Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 219

Zustandsverwaltung mit SKilL
State management with SKilL

Christopher Völker

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	8. April 2015
Beendet am:	8. Oktober 2015
CR-Nummer:	D.1.5, D.2, D.3.3, E.2, E.5

Kurzfassung

In dieser Bachelorarbeit wird die Verwaltung von Zuständen und der darin beinhalteten Typen sowie deren Instanzen untersucht. Hierbei soll die Funktionalität des Suchens, Löschens und Kopierens von Instanzen implementiert werden. Die Arbeit untersucht dafür zunächst die nötigen Schritte, um die gewünschten Funktionen zu implementieren und dabei den Aufwand für die Nutzung dieser so gering wie möglich zu halten. Daher liegt der Schwerpunkt dieser Implementierung darin, dem Nutzer eindeutige Abläufe vorzugeben, für welche er keine tiefere Kenntnis bezüglich ihrer genauen Funktionalität benötigt. Durch die Vorgabe dieser Abläufe soll eine unsachgemäße Nutzung der Funktionen vermieden werden. Ein Weiterer Schwerpunkt liegt bei der Effizienz der Suche nach Instanzen. Ziel ist es, diese Suche so effizient wie möglich zu realisieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

In diesem Abschnitt werden neben einer kurzen Übersicht zum Aufbau des Dokuments einige Hintergrundinformationen zu dieser Arbeit beschrieben. Hierzu gehört neben SKilL und der Aufgabenstellung dieser Arbeit auch eine Übersicht über ähnliche Arbeiten zu diesem Thema.

1.1 Aufbau des Dokuments

In diesem Dokument werden zunächst die für das Verständnis notwendigen Informationen sowie eine kurze Einführung in die Sprache SKilL beschrieben. Das Einführungskapitel beschäftigt sich mit der Aufgabenstellung dieser Bachelorarbeit und es werden andere ähnliche wissenschaftliche Arbeiten kurz vorgestellt. Das anschließende zweite Kapitel handelt von der Java-Implementierung von SKilL, in welcher die Implementierung der gewünschten Funktionen erfolgt. Im nachfolgenden dritten Kapitel werden die Ergebnisse dieser Arbeit im Bezug auf die Verwaltung von Zuständen sowie damit verbundene Einschränkungen beschrieben. Das darauffolgende vierte Kapitel handelt von Verbesserungsvorschlägen, welche im Zuge dieser Arbeit entstanden sind und Aufgrund von mangelnder Zeit nicht mehr umzusetzen waren. Zuletzt wird im fünften Kapitel eine kurze Zusammenfassung dieser Arbeit und auch ein Ausblick auf mögliche weiterführende Entwicklungen gegeben.

1.2 Hintergründe der Arbeit

Im Zeitalter der Digitalisierung werden jeden Tag große Mengen an Daten generiert und verarbeitet. Es werden jeden Tag Millionen von Nachrichten, Bildern, Videos oder anderen Arten von Daten ausgetauscht, verschickt und gespeichert. Die Daten werden in einer Darstellung erstellt, welche von Menschen gelesen, verstanden und bearbeitet werden kann. Ein weitaus deutlicheres Beispiel für das Generieren von großen Mengen an Daten sind wissenschaftliche Projekte. In solchen Projekten werden Unmengen an Informationen generiert und müssen verarbeitet, gespeichert und zwischen verschiedenen Tools transferiert werden. Die Menge an Daten, welche dabei täglich entsteht, wird mit der fortschreitenden Digitalisierung und den dabei stetig wachsenden Möglichkeiten immer größer und wächst schneller als die Anzahl der für die Verarbeitung nötigen Ressourcen. Das heißt, dass die Verarbeitungsprozesse immer länger dauern, da der technische Fortschritt und dessen Ausbau mit der zunehmenden Digitalisierung nicht mithalten kann. Dies hängt unter Anderem auch damit zusammen, dass der Ausbau von schnellen Anbindungen an das Internet ein langwieriger und auch kostspieliger Prozess ist. Doch auf die lokale Verarbeitung und das dortige Speichern von Daten wird in diesem Zusammenhang immer langwieriger. Aus diesem Grund wird es immer wichtiger, dass Daten möglichst effizient gespeichert und verschickt werden können, wobei dies nicht in der initialen für Menschen lesbaren Darstellung geschehen kann. Damit Daten gespeichert oder verschickt werden können, müssen diese in ein bestimmtes Format gebracht werden. Dieses Format muss von digitalen Medien (wie zum Beispiel Computern), mit-

hilfe welcher die Daten in der Regel erstellt werden, gespeichert und ausgetauscht werden können. Das überführen der Daten in dieses Format wird „Serialisieren“ genannt. Ein solches Verfahren muss also in der Lage sein, Daten zwischen einer für Computer verständlichen Darstellung in eine Menschen verständliche Darstellung und wieder zurück zu überführen.

Das Serialisieren von Daten Das Serialisieren von Daten stellt natürlich einen zusätzlichen aber nötigen Aufwand in der Verarbeitung von Daten dar. Das Hauptproblem hierbei ist, dass die gängigsten Verfahren in der Regel alle Informationen Serialisieren und De-serialisieren statt nur diejenigen Daten zu verarbeiten, welche tatsächlich benötigt werden. Das populärste Beispiel für ein solches Verfahren stellt die Extensible Markup Language (XML) dar. [XML06] Dieses Verfahren stellt Daten hierarchisch dar und zwingt den Nutzer hierdurch dazu alle Daten dieser Hierarchie mindestens einmal „in die Hand zu nehmen“. Bei sehr großen Mengen an Daten wäre es aber zugunsten der Effizienz wünschenswert, dass tatsächlich nur diejenigen Daten verarbeitet werden, welche wirklich benötigt werden. Dabei soll es natürlich möglich sein, bei Bedarf bestimmte Daten möglichst effizient nachträglich zu laden, ohne dabei bereits geladene Daten erneut zu verarbeiten. Ein solches Verfahren, welches möglichst unabhängig von Sprachen und Plattformen sein soll und zeitgleich diese Funktionalität bietet existiert zum jetzigen Zeitpunkt noch nicht. An dieser Stelle setzt die Sprache „Serialization Killer Language“ (SKilL) an.

1.3 Die Sprache SKilL

Bei SKilL handelt es sich um die Entwicklung einer Serialisierungssprache an der Universität Stuttgart. Diese Sprache wird entwickelt um das Serialisieren und De-serialisieren von großen Mengen an Daten schneller und einfacher zu machen. Ein sehr großer Aufwand bei der Handhabung großer Mengen an Daten fließt in die Nutzung dieser in verschiedenen Programmiersprachen. In vielen Fällen müssen die Daten in verschiedenen Tools verarbeitet werden, welche in unterschiedlichen Programmiersprachen realisiert wurden. Um dies in den jeweiligen Tools zu ermöglichen, müssen die Datenstrukturen in jeder der verschiedenen Programmiersprachen erstellt werden. SKilL verfolgt das Ziel auch diesen Aufwand zu minimieren. Hierfür wurde zunächst eine Sprache für die Spezifizierung von Datenstrukturen entwickelt, welche es ermöglicht zum Teil komplexe Datenstrukturen zu definieren und diese von Tools verschiedener Programmiersprachen nutzen zu können. Dabei wurde darauf geachtet, dass die Spezifikation Ähnlichkeiten zu Objekt-orientierten Programmiersprachen hat um die Verwendung in diesen so leicht wie möglich zu machen. Dies bedeutet auch, dass abgesehen von den Basistypen der Programmiersprachen (wie zum Beispiel in Java int, long, string etc.) auch Benutzertypen definiert werden können. Dabei gilt natürlich, dass der letzte Benutzertyp in einer Typ-Hierarchie ausschließlich aus Basistypen bestehen kann. Diesem Prinzip entsprechend kann ein Benutzertyp auch in einem weiteren Benutzertyp verwendet werden. Den verschiedenen Datentypen werden dabei in SKilL so genannte „TypeIDs “ zugewiesen, wobei diese eindeutig spezifiziert sind. In Abbildung 1 ist die Verteilung dieser TypeIDs dar-

Type Name	Value
const i8	0
const i16	1
const i32	2
const i64	3
const v64	4
annotation	5
bool	6
i8	7
i16	8
i32	9
i64	10
v64	11
f32	12
f64	13
string	14
T[i]	15
T[]	17
list<T>	18
set<T>	19
map<T_1, ..., T_n>	20
T	$32 + index_T$

Abbildung 1: Typ IDs in SKilL
Quelle: [Fel13, Anhang E, Numerical Constants]

gestellt. Hieraus wird deutlich, dass Benutzertypen im Gegensatz zu Basistypen keine definierte ID besitzen, sondern diese aufsteigend eine ID zugewiesen bekommen. Dabei hat diese ID per Definition einen offset von 32 um sich nicht mit den Basistypen zu überschneiden. Hierbei werden die IDs der Benutzertypen in der Reihenfolge, in welcher sie eingelesen werden zugewiesen. Diese Reihenfolge hängt unter Anderem von den Abhängigkeiten in den Spezifikationen ab. In der Abbildung 2 (a) ist die naive Spezifikation eines Studenten zu sehen. Hier besitzt ein Student einen Vor- und Nachnamen und hat natürlich auch ein Geburtsdatum. Da es sich um einen Student handelt gibt es auch ein Feld für seinen Studiengang. In diesem Beispiel ist die Verwendung eines so genannten „Benutzertyps“ in einem weiteren „Benutzertyp“ zu sehen, welcher in Abbildung 2 (b) spezifiziert ist. In diesem Fall gibt es zwei Benutzertypen, welche eine entsprechende ID zugewiesen bekommen müssen. Um jedoch den Typ „Student“ korrekt darstellen zu können, muss zuvor der Typ „Geburtstag“ bekannt sein. Aus diesem Grund bekommt der Typ „Student“ seine ID erst nach dem Typ „Geburtstag“ . Grund dafür ist, dass der Student nicht ohne das bekannt sein des Geburtstags erstellt werden kann.

Ziel einer solchen Spezifikation ist nun, dass die Kern Datenstrukturen eines Tools Plattform unabhängig sind, denn die Programmiersprachen, in welcher ein Entwickler am besten entwickelt, ist diejenige die er am liebsten nutzt. [Fel13, Kapitel 1, Absatz 2]

Aus diesem Grund wird eine solche Spezifikation mithilfe eines Code-Generators nun zu Klassen einer bestimmten Sprache generiert, damit die darin definierten Datenstrukturen dort verwendet werden können. Ein wichtiger Aspekt hierbei ist, dass das Einlesen von Zuständen aus Dateien abhängig von der entsprechenden Spezifikation ist. Die hierfür nötigen Funktionen werden von dem Code-Generator aus einer Spezifikation erstellt und können dann mithilfe der generierten Klassen verwendet werden. Das heißt aber auch, dass ein Zustand nach dem Einlesen zunächst nur diejenigen Typen kennt, welche in der Spezifikation definiert wurden. Ein großer Vorteil der Generierung von Klassen aus einer Spezifikation ist dabei, dass die Datenstrukturen auf diese weise bereits serialisiert vorliegen! Dabei ist die Entwicklung eines Code-Generators für verschiedene Objektorientiere Sprachen, ausgehend von einer einzigen Spezifikation, wesentlich einfacher, als die Anpassung der Spezifikation an verschiedene Sprachen. Ein weiterer Vorteil der Generierung dieser Klassen ist, dass die nötigen Datenstrukturen nicht vom Entwickler in verschiedenen Sprachen realisiert werden müssen, sondern diese Arbeit vorab durch den Generator durchgeführt wird. Dies macht das Arbeiten mit bestimmten Datenstrukturen in Tools verschiedener Programmiersprachen wesentlich einfacher und erfordert deutlich weniger Aufwand. Auf diese Art und Weise kann also folglich ein und die selbe Spezifikation von Datenstrukturen in verschiedenen Programmiersprachen verwendet werden. Der Entwickler muss lediglich gegen die generierte Schnittstelle entwickeln anstatt diese zuerst erstellen zu müssen. Auch die Tatsache, dass der Aufbau dieser Schnittstelle in jeder Programmiersprache sehr ähnlich sein wird, bringt einige Vorteile in Sachen Benutzung mit sich.
Im Fall dieser Arbeit wird aus dieser Spezifikation Code für die Programmiersprache Java generiert. Hierbei stellen die generierten Klassen alle notwendigen Informationen und Funktionen zur Verfügung, welche benötigt werden um mit den Datenstrukturen der Spezifikation in Java arbeiten zu können.

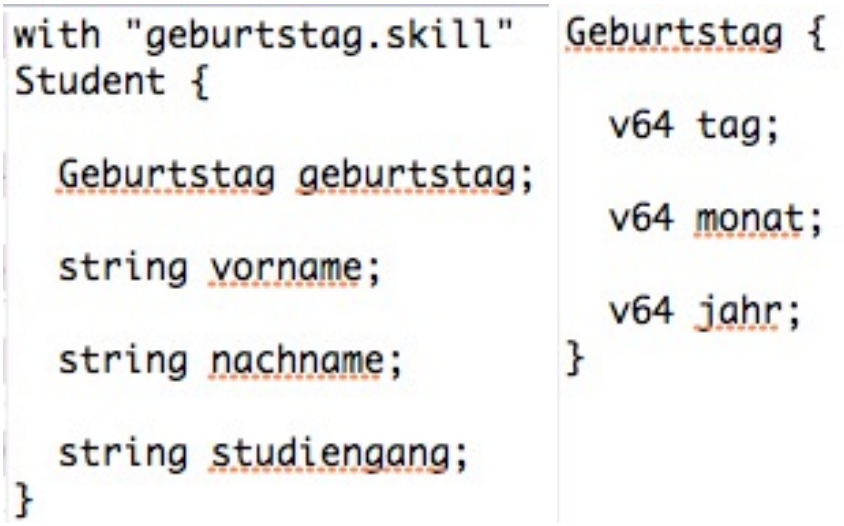
1.4 Aufgabenstellung

Die in SKilL verwendeten Zustandsobjekte (so genannte „States“) sind aus Sprachsicht Mengen von Typen, die wiederum Mengen von Instanzen sind. Diese Eigenschaft ist aktuell in keiner Implementierung direkt realisiert. Daher soll eine bereits existierende Implementierung um eine auf Mengenoperationen basierende API erweitert werden.

Löschen und Kopieren Es muss eine Funktionalität für das Löschen und Kopieren von Instanzen aus einem State geschaffen werden. Außerdem muss der Transfer von Instanzen zwischen unterschiedlichen States realisiert werden.

Element-Operator Es muss eine effiziente Implementierung des Element-Operators für Instanzen geschaffen werden.

Erstellung konsistenter States Es muss eine einfache Möglichkeit geschaffen werden einen konsistenten State aus mehreren Instanzen zu erzeugen. Insbesondere dürfen keine Zeiger auf Instanzen außerhalb des generierten States existieren.



```
with "geburtstag.skill"
Student {

  Geburtstag geburtstag;

  string vorname;

  string nachname;

  string studiengang;
}
```

(a) Student

```
Geburtstag {

  v64 tag;

  v64 monat;

  v64 jahr;
}
```

(b) Geburtstag

Abbildung 2: Spezifikation von Datenstrukturen

Mengen-Operationen Wünschenswerte wäre eine Realisierung von Mengensubtraktions- und Mengenvereinigungsoperationen für States, beziehungsweise für einzelne Typen innerhalb von States.

Benutzerdefinierte totale Ordnung Wünschenswert wäre die Möglichkeit States bezüglich einer benutzerdefinierten totalen Ordnung zu normalisieren, um zwei States vergleichen zu können.

Änderungsvorschläge Optional können Vorschläge gemacht werden, wie die Architektur bereits existierender SKilL-Bindings zu verändern ist um Mengenoperationen leichter oder effizienter zu realisieren.

1.5 Ähnliche Arbeiten

Im Bezug auf die sich in Entwicklung befindliche Sprache SKilL gab es noch weitere wissenschaftliche Arbeiten, welche hier kurz vorgestellt werden.

Plattform- und sprachunabhängige Serialisierung mit SKilL Bei dieser Arbeit handelt es sich um eine Diplomarbeit, welche von Fabian Harth verfasst wurde. Diese Arbeit wurde im Jahr 2014 erstellt und befasst sich mit der Anbindung von SKilL an die Programmiersprache C. Bei dieser Arbeit wird die Möglichkeit der Anbindung von SKilL an eine nicht Objekt-orientierte Programmiersprache beschrieben. [Har14, Kurzfassung]

Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache Diese Diplomarbeit wurde im Jahr 2014 von Wladislaw Ungur erstellt. Sie befasst sich mit der Nutzbarkeit und praktischen Tauglichkeit einer Schnittstelle und des passenden Code-Generators für die Serialisierungssprache SKilL. Hierbei liegt der Schwerpunkt auf der Entwicklung einer API und eines Code-Generators für die Programmiersprache Java. [Ung14, Kurzfassung]

Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache Im Jahr 2014 befasste sich Dennis Przytarski mit der Performanz von einer Serialisierungssprache und evaluierte diese mithilfe eines Scala- und Ada-Bindings. Das Scala-Binding existierte hierbei als Referenzimplementierung während für die Programmiersprache Ada ein Code-Generator entwickelt werden musste. [Prz14, Kurzfassung]

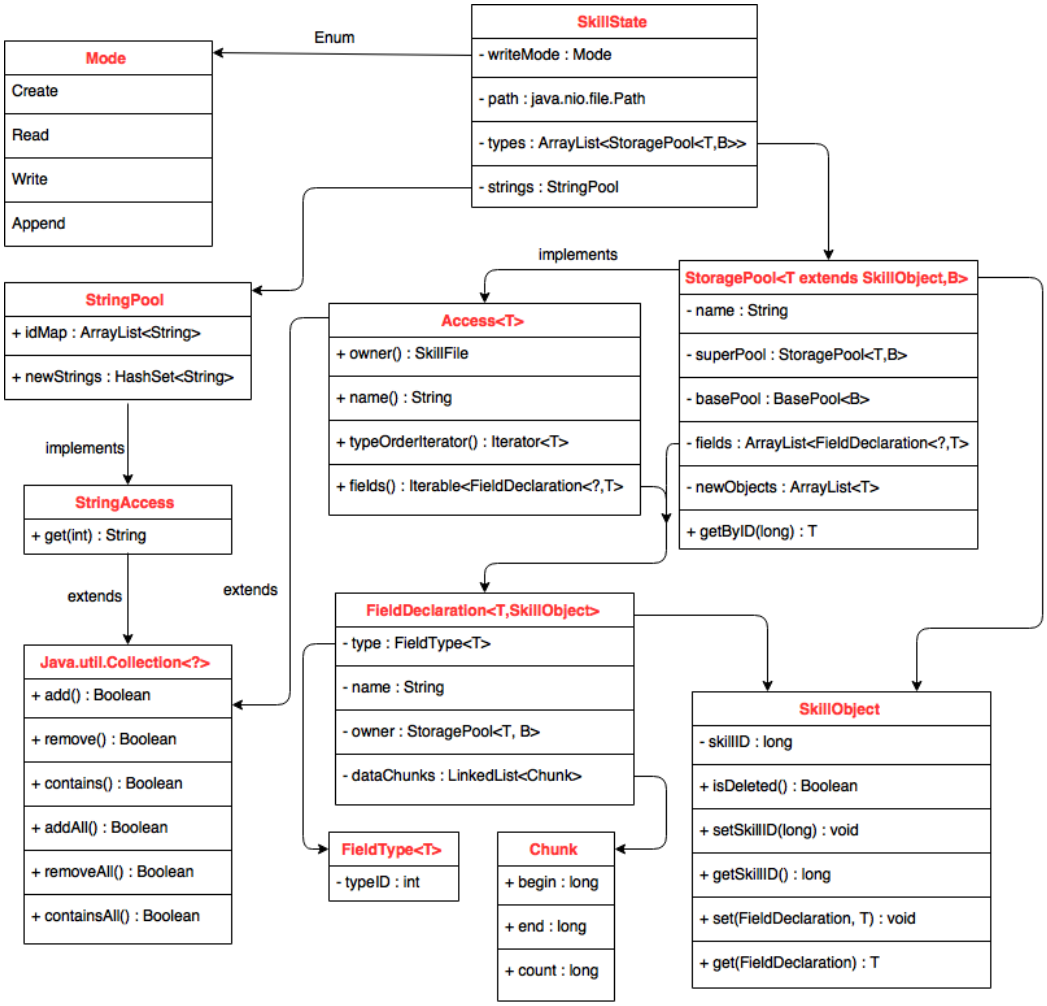


Abbildung 3: SKilL in Java (vereinfacht)

2 SKilL in Java

Die Umsetzung der Funktionalitäten, mit welchen sich diese Arbeit beschäftigt, erfolgt in der Java-Implementierung von SKilL. Aus diesem Grund wird die Implementierung von SKilL in Java in diesem Abschnitt beschrieben. Dies ist notwendig, um die Hintergründe der Umsetzung verstehen und nachvollziehen zu können. Hierbei beschränkt sich der Abschnitt auf die wichtigsten Informationen zur Implementierung. Die nachfolgende Beschreibung hat folglich keinen Anspruch auf Vollständigkeit, da nicht alle Einzelheiten für das Verständnis der Umsetzung nötig sind.

Zunächst erfolgt eine kurze Beschreibung eines so genannten SKilL-Zustands, welcher im folgenden „Zustand“ genannt wird. Anschließend werden die wichtigsten und für den weiteren Verlauf dieser Arbeit nötigen Bestandteile sowie deren Funktion etwas detaillierter beschrieben.

2.1 Der Zustand

Ein Zustand in SKilL, stellt die oberste Ebene in der Verwaltung von Daten dar. Ein Zustand wird als Java-Objekt im Sinne der objektorientierten Programmierung realisiert. Ein solcher Zustand beinhaltet alle für das spätere Serialisieren notwendigen Daten und dient als Repräsentation der Daten im Hauptspeicher.
Ein Zustand ist also für die Verwaltung von Typen, ihren Definitionen, Restriktionen und Instanzen zuständig und ermöglicht es diese geordnet zu Repräsentieren und mit ihnen zu arbeiten. Bei Instanzen handelt es sich hierbei um Objekte eines Typs, welche bestimmte Werte für dessen Felder beinhalten und repräsentieren. Der initiale Zugriff auf diese Daten wird somit über einen Zustand realisiert. Im Zuge der Verwaltung dieser Daten ist ein Zustand auch dafür verantwortlich, die Daten in Dateien zu schreiben und aus solchen auch wieder auslesen zu können. Konkret bedeutet dies, dass dieser auch das Serialisieren und De-serialisieren von Daten realisiert beziehungsweise den Prozess dessen anstößt.
Jeder Zustand zeichnet sich abgesehen von seinem Inhalt durch zwei bestimmte Attribute aus. Diese Attribute sind der „Pfad“ und der „Schreibmodus“.

Schreibmodus Jeder Zustand unterscheidet bei seiner Erstellung zunächst zwischen einem so genannten „openMode“ und einem „closeMode“.
Der „openMode“ differenziert hierbei zwischen einer Lese-Operation auf eine bereits vorhandene Datei oder einer Erstellen-Operation für eine neue Datei. Letzteres wird ausschließlich für das Erstellen neuer Zustände verwendet, für welche es noch keine Datei gibt. Der openMode lässt sich nachträglich nicht mehr verändern, da dieser nur bei der Erstellung eines Zustandes im Hauptspeicher zur Verwendung kommt.
Der „closeMode“ unterscheidet die Art der Schreib-Operation auf einen Zustand. Es gibt zum einen den Modus „Write“, welcher für das Schreiben von Änderungen im Bezug auf den Inhalt oder das Löschen von Inhalten verwendet wird. Zum Anderen gibt es einen „Append-“ Modus, der für das Hinzufügen neuer Inhalte zu einem (bestehenden) Zustand optimiert wurde. Hier gilt zu beachten, dass man den Modus zwar von Append zu Write ändern kann, die andere Richtung jedoch nachträglich nicht zulässig ist. Generell ist ein nachträgliches Ändern des Modus zu Append in keinem Fall zulässig. Dies dient der Fehlervermeidung, da ein Zustand im „Append“ Modus ausschließlich auf das Hinzufügen neuer Inhalte optimiert wurde. Hiermit soll vermieden werden, dass durch unzulässiges Ändern des Schreibmodus Änderungen in einem Zustand nicht übernommen werden und so unter Umständen Inkonsistenzen auftreten.
Bei den Modi eines Zustandes ist zu beachten, dass der „openMode“ zwingend angegeben werden muss, während der „closeMode“ zunächst nicht angegeben werden muss. Dabei ist aber zu beachten, dass das nicht angeben des „closeMode“ impliziert, dass man den Zustand nach dem Erstellen nur noch um den Modus „Write“ ergänzen kann.

Read Dieser Schreibmodus wird verwendet, wenn eine Datei ausschließlich gelesen aber nicht verändert wird. Im Falle der Notwendigkeit einer Änderung im Zustand kann der Schreibmodus nachträglich noch um „Write“ ergänzt werden.

Create Create als initialer Schreibmodus ist für das Erstellen neuer Zustände zu verwenden. Create initiiert dabei das Erstellen eines neuen (leeren) Zustandes, in welchem dann Instanzen zu entsprechenden Typen oder gegebenenfalls auch neue Typen hinzugefügt werden können.

Write Der Modus „Write“ kann nur als der zweite Parameter für den Modus beim erstellen eines Zustandes übergeben werden. Dieser Schreibmodus wird für das Ändern von Inhalten in einem Zustand verwendet. Die Besonderheit dieses Modus ist, dass das Schreiben eines Zustandes in diesem Modus bei der Vergabe der SkillIDs alle IDs aktualisiert. Dies betrifft in diesem Fall auch die bereits vorhandenen Instanzen des Zustandes.

Append Ein sich im Modus „Append“ befindlicher Zustand kommt beim Hinzufügen neuer Inhalte zur Verwendung. Dieser Modus ist hierbei auf das Hinzufügen optimiert und befasst sich bei der Vergabe von SkillIDs ausschließlich mit der Vergabe der SkillIDs neuer Instanzen.

Pfad Der Pfad eines Zustandes entspricht bei seiner Erstellung zunächst dem Pfad der Datei, aus welcher er entstanden ist. Im Falle einer Neuerstellung eines Zustandes kann dieses Attribut zunächst leer sein. Spätestens unmittelbar vor dem Aufruf der Schreib-Operation in eine (neue) Datei muss der Pfad allerdings angegeben werden. Es empfiehlt sich jedoch im Sinne der Eindeutigkeit den Pfad in jedem Fall bei der Lese-Operation (unabhängig ob im Create- oder Read-Modus) mit anzugeben.

2.2 Bestandteile eines Zustandes

Ein Zustand stellt ein Objekt dar, in welchem die für die Verwaltung von Typen und deren Daten notwendigen Strukturen beinhaltet sind. Der Zustand ist unter Anderem auch dafür zuständig Daten vom Nutzer abzuschirmen, welche der Nutzer nicht zwingend benötigt oder deren falsche Nutzung gravierende Auswirkungen haben können. Aus diesem Grund beinhaltet der Zustand neben den Objekten für die Repräsentation von Typen und Daten auch entsprechende Zugriffsobjekte. Abbildung 3 zeigt eine vereinfachte Darstellung der wichtigsten Bestandteile sowie ihrer Abhängigkeiten so wie sie in Java realisiert sind. Diese Bestandteile werden in den nachfolgenden Abschnitten beschrieben.

2.2.1 StringAccess

Der StringAccess ist ein Interface, welches von einer Java-Collection von Strings ableitet. Dieses Interface stellt neben den Funktionalitäten einer üblichen Java-Collection auch eine Funktion zur Ausgabe eines Strings zur Verfügung. Hierfür wird der Funktion („get“) ein Parameter vom Java-Typ „java.lang.long“ übergeben, welcher dann für die Suche eines bestimmten Strings verwendet werden kann.

2.2.2 Access

Das Interface Access leitet ähnlich wie das Interface StringAccess von einer Java-Collection ab. Anders als der StringAccess ist der Typ, welcher von dieser Collection gehalten wird, nicht vorgegeben sondern in gewissem Maße generisch. Dieser generische Typ ist in soweit eingeschränkt, dass er selbst von der Klasse SkillObject abgeleitet sein muss. Das Access Interface legt also die Grundlagen für den Zugriff auf Objekte der Klasse SkillObject fest. Dies wird unter Anderem durch das Vorschreiben einer Implementierung von mehreren Funktionen bewerkstelligt, welche beim Implementieren des Interfaces ausgearbeitet werden muss. Hierzu gehört eine Funktion, welche den Namen des repräsentierten Typs zurück gibt sowie eine für den Erhalt des Namens des Supertyps (falls vorhanden). Auch ist eine Funktion für das Abrufen des Zustands, zu welchem dieses Access-Objekt gehört, vorgeschrieben. Des Weiteren gibt es noch drei weitere Funktionen, welche implementiert werden müssen. Hierzu gehört zum einen eine Funktion, welche die Felder dieses bestimmten Typs kennt und (in Form von „FieldDeclarations“) zurückgibt. Die letzten zwei Funktionen beschäftigen sich hierbei mit den Instanzen des Typs. Es gibt dabei eine Funktion, welche alle Instanzen in „typeOrder“ als iterierbares Objekt zurückgibt. In „typeOrder“ heißt, dass zunächst die Instanzen dieses Access-Objekts, gefolgt von denjenigen aller Untertypen als iterierbares Objekt zurückgegeben werden. Dies beinhaltet in diesem Fall auch diejenigen Instanzen, welche neu hinzugefügt wurden und sich somit in der Liste „newObjects“ befinden. Die andere Funktion ermöglicht die Erstellung neuer Instanzen des Typs, welche mit Standard-Werten befüllt werden.

2.2.3 SkillObject

Die Klasse SkillObject stellt die Grundlage für den Umgang mit Typen und deren Instanzen dar. Jedes SkillObject besitzt eine so genannte „SkillID“. Diese wird zur Identifikation des Objekts im entsprechenden Typ verwendet und es gibt hier drei verschiedene Bedeutungen, welche sich aus der SkillID ableiten lassen. Bei neuen Objekten ist die SkillID zunächst -1 bis diese durch den Schreibvorgang des das Objekt beinhaltenden Zustands entsprechend auf einen Wert größer 0 gesetzt wird. Ist die SkillID größer 0, so dient sie innerhalb dieses Typs als eindeutige Identifikation. Der Fall, dass die SkillID gleich 0 ist bedeutet, dass das Objekt gelöscht werden soll. In diesem Fall wird das Objekt beim Schreibvorgang „ignoriert“ und die IDs nachfolgender Objekte entsprechend angepasst und verschoben. Für die SkillID gibt es drei Funktionen in einem SkillObject, wobei die erste eine simple Abfrage dieser ist. Die zweite Funktion ermöglicht das Ändern der SkillID, während die dritte prüft, ob das Objekt gelöscht wurde indem sie die SkillID auf 0 überprüft. Zum Umgang mit Typen und deren Instanzen gehört natürlich auch das Speichern und Abrufen von Werten. Dies wird von diesen Objekten in Form von so genannter „Reflection“ gelöst. Reflection erlaubt es, Klassen und Objekte, welche zur Laufzeit von der „Java Virtual Machine“ (kurz JVM) gehalten werden zu untersuchen und in begrenztem Umfang zu modifizieren. [Ull10, Kapitel 25, Absatz 1] Die Nutzung der Reflection ist notwendig, da man ohne diese den Zugriff auf Daten ausschließlich über Felder realisieren könnte oder diesen für jede Spezifikation implementieren müsste.

Bei der Nutzung von Reflection wird im Falle des Abrufens von Daten (eines bestimmten Feldes) der Instanz das entsprechende Feld als Parameter übergeben. Werden Daten verändert beziehungsweise neu gesetzt, so wird zusätzlich noch ein Objekt als Parameter erwartet, welches dem Typ des Feldes entspricht und die „neuen“ zu speichernden Daten beinhaltet.

2.2.4 StringPool

Ein StringPool ist für die Verwaltung von bestimmten Strings verantwortlich. Diese Strings können zum einen die Namen der Typen sein, welche von dem Zustand, in welchem sich der StringPool befindet, verwaltet werden. Zum anderen befinden sich aber auch die zu den verwalteten Typen zugehörigen Feldnamen in einem StringPool. Für die Verwaltung dieser Strings wird von einem StringPool das Interface „StringAccess“ sowie die damit verbundene get-Funktion implementiert. Jeder String in einem StringPool besitzt eine ID anhand derer er eindeutig identifiziert werden kann. Diese ID eines Strings entspricht dem Index, an welchem er sich in der Java-Collection befindet. Hierbei starten die IDs jedoch am Index 1. Da Java-Collections jedoch beim Index 0 beginnen, befindet sich an dieser Stelle eine Art „Platzhalter-“ Eintrag um diese Verschiebung des Indexes zu kompensieren. Im Falle des Hinzufügens von neuen Strings gibt es eine Variable im StringPool, welche die neuen Strings zunächst zwischenspeichert. Diese ist notwendig, da neue Strings zunächst keine ID erhalten, sondern diese beim Schreiben des Inhalts in eine Datei zugewiesen werden.

2.2.5 Chunk

Chunks sind Klassen, welche dabei helfen sollen, Felder korrekt zu serialisieren. Chunks sind für das Parsen von Felddaten notwendig, da diese Informationen über die Anzahl der Instanzen, sowie den Index des ersten Bytes der ersten Instanz enthalten. Auch findet sich der Index des letzten (nicht mehr zu lesenden) Bytes der letzten Instanz in einem Chunk wieder. Chunks werden in der Regel nur in FieldDeclarations benötigt, jedoch unter Anderem in BasePools gelesen und verändert.

2.2.6 FieldType

Ein FieldType ist ein generischer Typ, welcher dem Zweck dient Felder in Typen zur Laufzeit zu Repräsentieren. Ein FieldType stellt zudem die Funktionen zur Verfügung, welche für das Lesen und Schreiben notwendige Informationen bereitstellen. Hierzu gehört auch das speichern einer so genannten TypeID, welche auch hier der Identifikation dient.

2.2.7 FieldDeclaration

Eine FieldDeclaration dient unter Anderem als Beschreibung eines Feldes. Hierfür ermöglicht sie es, den Typ dieses Feldes (in Form eines „FieldTypes“) sowie dessen Name festzulegen. Eine FieldDeclaration implementiert ähnlich wie ein Access-Objekt eine Funktion

namens „owner()“. Diese gibt aber keinen Zustand (wie es bei einem Access-Objekt der Fall ist) sondern ein Access-Objekt zurück. Dabei handelt es sich um jenes Objekt, zu welchem dieses Feld (beziehungsweise diese FieldDeclaration) gehört. Auch FieldDeclarations machen von Reflection Gebrauch und implementieren entsprechende „get-“ und „set-“ Funktionen. Anders als bei Klassen vom Typ SkillObject werden hier jedoch keine "FieldDeclarations", sondern SkillObjects als Parameter erwartet. FieldDeclarations erleichtern zusätzlich das Serialisieren von Daten. Hierfür existieren verschiedene Funktionen, welche sich mit dem Lesen und Schreiben von Daten, sowie der Berechnung dafür notwendiger Werte (wie zum Beispiel Offsets) befassen. Eine große Rolle spielen dabei die „Chunks“, welche die für die Berechnungen nötigen Daten enthalten.

2.2.8 StoragePool

Der StoragePool implementiert das Access-Interface und stellt folglich eine Collection, welche SkillObjects hält, dar. Hierbei entspricht der Name des StoragePools dem Namen des durch ihn repräsentierten Typs. Genauer gesagt ist der StoragePool ein Objekt, welches einen Typ und seine Instanzen (in Form der Collection) im Hauptspeicher repräsentiert und die Verwaltung dieser ermöglicht. Ein StoragePool bildet folglich die Grundlage für das Arbeiten mit Typen sowie deren Instanzen in einem Zustand. Aus der Repräsentation eines Typs durch einen StoragePool geht durch selbigen auch die Typ-Hierarchie des Typs hervor. Das heißt, dass jeder StoragePool seinen „BasePool“ kennt. Der BasePool stellt den obersten Typ der Hierarchie dar und ist im Falle, dass der StoragePool selbst der oberste Typ ist leer. In Java wäre dies der Typ „object“. Die Typen unterhalb können in einer Liste von so genannten „SubPools“ gefunden werden. „SubPools“ und „BasePools“ sind Klassen, welche die Klasse StoragePool erweitern und später noch beschrieben werden. Natürlich muss der StoragePool auch seinen direkten oberen Nachbarn (Supertyp) in der Hierarchie kennen. Der Supertyp ist derjenige Typ, von welchem dieser StoragePool ableitet und ist dem Pool als „SuperPool“ bekannt, wobei dieser ebenfalls ein StoragePool ist. Ein StoragePool muss nicht unbedingt Instanzen enthalten, sondern kann auch leer sein. In diesem Fall steht die Existenz des Pools dafür, dass der entsprechende Typ dem Zustand bereits bekannt ist. In einem StoragePool wird zwischen so genannten „statischen Daten“ und neuen Objekten unterschieden. „statische Daten“ sind Instanzen, welche entweder aus einer Datei ausgelesen wurden oder nach dem nachträglichen Hinzufügen bereits in eine Datei geschrieben wurden. Alle diese Instanzen besitzen eine eindeutige SkillID und werden von der Funktion „typeOrderIterator“, welche vom Access Objekt vorgeschrieben wird, in Form eines iterierbaren Objekts zurückgegeben. Diese Funktion ermöglicht den Zugriff auf alle Instanzen dieses Typs, wobei hier sowohl die statischen (bereits geschriebenen) Instanzen als auch die neuen Instanzen erfasst werden. Dabei werden nicht nur die Instanzen dieses StoragePools, sondern auch diejenigen seiner SubPools erfasst. Es gibt auch die Möglichkeit einzelne Instanzen eines StoragePools abzurufen. Hierfür benötigt man die SkillID der gewünschten Instanz und übergibt diese der Funktion „getByID“, welche die zu der ID gehörende Instanz zurückgibt. Instanzen, welche neu hinzugefügt werden sollen, werden zunächst in eine separate Liste gelegt und erhalten dort die SkillID -1. Dieser Wert identifiziert das

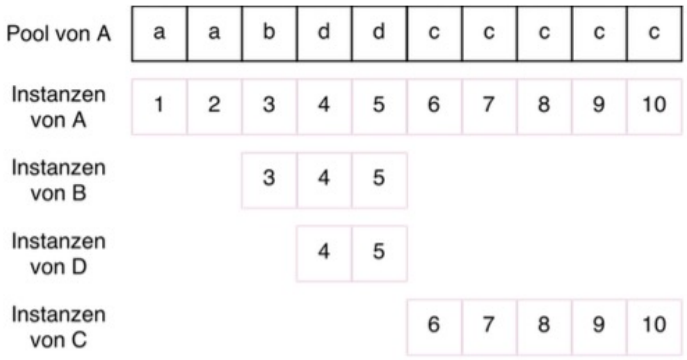


Abbildung 4: Instanzen von SubTypes eines BasePools
Quelle: [Ung14, Kapitel 1.3.3, Abbildung 1.2]

Objekt als neu und wird beim Schreiben der Datei und damit verbundenem vergeben der IDs geändert.

2.2.9 BasePool

Ein BasePool ist eine den StoragePool erweiternde Klasse. Der BasePool soll die oberste Ebene der Typ-Hierarchie darstellen. Eine Hauptfunktionalität eines BasePools ist die so genannte „compress-“ Funktion. Diese startet den Vorgang des Einfügens neuer Instanzen zu den statischen Daten und ist folglich fester Bestandteil des Schreibvorganges bei Zuständen im Modus Write. Anschließend wird durch diese Funktion eine Aktualisierung aller SkillIDs gestartet, um potentielle Verschiebungen nach unten zu propagieren. Eine Weitere wichtige Funktionalität des BasePools ist die „prepareAppend“-Funktion. Diese Funktion ist für das Hinzufügen von neuen Instanzen optimiert und betrachtet aus Effizienzgründen auch ausschließlich neue Instanzen. Diese Funktion wird beim Schreiben einer Datei, welche im Modus Append geöffnet wurde, aufgerufen. In dieser Funktion werden folglich nur neue SkillIDs vergeben und nicht wie bei compress alle SkillIDs aktualisiert.
Die Instanzen von Untertypen (SubPools) eines BasePool werden zusätzlich auch im BasePool gehalten. Das heißt einem BasePool ist zu jeder Zeit bekannt, welche Instanzen sich unter ihm in der Typ-Hierarchie befinden. In der Abbildung 4 ist ein Beispiel hierfür abgebildet. In dieser Abbildung sind die Instanzen A des BasePool A zu sehen sowie die Instanzen von drei seiner Untertypen B, C und D. Dabei fällt auf, dass der BasePool A auch die Instanzen der SubPools besitzt.

Die „compress“-Funktion Die Funktion „compress“ wird beim Schreiben von Zuständen im Modus Write verwendet. Diese Funktion durchläuft mithilfe des „typeOrderIterator“ alle statischen und neuen Instanzen. Dabei werden diese in ein neues Array gelegt, welches die Größe der beiden Listen besitzt. In diesem Zuge werden auch die SkillIDs al-

ler Instanzen (zum Teil) neu vergeben. Im Anschluss daran wird das bisherige Array der Daten mit dem hierbei erstellten überschrieben. Im Gegenzug zu der „prepareAppend“-Funktion für Zustände im Modus Append ist diese Funktion etwas langsamer, macht aber an sich aktuell noch das Gleiche, da die Funktionalität des Löschens in compress noch nicht implementiert wurde.

Die „prepareAppend“-Funktion Die Funktion „prepareAppend“, welche beim Schreiben von Zuständen im Modus Append verwendet wird ist auf das Hinzufügen neuer Instanzen optimiert. Diese Funktion überprüft ausschließlich das Vorhandensein neuer Instanzen und fügt diese entsprechend zu den statischen Daten hinzu. Hierbei wird zunächst ein neues Array erstellt, welches der Größe der bisherigen Daten zuzüglich der Anzahl neuer Instanzen entspricht. In dieses Array werden als erstes alle statischen Daten eingefügt. Im Anschluss daran wird nur über die Liste der neuen Instanzen iteriert und diese nacheinander in das neue Array eingefügt. Hierbei wird jeder Instanz eine aufsteigende ID, beginnend bei der Größe der bisherigen Liste der statischen Daten, zugewiesen. Anschließend wird entsprechend die Aktualisierung der zum Schreiben nötigen Daten durchgeführt. Auch hier werden alle Änderungen auch in den SubPools angepasst. Dieser Umstand, dass die Aktualisierung nur für neue Instanzen durchgeführt und nach unten propagiert werden muss, bringt mehr Effizienz beim Hinzufügen von Daten.

2.2.10 SubPool

Ein SubPool ist ähnlich wie der BasePool eine den StoragePool erweiternde Klasse. Diese Klasse ist dazu da, dass Instanzen korrekt verwaltet werden. Damit ist gemeint, dass jede Instanz welche in einem SubPool (unabhängig von der Ebene) hinzugefügt wird auch im BasePool hinzugefügt werden muss. Diese Funktionalität übernimmt der SubPool, zusätzlich zu den Funktionen, welche er vom StoragePool erbt.

3 Verwaltung von Zuständen

Zum jetzigen Zeitpunkt wird in der SKilL-Implementierung nur das Erstellen neuer Instanzen unterstützt. Hierbei wird eine neue Instanz eines bestimmten Typs generiert und diese mit Standard-Werten belegt, welche dann anschließend geändert werden können. Die Implementierung unterstützt jedoch leider noch keine wirkliche Verwaltung von Instanzen. Das heißt, dass das aktuell keine Instanzen effizient und ohne großen Aufwand kopiert oder gelöscht werden können. Auch die Suche nach einer Instanz in einem Zustand ist nur sehr aufwendig möglich. Diese Funktionalitäten wurden in dieser Arbeit untersucht und eingebaut und werden in den nachfolgenden Abschnitten beschrieben. An dieser Stelle sei angemerkt, dass die Änderungen durch Verwaltungsaktionen in einem Zustand nicht in die ursprüngliche Datei geschrieben werden müssen. Die Funktion „changePath“ ermöglicht das Speichern von Änderungen in andere Dateien, welche der Nutzer selbst angeben kann.

3.1 Ziel der Verwaltung

Ziel der Verwaltung von Zuständen ist in erster Linie natürlich das Löschen und Kopieren von Instanzen, um Daten einfach und unkompliziert aus Zuständen zu entfernen oder von einem in den nächsten Zustand zu kopieren. Für das Löschen und Kopieren ist es auch sinnvoll im voraus zu überprüfen, ob eine Instanz mit dem selben Inhalt nicht bereits vorhanden ist beziehungsweise dieses Vorhandensein der Instanz möglichst schnell festzustellen. Hierfür wird auch ein so genannter „Element-Operator“ benötigt der folglich ebenfalls Teil der Verwaltung ist. Wie in der Abbildung 5 zu sehen ist, ist eine solche Funktionalität bereits durchaus möglich, ist jedoch sehr aufwendig, da einige Funktionen zum Schutze des Nutzers (wie zum Beispiel die HashMap „poolByName“) diesem vorenthalten werden. Das heißt, dass das die fehlende Implementierung dieser Funktionalität auf Zustandsebene (und zum Teil auch auf Ebene der StoragePools), an dieser Stelle eine manuelle Implementierung dessen durch den Nutzer notwendig macht. Dies würde dann aber bedeuten, dass der Nutzer sämtliche Listen und Inhalte durchsuchen und vergleichen und hierfür einen sehr hohen Aufwand aufbringen muss. Dieser Umstand soll nun verbessert werden, indem dem Nutzer hierfür Funktionen innerhalb eines Zustandes bereitgestellt werden, welche von den Zustands-internen Funktionen Gebrauch machen, aber diese dem Nutzer nach wie vor nicht öffentlich zugänglich machen.

Den Nutzer vor sich selbst schützen Damit der Nutzer keinen all zu tiefen Einblick in die Funktionsweise von SKilL benötigt, werden viele Funktionen und Eigenschaften vor ihm verborgen. Diese laufen „im Hintergrund“ ab und der Nutzer muss sich nicht mit ihnen befassen. Diese Herangehensweise wird in dieser Arbeit, soweit möglich, fortgesetzt und dient auch dem Schutz des Nutzers. Denn die unsachgemäße Verwendung oder Veränderung dieser Elemente könnte Daten und Zustände unbrauchbar machen. Aus diesem Grund werden aktuelle und sinnvolle Sachverhalte erörtert um dadurch ein klares Verhalten zu ermöglichen und Fehlerquellen so gut wie möglich einzudämmen. Ein Beispiel wäre hier die Manipulation der HashMap „poolByName“, deren Verände-

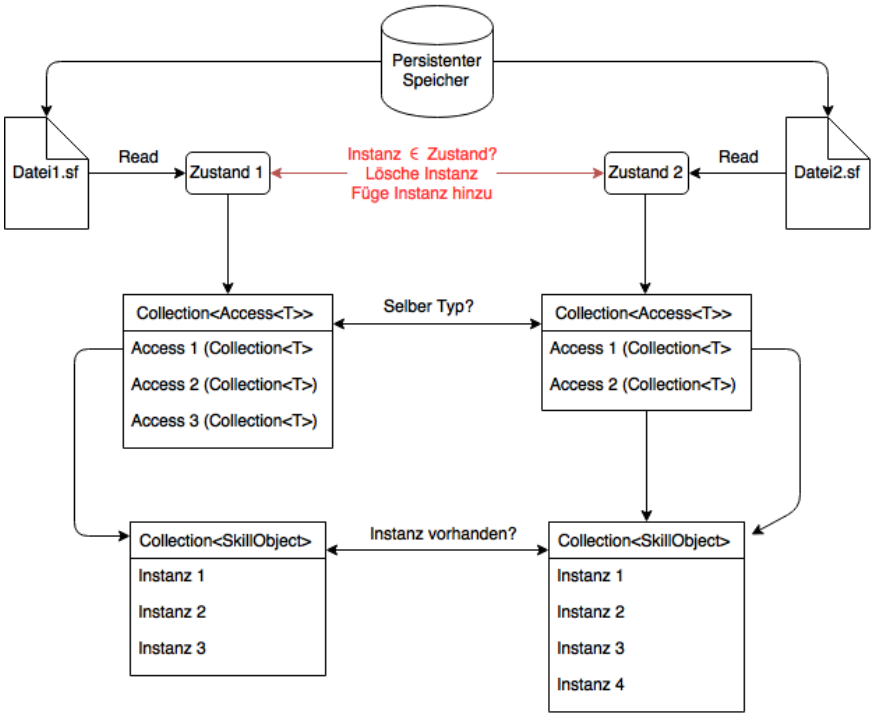


Abbildung 5: Ziel der Verwaltung

rung ohne Anpassung aller damit verbundener Abhängigkeiten (beispielsweise Typen Erstellung) unnötig Fehler entstehen lassen könnte. Dieses Vorhaben impliziert, dass die Möglichkeit, dass ein Nutzer in dieser Hinsicht eigene Entscheidungen trifft, auf einem Minimum gehalten werden.

3.2 Kritische Stellen der Verwaltung

Die Verwaltung von Zuständen bringt einige kritische Stellen mit sich, welche beachtet werden müssen. Besonders im Hinblick auf Effizienz ist es sehr ratsam, diese Stellen zu betrachten und sich genau zu überlegen, wie man mit diesen umgeht. Aus diesem Grund werden im folgenden Abschnitt die relevanten Stellen angesprochen und erklärt. Die Lösung dieser Probleme wird in den jeweiligen Abschnitten der „Umsetzung der Verwaltung“ erklärt und besprochen.

3.2.1 Suchen einer Instanz

Effiziente Suche Für das Suchen einer Instanz kann man naiv sämtliche Listen durchsuchen und deren Inhalte Vergleichen. Für eine effizientere Suche nach Instanzen wäre es jedoch wünschenswert, wenn man diesen Aufwand vermeidet. Zustände bieten hier eine interne Funktion an, welche einen Typ (StoragePool) über seinen Namen sucht. Hierfür wird eine HashMap verwendet, bei welcher das Wiederfinden der Werte nur über

Schlüssel sehr effizient möglich ist. [Ull10, Kapitel 13.8.2, Absatz 1] Hat man den zu der Instanz zugehörigen Typ gefunden, kann man in diesem prüfen, ob eine Instanz mit den selben Werten bereits existiert. Auch hier ist die selbe naive Suche möglich. Doch jede Instanz besitzt eine SkillID, welche sie eindeutig identifiziert und ein StoragePool ist in der Lage eine Instanz über ihre ID schnell zu finden. Denn die SkillID steht unter Anderem für den Index im Array der statischen Daten. An dieser Stelle stellt man jedoch zunächst fest, dass SkillIDs nur im jeweiligen Typ eines Zustandes eindeutig und einzigartig sind. SkillIDs werden in einem anderen Typ des selben Zustands bekanntlich unabhängig voneinander aufsteigend vergeben. Selbiges gilt natürlich Zustands übergreifend. Die effiziente Suche nach Instanzen über deren SkillID stellt an dieser Stelle folglich noch ein Problem dar.

3.2.2 Löschen einer Instanz

Das Löschen einer Instanz, welches ein vorheriges Überprüfen der Existenz einer Instanz voraussetzt, hat auch ein paar kleinere Hindernisse im Bezug auf die Umsetzung. Die wichtigsten zwei werden nachfolgend kurz beschrieben.

Verschiebung der Blöcke Auch beim Löschen von Instanzen gibt es einige Dinge zu beachten. Das Löschen einer Instanz hat zur Folge, dass durch die gelöschte Instanz eine Verschiebung der Blöcke innerhalb einer Datei entsteht sobald dieses Löschen abgespeichert wird. Diese Verschiebung führt beim schreiben aber zunächst zu einem Fehler, da die fehlende Instanz in den Chunks der Felder zum jetzigen Zeitpunkt nicht angepasst wird und folglich am Ende eine Instanz an einer Stelle erwartet wird, an welcher es keine mehr geben kann.

Existenz der zu löschenden Instanz Beim Löschen einer Instanz stellt sich zunächst die Frage, existiert diese überhaupt? Nun wurde im Abschnitt über die Suche nach Instanzen festgestellt, dass eine effiziente Suche nach Instanzen aktuell nicht möglich ist. Aus Effizienzgründen möchte man aber nicht bei jedem löschen mehrere Male überprüfen müssen, ob ein angesteuerter Typ oder eines der Felder nicht vielleicht ins Leere zeigt oder andere Fehlerquellen mit sich bringt. Dies führt zu dem Schluss, dass im Zuge des Löschens zunächst über die HashMap des Zustands die Existenz des Typs und anschließend in diesem die Existenz der Instanz sichergestellt werden soll um den Vorgang im Anderen Fall (fehlerfrei) abbrechen zu können. Damit verbunden unterliegt also das Löschen von Instanzen den gleichen Voraussetzung wie die Suche nach selbigen.

3.2.3 Kopieren einer Instanz

Existenz der Typen Das Hauptproblem beim Kopieren von Instanzen ist die Existenz der Typen. Da ein Zustand mithilfe einer bestimmten Spezifikation eingelesen wird, kennt dieser Zustand nur diejenigen Typen, welche in der Spezifikation vorhanden sind. Möchte man nun also Instanzen eines Typs einer anderen Spezifikation zu einem Zustand hinzufügen, muss man in dem Zielzustand zunächst den Typ bekannt machen.

3.3 Umsetzung der Verwaltung

Die allgemeine Umsetzung der Funktionalitäten hat sich als sehr Komplex herausgestellt. Das Arbeiten mit Typen und Instanzen erweist sich als recht schwer, da zum Beispiel auch eine Zustands unabhängige Erstellung und Handhabung dieser denkbar wäre, auch wenn dies zum aktuellen Zeitpunkt noch nicht unterstützt wird. Unabhängig hiervon gibt es viele Kriterien, welche beachtet und erfüllt sein wollen, wenn man mit Typen und ihren Instanzen arbeitet. Man muss beispielsweise im Falle des nicht Vorhandenseins eines Typs in einem Zustand diesen dort „erstellen“ oder diesen bei Bedarf aus einer Spezifikation nachladen. Nun werden Zustände und die darin enthaltenen Typen aber zunächst anhand ihrer Spezifikation erstellt, welche dem neuen Zustand nicht vorliegt. Hier wäre dann Zugriff auf die Spezifikation des aufrufenden Zustands notwendig, wodurch erneuter Aufwand und weitere Hindernisse entstehen. Man könnte nun den Typ (als StoragePool im Ganzen) in den neuen Zustand kopieren und hier all diejenigen Informationen (oder Instanzen) heraus nehmen, welche nicht kopiert werden sollen. Solche Informationen können zum Beispiel Restriktionen sein, welche einen Wert in einen bestimmten Bereich einschränken. Ein solcher Vorgang bringt abgesehen von hohem Aufwand aber auch viele fehleranfällige Dinge mit sich wie zum Beispiel die Notwendigkeit, dass der Verweis auf den Besitzer eines Pools geändert werden muss. Dies muss natürlich auf alle Super- und SubPools propagiert werden. Im Zielzustand selber muss die Liste der vorhanden Typen aktualisiert werden ebenso wie die damit zusammenhängende HashMap erweitert werden muss um Pools durch ihren Namen effizient abfragen zu können. Zusätzlich müssen neue Felder in den Zustand übertragen, sowie die Namen dieser in den StringPool eingetragen werden. Hierbei ist es ohne die Spezifikation sehr aufwendig und fehleranfällig die notwendigen Informationen zusammenzutragen und an den korrekten Stellen einzufügen. Bei alle dem sollte man darauf achten, dass aufgrund des Arbeitens mit Referenzen keine Fehler im ursprünglichen Zustand entstehen und der Zustand hierdurch unbrauchbar wird. Dabei stellt sich dann die Frage, ob man entsprechende Typen (oder gar Zustände) in Form einer „deepcopy“ kopieren soll. All diese Überlegungen und die damit verbundenen Hindernisse und Bedingungen sprengen den Rahmen dieser Arbeit. Aus diesem Grund wurde die Umsetzung der Verwaltung basierend auf bestimmten Annahmen durchgeführt, um die gewünschte Funktionalität zunächst auf „simplerer Ebene“ umzusetzen und hierauf anschließend aufzubauen. Den einfachsten Fall bilden bei der Verwaltung von Typen die jeweiligen Namen dieser.

3.3.1 Annahmen

Gleichheit von Typen Alle Zustände, mit welchen im folgenden gearbeitet wird, sind aus ein und der selben Spezifikation entstanden. Jeder Zustand kennt folglich jeden Typ und alle gleichnamigen Typen sind auch gleich!

Keine unbekannten Typen Da nur mit einer einzigen Spezifikation gearbeitet wird, gibt es in keinem der Zustände Typen, welche einem anderen Zustand unbekannt sind.

Keine Instanz ohne Zustand Es wird davon ausgegangen, dass zum jetzigen Zeitpunkt keine Instanzen ohne Zustand existieren.

3.3.2 Unifikation von Zuständen

Da für die Effizienz der nachfolgenden Aktionen die Zugriffe über SkillIDs Zustands übergreifend möglich und die IDs hierfür einzigartig sein müssen gilt es zunächst diese Eigenschaft sicherzustellen. Aus diesem Grund wird die Durchführung einer Unifikation zweier Zustände zuerst beschrieben. Zu Beginn sei an dieser Stelle kurz angemerkt, dass sich die Unifikation nicht mit Restriktionen von Typen auseinandersetzt, da dies aufgrund der Annahme, dass alle Typen bekannt sind und gleichnamige Typen äquivalent sind, nicht nötig ist. Um nun die Zugriffe über SkillIDs Zustands übergreifend zu ermöglichen, werden zwei Zustände so wie ihre Inhalte unifiziert. Dies bedeutet jedoch, dass in einem der beiden Zustände Daten verändert werden müssen, sofern es sich nicht um ein und denselben Zustand handelt. Diese Veränderung hat zur Folge, dass in demjenigen Zustand, in welchem etwas verändert wurde, Inkonsistenzen auftreten, welche bestimmte Funktionalitäten fehleranfällig machen.

Zerstörung eines Zustands Bei einer Unifikation wird durch die Zerstörung des übergebenen Zustands in Kauf genommen, dass seine SkillIDs verändert werden, um die Vergleichbarkeit mit dem anderen Zustand zu ermöglichen. Dies hat zur Folge, dass in diesem zerstörten Zustand keine Instanzen über die „getByID“-Funktion gefunden werden können. Dieser Umstand wirkt sich auch auf den später beschriebenen „Element-Operator“ und folglich auch die Funktionalität des Hinzufügens und Löschens von Instanzen aus. Aufgrund der Tatsache, dass eine Unifikation von jeglicher Verwaltungsaktion erwartet wird und vor dieser notwendig ist, wird an dieser Stelle natürlich auch die Effizienz bei der Unifikation in den Vordergrund gestellt. Aus diesem Grund wurde beschlossen auf einer Referenz des Zustands zu arbeiten, weil dies natürlich wesentlich schneller ist als zuvor eine „deepcopy“ dessen zu erstellen. Hierbei wird bei der Unifikation zwangsweise in einem der Zustände etwas geändert (in diesem Fall in Form von geänderten SkillIDs). Sobald jedoch eine SkillID geändert wurde, läuft man Gefahr, dass ein Zugriff mit dieser auf die vermeintliche Instanz in einen Fehler mündet, da die SkillID bei einer Unifikation zum einen höher werden kann, als tatsächlich Instanzen in dem Zustand vorhanden sind und zum anderen diese Instanz mit der neuen ID nicht mehr findet. Dies würde also entweder zu einem „IndexOutOufBounds“ Fehler führen, da die Liste der Instanzen nicht so lang ist oder es könnte womöglich auch zum Abrufen einer anderen (nicht gewollten) Instanz führen. Aus diesem Grund wurde festgelegt, dass ein neuer Modus für Zustände eingeführt wird, welcher einen Zustand als „Destroyed“ markiert. Diese Markierung hat zur Folge, dass die „getByID“-Funktion dieses Zustands prinzipiell „null“ zurückgibt, da sie unter der SkillID ihrer Instanzen möglicherweise nicht mehr die korrekte Instanz finden oder damit gar auf Indizes zugreifen könnte, welche nicht existieren. Da aber das Schreiben eines Zustandes keinen Gebrauch dieser Funktion macht, ist dies nach wie vor zulässig. Denn die IDs aller Instanzen werden beim Schreiben ohne hin aufsteigend und unabhängig von den bisherigen IDs (neu) vergeben, solange sich der Zustand nicht

im „Append“-Modus befindet. Da aber ein nachträgliches wechseln zu diesem Modus in keinem Falle zulässig ist, kann man hier per Definition davon ausgehen, dass bei einem zerstörten Zustand immer die Funktionalität des „Write“-Modus verwendet wird und somit IDs unabhängig von den alten aufsteigend vergeben werden.

Zuweisung (neuer) SkillIDs Die Zuweisung neuer SkillIDs bei der Unifikation orientiert sich an der Zuweisung von SkillIDs der SKilL-Implementierung selbst. Diese erfolgt aufsteigend, ausgehend von der letzten (gültigen) SkillID. Dabei wird die höchste vergebene SkillID der Instanzen eines internen Typs als letzte gültige SkillID verwendet. Aus diesem Grund wird während der Unifikation eben jene SkillID über die Größe der Liste der statischen Daten des internen Typs ermittelt und die SkillIDs der externen Instanzen werden davon ausgehend inkrementiert. Grund hierfür ist, dass der aufgerufene Zustand nicht verändert werden darf und dieser somit nur als Vergleichsobjekt dienen kann. Folglich ist es zwingend notwendig, dass ausschließlich die SkillIDs der externen Instanzen verändert werden.

Das Unifikationsverfahren Zunächst wird die Funktion „UnifyStates“ eines Zustands aufgerufen, welcher ein weiterer Zustand übergeben werden muss. Diese Funktion leitet nun die Unifikation der Zustände ein, wobei der übergebene Zustand als zerstört markiert wird. Der übergebene Zustand wird im weiteren Verlauf dieser Beschreibung als „externer Zustand“ bezeichnet. Als erstes wird über die Typen (StoragePools) des externen Zustands iteriert. Bei jeder Iteration wird der aktuelle Typ des externen Zustands mit dem äquivalenten Typ des ausführenden Zustands unifiziert. Hierfür wird der entsprechende Typ durch die Nutzung der „poolByName“ HashMap, in welcher nach dem Namen des Typs des externen Zustands gesucht wird, abgefragt. Aufgrund der Annahme, dass es keine unbekannten Typen gibt ist eine Prüfung auf einen Zeiger ins Leere an dieser Stelle (noch) nicht nötig. Nun wird als erstes die nächste zu vergebende SkillID ermittelt indem die size-Funktion der Liste der statischen Daten des Typs aus dem aufrufendem Zustand aufgerufen wird. Sollte hierbei festgestellt werden, dass der interne Typ keine statischen Instanzen beinhaltet, wird an dieser Stelle bereits abgebrochen, da keine Unifikation dieses Typs notwendig sein kann. Im Anschluss wird auch der externe Typ auf das Vorhandensein von statischen Instanzen überprüft und sollte dies hier nicht der Fall sein, kann an dieser Stelle ebenfalls abgebrochen werden. Falls es nicht zum Abbruch kommt, wird zunächst über alle statischen Instanzen des externen Typs und innerhalb dieser Iteration über diejenigen des internen Typs iteriert. Diese ineinander geschachtelte Schleife ist notwendig, da man zum einen überprüfen muss, ob es inhaltlich identische Instanzen in den beiden Typen gibt, deren SkillIDs sich unterscheiden und folglich auf die selbe ID gesetzt werden müssen. Zum Anderen muss man sicherstellen, dass eine neue SkillID nur dann vergeben wird, wenn für eine Instanz des externen Typs tatsächlich keine inhaltlich identische im internen Typ existiert. Daher ist es unvermeidbar, dass an dieser Stelle jede Instanz mit jeder verglichen wird. Die einzige Ausnahme ist, dass (frühzeitig) eine identische Instanz gefunden wird, wodurch die innere Schleife abgebrochen werden kann.

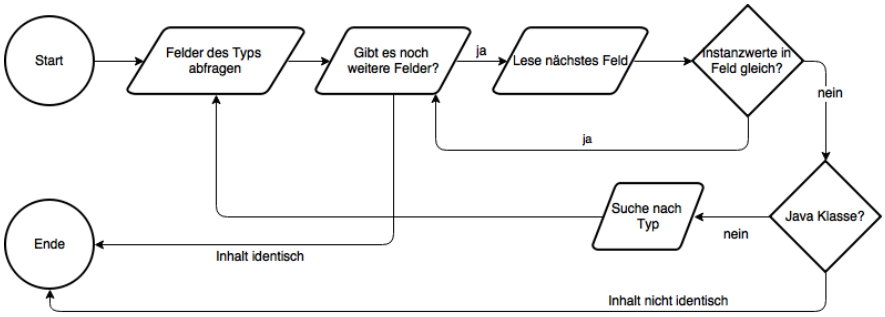


Abbildung 6: Vergleich zweier Instanzen (vereinfachter Ablauf)

Der inhaltliche Vergleich zweier Instanzen erfolgt über einen Aufruf einer Funktion, welche explizit für den Vergleich von Feldern zweier Instanzen existiert und rekursiv implementiert ist. In dieser Funktion werden die Werte der Instanzen für alle Felder ihres Typs abgerufen und verglichen. Sollten die Werte hier nicht übereinstimmen, stellt sich die Frage nach dem FieldType, mit welchem der Wert abgerufen wurde. Genauer gesagt ist die vom FieldType instantiierte Klasse relevant. Wird vom FieldType die selbe Klasse (zum Beispiel eine Java-Klasse wie „long“) instantiiert, dann liegen hier simple Typen vor, welche nicht den selben Wert haben. In diesem Fall wird der Vergleich abgebrochen und es kann zum Vergleich der nächsten Instanzen weiter gegangen werden. Andernfalls kann es sich an dieser Stelle um Komplexe Typen handeln deren Inhalt nicht ohne Weiteres über die „equals“ Funktion getestet werden kann. Aus diesem Grund erfolgt nun ein Vergleich der FieldTypes, welche hier repräsentiert werden. Kommt dieser Vergleich zu dem Schluss, dass es sich nicht um die selben Klassen handelt, wird auch hier der Vergleich abgebrochen und zum nächsten Vergleich fortgeschritten. Liegen jedoch die selben (komplexen) Klassen vor, erfolgt an dieser Stelle die Abfrage nach dem entsprechenden StoragePool und ein rekursiver Aufruf, welcher nun die Inhalte der gefundenen Instanzen des gefundenen komplexen Typs mit den Werten der Felder dieses Typs abgleicht. Diese Rekursivität wird fortgeführt, bis entweder die erhaltenen Werte der Instanzen für alle Felder gleich sind oder es sich bei allen Feldern um Java-Klassen handelt, deren Werte nicht alle gleich sind. Im ersten Fall wird der Vorgang beendet mit der Feststellung, dass die SkillID der externen Instanz gleich der ID der internen Instanz gesetzt werden muss, da es sich um inhaltlich gleiche Instanzen handelt. Im zweiten Fall hingegen wird eine Anpassung der SkillID der externen Instanz durchgeführt, indem sie die nächste verfügbare (also im internen Typ noch nicht vergebene) ID erhält. Durch diese Rekursivität wird sichergestellt, dass alle Untertypen der zu vergleichenden Instanz ebenfalls korrekt verglichen werden und somit in den „Gesamtvergleich“ der Instanzen einfließt. Dieser Ablauf des Vergleichens von Instanzen wird in Abbildung 6 vereinfacht illustriert. Nachdem alle Instanzen eines Typs verglichen wurden sind deren SkillIDs anschließend in den beiden unifizierten Zuständen in den jeweiligen Typen einzigartig. Eine Änderung der SkillID im externen Typ bleibt nur genau dann aus, wenn zwei Instanzen sowohl inhaltlich als auch im Bezug auf ihre SkillID gleich sind.

3.3.3 Element-Operator für Instanzen

Die Funktionalität des Suchens nach Instanzen wird über eine Funktion realisiert, welche ein SkillObject (Instanz) entgegen nimmt und nach dieser in demjenigen Zustand sucht, in welchem sie aufgerufen wurde. Das Ergebnis dieser Funktion ist ein Wert des Typs java.lang.long. Dies hat den Hintergrund, dass mit der Rückgabe eines Wertes, welcher nicht „null“ ist, auch gleich die SkillID der Instanz in dem entsprechenden Typ zurückgegeben wird. Dabei entspricht eine SkillID gleich -1 einer neuen (noch nicht geschriebenen) Instanz und eine ID gleich 0 einer beim nächsten schreiben zu löschenden Instanz. Alle Werte echt größer 0 stellen diejenige ID dar, unter welcher die Instanz in dem Zustand zu finden ist. Im Falle, dass die Instanz nicht gefunden werden kann wird wie bereits erwähnt „null“ zurückgegeben. Für den Element-Operator ist im voraus eine Unifikation der zwei betroffenen Zustände durchzuführen. Dabei wird in dem zu durchsuchenden Zustand die entsprechende Funktion aufgerufen, welche den Zustand, aus welchem die zu suchenden Instanzen kommen, übergeben bekommt. An dieser Stelle sei angemerkt, dass der Element-Operator in als „Destroyed“ markierten Zuständen nicht korrekt funktionieren kann, da hier die Zugriffe über SkillIDs nicht fehlerfrei möglich sind. Aus diesem Grund ist das Ergebnis einer Abfrage in einem zerstörten Zustand per Definition „null“.
Bei der zu findenden Instanz muss es sich um eine Klasse handeln, welche die SKilL-Klasse SkillObject erweitert. Zunächst wird nach einem der Instanz entsprechenden Typ in Form eines StoragePools gesucht. Um diesen zu finden, wird die HashMap „poolByName“ verwendet, welche nun den Namen der Klasse der Instanz entgegen nimmt. Um den genauen Namen zu erhalten, wird zunächst die „getClass“-Methode der Instanz aufgerufen. Das hierdurch erhaltene Java-Class-Object stellt nun verschiedene Funktionen zur Verfügung, welche genauere Details der hier genutzten Klasse preis geben. Für die Suche nach dem entsprechenden Typ ist die Funktion „getSimpleName“ interessant, welche wie der Name schon andeutet, schlicht den Namen der Klasse zurückgibt. Da Namen von Typen in SKilL prinzipiell in Kleinbuchstaben vorliegen, muss nun auf den erhaltenen String noch eine Konvertierung in Kleinbuchstaben erfolgen. Nun hat man den Namen des Typs wie er in dem Zustand (sofern er dort vorhanden ist) zu finden ist. An dieser Stelle wurde bewusst auf das explizite Übergeben des Namen des Typs verzichtet, da für SKilL auch geplant ist, dass Instanzen unabhängig von Zuständen existieren können wodurch diese ihren Typ nur auf die oben beschriebene Weise preisgeben können. Im Anschluss daran wird überprüft, ob ein dem Typ der Instanz entsprechender StoragePool im Zustand gefunden wurde. Ist dies nicht der Fall, so gibt die Funktion „null“ zurück. Andernfalls wird die Suche nach der Instanz in dem gefundenen StoragePool fortgesetzt. Für die Suche im StoragePool wurde die von der Java-Collection geerbte „contains“-Funktion implementiert und wird ihrem Zweck entsprechend genutzt. Diese überprüft zunächst, ob es sich bei dem übergebenen Objekt um ein SkillObject handelt. Diese Überprüfung ist an dieser Stelle notwendig, da die contains-Funktion zunächst einen Parameter vom Typ „Object“ entgegen nimmt. Dies ist bei der Vererbung so vorgegeben. Da diese Funktion aber überall dort verwendet werden kann, wo StoragePools verwendet werden, muss vor einer Konvertierung (in Form eines Casts) sichergestellt

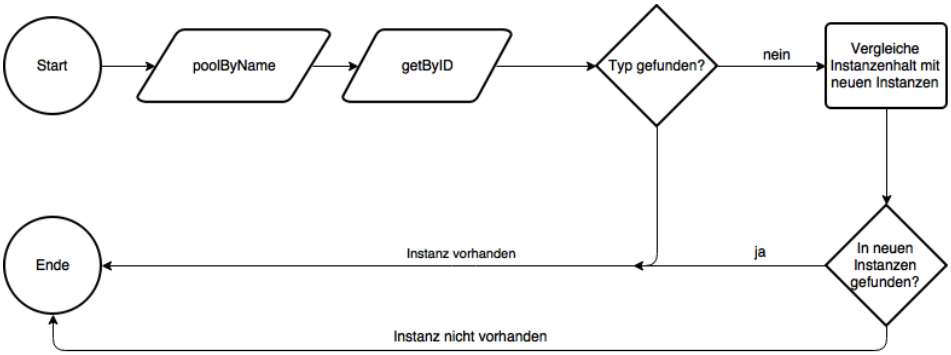


Abbildung 7: Element-Operator (vereinfachter Ablauf)

werden, dass der erwartete Typ übergeben wurde. Ist dies nicht der Fall, so wird an dieser Stelle „false“ zurückgegeben, ansonsten erfolgt eine entsprechende Konvertierung. Im Anschluss daran wird entschieden, ob nach der Instanz in den statischen Daten oder den neuen Daten gesucht werden muss, wofür die SkillID zur Hand genommen wird. Sollte sich die Instanz innerhalb der statischen Daten befinden, so darf die SkillID zum einen nicht gleich -1 sein und zum anderen nicht größer als die Größe der Liste der statischen Daten. Denn die SkillID dient bekanntlich zugleich als Index in der Liste. Befindet sich die ID also im Bereich zwischen 0 und der Größe der Liste, so kann die Existenz (oder nicht Existenz) der Instanz durch die Funktion „getByID“ nachgewiesen werden. Hierbei gilt zu beachten, dass diese Funktion nach einer Unifikation in dem Zustand, welcher zerstört wurde, per Definition „null“ zurückgibt. Sind die aufgeführten Kriterien nicht erfüllt, so muss die Instanz in den neuen Daten gesucht werden. Für die Effizienz der suche nach Instanzen in den statischen Daten wird sich hier zu nutze gemacht, dass das SKilL-Binding Arrays und HashMaps nutzt. Diese bieten schnelle get-Operationen und werden in der „getByID-“ und der „poolByName-“ Funktion genutzt. [Prz14, Kapitel 9, Absatz 1]

Suche in neuen Daten Die Suche nach einer Instanz in neuen Daten ist etwas aufwendiger, da hier nicht einfach die SkillID verwendet werden kann, denn hier sind alle SkillIDs gleich -1. Aus diesem Grund müssen an dieser Stelle die Werte der einzelnen Felder der Instanzen verglichen werden. Dies wird über eine Iteration über die Felder des Typs und des entsprechenden Vergleichs der Werte der Instanzen in diesen realisiert. Dabei wird das Feld für die SkillID bewusst übersprungen, da dieses ohne hin gleich -1 ist. Für alle anderen Felder wird als erstes der Wert verglichen und im Falle, dass dieser nicht gleich ist, erfolgt eine Überprüfung, ob es sich hierbei bereits um simple Typen handelt. Sofern es sich um simple Typen (also im Prinzip um Java-Typen wie zum Beispiel „long“ handelt, kann an dieser Stelle abgebrochen werden mit dem Ergebnis, dass es sich nicht um die selben Instanzen handelt. Handelt es sich jedoch um komplexe Typen, so müssen diese Rekursiv ebenfalls verglichen werden, denn der Vergleich über „equals“ schlägt an dieser Stelle fehl. Grund dafür ist, dass die native „equals“-Funktion

die Repräsentation des Objekts vergleicht, welche im Fall von Komplexen Typen aus dem Namen und Hashcode besteht. Diese Rekursion wird durch eine Suche nach einem zugehörigen Typ (StoragePool), ähnlich wie die Suche nach der Instanz begonnen hat, durchgeführt. Hierbei wird ausgenutzt, dass jeder StoragePool seinen Zustand, in welchem er sich befindet, kennt. Denn auf diese Weise kann auch hier auf die HashMap des Zustands zurückgegriffen und in dieser nach dem hier gefundenen komplexen Typ gesucht werden. Kann dieser nicht gefunden werden, wird die Suche nach der Instanz für gescheitert erklärt. Sofern der komplexe Typ gefunden werden konnte, wird schlicht die contains-Funktion dessen für den Vergleich des Inhalts der Instanzen verwendet, welche zu diesem komplexen Typ gehören. Diese Rekursion wird solange fortgeführt, wie komplexe Typen in den Instanzen gefunden werden. Da die Blätter jeder Typ-Hierarchie in dieser Java-Implementierung ihre Werte über das Java Typ System repräsentieren, kommt diese Rekursion immer zum Ende.

3.3.4 Löschen von Instanzen

Ein wesentlicher Bestandteil der Verwaltung von Zuständen ist das Löschen von Instanzen in diesen. Die Grundidee für das Löschen von Instanzen war schon immer, dass die SkillID dieser auf 0 gesetzt und hierdurch beim Schreiben in eine Datei nicht mehr berücksichtigt wird. Im wesentlichen fehlte die Realisierung des Ignorieren der Instanz beim Schreiben der Datei. Aus diesem Grund wird nun zunächst beschrieben, wie die zu löschende Instanz gefunden und als solche markiert werden kann. Im Anschluss daran wird die nötige Anpassung beim Schreiben der Datei erklärt.
Die Änderungen durch das Löschen von Instanzen in einem Zustand müssen nicht zwingend in die ursprüngliche Datei, aus welcher der Zustand entstanden ist, geschrieben werden. Über die „changePath“-Funktion kann eine andere (gegebenenfalls neue) Zieldatei angegeben werden, in welche die Änderungen geschrieben werden sollen.

Markieren einer Instanz Das Löschen einer Instanz durch Aufruf der entsprechenden Funktion in einem Zustand funktioniert zunächst ähnlich wie der Element-Operator. Das heißt, dass auch hier eine vorherige Unifikation der betroffenen Zustände nach dem bekannten Prinzip erforderlich ist. Abbildung 8 stellt den Ablauf des Löschens einer Instanz, wie er im folgenden beschrieben wird, vereinfacht dar. Es wird anhand der von SkillObject abgeleiteten Klasse nach dem Typ in Form eines StoragePools gesucht. Sollte ein solcher nicht gefunden werden können, schlägt das Löschen an dieser Stelle fehl und gibt „false“ zurück. Da jedoch laut der aktuellen Annahme alle Typen bekannt sind, wird dieser Aufruf nie mit leeren Händen enden, weshalb in Abbildung 8 dies auch nicht weiter ausgeführt wurde. Im anderen Fall wird die von der Java-Collection geerbte und aus diesem Grund selbst implementierte Funktion „remove“ des StoragePools aufgerufen. Auch hier muss aufgrund der Tatsache, dass die vererbte Funktion ein Objekt des Typs „java.lang.object“ erwartet zunächst eine Überprüfung der instantiierten Klasse erfolgen. Sollte diese fehlschlagen, wird der Vorgang mit der Rückgabe von „false“ abgebrochen. Andernfalls erfolgt auch hier ein Cast zu SkillObject um mit dessen SkillID nun über die „getByID“-Funktion an diejenige Instanz zu kommen, welche gelöscht werden soll.

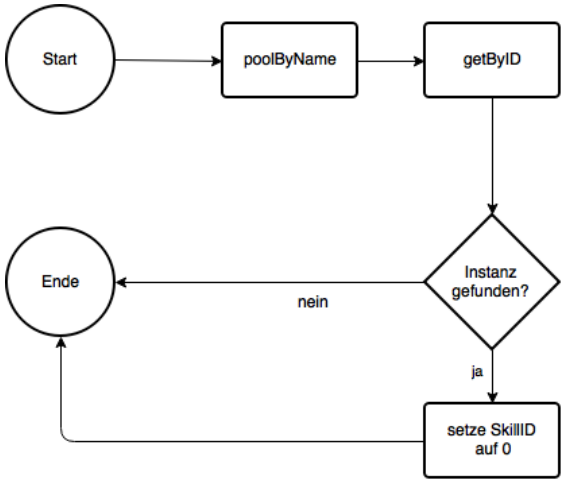


Abbildung 8: Löschen einer Instanz (vereinfachter Ablauf)

Sollte diese nicht existieren, erfolgt die Rückgabe „false“ und andernfalls wird die ID der erhaltenen Instanz auf 0 gesetzt und „true“ zurückgegeben. Auch hier gilt zu beachten, dass diese Funktion in zerstörten Zuständen nie eine Instanz finden kann.

Löschen der markierten Instanzen Beim Schreiben einer Datei wird für jeden BasePool des Zustands dessen „compress“-Funktion aufgerufen. Diese ist bisher dafür zuständig gewesen, neue Instanzen in die Liste der statischen Daten einzufügen und hierbei die entsprechenden SkillIDs zu aktualisieren. Da diese Funktion beim Schreiben eines Typs immer aufgerufen wird, ist sie der optimale Ort um hier die zu löschenden Instanzen zu entfernen beziehungsweise zu ignorieren.
Zunächst wird also zu Beginn des „compress“ Vorganges über die Liste der statischen Daten (von vorne nach hinten) iteriert und alle Instanzen mit einer SkillID gleich 0 entfernt werden. Dies wird dadurch bewerkstelligt, dass die Liste durchlaufen wird und parallel dazu eine Variable für den Index, unabhängig von dem Durchlauf, erstellt wird, welche bei 0 startet. Bei denjenigen Elementen, deren SkillID nicht gleich 0 ist, wird nun besagtes Element an der Stelle des Indexes in die Liste der statischen Daten eingefügt und das dort befindliche Element überschrieben. Nach diesem Überschreiben eines Elements wird der Index um 1 erhöht. Folglich wird im Falle, dass kein Element gelöscht wird, jedes Element mit sich selbst überschrieben. Andernfalls verschieben sich nach jedem gefundenen zu löschenden Element alle nachfolgenden Elemente um eine Weitere Stelle nach vorne. Im Anschluss an eine potentielle Verschiebung müssen nun natürlich noch die überflüssigen (beziehungsweise nun unter Umständen doppelt vorkommenden) Instanzen entfernt werden. Auch hierfür dient der Index, welcher zuvor mitgeführt wurde. Ist dieser nämlich echt kleiner als die Größe der Liste der statischen Daten, so müssen alle Elemente ab diesem Index gelöscht werden. Es wird also folglich solange das Element an der Stelle des Indexes gelöscht, bis die Größe der Liste nicht mehr echt Größer als der Index ist.

Im Falle, dass Elemente gelöscht wurden, müssen nun noch die Chunks der Felder dieses Typs angepasst werden, denn diese besitzen noch die Information, dass keine Instanzen gelöscht wurden und erwarten entsprechend mehr zu schreibende Instanzen als davon existieren. Hierfür wird über die Liste der Felder des Typs iteriert und für jedes Feld ein neuer Chunk erstellt, welcher die aktuelle Anzahl an Instanzen kennt. Alle anderen Informationen eines Chunks können an dieser Stelle von dem letzten vorhandenen Chunk übernommen werden, da diese später abhängig von der Anzahl der Instanzen neu berechnet werden. Aus diesem Grund muss an dieser Stelle nur die Anzahl der Instanzen aktualisiert werden.

Aktualisierung der SkillIDs Nachdem nun die zu löschenden Instanzen entfernt und entsprechende Daten aktualisiert wurden müssen nun noch die SkillIDs der verbleibenden Instanzen aktualisiert werden. Grund dafür ist, dass sich durch das Löschen unter Umständen Lücken bei den IDs gebildet haben, welche nun entsprechend angepasst werden müssen. Dies wird über die restliche (unveränderte) Funktionalität der „compress“-Funktion erledigt, da diese nun Mithilfe des „typeOrderIterator“ alle statischen und neuen Instanzen durchläuft und ihnen neue (aufsteigende) IDs zuweist. Der Umstand, dass nur die „compress“-Funktion zum jetzigen Zeitpunkt beim aktualisieren der IDs alle Instanzen aktualisiert setzt den Modus Write bei jeder Löschung von Instanzen voraus! Das Löschen von Instanzen auch für den Modus Append zu ermöglichen ist nicht sehr ratsam. Grund dafür ist, dass man die Optimierung des Hinzufügens neuer Instanzen dabei kaputt machen müsste, da man durch das Löschen gezwungener Maßen alle Instanzen anschauen muss. Würde man beim Löschen nicht alle Instanzen anschauen und entsprechend aktualisieren könnten Lücken in den SkillIDs und somit Inkonsistenzen und Fehler entstehen. Sollte man jedoch beim Öffnen eines Zustandes bereits entschieden haben nur neue Instanzen hinzuzufügen, stellt Append eine deutlich effizientere Variante dar.

„Dangling references“ Unter einer dangling reference (oder auch dangling pointer) versteht man eine Referenz (Zeiger) auf ein Objekt, welches nicht mehr verwendet wird und somit gelöscht werden kann. [Ull10, Kapitel 5.5.6] Dieses wird durch die bestehende Referenz allerdings nicht vom Garbage-Collector erkannt und folglich auch nicht eingesammelt. Eine solche Referenz wird in SKilL dadurch vermieden, dass alle Instanzen einer Typ Hierarchie im BasePool liegen und von den Subtypen lediglich referenziert werden. Da der BasePool nach jedem compress alle SubPools aktualisiert, werden die Referenzen an dieser Stelle auch entfernt. Auf diese Weise wird sichergestellt, dass keine so genannten „dangling references“ entstehen.

Löschen von Typen Sollte durch die Funktionalität des Löschens von Instanzen die letzte Instanz eines Typs gelöscht werden, war eine Überlegung, ob man an dieser Stelle nicht auch den Typ aus dem Zustand entfernt. Das Löschen eines Typs hätte aber zur Folge, dass man diesen unter Umständen in einem weiteren Schritt (zum Beispiel beim

Kopieren von Instanzen) wiederherstellen müsste. Der Aufwand für das Wiederherstellen eines Typs ist deutlich höher, als dieser in dem entsprechenden Zustand Platz verbraucht. Aus diesem Grund wird das Löschen von Typen an dieser Stelle nicht ermöglicht.

Mengensubtraktionsoperation für Typen Um alle Instanzen eines Typs eines Zustandes aus einem anderen Zustand zu subtrahieren wurde die Funktion „deleteInstancesOfType“ implementiert, welche einen Typ in Form eines StoragePools als Parameter entgegen nimmt. Diese Funktion iteriert nun über alle in diesem Typ vorhanden Instanzen und entfernt diese aus dem aufgerufenen Zustand indem die Funktion „deleteInstance“ für alle Instanzen aufgerufen wird. Aus diesem Grund ist auch bei dieser Operation eine Unifikation im voraus strikte Voraussetzung. In diesem Zuge wurde auch eine Funktion eingebaut, welche eine Liste von Typen entgegen nimmt und das entfernen von Instanzen verschiedener Typen in einem Aufruf ermöglicht. In dieser Funktion wird nun jeder Typ an die bereits beschriebene Funktion „deleteInstancesOfType“ übergeben. Der Nutzer muss an dieser Stelle selbst sicherstellen, dass in den übergebenen Typen nur diejenigen Instanzen vorhanden sind, welche im Zielzustand auch tatsächlich gelöscht werden sollen.

3.3.5 Kopieren von Instanzen

Ein weiterer Aspekt dieser Arbeit ist die Möglichkeit des Kopierens von Instanzen. Auch hierfür wurde eine Funktion in die Implementierung des Zustands integriert, welche den Prozess des Kopierens der übergebenen Instanz einleitet. Für das Hinzufügen von mehreren Instanzen in einem Aufruf wurde eine Funktion hinzugefügt, welche eine Liste von Instanzen entgegen nimmt. Diese Funktion iteriert dann durch diese Liste und übergibt jede Instanz an die entsprechende „add“-Funktion im Zustand. Die Rückgabe der Funktion ist nur dann true, wenn alle Instanzen hinzugefügt werden konnten.
Die „add“-Funktion eines Zustandes macht sich zunächst den Element-Operator zu Nutze, aus welchem Grund auch hier die korrekte Funktionsweise von der vorherigen Unifikation abhängt. Dies bedeutet aber auch, dass das Hinzufügen neuer Instanzen in einem Zustand, welcher sich im Modus „Destroyed“ befindet nicht erlaubt ist, da hier nicht gewährleistet werden kann, dass keine Instanz doppelt in einen Zustand eingefügt wird. Grund dafür ist, dass der Element-Operator in zerstörten Zuständen nicht korrekt funktionieren kann.
Beim Hinzufügen von Instanzen zu einem Zustand wird auch mithilfe der von SkillObject abgeleiteten Klasse über die HashMap poolByName nach dem Typ gesucht. Diese abgeleitete Klasse wird für die Bestimmung des Namens des Typs benötigt. Sollte der benötigte Typ nicht gefunden werden können, wird der Vorgang hier mit der Rückgabe „false“ abgebrochen. Ansonsten wird der Aufruf an den gefundenen StoragePool weitergeleitet, indem eine speziell hierfür implementierte Funktion namens „addInstance“ aufgerufen wird, welche letztendlich die Instanz an die von der Collection geerbte Funktion „add“ weiterleitet. Hierbei überprüft die geerbte Funktion „add“ lediglich, ob der StoragePool den Status „fixed“ besitzt. Im Falle, dass dies zutrifft, wird ein Fehler geworfen, da keine Instanzen zu fixierten Pools hinzugefügt werden dürfen. Andernfalls

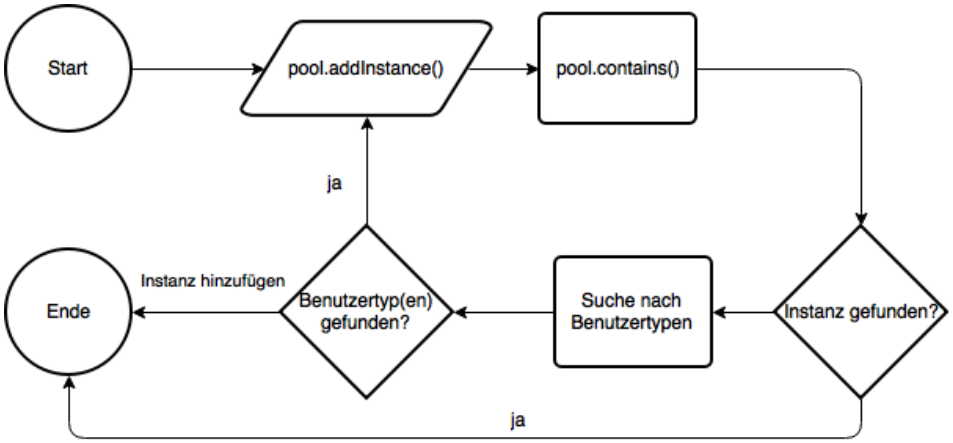


Abbildung 9: Hinzufügen einer Instanz (vereinfachter Ablauf)

fügt sie die Instanz einfach der Liste der neuen Objekte zu.
Der Grund für die Notwendigkeit der Funktion „addInstance“ ergibt sich aus der Tatsache, dass ein StoragePool generische Typen besitzt, wobei einer davon von der „add“-Funktion als Parameter erwartet wird. Das heißt, dass bei Klassen, welche von generischen Klassen erben, die geerbten Funktion mit den eindeutigen Typen implementiert werden müssen und an dieser Stelle keine Wildcard mehr verwendet werden kann. Stattdessen muss der genaue Typ an dieser Stelle verwendet werden. Eine Wildcard, welche in Java als „?“-Operator realisiert ist, repräsentiert eine Familie von Typen. Dabei ist wichtig zu verstehen, dass ? nicht für Object steht, sondern für einen (bisher) unbekannten Typ! [Ull10, Kapitel 9.5.3, Absatz 5] Da ein Zustand diesen Typ nicht kennt, ist ein Cast an dieser Stelle nicht möglich. Ein Weiterer Grund für die Nutzung einer Zwischenfunktion ist, dass auch hier eine Notwendigkeit für eine Rekursion besteht, welche in der „addInstance“-Funktion realisiert ist. Aus diesem Grund wird diese Funktion separat beschrieben. In Abbildung 9 ist die Nutzung der „addInstance“-Funktion beim Hinzufügen neuer Instanzen vereinfacht dargestellt.
Im Bezug auf jegliche Nutzung der Funktionalität des Hinzufügens von Instanzen ist es nicht zwingend notwendig, dass das Ergebnis in die Datei geschrieben wird, aus welcher der Zustand entstanden ist. Es gibt in Zuständen die Möglichkeit über die „changePath“-Funktion die Zieldatei zu ändern und sämtliche Änderungen in eine andere (gegebenenfalls neue) Datei zu schreiben.

Die „addInstance“-Funktion Diese Funktion überprüft zunächst, ob sich die übergebene Instanz nicht bereits in dem StoragePool befindet und würde falls dies zutrifft mit der Rückgabe „false“ abbrechen. Ansonsten wird durch diese Funktion die übergebene Instanz nach komplexen Typen durchsucht, welche separat hinzugefügt werden müssen. Hierfür werden alle Felder der Instanz betrachtet und sofern es sich um einen komplexen Typ handelt, wird der zugehörige Typ gesucht. Hierfür wird sich erneut die „owner“-Funktion zu Nutze gemacht, über welche man an den besitzenden Zustand herankommt

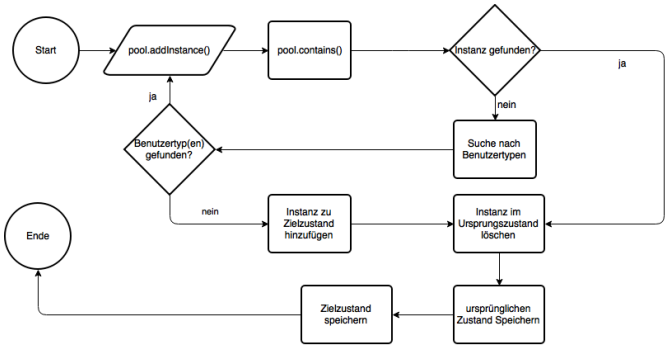


Abbildung 10: Verschieben einer Instanz (vereinfachter Ablauf)
Der ursprüngliche Zustand muss gespeichert werden, bevor der Zielzustand gespeichert wird.

um die HashMap dessen für die Suche zu verwenden. Sollte die Suche fehlschlagen, so wird zum jetzigen Zeitpunkt der Vorgang mit dem Werfen einer Ausnahme abgebrochen, um das Erstellen von inkonsistenten und fehlerhaften Zuständen zu vermeiden. Nach der aktuellen Annahme wird diese Ausnahme nie geworfen, denn der Fall, dass Typen unbekannt sind, wurde im voraus für diese Arbeit ausgeschlossen, da die Erstellung eines Typs etwas komplexer ist. Grund für die Ausnahme ist, dass im Falle, dass eine Instanz zwei komplexe Typen enthält, der rekursive Aufruf des ersten bereits abgeschlossen sein kann während der des Zweiten eventuell einen Fehler feststellt. In diesem Fall wären Instanzen bereits hinzugefügt worden, welche nicht hätten hinzugefügt werden dürfen. Nachdem der Pool gefunden wurde, wird nun mit dem betroffenen Komplexen Typ (welcher an sich eine Instanz darstellt) die Funktion „addInstance“ rekursiv aufgerufen.

Transfer von Instanzen Unter dem Transfer von Instanzen wird eine Verschiebung dieser von einem in einen anderen Zustand verstanden. Das heißt, dass die Instanz aus demjenigen Zustand, aus welchem sie ursprünglich kommt nach dem Kopieren in dem ursprünglichen Zustand nicht mehr vorhanden ist. Dieser Transfer ist aufgrund der notwendigen Unifikation der Zustände ein wenig komplizierter. Grund dafür ist, dass die Instanz, welche als Referenz übergeben wird, in zwei Zuständen verändert werden muss. Das Verschieben einer Instanz ist zwar prinzipiell möglich, aber die Funktion, welche die Verschiebung anordnet, muss sicherstellen, dass die Instanz aus dem Zustand, aus welchem sie verschoben werden soll, als erstes gespeichert wird. Wird der Zielzustand zuerst gespeichert, so wird die SkillID der Instanz auf einen Wert ungleich 0 gesetzt, wodurch sie nicht mehr als „zu Löschen“ markiert ist. In Abbildung 10 ist der Ablauf für das Verschieben von Instanzen dargestellt. In dieser Abbildung wird der Ablauf des Speicherns der beiden Zustände verdeutlicht. Diese Besonderheit beim Verschieben von Instanzen schränkt die Möglichkeit des Transfers von diesen auch auf nur eine Richtung ein. Das heißt, dass es nicht möglich ist eine Instanz C von Zustand A in Zustand B und vor dem nächsten Schreiben der Zustände eine Instanz D von Zustand B in Zustand A zu verschieben.

Erstellen eines konsistenten Zustands Um aus bestehenden Instanzen einen konsistenten Zustand zu erstellen, muss mit der für die Instanzen notwendigen Spezifikation zunächst ein Zustand im Modus „Create“ erstellt werden. In diesen Zustand können nun alle Instanzen kopiert werden, indem die Funktion für das Hinzufügen von Instanzen aufgerufen wird. Um dies zu bewerkstelligen, kann man entweder alle Instanzen einzeln an die Funktion „addInstance“ oder alle Instanzen als Liste an die Funktion „addInstances“ übergeben.

Mengenvereinigungsoperation für Typen Um alle Instanzen eines Typs zweier Zustände zu vereinigen existiert die Funktion „addInstancesOfType“, welche einen StoragePool als Parameter entgegen nimmt. Diese Funktion iteriert nun über alle in diesem Typ vorhanden Instanzen und fügt sie dem Typ des aufgerufenen Zustandes hinzu. Hierfür wird die Funktionalität des Hinzufügens von Instanzen, welche im Laufe dieser Arbeit entstanden ist, genutzt. Dabei gilt natürlich, dass vor dem Ausführen dieser Operation eine Unifikation der Zustände stattgefunden haben muss. An dieser Stelle greift auch die für diese Arbeit getroffene Annahme, dass alle Typen bereits bekannt sind. Des Weiteren existiert auch eine Funktion um die Instanzen mehrerer Typen einem Zustand hinzuzufügen. Diese nimmt entsprechend eine Liste von StoragePools entgegen, aus welcher sie alle StoragePools einzeln heraus nimmt und an die oben beschriebene Funktion „addInstancesOfType“ übergibt. Es gilt dabei zu beachten, dass der Nutzer bevor er die entsprechende Funktion aufruft sicherstellen muss, dass nur die Instanzen in den Typen vorhanden sind, welche auch in der Vereinigungsmenge vorhanden sein sollen. Damit verbunden muss der Nutzer sich darum kümmern, dass sowohl im Typ des Zielzustands als auch in dem des Ursprungszustandes nur diejenigen Instanzen befinden, welche Teil der Vereinigungsmenge werden sollen.

4 Mögliche nächste Schritte der Verwaltung

In diesem Abschnitt werden einige Ideen vorgestellt, welche im Rahmen dieser Arbeit nicht mehr näher untersucht und umgesetzt werden konnten. Bei diesen Ideen handelt es sich zum einen um Optimierungen in Sachen Zustandsverwaltung und zum Anderen um notwendige nächste Schritte um die Nutzbarkeit dieser zu Verbessern.

4.1 Unifikation

Einmalige Unifikation In der aktuellen Implementierung der Unifikation ist es durchaus möglich, dass zwei Zustände mehrfach miteinander unifiziert werden. Dies könnte man zum Beispiel dadurch verbessern, dass sich jeder Zustand merkt, mit welchem anderen Zustand er bereits unifiziert wurde. Als Kriterium zur Identifikation wäre hier zum Beispiel der Pfad der Datei, aus welcher der Zustand entstanden ist, denkbar. Denn der Modus „Destroyed“ sagt an dieser Stelle nichts darüber aus, mit welchem Zustand unifiziert wurde.

Verhindern einer Unifikation Im Laufe dieser Arbeit wurde festgestellt, dass es unter Umständen durchaus Anwendungsfälle gibt, bei welchen eine Unifikation (und die damit verbunden Zerstörung) nicht sinnvoll ist. Ein Beispiel hierfür wäre, dass Änderungen in drei verschiedenen Zuständen (A, B und C) gemacht werden sollen. Im Falle, dass sich Abhängigkeiten wie das verschieben von Instanzen von A nach B, C nach A und der Vereinigung eines Typs von B in C ergeben, muss hier sichergestellt werden, dass dies ohne auftretende Inkonsistenzen abläuft. Bei diesem Szenario müssten beim Ausführen aller Aktionen alle Zustände zerstört werden. Man sieht also, dass diese Aktionen zunächst in einer bestimmten Reihenfolge ablaufen müssen, um die Konsistenz von Zuständen zu bewahren und dennoch das gewünschte Resultat zu erhalten. Es wäre hierfür wünschenswert, dass ein Zustand eine Unifikation zunächst ablehnen kann, bis er alle hierfür kritischen Abschnitte überwunden hat und wieder bereit für eine neue Unifikation ist. Aktuell muss die Konsistenz der Zustände bei einem solchen Szenario durch den Nutzer sichergestellt werden.

Vermeiden der Zustandszerstörung Eine Unifikation zerstört aktuell den übergebenen Zustand, da nach dem Ändern der SkillIDs in diesem der korrekte Zugriff auf Instanzen über die IDs nicht mehr sichergestellt werden kann. Diesen Umstand könnte man vermeiden, indem ein Mechanismus entwickelt wird, welcher bei der Unifikation zum Beispiel den Offset zwischen der ursprünglichen ID und der neuen ID speichert und mithilfe dessen im zerstörten Zustand trotzdem die entsprechende Instanz finden kann. Kritisch ist hierbei, dass sichergestellt werden muss, dass dieser Offset nur im zerstörten Zustand eingerechnet wird, nicht jedoch in demjenigen Zustand mit welchem er unifiziert wurde, da die Einrechnung dessen hier natürlich zu einem falschen Zugriff führen würde.

4.2 Unbekannte Typen

Unifikation Ohne die Annahme, dass es keine unbekannten Typen gibt, gilt es natürlich auch zu bedenken, dass Instanzen von solchen unbekannten Typen in einem Zustand hinzugefügt werden sollen. In einem solchen Fall muss bei der Unifikation unter Anderem zusätzlich noch beachtet werden, dass diese beim Vergleich von Typen auch deren Anzahl an Feldern vergleicht. Denn im Falle von unbekannten Typen besteht auch die Gefahr, dass zwei gleichnamige Typen auftreten, welche aber nicht die selbe Datenstruktur darstellen. Neben dem Vergleich des Inhalts von Typen muss also auch die Definition des Typs selbst verglichen werden. An dieser Stelle kommen auch die TypeIDs und eine potentielle Unifikation dieser für die Unifikation in Frage, welche nach einem ähnlichen Schema wie die SkillIDs unifiziert werden könnten.

5 Zusammenfassung und Ausblick

Im Laufe dieser Arbeit ist die Grundlage für die Funktionalität des Kopierens, Löschens und Suchens von Instanzen entstanden. Die Arbeit beschränkt sich auf den einfachsten Fall dieser genannten Funktionalitäten, da die Abdeckung aller Fälle den vorgegebenen Rahmen dieser Arbeit überschreiten würde.

Unter der Annahme, dass es nur eine Spezifikation gibt, in welcher keine unbekannten Typen existieren und das es keine Instanzen ohne Zustand geben kann, wurden die in der Aufgabenstellung genannten Funktionen umgesetzt. Es wurde eine Möglichkeit geschaffen nach Instanzen in Zuständen zu suchen, sowie solche Instanzen zu entfernen. Auch das Kopieren von Instanzen aus einem Zustand in einen anderen Zustand wurde realisiert. Zudem wurde in Zusammenhang mit dem Kopieren von Zuständen das Verschieben dieser implementiert. Im Rahmen dieser Arbeit wurde hierfür die Notwendigkeit einer Unifikation von Zuständen festgestellt um diese Funktionen effizient zu ermöglichen. Die Durchführung einer solchen Unifikation wird folglich von allen hier entwickelten Funktionen, beziehungsweise für deren korrekte Funktionsweise, vorausgesetzt. Ein Nachteil, welcher für die Effizienz der implementierten Funktionen in Kauf genommen wurde, ist das zerstören des übergebenen Zustands bei der Unifikation. Der übergebene Zustand wird hiernach zwar unbrauchbar im Bezug auf die Suche von Instanzen, alle weiteren implementierten Aktionen in dem nicht zerstörten Zustand sind jedoch aufgrund von Zugriffen über Indizes und HashMaps wesentlich effizienter möglich. Die Instanzen in dem dabei zerstörten Zustand können dennoch für das Kopieren dieser verwendet werden. Die Tatsache, dass der Zustand zerstört wurde soll lediglich vermeiden, dass fehlerhafte Zugriffe über die SkillIDs durchgeführt werden können.

Der nächste Schritt ist nun nach und nach die komplizierteren Fälle zu untersuchen und die hier implementierte Funktionalität in soweit zu erweitern, dass auch diese Fälle abgedeckt werden. Hierzu gehört unter Anderem das Nachladen und Hinzufügen von fehlenden Typen in einen Zustand. Der Hauptbestandteil hierbei ist das Ermitteln der hierfür nötigen Spezifikationen und eine entsprechende Auswertung dieser im Hinblick darauf, was tatsächlich benötigt wird.

Literatur

[Fel13] Timm Felden. The skill language. Technical report, 2013.

[Har14] Fabian Harth. Plattform- und sprachunabhaengige serialisierung mit skill. Diplomarbeit, Universitaet Stuttgart, 2014.

[Prz14] Dennis Przytarski. Performance-evaluation einer sprach- und plattformunabhaengigen serialisierungssprache. Bachelorarbeit, Universitaet Stuttgart, 2014.

[Ull10] Christian Ullenboom. Java ist auch eine insel, 2010.

[Ung14] Wladislaw Ungur. Nutzbarkeitsevaluation einer sprach- und plattformunabhaengigen serialisierungssprache. Diplomarbeit, Universitaet Stuttgart, 2014.

[XML06] Extensible markup language, 2006.

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