die gleiche Weise generiert werden.

Wie auch der in Abschnitt 2.1.2 vorgestellte *JSysDG* von Walkinshaw u. a. [2003] ist der von uns generierte SDG nur eine Annäherung an den tatsächlichen SDG eines Programms. Daher könnte der tatsächliche SDG, der unseren Szenarien zugrunde liegt, eine andere Form als die von uns angenommene haben.

Externe Validität Wir haben versucht möglichst viele Szenarien und Varianten von Methoden und Feldern zu überprüfen. Jedoch lassen es die begrenzte Zeit und der Umfang dieser Thesis nur zu, eine gewisse Anzahl an Szenarien und Kandidaten näher zu betrachten. Durch die schnell wachsende Größe und Komplexität von Graphen im Verhältnis zur Größe von Klassen und Methoden ist die manuelle und visuelle Überprüfung sehr zeitaufwendig. In anderen Szenarien, also bei der Ausführung mit anderen Programmen, können Strukturen auftreten, die wir bisher noch nicht bedacht haben. Wir können daher nicht garantieren, alle Möglichkeiten für Kandidaten vollständig erkannt zu haben. Jedoch bietet sich durch die Modifizierbarkeit der CMQ die Option, weitere Muster in die CMQ einzubinden.

4.2. Evaluation der Transformation

In diesem Kapitel widmen wir uns der Evaluation der Vollständigkeit und Korrektheit unserer Transformation der gefundenen Kandidaten nach dem Nullobjekt-Muster. Das Experiment und die bei unseren Tests verwendete Hardware beschreiben wir in Abschnitt 4.2.1. In Abschnitt 4.2.2 präsentieren und diskutieren wir die Ergebnisse des Experiments. Anschließend beschäftigen wir uns in Abschnitt 4.2.3 mit möglichen Beeinträchtigungen der Validität unserer Vorgehensweise.

4.2.1. Methodik

Beschreibung des Experiments

Zur Evaluation der Vollständigkeit unserer Transformation stellen wir einen Vorher-Nachher-Vergleich unseres Beispielprogramms mit seiner transformierten Version an. Da die SDGs beider Programme sehr umfangreich sind, gestaltet es sich als umständlich, den Vergleich durch ihre Betrachtung durchzuführen. Daher benutzen wir das Tool *Sdg2Java*, um den SDG des automatisch transformierten Programms in Java-Quellcode zu übersetzen. Die Korrektheit der einzelnen Schritte der Transformation evaluieren wir durch einen textuellen Vergleich mit einer manuell umgewandelten Variante des Programms.

Da *Sdg2Java* sich noch in der Entwicklung befindet, kann es bei der Generierung von Quellcode zu einigen Fehlern kommen. Wir wollen bei unserem Vergleich Fehler ausschließen, die während dieses Prozesses entstehen können. Daher übersetzen wir die manuell transformierte Version ebenfalls einmal mit *Java2Neo4J* in einen SDG und anschließend mit *Sdg2Java* wieder in Quellcode-Format.

Unsere Vorgehensweise verdeutlichen wird in Abbildung 4.1. Das Ursprungsprogramm (1) enthält die Testklasse `MainClass` und die dazugehörigen Kandidatenklassen `Cache`, `Text` und `AbstractObject`. Diese werden in Listing 4.2, Listing 4.3, Listing 4.4 und Listing 4.5 aufgezeigt. Die manuell transformierte Version dieses Programms (2) ist im Anhang (Kapitel 6) ab Listing 6.4 zu finden. Die nach der Transformation erwarteten Klassen (3) werden durch die Anwendung der Tools *Java2Neo4J* und *Sdg2Java* gewonnen. Die entsprechenden Klassen werden von uns in Abschnitt 4.2.1 eingeführt. Den Vergleich zwischen unserem tatsächlichen, bei der automatischen Transformation erhaltenen (4), und dem erwarteten Code (3) stellen wir in Abschnitt 4.2.2 an.

Szenario

Zunächst schauen wir uns die Kandidatenklassen `Cache` und `Text` an. Die Klasse `AbstractObject` lassen wir an dieser Stelle aus, da sie nicht als Kandidat identifiziert (siehe Abschnitt 4.1.2) und somit nicht transformiert wird.

Die Klasse `Cache` wird während der Erkennung der Kandidaten erfolgreich identifiziert. Daher erwarten wir, dass bei der Transformation die Klassen `NullCache` (Listing 4.10),

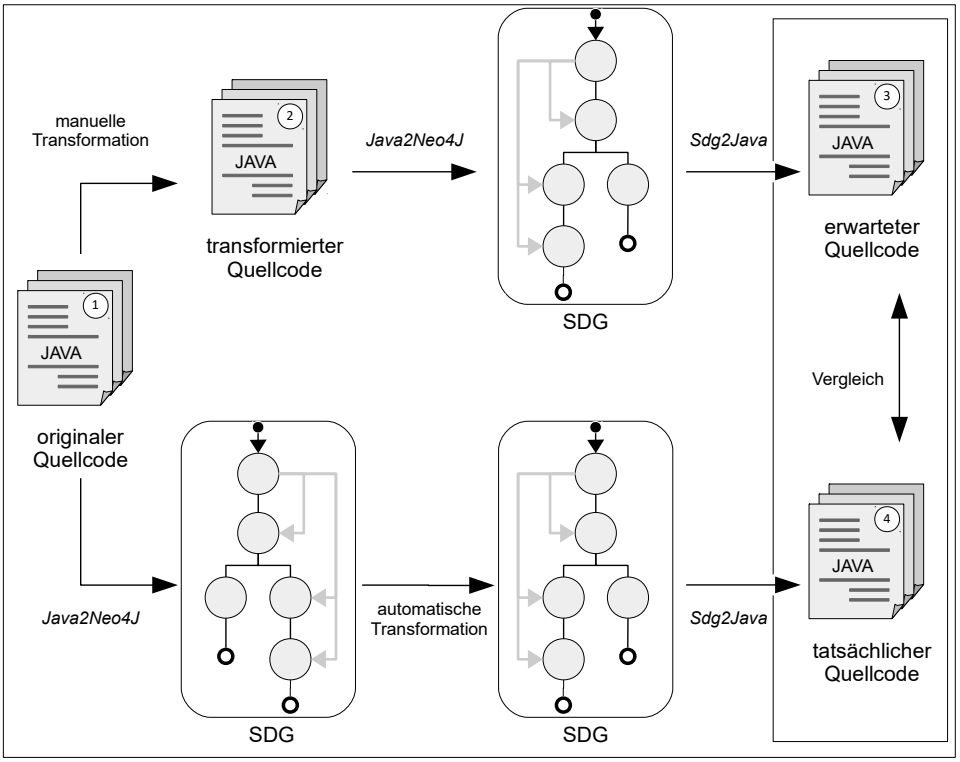

Abbildung 4.1. Ansatz der Evaluation der Transformation

`RealCache` (Listing 4.11) und `AbstractCache` (Listing 4.12) erstellt werden. Mit den Methoden `put()` und `reset()` (Listing 4.3 Z. 3, 7) testen wir, ob der Körper der Methoden bei der Transformation korrekt in die Methode der `Real`-Klasse übertragen und für die `Null`- und `Abstract`-Klasse ein eigener bzw. kein Körper hinzugefügt werden. Außerdem benötigt `put()` die Parameter `filename` und `size`. Deren korrekte Erstellung in den Methoden der neuen Knoten können wir somit testen. Die Methode `size()` (Z. 11) hilft uns bei der Evaluation, ob andere Rückgabewerte als `void` erkannt und übertragen werden. Außerdem können wir überprüfen, ob private Methoden nur für den `Real`-Knoten hinzugefügt werden.

Die Klasse `Text` (Listing 4.4) wird während der Erkennung der Kandidaten ebenfalls erfolgreich identifiziert. Daher erwarten wir, dass bei der Transformation die Klassen `NullText` (Listing 4.14), `RealText` (Listing 4.15) und `AbstractText` (Listing 4.13) erstellt werden. Unsere Klasse besitzt das Feld `text` (Listing 4.4 Z. 3) und die zwei Methoden `getText()` (Z. 5) und `setText()`(Z. 9). Diese dienen dem Test, ob Felder korrekt an den Real-Knoten übergeben werden. Außerdem können wir testen, ob die Methode `getText()` im `Null`-Knoten den neuen Standardwert `null` zurückgibt.

Listing 4.10. Die erwartete Klasse NullCache (vgl. Listing 6.4)

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class NullCache extends de.tnr.sdg.
       example.transformedCache.
       AbstractCache {
4
5      public NullCache() {
6          super();
7      }
8
9      public void reset() {}
10
11     public void put(java.lang.String
           filename,int size) {}
12 }
```

Listing 4.11. Die erwartete Klasse RealCache (vgl. Listing 6.5)

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class RealCache extends de.tnr.sdg.
       example.transformedCache.
       AbstractCache {
4
5      public RealCache() {
6          super();
7      }
8
9      private final int size() {
10         return 0;
11     }
12
13     public void reset() {}
14
15     public void put(java.lang.String
           filename,int size) {}
16 }
```

Listing 4.12. Die erwartete Klasse AbstractCache (vgl. Listing 6.6)

```
1  package de.tnr.sdg.example.transformedCache;
2
3  public class AbstractCache extends java.lang.Object {
4
5      public void put() {}
6
7      public void reset() {}
8
9      public AbstractCache() {
10         super();
11     }
12 }
```

Listing 4.13. Die erwartete Klasse AbstractText (vgl. Listing 6.9)

```
1  package de.tnr.sdg.example.transformedCache;
2
3  public class AbstractText extends java.lang.Object {
4
5      public void setText() {}
6
7      public java.lang.String getText() {}
8
9      public AbstractText() {
10         super();
11     }
12 }
```

Listing 4.14. Die erwartete Klasse NullText (vgl. Listing 6.7)

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class NullText extends de.tnr.sdg.
       example.transformedCache.AbstractText
       {
4
5      public NullText() {
6          super();
7      }
8
9      public void setText(java.lang.String
           text) {}
10
11     public java.lang.String getText() {
12         return null;
13     }
14 }
```

Listing 4.15. Die erwartete Klasse RealText (vgl. Listing 6.8)

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class RealText extends de.tnr.sdg.
       example.transformedCache.AbstractText
       {
4
5      private String text;
6
7      public RealText() {
8          super();
9          text = "Some text";
10     }
11
12     public void setText(java.lang.String
           text) {
13         text = text;
14     }
15
16     public java.lang.String getText() {
17         return this.text;
18     }
19 }
```

Durch die Änderungen der Kandidatenklassen sind auch die in Abschnitt 3.2 aufgeführten Änderungen an den Aufruferklassen notwendig. Die nach der Transformation der ursprünglichen Aufruferklasse `MainClass` (Listing 4.2) erwartete Klasse zeigen wir in Listing 4.16. Neben der Evaluation der Erkennung von Kandidaten dienen die Felder und Methoden auch dem Test, ob die Transformation der Aufruferklassen korrekt funktioniert. An den Feldern (Z. 5-10) können wir überprüfen, ob die Typen der Felder eines erkannten Kandidaten erfolgreich in die `Abstract`-Klasse geändert werden.

Die Kandidatenfelder `subtitle` und `cache2` werden bei der Erstellung eines neuen Objektes der `MainClass` initialisiert. Daher erwarten wir, dass in der transformierten Variante eine Initialisierung mit der `Real`-Klasse stattfindet (Listing 4.16 Z. 75, 81). Von den anderen Feldern einer Kandidatenklasse `cache`, `text` und `title` erwarten wir eine Initialisierung mit der `Null`-Klasse.

Die Methoden `clearCache()` (Z. 59), `postSubtitle()` (Z. 44), `cacheFile(String)` (Z. 40) und `postText()` (Z.32) enthalten relevante Aufrufe auf die Kandidatenfelder. Wir erwarten, dass die `if`-Anweisungen entsprechend der Vorgaben des Nullobjekt-Musters geändert oder entfernt werden. Alle anderen Methoden sollten während der Transformation keine Änderungen erfahren.

Listing 4.16. Die erwartete Aufruferklasse MainClass (vgl. Listing 6.10)

```
1  package de.tnr.sdg.example.transformedCache;
2
3  public class MainClass extends java.lang.Object {
4
```

```
5     private de.tnr.sdg.example.transformedCache.AbstractObject abstractObject;
6     private final de.tnr.sdg.example.transformedCache.AbstractText subtitle;
7     private de.tnr.sdg.example.transformedCache.AbstractText title;
8     private de.tnr.sdg.example.transformedCache.AbstractText text;
9     private de.tnr.sdg.example.transformedCache.AbstractCache cache2;
10    private de.tnr.sdg.example.transformedCache.AbstractCache cache;
11
12    public static void main(java.lang.String[] args) {...}
13
14    public java.lang.String getAbstractObjectName() {
15       java.lang.String tmp = null;
16       if (this.abstractObject != null) {
17          tmp = this.abstractObject.getName();
18       }
19       return tmp;
20    }
21
22    public boolean checkDifference() {
23       if (this.cache != this.cache2) {
24          this.cache.reset();
25          return 1;
26       }
27       return 0;
28    }
29
30    public void postText() {
31       java.lang.String testString = "";
32       testString = this.text.getText();
33       this.cache.put(this.text.getText(), 1);
34       this.text.setText(testString);
35       this.createCache();
36    }
37
38    public void cacheFile(java.lang.String filename) {
39       this.cache.put(filename, 0);
40    }
41
42    public void postSubtitle() {
43       System.out.println(this.subtitle.getText());
44    }
45
46    public void postTitle() {
47       System.out.println(this.title.getText());
48    }
49
50    public void createText() {
51       if (this.text == null) {
52          <init>();
53          text = new de.tnr.sdg.example.transformedCache.RealText;
54       }
55    }
56
57    public void clearCache() {
58       this.cache.reset();
59    }
60
61    public void createCache() {
62       if (this.cache == null) {
63          <init>();
64          cache = new de.tnr.sdg.example.transformedCache.RealCache;
65       }
66    }
```

```
67
68      public MainClass() {
69          super();
70          <init>();
71          cache = new de.tnr.sdg.example.transformedCache.NullCache;
72          <init>();
73          cache2 = new de.tnr.sdg.example.transformedCache.RealCache;
74          <init>();
75          text = new de.tnr.sdg.example.transformedCache.NullText;
76          <init>();
77          title = new de.tnr.sdg.example.transformedCache.NullText;
78          <init>();
79          subtitle = new de.tnr.sdg.example.transformedCache.RealText;
80      }
81  }
```

Experimentierumgebung

Hardware Das für die Evaluation genutzte System verfügt über einen Intel Core i7-Prozessor (2600k) mit 3,40GHz. Der Arbeitsspeicher besteht aus 16 GB DDR3 RAM und wird von einer SSD mit 250 GB Hauptspeicher ergänzt.

Software Unser System wird über die Windows 10 Home 64-Bit-Variante der Version 1703 betrieben. Die Ausführung unserer Transformationen erfolgt unter Benutzung der Eclipse IDE in der Version Neon.3 Release (4.6.3). Für die Transformation von Quellcode zum SDG verwenden wir Java2Neo4J, welches mit einer Java-Umgebung der Version 7 (Oracle jdk1.7.0_80) und der Community-Edition der Neo4J-Datenbank in Version 2.3.5 ausgeführt wird. Die Erkennung und Transformation der Kandidaten erfolgen unter der Java-Version 8 (Oracle jre1.8.0_112) und der Neo4J-Community-Edition in Version 3.2.3. Der Verwendete Neo4J-Browser entstammt ebenfalls dieser Version. Die Umwandlung des SDG zum *Sdg2Java*, das ebenfalls mit Java 8 und Neo4J in Version 3.2.3 ausgeführt wird.

4.2.2. Ergebnisse und Diskussion

Die Transformation der Kandidaten wird in Abschnitt 3.2 in die Transformation der Kandidatenklassen und die Transformation der Aufruferklassen unterteilt. Aus Gründen der Übersichtlichkeit werden wir dieses Vorgehen im Folgenden ebenfalls übernehmen. In Abschnitt 4.2.2 evaluieren wir die Transformation der Kandidatenklassen `Cache` und `Text`. Anschließend betrachten wir in 4.2.2 die Transformation der Aufruferklasse `MainClass`.

Transformation der Kandidatenklassen

Wie erwartet werden bei der Transformation des SDG aus den Kandidaten die Klassen `NullCache`, `RealCache`, `AbstractCache`, `NullText`, `RealText` und `AbstractText` erstellt. Diese betrachten wir im Folgenden genauer.

NullCache In Listing 4.17 präsentieren wir die durch unsere Transformation erstellte Klasse `NullCache`. Wie in unserer erwarteten Klasse (Listing 4.10) werden der Konstruktor (Z. 5) und die Methoden `put()` (Z. 9) und `reset()` (Z. 12) erstellt. Die Sichtbarkeit und Attribute der Klasse, sowie die Vererbung von der Klasse AbstractCache (Z. 3) stimmen mit der Zielklasse überein. In der manuell erstellten Variante der Klasse (Listing 6.4) kommt kein Konstruktor vor. Dies lässt sich dadurch erklären, dass es nicht explizit notwendig ist, einen Konstruktor zu definieren. Dennoch ist es nicht falsch, wenn der Konstruktor ebenso wie der `super()`-Aufruf in Zeile 6 von *Sdg2Java* explizit hinzugefügt werden. Der Konstruktor hat keine weiteren Funktionen, wodurch er korrekt implementiert ist.

Die Methoden `put()` und `reset()` stimmen in Rückgabewert und Anzahl der Parameter mit den Methoden der Kandidatenklasse (Listing 4.3) überein. Die Typen der Parameter sind die gleichen wie in der Ursprungsklasse. Lediglich ihre temporären Namen weichen ab, was nur die Lesbarkeit des Quellcodes für den Menschen erschwert. Entsprechend dem Nullobjekt-Muster ist es auch korrekt, dass die Methoden einen leeren Körper haben.

Da *Sdg2Java* keine `import`-Anweisungen erzeugt, werden die Verweise auf die nicht primitiven Klassen `AbstractCache` (Z. 3) und `String` (Z. 9) komplett ausgeschrieben.

Listing 4.17. Die tatsächliche Klasse NullCache

```
1  package de.tnr.sdg.example.cache;
2
3  public class NullCache extends de.tnr.sdg.example.cache.AbstractCache {
4
5      public NullCache() {
6          super();
7      }
8
9      public void put(java.lang.String arg0,int arg1) {}
10
11     public void reset() {}
12 }
```

RealCache Die in unserem Ansatz transformierte Klasse `RealCache` zeigen wir in Listing 4.18. Abgesehen von dem geänderten Paketnamen ist diese Klasse mit der in Listing 4.11 gezeigten erwarteten identisch. Auffällig ist jedoch, dass die Aufrufe der `size()`-Methode durch die Methoden `put(String, int)` (Z. 13) und `reset()` (Z. 16) nicht übernommen wird. Ein kurzer Blick auf den in fett dargestellten Kontrollfluss der automatisch transformierten Methode (Abbildung 6.1) und der manuell transformierten Methode (Abbildung 6.2) bestätigt uns jedoch in der Annahme, dass dieser Methodenaufruf von *Sdg2Java* entfernt wurde. Ein weiterer Unterschied ist der `final`-Modifizierer der Methode `size()` (Z. 5), der jedoch schon bei der ersten Transformation mit *Java2Neo4J* hinzugefügt wird. Da keiner dieser Unterschiede durch unsere Transformation hervorgerufen wird, können wir also davon ausgehen, dass die Klasse `RealCache` korrekt transformiert wird.

Listing 4.18. Die tatsächliche Klasse RealCache

```
1  package de.tnr.sdg.example.cache;
2
3  public class RealCache extends de.tnr.sdg.example.cache.AbstractCache {
4
5     private final int size() {
6        return 0;
7     }
8
9     public RealCache() {
10       super();
11    }
12
13    public void put(java.lang.String filename,int size) {}
14
15    public void reset() {}
16 }
```

AbstractCache Beim Vergleich der automatisch transformierten Klasse `AbstractCache` (Listing 4.19) mit der erwarteten Klasse (Listing 4.12) fallen bis auf den geänderten Paketnamen keine Unterschiede auf. Dennoch fallen beim Vergleich mit der manuell transformierten Klasse (Listing 6.6) zwei Fehler auf. Es werden weder die Klasse, noch die Methoden als `abstract` deklariert, noch werden die Parameter der Methode `put()` (Z. 5) übernommen. Die abstrakten Methoden beider Versionen besitzen jedoch in der Datenbank keinen Kontrollfluss. Auch die Properties `isabstract` und `parameterscount` sind in beiden Fällen identisch. Daher nehmen wir an, dass dieser Fehler durch *Sdg2Java* generiert wird. Somit können wir davon ausgehen, dass die Transformation der Klasse `AbstractCache` korrekt ist.

Listing 4.19. Die tatsächliche Klasse AbstractCache

```
1  package de.tnr.sdg.example.cache;
2
3  public class AbstractCache extends java.lang.Object {
4
5     public void put() {}
6
7     public void reset() {}
8
9     public AbstractCache() {
10       super();
11    }
12 }
```

NullText Die automatisch transformierte Klasse `NullText` (Listing 4.20) weicht abgesehen vom Parameternamen der Funktion `setText(String)` (Z. 5) und dem Paketnamen nicht von der erwarteten Klasse (Listing 4.14) ab. An der Methode `getText()` (Z. 12) erkennen

wir, dass bei der Erstellung des Kontrollflusses der Methoden für Nullklassen der Rückgabewert korrekt erkannt und der Standardwert `null` zurückgegeben wird. Somit können wir annehmen, dass auch diese Klasse korrekt transformiert wird.

Listing 4.20. Die tatsächliche Klasse NullText

```
1  package de.tnr.sdg.example.cache;
2
3  public class NullText extends de.tnr.sdg.example.cache.AbstractText {
4
5     public void setText(java.lang.String arg0) {}
6
7     public NullText() {
8        super();
9     }
10
11    public java.lang.String getText() {
12       return null;
13    }
14 }
```

RealText Die transformierte Klasse `RealText` in Listing 4.21 ist identisch mit der von uns erwarteten Klasse (Listing 4.15). Im Unterschied zu Zeile 5 der manuell transformierten Klasse (Listing 6.8) wird das Feld nach der Transformation im Konstruktor initialisiert (Z. 13). Dies geschieht durch die gleiche Darstellung beider Möglichkeiten in der Datenbank. Da die Initialisierung nach dem `super()`-Aufruf im Konstruktor stattfindet, ist das Ergebnis gleich. Somit funktioniert die Übertragung der Felder einwandfrei und diese Klasse wird korrekt transformiert.

Listing 4.21. Die tatsächliche Klasse RealText

```
1  package de.tnr.sdg.example.cache;
2
3  public class RealText extends de.tnr.sdg.example.cache.AbstractText {
4
5     private String text;
6
7     public void setText(java.lang.String text) {
8        text = text;
9     }
10
11    public RealText() {
12       super();
13       text = "Some text";
14    }
15
16    public java.lang.String getText() {
17       return this.text;
18    }
19 }
```

Tabelle 4.3. Evaluation der Transformation der Kandidatenklassen

Kandidat	Transformierte Klasse	generiert	Korrekte Methoden	Korrekte Felder	Korrekte Konstruktoren
Cache	NullCache	✓	✓	-	✓
	RealCache	✓	✓	-	✓
	AbstractCache	✓	✓	-	✓
Text	NullText	✓	✓	✓	✓
	RealText	✓	✓	✓	✓
	AbstractText	✓	✓	✓	✓

AbstractText Die Klasse `AbstractText` (Listing 4.22) ist abgesehen von den Paketnamen identisch zu der erwarteten Klasse (Listing 4.13). Die fälschlicherweise hinzugefügten Körper der Methoden und die fehlenden Modifizierer und Parameter sind auf die selbe Art entstanden wie bei der Klasse `AbstractCache`. Daher können wir auch bei dieser Klasse von einer korrekten Transformation ausgehen.

Listing 4.22. Die tatsächliche Klasse AbstractText

```
1  package de.tnr.sdg.example.cache;
2
3  public class AbstractText extends java.lang.Object {
4
5      public void setText() {}
6
7      public java.lang.String getText() {}
8
9      public AbstractText() {
10         super();
11     }
12 }
```

Zusammenfassung In Tabelle 4.3 können wir sehen, dass alle von uns geforderten Aspekte der Transformation der Kandidatenklasse korrekt funktionieren. Die Übertragung der Felder an die `Real`-Klasse können wir nur anhand der Transformation der `Text`-Klasse überprüfen, da die Klasse `Cache` kein Feld besitzt. Die von uns festgestellten Abweichungen von den Zielklassen sind entweder für die Funktion der Klassen und Methoden irrelevant oder entstehen durch eine noch fehlerhafte Generierung von *Sdg2Java*. In diesen Fällen stimmt jedoch die Darstellung der SDGs in der Datenbank überein. Daher können wir sagen, dass die Transformation der Kandidatenklassen insgesamt erfolgreich funktioniert.

Transformation der Aufruferklassen

Die transformierte Beispielklasse `MainClass` präsentieren wir in Listing 4.23. In Zeile 6-10 sehen wir, dass der Typ der Felder von Kandidatenklassen erfolgreich in die der abstrakten Version der Kandidaten geändert wird. Im Konstruktor der Klasse (Z. 66) sehen wir, dass die Felder `cache2` und `subtitle` wie erwartet mit den `Real`-Klassen initialisiert werden. Die Felder `cache`, `test` und `title`, die vor der Transformation nicht initialisiert wurden, werden nun korrekt mit der `Null`-Klasse initialisiert. Die `<init>()`-Aufrufe beim Konstruktoraufruf werden von *Sdg2Java* hinzugefügt. Dies geschieht ebenfalls bei der manuell transformierten Klasse (vergleiche Listing 4.16 Z. 72) und ist somit kein Fehler unserer Transformation.

Die Methoden `getAbstractObjectName()` (Z. 16), `checkDifference()` (Z. 24), `postTitle()` (Z. 50), `createText()` (Z. 54) und `createCache()` (Z. 65) sind identisch mit ihren manuell transformierten Gegenstücken in Listing 4.16. Die Änderungen der `return`-Werte der Methode `checkDifference()` wird durch *Sdg2Java* verursacht. Die Werte sind in der Datenbank als `boolean` hinterlegt und somit richtig.

Bei der Methode `postSubtitle()` (Z. 44) wird, anders als von uns bei der Erstellung des Szenarios angenommen, keine Transformation durchgeführt. Dies liegt an der Grundannahme, dass finale Felder keine Kandidaten sein können. Dadurch wird das Feld nicht von der CMQ erkannt und seine Aufrufe nicht für die Transformation vorgemerkt. Semantisch gesehen entsteht dadurch kein Problem, da die `if`-Anweisung bei einem korrekt initialisierten finalen Feld immer erfüllt werden sollte. Allerdings können im Kontext zu *PARROT* Probleme entstehen.

Die `if`-Anweisungen in den restlichen Methoden `postText()` (Z. 32), `cacheFile()` (Z. 40) und `clearCache()` (Z. 61), von denen wir eine Transformation erwartet haben, werden korrekt transformiert und sind identisch mit denen in unserer manuell transformierten Klasse. Zusammengefasst funktioniert die Transformation der Klasse so, wie wir es angenommen haben. Leider bleiben noch einige Abfragen auf Felder übrig, z.B. in der Methode `getAbstractObjectName()` in Zeile 18. Diese übrig gebliebenen Abfragen werden nicht transformiert, da wir die entsprechenden Felder oder Operatoren in den Abfragen in unserer CMQ ausgeschlossen haben. Somit gibt es auch noch keine Transformationsvorschriften für diese Fälle. Allerdings ist es möglich diese mit einer erneuten CMQ und entsprechenden Transformationen umzusetzen.

Listing 4.23. Die tatsächliche Klasse MainClass

```
1  package de.tnr.sdg.example.cache;
2
3  public class MainClass extends java.lang.Object {
4
5     private de.tnr.sdg.example.cache.AbstractObject abstractObject;
6     private final de.tnr.sdg.example.cache.AbstractText subtitle;
7     private de.tnr.sdg.example.cache.AbstractText title;
8     private de.tnr.sdg.example.cache.AbstractText text;
9     private de.tnr.sdg.example.cache.AbstractCache cache2;
10    private de.tnr.sdg.example.cache.AbstractCache cache;
11
```

```
12      public static void main(java.lang.String[] args) {...}
13
14      public java.lang.String getAbstractObjectName() {
15          java.lang.String tmp = null;
16          if (this.abstractObject != null) {
17              tmp = this.abstractObject.getName();
18          }
19          return tmp;
20      }
21
22      public boolean checkDifference() {
23          if (this.cache != this.cache2) {
24              this.cache.reset();
25              return 1;
26          }
27          return 0;
28      }
29
30      public void postText() {
31          java.lang.String testString = "";
32          testString = this.text.getText();
33          this.cache.put(this.text.getText(), 1);
34          this.text.setText(testString);
35          this.createCache();
36      }
37
38      public void cacheFile(java.lang.String filename) {
39          this.cache.put(filename, 0);
40      }
41
42      public void postSubtitle() {
43          if (this.subtitle != null) {
44              System.out.println(this.subtitle.getText());
45          }
46      }
47
48      public void postTitle() {
49          System.out.println(this.title.getText());
50      }
51
52      public void createText() {
53          if (this.text == null) {
54              <init>();
55              text = new de.tnr.sdg.example.cache.RealText;
56          }
57      }
58
59      public void clearCache() {
60          this.cache.reset();
61      }
62
63      public void createCache() {
64          if (this.cache == null) {
65              <init>();
66              cache = new de.tnr.sdg.example.cache.RealCache;
67          }
68      }
69
70      public MainClass() {
71          super();
72          <init>();
73          cache = new de.tnr.sdg.example.cache.NullCache;
```

```
74          <init>();
75          text = new de.tnr.sdg.example.cache.NullText;
76          <init>();
77          title = new de.tnr.sdg.example.cache.NullText;
78          <init>();
79          cache2 = new de.tnr.sdg.example.cache.RealCache;
80          <init>();
81          subtitle = new de.tnr.sdg.example.cache.RealText;
82      }
83
84  }
```

4.2.3. Beeinträchtigungen der Validität

Interne Validität Durch die Abhängigkeit der Transformation von der Erkennung der Kandidaten können wir auch in diesem Schritt nur *Java*-Quellcode bis inklusive Version 7 transformieren. Somit sind für neuere Konstrukte und Funktionen, wie z.B. Lambda-Ausdrücke, noch keine Transformationsvorschriften festgelegt. Auch könnten bei einer Aktualisierung des SDG-Modells von *Java2Neo4J* neue Kanten- und Knotentypen hinzugefügt werden und somit der SDG auf eine andere Weise generiert werden. Daher können wir nicht garantieren, dass die Transformation zukünftig ohne Anpassungen funktioniert.

Ein weiterer Aspekt ist die noch nicht vollständig implementierte Übersetzung des SDGs in Quellcode mit dem Tool *Sdg2Java*, die jedoch eine Voraussetzung für den fehlerfreien Ablauf unseres Ansatzes ist. Durch Überprüfung und genaue Erstellung des SDG haben wir versucht, Fehlerquellen zu vermeiden, können jedoch aufgrund der enormen Größe des SDG keine Fehlerfreiheit garantieren.

Externe Validität Die externe Validität unserer Transformation wird von der recht geringen Zahl von Testfällen eingeschränkt. Wir haben uns bemüht, in den getesteten Klassen möglichst viele Fälle von möglichen Kandidaten und den dazugehörigen Aufrufen abzudecken, können aber durch die enorme Anzahl an möglichen Klassen und Methoden nur eine geringe Zahl davon überprüfen. Durch die oben genannte Limitierung auf Programme der *Java*-Versionen 7 und geringer lässt sich die Transformation nicht auf alle Programme anwenden. Die verhältnismäßige Einfachheit der Darstellung des SDG und der *Java*-API von *Neo4J* ermöglichen jedoch eine leichte Implementierung weiterer Transformationsschritte.

Kapitel 5

Verwandte Arbeiten

Die grundlegende Motivation dieser Arbeit bildet der in Abschnitt 1.2 vorgestellte Ansatz von Wulf [2014], der als Ziel eine halbautomatische Parallelisierung sequentiellen Programmcodes verfolgt. Die Basis der Transformation ist ein durch Informationen aus dynamischer und statischer Analyse angereicherter SDG. In diesem können mit Mustererkennung Kandidaten zur Parallelisierung erkannt und anschließend transformiert werden. Durch die Verwendung der *Neo4J*-Graphdatenbank als Back-End soll eine hohe Performanz bei der Berechnung des Graphen erreicht werden.

In der Arbeit von Krause [2015], die ebenfalls im Kontext von *PARROT* entstanden ist, wird zur Erkennung und Transformation von parallelisierbaren Kandidaten im SDG ein ähnliches Vorgehen wie in unserem Ansatz verfolgt. Dabei werden mit Hilfe einer CMQ und einer Kombination aus Cypher und der *Neo4J*-API drei verschiedene Arten von Kandidaten identifiziert und in eine parallele Form transformiert.

Die Arbeit von Finkes [2016] befasst sich aufbauend auf die Arbeiten von Blümke [2015] und Benekov [2015] mit der Entwicklung eines Grapheditors für Systemabhängigkeitsgraphen. Der Grapheditor kann über das *Neo4J-Reader*-Plugin SDGs aus einer *Neo4J*-Datenbank importieren und mit dem *Neo4J-Writer*-Plugin wieder exportieren. Die importierten Graphen können vielfältiger als im *Neo4J*-Browser visualisiert werden. Dies soll die Analyse des SDG und die Identifizierung von parallelisierbaren Klassen vereinfachen.

Der Ansatz von Koppenhagen [2016] greift auf den von Blümke [2015] erstellten Grapheditor zurück, um aus bestimmten Strukturen in einem SDG automatisch Cypher-Abfragen zu generieren. Dafür ergänzt er den Grapheditor um weitere Funktionen und eine Implementierung des *User-Interaction-Protocols*.

Eine automatische Identifizierung so genannter Cache-Klassen in einem SDG wird in der Arbeit von Reckling [2017] vorgestellt. Dies sind Klassen, die zum Speichern von Daten benutzt werden. Stehen sie in Abhängigkeit zu mehreren Ausführungssträngen eines Programms, so kann dies die Parallelisierung erschweren. Für die Identifizierung der Cache-Klassen greift Reckling dabei ebenfalls auf Cypher-Abfragen zurück.

Eine andere Arbeit, die sich mit der Identifizierung und Transformation von Subgraphen auseinandersetzt, stammt von Sun u. a. [2010]. Sie stellen mit dem Tool *PatternBuild* eine Möglichkeit zur Erstellung von Fehlermustern vor. Diese werden durch eine schnelle Subgraph-Erkennung mit dem von ihnen entwickelten Algorithmus *GADDI* im SDG eines Programms gefunden. Bei der Behebung des Fehler wird automatisch ein Lösungsmuster erstellt. Zukünftig soll dieses halbautomatisch auf andere Instanzen des selben Fehler

angewendet werden können.

Wang u. a. [2011] betrachten in ihrer Arbeit einen Ansatz zur Suche von Codeausschnitten im Quellcode von Programmen. Bisherige Ansätze durch Betriebssysteme oder IDEs (z.B. *Eclipse*) stellen nur ineffiziente Funktionen zur Suche nach genau übereinstimmenden Ausschnitten oder regulären Ausdrücken zur Verfügung. Der Ansatz kombiniert die Vorteile natürlicher Sprachen und SDGs zur Identifizierung von semantisch übereinstimmenden Ausschnitten. Eine automatische Transformation dieser Codestellen im SDG ist allerdings noch nicht vorgesehen.

Mit der Adaption von SDGs für funktionale Programmiersprachen beschäftigen sich [Silva u. a. 2012] in ihrer Arbeit. Dafür wenden sie *Program Slicing* auf *Erlang*-Programme an, um Codeausschnitte zu generieren, die in einem SDG dargestellt werden können. Durch diese Arbeit könnte das Konzept der Identifizierung und Transformation von bestimmten Kandidaten auf andere Programmierparadigmen übertragen werden.

Kapitel 6

Fazit und Ausblick

In dieser Arbeit zeigen wir die Durchführbarkeit der automatischen Transformation von Programmen nach dem Nullobjekt-Muster. Wir arbeiten mit dem SDG von Programmen, der in einer *Neo4J*-Graphdatenbank gespeichert wird. Diese ermöglicht uns die Benutzung der Abfragesprache Cypher. Aus dem durch vorhergehende Arbeiten mit Informationen angereicherten SDG können wir die relevanten Kandidaten durch Mustererkennung identifizieren. Irrelevante Kandidaten wie finale Felder können wir gezielt ausschließen.

Der ursprüngliche Plan war es, eine einzige CMQ und CUQ für den gesamten Prozess zu entwickeln. Allerdings stellte sich heraus, dass sich die Cypher-Abfragen zwar dafür eignen, bestimmte Muster einfach zu erkennen, jedoch bei der Sammlung von wichtigen Informationen für die Transformation schnell zu umfangreich werden. Dies liegt in der Vielzahl unterschiedlich aufgebauter Klassen und Methoden, die berücksichtigt werden müssen. Eine CUQ, die alle Möglichkeiten abdeckt, wäre sehr komplex und unübersichtlich. Daher haben wir uns entschieden, die Erkennung von Kandidaten mit einer einzelnen CMQ durchzuführen und für die Transformation auf die eine Kombination aus Cypher und der *Java*-API von *Neo4J* zurückzugreifen. Dies ermöglicht es uns, Nutzen aus der Wiederverwendbarkeit von *Java*-Quellcode und der Mächtigkeit von Cypher zu ziehen.

Die implementierte Erkennung und Transformation funktionieren wie erwartet. Allerdings fallen bei der Evaluation Situationen auf, die wir im Vorfeld nicht bedacht haben. Wir schließen bei der Erkennung abstrakte Klassen und Interfaces von der Transformation aus, für die wir noch keine Transformationsvorschriften erstellen können. Durch eine Weiterentwicklung des Graphmodells von *Java2Neo4J* könnte es zukünftig möglich sein, entsprechende Transformationen durchzuführen. Weiterhin ist die bisher implementierte Transformation noch nicht für alle Klassen sinnvoll. So kann es vorkommen, dass nach der Entfernung von `if`-Anweisungen automatisch `Exceptions` geworfen werden (vgl. Listing 4.6, Z. 9). Auch ist die korrekte Funktion von Abfragen, ob ein Feld gleich `null` ist (vgl. Listing 4.2, Z. 15), durch die Transformation der Kandidatenklassen nicht mehr garantiert, da diese Abfrage immer `false` ergibt. Schlussendlich bleibt noch zu sagen, dass die Transformation von Feldern, deren Datentyp einer Java-Klasse entspricht, nicht immer sinnvoll erscheint. Die genannten Fälle müssen im Kontext der Parallelisierung durch *PARROT* erneut betrachtet und entsprechend zur Transformation hinzugefügt werden.

Literaturverzeichnis

[Benekov 2015] Y. Benekov. Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung und Verwaltung von Graphen. Bachelorarbeit. 2015. (Siehe Seite 49)

[Blümke 2015] L. E. Blümke. Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung von graphbasierten Quellcodemustern. Bachelorarbeit. 2015. (Siehe Seite 49)

[Ferrante u. a. 1987] J. Ferrante, K. J. Ottenstein und J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9.3 (Juli 1987), Seiten 319–349. URL: http://doi.acm.org/10.1145/24039.24041. (Siehe Seite 5)

[Finkes 2016] D. J. Finkes. A Hierarchical Eclipse-based Editor for System Dependency Graphs. Masterarbeit. 2016. (Siehe Seiten 6, 8, 10 und 49)

[Fowler 2000] M. Fowler. Refactoring oder: Wie Sie das Design vorhandener Software verbessern. Professionelle Softwareentwicklung. pp. 264-272. Addison-Wesley, 2000. URL: https://books.google.de/books?id=UUpIAAAACAAJ. (Siehe Seite 11)

[Golumbic 2004] M. Golumbic. Algorithmic Graph Theory and Perfect Graphs. Second Edition. Band 57. Elsevier, 2004. (Siehe Seite 5)

[Horwitz u. a. 1988] S. Horwitz, T. Reps und D. Binkley. Interprocedural Slicing Using Dependence Graphs. *SIGPLAN Not.* 23.7 (Juni 1988), Seiten 35–46. URL: http://doi.acm.org/10.1145/960116.53994. (Siehe Seite 5)

[Koppenhagen 2016] E. Koppenhagen. GUI-based Automated Generation of Neo4j Cypher Queries for Candidate Patterns and Parallelization Patterns. Masterarbeit. 2016. (Siehe Seiten 8, 9 und 49)

[Krause 2015] J. E. Krause. Approach to Parallelise Software Systems using a System Dependency Graph. Masterarbeit. 2015. (Siehe Seiten 18, 19 und 49)

[Neo Technology 2015a] Neo Technology. The Neo4J Developer Manual. http://neo4j.com/docs/developer-manual/current/cypher/. Zugriff 26-05-2017. 2015. (Siehe Seite 10)

[Neo Technology 2015b] Neo Technology. The Neo4J Manual. http://neo4j.com/docs/pdf/neo4j-manual-2.2.5.pdf. Zugriff 17-05-2017. 2015. (Siehe Seiten 8 und 10)

[Reckling 2017] D. Reckling. Automatic Detection of Cache Classes using PARROT. Seminararbeit. 2017. (Siehe Seite 49)

[Rodriguez und Neubauer 2010] M. A. Rodriguez und P. Neubauer. Constructions from Dots and Lines. *CoRR* abs/1006.2361 (2010). URL: http://arxiv.org/abs/1006.2361. (Siehe Seiten 5, 6)

[Silva u. a. 2012] J. Silva, S. Tamarit und C. Tomás. System Dependence Graphs in Sequential Erlang. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*. FASE'12. Tallinn, Estonia: Springer-Verlag, 2012, Seiten 486–500. URL: http://dx.doi.org/10.1007/978-3-642-28872-2_33. (Siehe Seite 50)

[Sullivan 2015] D. Sullivan. NoSQL for Mere Mortals. Addison-Wesley Professional, 2015. (Siehe Seite 8)

[Sun u. a. 2010] B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhand und J. Yang. Propagating Bug Fixes with Fast Subgraph Matching. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. San Jose, CA, USA: IEEE, 2010. (Siehe Seite 49)

[Walkinshaw u. a. 2003] N. Walkinshaw, M. Roper und M. Wood. The Java system dependence graph. In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 2003, Seiten 55–64. (Siehe Seiten 6 und 34)

[Wang u. a. 2011] S. Wang, D. Lo und L. Jiang. Code search via topic-enriched dependence graph matching. In: *18th Working Conference on Reverse Engineering (WCRE 2011): Limerick, Ireland, 17-20 October: Proceedings*. Los Alamitos, CA, 2011, Seiten 119–123. (Siehe Seite 50)

[Wulf 2014] C. Wulf. Pattern-based Detection and Utilization of Potential Parallelism in Software Systems. *Software Enginnering 2014* (2014). (Siehe Seiten 1, 2, 6 und 49)

Anhang

Listing 6.1. Ausschnitt der Ant-Klasse UpToDate.java

```
1  package org.apache.tools.ant.taskdefs;
2  ...
3  public class UpToDate extends Task implements Condition {
4  ...
5     protected Mapper mapperElement = null;
6  ...
7     public Mapper createMapper() throws BuildException {
8           if (mapperElement != null) {
9               throw new BuildException("Cannot␣define␣more␣than␣one␣mapper",
10                                       getLocation());
11          }
12          mapperElement = new Mapper(getProject());
13          return mapperElement;
14    }
15 ...
16 }
```

Listing 6.2. Ausschnitt der Ant-Klasse PresentSelector.java

```
1  package org.apache.tools.ant.types.selectors;
2  ...
3  public class PresentSelector extends BaseSelector {
4  ...
5     private Mapper mapperElement = null;
6  ...
7     public String toString() {
8           final StringBuilder buf = new StringBuilder("{presentselector␣targetdir:␣");
9           if (targetdir == null) {
10              buf.append("NOT␣YET␣SET");
11          } else {
12              buf.append(targetdir.getName());
13          }
14          buf.append("␣present:␣");
15          if (destmustexist) {
16              buf.append("both");
17          } else {
18              buf.append("srconly");
19          }
20          if (map != null) {
21              buf.append(map.toString());
22          } else if (mapperElement != null) {
23              buf.append(mapperElement.toString());
24          }
25          buf.append("}");
26          return buf.toString();
27    }
28 ...
29    public Mapper createMapper() throws BuildException {
```

```
30          if (map != null || mapperElement != null) {
31              throw new BuildException("Cannot␣define␣more␣than␣one␣mapper");
32          }
33          mapperElement = new Mapper(getProject());
34          return mapperElement;
35      }
36
37      public void addConfigured(final FileNameMapper fileNameMapper) {
38          if (map != null || mapperElement != null) {
39              throw new BuildException("Cannot␣define␣more␣than␣one␣mapper");
40          }
41          this.map = fileNameMapper;
42      }
43      ...
44  }
```

Listing 6.3. Ausschnitt der Ant-Klasse ConcatFileInputStream.java

```
1   package org.apache.tools.ant.types.selectors;
2   ...
3   public class MappingSelector extends BaseSelector {
4   ...
5       protected Mapper mapperElement = null;
6   ...
7       public Mapper createMapper() throws BuildException {
8           if (map != null || mapperElement != null) {
9               throw new BuildException("Cannot␣define␣more␣than␣one␣mapper");
10          }
11          mapperElement = new Mapper(getProject());
12          return mapperElement;
13      }
14
15      public void addConfigured(FileNameMapper fileNameMapper) {
16          if (map != null || mapperElement != null) {
17              throw new BuildException("Cannot␣define␣more␣than␣one␣mapper");
18          }
19          this.map = fileNameMapper;
20      }
21      ...
22  }
```

Listing 6.4. Die manuell transformierte Klasse NullCache

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class NullCache extends
       AbstractCache {
4
5     public void put(String filename, int
          size) {}
6
7     public void reset() {}
8
9  }
```

Listing 6.5. Die manuell transformierte Klasse RealCache

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class RealCache extends
       AbstractCache {
4
5     public void put(String filename, int
          size) {
6        size();
7     }
8
9     public void reset() {
10       size();
11    }
12
13    private int size() {
14       return 0;
15    }
16
17 }
```

Listing 6.6. Die manuell transformierte Klasse AbstractCache

```
1  package de.tnr.sdg.example.transformedCache;
2
3  public abstract class AbstractCache {
4
5     public abstract void put(String filename, int size);
6     public abstract void reset();
7  }
```

Listing 6.7. Die manuell transformierte Klasse NullText

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class NullText extends AbstractText
       {
4
5     public String getText() {
6        return null;
7     }
8
9     public void setText(String text) {}
10 }
```

Listing 6.8. Die manuell transformierte Klasse RealText

```
1  package de.tnr.sdg.example.
       transformedCache;
2
3  public class RealText extends AbstractText
       {
4
5     private String text = "Some text";
6
7     public String getText() {
8        return text;
9     }
10
11    public void setText(String text) {
12       this.text = text;
13    }
14 }
```

Listing 6.9. Die manuell transformierte Klasse AbstractText

```java
package de.tnr.sdg.example.transformedCache;

public abstract class AbstractText {

   public abstract String getText();
   public abstract void setText(String text);
}
```

Listing 6.10. Die manuell transformierte Klasse MainClass

```java
package de.tnr.sdg.example.transformedCache;

public class MainClass {

   private AbstractCache cache = new NullCache();
   private AbstractCache cache2 = new RealCache();
   private AbstractText text = new NullText();
   private AbstractText title = new NullText();
   private final AbstractText subtitle;
   private AbstractObject abstractObject;

   public MainClass(){
      subtitle = new RealText();
   }

   public void createCache(){
      if (cache == null) {
         this.cache = new RealCache();
      }
   }

   public void clearCache(){
      cache.reset();
   }

   public void createText(){
      if (text == null) {
         text = new RealText();
      }
   }

   public void postTitle(){
      System.out.println(title.getText());
   }

   public void postSubtitle(){
      System.out.println(subtitle.getText());
   }

   public void cacheFile(String filename){
      cache.put(filename, 0);
   }

   public void postText(){
      String testString = "";
      testString = text.getText();

      cache.put(text.getText(), 1);
      text.setText(testString);
      createCache();
   }
```

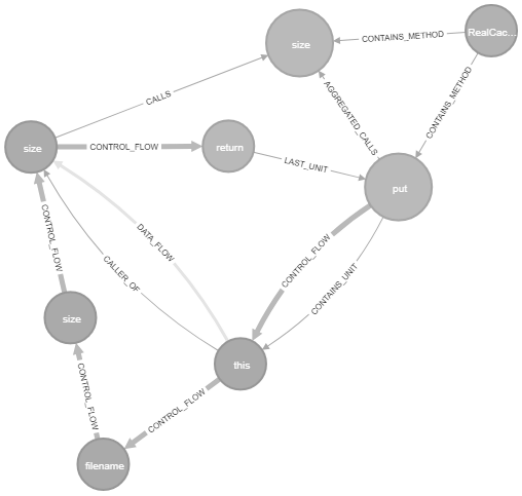

Abbildung 6.1. Kontrollfluss der automatisch transformierten Methode

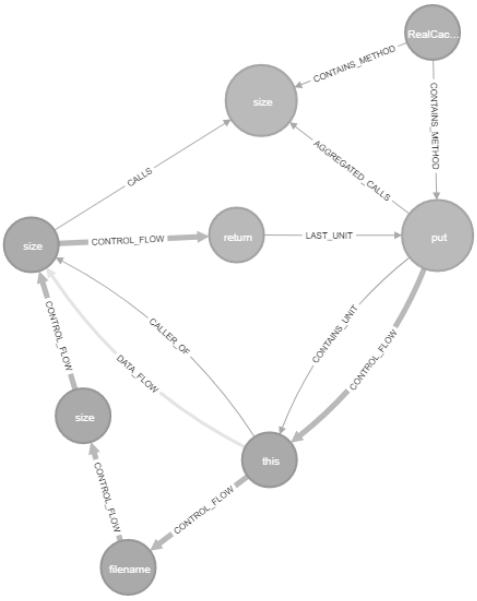

Abbildung 6.2. Kontrollfluss der manuell transformierten Methode

```
52
53      public boolean checkDifference() {
54          if (cache!=cache2) {
55              cache.reset();
56              return true;
57          } else {
58              return false;
59          }
60      }
61
62      public String getAbstractObjectName(){
63          String tmp = null;
64          if (abstractObject != null){
65              tmp = abstractObject.getName();
66          }
67          return tmp;
68      }
69
70      public static void main(String[] args) {...}
71  }
```