Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Manipulation der Typisierung serialisierter SKilL-Graphen

Oliver Brösamle

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Dr. Timm Felden
Beginn am:	10. April 2018
Beendet am:	10. Oktober 2018

Kurzfassung

Das Ziel der Masterarbeit ist es, ein Werkzeug zur Manipulation der Typen und Objekte von serialisierten SKilL-Graphen zu entwickeln. Dafür bietet das Werkzeug verschiedene Möglichkeiten der Manipulation. Zunächst wird eine Methode bereitgestellt, welche effizient alle Objekte löscht, die von einer Menge von Wurzelobjekten nicht erreichbar sind. Ebenso können einzelne Felder aus Typen sowie ganze Typen aus dem Graphen entfernt werden. Darüber hinaus ist es möglich, den Graphen auf eine neue Spezifikation zu projizieren. Hierbei wird sichergestellt, dass eine Projektion des Graphen auf die gegebene Spezifikation durchführbar ist. Zusätzlich ist eine Änderung und Prüfung der Restrictions während bzw. nach der Projektion möglich. Ferner konnte die Normalisierung durch eine Ordnungsfunktion auf eine abstrakte Weise umgesetzt werden. Jede der Methoden liest zunächst eine Binärdatei ein, manipuliert dann den Graph und schreibt die Daten anschließend wieder in eine Binärdatei. Durch geeignete Tests wird die Funktionsfähigkeit der Methoden gezeigt.

Inhaltsverzeichnis

1. Einleitung

SKilL ist ein Serialisierungskonzept, welches für den Einsatz im Bereich der Programmanalysen entworfen wurde. Das Konzept ermöglicht die Nutzung verschiedener Programmiersprachen, um in den einzelnen Phasen der Analyse bereits vorhandene Bibliotheken verwenden zu können. Hierzu wurde ein plattform- und sprachunabhängiges Binärformat entworfen. Dieses bietet die Möglichkeit Instanzen objektorientierter Typen in einer Austauschdatei zu speichern. Der Fokus des Serialisierungsformats liegt dabei auf der IO-Performance, da im Rahmen der Softwareanalysen große Datenmengen anfallen können.

Der Anwender soll nicht mit den Details des Binärformats belastet werden. Aus diesem Grund existiert ein Code-Generator, welcher ein anwendungsspezifisches API erzeugt. Die Beschreibung des Typsystems erfolgt dabei durch eine speziell für SKilL entwickelte Spezifikationssprache. Aus Sicht des Anwenders ergibt sich die in Abbildung 1.1 dargestellte Vorgehensweise. Nach Ausführung des Anwendercodes auf der Eingabedatei wird eine neue Binärdatei geschrieben. Somit wird eine Verkettung von mehreren Analysephasen ermöglicht [Fel17a] §5, §6.2.

1.1. Motivation

Die gespeicherten Instanzen objektorientierter Typen können Referenzen aufeinander haben. Aus diesem Grund lassen sich die Binärdateien als SKilL-Graphen interpretieren in welchen die Objekte als Knoten und die Referenzen als Kanten angesehen werden. Mit der vorgestellten Anbindung des Anwendercodes durch ein generiertes API lassen sich Knoten entfernen sowie Kanten verändern oder löschen. Diese Operationen unterliegen den Einschränkungen des Typsystems. Das Typsystem selbst lässt sich durch die normale Anwendung des APIs nicht verändern.

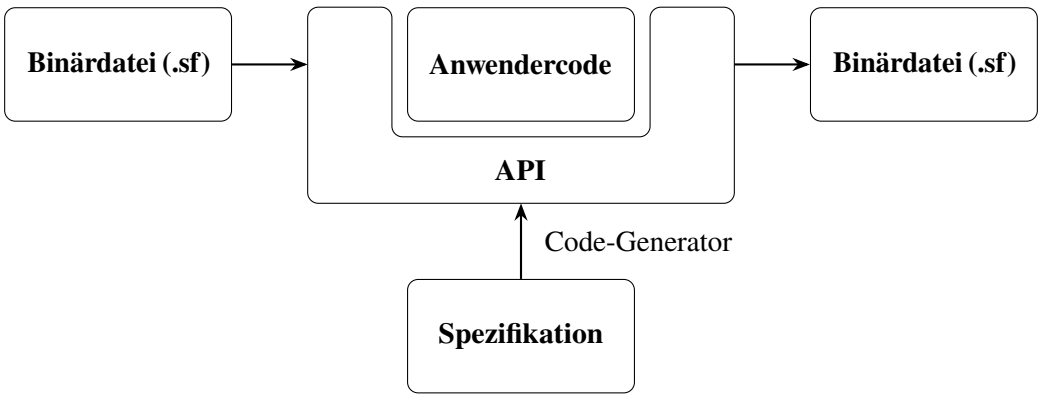

Abbildung 1.1.: Verwendung von SKilL aus Anwendersicht

Ziel dieser Arbeit ist die Entwicklung eines Werkzeugs, welches die Typen und Objekte von serialisierten SKilL-Graphen manipulieren kann. Somit kann in einer Werkzeugkette kurzfristig die Typisierung verändert werden, um bestimmte Bedürfnisse der folgenden Werkzeuge abzudecken. Ferner ermöglicht das Werkzeug eine Migration alter Datensätze auf eine neue Spezifikation. Unter Anderem sind folgende Operationen zur Veränderung des Typsystems und der Objekte möglich:

- **Typen hinzufügen und entfernen:** Typen können an beliebiger Stelle in der Typhierarchie hinzugefügt werden, ohne die Objekte des SKilL-Graphen zu verändern. Zudem ist die Entfernung von Typen möglich. Dabei können die zugehörigen Objekte gelöscht oder auf den Supertyp projiziert werden.
- **Typhierarchie verändern:** Das Werkzeug bietet die Möglichkeit die Typhierarchie zu ändern. Dabei wird überprüft, ob alle Daten in der neuen Hierarchie darstellbar sind.
- **Felder hinzufügen und entfernen:** Das Hinzufügen und Entfernen von Feldern ist ebenfalls möglich. Somit können die Eigenschaften der Typen neu definiert werden.
- **Feldtypen verändern:** Ferner ist es möglich den Feldtyp jedes Felds zu ändern. Dabei sind sowohl Up- als auch Downcasts in der jeweiligen Hierarchie durchführbar. Im Falle eines Downcasts ist eine Prüfung aller Werte inbegriffen.
- **Felder verschieben:** Jedes Feld kann in der Hierarchie nach oben oder unten verschoben werden. Dabei wird überprüft, ob alle Daten durch die verschobenen Felder speicherbar sind.
- **Restrictions verändern:** Das Werkzeug ermöglicht das Entfernen, Hinzufügen oder Ändern von Restrictions.
- **Bereinigung von Typen und Objekten:** Durch einige der genannten Operationen kann es zu toten Objekten und Typen kommen. Das Werkzeug bietet hierfür eine Bereinigung (Garbage Collection) an. Diese entfernt alle nicht mehr benötigten Objekte und Typen.

1.2. Teilaufgaben

Die Aufgabenstellung dieser Arbeit ist in mehrere Teilaufgaben gegliedert. Diese haben nahezu keine Abhängigkeiten untereinander und können deshalb separat bearbeitet werden.[1] Aus didaktischen Gründen werden die Aufgaben in Reihenfolge der folgenden Auflistung bearbeitet:

- **Garbage Collection:** Ziel der Aufgabe ist die Entwicklung einer Transformationsoperation zur Entfernung aller Objekte, welche von einer Menge von Wurzelobjekten nicht erreichbar sind. Die Definition der Wurzelobjekte soll dabei sowohl alle Instanzen eines Typs als auch einzelne Objekte umfassen können. Aufgrund der potentiell großen Datenmengen muss die entwickelte Methode effizient ablaufen.
- **Felder und Typen entfernen:** Aus dem Typsystem des SKilL-Graphen soll das Entfernen einzelner Felder aus Typen sowie ganzer Typen möglich sein. Die entwickelte Entfernungsoperation muss ebenfalls effizient sein.

[1] Die Änderung der Restrictions bilden eine Ausnahme. Für diese muss die Projektion auf eine neue Spezifikation bereits bearbeitet sein.

- **Spezifikationsprojektion:** Diese Aufgabe beinhaltet die Entwicklung einer Operation zur Projektion des eingelesenen SKilL-Graphen auf eine neue Spezifikation. Dabei muss sichergestellt werden, dass die Werte des Graphen zur Zielspezifikation passen.
- **Prüfen und Ändern von Restrictions:** Dieser Teil der Arbeit beschäftigt sich mit der Prüfung von Restrictions. Zudem sollen die Restrictions für einzelne Felder oder Typen verändert werden können.
- **Normalisierung mit Ordnungsfunktion:** Als letzte Teilaufgabe soll eine Transformationsoperation zur Normalisierung des Graphen mithilfe einer Ordnungsfunktion entwickelt werden.

Die Bearbeitung dieser Aufgaben ist in Kapitel 3 bis 7 beschrieben. Nach der Verarbeitung des SKilL-Graphen durch die Transformationsoperationen muss eine Binärdatei geschrieben werden können. Zudem muss die Funktionsfähigkeit der Operationen durch geeignete Tests sichergestellt werden. Diese Evaluation der Teilaufgaben findet in Kapitel 8 statt.

Für die Anwendung aller Methoden mit Ausnahme der Normalisierung mit einer Ordnungsfunktion wurde ein Command Line Interface (CLI) entwickelt. Dieses basiert auf der Apache Commons CLI Bibliothek [Apa17]. Die Umsetzung des CLI ist keine Aufgabe dieser Arbeit und wird deshalb nicht extra ausgeführt. Der Hilfetext zur Anwendung ist in Anhang B zu sehen.

2. Einführung in SKilL

In diesem Abschnitt werden die notwendigen Grundlagen des SKilL-Serialisierungskonzepts vermittelt. Das SKilL-Binärformat enthält das verwendete Typsystem sowie alle Instanzen der Typen. Eine detaillierte Beschreibung des Formats befindet sich in [Fel17b]. Ein genaues Verständnis des Aufbaus der Binärdatei ist für diese Arbeit nicht notwendig, da die Transformationsoperationen auf ein bestehendes Java-API aufgebaut werden.

2.1. Spezifikationssprache

Die Spezifikationssprache ermöglicht eine Deklaration der Typen und Felder, welche im Binärformat gespeichert werden können. Spezifikationsdateien tragen per Konvention die Endung `.skill`. Dieser Abschnitt vermittelt einen Überblick über die grundlegenden Bestandteile der Sprache, welche für diese Arbeit wichtig sind. Eine vollständige Beschreibung ist in [Fel17b] zu finden.

2.1.1. Typen und Vererbung

Listing 2.1 zeigt die Deklaration eines einfachen Typs. Zudem wird ein Feld in diesem Typ deklariert. Das Feld hat den Typ `i32`, welcher einem normalen Integer in Java entspricht.

Im folgenden Listing 2.2 wird das Beispiel um Vererbung erweitert. Der Doppelpunkt steht hierbei vor dem erweiterten Supertyp. Die Reihenfolge der Deklarationen spielt für die Vererbung keine Rolle. Abbildung 2.1 zeigt die entstehende Vererbungshierarchie der Spezifikation. Hier bildet `A` die Wurzel der Hierarchie. Der Wurzelknoten nimmt in der Implementierung eine spezielle Rolle ein und wird als Basistyp bezeichnet.

2.1.2. Felder

Felder spielen eine entscheidende Rolle für die Spezifikation, da sie angeben welche Daten in einem Typ gespeichert werden können. Dabei lassen sich Felder in drei Kategorien einteilen, welche im folgenden Abschnitt näher beschrieben werden. Eine Spezifikation mit Beispielen zu allen strukturell möglichen Feldtypen, die im Binärformat vorhanden sind, ist in Anhang A dargestellt.

Listing 2.1 Einfacher Typ C mit Integerfeld f

```
1 // einfache Typdeklaration mit Feld
2 C {
3         i32 f;
4 }
```

Listing 2.2 Einfaches Beispiel einer Vererbungshierarchie

```
1  // Typ C erbt von Typ A
2  C : A {
3          i32 f;
4  }
5
6  // Deklaration von Typ A und weiteren Subtypen
7  A {}
8  B : A {}
9  D : B {}
```

Grunddatentypen In diese Kategorie fallen eine Reihe vordefinierter Typen. Diese bilden Daten ab, die es in nahezu jeder Programmiersprache gibt. Dazu gehören zunächst Ganzzahlen in verschiedenen Längen, welche den Java Typen `byte`, `short`, `int` und `long` entsprechen. Gleitkommazahlen lassen sich in 32 und 64 Bit darstellen (`float` und `double`). Zudem bietet SKilL die Möglichkeit Wahrheitswerte und Zeichenketten (Strings) als Datentyp zu verwenden.

Referenzen Mit diesem Datentyp lassen sich Referenzen der Instanzen aufeinander realisieren. Jede Referenz hat einen festgelegten Typ (starke Typisierung), kann aber auf ein Objekt von Subtypen des festgelegten Typs verweisen (Polymorphie). Zudem gibt es die Möglichkeit auf `null` zu referenzieren.

Container Unter Containern ist alles zu verstehen, das mehr als eine Instanz oder Referenz eines Datentyps aufnehmen kann. SKilL bietet in dieser Kategorie Arrays, Listen, Sets und mehrstellige Maps. Für jeden Container müssen die beinhalteten Datentypen in der Spezifikation angegeben werden. Diese Typen gehören immer zu den Grunddatentypen oder Referenzen. Eine Schachtelung von Containern ist demnach nicht möglich.

2.1.3. Restrictions

Restrictions stellen eine Erweiterung des bisher vorgestellten Typsystems aus Typen und Feldern dar. Sie bieten einen Weg, die Werte eines Feldes oder ganze Instanzen eines Typs einzuschränken. Die Angabe von Restrictions erfolgt in der Spezifikation direkt über dem Typ oder dem Feld auf welchem sie angewandt werden. Notiert werden sie dabei mit einem @, dem Namen der Restriction und einem oder mehreren möglichen Argumenten.

Felden [Fel17b] §5.1 enthält alle verfügbaren Typ- und Feldrestrictions. Zudem werden die nötigen Argumente und eine Beschreibung der Auswirkung angegeben. In der bisherigen Java-Implementierung des API sind lediglich die Feld-Restrictions `@range` und `@nonnull` enthalten. Im Zuge dieser Arbeit wurden die fehlenden Restrictions implementiert, um die Aufgabe der Änderung und Prüfung der Restrictions bearbeiten zu können. Ferner wurde die Serialisierung um die neu umgesetzten Restrictions erweitert.

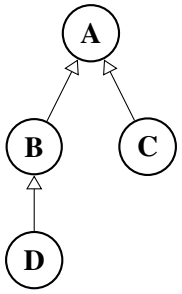

Abbildung 2.1.: Vererbungshierarchie aus Listing 2.2

2.2. Typordnung

Für die Serialisierung der Typen wird in SKilL eine Typordnung eingeführt [Fel17b] §7.2.3. Diese bringt die definierte Typhierarchie in eine sortierte Abfolge von Typen. Die Sortierung erfolgt hierbei topologisch, das heißt Typen kommen vor ihren Subtypen. Somit ergibt sich eine partielle Ordnung. Die Typordnung kann zu einer totalen Ordnung erweitert werden, indem zusätzlich eine lexikalische Ordnung eingesetzt wird.

Die Beispielhierarchie in Abbildung 2.1 hat für die topologische Sortierung zwei mögliche Resultate:

1. A B D C
2. A C B D

Mehrere Resultate können nur auftreten, wenn ein Typ mehr als einen Subtypen besitzt. In diesen Fällen kann die lexikalische Ordnung angewendet werden, um eine totale Ordnung zu erhalten. Im Beispielfall führt dies zu Resultat 1.

2.3. SKilL-Graph

Die Instanzen aller Typen und ihre Referenzen aufeinander können als gerichteter Multigraph aufgefasst werden. Dabei wird jede Instanz als ein Knoten angesehen und jede Referenz bildet eine gerichtete Kante zum Referenzwert hin. Jede Instanz wird dabei eindeutig durch ihren Typ und ihre SKilL-ID identifiziert.

SKilL-IDs sind positive, natürliche Zahlen. Der Nullzeiger wird durch die Zahl null kodiert. Deshalb starten die IDs bei eins. Die Vergabe der IDs erfolgt durch eine laufende Nummerierung vom Basistyp aus. Die Reihenfolge der Typen entspricht dabei der Typordnung. Angenommen jeder Typ aus der Vererbungshierarchie in Abbildung 2.1 hat zwei Instanzen, dann ergeben sich folgende Tupel aus Typ und SKilL-ID[1]:

$$a\#1\ a\#2\ b\#3\ b\#4\ d\#5\ d\#6\ c\#7\ d\#8$$

[1] Die Tupel werden hier durch ein Hash-Symbol getrennt.

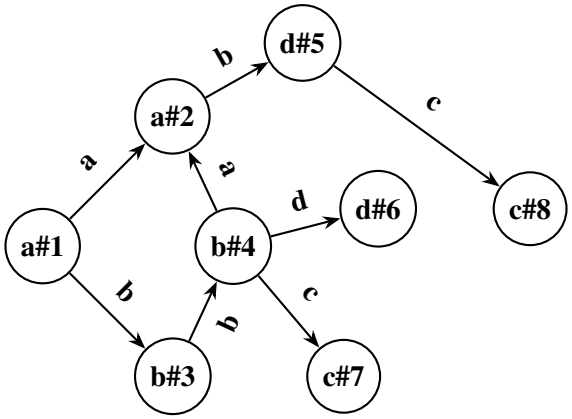

Abbildung 2.2.: Beispiel für einen SKilL-Graphen. Knoten sind mit Typ und SKilL-ID beschriftet. Kanten tragen die Namen der entsprechenden Felder.

Durch Referenzen dieser Objekte aufeinander entsteht ein SKilL-Graph. Abbildung 2.2 zeigt ein Beispiel für einen SKilL-Graphen, welcher aus den genannten Objekten besteht. Die Spezifikation der Typen ist für dieses Beispiel eine andere als bisher verwendet. In dieser finden sich nun Referenzfelder der Typen aufeinander. Der Einfachheit halber wird die Spezifikation für den Graphen hier nicht angegeben.

2.4. API

Das API bietet eine Zugriffsmöglichkeit auf das Typsystem und die Objekte einer Binärdatei. Nach dem Öffnen und Einlesen der Datei ist es möglich über alle Typen und Felder zu iterieren. Zudem ist der Zugriff auf die Objekte über ihre Typen realisierbar. Die Feldwerte jedes Objekts können gelesen und geschrieben werden. Des Weiteren ist das Entfernen von Objekten möglich. Zusammengefasst bietet das API eine Möglichkeit die Objekte des Typsystems zu modifizieren oder zu entfernen. Über das API ist jedoch keine Änderung am Typsystem möglich.

Da die Aufgabenstellung der Arbeit eine Modifikation des Typsystems vorsieht, ist es unverzichtbar auf die internen Datenstrukturen des API zuzugreifen. Zudem sind einige Aktionen über die internen Darstellungen effizienter realisierbar. Um mit den Datenstrukturen arbeiten zu können, wurden einige Anpassungen vorgenommen. Zum größten Teil sind diese dazu da, den Zugriff auf die internen Strukturen zu erlauben.

Abbildung 2.3 zeigt einen groben Überblick über die verwendeten Strukturen. Die Darstellung ist stark vereinfacht und die Namen sind zum besseren Verständnis teilweise verändert. Der *SkillState* repräsentiert den internen Zustand der Binärdatei. Über diesen ist das Einlesen und die Ausgabe der Datei möglich.

Type bildet das zentrale Element in der Struktur. Von diesem sind sowohl Felder als auch Objekte erreichbar. Gleichzeitig ist *Type* auch ein *FieldType*, da Referenzen in Feldern gespeichert werden können. Stellvertretend für die Grund- und Containertypen sind hier die beiden Gruppierungen angegeben.

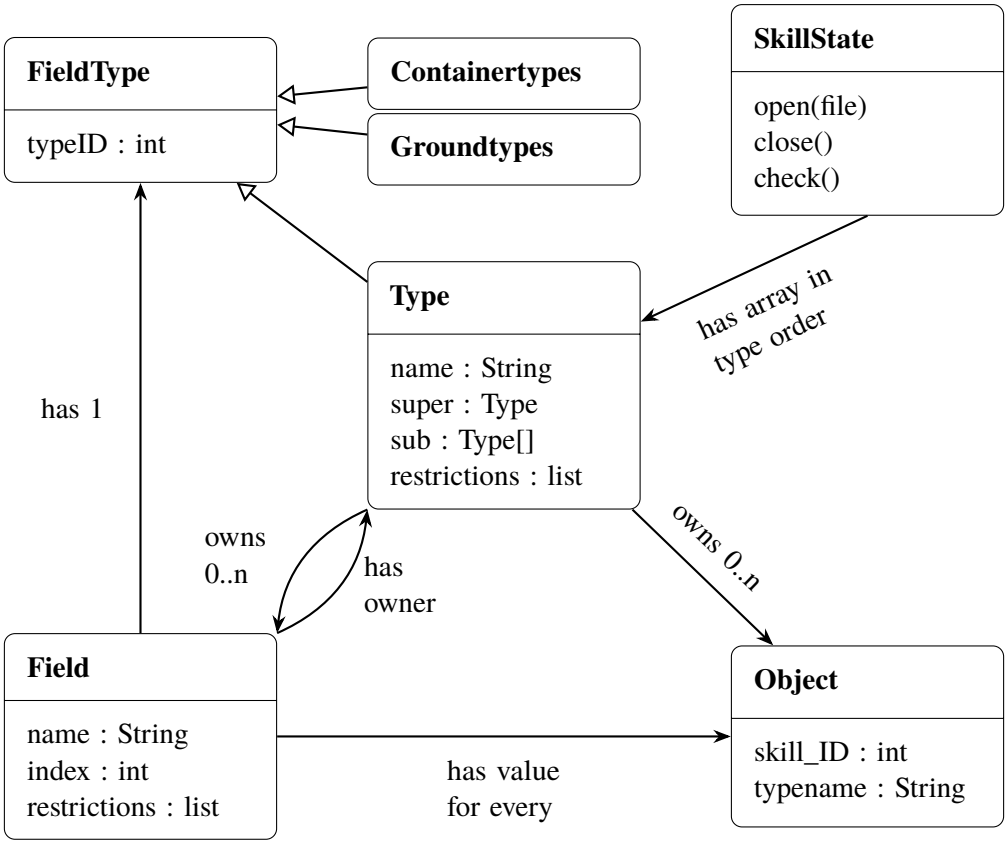

Abbildung 2.3.: Stark vereinfachte Darstellung der internen API-Datenstruktur

3. Garbage Collection

Als Garbage Collection wird die automatische Wiederverwertung von Speicher bezeichnet [Coh81]. Die Aufgabe des Garbage Collectors ist es, nicht mehr verwendete Objekte auf dem Heap zu finden und deren Speicherplatz für die Wiederverwendung freizugeben. Garbage (deutsch: Müll) bezeichnet dabei alle Objekte, die nicht mehr durch das laufende Programm erreichbar sind. Die Bezeichnung „nicht erreichbar“ bedeutet in diesem Zusammenhang, dass ein Objekt nicht durch Traversierung einer Kette von Zeigern vom Programm erreicht werden kann.

Dieses Konzept lässt sich auf SKilL-Graphen übertragen. Durch das Entfernen von nicht mehr verwendeten Objekten soll Speicherplatz eingespart werden. Als nicht mehr verwendete Objekte werden alle Knoten des Graphen bezeichnet, welche von einer Menge von Wurzelknoten nicht erreichbar sind. Anders als bei einem Garbage Collector für ein Programm ergeben sich die Wurzelreferenzen nicht automatisch durch Referenzen in Prozessorregistern, Stack und globalen Variablen. Deshalb muss die Menge der Wurzelknoten zusätzlich angegeben werden.

3.1. Allgemeine Funktionsweise

Als Grundlage für die Übertragung der Garbage Collection auf einen SKilL-Graphen wird zunächst die allgemeine Funktionsweise des Konzepts näher beleuchtet. Die Garbage Collection lässt sich zunächst in zwei abstrakte Phasen aufteilen. Diese werden in unterschiedlicher Weise durch die in den darauf folgenden Abschnitten beschriebenen Verfahren umgesetzt. Sowohl Jones und Lins [JL96] §2 als auch Wilson [Wil92] §2 sehen diese Verfahren als Basis für fortgeschrittenere Algorithmen der Garbage Collection. Jedes der Basisverfahren besitzt individuelle Eigenschaften, aufgrund welcher ein passendes Verfahren für diese Arbeit ausgewählt wird. Anschließend werden die Probleme der vorgestellten Verfahren behandelt.

Zwei Phasen Modell Cohen [Coh81] sowie Wilson [Wil92] §1 teilen die Garbage Collection in zwei abstrakte Phasen. Diese Phasen sind in Abbildung 3.1 zu sehen. Die Collection-Phase dient der Identifikation aller nicht mehr benötigter Objekte. Anschließend werden die identifizierten Objekte in der Freigabe-Phase als wiederverwendbarer Speicher an das Programm zurückgegeben. Beide Phasen können auf unterschiedliche Weisen implementiert werden. Die verschiedenen Implementierungsarten werden bei ihrer Verwendung in den Basisverfahren näher erläutert. Ebenso hängt die Art der Phasentrennung vom gewählten Verfahren ab. Die Trennung kann auf temporärer oder funktionaler Ebene stattfinden. Das bedeutet, die Phasen können auch ineinander verschachtelt sein.

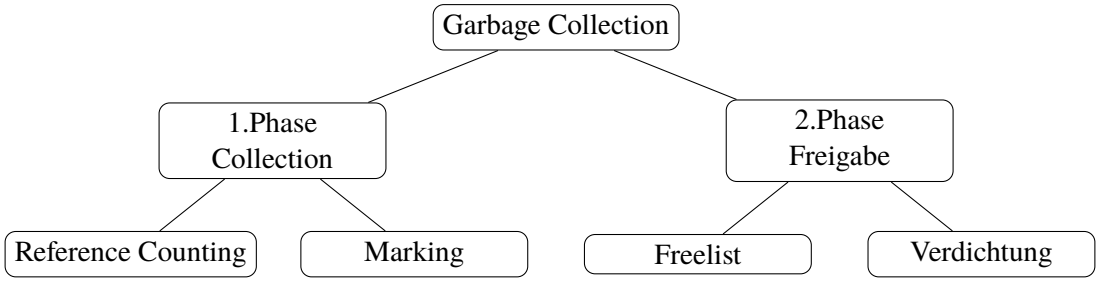

Abbildung 3.1.: Phasen der Garbage Collection

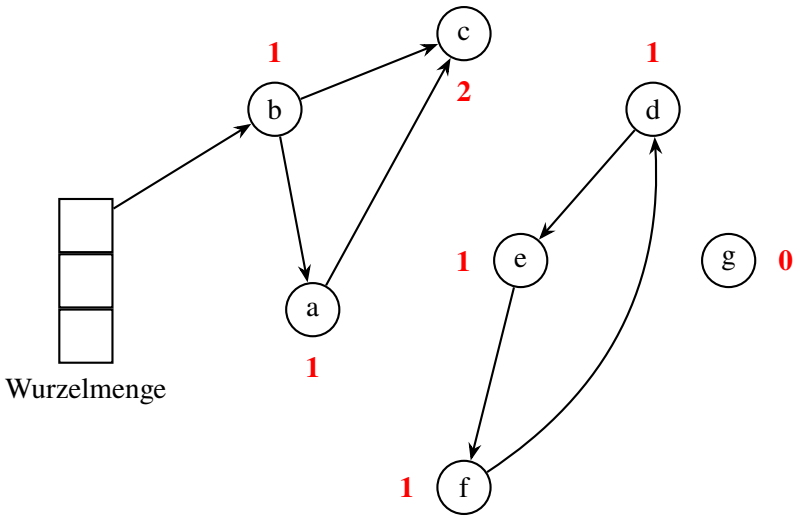

Abbildung 3.2.: Reference Counting. Objekte sind als Kreise dargestellt, Kanten sind Referenzen. Rote Nummern repräsentieren Referenzanzahl. d-e-f bilden einen Kreis. g kann entfernt werden.

3.1.1. Reference Counting

In diesem Verfahren wird jedem Objekt ein Zähler zugeordnet, welcher die Anzahl der Referenzen beinhaltet. Abbildung 3.2 zeigt ein Beispiel für einen Objektgraphen mit zugeordneten Zählern. Für jede neue Referenz auf das Objekt wird die Anzahl inkrementiert. Beim Löschen einer Referenz wird der Zähler des entsprechenden Objekts dekrementiert. Die Inkrement- und Dekrement-Operationen bilden die erste Phase der Garbage Collection. Eine Referenzanzahl von null bedeutet, dass es keine Referenzen mehr auf dieses Objekt gibt. Somit kann eine Freigabe des Speicherplatzes in eine Freelist erfolgen (2.Phase). Da das Objekt freigegeben wird, müssen alle Referenzfelder verfolgt und die entsprechenden Zähler dekrementiert werden. Somit ergibt sich eine Verschachtelung der abstrakten Phasen.

Für die Umsetzung des Verfahrens benötigt jedes Objekt einen Zähler. Dieser ist bei normalen Programmen in einem speziellen Header implementiert und vom Anwender nicht zu sehen. Zudem erfordert das Verfahren Inkrement- und Dekrement-Operationen. Die Operationen erfolgen sobald eine Änderung von Referenzen stattfindet. Somit hat die erste Phase der Garbage Collection einen

inkrementellen Charakter. Die zusätzliche Zeit durch das Verfahren wird im Idealfall auf viele Stellen verteilt. Dies bietet den Vorteil, dass das Programm keine langen Unterbrechungen durch die Garbage Collection erfährt.

Ein Nachteil des Reference Counting entsteht durch zyklische Referenzen im Graphen. Die Knoten im Zyklus erreichen nie die Referenzanzahl null, da es immer eine Referenz auf sie gibt. Dabei kann es vorkommen, dass ganze Kreisstrukturen nicht mehr von den Wurzelreferenzen erreichbar sind. Dennoch können sie durch das Reference Counting Verfahren ohne Erweiterungen nicht entfernt werden. In Abbildung 3.2 ist dies bei den Knoten *d*,*e* und *f* der Fall.

3.1.2. Mark-Sweep und Mark-Compact

Der Name Mark-Sweep bezeichnet die beiden Phasen, welche dieses Verfahren aufweist. In der Mark-Phase werden alle Objekte „markiert“, welche von den Wurzelreferenzen aus erreichbar sind. Dies geschieht, indem die Feldreferenzen der Objekte rekursiv verfolgt werden. Die Markierung kann dabei entweder durch ein Bit in jedem Objekt oder eine Tabelle für alle Objekte realisiert werden. Nach dem Setzen der Markierungen erfolgt die Freigabe der nicht erreichbaren Objekte in der Sweep-Phase. In dieser wird der gesamte Speicher nach unmarkierten Objekten durchsucht und deren Speicherplatz in einer Freelist freigegeben.

Mark-Compact ist eine Abwandlung des Mark-Sweep-Verfahrens. Die erste Phase funktioniert identisch. In der zweiten Phase wird dagegen der Speicherplatz verdichtet. Die Verdichtung erfolgt durch Verschiebung aller markierter Objekte in einen zusammenhängenden Speicherblock. Dadurch bleibt der restliche Speicher als ein großer Speicherblock zurück. Dies wirkt der Fragmentierung des Speichers entgegen, benötigt jedoch zusätzliche Zeit gegenüber der Freigabe in eine Freelist.

3.1.3. Copying

Beim Copying werden die Objekte, wie beim Mark-Compact Verfahren, in einen zusammenhängenden Speicherbereich verschoben. Dadurch ergibt sich eine implizite Garbage Collection. Der Unterschied zu Mark-Compact besteht in der Trennung der beiden abstrakten Phasen. Während die Mark- und Compact-Phasen zeitlich getrennt stattfinden, werden diese beim Copying kombiniert. Das Verfahren traversiert den Objektgraphen und kopiert die erreichbaren Objekte zeitgleich in einen zusammenhängenden Speicherbereich. Somit muss jedes Objekt nur einmal traversiert werden, wodurch Zeit eingespart wird.

Die Einsparung der Zeit wird durch zusätzlich benötigten Speicherplatz erkauft. Da während des Copying der Objektgraph zweifach im Speicher liegt, nutzt das Verfahren doppelt so viel Speicherplatz als eigentlich benötigt. Das Copying kann zum Beispiel implementiert werden, indem der Heap in zwei Hälften geteilt wird. Daraus folgt, dass nur die Hälfte des Heapspeichers verwendet werden kann.

3.1.4. Vergleich der Verfahren

Ein Vergleich der vorgestellten Verfahren kann nur im Bezug auf eine Anwendung erfolgen, da die Eigenschaften je nach Anwendungsfall verschiedene Auswirkungen haben. Somit werden in diesem Abschnitt die Verfahren auf den SKilL-Graphen übertragen und die Eignung für diesen Anwendungsfall überprüft.

Um das Reference Counting Verfahren umzusetzen, müssen in das generierte API ein Zähler in jedem Knoten des Graphen sowie Inkrement und Dekrement-Operationen implementiert werden. Dies führt zu Veränderungen an vielen Stellen des API. Diese Vorgehensweise soll den inkrementellen Charakter des Reference Counting und damit die Verteilung der zusätzlichen Zeit sicherstellen. Aufgrund des Einlesevorgangs vor der Garbage Collection entfällt dieser Vorteil, da sich die Inkrement- und Dekrement-Operationen auf die Zeit des Einlesens konzentrieren. Da der Graph nach dem Entfernen der nicht benötigten Objekte wieder in eine Datei geschrieben wird, bringt der inkrementelle Charakter des Reference Counting hier keinen Vorteil gegenüber den anderen Verfahren. Zudem muss für das Reference Counting eine Zyklenerkennung implementiert werden, welche bei den anderen Verfahren nicht notwendig ist.

Mark-Compact und Copying verhindern die Fragmentierung des Speichers auf Kosten zusätzlicher Rechenzeit oder Speicherplatzes. Diese Verfahren sind im Bezug auf den SKilL-Graphen nicht notwendig, da während des Schreibvorgang eine implizite Verdichtung des Speicherplatzes erfolgt[1]. Somit sind zusätzliche Rechenzeit bzw. Speicher überflüssig und können durch die Verwendung eines anderen Verfahrens eingespart werden.

Das Mark-Sweep Verfahren erfüllt alle Voraussetzungen, die in dieser Arbeit benötigt werden. Mark-Sweep bedarf keiner Zyklenerkennung oder starker Veränderung des API durch Inkrement- und Dekrement-Operationen. Somit ist es dem Reference Counting vorzuziehen. Gegenüber den Verfahren, welche den Speicher verdichten (Mark-Compact und Copying), wird Speicherplatz oder Rechenzeit eingespart. Damit ist Mark-Sweep das beste Verfahren für diese Arbeit.

3.1.5. Probleme der Verfahren

Die vorgestellten Verfahren weisen einige Probleme auf, welche vor allem technischer Natur sind. Diese werden im Folgenden kurz erläutert und im Bezug auf diese Arbeit untersucht.

Boehm und Weiser [BW88] und Wilson [Wil92] §1.3 sehen die Objektrepräsentation als Problem. Referenzen auf Objekte müssen von normalen Daten unterscheidbar sein, um die Wurzelreferenzen finden zu können. Zudem muss die Struktur der Objekte bekannt sein, da die Feldreferenzen verfolgt werden müssen. Diese Probleme entfallen für diese Arbeit, da die Knoten und Kanten des SKilL-Graphen klar durch das API definiert sind. Ferner wird die Menge der Wurzelknoten separat eingegeben. Eine Traversierung des Graphen ausgehend von den Wurzeln ist durch das API möglich, indem die Feldwerte der Objekte abgerufen werden.

[1] Bei Verwendung des Mark-Sweep-Verfahrens zwischen der Ein- und Ausgabe der Datei ergibt sich als Gesamtvorgang das Mark-Compact-Verfahren.

Algorithmus 3.1 Rekursiver Markierungs-Algorithmus

```
procedure ProcessObject(Reference ref)
    MarkObject(ref)
    type ← Type(ref)

    for all f ∈ Fields(type) do
        if ContainsReference(f) then
            for all r ∈ GetReferences(f, ref) do
                if NotMarked(r) then
                    processObject(r)
                end if
            end for
        end if
    end for
end procedure
```

Außerdem stellt die Unterbrechung des Programms durch die Garbage Collection ein Problem dar. Die Pausen im Programm sorgen für eine nichtdeterministische Komponente im zeitlichen Ablauf. Dies führt vor allem bei parallelen Anwendungen oder Echtzeitsystemen zu unerwünschtem Verhalten. Für diese Arbeit ist auch dieses Problem unerheblich, da die Aufgabe des Programms nur die Garbage Collection ist.

3.2. Umsetzung des Mark-Sweep Verfahrens

Der Vergleich in Abschnitt 3.1.4 legt Mark-Sweep als das am besten geeignete Verfahren für diese Arbeit fest. Die Markierungsphase kann durch unterschiedliche Algorithmen umgesetzt werden. Im Folgenden wird die Implementierung durch Rekursion und einen Stack behandelt. Beide Möglichkeiten benötigen eine Datenstruktur um die Markierung zu speichern. Diese wird in der anschließenden Sweep-Phase zum Entfernen der nicht markierten Objekte verwendet. Die Umsetzung der beiden Phasen erfolgt in der Programmiersprache Java.

3.2.1. Rekursive Implementierung

Die einfachste Möglichkeit, um die Markierungsphase umzusetzen, ist durch Rekursion. Von einem Startknoten ausgehend wird eine rekursive Prozedur für jeden direkten Nachbarknoten im Graph aufgerufen. Algorithmus 3.1 zeigt die rekursive Prozedur als Pseudocode. In dieser erfolgt zunächst eine Markierung des Objekts. Darauf folgend werden alle Felder des Objekts nach Referenzen durchsucht, um die zugehörigen Objekte rekursiv zu markieren.

Die Anzahl der Referenzen in jedem Feld ist abhängig von Feldtyp und Objekt. Grunddatentypen beinhalten keine Referenzen und können deshalb ignoriert werden. Referenzfelder enthalten entweder eine oder keine Kante (Nullreferenz). Zudem müssen Containertypen beachtet werden, welche eine beliebige Anzahl Referenzen enthalten können. Referenzfelder und Container definieren somit die Kanten im SKilL-Graph und müssen deshalb rekursiv traversiert werden.

Algorithmus 3.2 Markierungs-Algorithmus mit einem Stack

```
objectStack ← rootObjects
procedure ProcessObject
    while objectStack ≠ ∅ do
        ref ← Pop(objectStack)
        MarkObject(ref)
        type ← Type(ref)

        for all f ∈ Fields(type) do
            if ContainsReference(f) then
                for all r ∈ GetReferences(f, ref) do
                    if NotMarked(r) then
                        Push(objectStack, r)
                    end if
                end for
            end if
        end for
    end while
end procedure
```

3.2.2. Implementierung mit einen Stack

Ein Stack kann die rekursive Implementierung ersetzen. Die Basisidee hierfür wurde aus einer Implementierung von Felden [Fel18b] entnommen. Durch Änderung weniger Zeilen des vorherigen Algorithmus 3.1 entsteht Algorithmus 3.2. Anstelle eines rekursiven Aufrufs des Algorithmus werden alle neuen Referenzen auf einem Stack gesammelt (`Push`-Operation). Dafür wird der Heap-Speicherbereich des Programms verwendet. Der Algorithmus arbeitet den Stack ab bis keine Referenz mehr vorhanden ist. Anders als bei den rekursiven Aufrufen wird hier nur ein Stack-Frame verwendet.

3.2.3. Datenstruktur zur Markierung

Die Markierung eines Objektes kann zwei Zustände (markiert/nicht markiert) annehmen. Somit wird pro Objekt mindestens 1 Bit benötigt. Das Markierungsbit kann entweder als Feld in die Struktur des Objekts integriert oder in einer Tabelle umgesetzt werden. Aufgrund der Umsetzung der Algorithmen in Java, wird im folgenden speziell auf die Datenstrukturen dieser Programmiersprache eingegangen.

Eine Integration der Markierung als Feld in die Objekte erfordert in Java eine `boolean`-Variable. Die JVM-Spezifikation schreibt für diesen Datentyp keine exakte Größe vor [LYBB15] §2.3.4. Damit kann über die Größe der Variable zunächst keine Aussage getroffen werden. Die Berechnungen auf dem Typ `boolean` erfolgen aber mit Integer Operationen, somit ist eine maximale Größe von 32 Bit möglich.

Datei	Objektanzahl	Zeit in s		
		boolean	boolean[]	BitSet
php-cgi.iml.sf	14.525.541	37,34	66,86	66,09
php.iml.sf	14.576.961	37,87	65,19	66,90
php7-cgi.iml.sf	18.658.498	53,10	85,61	86,51
php7.iml.sf	18.725.527	53,45	86,08	86,88
php7dbg.iml.sf	19.528.973	55,96	93,93	90,94

Tabelle 3.1.: Durchschnittszeiten bei Verwendung der Markierungsdatenstrukturen. Für jede Datei wurde die Garbage Collection zehn mal durchgeführt.

Die Umsetzung der Markierungsbits in einer Tabelle kann durch ein `boolean`-Array implementiert werden. In der JVM wird das `boolean`-Array auf ein `byte`-Array abgebildet. Dadurch ergibt sich pro Objekt ein Speicherplatz von 8 Bit. Für jedes Objekt wird durch eine Indexberechnung ein Platz im Array bestimmt. Der Index hängt dabei von Typ und SKilL-ID ab:

$$\mathsf{Index}(obj) = \sum_{t \in BT} \mathsf{AnzahlObjekte}(t) + \mathsf{SKilL\text{-}ID}(obj) - 1$$
$$\text{mit } BT = \{Basistyp \mid Basistyp.ID < \mathsf{Basistyp}(obj).ID\}$$

Da die SKilL-ID durch eine laufende Nummerierung vom Basistyp aus vergeben wird, eignet sie sich als Index. Die erste SKilL-ID ist die eins, deshalb erfolgt in der Formel eine Normalisierung um den Array-Index null nutzen zu können. Da die SKilL-IDs jeweils vom Basistyp ausgehend nummeriert sind, gibt es für jeden Basistyp einen Offset im Array. Dieser wird durch die Summe berechnet.

Um den Speicherverbrauch noch weiter zu reduzieren, bietet Java den Datentyp `BitSet` [Ora15c]. Dieser bildet 64 `boolean`-Werte auf einen `long`-Wert ab. Somit wird im Idealfall[2] pro `boolean`-Wert nur ein Bit im Speicher verwendet. Der geringere Speicherverbrauch wird hier durch einen Overhead für das Lesen und Schreiben der Werte erkauft.

Für die effiziente Implementierung der Garbage Collection muss die Datenstruktur sehr schnell sein. Der Unterschied im Speicherverbrauch der Möglichkeiten ist gering genug, um diesen zu vernachlässigen. Tabelle 3.1 zeigt einen Performancevergleich der drei Datenstrukturen. Für den Vergleich wurden die Datenstrukturen in den Markierungsalgorithmus mit einem Stack eingebaut. Der komplette Garbage Collection Algorithmus wurde auf fünf verschiedenen Dateien jeweils zehn mal ausgeführt und der Durchschnitt errechnet. Die Datensätze entsprechen dabei den fünf größten der 60 Testdateien von Felden [Fel17a] §7.1.5. In allen Fällen ist die Wurzelmenge so gewählt, dass alle Knoten erreicht werden. Das `boolean`-Feld ist bei allen Messungen sehr viel schneller als die anderen beiden Möglichkeiten.

[2] Idealfall = Anzahl der `boolean`-Werte ist Vielfaches von 64 Bit.

3.2.4. Zusammensetzung der Implementierung

Zunächst muss ein Algorithmus für die Markierungsphase ausgewählt werden. Die rekursive Implementierung erstellt für jeden Aufruf einen neuen Stack-Frame. Somit ergibt sich ein höherer Platz- und Zeitbedarf gegenüber der Implementierung mit einer Stackstruktur auf dem Heap. Da SKilL-Graphen sehr viele Knoten haben können, führt der erhöhte Platzbedarf schnell zu einem Stack Overflow. Daher wird für die Markierungsphase die Implementierung mit einem Stack gewählt. Java bietet hierfür den Datentyp `ArrayDeque` [Ora15a].

Als Markierungsdatenstruktur wird die Integration eines `boolean`-Felds in jedem Objekt gewählt. Diese Variante hat im Vergleich die schnellsten Zeiten. Die Sweep Phase besteht im Anschluss aus einer Iteration über alle Objekte und dem Entfernen von nicht benötigten Objekten. Im Vergleich zur Markierungsphase besitzt diese Phase einen sehr viel geringeren Zeitaufwand, da die Objekte in Listenform durchlaufen werden.

3.3. Angabe der Wurzelmenge

Die Wurzelobjekte sollen sowohl alle Instanzen eines Typs als auch einzelne Objekte sein können. In der Implementierung des Markierungs-Algorithmus 3.2 werden die Objekte zu Beginn auf den Stack geschoben. Somit muss die Eingabe der Wurzelmenge lediglich in Objekte übersetzt werden. Dazu bekommt die Garbage Collection eine Menge von `CollectionRoots` übergeben.

`CollectionRoot` bezeichnet eine Datenstruktur, welche entweder einen Typ oder ein einzelnes Objekt speichert. Dazu wird immer der Name des Typs benötigt. Bei einer reinen Übergabe des Typnamens werden bei der Garbage Collection alle Objekte dieses Typs als Wurzeln verwendet. Um einzelne Wurzelobjekte definieren zu können, bietet die `CollectionRoot`-Struktur einen zweiten Übergabeparameter in Form eines Integers. Dieser stellt die SKilL-ID des Objekts dar. Das Objekt ist somit durch Typname und SKilL-ID eindeutig identifiziert und kann als Wurzel verwendet werden. Da die Datenstruktur in der Lage ist alle Instanzen eines Typs oder einzelne Objekte zu definieren, ist auch eine Kombination beider Wurzeldefinitionen möglich.

3.4. Weitere Schritte

Eine Garbage Collection ist nach dem Entfernen der nicht benötigten Objekte im Standardfall abgeschlossen. Für diese Arbeit bieten sich jedoch weitere Schritte an, um den Speicherplatz der Binärdatei weiter zu verkleinern.

Zunächst kann das Typsystem, welches in der Datei gespeichert wird, verkleinert werden. Dies geschieht in zwei Schritten:

1. **Nicht benötigte Typen entfernen:** Sobald es von einem Typ keine Objekte mehr gibt, gilt dieser Typ als „leer“. Leere Typen beinhalten keine zusätzliche Information für den SKilL-Graphen und können deshalb entfernt werden. Zu beachten ist, dass auch Objekte von Subtypen für die Objektanzahl relevant sind.

2. **Nicht benötigte Felder entfernen:** Nachdem im 1.Schritt einige Typen entfernt wurden, können Felder mit Referenzen auf diese Typen nur den Wert null haben. Somit speichern diese Felder keine nützliche Information und sollten ebenfalls entfernt werden. Einen Sonderfall bilden hier Container, da in diesen Informationen über die Menge der null-Referenzen übertragen werden können. Für diesen Sonderfall wird ein Flag implementiert, welches die Container und deren Grunddatentypen erhält.

Die beiden Schritte erfolgen durch ein ähnliches Vorgehen wie in Abschnitt 4.2, welches auf das Entfernen einzelner Typen ausgelegt ist. In beiden Fällen sind nahezu die gleichen Problemstellungen vorhanden[3]. Die Umsetzung wird für die Garbage Collection lediglich auf das Entfernen von mehreren Typen optimiert.

In Folge der Verkleinerung des Typsystems können die zugehörigen Strings ebenfalls aus der Datei entfernt werden. Für die Umsetzung dieses Entfernens wird ein Teil des Schreibvorgangs ausgenutzt. Das API sammelt vor dem Schreiben der Binärdatei alle Strings, um die Vollständigkeit der String-Menge sicherzustellen. Dieser Vorgang kann genutzt werden, indem am Ende der Garbage Collection alle Strings gelöscht werden. Infolgedessen werden nur die für das Typsystem notwendigen Strings in den Schreibvorgang einbezogen.

[3] Alle Probleme bis auf das Löschen der Objekte, welches für die Garbage Collection nicht auftritt.

4. Entfernen von Typen und Feldern

Dieses Kapitel beschäftigt sich mit dem Entfernen einzelner Felder oder ganzer Typen aus der Binärdatei. Aufgrund des Aufbaus der Binärdatei führt ein einfaches Löschen von Feldern oder Typen zu einer beschädigten Binärdatei, da sie noch verwendet werden. In der Regel ist die Datei anschließend nicht mehr lesbar. Deshalb sind weitere Schritte notwendig, um eine valide Ergebnisdatei zu erhalten.

4.1. Felder entfernen

Um ein Feld entfernen zu können, ist ein Verständnis für die Beziehung zwischen Feld und Typ notwendig. Jedes Feld ist einem bestimmten Typ zugeordnet. Für die Serialisierung besitzen Felder einen Index, der ihre Position in ihrem zugeordneten Typ angibt. Die Details der Serialisierung sind an dieser Stelle nicht wichtig. Der Index folgt dem Prinzip einer laufenden Nummer. Das heißt, er beginnt in jedem Typ bei eins und wird für die folgenden Felder jeweils um eins inkrementiert. Es darf keine Lücken in der Indizierung geben, da dies zu Fehlern beim Einlesen führt. Somit müssen beim Entfernen eines Feldes die Indizes aller nachfolgender Felder um eins dekrementiert werden.

Die Zuordnung eines Feldes zu einem bestimmten Typ ermöglicht die mehrfache Nutzung eines Feldnamens. Eine Ausnahme sind Super- oder Subtypen, welche kein Feld des gleichen Namens haben dürfen. Aufgrund der Möglichkeit einer Mehrfachnutzung muss das Feld durch Verwendung des Typnamens eindeutig bestimmt werden. Die Kombination aus Feld- und Typname bildet die Eingabe für das Entfernen des Feldes. Somit ergeben sich folgende logische Schritte, um ein Feld zu löschen:

1. **Typ finden:** Über den Namen wird der passende Typ im Typsystem gefunden. Diese Operation ist in konstanter Zeit möglich, da bereits eine Abbildung der Namen auf ihre Typen besteht.
2. **Feld finden und entfernen:** Die Felder eines Typs sind in einer Liste gespeichert. Durch eine lineare Suche über diese Liste wird das zu entfernende Feld ermittelt. Anschließend wird das gefundene Feld aus der Liste entfernt.
3. **Indizes erneuern:** Aufgrund der Feldentfernung verschiebt sich der Index der nachfolgenden Felder. Es ist an dieser Stelle ausreichend die Indizes aller folgenden Felder um eins zu dekrementieren.

In der Implementierung der Feldentfernung werden die logischen Schritte zwei und drei zu einem Schritt verbunden. Damit genügt eine Iteration über alle Felder des Typen. Aus diesem Grund ist die Laufzeit des Algorithmus linear zur Menge der Felder im Zieltyp.

4.2. Typen entfernen

Das Entfernen von Typen erfordert einige Schritte mehr als die Feldentfernung. Der Grund dafür ist die zentrale Position der Typen im API. Damit müssen einige Punkte mehr beachtet werden als bei den Feldern. Nachfolgend sind alle sich ergebenden Probleme und ihre möglichen Lösungen aufgeführt. Falls es mehrere Möglichkeiten gibt, erfolgt eine begründete Entscheidung für eine der Lösungen. Anschließend fließen die gelösten Probleme in die Umsetzung ein.

Objekte Das Entfernen eines Typs wirkt sich auf seine Objekte aus. Diese müssen in Folge des Entfernens entweder gelöscht oder, falls vorhanden, auf den Supertyp projiziert werden. Da eine Projektion bereits durch Kapitel 5 abgedeckt ist, werden die Objekte in der Umsetzung der Typentfernung gelöscht. Damit sind auch die Objekte aller Subtypen gemeint, da diese auch Instanzen des Supertyps sind.

Subtypen Ein weiteres Problem stellen Subtypen dar. Diese verlieren durch das Entfernen ihren Supertyp. Damit gehen auch alle Typinformationen verloren, welche weiter oben in der Typhierarchie liegen. Die Folge sind Probleme in der Typisierung und Informationsverlust. Die Lösungsmöglichkeiten sind hier das Entfernen der Subtypen oder die Veränderung der Typhierarchie. Wie bei den Objekten fällt hier die Entscheidung auf das Löschen, da Kapitel 5 die Projektionsmöglichkeit bietet.

Feldreferenzen Da die Objekte des Typs von Feldern anderer Typen referenziert werden können, sind diese Felder ebenfalls zu betrachten. Zunächst muss jedes Feld entfernt werden, welches den Feldtyp des entfernten Typs oder seiner Subtypen verwendet. Der Grund hierfür ist, dass der Feldtyp nicht mehr existiert und damit auch nicht referenziert werden kann. Ebenso sind Referenzen auf die Objekte durch einen der Supertypen des entfernten Typs möglich. In diesem Fall ist keine Aktion nötig, da beim Schreibvorgang automatisch eine Nullreferenz geschrieben wird falls das Objekt gelöscht wurde.

Typ-IDs Jeder Typ besitzt eine Typ-ID, welche ähnlich wie der Index bei den Feldern wichtig für die Serialisierung ist. Für Anwendertypen ist die Typ-ID eine laufende Nummer ab der Startnummer 32. Die Anfangsnummer ist im Binärformat festgelegt. Nach dem Entfernen des Typs und seiner Subtypen müssen die Typ-IDs der in Typordnung nachfolgenden Typen neu vergeben werden.

Typhierarchie Für die effiziente Traversierung der Typhierarchie enthält jeder Typ eine Referenz auf den nächsten Typ in der Typhierarchie. Abbildung 4.1 zeigt ein Beispiel für diese Referenzen. Das Entfernen des Typ *D* in diesem Beispiel hätte zur Folge, dass Typ *C* von *A* aus nicht mehr erreichbar ist. Eine Neugenerierung erfolgt in umgekehrter Typordnung. Die Subtypen setzen die Referenzen ihrer Supertypen. Falls eine Überschreibung erfolgen soll, wird die alte Referenz an den zuletzt besuchten tiefsten Punkt der Typhierarchie verschoben. In Abbildung 4.1 ist diese Verschiebung für die Referenz auf *C* erfolgt.

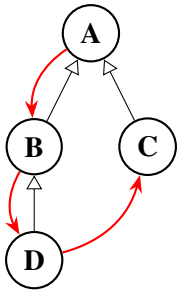

Abbildung 4.1.: Typhierarchie mit Referenzen zur effizienten Traversierung der Hierarchie in rot. Jeder Typ ist damit in der Lage die Typhierarchie nach unten zu traversieren. Die Traversierung endet wenn es keine Referenz gibt (null-Pointer) oder die Starthöhe in der Hierarchie erreicht ist(z.B. Start bei B und Ende bei C).

Aus den Problemen und zugehörigen Lösungen ergeben sich Teile der Umsetzung. Der Name des zu entfernenden Typs dient als Eingabe. Damit werden folgende Teilschritte zum Entfernen eines Typs benötigt:

1. **Typ finden:** Zunächst muss der Typ über seinen Namen identifiziert werden. Wie beim Entfernen eines Feldes ist dies in konstanter Zeit möglich, da bereits eine Abbildung der Namen auf ihre Typen besteht.
2. **Objekte löschen:** Im zweiten Schritt werden die Objekte des Typs gelöscht. Im Entfernen sind auch die Instanzen der Subtypen eingeschlossen. Diese Aktion hat zur Folge, dass während des Schreibvorgangs Referenzen auf diese Objekte zu Nullreferenzen werden.
3. **Subtypen identifizieren:** Durch Traversierung der Typhierarchie ist eine Identifikation aller Subtypen möglich. Dazu werden die in Abbildung 4.1 beschriebenen Referenzen verwendet. Nach dieser Operation ergibt sich eine Menge zu entfernender Typen.
4. **Felder entfernen:** Nach Identifizierung der Subtypen erfolgt das Entfernen aller Felder, welche den zu entfernenden Typ oder einen seiner Subtypen referenzieren. In dieser Phase muss über alle Felder der übrigen Typen iteriert werden. Das Entfernen erfolgt ähnlich zur Umsetzung der Feldentfernung in Abschnitt 4.1. Der Unterschied besteht in der Suche nach den Typen der Felder anstatt ihrer Namen.
5. **Typen entfernen:** In diesem Schritt wird der Typ und seine Subtypen entfernt. Diese Operation lässt sich durch ein einfaches Löschen der Typen aus der Gesamtliste umsetzen. Das Entfernen ist an dieser Stelle gefahrlos möglich, da in den vorherigen Schritten sämtliche Abhängigkeiten getrennt und die Typen damit isoliert wurden.
6. **Typ-IDs und Typhierarchie:** Aufgrund des Entfernens werden im letzten Schritt alle Typ-IDs sowie die Referenzen zur Traversierung der Typhierarchie neu generiert. Dies erfolgt in zwei Iterationen über die Typenliste. In der ersten Iteration werden die Typ-IDs in Typordnung vergeben. Darauf folgt eine zweite Iteration, welche in umgekehrter Reihenfolge die Referenzen zur Traversierung der Typhierarchie generiert.

5. Spezifikationsprojektion

Die Spezifikationsprojektion bildet den größten Teil dieser Arbeit. In diesem Kapitel wird zunächst die Grundidee der Spezifikation behandelt. Anschließend werden bereits vorhandene Ansätze vorgestellt und analysiert. Danach erfolgt die Entwicklung einer Umsetzung für die Projektion.

5.1. Grundidee

Die Grundidee der Projektion auf eine andere Spezifikation ist in Abbildung 5.1 dargestellt. Als Eingabe dienen eine Binärdatei und eine Spezifikation. Der SKilL-Graph wird während der Projektion auf das neue Typsystem aus der Spezifikation abgebildet. Das Ergebnis ist eine Binärdatei, welche den projizierten SKilL-Graphen und das neue Typsystem enthält.

Vor der Projektion auf die Zielspezifikation muss zunächst überprüft werden, ob dies überhaupt möglich ist. Das bedeutet, alle Werte im SKilL-Graphen müssen zu den Vorgaben der Spezifikation passen. Diese Prüfung ist insbesondere für die Änderung von Feldtypen wichtig, weil die Daten der Felder zum neuen Feldtyp passen müssen.

5.2. Verwandte Arbeiten

Für das Problem der Projektion auf ein neues Typsystem existieren bereits Ansätze. Diese werden in diesem Abschnitt vorgestellt und im Bezug auf diese Arbeit analysiert.

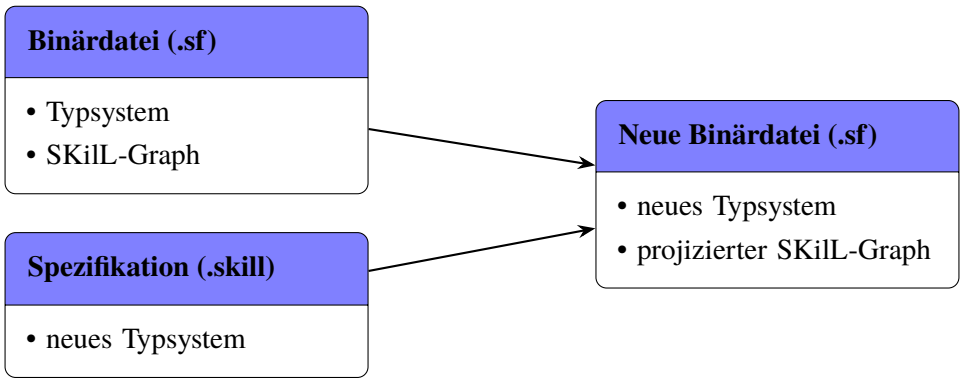

Abbildung 5.1.: Grundidee der Projektion auf eine andere Spezifikation. Die Spezifikation gibt das neue Typsystem vor. Der SKilL-Graph aus der Binärdatei wird auf das neue Typsystem projiziert.

Listing 5.1 Einfaches Anwendungsbeispiel von Dozer

```
1 Mapper mapper = DozerBeanMapperBuilder.buildDefault();
2 ObjectA a = ...;
3 ObjectB b = mapper.map(a, ObjectB.class);
```

5.2.1. Dozer

Dozer[Buz18] ist in der Lage eine JavaBean-Komponente auf eine andere abzubilden. Java Beans sind besondere Softwarekomponenten [Mic97]. Die Abbildung wird dazu benötigt, Daten von einem Objekt in ein Neues zu kopieren. Der häufigste Anwendungsfall für das Framework ist eine Schichtarchitektur, in welcher die Objekte vor der Weitergabe in die nächste Schicht umgewandelt werden müssen. Das Konzept lässt sich aber auch auf Objekte von einfachen Java-Klassen anwenden.

Listing 5.1 zeigt ein einfaches Anwendungsbeispiel des Frameworks. Das Standard-Mapping bildet Felder mit gleichen Namen aufeinander ab. Im Beispiel wird ein neues Objekt der Klasse `ObjectB` erzeugt. Jedes Feld von `a` wird in das neue Objekt übertragen, falls ein Feld mit gleichem Namen dort existiert. Haben zwei gleichnamige Felder unterschiedliche Datentypen erfolgt eine Konvertierung, falls dies möglich ist.

Das Standard-Mapping von Dozer kann vom Anwender angepasst werden. Damit ist es möglich auch Felder unterschiedlicher Namen aufeinander abzubilden. Die Anpassungen können durch eine XML-Datei, Annotations oder ein API erfolgen. Zudem ist es möglich Abbildungen zwischen Klassen zu definieren. Dies ist notwendig, da Felder Anwenderklassen referenzieren können.

Referenzen auf Anwenderklassen werden rekursiv abgearbeitet, falls eine Abbildung existiert. Dies ermöglicht die automatische Übertragung ganzer Bäume in ein neues Typsystem. Bei Anwendung dieses Konzepts auf einen Graphen mit zyklischen Referenzen bildet sich eine Endlosschleife. Um die rekursive Abarbeitung auf einem Graphen anwenden zu können, müssen also zunächst die Zyklen durchbrochen werden.

Dozer verwendet eine Abbildung zwischen bestehenden Typen. Daher muss das Typsystem schon im Voraus durch entsprechende Klassen definiert sein. SKilL verwendet ein änderungstolerantes Typsystem, welches in den Dateien enthalten ist. Eine Anwendung von Dozer im Kontext von SKilL ist somit nicht möglich, da das Typsystem auf Ein- und Ausgabeseite erst zur Laufzeit bekannt ist.

5.2.2. Stream API

Das Java Stream API [Ora15e] ermöglicht eine ähnliche Objektumwandlung wie Dozer. Das API beinhaltet verschiedenen Operationen, welche einen Stream von Objektreferenzen als Ein- und Ausgabe haben. Somit können die Stream-Operationen in einer Pipeline hintereinander ausgeführt werden. Eine dieser Operationen ist die `map()`-Funktion, welche eine Objektkonvertierung ermöglicht.

Listing 5.2 Einfaches Anwendungsbeispiel der Stream-Operation *map()*

```
1  // example list of ObjectA
2  List<ObjectA> as = ...;
3
4  List<ObjectB> bs = as.stream()
5          .map(a -> {
6                  ObjectB b = new ObjectB();
7                  // transfer data of a to b
8                  return b;
9          })
10         .collect(Collectors.toList());
```

In Listing 5.2 ist ein einfaches Beispiel zur Objektkonvertierung zu sehen. In diesem wird die `map()`-Funktion auf einen Stream der Klasse `ObjectA` angewendet. Als Parameter der Operation wird die Konvertierungsfunktion übergeben. Anstelle einer Lambda-Funktion kann hier auch eine Funktion der Klasse `ObjectA` verwendet werden.

Durch die Angabe einer Konvertierungsfunktion ist das Stream API sehr flexibel. Damit ist die Konvertierung mächtiger als die von Dozer. Gleichzeitig bildet die Konvertierungsfunktion einen Nachteil gegenüber Dozer. Für jede Konvertierung zweier Objekte muss eine eigene Funktion erstellt werden. Das Konzept von Dozer ist an dieser Stelle einfacher.

Das Problem von Dozer im Bezug auf diese Arbeit bleibt auch für das Stream API bestehen. Alle an den Konvertierungen beteiligten Klassen müssen schon im Voraus bekannt sein. Dies ist durch die änderungstolerante Implementierung des Typsystems von SKilL aber nicht möglich. Daher ist auch das Stream API ungeeignet für diese Arbeit.

5.2.3. Serialisierung fremder Typen mit SKilL

Weißer [Wei16] verfolgt ein anderes Konzept als die anderen beiden Ansätze. Die Arbeit macht es möglich eine bestehende Java-Codebasis in die SKilL-Serialisierung einzubinden. Damit wird die Integration von bestehenden Bibliotheken ermöglicht, welche nur als `.class`-Dateien vorliegen. Abbildung 5.2 zeigt die hierzu entwickelte Architektur. Die Begriffe wurden teilweise ersetzt, um zu den bisher verwendeten Begriffen zu passen.

Die Architektur ist eine Weiterentwicklung der Standardarchitektur aus [Fel17b], welche nur die linke Seite der Abbildung und die Codegenerierung umfasst. Eine Einbindung fremder Typen erfordert als Eingabe die Java `.class`-Dateien dieser Typen und eine Mappingdatei. Die Grammatik für die Erstellung der Mappingdatei kann [Wei16] §2.3 entnommen werden. Diese ist für ein Grundverständnis der Architektur nicht wichtig. Aus den `.class`-Dateien wird ein Typkontext erstellt. Anschließend werden die beiden Typkontexte mithilfe des Mappings vereinigt. Dabei entsteht eine Menge von Typregeln. Diese werden durch einen Typ Checker überprüft. Zum Schluss erzeugt der Code Generator das Serialisierungs-API.

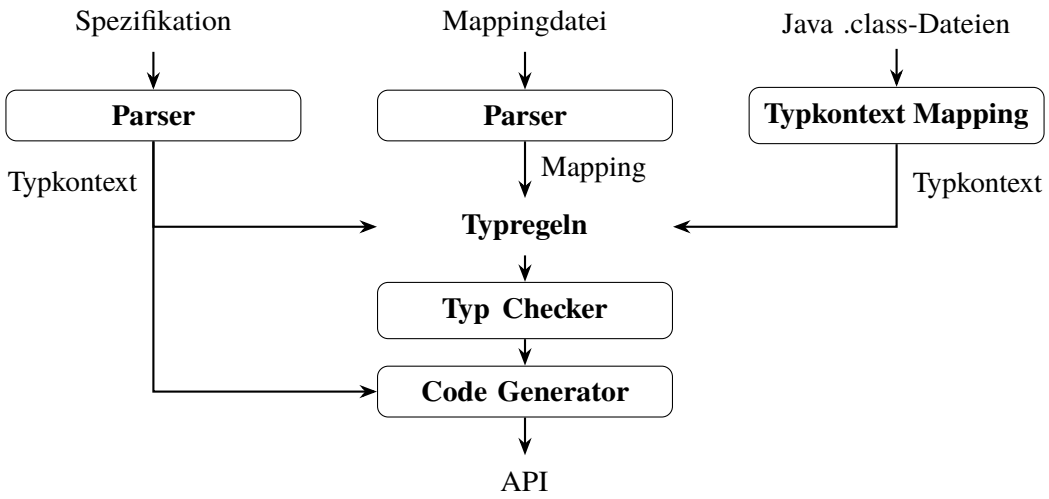

Abbildung 5.2.: Architektur zur Erzeugung der Serialisierungs-API fremder Typen
Quelle: eigene Darstellung, in Anlehnung an [Wei16] §2.1

Die Übertragung dieses Konzepts auf diese Arbeit scheint zunächst möglich, indem anstatt der Java .class-Dateien die Binärdatei als Eingabe verwendet wird. Aus dieser muss das Typsystem extrahiert und in einen Typkontext umgewandelt werden. Durch das Mapping wird die neue Spezifikation mit dem alten Typkontext verbunden. Das erzeugte API kann anschließend zur Umwandlung des alten in das neue Typsystem verwendet werden.

Allerdings, können aufgrund der anderen Aufgabenstellung einige Typsystem-Operationen nicht durchgeführt werden. Der Grund hierfür ist das verwendete Eins-zu-Eins Mapping zwischen Typen, welches in [Wei16] §2.5.2 erläutert wird. Damit ist es nicht möglich den Typ eines Felds zu verändern. Sobald zwei Felder aufeinander abgebildet werden, bilden die Typen eine Eins-zu-Eins Abbildung. Somit müssen alle Felder des Ausgangstyps auf den gleichen Zieltyp abgebildet werden, da sonst ein Fehler im Typ Checker auftritt. Zudem kann jeder Typ genau einmal abgebildet werden. Somit werden auch Projektionen von zwei oder mehr Typen auf einen Typ unterbunden.

Ein weiterer Nachteil dieser Vorgehensweise ist die Erzeugung eines APIs. Damit müsste für jede Abbildung auf eine neue Spezifikation ein API erzeugt, und dieses dann auf die Binärdatei angewendet werden. Somit ergibt sich hier ein Zeit- und Platzoverhead, der nicht notwendig ist. Aufgrund der fehlenden Typsystemoperationen und dem Overhead durch die API-Erzeugung ist auch diese Vorgehensweise nicht ausreichend. Anders als bei den anderen beiden Methoden wäre die Umsetzung dieser Arbeit mit einigen Kompromissen mithilfe dieses Konzepts möglich.

5.3. Architektur der Projektion

Für die Projektion gibt es grundsätzlich zwei verschiedene Ansätze. Zum Einen kann die bestehende Binärdatei so angepasst werden, dass sie auf die neue Spezifikation passt. Die andere Möglichkeit baut zunächst das neue Typsystem auf, um anschließend die Daten vom alten in das neue Typsystem

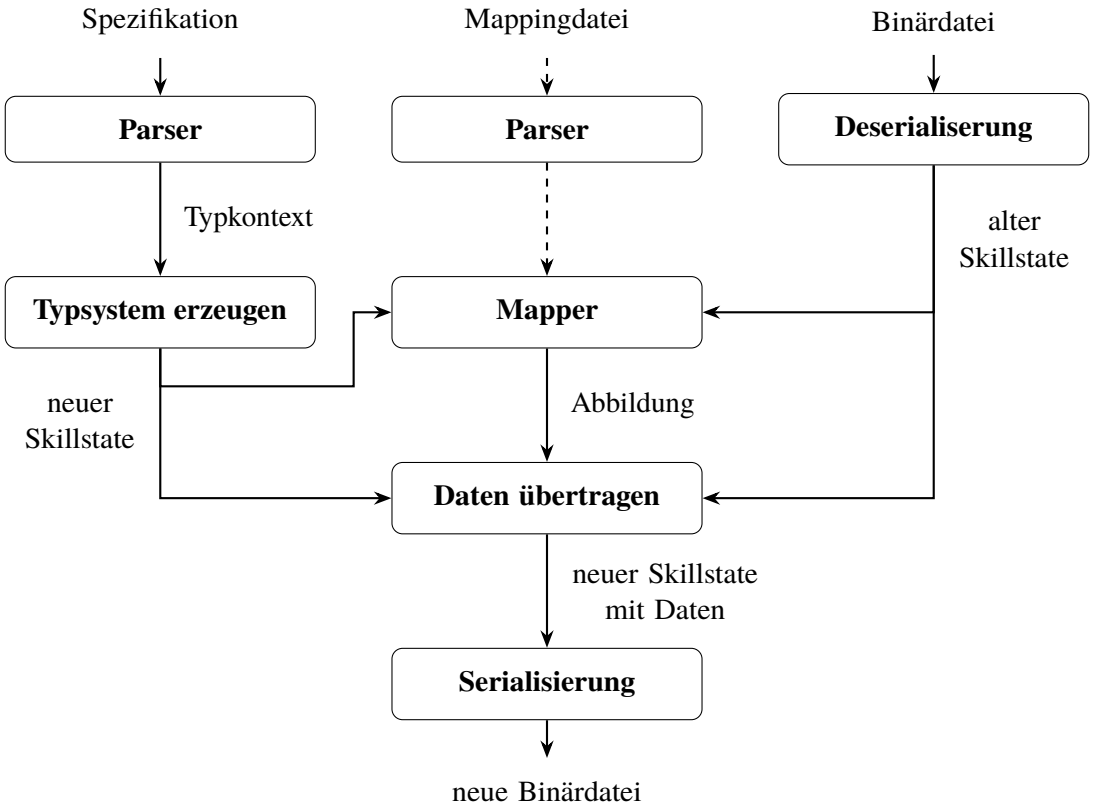

Abbildung 5.3.: Architektur der Spezifikationsprojektion

zu übertragen. Die einzig sinnvolle Variante für diese Arbeit ist die Letztere, da diese eine unkompliziertere Struktur hat. Eine Anpassung des Typsystems hat zur Folge, dass eine Datenstruktur sowohl das alte als auch das neue Typsystem repräsentieren muss.

Die Architektur der Spezifikationsprojektion ist in Abbildung 5.3 zu sehen. Im oberen Teil sind die Eingabedateien und Einlesevorgänge dargestellt. Der Parser für die Spezifikation ist bereits vorhanden und der Code zur Deserialisierung wird vom API bereitgestellt. Anschließend wird aus dem Typkontext das Typsystem erzeugt. Nun liegen beide Eingabedateien in derselben internen Darstellung vor. Der Mapper erzeugt eine Abbildung zwischen den Typsystemen der Skillstates. Mithilfe dieser Abbildung werden die Daten des alten Skillstate auf den Neuen übertragen. Die anschließende Serialisierung des neuen Skillstate erfolgt durch eine Funktion des API. In den folgenden Abschnitte 5.4 bis 5.7 werden die einzelnen Teilschritte der Projektion näher betrachtet.

Zusätzlich zu Spezifikation und Binärdatei kann eine dritte Eingabedatei verwendet werden. Diese ermöglicht eine Erweiterung der im Mapper erzeugten Abbildung. Ohne Angabe dieser Datei wird als Abbildung eine reine Namensäquivalenz verwendet. Mithilfe der optionalen Zusatzdatei können die Namen und somit die Abbildungen verändert werden. Eine detaillierte Beschreibung dieser Erweiterung erfolgt in Abschnitt 5.8.

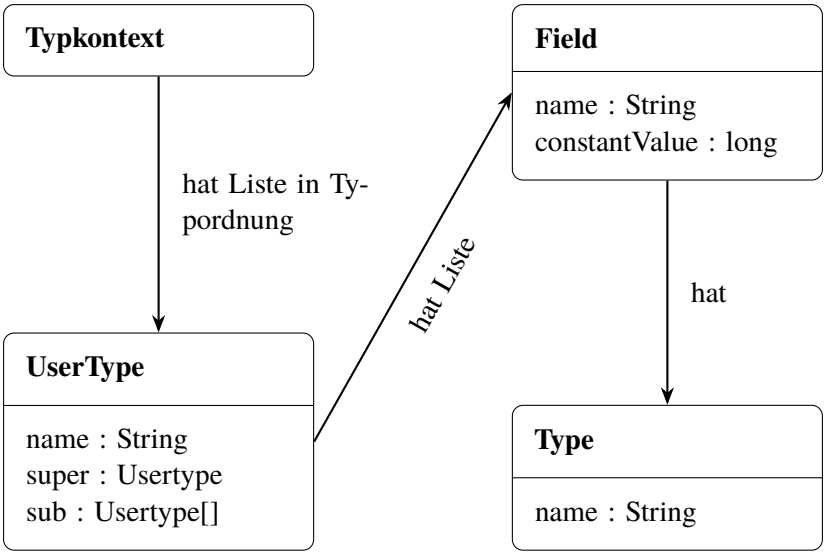

Abbildung 5.4.: Typkontext der Spezifikation

Auf den ersten Blick ähnelt die Architektur dem zuvor vorgestellten Konzept von Weißer [Wei16]. Jedoch bildet die Ebene auf welcher die Abbildung stattfindet einen großen Unterschied. Das in dieser Arbeit verwendete Mapping findet auf der internen Darstellung des API anstelle der Spezifikationsebene statt. Dies ermöglicht die direkte Umsetzung der Projektion und Ausgabe einer Binärdatei ohne den Umweg über ein API gehen zu müssen.

5.4. Typsystem erzeugen

Die Spezifikation wird in einer für Menschen lesbaren Datei mit der Endung `.skill` angegeben. Diese kann nicht durch das API der Binärdateien eingelesen werden. Somit wird hier ein spezieller Parser benötigt. Eine Lösung hierfür bietet das Frontend des Codegenerators [Fel18a]. Der enthaltene Parser ist in der Lage die Spezifikation einzulesen und in eine Objektdarstellung umzuwandeln. Die Aufgabe dieses Teilschritts ist es, das Typsystem aus dieser Objektdarstellung aufzubauen.

Die Objektdarstellung der Spezifikation wird Typkontext genannt. Abbildung 5.4 zeigt eine strukturelle Darstellung des Typkontexts. Alle Typen aus der Spezifikation sind als Objekte der Klasse `UserType` vorhanden. Diese sind in einer Liste des Typkontext in Typordnung enthalten. Jeder Typ enthält Referenzen auf andere Typen, um die Typhierarchie darzustellen. Zudem ist eine Liste von Feldern enthalten, welche wiederum ihren Typ kennen. Falls das Feld einen konstanten Wert besitzt, ist dieser im Feld gespeichert. Dies kann nur bei ganzzahligen Datentypen vorkommen. Deshalb ist der konstante Wert ein `long`.

Aus dem vorhandenen Typkontext muss die interne Darstellung des API erzeugt werden, um in den weiteren Schritten über eine Abbildung die Daten zu übertragen. Die Erzeugung der Darstellung erfolgt in zwei Phasen. Zunächst müssen alle Typen erzeugt werden, da Felder Referenzen auf diese Typen bilden können. Während der Typerzeugung wird gleichzeitig die Typhierarchie aufgebaut. Dies

ist möglich indem die Typen in Typordnung erstellt werden. Somit ist garantiert, dass Supertypen bereits vor ihren Subtypen existieren. Im Anschluss an die Erzeugung erfolgt eine erneute Iteration über die Typen um deren Felder zu instanziieren.

5.5. Abbildung zwischen Typsystemen

Um die Daten vom alten auf das neue Typsystem übertragen zu können, wird eine Abbildung benötigt. Ein Mapping zwischen den Typen des alten und neuen Systems ist an dieser Stelle ausreichend. Über diese Abbildung ist es möglich sowohl die Objekte als auch die Werte der Felder zu übertragen.

5.5.1. Abbildungsfunktion

Die Abbildung wird auf Basis von Namensäquivalenzen gebildet. Das bedeutet ein Typ des alten Systems wird auf den Typ mit demselben Namen im neuen Typsystem abgebildet. Falls kein Typ mit diesem Namen mehr existiert, wird der Typ gelöscht oder seine Daten auf einen Supertyp projiziert.

In Gleichung (5.1) ist die Funktion der entstehenden Abbildung zu sehen. Diese hat die Form: $f : Typ \rightarrow Typ$. Der erste Fall ist die Namensäquivalenz zwischen altem und neuem Typsystem. Die Funktion *name()* extrahiert hier lediglich den Namen des Typs. In Abschnitt 5.8 wird diese Funktion erweitert, um flexiblere Abbildungen zu ermöglichen. Falls es keine direkte Namensäquivalenz gibt, wird eine Projektion auf einen Supertyp versucht. Dies erfolgt durch eine rekursive Suche nach einer Namensäquivalenz über alle Supertypen bis zur Wurzel der Typhierarchie. Für den Fall, dass weder eine Namensäquivalenz existiert noch eine Projektion auf einen Supertypen möglich ist, gibt es den letzten Fall. Eine Abbildung auf null repräsentiert somit den Funktionswert eines Typs für den kein Mapping gefunden werden kann. Dies entspricht der Löschung eines Typs.

$$f(alterTyp) = \begin{cases} neuerTyp & \text{falls } name(alterTyp) = name(neuerTyp) \\ f(supertyp(alterTyp)) & \text{falls } name(alterTyp) \neq name(neuerTyp) \\ & \quad \wedge \quad supertyp(alterTyp) \neq null \\ null & \text{sonst} \end{cases} \tag{5.1}$$

Die Funktion vereinfacht sich stark, wenn sie in Typordnung berechnet wird. In diesem Fall sind die Funktionswerte von Supertypen zur Berechnungszeit bereits bekannt. Damit ist der zweite Fall durch einen einfachen Look-up realisierbar.

Im zweiten und dritten Fall der Gleichung wird der Anwender am Ende der Projektion informiert. Dadurch soll der Anwender über die Operationen der Abbildung aufgeklärt werden. Falls die Intention des Anwenders eine Projektion oder ein Entfernen des Typs war, kann die Information ignoriert werden.

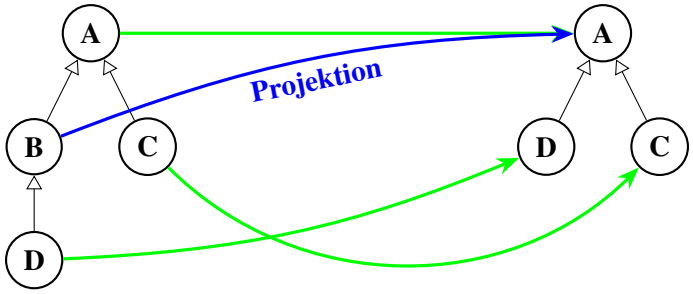

Abbildung 5.5.: Beispielabbildung zwischen Typsystemen 1. grün = Namensäquivalenz, blau = Projektion.

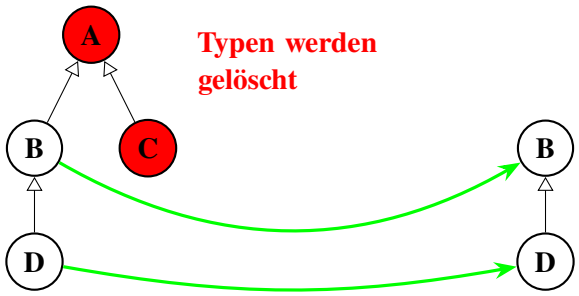

Abbildung 5.6.: Beispielabbildung zwischen Typsystemen 2. grün = Namensäquivalenz, rot markiert sind Abbildungen auf null.

5.5.2. Beispiele

In diesem Abschnitt werden Beispiele aufgeführt, um die Auswirkungen der Abbildungsfunktion deutlicher zu machen. Die Abbildung geht in beiden Fällen von links nach rechts. Somit ist links das alte und rechts das neue Typsystem.

Abbildung 5.5 zeigt die Fälle eins und zwei der Funktion. Die grünen Pfeile zeigen die trivialen Namensäquivalenzen. Der blaue Pfeil zeigt die Projektion des Typs *B* auf den Typ *A*. *B* hat selbst kein Namensäquivalent. Deshalb wird die Abbildung des Supertyps verwendet. Da dieser einen Funktionswert besitzt, erfolgt eine Projektion auf den Typ *A*.

Im zweiten Beispiel (Abbildung 5.6) werden die Fälle eins und drei dargestellt. Wieder sind die grünen Pfeile die trivialen Namensäquivalenzen. Typ *A* hat keine Namensäquivalenz im neuen Typsystem und besitzt keinen Supertypen. Daher wird der Typ auf null abgebildet. *C* hat ebenfalls keine Namensäquivalenz. Anders als im ersten Beispiel hat hier der Supertyp eine Abbildung auf null. Damit wird Typ *C* auch auf null gemappt.

Nach Verbindung des alten mit dem neuen Typsystem durch die Abbildung können die Daten übertragen werden. Unter Daten werden in diesem Kontext die Instanzen der Typen und deren Feldwerte verstanden. Für die Übertragung der Felddaten müssen die Instanzen der Objekte bereits übertragen sein. Deshalb erfolgt der Datentransfer in zwei Schritten, welche in den folgenden beiden Abschnitten erläutert werden.

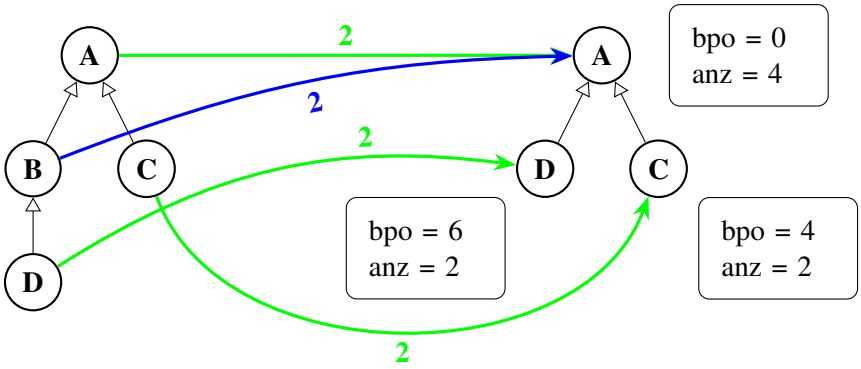

Abbildung 5.7.: Beispielabbildung zwischen Typsystemen 1. grün = Namensäquivalenz, blau = Projektion. Die Zahlen auf den Verbindungspfeilen geben die statischen Objektanzahlen an. In den Kästen sind die Blockinformationen (Base Pool Offset und statische Objektanzahl) zu sehen.

5.6. Übertragung der Objekte

Das Verhalten bei der Übertragung von Objekten hängt stark vom Fall der Abbildungsfunktion aus Gleichung (5.1) ab. Bei Namensäquivalenz erhält der neue Typ alle Objekte des alten Typs. Im Fall einer Projektion erfolgt ein Upcast auf den neuen Typ. Anschließend werden die Objekte diesem Typ hinzugefügt. Falls keine Abbildung existiert (Mapping auf null), gehen die Objekte verloren. Dies entspricht dem Löschen von Objekten.

Aufgrund der möglichen Veränderungen im Typsystem ist ein exaktes Kopieren der Objekte nicht möglich. Die Projektion auf einen Supertyp ist hierfür der Hauptgrund. Während der Übertragung der Objekte muss bei dieser implizit ein Upcast stattfinden. Daher werden anstatt der Objekte selbst Objektinformationen an das neue Typsystem übermittelt. Mit diesen Informationen können die neuen Typen ihre Objekte selbst generieren. Somit besteht die Übertragung aus einer Übermittlung von Objektinformationen und anschließender Instanziierung auf Basis dieser Informationen.

Aus Abschnitt 2.3 geht hervor, dass jedes Objekt eindeutig durch seinen Typ und seine SKilL-ID identifizierbar ist. Dabei sind die IDs nicht vollkommen unabhängig vom Typ. Die SKilL-IDs der Objekte eines bestimmten Typs liegen in einem zusammenhängenden Indexbereich. Dieser Bereich lässt sich durch den Startindex und die Länge darstellen, welche zusammengefasst als Block bezeichnet werden. Im Sprachgebrauch von SKilL wird der Startindex als Base Pool Offset (BPO) bezeichnet. Die Länge des Bereichs ist eine Repräsentation der statischen Objektanzahl. Statisch bedeutet an dieser Stelle, dass die Objekte von Subtypen nicht eingerechnet werden.

Die Übertragung der Objekte auf das neue Typsystem erfolgt durch eine Iteration in Typordnung über die neuen Typen. Schrittweise werden die statischen Objektanzahlen vom alten Typsystem übertragen und währenddessen die BPOs berechnet. Zusammen können durch diese Informationen die Objekte instanziiert werden. Einen Sonderfall bildet die Projektion auf einen Supertypen. In diesem Fall erhält ein Typ des neuen Typsystems mehrere Objektanzahlen. Diese müssen vor der Instanziierung summiert werden. Der resultierende Block besteht aus dem berechneten BPO und der Summe der statischen Objektanzahlen.

Algorithmus 5.1 Berechnung der neuen SKilL-ID

```
procedure CALCULATENEWSKILLID(oldID, oldType)
    newType ← TYPEMAPPING(oldType)
    newID ← oldID − BPO(oldType) + BPO(newType) + PROJECTIONOFFSET(newType)
    return newID
end procedure
```

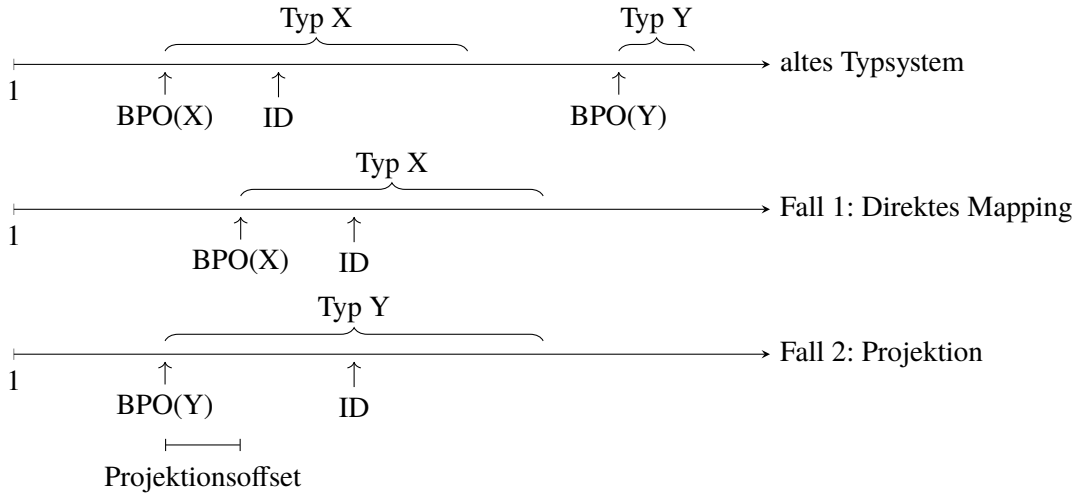

Abbildung 5.8.: Graphische Darstellung der Berechnung von SKilL-Objekten. Oben ist die serielle Darstellung der alten Objekte zu sehen. Darunter ein neues Typsystem in welchem Typ X ein direktes Mapping hat. Der letzte Fall zeigt eine Projektion von Typ X auf Y. Mithilfe des Algorithmus 5.1 lassen sich die neuen IDs berechnen.

Um den Ablauf der Objektübertragung zu verdeutlichen, wird das Beispiel aus Abbildung 5.5 weiter ausgebaut. Es gilt die Annahme, dass jeder Typ zwei Objekte besitzt. Abbildung 5.7 zeigt das Beispiel mit den zusätzlichen Informationen. Die Berechnung des BPO erfolgt in Typordnung durch Summieren der statischen Instanzen der vorherigen Typen. Für jeden Basistyp erfolgt ein Neustart der Berechnung, da der BPO von Basistypen immer null ist.

Berechnung der neuen SKilL-ID Da durch die Objektübertragung eine Verschiebung der IDs möglich ist, müssen die IDs im neuen Typsystem berechnet werden. Eine Berechnung ist mit Algorithmus 5.1 möglich. Dieser beruht darauf, dass die IDs eines Typs vor und nach der Übertragung in zusammenhängenden Indexbereichen liegen. Durch das Abziehen des alten BPO ergibt sich die relative Position der ID im Indexbereich. Das Addieren des neuen BPO ergibt somit die absolute ID im neuen Typsystem. Da eine Projektion möglich ist, kann es zu einer zusätzliche Verschiebung kommen. Diese besteht aus der statischen Anzahl an Objekten des Typs auf welchen die Projektion stattfindet und der Anzahl der Objekte von zuvor projizierten Typen. In der Umsetzung dieses Algorithmus sind die BPOs sowie der Projektionsoffset in Arrays abgelegt, um einen effizienteren Ablauf zu gewährleisten.

Algorithmus 5.2 Grober Aufbau eines Algorithmus zur Feldübertragung

```
procedure TransferFields(oldTypesystem, newTypesystem)
    for all oldType ∈ oldTypesystem do
        FieldMap ← SearchFieldMappings(oldType)                    // Kap. 5.7.1
        for all (oldField, newField) ∈ FieldMap do
            if FieldsCompatible(oldField, newField) then           // Kap. 5.7.2
                TransferValues(oldField, newField, oldType)        // Kap. 5.7.3
            end if
        end for
    end for
end procedure
```

Die Beispiele in Abbildung 5.8 zeigen die Berechnung der SKilL-Objekte graphisch. Oben wird die serielle Darstellung des alten Typsystems gezeigt. Darunter sind zwei neue Typsysteme zu sehen, um das direkte Mapping als auch die Projektion abzudecken. Bei einer direkten Abbildung entfällt der Projektionsoffset. Durch Abziehen des alten und Addieren des neuen BPO erhält man die neue ID. Im Falle einer Projektion, wie in der dritten Darstellung zu sehen, muss der Projektionsoffset hinzugerechnet werden. Dieser entspricht hier exakt der statischen Objektanzahl von Typ Y.

5.7. Übertragung der Felder

Nachdem die Objekte übertragen wurden, müssen alle Feldwerte an das neue Typsystem übergeben werden. Dabei können die Felder in der Typhierarchie verschoben sein und einen anderen Typ haben. Deshalb ist es notwendig zunächst das passende Feld über seinen Namen zu finden und anschließend eine Typprüfung für die Felder durchzuführen. Algorithmus 5.2 zeigt den groben Aufbau der Übertragung. Die folgenden Abschnitte 5.7.1 bis 5.7.3 erläutern die Teilschritte näher. In der tatsächlichen Umsetzung wurden einige Optimierungen durchgeführt. Das hier vorgestellte Prinzip bleibt aber erhalten.

5.7.1. Felder finden

Für die Übertragung der Feldwerte muss zunächst das passende Feld im neuen Typsystem gefunden werden. Passend bedeutet an dieser Stelle ein namensäquivalentes Feld im neuen Typ. Das Vorgehen zur Suche aller Felder wird in Algorithmus 5.3 dargestellt. Über die Abbildung erfolgt der Übergang ins neue Typsystem. Ist kein Mapping vorhanden, wurden im vorherigen Abschnitt 5.6 auch keine Objekte übertragen. Damit müssen auch keine Feldwerte übertragen werden.

Um die mögliche Verschiebung in der Typhierarchie zu gewährleisten, sind die Felder von Supertypen auf Seiten des alten und des neuen Typsystems mit einzubeziehen. Abbildung 5.9 zeigt zwei mögliche Feldverschiebungen. Das Feld *b* des alten Typs *B* ist im neuen Typsystem nach unten in den Typ *D* verschoben. Durch die Suche nach allen Feldern im alten Typsystem ist eine Übertragung der Feldwerte dennoch möglich. Eine Feldverschiebung nach oben ist für das Feld *c* zu sehen. In diesem Fall ist es wichtig im neuen Typsystem alle Supertypen nach Feldern zu durchsuchen.

Algorithmus 5.3 Suchalgorithmus für alle Felder eines alten Typs im neuen Typsystem

```
procedure SearchFieldMappings(oldType)
    FieldMap ← ∅
    newType ← TypeMapping(oldType)
    for all oldField ∈ AllFields(oldType) do          // Including fields of supertypes
        for all newField ∈ AllFields(newType) do      // Including fields of supertypes
            if oldField.name == newField.name then
                FieldMap ← (oldField, newField)
            end if
        end for
    end for
    return FieldMap
end procedure
```

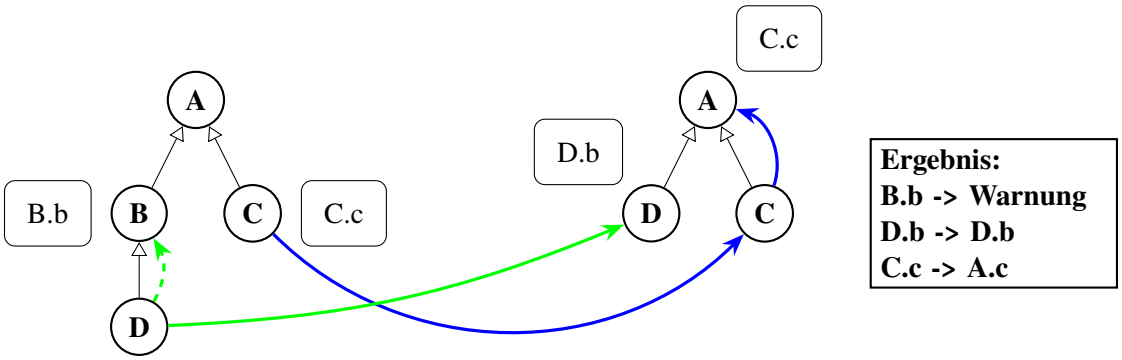

Abbildung 5.9.: Beispiele für Feldsuche. Grün eingezeichnetes Beispiel sucht nach Feld mit Namen b. Blau eingezeichnetes Beispiel sucht nach Feld mit Namen c.

Wird für ein Feld keine passende Abbildung im neuen Typsystem gefunden, so wird am Ende der Projektion eine Warnung ausgegeben. Dennoch ist es dem Anwender erlaubt, Felder aus der Spezifikation zu entfernen.

5.7.2. Typprüfung

Nachdem zwei Felder mit äquivalenten Namen gefunden wurden, müssen die Typen der Felder überprüft werden. Dieser Schritt ist notwendig, da sich der Typ eines Feldes in der Spezifikation des neuen Typsystems ändern lässt. Aus diesem Grund kann es vorkommen, dass die Feldwerte nicht zum Typ des neuen Feldes passen. Zum Beispiel kann ein String nicht in einen Referenzdatentyp übertragen werden.

Die Typprüfung findet in zwei Schritten statt. Zunächst erfolgt eine statische Prüfung, welche die beiden Typen auf ihre grundsätzliche Kompatibilität überprüft. Dabei treten drei mögliche Ergebnisse auf:

- **kompatibel:** Die Typen sind unabhängig von ihren Werten immer kompatibel.

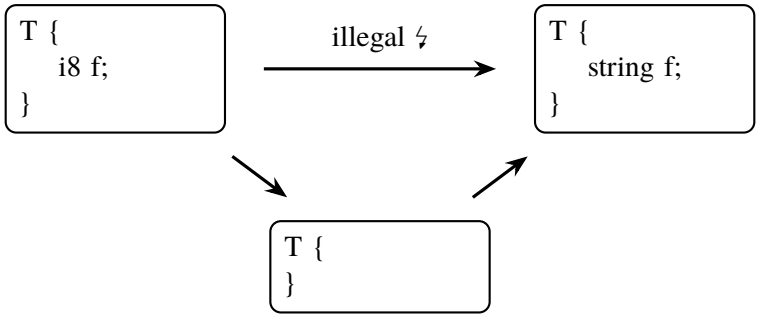

Abbildung 5.10.: Möglichkeit zur indirekten (illegalen) Änderung eines Typs

SKilL	bool	i8	i16	i32	i64	v64	f32	f64	string
Java	boolean	byte	short	int	long	long	float	double	String

Tabelle 5.1.: Repräsentation der SKilL-Grunddatentypen durch Java Typen

- **eventuell kompatibel:** Die Kompatibilität kann statisch nicht herausgefunden werden. Abhängig von den Werten können die Typen kompatibel sein oder nicht.
- **nicht kompatibel:** Die Typen sind unabhängig von ihren Werten in keinem Fall kompatibel.

Der zweite Schritt wird nur benötigt, wenn die Kompatibilität statisch nicht überprüft werden kann. Dann erfolgt eine dynamische Prüfung bei welcher alle Feldwerte auf ihre Kompatibilität mit dem neuen Typ geprüft werden. Sobald ein Wert nicht kompatibel ist, endet die dynamische Prüfung mit dem Ergebnis nicht kompatibel.

Endet die Typprüfung mit dem Resultat nicht kompatibel, so schreibt die Spezifikationsprojektion keine Ausgabedatei. Wie in Abbildung 5.10 zu sehen ist, kann der Anwender dennoch durch zwei legale Operationen sein Ziel über den indirekten Weg erreichen.

Das Konzept wird nun auf die Datentypen von SKilL übertragen. Dabei lassen sich drei Gruppen bilden, welche untereinander nicht kompatibel sind. Die Kompatibilität innerhalb der Gruppen wird in den folgenden Abschnitten näher betrachtet.

Grunddatentypen Die Kompatibilität der Grunddatentypen richtet sich weitestgehend nach den Konversionsregeln für primitive Datentypen von Java [LYBB15] §2.11.4. Jeder der Datentypen hat eine Repräsentation durch einen Typ in Java. Tabelle 5.1 gibt den entsprechenden Java-Typ für alle SKilL-Grunddatentypen an.

Tabelle 5.2 zeigt die Ergebnisse der statischen Typprüfung. Die Datentypen `bool` und `string` sind unabhängig von den Zahltypen. Für Ganz- und Gleitkommazahltypen kann durch Hinzufügen von Bits eine Konversion auf einen größeren Datentyp erfolgen. Die mit Stern markierten Umrechnungen von Ganzzahl- auf Gleitkommatypen führen möglicherweise zu einem Präzisionsverlust, da nicht alle Zahlen des Ganzzahltyps durch den Gleitkommatyp darstellbar sind.

		nach Typ								
		bool	**i8**	**i16**	**i32**	**i64**	**v64**	**f32**	**f64**	**string**
von Typ	**bool**	✓	✗	✗	✗	✗	✗	✗	✗	✗
	i8	✗	✓	✓	✓	✓	✓	✓	✓	✗
	i16	✗	?	✓	✓	✓	✓	✓	✓	✗
	i32	✗	?	?	✓	✓	✓	✓*	✓	✗
	i64	✗	?	?	?	✓	✓	✓*	✓*	✗
	v64	✗	?	?	?	✓	✓	✓*	✓*	✗
	f32	✗	?	?	?	?	?	✓	✓	✗
	f64	✗	?	?	?	?	?	✓*	✓	✗
	string	✗	✗	✗	✗	✗	✗	✗	✗	✓

Tabelle 5.2.: Kompatibilität der Grunddatentypen. ✓ = immer kompatibel. ✗ = nicht kompatibel. **?** = eventuell kompatibel, dynamischer Check erforderlich. * = möglicher Präzisionsverlust.

Die mit Fragezeichen markierten Stellen werden zusätzlich einer dynamischen Prüfung unterzogen. Diese überprüft, ob alle Werte in den Wertebereich des neuen Typs passen. Zudem werden eventuell auftretende Gleitkommazahlen auf not-a-number-Werte überprüft. Die Konvertierung von Gleitkommazahlen auf ganze Zahlen wird wie in [LYBB15] spezifiziert durchgeführt. An dieser Stelle ist der Präzisionsverlust offensichtlich.

Anwendertypen Die Prüfung zweier Anwendertypen ist wesentlich umfangreicher als die Checks für die Grunddatentypen. Sowohl bei der statischen als auch bei der dynamischen Prüfung ist eine Traversierung der Typhierarchie notwendig.

Bei der statischen Typprüfung wird zunächst untersucht, ob der Typ des alten Systems eine Abbildung im neuen System hat. Ist dies der Fall, wird der abgebildete Typ und alle seine Supertypen mit dem neuen Typ verglichen. Bei einem erfolgreichen Vergleich handelt es sich um eine Typgleichheit oder einen Upcast. In beiden Fällen wird kein dynamischer Vergleich benötigt, da die Typen aufgrund ihrer Stellung in der Typhierarchie immer kompatibel sind.

Sind die vorherigen Vergleiche negativ, kann zunächst statisch geprüft werden, ob ein Downcast möglich ist. Diese Prüfung erfolgt nur durch die Betrachtung des Typsystems. Das Ergebnis gibt an, ob ein Downcast überhaupt machbar ist. Bei einer erfolgreichen statischen Prüfung ist nicht garantiert, dass der Downcast realisiert werden kann. Dies muss in der anschließenden dynamischen Prüfung sichergestellt werden. Der statische Downcast-Check bringt in diesem Fall keinen Mehrwert, da der dynamische Check dennoch ausgeführt werden muss. Aus diesem Grund wird die statische Downcast-Prüfung nicht umgesetzt.

Die statische Prüfung hat ein negatives Ergebnis für die Fälle der Inkompatibilität und des Downcast. Um diese beiden Fälle zu unterscheiden, wird die dynamische Prüfung verwendet. In dieser werden alle Objekte auf ihre Kompatibilität mit dem neuen Typ überprüft. Diese Prüfung ist in Algorithmus 5.4 dargestellt und basiert auf einem Verfahren von Schubert et al. [SPT83]. Durch den

Algorithmus 5.4 Prüfung auf möglichen Downcast durch die SKilL-ID des Objekts

```
procedure DowncastCheck(newID, newType)
    bpo ← BPO(newType)
    return bpo < newID ∧ newID <= bpo + Size(newType)
end procedure
```

Algorithmus 5.5 Algorithmus zur Übertragung der Feldwerte

```
procedure TransferValues(oldField, newField, oldType)
    for all oldObject ∈ oldType.staticInstances do
        newObject ← CalculateNewSkillObject(oldObject)
        oldValue ← GetValue(oldField, oldObject)
        SetNewValue(newField, newObject, oldValue)
    end for
end procedure
```

BPO und die dynamische Anzahl der Objekte ist das Intervall der SKilL-IDs eines Typs bestimmt. Liegt die ID zwischen den beiden Grenzen des Intervalls, ist das durch die ID identifizierte Objekt eine Instanz des Typen oder seiner Subtypen.

Container Die Container `list`, `set` und Array sind zueinander kompatibel. Der Datentyp `Map` ist nur zu sich selbst kompatibel. In allen Fällen müssen nach Prüfung der Containertypen anschließend die Basistypen geprüft werden. Diese können hier nur Grunddatentypen und Anwendertypen sein. Deshalb werden für die Prüfung der Basistypen die Methoden aus den vorherigen beiden Abschnitten verwendet. Falls die Prüfung der Basistypen einen dynamischen Check erfordert, muss dieser für jedes Objekt jedes Containers durchgeführt werden. Somit ergibt sich hier eine weitere Iteration über alle Container, bevor der Check des Basistyps angewendet wird.

Der Datentyp Array ist aufgeteilt in feste und variable Länge. Besitzt der neue Typ eine feste Länge, muss diese von allen alten Containern eingehalten werden. Eine statische Prüfung ist an dieser Stelle nur ausreichend, falls der alte Datentyp ebenfalls die gleiche feste Länge aufweist. Andernfalls ist zwingend eine dynamische Prüfung erforderlich, welche die Länge aller alten Container mit der des neuen Arrays fester Länge vergleicht.

5.7.3. Werte übertragen

Nachdem in den vorherigen Abschnitten zwei passende Felder gefunden und deren Typen überprüft wurden, können nun die Werte der Felder übertragen werden. Dazu erfolgt, wie in Algorithmus 5.5 zu sehen, eine Iteration über alle Objekte des übergebenen alten Typs. Für diese wird jeweils ein Objekt des neuen Typsystems berechnet. Anschließend wird der alte Wert abgerufen und in das neue Feld übertragen.

Die Übertragung des alten Wertes in das neue Feld findet abhängig vom Datentype auf unterschiedliche Weise statt. Für die Typen `bool` und `string` kann der Wert einfach kopiert werden. Bei den Zahlentypen wird eine Konversion auf den Zieltyp benötigt. Da die neuen Objekte an anderer Stelle

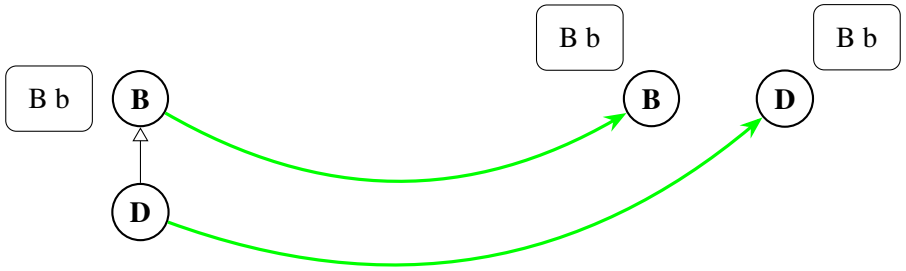

Abbildung 5.11.: Übertragung eines alten Feldes auf zwei neue Felder. Die Hierarchie wird im neuen Typsystem aufgegeben und beide Typen bekommen ein eigenes Feld.

⟨*mapping-file*⟩ ::= ⟨*mapping*⟩*

⟨*mapping*⟩ ::= ⟨*NAME*⟩ (⟨*type-mapping*⟩ | ⟨*field-mapping*⟩)

⟨*type-mapping*⟩ ::= '->' ⟨*NAME*⟩ ⟨*field-mapping*⟩?

⟨*field-mapping*⟩ ::= '{' (⟨*NAME*⟩ '->' ⟨*NAME*⟩)* '}'

Abbildung 5.12.: Grammatik der Mappingdatei in EBNF

sein können als im alten Typsystem, müssen die Referenzen durch die `CalculateNewSkillObject` Funktion neu berechnet werden. Ist der Zieltyp ein Container muss dieser komplett neu instanziiert werden und die Daten der Basistypen auf die eben genannten Arten übergeben werden.

Im Algorithmus ist eine zusätzliche Übergabe des alten Typs notwendig. Der Grund dafür ist, dass es für ein altes Feld mehrere neue Felder geben kann. Ein Beispiel ist in Abbildung 5.11 dargestellt. Im alten Typsystem beinhaltet Typ *B* das Feld *b*. Durch die Vererbung gibt es dieses Feld auch für Objekte des Typs *D*. Das neue Typsystem gibt die alte Hierarchie auf und gibt jedem Typ ein eigenes Feld *b*. Die Übertragung muss also jeweils nur für alle statischen Instanzen des alten Typs stattfinden.

5.8. Erweiterung: Mappingdatei

Mit der bisherigen Umsetzung ist eine Abbildung nur durch Namensäquivalenz möglich. In diesem Abschnitt wird die Umsetzung um eine Mappingdatei erweitert. In dieser können die Namen des alten Typsystems auf andere Namen abgebildet werden. Die Grammatik der Datei ist angelehnt an [Wei16] §2.3 und ist in Abbildung 5.12 zu sehen. Das Token *NAME* entspricht dem von Felden [Fel17b] §2 verwendeten Token *id*. Der Einfachheit halber kann hier aber von einfachen Identifiern wie zum Beispiel in Java ausgegangen werden.

Mit der dargestellten Grammatik lassen sich drei Arten von Mappings beschreiben. Diese sind auch in der gleichen Datei mischbar. Listing 5.3 zeigt eine Beispieldatei, welche alle drei Arten beinhaltet. Das erste Mapping bildet nur den Typnamen *A* auf *B* ab. Eine reine Abbildung der Feldnamen ist

Listing 5.3 Beispiel für eine Mappingdatei

```
1  A -> B
2
3  C {
4          myField -> anotherField
5  }
6
7  D -> E {
8          field1 -> field3
9          field2 -> field4
10 }
```

im zweiten Mapping zu sehen. Der alte Typname muss hier immer mit angegeben werden, um die Felder eindeutig zuordnen zu können. Im dritten Fall werden die ersten beiden Mappings kombiniert. Sowohl der Typ- als auch der Feldname werden auf neue Namen abgebildet.

Der alte Typname ist für alle Mappingarten zwingend erforderlich, da er zur Identifikation dient. Die Abbildungen eines bestimmten Typs werden in einer Datenstruktur gespeichert. Diese kann einen neuen Typnamen sowie beliebig viele Abbildungen von Feldnamen enthalten. Die Datenstruktur selbst wird wiederum in einer Map abgelegt und kann durch den alten Typnamen erreicht werden. Bei mehrfacher Nennung des gleichen alten Typnamens in der Mappindatei wird das zuletzt genannte Mapping gespeichert und eine Warnung ausgegeben.

Die Integration der Mappings in die Spezifikationsprojektion gestaltet sich einfach. Zunächst wird in der Abbildungsfunktion in Gleichung (5.1) die Funktion *name(alterTyp)* angepasst und lässt sich folgendermaßen beschreiben:

$$name(alterTyp) = \begin{cases} mapping(alterTyp) & \text{falls ein Mapping für } alterTyp \text{ existiert} \\ alterTyp.name & \text{sonst} \end{cases}$$

Somit wird nach dem in der Mappingdatei angegebenen Namen gesucht. Andererseits ist die Funktion abwärtskompatibel zur vorherigen Implementierung. Das heißt, falls kein Mapping existiert, wird der Name des alten Typs übernommen. Zudem müssen in der Mappingdatei nicht für alle Typen Abbildungen angegeben werden, sondern nur notwendige Mappings. Die Abbildung für Feldnamen lässt sich ähnlich leicht integrieren. In Algorithmus 5.3 ist die Suche nach passenden Feldern beschrieben. Hier muss lediglich die Zeile getauscht werden, welche den Namensvergleich vornimmt. Anstatt *oldField.name* wird an diese Stelle die Abbildung des Feldnamens gesetzt. Wie bei den Typnamen wird bei fehlendem Mapping der alte Feldname verwendet.

6. Prüfen und Ändern von Restrictions

Dieser Abschnitt beschreibt die Umsetzung einer Methode, um Restrictions von einzelnen Typen und Feldern zu verändern. Zudem müssen die Restrictions geprüft werden, um valide Binärdateien garantieren zu können.

6.1. Implementierung der fehlenden Restrictions

Um die Aufgaben der Prüfung und Änderung von Restrictions bearbeiten zu können, müssen zunächst die fehlenden Restrictions in das Java-API implementiert werden. Bisher sind die beiden Feld-Restrictions `@range` und `@nonnull` vorhanden. Somit müssen nach Felden [Fel17b] §5.1 die Feld-Restrictions `@default`, `@coding`, `@constantLengthPointer` und `@oneOf` umgesetzt werden. Ebenso fehlt die Implementierung aller Typ-Restrictions. Im Detail umfasst dies die Restrictions `@unique`, `@singleton`, `@monotone`, `@abstract` und `@default`.

Die Implementierung der Restrictions erfordert zunächst die Integration in den bestehenden Serialisierungs- und Deserialisierungsprozess. Felden [Fel17b] §5.1 enthält die Regeln zur Serialisierung aller Restrictions. Diese werden verwendet, um sowohl die Serialisierung als auch die Deserialisierung zu implementieren.

Jede Restriction muss eine `check`-Methode implementieren, in welcher die Prüfung stattfindet. Für Typ-Restrictions wird ein Typ als Parameter übergeben. Hier kann zum Beispiel geprüft werden, ob der Typ ein Singleton ist, indem die Anzahl der Instanzen überprüft wird. Die Feld-Restrictions prüfen den Feldwert aller Objekte einzeln. Hier kann etwa sichergestellt werden, dass ein Zahlentyp einen bestimmten Zahlenbereich nicht überschreitet. Die Vorgaben für die Umsetzung der `check`-Methode werden ebenfalls durch Felden [Fel17b] §5.1 beschrieben.

6.2. Prüfen von Restrictions

In der bisherigen Implementierung des Java-API werden nur die Feld-Restrictions geprüft, da keine Typ-Restrictions vorhanden waren. In diesem Abschnitt wird die Umsetzung der Prüfung erweitert, um alle Restrictions abzudecken.

Die Prüfung aller Restrictions erfordert ein Durchlaufen aller Typen und Felder. Algorithmus 6.1 zeigt die komplette Prüfung. Im API wird diese Prüfung aus Effizienzgründen aufgeteilt. Zunächst werden die Typ-Restrictions geprüft. Anschließend erfolgen die Checks für jede Restriction in jedem Feld des Typs. Dabei müssen jeweils alle Feldwerte einzeln überprüft werden. Bei Fehlschlagen eines Checks wird eine Exception geworfen. Diese gibt Informationen über die Art der Restriction und den Typ oder das Feld, welches gegen die Restriction verstößt.

Algorithmus 6.1 Prüfung aller Restrictions

```
procedure CHECKRESTRICTIONS()
    for all t ∈ types do
        for all typeRest ∈ RESTRICTIONS(t) do
            typeRest.CHECK(t)
        end for
        for all f ∈ FIELDS(t) do                    // Only fields owned by t
            for all fieldRest ∈ RESTRICTIONS(f) do
                for all obj ∈ OBJECT(t) do
                    fValue ← GETVALUE(f, obj)
                    fieldRest.CHECK(fValue)
                end for
            end for
        end for
    end for
end procedure
```

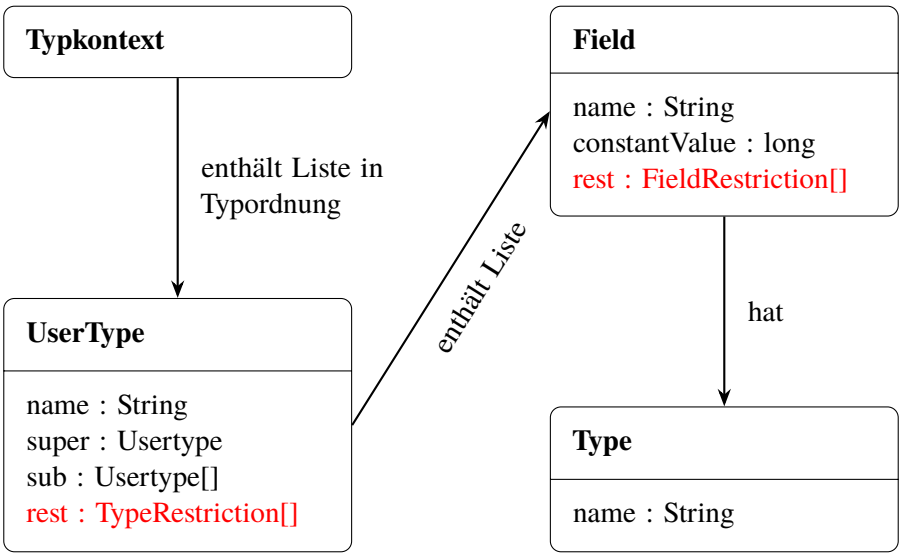

Abbildung 6.1.: Typkontext der Spezifikation mit Restrictions

6.3. Änderung der Restrictions

Restrictions werden in der Spezifikation angegeben. Deshalb ist es an dieser Stelle sinnvoll, die Änderung der Restrictions in die Spezifikationsprojektion zu integrieren. Für die Integration ist die in Abschnitt 5.4 beschriebene Erzeugung des Typsystems eine geeignete Stelle. Der Typkontext enthält nach dem Parsen bereits eine interne Darstellung der Restrictions. Abbildung 6.1 zeigt die Erweiterung des Typkontexts in rot.

Im Zuge der Erzeugung des Typsystems werden die Restrictions in die Typen und Felder generiert. Am Ende der Projektion wird automatisch eine Prüfung aller Restrictions durchgeführt, um eine valide Datei zu garantieren. Dies ist wichtig, da eine Änderung der Restrictions im Zusammenhang

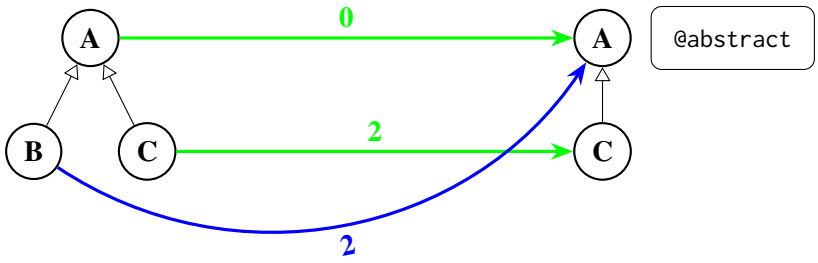

Abbildung 6.2.: Fehler bei Änderung einer Restriction im Zusammenhang mit einer Projektion. Über den Projektionspfeilen ist die Anzahl der übertragenen Objekte zu sehen. Die `@abstract`-Restriction wird eine Exception werfen.

mit einer Projektion leicht zu Fehlern führen kann. Abbildung 6.2 zeigt ein einfaches Beispiel für diese Art von Fehlern. Typ *A* hat keine Objekte und soll im neuen Typsystem eine `@abstract`-Restriction bekommen. Gleichzeitig wird Typ *B* auf seinen Supertyp projiziert. Bei Prüfung der Restrictions wird eine Exception geworfen, da Typ *A* aufgrund der `@abstract`-Restriction keine Objekte enthalten darf.

7. Normalisierung mit einer Ordnungsfunktion

Eine Ordnungsfunktion bildet eine beliebige Menge auf eine geordnete Menge ab. Die Ordnung wird dabei durch eine Ordnungsrelation bestimmt. Ein einfaches Beispiel ist die „kleiner gleich“-Beziehung auf den Zahlenmengen. Im Bereich von SKilL kann eine Ordnungsrelation zum Beispiel zur Prüfung der strukturellen Äquivalenz verwendet werden.

Abbildung 7.1 zeigt zwei strukturell äquivalente Graphen. Diese Eigenschaft zweier Graphen wird als Graphenisomorphie bezeichnet. Für die Prüfung auf Isomorphie ist kein Algorithmus in polynomialer Zeit bekannt. Der beste bekannte Algorithmus hat eine quasipolynomiale Laufzeit [Bab16]. Eine Vereinfachung ist möglich, falls eine Ordnungsrelation für die Knotenmenge vorhanden ist. Mithilfe der geordneten Menge kann jedem Knoten ein neuer Index vergeben werden. Anschließend reicht eine Prüfung der Abbildung gleicher Indizes aufeinander aus.

7.1. Übertragung auf SKilL

In SKilL sind die Knoten des Graphen durch Objekte definiert. Die Identifikation der Knoten erfolgt durch Typ- und SKilL-ID. Eine Normalisierung mit einer Ordnungsfunktion gleicht in diesem Fall der Neuvergabe der SKilL-IDs. Durch die Typordnung ist der Indexbereich für jeden Typ bereits vorgegeben. Deshalb ist die Anwendung einer Ordnungsrelation nur innerhalb eines Typs möglich.

Alle statischen Objekte eines Typs bilden die Menge, auf welcher die Relation angewendet wird. Somit muss die Relation zwei Objekte des Typs vergleichen können. Die einfachste Möglichkeit ist der Vergleich eines festen Feldes. Ein Beispiel ist ein Integer-Feld, welches eine ID repräsentiert. Hierauf kann eine kleiner-gleich-Beziehung definiert werden. Es sind auch andere Möglichkeiten als Feldvergleiche denkbar, jedoch werden diese dem Anwender überlassen.

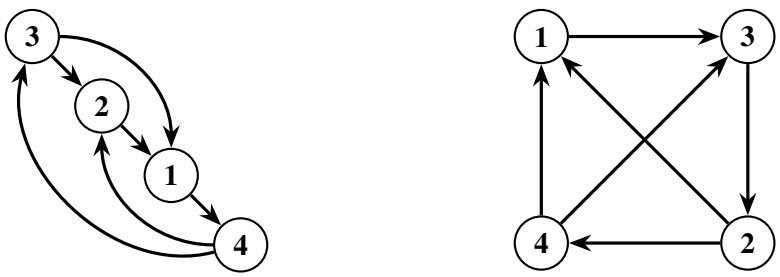

Abbildung 7.1.: Beispiel für strukturelle Äquivalenz

Listing 7.1 Abstrakte Klasse zur Definition von Relationen in Java

```
1  public abstract class SkillObjectComparator implements Comparator<SkillObject> {
2          @Override
3          public abstract int compare(SkillObject o1, SkillObject o2);
4  }
```

Algorithmus 7.1 Algorithmus zur Sortierung der Objekte mit einer Ordnungsrelation

procedure SORTOBJECTS(*comparator*)
  for all $t \in types$ **do**
    SORT($t.staticInstances$, *comparator*)
  end for
end procedure

Aufgrund der typbezogenen Anwendung der Ordnungsrelation müssen zur Neusortierung aller Objekte mehrere Vergleichsoperationen angegeben werden. Dabei ist eine Ordnungsrelation auf einen Typ und alle seine Subtypen anwendbar. Somit ist eine Definition einer Relation für jeden Typ nicht notwendig. Jeder Typ verwendet die erste Ordnungsrelation, welche bei der Traversierung der Supertypen gefunden wird.

7.2. Umsetzung

Die Umsetzung der Normalisierung mit einer Ordnungsfunktion besteht aus zwei Teilen. In einem Teil muss dem Anwender die Möglichkeit gegeben werden, eine eigene Ordnungsrelation zu definieren. Der zweite Teil bekommt die Relation übergeben und wendet sie auf die entsprechenden Objekte an.

Um eine Relation zu definieren, muss der Anwender lediglich von einer abstrakten Klasse erben und die darin enthaltene Vergleichsmethode implementieren. Die Struktur der Klasse ist in Listing 7.1 zu sehen. Das generische Interface `Comparator` ist im Java-API definiert [Ora15d]. Ein Beispiel für die Implementierung einer Relation wird in Abschnitt 8.6 gezeigt.

Die Neusortierung der SKilL-IDs erfolgt durch Algorithmus 7.1. Es reicht aus die Objekte im Typ neu anzuordnen. Während des Schreibvorgangs werden die IDs automatisch neu vergeben. Die `Sort`-Funktion stammt aus der Java-Klasse Arrays [Ora15b]. Diese sortiert die statischen Instanzen des Typs unter Einbeziehung der übergebenen Relation.

Eine Erweiterung des Algorithmus ist die Unterstützung von Typ-Relation-Zuordnungen. Damit ist es möglich für jeden Typ eine eigene Relation anzugeben. Ist dem Typ selbst keine Relation zugeordnet, wird in der Typhierarchie nach oben gesucht. Hierbei wird die erste gefundene Relation verwendet. Wird keine Relation für einen Typ gefunden, erfolgt keine Neusortierung der IDs dieses Typs.

8. Evaluation

Alle in dieser Arbeit entwickelten Methoden müssen auf ihre Funktionsfähigkeit geprüft werden. Dies wird durch geeignete Tests gezeigt. Zudem soll die Garbage Collection sowie das Entfernen von Typen und Feldern effizient ablaufen. Um dies sicherzustellen, werden Messungen der Performance für diese Methoden verwendet.

8.1. Allgemeiner Testaufbau

Die Methoden in dieser Arbeit funktionieren alle nach dem gleichen Prinzip. Zunächst wird eine Binärdatei eingelesen, dann erfolgt eine Verarbeitung und anschließend die Ausgabe in eine (neue) Binärdatei. Aus diesem Grund ist der allgemeine Aufbau der Funktions- und Performance Tests gleich. Außerdem wird die Testumgebung für die Performance-Tests vorgestellt.

8.1.1. Aufbau der Funktionstests

Um die Funktionsfähigkeit der Umsetzungen zu garantieren werden Funktionstests eingesetzt. Da die Implementierungen alle in der Programmiersprache Java geschrieben sind, wird für die Tests das JUnit Framework verwendet [The18]. Für alle Lese- und Schreiboperationen wird die modifizierte Version des Java-API verwendet. Die Unit-Tests arbeiten alle nach dem gleichen Schema, welches die folgenden Schritte beinhaltet:

1. Einlesen des SKilL-Graphen und Typsystems aus der Binärdatei
2. Ausführung der zu testenden Aktion
3. Ausgabe des Ergebnisses in einer temporären Datei
4. Erneutes Einlesen der temporären Datei
5. Abgleich mit einer eingelesenen Erwartungsdatei

Die ersten drei Schritte bilden den normalen Ablauf der zu testenden Operation ab. Der Unterschied besteht im Schreiben einer temporären Datei anstatt die Eingabedatei zu ersetzen oder eine neue Datei zu erstellen. Mit dem erneuten Einlesen der Ausgabedatei wird überprüft, ob die Binärdatei valide ist. Invalide Dateien ergeben eine Exception im Einleseprozess, welche den Test fehlschlagen lässt.

Für jeden Test wird eine Erwartungsdatei benötigt, welche mit der vom Test generierten Datei abgeglichen wird. Unterschiede in den Dateien werden vom Test bemerkt und an den Anwender weitergegeben. Diese helfen bei der Identifikation möglicher Fehler. Alle Erwartungsdateien wurden durch ein Darstellungstool für SKilL-Graphen [Rat17] aufgerufen. Dieses benutzt die unmodifizierte

Version der API und stellt somit sicher, dass die Dateien auch mit der ursprünglichen API-Version lesbar sind. Zudem kann ein Teil der Erwartungsdateien durch dieses Tool manuell überprüft werden. Die Prüfung dieser Dateien ist zuletzt am 02.10.2018 erfolgt.

Der Abgleich mit einer Erwartungsdatei kann nicht auf Bytestromebene stattfinden, da es Freiheitsgrade in der Kodierung der Binärdatei gibt. Diese werden vor allem genutzt, um einen schnelleren Schreibprozess zu ermöglichen [Fel17a] §7.3. Ein Beispiel ist die Anordnung von Werten oder Referenzen in einem Set. Deshalb muss die Datei für den Vergleich ebenfalls in den Hauptspeicher eingelesen werden. Anschließend erfolgt ein struktureller Vergleich der Dateien durch eine speziell hierfür entwickelte Funktion.

8.1.2. Aufbau der Performancetests

Die Analyse der Performance erfolgt durch Messung der Ausführungszeit. In dieser Arbeit wird der Speicherverbrauch vernachlässigt, da der größte Teil des Verbrauchs durch die Hauptspeicherrepräsentation der Binärdateien verursacht wird. Die Implementierung der Messungen erfolgt durch ein Shell-Skript. Dieses ruft die Transformationsoperation auf, welche in einem `.jar`-Archive enthalten ist.

Die Messungen sollen nah an der realen Nutzung sein, um aussagekräftige Ergebnisse zu erhalten. Deshalb wird für jede Messung folgendes Schema verwendet:

1. Starten der Zeitmessung im Shell-Skript
2. Einlesen der Binärdatei
3. Starten der Zeitmessung im `.jar`-Archiv
4. Ausführung der Methode
5. Ende der Zeitmessung im `.jar`-Archiv
6. Schreiben der Binärdatei
7. Ende der Zeitmessung im Shell-Skript

Im Schema sind eine innere und äußere Zeitmessung enthalten. Die innere Messung repräsentiert die Ausführungszeit der Methode. Mit der äußeren Messung lässt sich die Gesamtzeit des Vorgangs inklusive Eingabe- und Ausgabeoperation bestimmen. Somit lässt sich auch bestimmen, welchen Zeitanteil die Transformationsoperation von der Gesamtzeit verwendet.

Das genannte Schema wird in den einzelnen Tests auf mehrere Testdateien angewendet. Ferner können die Messungen pro Datei mehr als einmal erfolgen. Die Ergebnisse werden in eine `.csv`-Datei geschrieben zu werden. Für jede Ausführung ist dort die Eingabedatei, die Anzahl Objekte und die beiden Zeitmessungen hinterlegt.

Komponente	Beschreibung
CPU	Intel Core i7-6700HQ (8 x 2.60GHz)
RAM	16 GB DDR4 @ 2133 MHz
Disk	169 MB/s Samsung SpinPoint M9T 2TB
OS	Ubuntu 16.10
JDK Version	1.8.0_181

Tabelle 8.1.: Testsystem

8.1.3. Testumgebung

Alle Tests wurden auf demselben System durchgeführt, um vergleichbare Ergebnisse zu erhalten. Tabelle 8.1 zeigt die Daten des Systems. Die wichtigsten Daten sind CPU-Geschwindigkeit sowie Geschwindigkeit und Größe des Hauptspeichers. Zudem ist die Lese- und Schreibgeschwindigkeit der Festplatte ein Faktor für die Laufzeit der Ein- und Ausgabe der Binärdateien.

8.2. Garbage Collection

Die Evaluation der implementierten Garbage Collection erfolgt in zwei Schritten. Im ersten Schritt wird die Funktionsfähigkeit der Umsetzung sichergestellt. Dies geschieht durch eine Reihe von Tests. Im Anschluss erfolgt eine Betrachtung der Performance und deren Faktoren um die Effizienz sicherzustellen.

8.2.1. Funktionstests

Durch verschiedene Testdateien wird die komplette Umsetzung der Garbage Collection getestet. Im Detail sind das folgende Bestandteile der Implementierung:

- Eingabe der Wurzelmenge
- Markierungsdatenstruktur
- Markierungsalgorithmus
- Entfernen der Knoten
- Verkleinerung des Typsystems
- Entfernen der Strings

Zunächst wird die Eingabe der Wurzelmenge, die Datenstruktur zur Markierung und das Entfernen der Objekte getestet. Dazu wird eine Datei verwendet, welche zwei Knoten und keine Kanten besitzt. Auf diesem Graph wird die Garbage Collection mit den vier möglichen Wurzelmengen

beide Knoten, *Knoten 1*, *Knoten 2* und *leer* durchgeführt. Die Ergebnisdateien werden manuell durch ein Darstellungstool für SKilL-Graphen überprüft. Bei Korrektheit der Dateien können diese als Erwartungsdateien übernommen werden.

Um den Markierungsalgorithmus zu testen, wird ein größerer SKilL-Graph benötigt. Hier kann das Ergebnis nicht mehr manuell überprüft werden. Jedoch wurde die Ausgabedatei mit dem Ergebnis eines anderen Garbage Collectors [Fel18b] verglichen. Im Fall dieser Datei sind alle Knoten von einem einzigen Wurzelknoten aus erreichbar. Somit ist der eingegebene Graph identisch zum ausgegebenen SKilL-Graph. Im Graph sind fast alle möglichen Typen von Kanten vorhanden. Das heißt sowohl Referenztypen als auch Container mit Referenzen. Somit wird ein Großteil des Markierungsalgorithmus getestet. Die übrigen Sonderfälle des Algorithmus werden durch ein Testabdeckungstool gefunden. Diese Fälle können durch speziell angefertigte Testdateien geprüft werden. In diesen Spezialfällen ist eine manuelle Prüfung der Ergebnisdateien durch ein Darstellungstool möglich.

Die Erweiterung der Garbage Collection durch eine Verkleinerung des Typsystems und des Entfernens von Strings erfordert weitere Tests. Der einfachste Test ist die Garbage Collection für die bisherigen Dateien mit einer leere Wurzelmenge durchzuführen. In Folge werden alle Objekte entfernt. Aus diesem Grund wird kein Typsystem und damit auch kein String benötigt. Das Ergebnis ist eine leere Binärdatei[1]. Ebenso können die Tests für den Markierungsalgorithmus erweitert werden. Bei diesen wird häufig nicht das gesamte Typsystem benötigt und kann entfernt werden. Die Ergebnisdatei muss den kompletten SKilL-Graph enthalten, jedoch ohne überflüssige Typinformationen. Durch manuelle Überprüfung wird festgestellt, ob das Typsystem verkleinert wurde. Ein zusätzlicher Durchlauf der Garbage Collection zeigt, dass immer noch alle Knoten und Kanten vorhanden sind. Hier ist die Ergebnisdatei gleich der Eingabe.

8.2.2. Betrachtung der Performance

SKilL-Graphen können eine sehr große Anzahl an Knoten enthalten. Aus diesem Grund muss die Garbage Collection sehr effizient arbeiten. Für die Analyse der Performance stehen 60 Testdateien zur Verfügung mit einer Knotenanzahl zwischen 732 und knapp 20 Millionen [Fel17a] §7.1.5.

Die Garbage Collection besitzt zwei Extremfälle: alle Knoten erreichbar oder kein Knoten erreichbar. Für beide Fälle wurde die Ausführungszeit für jede Testdatei hundertmal gemessen. Das Ergebnis der Messungen ohne Ein- und Ausgabezeit ist in Abbildung 8.1 als doppelt logarithmisches Diagramm dargestellt. Anschließend erfolgt für die beiden Messreihen eine Regressionsanalyse. Aufgrund der Lage der Messwerte wird ein lineares Regressionsmodell gewählt. Das Ergebnis der Analyse ist in Tabelle 8.2 dargestellt.

Zunächst fällt die negative Konstante des oberen Extremfalls auf. Im Bereich der Messungen verhält sich die Ausführungszeit nahezu linear zur Graphgröße. Aufgrund der negativen Konstante kann sich die Zeit jedoch für kleinere Graphen nicht linear verhalten.

[1] Eine leere Binärdatei besteht hier aus zwei Nullbytes.

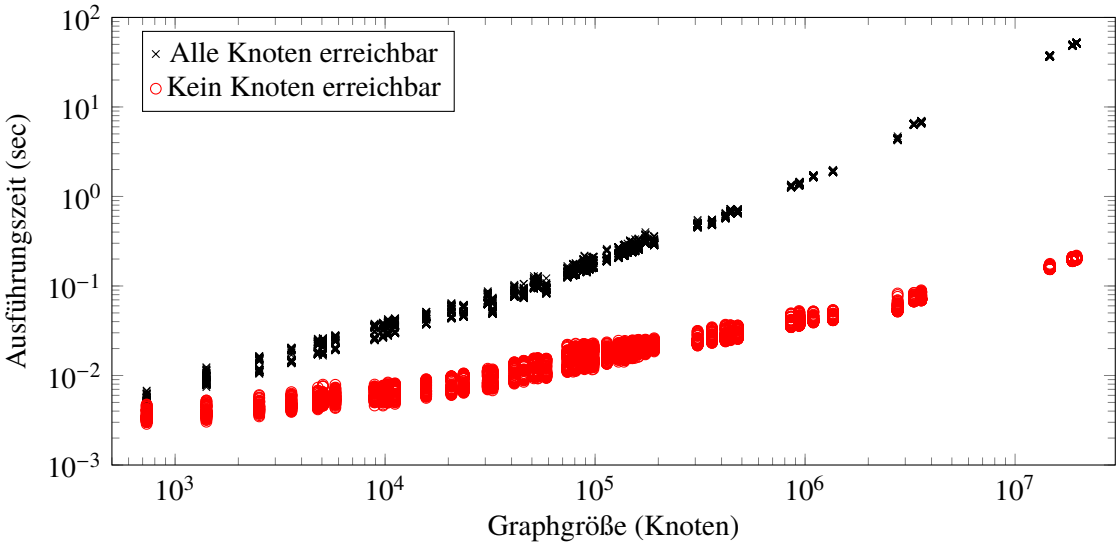

Abbildung 8.1.: Ausführungszeit der Garbage Collection ohne Ein- und Ausgabe für Graphen verschiedener Größe mit 14 GB Heap. Abgebildet sind Messungen für die Extremfälle „alle Knoten erreichbar“ und „kein Knoten erreichbar“.

	Alle Knoten erreichbar	Kein Knoten erreichbar
10^6 Knoten	2.588^{***}	0.010^{***}
	(0.002)	(0.00003)
Constant	-0.341^{***}	0.016^{***}
	(0.010)	(0.0001)
Observations	6,000	6,000
R^2	0.997	0.961
Note:		*p<0.1; **p<0.05; ***p<0.01

Tabelle 8.2.: Lineare Regression über die Ausführungszeiten der Garbage Collection ohne Ein- und Ausgabe in Abhängigkeit der Knotenzahl. Konstanter Anteil in Sekunden und Steigung in Sekunden pro 10^6 Knoten. Tabelle wurde mit der R-Bibliothek *stargazer* [Hla18] generiert.

Falls alle Knoten erreichbar sind, benötigt die Garbage Collection etwa 2.6 Sekunden pro eine Million Knoten. Somit können auch die größten Testdateien mit etwa 20 Millionen Knoten in unter einer Minute bearbeitet werden. Erwartungsgemäß steigen die Zeiten der zweiten Messreihe kaum an, da hier die Markierungsphase aufgrund fehlender Wurzelknoten übersprungen wird.

8.3. Entfernen von Typen und Feldern

Folgend wird mittels Tests die Funktionsfähigkeit der Typ- und Feldentfernung gezeigt. Anschließend erfolgt eine Betrachtung der Performance. Erwartet wird eine wesentlich kleinere Laufzeit als bei der Garbage Collection.

8.3.1. Funktionstests

Für das Entfernen von Feldern oder ganzer Typen werden jeweils separate Tests entwickelt. Die Feldentfernung bildet einen Arbeitsschritt der Typentfernung und wird deshalb zuerst getestet. Anschließend erfolgt die Entwicklung von Tests für das wesentlich komplexere Entfernen von Typen.

Felder entfernen Da das Entfernen von Feldern nicht sehr komplex ist, reichen hier wenige Testfälle aus. Den wichtigsten Teil bildet die Erneuerung der Indizes. Um hier nahezu alle Fälle abzudecken, werden Testfälle mit Feldern am Anfang, am Ende und in der Mitte eines Typs gewählt. Außerdem gibt es den Sonderfall mit nur einem Feld.

Die Erwartungsdateien werden manuell durch ein Darstellungstool überprüft. Dies ist auch bei Dateien mit sehr vielen Objekten möglich, da nur das Typsystem geprüft werden muss. Das erneute Einlesen garantiert die Lesbarkeit der Datei. Damit wird eine Beschädigung der Datei durch die Feldentfernung überprüft.

Typen entfernen Die Typentfernung besteht aus einigen Teilschritten mehr als die Feldentfernung. Dennoch kann die Umsetzung durch wenige Tests komplett getestet werden. Ähnlich wie bei den Tests für die Feldentfernung werden für verschiedene Testdateien Typen an verschiedenen Stellen der Typhierarchie entfernt. Bedingt durch die zusätzlichen Schritte des Entfernens fällt die manuelle Prüfung der Erwartungsdateien wesentlich komplexer aus. Zusammengefasst bilden die Trennung der Abhängigkeiten auf den zu entfernenden Typ und seine Subtypen sowie die Neugenerierung der Typ-IDs und Referenzen den Kern der Tests.

Um die Erwartungsdateien manuell zu überprüfen wird erneut ein Darstellungstool eingesetzt. Die Prüfung besteht aus mehreren Schritten. Zunächst muss sichergestellt werden, dass die Typen wirklich entfernt wurden. Zudem ist zu prüfen, ob die übrige Typhierarchie gleich geblieben ist und alle Referenzfelder auf die entfernten Typen gelöscht wurden. Diese Prüfungen stellen in Verbindung mit dem erneuten Einlesen die Funktionsfähigkeit aller Teilschritte sicher.

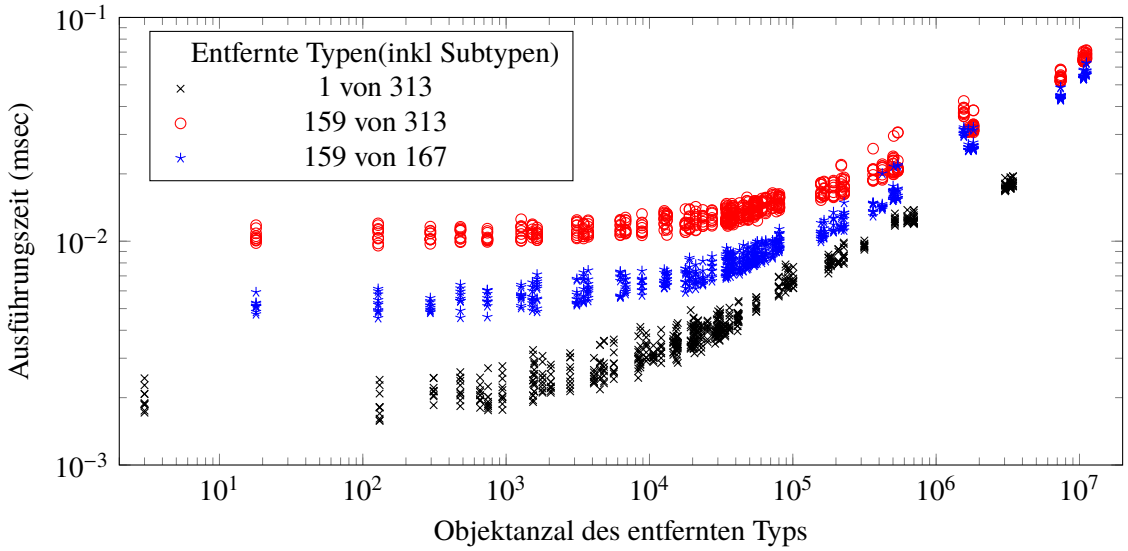

Abbildung 8.2.: Ausführungszeit der Typentfernung ohne Ein- und Ausgabe in Abhängigkeit der Anzahl der entfernten Objekte des Typs. Die Messreihen unterscheiden sich in der Anzahl der entfernten Typen und der Gesamtanzahl der Typen.

8.3.2. Performance

Beide Entfernungsoperationen sollen effizient ablaufen. Aus diesem Grund beschäftigt sich dieser Abschnitt mit der Performance der Operationen. Die beiden Entfernungsoperationen werden im Folgenden getrennt betrachtet, da sie von unterschiedlichen Faktoren abhängen.

Feldentfernung Die Performance der Feldentfernung hängt nur von der Anzahl der Felder im zugehörigen Typ ab. Dabei werden die Felder von Supertypen nicht mitgezählt. Um ein Feld zu entfernen, wird lediglich ein Durchlauf durch die Feldliste benötigt. In diesem erfolgt sowohl die Feldentfernung als auch die Anpassung der Indizes der nachfolgenden Felder.

Eine Analyse der zur Verfügung stehenden Testdateien ergab ein Maximum von 24 Feldern in einem Typ. Für diese Anzahl benötigt die Feldentfernung weniger als eine Millisekunde. Die Feldanzahl könnte noch wesentlich höher sein und würde dennoch kaum Zeit in Anspruch nehmen. Somit ist ein effizientes Entfernen von Feldern sichergestellt.

Typentfernung Bei der Typentfernung spielt die Größe des Typsystems sowie die Anzahl der Objekte des zu entfernenden Typs eine wichtige Rolle. Es erfolgen drei Iterationen über alle Typen, welche zum Entfernen der Felder sowie zur Neugenerierung von Typ-IDs und Referenzen benötigt werden.

Abbildung 8.2 zeigt die Ausführungszeit der Typentfernung in Abhängigkeit der Objektanzahl des Typs. Den beiden Messreihen mit der gleichen Typsystemgröße ist zu entnehmen, dass die Anzahl der entfernten Typen einen deutlichen Einfluss hat.

Ein weiterer Faktor für die Performance der Typentfernung bildet die Gesamtgröße des Typsystems. Alle zur Verfügung stehenden Testdateien haben ein identisches Typsystem mit einer Gesamtzahl von 313 Typen. Um den Einfluss der Typsystemgröße zu messen, wird etwa die Hälfte des Typsystems entfernt. Das kleinere Typsystem hat damit 167 Typen. In Abbildung 8.2 ist die Messung in blau zu sehen.

Ein offensichtlicher Faktor für die Performance ist die Objektanzahl des entfernten Typs, da die Objekte alle einzeln gelöscht werden müssen. Die Ausführungszeit für 10 Millionen Objekte des entfernten Typs liegt bei etwa 70 ms. Bei dieser Objektanzahl ergibt sich durch Hinzunahme der Lese- und Schreiboperation in die Messung eine Zeit von etwa 28 s. Der Zeitanteil der Typentfernung selbst liegt somit unter einem Prozent. Damit ist eine effiziente Umsetzung der Typentfernung gewährleistet.

8.4. Spezifikationsprojektion

Die Spezifikationsprojektion stellt den größten Teil dieser Arbeit dar. Aufgrund der Komplexität der Aufgabe ist eine große Testabdeckung wichtig. Die Tests werden in drei aufeinander aufbauende Gruppen geteilt. In der ersten Gruppe wird der Aufbau des neuen Typsystems, sowie die Abbildung vom alten auf das neuen Typsystem getestet. Darauf folgend werden in der zweiten Testgruppe Objekte übertragen und die Neuberechnung der IDs im neuen Typsystem geprüft. Den größten Teil enthält die dritte Gruppe mit der Übertragung der Felder. Ein Fehler in einer Testgruppe wirkt sich in den meisten Fällen auch auf die nachfolgenden Gruppen aus. Zum Beispiel führt ein Fehler beim Aufbau der Abbildung unweigerlich zur fehlerhaften Übertragung von Objekten und Feldern.

8.4.1. Typsystem

Der Aufbau des neuen Typsystems wird durch einige Spezifikationen getestet. Dabei reichen kleine Typhierarchien aus, um die Grundfunktionalität sicherzustellen. Die verwendeten Hierarchien sind in Abbildung 8.3 dargestellt. Jede der Typhierarchien ist in einer eigenen Spezifikation erfasst. Die Binärdatei, welche projiziert werden soll, enthält selbst das in der Mitte dargestellte Typsystem. Dieses Typsystem stammt aus den Testdaten von Felden [Fel18a] und trägt den Namen *subtypes*.

Die Abbildung zwischen den Typsystemen wird getestet indem die in der Mitte dargestellte *subtypes*-Hierarchie auf die anderen Hierarchien abgebildet wird. Auch eine Abbildung auf das *subtypes*-Typsystem selbst ist möglich und kann als Validierung verwendet werden. In diesem Fall ist die Ausgangsdatei gleich der Zieldatei. Über die Abbildungen auf die sieben anderen Spezifikationen werden alle Fälle der Abbildungsfunktion abgedeckt.

8.4.2. Objektübertragung

Die Objektübertragung kann durch Erweiterung der Typsystem-Tests überprüft werden. Zu diesem Zweck bekommt das Ausgangstypsystem zwei Objekte pro Typ. Diese müssen je nach Wert der Abbildungsfunktion ins neue Typsystem übertragen oder gelöscht werden. Da das Typsystem sehr klein ist, können die Ergebnisdateien manuell überprüft und dann als Erwartungsdateien übernommen werden.

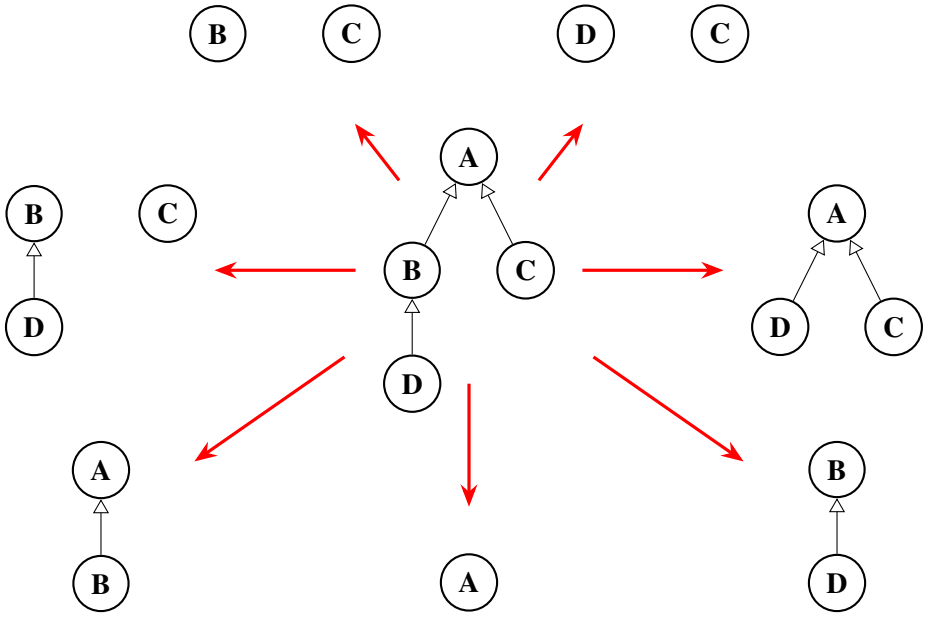

Abbildung 8.3.: Typhierarchien zum Testen von Aufbau und Abbildung von Typsystemen. In der Mitte ist die Ausgangshierarchie *subtypes* dargestellt.

Im Zuge der Objektübertragung wird auch die Berechnung der neuen SKilL-IDs getestet. Hier können die Typsystem-Tests wiederverwendet werden. Das *subtypes*-Typsystem von Felden [Fel18a] bietet ein Selbstreferenzfeld pro Typ. Somit haben durch die Vererbung Objekte des Typs *D* zum Beispiel drei Selbstreferenzen, wohingegen Objekte des Typs *A* nur eine haben. Diese Referenzen müssen im neuen Typsystem für alle Abbildungen aus den Typsystem-Tests richtig berechnet werden.

8.4.3. Feldübertragung

Nachdem das Typsystem und die Objektübertragung getestet wurden, kann als letztes die Feldübertragung geprüft werden. Dieser Abschnitt beinhaltet den größten Teil der Tests für die Spezifikationsprojektion. Zunächst wird die Feldsuche im neuen Typsystem getestet. Anschließend erfolgen die umfangreichen Tests zur Typprüfung. Für die positiv erwarteten Typprüfungen wird zudem die Übertragung mitgetestet.

Feldsuche Bei den Tests der Objektübertragung wurde bereits der Fall abgedeckt, in dem das gesuchte Feld direkt über die Abbildung zu erreichen ist. Somit ist an dieser Stelle noch das Verschieben von Feldern in der Hierarchie nach oben und unten abzudecken. Dabei genügt ein kleiner Testfall, wie in Abbildung 8.4 dargestellt. Nach rechts erfolgt eine Verschiebung nach oben. Diese ist immer möglich, da mögliche Objekte von *A* den Standardfeldwert für `i8 x` bekommen. Die Verschiebung in die andere Richtung ist schwieriger. Hier verliert der Typ *A* ein Feld und somit auch alle seine Objekte. Damit gehen Informationen verloren, es sei denn *A* hat keine

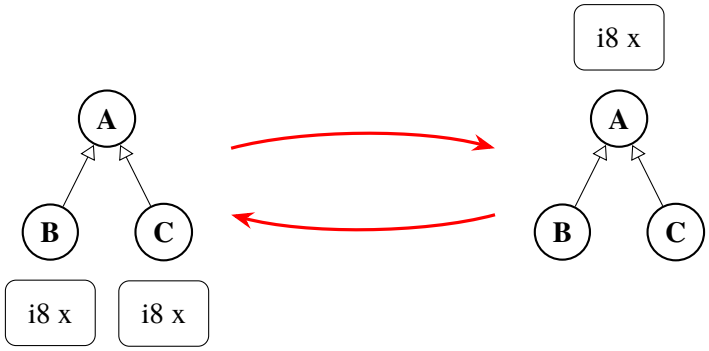

Abbildung 8.4.: Test zur Verschiebung von Feldern im Typsystem. Nach rechts: Verschiebung nach oben. Nach links: Verschiebung nach unten.

Objekte. Aus Gründen der Umkehrbarkeit der Spezifikationsprojektion ist es trotzdem möglich die Operation durchzuführen. Es wird allerdings eine Ausgabe erzeugt, welche auf den möglichen Informationsverlust hinweist.

Typprüfung Die Typprüfung kann in zwei Teile unterteilt werden. Der erste Teil besteht aus Tests, welche einen positiven Ausgang haben. Diese werden hier nicht extra erwähnt, da diese Fälle durch die Feldübertragung geprüft werden können. Wichtiger sind in diesem Abschnitt die fehlschlagenden Tests. Für diese Tests wird keine Ausgabedatei geschrieben. Deshalb ist hier ein Testen durch die Erwartungsdateien nicht möglich. In diesem Fall müssen die Fehlermeldungen geprüft werden.

Zunächst wird die Typprüfung innerhalb der drei Datentyp-Gruppen (Grunddatentypen, Anwendertypen, Container) getestet. Anschließend erfolgen Tests welche die Inkompatibilität zwischen den Gruppen sicherstellen.

Bei den Grunddatentypen sind die unabhängigen Typen `bool` und `string` am Einfachsten zu testen. Für diese erfolgt jeweils ein fehlschlagender Test auf alle Zahlentypen. Für die Zahlentypen untereinander können nur die mit Fragezeichen markierten Fälle in Tabelle 5.2 inkompatibel sein. Diese können durch Tests mit den maximalen Werten von `long` und `double` (unendlich) sowie dem not-a-number Wert erreicht werden. Diese Werte werden auf ihre Kompatibilität mit jedem einzelnen Zahlentyp geprüft. Die Ergebnisse müssen der Kompatibilitstabelle entsprechen.

Aufgrund des Aufbaus der Typprüfungen werden die Fälle eventuell kompatibel und nicht kompatibel in der statischen Typprüfung für Anwendertypen nicht unterschieden. Beide werden erst in der dynamischen Prüfung in kompatibel und nicht kompatibel getrennt. Dennoch ist es für die Tests wichtig, beide Fälle zu berücksichtigen. Ein Teil der Fälle ist bereits durch die Tests zur Prüfung der Objektübertragung und SKilL-ID Berechnung abgedeckt. Die fehlenden Möglichkeiten wurden durch ein Coverage Tool ermittelt und durch speziell entwickelte Tests abgedeckt.

Die linearen Containertypen sind alle zueinander kompatibel. Lediglich das Array mit fester Größe bildet eine Ausnahme. Hier müssen Tests entwickelt werden, welche zu viele oder zu wenige Elemente in den Containern aufweisen. Zusätzlich müssen nicht kompatible Basistypen berücksichtigt werden. Diese können entweder schon in der statischen Prüfung fehlschlagen oder erst in der dynamischen Prüfung. Der Datentyp `Map` ist zu den anderen inkompatibel. Diese

Inkompatibilität wird in beide Richtungen überprüft. Die Basistypen der `Map` müssen ebenfalls berücksichtigt werden. Hier kommt erschwerend hinzu, dass alle Basistypen kompatibel zueinander sein müssen. Um dies abzudecken, werden fehlschlagende Test für den ersten und letzten Typ entwickelt, welche jeweils einmal statisch oder dynamisch fehlschlagen.

Für die Gruppenübergreifenden Tests können beliebige Datentypen aus den Gruppen gewählt werden. Die Tests schlagen alle bereits in der statischen Prüfung fehl. Wichtig ist an dieser Stelle mindestens einmal von jeder Gruppe in beide anderen Gruppen zu prüfen.

Übertragung der Feldwerte Nach den Tests für die negative Typprüfung sind nun noch die positiven Ergebnisse übrig. Diese werden getestet indem die Werte vor und nach der Projektion auf ihre Gleichheit überprüft werden. Im Speziellen wird versucht, alle Zahlentypen aufeinander zu übertragen, alle Container ineinander übergehen zu lassen sowie einige Tests zur Übertragung von Referenzen.

8.4.4. Mappingdatei

Um die komplette Funktionalität der Mappingdatei zu testen, reichen wenige Tests aus. Dabei können die Tests aus den vorherigen Kapiteln wiederverwendet werden. Es wird lediglich eine zusätzliche Mappingdatei hinzugefügt. Die veränderten Namen werden durch ein manuelles Darstellungstool geprüft. Zusätzlich kann durch kleine Mappingdateien die Funktionsfähigkeit des Parsers sichergestellt werden. Diese Tests umfassen das Einlesen der drei vorgestellten Mappingarten.

8.5. Restrictions prüfen und ändern

Da die meisten Restrictions für diese Arbeit neu implementiert wurden, müssen zunächst Tests für die Deserialisierung und Serialisierung entwickelt werden. Im Anschluss erfolgen Tests zur Prüfung der Restrictions. Als Letztes ist die Änderung der Restrictions als Teil der Spezifikationsprojektion zu testen.

8.5.1. Deserialisierung und Serialisierung

Für die Tests in diesem Abschnitt wird eine modifizierte Form einer bereits existierenden Binärdatei (*restrictionsAll*) von Felden [Fel18a] verwendet. Diese Datei enthält bereits einige Arten der Feld-Restrictions in serialisierter Form. Die Modifikation der Datei besteht darin, die übrigen Restrictions hinzuzufügen. Durch Einlesen und Ausgeben der Datei wird die Deserialisierung und Serialisierung der Restrictions getestet. Dies ist notwendig, da die meisten Restrictions für diese Arbeit in das Java-API implementiert wurden.

8.5.2. Restrictions prüfen

Bei der Prüfung der Restrictions werden zwei Klassen von Tests benötigt. Die erste Klasse bilden die positiven Tests. Das bedeutet, alle Prüfungen fallen positiv aus und es wird eine entsprechende Binärdatei geschrieben. Wichtiger sind an dieser Stelle die negativen Tests. Für jede Restriction werden spezielle Testdateien entwickelt, welche eine negatives Prüfungsergebnis hervorrufen. Für die `@abstract`-Restriction wird hier zum Beispiel ein Objekt der entsprechenden Klasse angelegt. In diesen Tests wird eine Exception erwartet, welche der negative Check wirft.

8.5.3. Restrictions ändern

Die Restrictions können in der Spezifikation verändert werden. Das Prinzip der Veränderung besteht zunächst aus Ignorieren der alten und Erzeugen der neuen Restrictions. Somit ist eine Spezifikation, welche alle Restrictions enthält ausreichend. In diesem Fall wird die Binärdatei aus dem Test für die Deserialisierung und Serialisierung verwendet. Diese wird auf seine eigene Spezifikation projiziert. Das Ergebnis der Projektion ist die Ausgangsdatei. Da im Verlauf der Spezifikationsprojektion zunächst alle Restrictions verworfen und anschließend neu generiert werden, wird die komplette Erzeugung der Restrictions getestet.

Einen Sonderfall bilden die `@default`-Restrictions für Typen und Felder. Hier müssen jeweils spezielle Tests entwickelt werden, welche das Setzen der Standardwerte prüfen. Dabei wird ausgenutzt, dass bei der Erzeugung neuer Felder immer die Standardwerte gesetzt werden. Somit wird sowohl die Feld- als auch die Typ-Restriction überprüft.

8.6. Normalisierung mit einer Ordnungsfunktion

Die Normalisierung mit einer Ordnungsfunktion ist abstrakt definiert, um dem Anwender die Möglichkeit zu geben seine Ordnungsrelation umzusetzen. Deshalb werden in den Tests einige Beispielrelationen entworfen. Listing 8.1 zeigt ein Beispiel für eine Relation, welche die Objekte anhand ihrer Feldwerte sortiert. Die definierte Relation muss durch Angabe eines Feldes instanziiert werden. Aufgrund des Vergleichs in der `compare`-Funktion überprüft der Konstruktor von `FieldComparator`, ob ein Vergleich der Werte überhaupt möglich ist. In den Tests wird die Relation auf ein Integerfeld angewendet. Durch Betrachtung der Testdatei mit einem Darstellungstool erfolgt eine Überprüfung der Sortierung. Anschließend wird die Datei als Erwartungsdatei übernommen.

In einem weiteren Test wird der Feldvergleich so umgebaut, dass die SKilL-IDs von Objekten verglichen werden. Dies ist möglich, indem der `compare`-Aufruf nicht auf den Objekten selbst, sondern auf ihren IDs durchgeführt wird. Die Erweiterung durch Typ-Relation-Zuordnungen wird über die Kombination verschiedener Relationen getestet.

Listing 8.1 Pseudocode einer Ordnungsrelation durch Vergleich von Feldwerten

```
public abstract class FieldComparator extends SkillObjectComparator {
        private Field f;

        public FieldComparator(Field f) {
                if(/* values of f are not comparable */) throw Exception();
                this.f = f;
        }

        @Override
        public int compare(SkillObject o1, SkillObject o2) {
                Object value1 = f.get(o1);
                Object value2 = f.get(o2);

                if(value1 == value2) return 0;
                if(value1 == null) return 1;
                if(value2 == null) return -1;

                return value1.compareTo(value2);
        }
}
```

Modul	Befehle			Verzweigungen		
	Abgedeckt	Gesamt	Anteil	Abgedeckt	Gesamt	Anteil
Garbage Collector	666	66	100 %	121	121	100 %
Typ- und Feldentfernung	391	391	100 %	68	68	100 %
Spezifikationsprojektion (inklusive Restrictions)	2596	2632	98,6 %	375	402	93,3 %
Ordnungsrelation	122	122	100 %	16	16	100 %

Tabelle 8.3.: Testabdeckung der Befehle und Verzweigungen. Prozentzahlen sind auf eine Nachkommastelle gerundet. Die Messung wurde am 01.10.2018 mit dem Tool *EclEmma* [Mou18] durchgeführt.

8.7. Zusammenfassung

Für jeden Teil der Arbeit wurden entsprechende Tests entwickelt. Ein Anhaltspunkt der Testabdeckung geben die abgedeckten Befehle und Verzweigungen. Diese sind in Tabelle 8.3 zu sehen. Drei der vier Module haben eine vollständige Testabdeckung. Lediglich bei der Spezifikationsprojektion fehlen wenige Befehle und Verzweigungen. Der Grund hierfür sind Codestellen, welche nie vom Programmfluss erreicht werden können. Jedoch führt das Weglassen dieser Befehle zu inkorrekten Java-Sprachregeln des Programms. Zum Beispiel müssen beim Erben von einer Klasse alle abstrakten Methoden implementiert werden, um die Subklasse instanziieren zu können. Dabei kann es vorkommen, dass einige der geerbten Methoden aufgrund der Programmlogik niemals aufgerufen werden. Dennoch muss hier eine Dummy-Implementierung angegeben werden, um den Java-Sprachregeln zu genügen.

Die hohe Testabdeckung zeigt, dass alle Transformationsoperationen in ausreichender Weise getestet wurden. Somit ist die Funktionsfähigkeit der Module für die Testdaten sichergestellt. Zudem ist eine fehlerhafte Ausführung auf anderen Daten nicht zu erwarten.

9. Zusammenfassung und Ausblick

In dieser Arbeit wurden mehrere Operationen zur Manipulation der Typisierung serialisierter SKilL-Graphen umgesetzt. Alle Transformationsoperationen sind in der Lage einen Graphen aus einer Binärdatei zu lesen, diesen zu bearbeiten und wieder in eine Binärdatei zu schreiben.

Zunächst wurde eine effiziente Garbage Collection zur Entfernung nicht benötigter Knoten entwickelt. Der Algorithmus funktioniert nach dem Mark-Sweep Verfahren. Dieses wurde nach einer Analyse mehrerer Grundverfahren ausgewählt. Die Umsetzung des Algorithmus erfolgt durch einen Stack, welcher deutliche Vorteile gegenüber der rekursiven Implementierung aufweist. Das Markieren der Objekte ist durch ein `boolean`-Feld möglich. Dieses bietet die beste Performance unter den verglichenen Markierungsdatenstrukturen. Zur zusätzlichen Einsparung von Speicherplatz wird das Typsystem und die zugehörigen Strings auf ein Minimum reduziert.

Eine weitere Operation ermöglicht das Entfernen von ganzen Typen und einzelnen Feldern aus Typen. Die Laufzeit der Feldentfernung ist linear zur Anzahl der Felder im Zieltyp. Ferner ist das Entfernen von Feldern ein Teilschritt der Typentfernung. Diese ist wesentlich komplexer, da ein Typ durch die zentrale Position im API mehr Abhängigkeiten hat. Nach Durchführung der Entfernungsoperation wird in beiden Fällen ein konsistenter Zustand erreicht.

Den Hauptteil der Arbeit bildet die Spezifikationsprojektion. Diese ist in der Lage einen SKilL-Graphen auf eine neue Spezifikation zu projizieren. Die Betrachtung verwandter Arbeiten bringt keine Lösung für das Problem. Aus diesem Grund erfolgt die Entwicklung einer eigenen Architektur, um die Projektion durchzuführen. Diese Lösung besteht im Wesentlichen aus drei Teilschritten. Im ersten Schritt wird das Typsystem aus der Spezifikation erzeugt. Anschließend erfolgt der Aufbau einer Abbildung durch Namensäquivalenzen zwischen den Typsystemen. Diese wird im letzten Schritt benutzt, um die Daten zu übertragen. Während der Datenübertragung erfolgen Überprüfungen, ob alle Daten übertragen werden konnten. Eine zusätzliche Erweiterung der Projektion durch eine Mappingdatei ermöglicht die Änderung von Namen in der neuen Spezifikation.

Durch eine Erweiterung der Spezifikationsprojektion wurde eine Operation zur Änderung von Restrictions umgesetzt. Somit ermöglicht die Angabe einer neuen Spezifikation das Ändern oder Löschen von Restrictions. Zudem werden alle Restrictions durch Aufrufen ihrer `check`-Methoden am Ende der Projektion geprüft.

Die entwickelte Operation zur Normalisierung einer Ordnungsfunktion bewirkt die Umsortierung der SKilL-Objekte in einem Typ. Diese Sortierung wirkt sich auf die SKilL-IDs der Objekte aus und ermöglicht damit zum Beispiel eine Vereinfachung der Prüfung auf Isomorphie zweier Graphen. Die abstrakte Implementierung der Operation ermöglicht dem Anwender eine freie Definition der Ordnungsrelation.

Für alle Transformationsoperationen wurden Tests entwickelt, um ihre Funktionsfähigkeit sicherzustellen. Die Tests für die Module der Garbage Collection, Typ- und Feldentfernung sowie der Normalisierung mit einer Ordnungsfunktion zeigen eine vollständige Testabdeckung. Im Modul der

Spezifikationsprojektion, welches auch die Änderung der Restrictions beinhaltet, fehlen wenige Befehle und Verzweigungen. Diese sind im Programmfluss unerreichbar, werden aber aus syntaktischen Gründen benötigt. Eine Performanceanalyse ergab einen Laufzeitanteil der Typ- und Feldentfernung von unter einem Prozent der Gesamtlaufzeit inklusive der Lese- und Schreiboperationen. Die Garbage Collection zeigt eine lineare Abhängigkeit zwischen Ausführungszeit und Knotenanzahl des Graphen. Im Extremfall, bei welchem alle Knoten erreichbar sind, werden 2,6 Sekunden pro eine Million Knoten benötigt.

Eine Implementierung der Garbage Collection in einer anderen Programmiersprache könnte eine Verbesserung der Ausführungszeit herbeiführen. Hier wäre ein Vergleich mit anderen Programmiersprachen wie etwa C++ interessant, um eventuell eine effizientere Lösung zu finden. Die Umsetzung kann für diesen Vergleich migriert werden, da nur der Unterschied im Laufzeitsystem interessant ist.

Ferner kann eine Parallelisierung der Spezifikationsprojektion untersucht werden. Diese könnte vor allem die Übertragung der Daten beschleunigen. Dies ist nicht trivial möglich, da die Abbildung zwischen den Typsystemen kein eins-zu-eins Mapping ist.

A. Beispielspezifikation mit allen strukturell möglichen Feldern

Listing A.1 Beispieldeklarationen aller strukturell möglicher Felder

```
1  S {}
2  T:S {
3          const i8 a = 1;    // 8-Bit Ganzzahl mit konstantem Wert
4          const i16 b = 1;   // 16-Bit Ganzzahl mit konstantem Wert
5          const i32 c = 1;   // 32-Bit Ganzzahl mit konstantem Wert
6          const i64 d = 1;   // 64-Bit Ganzzahl mit konstantem Wert
7          const v64 e = 1;   // Ganzzahl mit variabler Größe(abhängig von Wert, max 64 Bit) mit
               konstantem Wert
8
9          bool f;            // Wahrheitswert
10
11         i8 g;              // 8-Bit Ganzzahl
12         i16 h;             // 16-Bit Ganzzahl
13         i32 i;             // 32-Bit Ganzzahl
14         i64 j;             // 64-Bit Ganzzahl
15         v64 k;             // Ganzzahl mit variabler Größe(abhängig von Wert, max 64 Bit)
16
17         f32 l;             // 32-Bit Gleitkommzahl
18         f64 m;             // 64-Bit Gleitkommazahl
19
20         string n;          // Zeichenkette
21
22         T t;               // Referenz auf Typ T
23         U u;               // Referenz auf Typ U
24 }
25 U:S {
26         i8[5] a;           // Array mit fester Größe(5) vom Typ 8-Bit Ganzzahl
27         i8[] b;            // Array mit variabler Größe vom Typ 8-Bit Ganzzahl
28         list<i8> c;        // Liste vom Typ 8-Bit Ganzzahl
29         set<i8> d;         // Set vom Typ 8-Bit Ganzzahl
30         map<i8,i8,i8> e;   // Mehrstellige Map: Bildet ein Paar von 8-Bit Ganzzahlen auf eine weitere
               Ganzzahl ab
31 }
32 V:S {
33         S[5] a;            // Array mit fester Größe(5) vom Typ S
34         S[] b;             // Array mit variabler Größe vom Typ S
35         list<S> c;         // Liste vom Typ S
36         set<S> d;          // Set vom Typ S
37         map<T,U,T> e;      // Mehrstellige Map: Bildet ein Referenztupel der Typen T und U auf den Typ
               T ab
38 }
```

B. Command Line Interface

Listing B.1 Hilfetext des Command Line Interface

```
usage: java -jar skillManipulator.jar (-gc/-specmap/-rm) -i skillfile [-o outfile] [-d] (mode
            dependent options)
This tool comes with three modes:
  gc: Garbage Collection
  specmap: Specification mapping
  rm: Remove field or type
Every mode MUST have an input skillfile and CAN have an outfile.
Every mode has its own options.

If you get an OutOfMemory-Error, try with more heap space.

Options:
 -h                          print this help page
 -gc                         gc mode
 -specmap                    specification mapping mode
 -rm                         remove type or field mode
 -i <file>                   Specifiy the binary file the chosen method is used on
 -d                          Set this flag for a dry run. No file is written if this option is
                             set.
 -o <file>                   Specify output file. Otherwise the input file is overwritten.
 -silent                     Disable all output operations.
 -r <root1,root2,...>        Specifiy the roots for the garbage collection. Roots can have the
                             form 'type' or the form 'type#id'.Examples: 'metainformation'
                             'imlgraph#1'
 -kC                         Keep empty collections and their referenced types after garbage
                             colletion.
 -s                          Print garbage collection statistics.
 -p                          Print removed objects.
 -spec <specification.skill> The specification file with the typesystem to map the binary file on.
 -map <mapping.map>          The optional mapping file.
 -f <type.field>             Remove specified field from the typesystem.
 -t <type>                   Remove the given type from the typesystem including all subtypes and
                             objects.
```

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Literaturverzeichnis

[Apa17] Apache. *Apache Commons CLI 1.4*. 2017. URL: https://commons.apache.org/proper/commons-cli/ (zitiert auf S. 9).

[Bab16] L. Babai. „Graph Isomorphism in Quasipolynomial Time [Extended Abstract]“. In: *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*. ACM, 2016, S. 684–697 (zitiert auf S. 53).

[Buz18] D. Buzdin. *Dozer 6.4.1*. 2018. URL: https://github.com/DozerMapper (zitiert auf S. 32).

[BW88] H.-J. Boehm, M. Weiser. „Garbage collection in an uncooperative environment“. In: *Software: Practice and Experience* 18.9 (1988), S. 807–820 (zitiert auf S. 20).

[Coh81] J. Cohen. „Garbage collection of linked data structures“. In: *ACM Computing Surveys (CSUR)* 13.3 (1981), S. 341–367 (zitiert auf S. 17).

[Fel17a] T. Felden. „Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen“. Diss. Universität Stuttgart, 2017 (zitiert auf S. 7, 23, 56, 58).

[Fel17b] T. Felden. *The SKilL Language V1.0*. Technical Report Computer Science 2017/01. Universität Stuttgart, 2017, S. 64 (zitiert auf S. 11–13, 33, 46, 49).

[Fel18a] T. Felden. *skill*. 2018. URL: https://github.com/skill-lang/skill (zitiert auf S. 36, 62, 63, 65).

[Fel18b] T. Felden. *skillGC*. 2018. URL: https://github.com/skill-lang/skillGC (zitiert auf S. 22, 58).

[Hla18] M. Hlavac. *stargazer: Well-Formatted Regression and Summary Statistics Tables*. 2018. URL: https://CRAN.R-project.org/package=stargazer (zitiert auf S. 59).

[JL96] R. Jones, R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley Chichester, 1996 (zitiert auf S. 17).

[LYBB15] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (zitiert auf S. 22, 43, 44).

[Mic97] S. Microsystems. *Java Beans Specification*. 1997. URL: http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/ (zitiert auf S. 32).

[Mou18] Mountainminds GmbH & Co. KG and Contributors. *EclEmma 3.1.0*. 2018. URL: https://www.eclemma.org/index.html (zitiert auf S. 67).

[Ora15a] Oracle. *Java Platform Standard Edition 8: Class ArrayDeque*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/index.html?java/util/ArrayDeque.html (zitiert auf S. 24).

[Ora15b] Oracle. *Java Platform Standard Edition 8: Class Arrays*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html (zitiert auf S. 54).

[Ora15c] Oracle. *Java Platform Standard Edition 8: Class BitSet*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html (zitiert auf S. 23).

[Ora15d] Oracle. *Java Platform Standard Edition 8: Interface Comparator*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html (zitiert auf S. 54).

[Ora15e] Oracle. *Java Platform Standard Edition 8: Interface Stream*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html (zitiert auf S. 32).

[Rat17] M. Rathgeber. „SKilL Graph Visualiziation and Manipulation“. Diplomarbeit. Universität Stuttgart, 2017 (zitiert auf S. 55).

[SPT83] L. K. Schubert, M. A. Papalaskaris, J. Taugher. „Determining type, part, color and time relationships“. In: *IEEE Computer* 16.10 (1983), S. 53–60 (zitiert auf S. 44).

[The18] The JUnit Team. *JUnit 5 Version 5.3.1*. 2018. URL: https://junit.org/junit5/ (zitiert auf S. 55).

[Wei16] C. Weißer. „Serialization of foreign types with SKilL“. Masterarbeit. Universität Stuttgart, 2016 (zitiert auf S. 33, 34, 36, 46).

[Wil92] P. R. Wilson. „Uniprocessor garbage collection techniques“. In: *Memory Management*. Springer, 1992, S. 1–42 (zitiert auf S. 17, 20).

Alle URLs wurden zuletzt am 02. 10. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift