Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Implementierung von änderungstoleranten Enumerationen in SKilL

Sarah Sophie Stieß

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Dr. Timm Felden
Beginn am:	1. Februar 2018
Beendet am:	1. August 2018

Kurzfassung

Diese Arbeit befasst sich mit der Implementierung von enum-Typen in SKilL. Die bisherige Implementierung wird durch eine neue ersetzt, in der enum-Typen keine Felder mehr haben und auf enum-Typen der Zielsprache abgebildet werden. Diese Abbildung hat das Problem, dass die enum-Typen der Zielsprachen nicht änderungstolerant sind, Änderungstoleranz aber eine zentrale Eigenschaft von SKilL ist. In dieser Arbeit wird die Lösung dieses Problems beschrieben. Die Arbeit beschreibt, welche Auswirkungen die gefundene Lösung auf die bereits vorhandenen SKilL-Komponenten hat. Es wird beschrieben, welche von ihnen weiter verwendet werden können und welche verändert beziehungsweise neu hinzugefügt werden müssen. Die Arbeit beschreibt außerdem eine entsprechende Implementierung in Java und C++. Die neuen enum-Typen erfordern Änderungen an den Code-Generatoren, die von dieser Arbeit ebenfalls beschrieben werden. Die entstandene Implementierung von enum-Typen in SKilL wurde ausgiebig getestet. Eine Beschreibung der durchgeführten Tests ist auch Teil dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 SKilL . . . 8
- 1.2 Begriffe und Notationen . . . 8
- 1.3 Änderungstoleranz . . . 9

2 Design **11**
- 2.1 Implikationen aus dem Binärformat . . . 11
- 2.2 `enum`-Felder . . . 12
  - 2.2.1 Verwendung von `int` als Feldtyp . . . 13
  - 2.2.2 Verwendung von `String` als Feldtyp . . . 15
  - 2.2.3 Verwendung eines eigenen Feldtyps . . . 16

3 Implementierung in Java **19**
- 3.1 `EnumProxy` . . . 19
- 3.2 `EnumPool` . . . 19
- 3.3 Anpassungen bereits vorhandener Bibliotheksklassen . . . 22
- 3.4 Generierter Code . . . 23
  - 3.4.1 Enumerationen . . . 23
  - 3.4.2 Nutzertypklassen . . . 23
  - 3.4.3 Pools der `enum`-Typen . . . 24
  - 3.4.4 Pools der Nutzertypen . . . 24
  - 3.4.5 Feldrepräsentanten der `enum`-Felder . . . 25
  - 3.4.6 Feldrepräsentanten der `enum`-Konstanten . . . 26
  - 3.4.7 Änderungen an `Parser` . . . 26
  - 3.4.8 Änderungen an `SkillState` . . . 27

4 Implementierung in C++ **29**
- 4.1 `EnumProxy` . . . 29
- 4.2 `EnumPool` . . . 30
- 4.3 Anpassungen bereits vorhandener Bibliotheksklassen . . . 32
- 4.4 Generierter Code . . . 33
  - 4.4.1 Enumerationen . . . 33
  - 4.4.2 Nutzertypklassen . . . 33
  - 4.4.3 Pools der `enum`-Typen . . . 35
  - 4.4.4 Pools der Nutzertypen . . . 36
  - 4.4.5 Feldrepräsentanten der `enum`-Felder . . . 37
  - 4.4.6 Feldrepräsentanten der `enum`-Konstanten . . . 38
  - 4.4.7 Änderungen an `makeState` . . . 39
  - 4.4.8 Änderungen an `makePool` . . . 39

4.4.9 Änderungen an `SkillFile` 39

5 Änderungen am Codegenerator **41**
5.1 Parser 41
5.2 Java Quellcode-Generator 42
5.3 C++ Quellcode-Generator 43

6 Container **45**

7 Anpassungen an der Sprachspezifikation **47**

8 Tests **49**
8.1 Verwendete Spezifikationen und Austauschdateien 49
8.2 Generierte Tests 51
8.2.1 API Tests 51
8.2.2 Lesetests 51
8.3 Manuell geschriebene Tests 51
8.3.1 EnumRestriction Tests 52
8.3.2 Read Tests 53
8.3.3 Write Tests 53
8.3.4 Append Tests 53
8.3.5 Access Tests 54
8.3.6 EnumProxy Tests 54
8.3.7 FieldType Tests 55
8.3.8 EnumPool Tests 55
8.3.9 ContainedEnums Tests 55
8.4 Codecoverage der Java Tests 56

9 Zusammenfassung **57**

Glossar **58**

Akronyme **59**

Literaturverzeichnis **60**

1 Einleitung

Mit Serialization Killer Language (SKilL)[Fel17a] können Instanzen von objektorientierten Typen sprachunabhängig serialisiert werden. Die serialisierten Typen und ihre Instanzen stehen dann in einer Austauschdatei, die von Anbindungen verschiedener Zielsprachen gelesen und wieder geschrieben werden kann.

Diese Anbindungen werden von Quellcode-Generatoren generiert. SKilL verfügt über Generatoren für verschiedene Zielsprachen. In einigen der Sprachen gibt es Enumerationen. Diese werden je nach Sprache anders dargestellt. So werden `enum`-Typen in Java in Klassen übersetzt (siehe [GJS+14] §8.9), wogegen sie in C++ benannte numerische Konstanten sind (siehe [ISO12] §7.2).

Auch in SKilL gibt es `enum`-Typen. Sie werden jedoch nicht auf `enum`-Typen der Zielspache, sondern auf normale Nutzertypen abgebildet. Für jeden `enum`-Typ wird ein Nutzertyp generiert und für die Konstanten jedes `enum`-Typs werden Subklassen dieses Nutzertyps generiert (siehe [Fel17a] §5.4.3). Das Ziel dieser Arbeit ist es, die Umsetzung der `enum`-Typen in SKilL zu verbessern.

Die Aufgabenstellung gibt vor, dass mit der neuen Implementierung die `enum`-Typen einer Spezifikation auf `enum`-Typen der Zielsprache abgebildet werden müssen. Außerdem müssen sie weiterhin änderungstolerant sein. Die neue Implementierung der `enum`-Typen soll für zwei Sprachen umgesetzt werden. Für diese Arbeit wurden die Sprachen Java und C++ ausgewählt.

Kapitel 2 befasst sich mit den Eigenschaften der Neuimplementierung der `enum`-Typen. Es befasst sich mit den dabei entstehenden zielsprachenunabhängigen Problemen und wie diese gelöst werden können. Kapitel 3 beschreibt die Implementierung der `enum`-Typen für Java. Kapitel 4 beschreibt sie für C++. Beide Implementierungen basieren auf den in Kapitel 2 getroffenen Entscheidungen. Kapitel 5 beschreibt die Veränderungen, die an der Grammatik des Parsers und der Zwischendarstellung des Codegenerators vorgenommen werden müssen, damit `enum`-Typen keine Felder mehr haben. Es beschreibt außerdem, wie der Java und der C++ Quellcode-Generator angepasst werden müssen, damit Anbindungen wie in Kapitel 3 und 4 beschrieben generiert werden. In Kapitel 6 werden `enum`-Typen in Containern betrachtet. `enum`-Typen in Containern erhielten ein eigenes Kapitel, da sie nur selten auftreten und nicht wie flache `enum`-Typen behandelt werden. Kapitel 7 beschreibt wie die Sprachspezifikation von SKilL an die neuen `enum`-Typen angepasst werden muss. Kapitel 8 geht auf die Tests der neuen Implementierung ein.

Am Ende dieser Arbeit befindet sich eine Zusammenfassung mit Ausblick. Die folgenden Abschnitten geben einen kurzen Einblick in SKilL, einen Überblick über die verwendeten Begriffe und Notationen und einen Überblick der für die Implementierung von `enum`-Typen relevanten Aspekte der Änderungstoleranz.

1.1 SKilL

SKilL in ausreichender Tiefe zu erklären würde den Rahmen dieser Arbeit sprengen. Dieser Abschnitt bietet lediglich einige oberflächliche Erläuterungen zu den Begriffen, die im Laufe dieser Arbeit besonders häufig auftauchen. Alle weiteren Details zu SKilL können in [Fel17a] Kapitel 5 und 6 nachgelesen werden.

SKilL beinhaltete einen Quellcode-Generator, der eine Spezifikation liest und dazu passenden Code in einer Zielsprache generiert. Außerdem gibt es Bibliotheken für verschiedene Zielsprachen. Der generierte Code ist spezifikationsabhängig, die Bibliotheken nicht. Zusammen ergeben der generierte Code und eine Bibliothek eine Anbindung, die von einem Werkzeugbauer verwendet werden kann, um Austauschdateien zu lesen und zu schreiben.

Wenn der Werkzeugbauer mit der Anbindung eine Austauschdatei liest, hat er nach dem Lesevorgang ein Zustandsobjekt, das alle der Anbindung bekannten und alle in der Austauschdatei enthaltenen Typen und Felder, sowie alle in der Austauschdatei enthaltenen Typinstanzen kennt. Er kann mit der Anbindung außerdem neue Zustandsobjekte erstellen. Diese kennen die der Anbindung bekannten Typen und Felder. Instanz der Typen gibt es noch keine. Über ein Zustandsobjekt kann der Werkzeugbauer auf Typen und Felder zugreifen. Er kann die Werte der Felder verändern oder neue Instanzen hinzufügen.

Die Instanzen von Typen werden von Pools verwaltet. Diese Pools sind auch die Repräsentanten ihrer Typen im SKilL-Laufzeittypsystem. Um Felder zu repräsentieren gibt es Feldrepräsentanten. Jeder Pool kennt alle Feldrepräsentanten, die Felder seines Typs repräsentieren. Pools und Feldrepräsentanten repräsentieren sowohl unbekannte als auch bekannte Typen und Felder. Sie haben außerdem Felddatenserialisierungsmethoden. Mit den Felddatenserialisierungsmethoden eines Feldrepräsentanten werden die Felddaten des repräsentierten Feldes serialisiert. Die Felddatenserialisierungsmethoden eines Pools serialisieren die Felddaten der Felder, deren Feldtyp der von diesem Pool repräsentierte Typ ist. Manche Feldrepräsentanten, insbesondere die von unbekannten Feldern, rufen die Felddatenserialisierungsmethoden eines Pools auf.

1.2 Begriffe und Notationen

In dieser Arbeit wird `typewriter font` für Klassen-, Methodennamen und andere Codeelemente verwendet. Methoden werden im Allgemeinen nur mit ihren Namen bezeichnet. Wenn mehrere Methoden denselben Namen tragen, werden weitere Teile der Signatur mit angegeben, damit die Methoden eindeutig identifiziert werden können. Logische Variablen werden *kursiv* geschrieben. In späteren Kapiteln dieser Arbeit werden die Klassen `EnumProxy` und `EnumPool` auftauchen. Instanzen dieser Klassen werden als `enum`-Proxys und `enum`-Pools bezeichnet. Die Instanzen mehrerer Pool-Klassen werden mit dem Begriff **Pools** zusammengefasst. Codeabschnitte und Spezifikationen werden in Listings, wie zum Beispiel Listing 1.1, dargestellt. Im Laufe der Arbeit wird mehrfach auf die Spezifikation aus Listing 1.1 zurückgegriffen.

Am Ende dieser Arbeit befindet sich ein Glossar. Einige zentrale Begriffe werden im Folgenden bereits erläutert.

Listing 1.1 Eine einfache Spezifikation

```
1 enum Farbe {
2     rot,
3     blau;
4 }
5 T {
6     i32 i;
7     Farbe f;
8 }
```

enum-Typ Ein Aufzählungstyp. Wird verwendet, wann immer eine Enumeration als Typ gemeint ist. In Listing 1.1 ist `Farbe` ein `enum`-Typ.

Nutzertyp Ein Typ, der keiner der von SKilL vordefinierten Typen und auch kein `enum`-Typ ist. Also alle nicht-`enum`-Typen, die der Werkzeugbauer spezifiziert hat. In Listing 1.1 ist `T` ein Nutzertyp.

enum-Konstante Eine in einem `enum`-Typ deklarierte Konstante. Der Begriff wird sowohl für Konstanten in Spezifikationen, als auch für Konstanten in generiertem Code und Konstanten, für die es keinen Code gibt, verwendet. In Listing 1.1 sind `rot` und `blau` `enum`-Konstanten.

enum-Feld Ein Feld in einem Nutzertyp, dessen Feldtyp ein `enum`-Typ ist. Dieser Begriff existiert, um Felder mit `enum`-Typ eindeutig von Feldern mit anderen Feldtypen zu unterscheiden. Letztere werden als **normale Felder** bezeichnet. In Listing 1.1 ist `f` ein `enum`-Feld und `i` ein normales Feld.

bekannte Typen und Felder Typen und Felder, sowohl `enum` als auch nicht `enum`, die in der Werkzeugspezifikation enthalten sind.

unbekannte Typen und Felder Typen und Felder, sowohl `enum` als auch nicht `enum`, die nicht in der Werkzeugspezifikation enthalten sind. Unbekannte Felder werden auch **verteilte Felder** genannt.

Serialisierung Das Serialisieren wird als **schreiben** bezeichnet und das Deserialisieren als **lesen**. Der Begriff Serialisierung beinhaltet sowohl das Lesen, als auch das Schreiben einer Datei.

1.3 Änderungstoleranz

Eines der Ziele dieser Arbeit ist die Implementierung von änderungstoleranten `enum`-Typen. Dieser Abschnitt beleuchtet daher, was Änderungstoleranz für eine Implementierung von `enum`-Typen nach den gegebenen Vorgaben bedeutet. Da auf das Binärformat erst in Abschnitt 2.1 eingegangen wird, sei an dieser Stelle schon erwähnt, dass `enum`-Typen durch Typdeklarationen und `enum`-Konstanten durch Felddeklarationen mit einem fest vorgegebenen Feldtyp repräsentiert werden.

Die von Felden [Fel17a] beschriebenen Änderungen sind Änderungen am Typ eines Feldes, das Entfernen eines Feldes, das Hinzufügen eines Feldes, das Entfernen eines Typs, das Hinzufügen eines Typs und das Verändern einer Supertyp-Beziehung (siehe [Fel17a] § 2.2).

Änderungen am Typ eines Feldes Da `enum`-Typen keine Felder haben, scheint diese Art der Änderung auf den ersten Blick nicht relevant zu sein. Jedoch steht der Typ eines Feldes nicht nur in der Spezifikation, sondern auch in der Austauschdatei. Das heißt, dass auch `enum`-Konstanten einen Feldtyp haben. Da dieser Feldtyp in der Spezifikation nicht angegeben wird, kann er vom Werkzeugbauer nicht verändert werden. Es kann aber vorkommen, dass eine fehlerhafte Austauschdatei eingelesen wird, in der der Feldtyp einer `enum`-Konstanten verändert wurde. Wenn sich der Feldtyps einer `enum`-Konstante ändert, muss ein Fehler geworfen werden.

Der Feldtyp eines Feldes in einem Nutzertyp hat sich geändert, wenn das Feld in der Spezifikation den einen Feldtyp hat und in der Austauschdatei einen anderen. Die für die Implementierung von `enum`-Typen zu beachtenden Fälle sind, wenn sich der Feldtyp von oder zu einem `enum`-Typ ändert. Das heißt also, wenn der Feldtyp `f` aus Listing 1.1 in einer Austauschdatei nicht `Farbe` ist oder wenn der Feldtyp von `i` aus Listing 1.1 in einer Austauschdatei ein `enum`-Typ ist. Wenn sich der Feldtyp von `i` zu einem Typ ändert, der kein `enum`-Typ ist, ist das für diese Arbeit irrelevant.

Entfernen oder Hinzufügen eines Feldes Bei der Implementierung von `enum`-Typen ist diese Änderung an zwei Stellen zu beachten. Zum Einen bei Nutzertypen, zu denen ein `enum`-Feld hinzugefügt oder entfernt wird und zum Anderen bei `enum`-Typen selbst. `enum`-Typen haben zwar keine Felder, aber `enum`-Konstanten, die hinzugefügt oder entfernt werden können. Da die `enum`-Konstanten im Binärformat auch durch Felddeklarationen repräsentiert werden, hat das Entfernen oder Hinzufügen einer `enum`-Konstanten die gleichen Auswirkungen für einen `enum`-Typ, wie das Entfernen oder Hinzufügen eines Feldes für einen Nutzertyp. Das Hinzufügen einer `enum`-Konstanten hat zudem Auswirkungen auf alle `enum`-Felder dieses `enum`-Typs. Sie müssen in der Lage sein, Werte zu speichern, die keiner bekannten `enum`-Konstanten entsprechen. Wenn zum Beispiel der `enum`-Typ Farbe aus Listing 1.1 in einer Austauschdatei noch eine weitere Konstante `gelb` hat, dann muss das Feld `f` der zu der Spezifikation aus Listing 1.1 generierten Anbindung in der Lage sein, den Wert `gelb` zu speichern.

Entfernen oder Hinzufügen eines Typs Ob ein Nutzertyp hinzugefügt oder entfernt wird, ist für die Implementierung von `enum`-Typen irrelevant. Der Typ könnte zwar `enum`-Felder haben, aber das wurde bereits im vorhergehenden Paragraph bedacht. Das Hinzufügen oder Entfernen eines `enum`-Typs ist relevant. Die Implementierung muss so umgesetzt werden, dass sie dies toleriert. Das heißt wenn eine Austauschdatei einen `enum`-Typ kennt, den die einlesende Anbindung nicht kennt, muss die Anbindung trotzdem mit diesem `enum`-Typ umgehen können. Sie muss ihn zur Laufzeit repräsentieren können und sie muss ihn ohne Informationsverlust in eine Austauschdatei schreiben können.

Veränderung der Supertyp-Beziehung `enum`-Typen haben keine Vererbungsbeziehungen. Daher ist diese Art der Änderung für sie nicht zulässig. Sobald eine Austauschdatei eingelesen wird, in der ein `enum`-Typ Teil einer Vererbungsbeziehung ist, muss ein Fehler geworfen werden.

2 Design

In diesem Kapitel wird beschrieben, wie enum-Typen umgesetzt werden sollen. Die Aufgabenstellung fordert, dass enum-Typen einer Spezifikation auf Enumerationen der Zielsprache abgebildet werden. Um diese Abbildung zu realisieren, wird für jeden enum-Typ einer Spezifikation eine Enumeration der Zielsprache generiert. Die Namen der Enumerationen und die Namen ihrer Konstanten entsprechen den Namen der enum-Typen und -Konstanten aus der Spezifikation.

Des Weiteren fordert die Aufgabenstellung, dass enum-Typen keine Felder haben, dass keine anderen Typen von enum-Typen erben und dass es möglich ist, mit der generierten Anbindung auf die IDs der enum-Werte zuzugreifen.

Der erste Abschnitt befasst sich mit enum-Typen und -Konstanten. Die meisten Entscheidungen folgen direkt aus dem von der Aufgabenstellung vorgegebenen Binärformat.

Der zweite Abschnitt befasst sich mit den enum-Feldern in Nutzertypen. Die dort getroffenen Entscheidungen ergeben sich nicht direkt aus der Aufgabenstellung. Stattdessen wurden die drei beschriebenen Möglichkeiten ausprobiert und schließlich eine davon ausgewählt.

2.1 Implikationen aus dem Binärformat

Im Binärformat werden enum-Typen durch Typdeskriptoren und enum-Konstanten durch Felddeskriptoren dargestellt. Der Typdeskriptor eines enum-Typs besteht aus dem Namen des Typs, der Anzahl der Instanzen, einer enum-Restriktion, der ID des Supertyps und der Anzahl der enum-Konstanten des Typs. Die Anzahl der Instanzen und die ID des Supertyps sind für alle enum-Typen immer 0. Die enum-Restriktion ist eine Typ-Restriktion. Sie hat die ID 7 und keine Parameter und kennzeichnet einen Typ als enum-Typ. Die Felddeskriptoren von enum-Konstanten setzen sich zusammen aus einer ID und dem Namen der Konstanten, einer leeren Menge an Feldrestriktionen und dem Feldtyp `i8`.

Wenn eine binäre Austauschdatei eingelesen wird, wird für jeden Typdeskriptor ein Pool erzeugt und für jeden Felddeskriptor ein Feldrepräsentant. Wenn ein Zustand in eine Austauschdatei geschrieben wird, wird für jeden Pool ein Typdeskriptor geschrieben und für jeden Feldrepräsentanten ein Felddeskriptor. Um die bereits vorhandenen Serialisierungsfunktionen auch auf enum-Typen und enum-Konstanten anwenden zu können, werden diese zur Laufzeit ebenfalls durch Pools und Feldrepräsentanten repräsentiert.

Pools sind jedoch nicht nur die Laufzeitrepräsentanten ihres Typs, sondern verwalten auch deren Instanzen. Dem Binärformat zufolge kann es von enum-Typen keine Instanzen geben, sodass keine der vorhandenen Pool-Klassen `StoragePool` oder `BasePool` für enum-Typen verwendet werden kann. Es muss eine neue Pool-Klasse geschaffen werden, die von den vorhandenen Pool-Klassen erbt. Die neue Klasse ist `EnumPool`. Die Vererbungsbeziehung ist notwendig, da Instanzen von `EnumPool` sonst nicht mehr von den schon vorhandenen Funktionen serialisiert werden können. Da

die Supertyp-ID immer 0 ist, sind enum-Typen Basistypen, also ist EnumPool eine Subklasse von BasePool. In EnumPool müssen die geerbten Methoden zur Manipulation der Typinstanzen dem Anwender unzugänglich gemacht werden. Außerdem müssen die Felddatenserialisierungsmethoden überschrieben werden. Die Funktion zum Serialisieren der Felddaten von enum-Feldern ist von der Aufgabenstellung vorgegeben. Das Felddatum v eines enum-Feldes vom enum-Typ t wird mit $[\![t.fieldByName(v.name).fieldID]\!]_{v64}$ serialisiert. Für enum-Felder wird also die Feld-ID des Feldrepräsentanten einer enum-Konstanten in Austauschdateien geschrieben.

Um enum-Restriktionen des Binärformats darstellen zu können, müssen die SKilL-Bibliotheken von Java und C++ um eine Klasse EnumRestriction erweitert werden. Instanzen dieser Klasse markieren einen Typ als enum-Typ. Bei Lesen einer Austauschdatei muss für jeden Typ mit enum-Restriktion eine Instanz von EnumPool anstelle eines normalen Pools erstellt werden. Eine enum-Restriktion ist im Binärformat zwar eine Typ-Restriktion, unterscheidet sich von den anderen Typ-Restriktionen jedoch in so fern, als dass sie kein Bestandteil der SKilL-Spezifikationssprache ist. Die Klasse EnumRestriction kann nach dem Singleton-Entwurfsmuster (siehe [GHJV15] §3.5) implementiert werden, da eine Instanz ausreicht, um alle enum-Pools zu markieren.

Ein Feldrepräsentant, der eine enum-Konstante repräsentiert, muss, wie es im Binärformat geschrieben steht, den Feldtyp i8 haben. Der Feldname entspricht dem Namen der repräsentierten Konstanten. Die Feld-ID entspricht, falls eine Austauschdatei eingelesen wird, der ID des Felddeskriptors und stimmt damit im Allgemeinen nicht mit dem Wert der repräsentierten Konstanten überein. Feldrepräsentanten haben (De-)Serialisierungs- und Zugriffsmethoden für die Felddaten der von ihnen repräsentierten Felder. enum-Konstanten sind keine Felder und haben auch keine Felddaten. Feldrepräsentanten von enum-Konstanten sollten daher weder Serialisierungs- noch Zugriffsmethoden haben.

2.2 enum-Felder

Der Feldtyp von enum-Feldern im generierten Code darf kein enum-Typ sein, da diese nicht änderungstolerant sind. Für die Spezifikation aus Listing 1.1 kann f im generierten Code also nicht vom Typ Farbe sein, denn dann kann f nur die Werte rot und blau annehmen. Wenn eine Austauschdatei den enum-Typ Farbe mit einer zusätzlichen enum-Konstante gelb und Instanzen des Nutzertyp T, bei denen f den Wert gelb hat, enthält, dann kann die zu der Spezifikation aus Listing 1.1 generierte Anbindung mit dieser Austauschdatei nicht umgehen. Das Feld f muss also von einem Typ $T_?$ sein, der kein enum-Typ ist.

Um dem Werkzeugbauer einen intuitiven Umgang mit enum-Typen zu ermöglichen, müssen die Zugriffsmethoden im generierten Code trotzdem mit enum-Konstanten umgehen können. Das heißt, der Setter von f muss eine enum-Konstante vom Typ Farbe entgegennehmen und diese in $T_?$ konvertieren, sodass sie in f gespeichert werden kann. Analog dazu muss der Getter von f die Felddaten von $T_?$ in eine enum-Konstanten vom Typ Farbe konvertieren und diese zurückgeben. Wenn f einen unbekannten Wert enthält, kann dieser nicht konvertiert werden. Es muss also noch eine weitere get-Methode geben, die f ohne Konvertierung als $T_?$ zurückgibt.

Da bei jedem Zugriff über die ersten beiden Zugriffsmethoden eine Konvertierung zwischen $T_?$ und dem eigentlichen enum-Typ erfolgt, muss $T_?$ ein Typ sein, mit dem diese Konvertierung möglichst einfach ist. Für Java und C++ bietet sich int als Feldtyp an. Für Java bietet es sich außerdem

an, String als Feldtyp zu verwenden. Nach einigen Versuchen mit String und int wurden diese beiden Möglichkeiten verworfen. In den folgenden beiden Abschnitte wird auf die Gründe dafür eingegangen. Es wurde eine dritte Möglichkeit gewählt. Diese verwendet einen eigenen Typ EnumProxy als Feldtyp von enum-Feldern. Der Dritte der folgenden Abschnitte geht darauf ein, welche Anforderungen die Klasse EnumProxy erfüllen muss und welche Auswirkung ihre Existenz auf die restliche Implementierung hat. Der Versuch, für enum-Felder int oder String als Typ zu verwenden, wurde nur für Java unternommen, daher beziehen sich die beiden folgenden Abschnitte, solange nicht anders angegeben, ausschließlich auf Java.

2.2.1 Verwendung von int als Feldtyp

Als Erstes werden die enum-Pools betrachtet. Die für enum-Typen generierten Enumerationen der Zielsprache sind keine Subklasse von SkillObject und können daher nicht als Typ für ihre Pools verwendet werden. Aus Ermangelung eines anderen verwendbaren Typs wird SkillObject als Typ für enum-Pools verwendet. Das heißt, enum-Pools verwalten SKilL-Objekte und haben Felddatenserialisierungsmethoden für SKilL-Objekte. Ersteres ist kein Problem, da es von enum-Typen keine Instanzen gibt, letzteres stellt für bekannte enum-Felder noch kein Problem dar, sondern nur für unbekannte, daher wird darauf erst später eingegangen.

Es gibt also Felder, die im generierten Code vom Typ int sind, deren Feldtyp im skill-Typsystem aber ein enum-Pool ist, der SKilL-Objekte verwaltet. Diese beiden Feldtypen passen nicht zusammen. Die Komponenten, die diese Fehler bemerken, sind die Feldrepräsentanten, da sie sowohl den Feldtyp im SKilL-Typsystem als auch den generierten Feldtyp kennen. Um die Feldrepräsentanten zu täuschen, werden alle enum-Pools in Integer-Feldtypen konvertiert. Eigentlich ist eine solche Konvertierung in Java nicht erlaubt. In der Java-Implementierung gibt es jedoch eine Methode cast, die das Java-Typsystem umgeht und die Konvertierung ermöglicht.

Nun stellt sich die Frage, welche Werte als Felddaten gespeichert werden sollen. Die Werte müssen sowohl bekannte als auch unbekannte enum-Konstanten darstellen können und es muss eindeutige Abbildungen zwischen ihnen, den enum-Konstanten und den Feldrepräsentanten der enum-Konstanten geben. Die Ordinalzahlen der enum-Konstanten bieten eine eindeutige Abbildung zwischen Feldrepräsentanten und enum-Konstanten, können aber nicht verwendet werden, da sie nur die bekannten enum-Konstanten abdecken. Stattdessen werden die Feld-IDs der Feldrepräsentanten in den enum-Feldern gespeichert. Das funktioniert, da es für jede enum-Konstante, sowohl für die bekannten als auch für die unbekannten, einen Feldrepräsentanten mit einer innerhalb eines enum-Typs eindeutigen Feld-ID gibt.

Die Felddaten so zu wählen, ermöglicht es außerdem, sie sehr einfach zu serialisieren. Wie von der Aufgabenstellung vorgegeben, sollen für die Felddaten von enum-Feldern die Feld-IDs der Feldrepräsentanten in die Austauschdatei geschrieben werden. Wenn die Felddaten von enum-Feldern nun die Feld-IDs sind, können sie direkt gelesen und geschrieben werden.

Um die im vorhergehenden Abschnitt erwähnten Zugriffsmethoden zu realisieren, muss es eine Abbildung zwischen den Felddaten des enum-Feldes, also den Feld-IDs, und den enum-Konstanten geben. Da alleine der verwaltende Pool des enum-Typs alle Feldrepräsentanten kennt, wird die Abbildung in der Klasse EnumPool implementiert. Die Abbildung wird mit Hilfe der Arrays intToEnum und enumToInt implementiert. intToEnum enthält die enum-Konstanten des enum-Typs und enumToInt enthält die Feld-IDs aller Feldrepräsentanten eines enum-Typs. An der Stelle i in intToEnum steht

die enum-Konstante, zu der der Feldrepräsentant mit der Feld-ID $i + 1$ gehört, und an der Stelle i in enumToInt steht die Feld-ID des Feldrepräsentanten, der zu der enum-Konstanten mit der Ordinalzahl i gehört. Die Verschiebung der Feld-ID um Eins bei intToEnum ist notwendig, da die Feld-IDs in SKilL nicht bei 0, sondern erst bei 1 beginnen.

Damit der Nutzertyp, zu dem ein enum-Feld gehört, die eben beschriebene Abbildung verwenden kann, braucht er eine Referenz auf den enum-Pool. In der bisherigen Implementierung von SKilL kennen die Pools zwar die Instanzen ihrer Typen, aber die Instanzen haben weder Kenntnis darüber, zu welchem Pool sie gehören, noch kennen sie das Zustandsobjekt, über das man Zugriff auf alle Pools hätte.

In der Java-Implementierung gibt es drei Möglichkeiten um neue Nutzertypinstanzen zu einem Pool hinzuzufügen. Die ersten beiden Möglichkeiten sind die Fabrikmethoden (siehe [GHJV15] §3.3) und die Erbauerklasse (siehe [GHJV15] §3.2) der Pools. Die dritte Möglichkeit ist selbst eine neue Instanz des Nutzertyps zu erstellen und mit add zum Pool des Nutzertyps hinzuzufügen. Die Konstruktoren der Nutzertypen werden so erweitert, dass ihnen für jedes enum-Feld des Nutzertyps eine Referenz auf den entsprechenden enum-Pool übergeben werden muss. Da jeder Pool das Zustandsobjekt kennt und darüber Zugriff zu allen anderen Pools hat, sind die Fabrikmethoden und die Erbauerklasse dazu fähig, die Konstruktoren mit den erwarteten enum-Pool-Referenzen als Parameterwerte aufzurufen. Da der Werkzeugbauer selbst das Zustandsobjekt auch kennt, kann auch die dritte Möglichkeit weiterhin verwendet werden.

Die bis hierhin beschriebene Umsetzung funktioniert ausschließlich für bekannte enum-Felder, für unbekannte enum-Felder muss noch mehr getan werden. Wie am Anfang dieses Abschnitts erwähnt, werden bei diesem Ansatz enum-Pools in Integer-Feldtypen konvertiert und enum-Pools verwendet, die Felddatenserialisierungsmethoden für SKilL-Objekte zur Verfügung stellen. Aus dieser Kombination ergibt sich ein Problem für unbekannte enum-Felder.

Die zur Repräsentation der unbekannten enum-Felder verwendeten verteilten Felder greifen bei der Serialisierung der Felddaten auf die Serialisierungsmethoden des Feldtyps zu. Die Felddatenserialisierungsmethoden der enum-Pools erwarten Felddaten vom Typ SkillObject, die Felddaten des verteilten Feldes sind aber vom Typ int. Die Methoden und die Felddaten sind nicht kompatibel. Bevor die Idee, int als Feldtyp zu verwenden, verworfen wurde, gab es zwei Lösungsansätze.

Der erste Ansatz war, jeden Aufruf von Felddatenserialisierungsmethoden abzufangen, zu überprüfen, ob der Feldtyp ein enum-Pool ist und, falls dem so ist, alternative Felddatenserialisierungsmethoden aufzurufen. Der Implementierungsaufwand dieser Lösung ist gering, die Ausführungszeit erhöht sich jedoch und zwar unabhängig davon, ob eine Spezifikation enum-Typen enthält oder nicht. Darum wurde der Ansatz wieder verworfen.

Der zweite Ansatz war, die Konvertierung des Feldtyps zu einem Integer-Feldtyp zu unterlassen, sodass die Felddaten der verteilten Felder SKilL-Objekte sind, und eine zusätzliche Klasse zu implementieren, die ein Proxy (siehe [GHJV15] §4.7) für die Felddaten ist. Die Proxy-Klasse ist eine Subklasse von SkillObject und für den Werkzeugbauer unzugänglich. Beim Einlesen einer Austauschdatei geben die enum-Pools den verteilten Feldern Proxy-Objekte zurück. Die Proxy-Objekte werden als Felddaten gespeichert und beim Schreiben einer Austauschdatei wieder zurück an die enum-Pools gegeben. Der Ansatz wurde in dieser Form verworfen, gab jedoch die Idee für den letztlich gewählten Ansatz.

2.2.2 Verwendung von String als Feldtyp

Wie auch im vorhergehenden Abschnitt müssen die enum-Pools konvertiert werden, hier jedoch nicht in Integer-, sondern in String-Feldtypen. Es stellt sich erneut die Frage, welche Felddaten in den enum-Feldern stehen sollen. Sie müssen wieder so gewählt sein, dass eine eindeutige Zuordnung der Felddaten zu den enum-Konstanten und den Feldrepräsentanten möglich ist. Als Felddaten wurden die Namen der enum-Konstanten verwendet.

In Java gibt es für Enumerationen die Methoden valueOf und toString (siehe [Ora]). Die Methode valueOf ist case-sensitive und wirft einen Fehler, wenn der übergebene String mit keinem der Namen der enum-Konstanten übereinstimmt. Mit toString erhält man den Namen einer enum-Konstanten als String. Mit valueOf erhält man die enum-Konstante, deren Namen mit dem übergebenen String identisch ist. Damit gibt es für bekannte enum-Konstanten eine Abbildung zwischen diesen und den Felddaten.

Jeder Feldrepräsentant hat einen Namen. Dieser entspricht dem Namen des repräsentierten Feldes. Falls der Feldrepräsentant eine enum-Konstante repräsentiert, entspricht sein Name dem Namen der repräsentierten enum-Konstanten. Da alle enum-Konstanten desselben enum-Typs unterschiedliche Namen haben, gibt es damit eine eindeutige Abbildung zwischen den Felddaten und den Feldrepräsentanten.

Die Zugriffsmethoden für die enum-Felder können mit Hilfe von valueOf und toString realisiert werden. Die Verwendung von String als Feldtyp hat damit gegenüber der Verwendung von int den Vorteil, dass die Nutzertypen den Pool der enum-Konstanten nicht kennen müssen.

Die Serialisierung hingegen gestaltet sich unter Verwendung von String aufwändiger. Da in der Austauschdatei die Feld-IDs der Feldrepräsentanten der enum-Konstanten stehen sollen, die Felder aber die Namen von enum-Konstanten enthalten, muss beim Schreiben für jedes enum-Feld die Feld-ID zum Namen und beim Lesen der Name zur Feld-ID ermittelt werden. Da jeder enum-Pool alle seine Feldrepräsentanten kennt und die Namen der Feldrepräsentanten den Namen der enum-Konstanten entsprechen, gibt es dafür zwei neue Methoden in EnumPool. Eine, die einen Namen als String entgegen nimmt, den Feldrepräsentanten mit diesem Namen ermittelt und dessen Feld-ID zurückgibt, und eine, die eine Feld-ID entgegen nimmt, den Feldrepräsentanten mit dieser Feld-ID nimmt und dessen Namen als String zurückgibt. Diese Methoden sind den Feldrepräsentanten von enum-Feldern zugänglich, da die Feldrepräsentanten die Pools der Nutzertypen, zu denen sie gehören, kennen und sie über die Pools Zugriff auf das Zustandsobjekt, und damit auf alle anderen Pools, haben.

Außerdem muss bei der Verwendung von String als Feldtyp die Groß- und Kleinschreibung beachtet werden. SKilL-intern werden alle Namen klein geschrieben, die enum-Konstanten könnten aber auch anders geschrieben sein. Der Verbindungspunkt zwischen den internen Namen und den Namen der enum-Konstanten sind die enum-Felder. Da die enum-Felder beim Serialisieren benötigt werden und dort aus Laufzeitgründen keine Konvertierung zwischen Groß- und Kleinschrift anfallen soll, sind die Werte von enum-Feldern in Kleinschrift. Folglich finden alle Konvertierungen zwischen Groß- und Kleinschrift in den Zugriffsmethoden statt. Dies betrifft sowohl die Zugriffsmethoden der enum-Felder in den generierten Nutzertypklassen als auch die Zugriffsmethoden der Feldrepräsentanten.

Auch für diesen Ansatz ergeben sich Probleme aufgrund der Konvertierung des Feldtyps. Die Probleme sind die gleichen, die im vorhergehenden Abschnitt beschriebenen sind. Ein weiterer Nachteil von `String` als Feldtyp ist, dass sich dieser Ansatz nur in Java umsetzen lässt. Es wäre zwar möglich, ihn auch auf C++ zu übertragen, allerdings müsste man dazu Funktionen implementieren, die analog zu den Java-Methoden `valueOf` und `toString` funktionieren.

2.2.3 Verwendung eines eigenen Feldtyps

Der neu geschaffene Feldtyp ist die Klasse `EnumProxy`, ein Proxy (siehe [GHJV15] §4.7) für die `enum`-Typen. `EnumProxy` wird dort eingesetzt, wo eigentlich `enum`-Typen stehen sollten, aber nicht stehen dürfen, weil sie keine Subklassen von `SkillObject` sind oder weil sie nicht änderungstolerant sind. Die Klasse wird also als Typ von `enum`-Pools verwendet und als Feldtyp von `enum`-Feldern. Um ihre Aufgabe erfüllen zu können, muss `EnumProxy` eine Subklasse von `SkillObject` sein. Ihre Instanzen repräsentieren die `enum`-Konstanten als Felddatum. Da es zu jeder `enum`-Konstanten auch einen Feldrepräsentanten gibt, werden ein `enum`-Proxy und ein Feldrepräsentant in dieser Arbeit als zusammengehörig bezeichnet, wenn sie Repräsentanten derselben `enum`-Konstanten sind.

Die Instanzen von `EnumProxy` werden von `enum`-Pools verwaltet. Jeder `enum`-Pool ist für die Instanzen verantwortlich, die zu `enum`-Konstanten seines `enum`-Typs gehören. Die Klasse `EnumProxy` ist dem Werkzeugbauer als Stellvertreter von `enum`-Werten bekannt. Falls ein `enum`-Feld eine unbekannte `enum`-Konstante enthält, muss der Wergzeugbauer mit `enum`-Proxys arbeiten. Darum ist es angebracht, dem Werkzeugbauer Zugriff auf alle `enum`-Proxys zu gewähren. Dazu gibt es eine neue Klasse `EnumProxyIterator`, einen Iterator (siehe [GHJV15] §5.4) über alle `enum`-Proxys eines Pools. Die `enum`-Proxys sind außerdem zustandsgebunden.

`enum`-Felder haben nun den Feldtyp `EnumProxy`. `EnumProxy` ist jedoch nur ein Typ, sodass alle `enum`-Felder denselben Feldtyp haben, auch wenn sie laut Spezifikation unterschiedliche `enum`-Typen haben. Wenn jeder `enum`-Proxy eine Referenz auf den `enum`-Pool, zu dem er gehört, hat, dann ist das Wissen über den eigentlichen Typ eines `enum`-Feldes im Felddatum selbst gespeichert. Das hat zur Folge, dass ein `enum`-Feld niemals einen illegalen Wert annehmen darf, da dann das Wissen darüber, welchen Typ das `enum`-Feld laut Spezifikation hat, verloren geht.

Wenn man zum Beispiel die Spezifikation aus Listing 1.1 um einem `enum`-Typ `Brot` mit der Konstanten `roggen` erweitert, gibt es zur Laufzeit drei `enum`-Proxys. Diese werden im Folgenden als $proxy_{rot}$, $proxy_{blau}$ und $proxy_{roggen}$ bezeichnet. Außerdem gibt es in der für `T` generierten Klasse ein Feld `f`, das den Feldtyp `EnumProxy` hat. Theoretisch könnte `f` jeden der drei `enum`-Proxys als Wert haben. Da `f` in der Spezifikation als `Farbe` deklariert wurde, sind aber nur $proxy_{rot}$ und $proxy_{blau}$ legale Werte.

Der Übergabeparameter eines `enum`-Feld-Setters in einer generierten Nutzertypklasse ist eine `enum`-Konstante. `f` ist vom Typ `EnumProxy` und nur der `enum`-Pool zu `Farbe` kennt die `enum`-Proxys, die für `f` als Werte in Frage kommen. Die Referenz zum `enum`-Pool wird also benötigt, um beim `enum`-Pool den zur einer übergebenen `enum`-Konstanten gehörenden `enum`-Proxy zu erfragen. Wenn `f` den Wert `null` oder einen anderen Wert annimmt, in dem es keine Referenz zum `enum`-Pool von `Farbe` gibt, dann kann `f` nicht wieder auf einen legalen Wert gesetzt werden.

Um dies zu verhindern, muss garantiert werden, dass f nur Werte vom richtigen enum-Typ annimmt. Im Setter der Nutzertypklasse und im Setter des Feldrepräsentanten von f muss überprüft werden, dass f nur auf legale Werte gesetzt wird. Die Konstruktoren der Nutzertypklasse müssen so angepasst werden, dass f für jede neue Instanz des Nutzertyps auf einen Defaultwert gesetzt wird. Da die Nutzertypklasse keinen Zugriff auf die Pools hat, muss der Defaultwert beim Aufruf des Konstruktors als Parameter übergeben werden. Bei einem bekannten enum-Typ ist der Defaultwert die erste Konstante der enum-Typ-Deklaration (siehe [Fel17b] §4.5). Bei einem unbekannten enum-Typ gibt es in der Anbindung keine erste Konstante, daher wird stattdessen die enum-Konstante, die zum Feldrepräsentanten mit der Feld-ID 1 gehört, als Defaultwert verwendet. Diese Konstante war in der ersten Anbindung, mit der die Austauschdatei geschrieben wurde die als erstes deklarierte enum-Konstante.

Der Defaultwert wird als enum-Proxy benötigt. Auf enum-Proxys kann nur über ihren enum-Pool zugegriffen werden. Beim Lesen einer Austauschdatei kann es passieren, dass ein Nutzertyp instanziiert wird, bevor den enum-Pools alle Feldrepräsentanten und enum-Proxys bekannt sind. Das wird zu einem Problem, wenn der Nutzertyp den Defaultwert braucht, bevor der enum-Pool ihn kennt. Dieser Fall tritt zum Beispiel ein, wenn mit der Java-Anbindung der Spezifikation aus Listing 1.1 eine Austauschdatei gelesen wird, die nur einen feldlosen Nutzertyp namens *T* und Instanzen von diesem enthält. Beim Einlesen der Datei werden Instanzen des der Anbindung bekannten Nutzertyps T erstellt. Dieser hat zwei Felder, eines davon ein enum-Feld, sodass dem Konstruktor von T der Defaultwert für das enum-Feld übergeben werden muss. Da der enum-Typ Farbe nicht in der Austauschdatei enthalten ist, werden seine Feldrepräsentanten erst nach der Allokation der Instanzen hinzugefügt. Folglich ist der Defaultwert von Farbe in dem Moment, in dem er bei der Allokation der Instanzen von T benötigt wird, noch nicht bekannt. Die Lösung des Problems ist, dass EnumPool ein Feld hat, das für jeden bekannten enum-Typ ab dem Moment der Instanziierung des Pools einen Defaultwert enthält. Dass dieses Feld den tatsächlichen Defaultwert enthält, ist zu diesem Zeitpunkt natürlich noch nicht möglich, stattdessen enthält es erst mal irgendeinen Dummy-enum-Proxy. Die Idee lässt sich nur umsetzen, wenn enum-Felder nur Referenzen auf enum-Proxys enthalten und der Defaultwert ebenfalls eine Referenz ist. Sobald der enum-Pool den echten Defaultwert kennt, wird der Dummy-Defaultwert durch den echten ersetzt. Das soll heißen, der Defaultwert enum-Proxy selbst wird verändert, die Referenz auf diesen bleibt unverändert.

Auch bei diesem Ansatz muss es eine Abbildung zwischen den enum-Proxys und ihren zugehörigen Feldrepräsentanten und enum-Konstanten geben. EnumProxy ist eine Subklasse von SkillObject. Das heißt hat EnumProxy eine SKilL-ID und einen SKilL-Namen. Indem man SKilL-ID und SKilL-Namen einer Instanz von EnumProxy genauso wählt wie die Feld-ID und den Feldnamen des zugehörigen Feldrepräsentanten, kann man die beiden einander eindeutig zuordnen. Da sowohl der enum-Proxy als auch der Feldrepräsentant zudem noch wissen, zu welchem enum-Pool sie gehören, lassen sie sich einander enum-Typ übergreifend zuordnen.

Die Zuordnung zu den Konstanten kann in der Java-Implementierung, analog zu dem Ansatz mit String als Feldtyp, mit toString und valueOf erfolgen. In der C++-Implementierung muss die Zuordnung, analog zu dem Ansatz mit int als Feldtyp, über die Feld-ID und die Ordinalzahlen der enum-Konstanten erfolgen.

Wenn EnumProxy als Feldtyp für enum-Felder verwendet wird, müssen unbekannte enum-Felder nicht gesondert betrachtet werden. Mit EnumProxy passt der nach außen sichtbare Feldtyp von enum-Feldern zum SKilL-internen Feldtyp, sodass, anders als bei int und String, der Feldtyp nicht

konvertiert werden muss. Ohne Konvertierung treten alle daraus folgenden Probleme nicht mehr auf. Die Typen der Übergabeparameter und Rückgabewerte der Felddatenserialisierungsmethoden des SKilL-internen Feldtyps stimmen nun mit dem Typ der Felddaten überein.

Es gibt trotzdem noch zwei Punkte, die bezüglich der Felddatenserialisierung von `enum`-Feldern beachtet werden müssen. Der erste ist, dass ein Fehler geworfen werden muss, falls eine korrumpierte Austauschdatei eingelesen wird, die für ein `enum`-Feld ein illegales Felddatum enthält. Der zweite ist, dass in den Serialisierungsmethoden von `EnumPool` überprüft werden muss, ob ein Felddatum uninitialisiert ist. Dies kann passieren, wenn in einer Anbindung ein Typ T mit einem unbekannten Feld g existiert. Da es für g keinen generierten Code gibt, kann nicht sichergestellt werden, dass g keine illegalen Werte annimmt. Das Feld g bleibt zudem in allen mit dieser Anbindung neu erstellten Instanzen von T uninitialisiert. Falls das Felddatum eines uninitialisierten `enum`-Feldes serialisiert werden soll, wird der Defaultwert des `enum`-Typs als Felddatum serialisiert.

3 Implementierung in Java

Dieses Kapitel befasst sich mit der Implementierung von enum-Typen in Java. Zuerst werden die neu hinzugefügten Bibliotheksklassen EnumProxy und EnumPool beschrieben. Dann wird beschrieben, welche Änderungen an welchen der bereits vorhandenen Bibliotheksklassen vorgenommen werden müssen. Zuletzt wird der generierte Code beschrieben. Die neu hinzugefügten Bibliotheksklassen EnumProxyIterator und EnumRestriction werden nicht betrachtet, da sie die Designentscheidungen aus Kapitel 2 direkt umsetzen. EnumProxyIterator implementiert java.util.Iterator (siehe [Ora]) und iteriert über die enum-Proxys eines enum-Pools und EnumRestriction implementiert TypeRestriction und entspricht dem Singleton-Entwurfsmuster (siehe [GHJV15] §3.5).

Es werden an UML-Klassendiagramme (siehe [BRJ05] §8) angelehnte Darstellungen verwendet, um eine Übersicht über die Methoden einer Klasse zu geben, und Listings, um einzelne Methoden oder Codeabschnitte genauer zu erläutern. Um Platz zu sparen, wurden in mehreren Listings die Fehlernachrichten der Ausnahmen gekürzt.

3.1 EnumProxy

Die Klasse EnumProxy ist eine Subklasse von SkillObject. Sie erbt von SkillObject das Feld skillID vom Typ int und die Methoden skillName und getSkillID. Das Feld skillID wird verwendet um die ID eines enum-Proxys zu speichern und mit getSkillID kann darauf zugegriffen werden. skillName ist in SkillObject abstrakt und muss in EnumProxy überschrieben werden. Darum hat EnumProxy das Feld name vom Typ String. Es speichert den Namen eines enum-Proxys. skillName gibt den Inhalt dieses Feldes zurück. Außerdem hat EnumProxy ein Feld und eine Methode namens owner. Das Feld owner ist vom Typ EnumPool<EnumProxy> und speichert eine Referenz auf den enum-Pool, zu dem ein enum-Proxy gehört und die Methode owner gibt besagte Referenz zurück. Alle drei Felder werden im Konstruktor von EnumProxy initialisiert. EnumProxy gehört zu common.api.

3.2 EnumPool

Abbildung 3.1 bietet einen Überblick über die Methoden und Felder von EnumPool. Die Methoden getByID, makeSubPool, make, size, add und delete sind in Abbildung 3.1 nicht enthalten, da EnumPool sie nicht implementieren darf. Falls sie trotzdem aufgerufen werden, werfen sie einen NoSuchMethodError, sonst tun sie nichts. Die Implementierungen einiger Methoden sind in Listing 3.1, 3.2 und 3.3 zu finden.

Die Liste fieldProxies enthält alle enum-Proxys eines Pools. Es ist eine ArrayList, da die enum-Proxys sukzessiv zu fieldProxies hinzugefügt werden, was eine Datenstruktur mit variabler Länge erfordert. Jedes Mal, wenn ein Feldrepräsentant zu einem enum-Pool hinzugefügt wird, wird auch

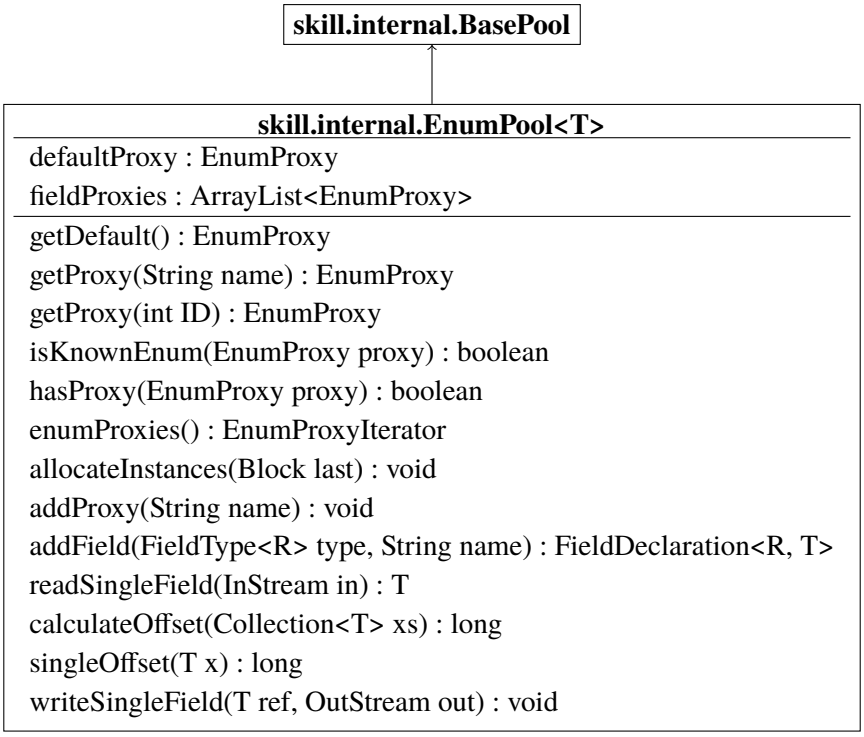

Abbildung 3.1: EnumPool.java

der zu diesem Feldrepräsentanten gehörende `enum`-Proxy hinzugefügt. Für bekannte `enum`-Typen und -Konstanten gibt es die Methode `addProxy`, die im Code der generierten Subklassen von `EnumPool` jedes Mal aufgerufen wird, wenn ein Feldrepräsentant hinzugefügt wird. Die Implementierung von `addProxy` steht in Listing 3.3. Die SKilL-ID des neuen `enum`-Proxys ist $dataFields.size() + 1$, da `addProxy` aufgerufen wird, bevor der zugehörige Feldrepräsentant hinzugefügt wird. Für unbekannte `enum`-Typen und -Konstanten muss die von `StoragePool` geerbte Methode `addField` in `EnumPool` wie in Listing 3.3 überschrieben werden, sodass auch für unbekannte `enum`-Konstanten der zugehörige `enum`-Proxy hinzugefügt wird. Auf die weiteren Eigenheiten von `addProxy` wird später in diesem Abschnitt noch eingegangen.

Die Methoden `enumProxies`, `getDefault`, `getProxy(String name)` und `getProxy(int ID)` existieren, um Zugriff auf die `enum`-Proxys eines Pools zu gewähren. `enumProxies` gibt einen Iterator über alle `enum`-Proxys eines Pool zurück. `getDefault` gibt `defaultProxy`, also den Defaultwert, zurück. `getProxy(String name)` und `getProxy(int ID)` durchsuchen `fieldProxies` nach einem `enum`-Proxy mit der SKilL-ID `ID`, beziehungsweise dem SKilL-Namen `name`. Wenn sie keinen passenden `enum`-Proxy finden, werfen sie eine `IllegalArgumentException`. Listing 3.1 zeigt die Implementierung der beiden `getProxy`-Methoden. In `getProxy(int ID)` kann, nachdem überprüft wurde, dass `ID` im Bereich von `fieldProxies` liegt, direkt auf `fieldProxies` zugegriffen werden. In `getProxy(String name)` muss über `fieldProxies` iteriert werden, bis der zu `name` passende `enum`-Proxy gefunden ist.

Die Methoden `hasProxy` und `isKnownProxy` existieren, um zu überprüfen, ob ein `enum`-Proxy zu einem Pool gehört. `isKnownProxy` überprüft außerdem, ob der `enum`-Proxy zu einer bekannten `enum`-Konstanten gehört. Listing 3.2 zeigt die Implementierung der beiden Methoden. `hasProxy` kann wie

Listing 3.1 `getProxy(String name)` und `getProxy(int ID)` aus EnumPool.java

```
1  public EnumProxy getProxy(String name){
2      for (EnumProxy proxy : fieldProxies)
3          if (name.toLowerCase().equals(proxy.skillName())) return proxy;
4      throw new IllegalArgumentException(name + " is no name of any constant of " + this.name());
5  }
6  public EnumProxy getProxy(int ID){
7      int index = ID - 1;
8      if (index < 0 || fieldProxies.size() <= index)
9          throw new IllegalArgumentException(ID + " is no ID of any constant of " + this.name());
10     return fieldProxies.get(index);
11 }
```

Listing 3.2 `hasProxy` und `isKnownEnum` aus EnumPool.java

```
1 public boolean isKnownEnum(EnumProxy proxy){
2     if (!this.hasProxy(proxy))
3         return false;
4     if (dataFields.get(proxy.getSkillID()-1) instanceof KnownDataField<?,?>)
5         return true;
6     return false;
7 }
8 public boolean hasProxy(EnumProxy proxy){ return fieldProxies.contains(proxy); }
```

dargestellt realisiert werden, da `fieldProxies` alle `enum`-Proxys eines Pools enthält. `isKnownProxy` überprüft, ob der übergebene `enum`-Proxy zum Pool gehört und ob der zugehörige Feldrepräsentant Repräsentant einer bekannten `enum`-Konstanten ist.

Die Felddatenserialisierungsmethoden `readSingleField`, `writeSingleField`, `calculateOffset` und `singleOffset` sind ähnlich implementiert wie in `StoragePool`. Sie unterscheiden sind durch ihren Umgang mit dem Felddatum `null`. In `writeSingleField`, `calculateOffset` und `singleOffset` in `EnumPool` wird, wenn ein Felddatum `null` ist, `defaultProxy` als Felddatum verwendet. Das heißt `calculateOffset` und `singleOffset` berechnen den Offset für die SKilL-ID des Defaultwerts und `writeSingleField` schreibt die SKilL-ID des Defaultwerts in Austauschdateien. In `EnumPool` entfällt die in `StoragePool` nötige Überprüfung des eingelesenen Wertes. Es wird direkt auf die Datenstruktur der Felddaten, also `fieldProxies`, zugegriffen. Wenn der eingelesene Wert nicht im Bereich von `fieldProxies` liegt, wird ein zu erwartender Fehler geworfen.

Das Feld `defaultProxy` enthält den Defaultwert des Pools. Es wird im Konstruktor von `EnumPool` wie in Listing 3.3 gezeigt initialisiert. Das Feld `owner` ist also schon richtig gesetzt. Die übrigen Felder werden in `addProxy` auf die richtigen Werte gesetzt. Die Methode `addProxy` ist wie in Listing 3.3 implementiert. Als Erstes überprüft sie anhand der ID von `defaultProxy`, ob dieser schon angepasst wurde. Wenn dem nicht so ist und es entweder keine bekannten `enum`-Konstanten gibt oder `name` mit dem Namen der ersten bekannten `enum`-Konstanten übereinstimmt, wird `defaultProxy` angepasst und zu `fieldProxies` hinzugefügt. Falls es keine bekannten Felder gibt, wird `defaultProxy` gleich beim ersten Aufruf von `addProxy` angepasst, sodass der Defaultwert, wie gefordert, die SKilL-ID 1 hat.

Listing 3.3 addProxy, addField und der Konstruktor von EnumPool.java

```
1  public EnumPool(int poolIndex, String name, String[] knownFields, AutoField<?, T>[] autoFields) {
2      super(poolIndex, name, knownFields, autoFields);
3      defaultProxy = new EnumProxy(-1, "", (EnumPool<EnumProxy>) this);
4      this.fieldProxies = new ArrayList<>();
5  }
6  protected void addProxy(String name) {
7      if (defaultProxy.skillID == -1 && (knownFields.length == 0 || name.equals(knownFields[0]))){
8          defaultProxy.skillID = dataFields.size()+1;
9          defaultProxy.name = name;
10         fieldProxies.add(firstOrdinal);
11     } else fieldProxies.add(new EnumProxy(dataFields.size()+1, name, (EnumPool<EnumProxy>) this));
12 }
13 public <R> FieldDeclaration<R, T> addField(FieldType<R> type, String name) {
14         addProxy(name);
15     return new LazyField<R, T>(type, name, this);
16 }
```

3.3 Anpassungen bereits vorhandener Bibliotheksklassen

Die meisten Bibliotheksklassen bleiben unverändert. Nur `FileParser`, `SerializationFunction`, `StoragePool`, `LazyField` und `DistributedField` müssen erweitert oder verändert werden.

`common.internal.FileParser` In `FileParser` muss die Methode `typeRestrictions` so erweitert werden, dass sie `enum`-Restriktionen aus Austauschdateien lesen kann. Dazu muss die `switch`-Anweisung in `typeRestrictions` um einen Fall für die ID 7 erweitert werden, in dem mit `rval.add(EnumRestriction.get())` eine `enum`-Restriktion zu `rval`, der Menge der Restriktionen des im Moment eingelesenen Typs, hinzugefügt wird. Da `enum`-Typen in der SKilL-Spezifikationssprache keine Typ-Restriktionen haben (siehe [Fel17a] §5.4.3), muss in diesem Fall außerdem sichergestellt werden, dass der Typ insgesamt nur eine Typ-Restriktion hat.

`common.internal.SerializationFunctions` In der Klasse `SerializationFunctions` muss die Methode `restrictions(StoragePool<?, ?> p, OutStream out)` so erweitert werden, dass sie auch `enum`-Restriktionen in Austauschdateien schreiben kann. Dazu wird mit `instanceof` überprüft, ob `p` eine Instanz von `EnumPool` ist. Wenn dem so ist, werden die Bytes `01` und `07` in die Austauschdatei geschrieben. Die Serialisierung der Restriktion muss mit `instanceof` implementiert werden, da die Pools in der Java-Implementierung ihre eigenen Restriktionen nicht kennen. Da beim Lesen sichergestellt wird, dass ein enum-Pool keine weiteren Typ-Restriktionen hat, kann im Falle einer `enum`-Restriktion immer `01` als Anzahl an Typ-Restriktionen geschrieben werden.

`common.internal.StoragePool` In `StoragePool` muss der `final`-Modifikator aus den Signaturen der Methoden `getByID`, `readSingleField`, `calculateOffset`, `singleOffset`, `writeSingleField`, `size`, `add` und `delete` entfernt werden, damit diese Methoden in `EnumPool` überschrieben werden können.

`common.internal.LazyField` und `common.internal.DistributedField` Die Zugriffsmethoden der verteilten Felder müssen so erweitert werden, dass sie überprüfen, ob ein verteiltes Feld zu einem `enum`-Pool gehört. Wenn dem so ist, werfen sie einen `NoSuchMethodError`. Wie bei der Serialisierung der Typ-Restriktionen wird `instanceof` verwendet, um heraus zu finden, ob ein Pool ein `enum`-Pool ist.

3.4 Generierter Code

Dieser Abschnitt befasst sich mit dem generierten Code. Das beinhaltet die für die `enum`-Typen generierten Enumerationen, die für die Nutzertypen generierten Subklassen von `SkillObject`, die generierten Pools der `enum`-Typen und der Nutzertypen, die generierten Feldrepräsentanten der `enum`-Felder und `enum`-Konstanten und die Klassen `Parser` und `SkillState`. Die Feldrepräsentanten von Feldern, die keine `enum`-Felder sind, und die Pools und Klassen von Nutzertypen, die keine `enum`-Felder haben, werden außen vorgelassen, da sie durch die neue Implementierung der `enum`-Typen nicht verändert werden.

Im Folgenden wird mehrfach die zur Spezifkation aus Listing 1.1 generierte Anbindung als Beispiel verwendet. Dabei wird angenommen, dass die Anbindung in das Paket `bsp` generiert wurde.

3.4.1 Enumerationen

Für jeden `enum`-Typ der Spezifikation wird eine Java Enumeration (siehe [GJS+14] §8.9) generiert. Der Name der Enumeration und die Namen ihrer Konstanten entsprechen den Namen des `enum`-Typs und seiner `enum`-Konstanten aus der Spezifiaktion. Die generierten Konstanten haben dieselbe Reihenfolge wie die Konstanten der Spezifikation. Sie sind in Großschrift, da das die Formatierung von Konstanten im Java Styleguide ist (siehe [KND+97] §9).

3.4.2 Nutzertypklassen

Für den Nutzertyp `T` aus Listing 1.1 wird die Klasse `T` in Listing 3.4 generiert. Listing 3.4 enthält nur die im Folgenden relevanten Abschnitte der Klasse `T`. Das Feld `i` und seine Zugriffsmethoden, sowie die innere Subklasse, die Methode `skillName` und das Feld `serialVersionUID` wurden weggelassen.

`T` hat zwei Konstruktoren. Der Erste wird verwendet, um Instanzen mit Defaultwerten zu erzeugen und der Zweite wird verwendet, um die Felder der Instanz mit anderen Werten zu initialisieren. Für Nutzertypen, die nur `enum`-Felder haben, haben die beiden Konstruktoren die gleiche Signatur, daher wird in diesen Fällen nur ein Konstruktor generiert.

Es gibt drei Zugriffsmethoden für das `enum`-Feld. Die ersten beiden sind nach demselben Schema benannt, wie alle Zugriffsmethoden. Für das Feld `f` also `getF` und `setF`. Die Dritte heißt `getFToEnumProxy`, da sie `f` als `enum`-Proxy zurückgibt. `getF` nutzt die Methode `isKnownEnum` von `EnumPool`, um zu überprüfen, ob der momentane Wert von `f` zu einer bekannten `enum`-Konstanten gehört und die Methode `valueOf` der Java-Enumerationen, um die entsprechende `enum`-Konstante

Listing 3.4 T.java

```
1  public class T extends SkillObject {
2      public T(int skillID, EnumProxy f ) {
3          super(skillID);
4          this.f = f;
5      }
6      public T(int skillID, int i, EnumProxy f ) {
7          super(skillID);
8          this.f = f;
9          this.i = i;
10     }
11     protected EnumProxy f;
12     final public bsp.Farbe getF() {
13         if(f.owner().isKnownEnum(f)) return bsp.Farbe.valueOf(f.skillName().toUpperCase());
14         throw new IllegalStateException("value of f unknown, use getFToEnumProxy()");
15     }
16     final public void setF(bsp.Farbe f) {
17         if (null == f) throw new IllegalArgumentException("EnumFields may not be set to null.");
18         this.f = this.f.owner().getProxy(f.toString().toLowerCase());
19     }
20     final public EnumProxy getFToEnumProxy() { return f; }
21 }
```

zu erhalten. `setF` nutzt die Methode `toString` der Java-Enumerationen, um den Namen einer `enum`-Konstanten zu erhalten und `getProxy` von `EnumPool`, um den entsprechenden `enum`-Proxy zu erhalten.

3.4.3 Pools der `enum`-Typen

Die für `enum`-Typen generierten Pools sind Subklassen von `EnumPool` und mit `EnumProxy` parametrisiert. Sie werden nach demselben Schema wie die Pools von Nutzertypen benannt, heißen also alle Pi, wobei i eine ganze Zahl ist und, bei 0 beginnend, für jeden Pool um Eins inkrementiert wird. Jede generierte Subklasse von `EnumPool` hat einen Konstruktor, in dem der Konstruktor des Supertyps aufgerufen wird. Als Name wird der Name des `enum`-Typs übergeben und als bekannte Felder werden die Namen der `enum`-Konstanten in der Reihenfolge ihrer Deklaration übergeben.

Generierte `enum`-Pools haben die Methoden `addField` und `addKnownField`. Sie bekommen einen String übergeben, vergleichen ihn mit den Namen aller bekannten `enum`-Konstanten und instanziieren bei einer Übereinstimmung den zur `enum`-Konstanten gehörenden Feldrepräsentanten. Vor jeder Instanziierung wird `addProxy` aufgerufen, um den zugehörigen `enum`-Proxy zum `enum`-Pool hinzuzufügen. Die einzige Ausnahme davon ist der default-Fall in `addField`. Dort wird `addField` der Superklasse `EnumPool` aufgerufen, daher wird `addProxy` nicht aufgerufen.

3.4.4 Pools der Nutzertypen

Ein Pool, der zu einem Nutzertyp mit `enum`-Feldern gehört, muss so generiert werden, dass bei jedem Aufruf der Nutzertyp-Konstruktoren die richtigen Parameter übergeben werden. Das betrifft die Methoden `allocateInstances`, `build` und beide Varianten von `make`. Für den Typ `T` aus Listing 1.1

Listing 3.5 Feldrepräsentanten von `f` ohne Felddatenserialisierungsmethoden

```
1  static final class f0 extends KnownDataField<EnumProxy, bsp.T> {
2      public f0(FieldType<EnumProxy> type, P0 owner) {
3          super(type, "f", owner);
4          if (type.typeID() < 32 || !((StoragePool<?,?>)type).name().equals("farbe"))
5              throw new SkillException("Expected field type farbe in T.f but found " + type);
6      }
7      public EnumProxy get(SkillObject ref) { return ((bsp.T) ref).f; }
8      public void set(SkillObject ref, EnumProxy value) {
9          if(!((SkillState) owner.owner()).Farbes().hasProxy(value))
10                 throw new IllegalArgumentException("value does not match any constant of farbe");
11         ((bsp.T) ref).f = value;
12     }
13 }
```

sind `make()` und `make(int i, bsp.Farbe f)` die Varianten der `make`-Methode. Mit `make()` wird eine Instanz von `T` mit Defaultwerten erstellt. Der Defaultwert des `enum`-Feldes muss dem Konstruktor von `T` als Parameter übergeben werden. Um den Defaultwert zu erhalten nutzt `make()` die Methode `getDefault` aus `EnumPool`. Mit `make(int i, bsp.Farbe f)` wird eine Instanz von `T` mit den übergebenen Werten erstellt. Um den `enum`-Proxy zu `f` zu erhalten, wird `getProxy` aus `EnumPool` verwendet. In `allocateInstances` und `build` werden, wie in `make()`, die Defaultwert-Konstruktoren aufgerufen.

3.4.5 Feldrepräsentanten der `enum`-Felder

Die Feldrepräsentanten von `enum`-Feldern sind gleich aufgebaut wie die Feldrepräsentanten von normalen Feldern. Sie haben den Feldtyp `EnumProxy`. Listing 3.5 zeigt den für das Feld `f` aus Listing 1.1 generierten Feldrepräsentanten. Die Felddatenserialisierungsmethoden sind nicht mit dargestellt.

Die Methode `get` kann den Wert des repräsentierten Feldes direkt zurückgeben. Die Methode `set` muss, bevor sie das repräsentierte Feld auf `value` setzt, mit `hasProxy` aus `EnumPool` überprüfen, ob der `enum`-Proxy zum `enum`-Typ des Feldes gehört.

Im Konstruktor wird überprüft, ob der übergebene dem erwarteten Feldtyp entspricht. Da `enum`-Typen von Pools verwaltet werden, wird der Name des Pools verwendet, um den Feldtyp zu überprüfen. Pools haben die Methode `name`, die den SKilL-Namen ihres Typs zurückgibt. Da vordefinierte Feldtypen diese Methode nicht haben, muss `type` zuerst in einen `StoragePool` konvertiert werden. Davor muss überprüft werden, ob `type` ein vordefinierter Feldtyp ist, da bei dem Versuch, einen vordefinierten Feldtyp in einen Pool zu konvertieren, eine `ClassCastException` geworfen wird, bevor die eigentlich gewollte `SkillException` geworfen werden kann. Der hintere Teil der Überprüfung ist von den Feldrepräsentanten normaler Felder übernommen. Der vordere Teil wurde für die Feldrepräsentanten von `enum`-Feldern hinzugefügt, ist aber auch bei den Feldrepräsentanten von Nutzertyp-Feldern nötig, da es auch bei diesen vorkommen kann, dass ein Feldtyp falsch und einer der vordefinierten Feldtypen ist.

Die nicht dargestellten Felddatenserialisierungsmethoden sind `rsc`, `osc` und `wsc`. Sie iterieren über alle Instanzen von `T` und tun dabei in jeder Iteration das im Folgenden beschrieben. `rsc` liest mit `p.getProxy((int) in.v64())` eine ID aus einer Austauschdatei, erfragt beim `enum`-Pool `p` den `enum`-

Proxy mit dieser ID und setzt das Feld `f` der aktuellen Instanz von `T` auf diesen Wert. `osc` berechnet mit `V64.singleV64Offset(d[i].f.getSkillID())` den Offset des Felddatums von `f` der i-ten Instanz von `T`. `wsc` schreibt mit `out.v64(d[i].farb.getSkillID())` das Felddatum von `f` der i-ten Instanz von `T` in die Austauschdatei. Das Array `d` enthält alle Instanzen von `T`. In den Methoden `osc` und `wsc` wird nicht überprüft, ob `f` den Wert `null` hat. Das ist nicht nötig, da die Konstruktoren der Nutzertypklassen und die Setter der `enum`-Felder so implementiert sind, dass `enum`-Felder nicht auf `null` gesetzt werden können. Die in `rsc` eingelesene ID muss auch nicht überprüft werden, da `getProxy` aus `EnumPool` einen Fehler wirft, wenn es keinen `enum`-Proxy mit der übergebenen ID gibt.

3.4.6 Feldrepräsentanten der `enum`-Konstanten

Die Feldrepräsentanten von `enum`-Konstanten sind, anders als die Feldrepräsentanten von Feldern, nicht mit `f`i, sondern mit `i`i benannt. i is eine ganze Zahl und wird bei 0 beginnend für jeden Feldrepräsentanten einer `enum`-Konstanten inkrementiert.

Der Feldtyp von Feldrepräsentanten von `enum`-Konstanten ist `java.lang.Byte` und als besitzender Typ wird `EnumProxy` verwendet. Die generierten Feldrepräsentanten sind also Subklassen von `KnownDataField<java.lang.Byte, EnumProxy>`. Die Felddatenserialisierungsmethoden `rsc`, `osc` und `wsc` werden mit leeren Methodenkörpern generiert. Die Methoden `get` und `set` werfen einen `NochSuchMethodError`, wenn sie aufgerufen werden.

3.4.7 Änderungen an `Parser`

Die Klasse `Parser` wird als Subklasse `common.internal.FileParser` generiert und überschreibt die dort abstrakte Methode `newPool`. Diese wird im Deserialisierungsvorgang aufgerufen, um Pools zu erstellen. `Parser` muss erweitert werden, damit `newPool` auch mit `enum`-Typen umgehen kann.

Bisher gibt es in `Parser` zwei Methoden `newPool`. Die erste ist die, die `Parser` von `FileParser` erbt. Einer ihrer Parameter ist die Menge der Typ-Restriktionen eines Typs. Die zweite `newPool`-Methode wird von der ersten aufgerufen und bekommt die Typ-Restriktionen nicht übergeben. Da die zweite `newPool`-Methode die Typ-Restriktionen nicht kennt, muss die Unterscheidung zwischen `enum`-Typen und Nutzertypen in der ersten `newPool`-Methode stattfinden. Dort ist `restrictions` die Menge der Typ-Restriktionen. Mit `restrictions.contains(EnumRestriction.get())` kann die die erste `newPool`-Methode heraus finden, ob ein `enum`-Pool oder ein normaler Pool instanziiert werden soll. Für einen normalen Pool wird die zweite `newPool`-Methode aufgerufen und für `enum`-Pools wird die neue Methode `newEnumPool` aufgerufen. `newEnumPool` wird für die Spezifikation aus Listing 1.1 wie in Listing 3.6 generiert.

Die Methode `newEnumPool` vergleicht `name` mit den Namen aller bekannten Typen. Der Vergleich mit den Namen der bekannten `enum`-Typen ist notwendig, um die Pool-Klasse zu bestimmen, von der eine neue Instanz erstellt werden muss. Diese Instanz repräsentiert und verwaltet dann den `enum`-Typ mit dem Namen `name`. Der Vergleich mit den Namen der bekannten Nutzertypen ist notwendig, um sicherzustellen, dass die Anbindung keinen Nutzertyp mit dem Namen `name` kennt. Ohne den Vergleich mit den Namen der Nutzertypen, kann es passieren, dass ein `enum`-Pool mit dem Namen eines Nutzertyps erstellt wird, was nicht sein darf. Die Überprüfung des Supertyps im `default`-Fall ist notwendig, da `enum`-Typen keine Vererbungsbeziehung zu anderen Typen haben dürfen.

Listing 3.6 `newPool` und `newEnumPool` aus `internal.Parser`

```
1  static <T extends B, B extends SkillObject, P extends StoragePool<T, B>> P newPool(String name,
2          StoragePool<?, ?> superPool, ArrayList<StoragePool<?, ?>> types) {
3      try {
4          switch (name) {
5              case "t": return (P) (superPool = new P0(types.size()));
6              case "farbe": throw new SkillException(...);
7              default:
8                  if (null == superPool) return (P) (superPool = new BasePool<T>(types.size(), name,
9                          StoragePool.noKnownFields, noAutoFields()));
10                 else if (superPool instanceof EnumPool)
11                     throw new SkillException("Cannot make Subpool of EnumPools");
12                 else return (P) (superPool = superPool.makeSubPool(types.size(), name));
13         }
14     } finally { types.add(superPool); }
15 }
16
17 static <T extends B, B extends SkillObject, P extends StoragePool<T, B>> P newEnumPool(String name,
18         StoragePool<?, ?> superPool, ArrayList<StoragePool<?, ?>> types) {
19     try {
20         switch (name) {
21             case "farbe": return (P) (superPool = new P1(types.size()));
22             case "t": throw new SkillException(...);
23             default:
24                 if (null == superPool) return (P) (superPool = new EnumPool<T>(types.size(), name,
25                         StoragePool.noKnownFields, noAutoFields()));
26                 else throw new SkillException("Cannot make Subpool of EnumPools");
27         }
28     } finally { types.add(superPool); }
29 }
```

Listing 3.6 enthält auch die Veränderungen der zweiten `newPool`-Methode. Der Änderungsgrund ist derselbe wie bei `newEnumPool`, nur mit vertauschten Rollen der `enum`- und der normalen Pools.

3.4.8 Änderungen an `SkillState`

Die Klasse `SkillState` hat Felder und Zugriffsmethoden für die Pools aller bekannten Typen, auch für die Pools der bekannten `enum`-Typen. Jedes dieser Felder wird im Konstruktor von `SkillState` initialisiert. Der Konstruktor muss so generiert werden, dass für alle `enum`-Typen `newEnumPool` anstelle von `newPool` aufgerufen wird.

4 Implementierung in C++

Dieses Kapitel befasst sich mit der Umsetzung der neuen Implementierung der `enum`-Typen in C++. Zuerst werden die neu hinzugefügten Bibliotheksklassen `EnumProxy` und `EnumPool` beschrieben. Danach wird beschrieben, welche Änderungen an den bereits vorhandenen Bibliotheksklassen vorgenommen werden müssen. Zuletzt wird der generierte Code beschrieben. Die ebenfalls neu hinzugefügten Bibliotheksklassen `EnumProxyIterator` und `EnumRestriction` werden nicht betrachtet, da sie die Designentscheidungen aus Kapitel 2 direkt umsetzen. `EnumProxyIterator` liegt im Namensraum `::skill::iterators` und ist eine Subklasse von `std::iterator` (siehe [ISO12] §24) und `EnumRestriction` liegt im Namensraum `::skill::restrictions` und entspricht dem Singleton-Entwurfsmuster (siehe [GHJV15] §3.5).

In diesem Kapitel werden an UML-Klassendiagramme (siehe [BRJ05] §8) angelehnte Darstellungen verwendet, um eine Übersicht über die Funktionen von Klassen zu geben. Es werden Listings verwendet, um einzelne Funktionen oder Codeabschnitte darzustellen. Um der Lesbarkeit Willen und um Platz zu sparen, wird in den Diagrammen und Listings mehrfach auf voll qualifizierte Klassen- und Funktionsnamen verzichtet. Aus denselben Gründen sind `const`-Modifikatoren nicht dargestellt und Fehlermeldungen von Ausnahmen gekürzt. In den Listings und Diagrammen kommen `SKilLID` und `TypeID` vor. Dabei handelt es sich um Datentypen von SKilL-IDs und Typ-IDs.

4.1 `EnumProxy`

Die Klasse `EnumProxy` liegt im Namensraum `skill::api`. Sie erbt von `skill::api::Object`, der Klasse der SKilL-Objekte in der C++-Implementierung. Abbildung 4.1 zeigt einen Überblick über die Funktionen und Felder von `EnumProxy`. Das Feld `owner` enthält eine Referenz auf den `enum`-Pool, zu dem ein `enum`-Proxy gehört, `typeName` ist der Name des `enum`-Proxys und `id` seine ID. Die Funktionen `getOwner`, `skillName` und `skillID` geben die Werte dieser Felder zurück.

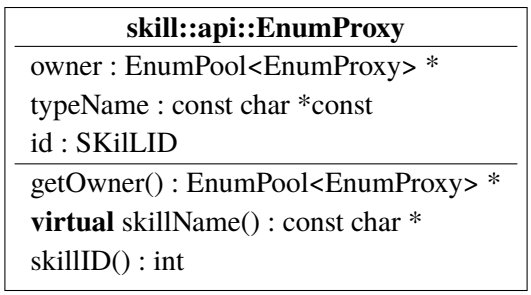

Abbildung 4.1: EnumProxy.h

`EnumProxy` hat drei Konstruktoren. Alle drei sind `protected`. Die Klassen `EnumPool`, `StoragePool` und `Book` sind Freunde von `EnumProxy`, damit sie auf die Konstruktoren zugreifen können. `StoragePool` verwendet `EnumProxy(SKilLID id)` und `Book` verwendet `EnumProxy()`. `EnumPool` verwendet `EnumProxy(SKilLID id)` und `EnumProxy(SKilLID id, char *type, EnumPool<EnumProxy> *owner)`.

4.2 `EnumPool`

Abbildung 4.2 zeigt einen Teil der Felder und Funktionen von `EnumPool`. Die Implementierungen einiger Funktionen stehen in Listing 4.1, 4.2 und 4.3. Die Funktionen `get`, `getAsAnnotation`, `makeSubPool`, `allocateInstances` und `newObjectsSize` sind in Abbildung 4.2 nicht enthalten. Sie sind die Funktionen, die `EnumPool` nicht implementieren darf. Die ersten drei werfen einen `std::bad_function_call`, wenn sie aufgerufen werden. Die letzten beiden tun dies nicht, da sie bei der Serialisierung aufgerufen werden und diese, wenn `allocateInstances` oder `newObjectsSize` einen Fehler werfen würden, für `enum`-Typen nicht mehr funktionieren würde. Darum sind `allocateInstances` und `newObjectsSize` so implementiert, das erstere nichts tut und letztere immer den Wert `0` zurückgibt.

Die Felder `idToOrdinal` und `ordinalToId` werden verwendet, um bekannte `enum`-Konstanten und `enum`-Proxys einander zuzuordnen. Bekannte `enum`-Konstanten gibt es nur für bekannte `enum`-Typen. Diese werden von generierten Subklassen von `EnumPool` repräsentiert. In `EnumPool` selbst gibt es

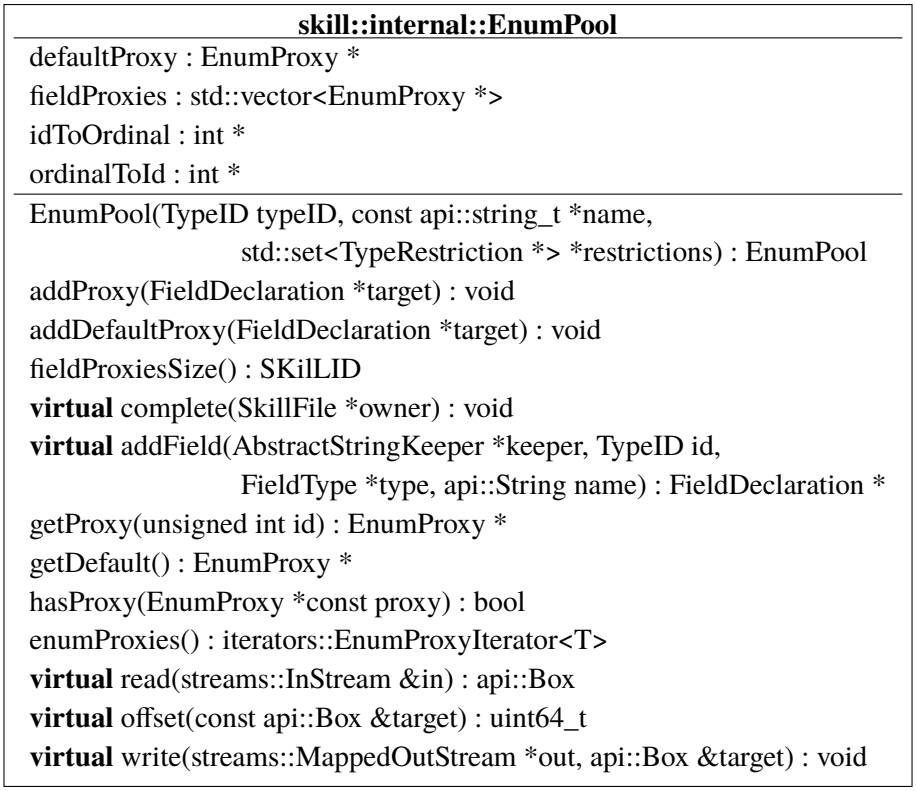

Abbildung 4.2: EnumPool.h

Listing 4.1 getProxy und hasProxy aus EnumPool.h

```
1  EnumProxy* getProxy(unsigned int id) const {
2      unsigned int idx = ID - 1;
3      if (idx < 0 || fieldProxies.size() <= idx) throw std::invalid_argument("No proxy with that id.");
4      return fieldProxies.at(idx);
5  }
6  bool hasProxy(EnumProxy *const proxy) {
7      for (auto p : fieldProxies)
8          if (p == proxy) return true;
9      return false;
10 }
```

keine Verwendung für die beiden Felder, sie bleiben uninitialisiert. In Abschnitt 4.4.2 und 4.4.3 wird darauf eingegangen, wie `idToOrdinal` und `ordinalToId` funktionieren und warum sie trotzdem schon in `EnumProxy` deklariert sind.

Mit der Funktion `hasProxy` kann überprüft werden, ob ein `enum`-Proxy zu einem `enum`-Pool gehört. Mit den Funktionen `getProxy` und `getDefault` kann auf die `enum`-Proxys eines Pools zugegriffen werden. `getDefault` gibt den Defaultwert zurück. `getProxy` gibt, falls vorhanden, den `enum`-Proxy mit der ID `id` zurück. `hasProxy` und `getProxy` sind wie in Listing 4.1 implementiert.

Die Felddatenserialisierungsfunktionen `read`, `offset` und `write` sind analog zu den von ihnen überschriebenen Funktionen aus `AbstractStoragePool` implementiert. Sie unterscheiden sich dadurch, dass sie in `EnumPool` `enum`-Proxys anstatt Nutzertypinstanzen serialisieren. Das hat zur Folge, dass das Felddatum, wenn es beim Schreiben `nullptr` ist, durch den Defaultwert ersetzt wird. Das heißt `offset` berechnet dann mit `fieldTypes::V64FieldType::offset(defaultProxy->skillID())` den Offset für den Defaultwert und in `write` schreibt `out->v64(defaultProxy-skillID())` den Defaultwert. `read` gibt keine Nutzertypinstanzen, sondern `enum`-Proxys zurück. `read` muss außerdem die eingelesene ID nicht mehr überprüfen. Wenn sie kein zulässiges Felddatum ist, wird ein Fehler geworfen, sobald `read` auf die `enum`-Proxys des Pools zugreift.

In `fieldProxies` sind die `enum`-Proxys eines Pools hinterlegt. Die Funktion `fieldProxiesSize` gibt die Länge von `fieldProxies` zurück und die Funktion `enumProxies` gibt einen Iterator über `fieldProxies` zurück. `fieldProxies` muss eine Datenstruktur von variabler Größe sein. Sie ist daher ein `std::vector`. Die variable Größe ist notwendig, da die `enum`-Proxys nacheinander zu `fieldProxies` hinzugefügt werden und die Gesamtanzahl der `enum`-Proxys eines Pools nicht immer von Anfang an bekannt ist. Mit den Funktionen `addProxy` und `addDefaultProxy` aus Listing 4.2 werden `enum`-Proxys hinzugefügt. Sie werden in `addField` jedes Mal, nachdem ein neuer Feldrepräsentant instanziiert wurde, aufgerufen. Als Parameter `target` wird der neue Feldrepräsentant übergeben. Der neue `enum`-Proxy und der Feldrepräsentant `target` repräsentieren also dieselbe `enum`-Konstante. `addProxy` wird verwendet um einen normalen `enum`-Proxy hinzuzufügen, `addDefaultProxy` wird verwendet, wenn der neue `enum`-Proxy der Defaultwert ist. In diesem Fall muss der neue Proxy so instanziiert werden, dass seine Adresse, die des zuvor als Defaultwert verwendeten Dummy-Proxys ist. `defaultProxy` wird im Konstruktor von `EnumPool` so initialisiert, dass das Feld die Adresse eines `enum`-Proxys mit der SKilL-ID -1 enthält. Dieser Proxy wird nicht zu `fieldProxies` hinzugefügt.

Die Funktion `addDefaultProxy` wird nur von generierten `addField`-Funktionen aufgerufen. Bei unbekannten `enum`-Typen wird die Funktion `addField` von `EnumPool` selbst aufgerufen, die wie in Listing 4.2 implementiert ist und `addDefaultProxy` nicht aufruft. Das heißt, der Defaultwert von

Listing 4.2 `addProxy`, `addDefaultProxy` und `addField` aus EnumPool.h

```
1  void addProxy(FieldDeclaration *target) {
2      fieldProxies.push_back(new EnumProxy(target->index,target->name->c_str(),this));
3  }
4  void addDefaultProxy(FieldDeclaration *target) {
5      fieldProxies.push_back(new(defaultProxy) EnumProxy(target->index,target->name->c_str(),this));
6  }
7  FieldDeclaration *addField(AbstractStringKeeper *keeper, TypeID id, FieldType *type, String name) {
8      auto rval = new internal::LazyField(type, name, (SKilLID)this->dataFields.size() + 1, this);
9      this->dataFields.push_back(rval);
10     this->addProxy(rval);
11     return rval;
12 }
```

Listing 4.3 `complete` aus EnumPool.h

```
1      virtual void complete(SkillFile *owner) {
2          this->owner = owner;
3          defaultProxy = getProxy(1);
4      }
```

unbekannten `enum`-Typen muss an anderer Stelle richtig gesetzt werden. Dazu wird die Funktion `complete` verwendet. Wenn diese Funktion bei der Zustandserstellung aufgerufen wird, kennen die Pools von unbekannten `enum`-Typen schon alle ihre Feldrepräsentanten und `enum`-Proxys. Sowohl die Klasse `EnumPool` als auch ihre generierten Subklassen überschreiben sie.

Wenn `complete` von `EnumPool` aufgerufen wird, bedeutet das, dass ein `enum`-Pool einen unbekannten `enum`-Typ hat. Das wiederum impliziert zwei Dinge. Erstens wurde `addDefaultProxy` nicht aufgerufen und der Defaultwert ist immer noch der Dummy-Proxy mit ID -1. Zweitens kann keiner der bekannten Nutzertypen ein Feld vom `enum`-Typ dieses Pools haben, sodass bisher noch nicht auf `defaultProxy` zugegriffen wurde und das Feld beliebig neu gesetzt werden kann. Unter diesen Voraussetzungen kann `complete` in `EnumPool` wie in Listing 4.3 implementiert werden.

4.3 Anpassungen bereits vorhandener Bibliotheksklassen

Die meisten Komponenten der C++-Bibliothek bleiben unverändert. Nur `FileWriter`, `LazyField` und die Funktion `parseFile` müssen erweitert werden.

`skill::internal::parseFile` Die Funktion `parseFile` muss so erweitert werden, dass sie auch mit `enum`-Restriktionen umgehen kann. Typ-Restriktion werden mit einer `switch`-Anweisung über ihre ID bestimmt. Für `enum`-Restriktionen muss die `switch`-Anweisung um einen Fall für die ID 7 erweitert werden, in dem mit `rest->insert(EnumRestriction::getInstance())` die `enum`-Restriktion zu `rest`, der Menge der Restriktionen des im Moment eingelesenen Typs, hinzugefügt wird. Da `enum`-Typen in der SKilL-Spezifikationssprache keine Typ-Restriktionen haben (siehe [Fel17a] §5.4.3), muss sichergestellt werden, dass ein Typ mit `enum`-Restriktion keine weiteren Restriktionen hat.

skill::internal::FileWriter In FileWriter muss die Funktion zum Serialisieren von Typ-Restriktionen so erweitert werden, dass sie auch enum-Restriktionen in Austauschdateien schreiben kann. Jeder Pool kennt die Menge seiner Typ-Restriktionen. Wenn diese eine enum-Restriktion enthält, werden die Bytes 01 und 07 in die Austauschdatei geschrieben. Als Anzahl kann immer 01 geschrieben werden, da beim Lesen sichergestellt wird, dass ein enum-Typ keine weiteren Typ-Restriktionen hat.

skill::internal::LazyField In den Zugriffsmethoden von verteilten Feldern muss, mit demselben Vorgehen wie beim Serialisieren der Typ-Restriktionen, überprüft werden, ob das verteilte Feld zu einem enum-Pool gehört. Wenn dem so ist, wird der Fehler std::bad_function_call geworfen.

4.4 Generierter Code

Dieser Abschnitt befasst sich mit dem generierten Code. Das beinhaltet die für die enum-Typen generierten Enumerationen, die für die Nutzertypen generierten Subklassen von Object, die generierten Pools der enum- und Nutzertypen, die generierten Feldrepräsentanten der enum-Felder und enum-Konstanten, SkillFile und die Funktionen makeState und makePool. Feldrepräsentanten von Feldern, die keine enum-Felder sind, sowie Pools und Klassen von Nutzertypen, die keine enum-Felder haben, werden nicht betrachtet, da sie von der neuen Implementierung der enum-Typen nicht verändert werden.

4.4.1 Enumerationen

Für jeden enum-Typ der Spezifikation wird eine C++-Enumeration generiert. Es werden *scoped enumerations* (siehe [ISO12] §7.2) mit Konstanten vom Typ int verwendet. Die generierten Konstanten haben dieselbe Reihenfolge wie die enum-Konstanten in der Spezifikation. Die Werte der enum-Konstanten beginnen mit 0 und werden für jede weitere enum-Konstante inkrementiert. Die enum-Konstanten sind in CamelCase, da es zu viele verschiedene C++-Styleguides gibt um sich für einen zu entscheiden und CamelCase im Einklang mit dem Styleguide von SKilL ist (siehe [Fel17b] §2.4). Alle Enumerationen werden in die Datei Enums.h generiert.

4.4.2 Nutzertypklassen

Listing 4.4 zeigt die für den Nutzertyp T aus Listing 1.1 generierte Klasse. Das Feld und die Zugriffsmethoden für i sowie die Funktionen und Felder, die durch die Existenz von enum-Feldern nicht beeinflusst werden, sind nicht mit abgebildet.

Die Funktionen getF und setF sind nach demselben Schema benannt, wie die Zugriffsmethoden von normalen Feldern. Sie werden verwendet um auf _f zuzugreifen, als wäre es tatsächlich vom Typ Farbe. In Zeile 12 in Listing 4.4 überprüft getF, ob _f zu einer bekannten enum-Konstanten gehört. Die Überprüfung funktioniert, da idToOrdinal für alle unbekannten enum-Konstanten den Wert -1 enthält. setF prüft, ob die übergebene enum-Konstante f im Bereich des enum-Typs liegt. Da für die generierten Enumerationen *scoped enumerations* verwendet werden, ist sichergestellt, dass nur Konstanten des enum-Typs Farbe übergeben werden können. Die Überprüfung in Zeile 17 in

Listing 4.4 Auszug aus TypesOfT.h

```
1  class T : public ::skill::api::Object {
2  protected:
3      ::skill::api::EnumProxy* _f;
4      T() { }
5      T(::skill::SKilLID _skillID, ::skill::api::EnumProxy* __f = nullptr, int32_t __i = 0) {
6          this->id = _skillID;
7          this->_f = __f;
8          this->_i = __i;
9      }
10 public:
11     ::bsp::Farbe getF() const {
12         if(f->getOwner()->idToOrdinal[_f->skillID()] < 0)
13             throw std::out_of_range("_f not in range of Farbe. Use getFAsProxy() to access _f.");
14         return static_cast<::bsp::Farbe>(_f->getOwner()->idToOrdinal[_f->skillID()]);
15     }
16     void setF(::bsp::Farbe f) {
17         if(f < ::bsp::Farbe::rot || ::bsp::Farbe::blau < f)
18               throw std::invalid_argument("no matching constant for argument value");
19         _f = _f->getOwner()->getProxy(_f->getOwner()->ordinalToId[static_cast<int>(f)]);
20     }
21     ::skill::api::EnumProxy* getFAsProxy() const { return _f; }
22 }
```

Listing 4.4 ist trotzdem notwendig, da man mit `static_cast` einen beliebigen Integerwert in eine Konstante des `enum`-Typs `Farbe` konvertieren kann. Wenn eine Konstante übergeben wird, deren Wert nicht im Bereich von `Farbe` liegt, dann liegt er auch nicht im Bereich von `ordinalToId` und das Ergebnis des Zugriffs auf `ordinalToId` in Zeile 21 ist undefiniert. `getFAsProxy` kann `_f` direkt zurückgeben.

Nun lässt sich auch begründen, warum `idToOrdinal` und `ordinalToId` schon in `EnumPool` deklariert wurden. Die Zugriffsmethoden verwenden `getOwner`, um eine Referenz zu den Pools der `enum`-Proxys zu erhalten. Diese Referenz wird von `getOwner` als `EnumPool` zurückgegeben. Wenn `EnumPool` `idToOrdinal` und `ordinalToId` nicht kennen würde, müsste man die Referenz vor jedem Zugriff auf diese Datenstrukturen in ihren tatsächlichen Typ konvertieren. Es kann nicht passieren, dass `getOwner` einen Pool zurückgibt, der nur eine Instanz von `EnumPool` ist, denn dann würde dieser Pool zu einem unbekannten `enum`-Typ gehören und von einem solchen kann es keine `enum`-Felder in bekannten Nutzertypen geben.

Der Konstruktor bekommt die Werte für die `enum`-Felder als `enum`-Proxy übergeben. Da in `allocateInstances` in `StoragePool` Nutzertypen instanziiert werden, obwohl die Werte ihrer Felder noch unbekannt sind, muss `nullptr` als Defaultwerte für die `enum`-Felder verwendet werden. Solange die `enum`-Felder noch vor Fertigstellung des Zustands den richtigen Defaultwert zugewiesen bekommen, ist das kein Problem. Dies geschieht in der Funktion `complete` der generierten Pools von Nutzertypen und wird in Abschnitt 4.4.4 erläutert.

Listing 4.5 Initialisierung von `idToOrdinal` und `ordinalToId` in `complete` aus FarbePool.cpp

```
1    idToOrdinal = new int[this->dataFields.size()+1];
2    ordinalToId = new int[static_cast<int>(::bsp::Farbe::blau)+1];
3    idToOrdinal[0] = -1;
4    for (int i = 0; i < this->dataFields.size(); ++i) {
5        int ordinal = inithelp.at(i);
6        if (-1 != ordinal)
7            ordinalToId[ordinal] = i+1;
8        idToOrdinal[i+1] = ordinal;
```

4.4.3 Pools der `enum`-Typen

Soweit nicht anders erwähnt, beziehen sich die Begriffe `enum`-Proxy, Feldrepräsentant und `enum`-Konstante in diesem Abschnitt nur auf die Proxys, Feldrepräsentanten und `enum`-Konstanten eines `enum`-Typs oder -Pools. Für jeden `enum`-Typ einer Spezifikation wird eine Subklasse von `EnumPool` generiert. Der Klassenname ist die Konkatenation des Names des `enum`-Typs und des Strings *Pool*, analog zu den für Nutzertypen generierten Pools. Die generierten Subklassen überschreiben die Funktionen `complete` und `addField` von `EnumPool` und haben ein neues Feld namens `inithelp` vom Typ `std::vector<int>`.

Die Funktionen `complete` und `addField` erfüllen in `enum`-Pools dieselbe Aufgaben wie in Pools von Nutzertypen. Mit `addField` werden Feldrepräsentanten für die aus einer Austauschdatei eingelesenen Felder zum Pool hinzugefügt. Mit `complete` wird sichergestellt, dass der Pool Feldrepräsentanten für alle bekannten Felder hat. In `enum`-Pools müssen sie noch mehr Aufgaben übernehmen.

Als Erstes muss `complete` um den Code aus Listing 4.5 erweitert werden, um `idToOrdinal` und `OrdinalToId` zu initialisieren. Er muss eingefügt werden, nachdem die Feldrepräsentanten aller bekannten Felder zum Pool hinzugefügt wurden. Wenn zum Initialisierungszeitpunkt von `idToOrdinal` und `OrdinalToId` noch Feldrepräsentanten fehlen würden, würden diese auch in der durch `idToOrdinal` und `OrdinalToId` realisierten Abbildung fehlen und könnten nicht zu ihren `enum`-Konstanten zugeordnet werden.

Das Array `idToOrdinal` bildet die IDs der `enum`-Proxys auf die Werte der `enum`-Konstanten ab. Da SKilL-IDs mit dem Wert `1` beginnen und es aufwändig wäre, immer mit einem Offset von -1 auf `idToOrdinal` zuzugreifen, ist das Array um ein Element länger, als es Feldrepräsentanten gibt. Da die Anzahl der Feldrepräsentanten gleich der Anzahl der `enum`-Proxys ist, gibt es in `idToOrdinal` so genug Platz für alle `enum`-Proxys. Das erste Element von `idToOrdinal` wird auf `-1` gesetzt, damit man einen definierten Wert, der trotzdem keiner `enum`-Konstanten entspricht, erhält, falls fälschlicherweise auf dieses Element zugegriffen wird. Im Normalfall sollte das jedoch nicht vorkommen, da es keinen `enum`-Proxy mit der ID `0` gibt.

Das Array `ordinalToId` bildet die `enum`-Konstanten auf die IDs der `enum`-Proxys ab. Die `enum`-Konstanten beginnen mit dem Wert `0` und der Wert der letzten `enum`-Konstanten ist um Eins kleiner als die Gesamtanzahl an Konstanten. Daher bietet `ordinalToId` in Listing 4.5 genug Platz für alle `enum`-Konstanten. Zumindest wenn `blau` die letzte Konstante von `::bsp::Farbe` ist. Da sich dieser Abschnitt mit generiertem Code befasst und der Quellcode-Generator weiß, in welcher Reihenfolge und mit welchen Werten er die `enum`-Konstanten generiert hat, ist ihm die letzte Konstante jedes `enum`-Typs bekannt.

Listing 4.6 Hinzufügen eines Feldrepräsentanten in `complete` in FarbePool.cpp

```
1 if (fields.end() != fields.find(sk->blau)){
2     dataFields.push_back(new ::bsp::internal::KnownField_Farbe_blau(&::skill::fieldTypes::I8, sk->blau,
3         dataFields.size() + 1, this));
4     inithelp.push_back(static_cast<int>(::bsp::Farbe::blau));
5     addProxy(dataFields.back());
6 }
```

Listing 4.7 Hinzufügen eines Feldrepräsentanten in `addField` aus FarbePool.cpp

```
1 if (name == sk->blau) {
2     target = new ::bsp::internal::KnownField_Farbe_blau(type, name, id, this);
3     inithelp.push_back(static_cast<int>(::bsp::Farbe::blau));
4     addProxy(target);
5 }
```

Nun stellt sich noch die Frage, woher `complete` weiß, mit welchen Werten `idToOrdinal` und `OrdinalToId` befüllt werden müssen. Dazu wird der Vektor `inithelp` benötigt. Jedes Mal, wenn ein Feldrepräsentant mit `addField` oder in `complete` hinzugefügt wird, wird der Wert der zugehörigen `enum`-Konstanten an `inithelp` angehängt. Wenn der Feldrepräsentant einer unbekannten `enum`-Konstanten hinzugefügt wird, wird der Wert `-1` an `inithelp` angehängt. Nachdem alle Feldrepräsentanten hinzugefügt wurden, enthält `inithelp` die `enum`-Konstanten in der Reihenfolge ihrer Repräsentanten und kann wie in Listing 4.5 verwendet werden, um `idToOrdinal` und `OrdinalToId` zu befüllen.

Als Zweites müssen sowohl `complete` als auch `addField` so erweitert werden, dass die `enum`-Proxys zum Pool hinzugefügt und die Werte der `enum`-Konstanten an `inithelp` angehängt werden. Die für die `enum`-Konstante `blau` aus Listing 1.1 verantwortlichen Codeabschnitte der beiden Funktionen sehen wie in Listing 4.7 und 4.6 aus. Wenn `blau` der Defaultwert wäre, würde in diesen Abschnitten `addDefaultProxy` an Stelle von `addProxy` aufgerufen werden.

4.4.4 Pools der Nutzertypen

Die für Nutzertypen mit `enum`-Felder generierten Pool-Klassen unterscheiden sich von denen ohne `enum`-Felder durch die Implementierung der Funktionen `complete` und `add`. Der Konstruktor und die beiden Funktionen `makeSubPool` und `addField` bleiben unverändert. Die Funktionen `complete` und `add` müssen für Nutzertypen mit `enum`-Feldern dafür sorgen, dass die `enum`-Felder für alle Instanzen des Nutzertyps die richtigen Defaultwerte haben. Dabei ist `complete` für Instanzen verantwortlich, die aus einer Austauschdatei eingelesen werden und `add` für Instanzen, die nach der Zustandserstellung hinzugefügt werden.

Beim Einlesen einer Austauschdatei werden mit `allocateInstances` aus `StoragePool` zu jedem Pool die Instanzen seines Typs hinzugefügt. Dabei wird dem Konstruktor des Typs nur die SKilL-ID der Instanz übergeben, die restlichen Felder des Typs erhalten die im Konstruktor definierten Defaultwerte. Für `enum`-Felder ist das `nullptr` (siehe Abschnitt 4.4.2), ein Wert, den `enum`-Felder nicht annehmen dürfen. Nun sind entweder alle der Anbindung bekannten `enum`-Felder auch in der Austauschdatei enthalten oder die Anbindung kennt `enum`-Felder, die in der Austauschdatei nicht enthalten sind. Im ersten Fall ist es kein Problem, dass die `enum`-Felder zeitweilig auf `nullptr` gesetzt

Listing 4.8 Hinzufügen eines Feldrepräsentanten in `complete` aus TPool.cpp

```
1 if (fields.end() != fields.find(sk->f)){
2     dataFields.push_back(new ::bsp::internal::KnownField_T_f(state->Farbe, sk->f,
3             dataFields.size() + 1, this));
4     for (auto &instance : *this) instance._f = ((::bsp::api::SkillFile *) owner)->Farbe->getDefault();
5 }
```

Listing 4.9 `add` aus TPool.cpp

```
1 T* add(::bsp::Farbe f = ::bsp::Farbe::rot, i = 0){
2     if(f < ::bsp::Farbe::rot || ::bsp::Farbe::blau < f)
3         throw std::invalid_argument("argument does not correspond to known constant of farbe.");
4     T* rval = book->next();
5     ::bsp::FarbePool *fpool = ((::bsp::api::SkillFile *) owner)->Farbe;
6     new(rval) T(-1 ,i ,fpool->getProxy(fpool->ordinalToId[static_cast<int>(f)]));
7     this->newObjects.push_back(rval);
8     return rval;
9 }
```

werden. Da die Austauschdatei Felddaten für alle `enum`-Felder enthält, werden letztlich alle `enum`-Felder auf diese Werte gesetzt. Im zweiten Fall ist `nullptr` ein Problem. Die Austauschdatei enthält für die ihr nicht bekannten `enum`-Felder keine Felddaten und die Felder würden nach Fertigstellung des Zustands immer noch den Wert `nullptr` haben. An dieser Stelle muss `complete` eingreifen. Wenn der Repräsentant eines Feldes erst in `complete` hinzugefügt wird, ist das zugehörige Feld nicht in der Austauschdatei enthalten. Darum kann jede `if`-Anweisung, die die Existenz des Repräsentanten eines `enum`-Feldes sicherstellt, um die Zeilen 4 aus Listing 4.8 erweitert werden. Listing 4.8 zeigt den Code für das Feld `f` aus Listing 1.1.

In `StoragePool` gibt es eigentlich schon eine Funktion `add` um Instanzen von Nutzertypen zu erstellen. Für Nutzertypen mit `enum`-Feldern kann dieses `add` aber nicht verwendet werden, da `StoragePool` nicht zusichern kann, dass die `enum`-Felder die richtigen Defaultwerte bekommen. Für Nutzertypen mit `enum`-Feldern muss `add` also generiert werden. Für den Nutzertyp `T` aus Listing 1.1 sieht `add` wie in Listing 4.9 aus. Die Funktion hat Defaultwerte für alle Parameter, damit auch Instanzen erstellt werden können, ohne Werte für alle Felder zu übergeben. Außerdem bekommt sie `enum`-Konstanten als Werte für die Felddaten übergeben und muss, so wie der Setter, überprüfen, ob die Konstanten im Bereich des `enum`-Typs liegen und ermitteln, welche `enum`-Proxys zu ihnen gehören.

4.4.5 Feldrepräsentanten der `enum`-Felder

Für `enum`-Felder werden Feldrepräsentanten generiert, die Felder vom Typ `skill::api::EnumProxy` repräsentieren. Für `f` aus Listing 1.1 wird der Feldrepräsentant `KnownField_T_f` aus Listing 4.10 generiert. Der Konstruktor und die Funktionen `read`, `write`, `offset` und `check` sind außerhalb der Klasse implementiert, daher ist deren Implementierung nicht dargestellt.

Listing 4.10 KnownField_T_f.h

```
1  class KnownField_T_f : public ::skill::internal::FieldDeclaration {
2  public:
3      KnownField_T_f(FieldType *type, string_t *name, TypeID index, AbstractStoragePool *owner);
4      virtual bool check() const;
5      virtual void read(MappedInStream *in, Chunk *target);
6      virtual size_t offset() const;
7      virtual void write(MappedOutStream* out) const;
8
9      virtual ::skill::api::Box getR(const ::skill::api::Object *i) {
10         return ::skill::api::box(((::bsp::T*)i)->_f);
11     }
12     virtual void setR(::skill::api::Object *i, ::skill::api::Box v) {
13         if (((::bsp::api::SkillFile *)owner->getOwner())->Farbe->
14                 hasProxy(((::skill::api::EnumProxy *)v.annotation)))
15             ((::bsp::T*)i)->_f = ((::skill::api::EnumProxy *)v.annotation);
16         else throw std::invalid_argument("v does not correspond to any enumproxy of Farbe.");
17     }
18 };
```

Der Konstruktor tut bis auf die Typprüfung dasselbe wie alle Konstruktoren von Feldrepräsentanten. Der Feldtyp von `KnownField_T_f` wird mit `if(!dynamic_cast<const ::bsp::FarbePool * const>(type))` überprüft. `type` ist der dem Konstruktor übergebene Feldtyp und `FarbePool` der für den `enum`-Typ `Farbe` aus Listing 1.1 generierte Pool. Wenn `type` keine Instanz von `FarbePool` ist, wird eine `SkillException` geworfen.

Die Felddatenserialisierungsmethoden iterieren über alle Instanzen des Nutzertyps, zu dem sie gehören und verarbeiten in jeder Iteration das Felddatum der aktuellen Instanz. Dazu verwenden sie die Felddatenserialisierungsmethoden des Feldtyps, also die von `EnumPool`. In `read` werden Felddaten mit `(::skill::api::EnumProxy*)type->read(in).annotation` eingelesen, in `offset` werden mit `type->offset(::skill::box(d[i]->_f))` die Offsets der Felddaten in `_f` berechnet und in `write` werden mit `type->write(out, ::skill::box(d[i]->_f))` die Felddaten von `_f` serialisiert.

Die Funktion `getR` wird so generiert wie in normalen Feldrepräsentanten. In `setR` muss überprüft werden, ob das neue Felddatum eine `enum`-Konstante des richtigen `enum`-Typs ist. Dazu wird die Funktion `hasProxy` aus `EnumPool` verwendet. Die Funktion `check` gibt immer `true` zurück, da es nichts zu überprüfen gibt.

4.4.6 Feldrepräsentanten der `enum`-Konstanten

Die Feldrepräsentanten der `enum`-Konstanten erben von `::skill::internal::FieldDeclaration`. Sie werden nach demselben Schema wie normale Feldrepräsentanten benannt und liegen ebenfalls im Namensraum `internal`. Die Klassennamen von Feldreräsentanten setzten sich aus dem String *KnownField*, dem Namen des `enum`-Typs und dem Namen der Konstanten zusammen. Der Feldrepräsentant der Konstanten `rot` aus Listing 1.1 hat also den Klassennamen `KnownField_Farbe_rot` und liegt im Namensraum `::bsp::internal`.

Im Konstruktor der Feldrepräsentanten wird mit `if(type->typeID != 7)` überprüft, ob der Feldtyp stimmt. Wenn er nicht stimmt, wird eine `SkillException` geworfen.

Die Zugriffsmethoden `setR` und `getR` werfen den Fehler `std::bad_function_call`. Die Funktion `check` gibt immer `true` zurück, da es nichts zu überprüfen gibt.

Die Felddatenserialisierungsfunktionen `read`, `offset` and `write` haben leere Funktionskörper. Sie sind implementiert, da sie in `FieldDeclaration` als `abstract` deklariert sind und daher überschrieben werden müssen. Außerdem werfen sie keine Fehler, da diese beim Serialisieren abgefangen werden müssten.

4.4.7 Änderungen an `makeState`

Die Funktion `makeState` ist unter anderem dafür zuständig, sicherzustellen, dass Pools für alle bekannten Nutzertypen existieren. Mit der Einführung von `enum`-Pools muss sie außerdem sicherstellen, dass `enum`-Pools für alle bekannten `enum`-Typen existieren. Dazu wird für jeden `enum`-Typ noch einmal derselbe Codeblock generiert, der schon für jeden Nutzertyp generiert wird. Bei den `enum`-Typen muss die dem Pool-Konstruktor übergebene Menge an Typ-Restriktionen eine `enum`-Restriktion enthalten.

4.4.8 Änderungen an `makePool`

Die Funktion `makePool` wird aufgerufen, um einen neuen Pool zu instanziieren. Sie bekommt dabei unter anderem den Namen eines Typs, seinen Supertyp und die Menge seiner Typ-Restriktionen übergeben. Anhand dieser Parameter muss sie heraus finden, ob ein aus der Austauschdatei eingelesener Typ einer der bekannten Typen ist. Ursprünglich war es dafür ausreichend `name` mit allen bekannten Typnamen zu vergleichen und `supertyp` zu überprüfen, falls ein passender bekannter Typ gefunden wurde. Mit der Einführung von `enum`-Pools muss dieses Muster so erweitert werden, dass `makePool` die selben Fälle abdeckt wie die Methoden `newPool` und `newEnumPool` der Java-Implementierung (vgl. Listing 3.6).

Die Funktion `makePool` unterscheidet sich von den Methoden der Java-Implementierung insofern, dass sie die Typ-Restriktionen kennt, sodass die `enum`-Pools und die Pools der Nutzertypen nicht in getrennten Funktionen erstellt werden müssen. Stattdessen wird `makePool` so generiert, dass erst überprüft wird, ob der Name des eingelesenen Typs mit dem Namen eines bekannten Typs übereinstimmt. Danach wird überprüft, ob die Typ-Restriktionen und Supertypen übereinstimmen. Wenn alles übereinstimmt, wird der entsprechende Pool erstellt. Andernfalls wird ein Fehler geworfen.

4.4.9 Änderungen an `SkillFile`

Die Klasse `SkillFile` hat für jeden Nutzertyp T ein Feld `TPool *const T`, über das auf den Pool des Typs zugegriffen werden kann. Ein solches Feld muss auch für jeden `enum`-Typ generiert werden.

Die Felder werden im Konstruktor von `SkillFile` initialisiert. Für Nutzertypen-Pools wird dabei auf den Typnamen des Nutzertyps zugegriffen. Aus Ermangelung eines Typs mit zugreifbarem Namen muss bei `enum`-Typen der Name des `enum`-Typs als String generiert werden. Das Feld für einen `enum`-Typ E wird also mit `(EPool *) annotation->type("e")` initialisiert.

5 Änderungen am Codegenerator

Dieses Kapitel befasst sich mit den Änderungen an der Zwischendarstellung des Parsers und mit den Änderungen an den Codegeneratoren für Java und C++. Die meisten Änderungen der Codegeneratoren wurden schon im Rahmen der Abschnitte 3.4 und 4.4 beschrieben. Sie werden hier nicht erneut aufgegriffen. Die Abschnitte 5.2 und 5.3 befassen sich daher mit den Änderungen, die sich nicht direkt im generierten Code widerspiegeln.

5.1 Parser

In der Zwischendarstellung des Parsers gibt es die Klasse `EnumType` zur Darstellung von `enum`-Typen. Früher durften `enum`-Typen Felder haben. Daher implementierte `EnumType` das Interface `WithFields` und musste die von ihm geerbten Methoden `getFields`, `getAllFields`, `getViews` und `getCustomizations` überschreiben.

In dieser Arbeit haben `enum`-Typen keine Felder. Daher implementiert `EnumType` das Interface `WithFields` nicht mehr. Daraufhin können die Methoden `getFields`, `getAllFields`, `getViews`, `getCustomizations` und die Felder `fields`, `views` und `customizations` sowie die Methode `initialize` gelöscht werden. Die Methode `initialize` war dafür verantwortlich, die Felder `fields`, `views` und `customizations` zu initialisieren. Da es diese Felder nicht mehr gibt, ist die Methode überflüssig. Die Methode `isInitalized`, die zuvor überprüft hat, ob `fields` schon initialisiert wurde, gibt jetzt immer `true` zurück, da es in `EnumType` keine Felder mehr gibt, die initialisiert werden müssen. In der Methode `prettyPrint` muss der Zugriff auf `fields` entfernt werden. Der Konstruktor von `EnumType`, die Methoden `copy`, `newDeclaration`, `getInstance`, `isReadOnly`, `isMonotone` und das Feld `instances` bleiben unverändert vorhanden. Da die Methode `initialize` in `EnumType` nicht mehr existiert, muss ihr Aufruf in `IRBuilder` entfernt werden.

Dass `enum`-Typen keine Felder mehr haben, betrifft auch die Klasse `AST`. In `AST` gibt es die Klasse `EnumDefinition`, die bisher eine Subklasse von `DeclarationWithBody` war. Wenn `enum`-Typen keine Felder mehr haben, ist `EnumDefinition` nur noch eine Subklasse von `Declaration`. Das hat Auswirkungen auf `Parser`, da der Konstruktor von `EnumDefinition` jetzt einen Parameter weniger hat, und auf `TypeCheck`, da `EnumDefinition` keinen Körper mehr hat. In `Parser` müssen also alle Konstruktoraufrufe von `EnumDefinition` angepasst werden und in `TypeCheck` kann die betroffene Codezeile gelöscht werden.

Früher hatte der Parser außerdem einen Mechanismus um `enum`-Typen durch Nutzertypen zu ersetzen. Dazu gab es die Klasse `EnumSubstitution`, die eine Subklasse von `Substitution` war, sowie die Methode `removeEnums` der Klasse `TypeContext`. Mit der in Abschnitt 2.1 beschriebenen Repräsentation von `enum`-Typen im Binärformat können `enum`-Typen nicht mehr ersetzt werden, da kein anderer Typ die Felddaten von `enum`-Feldern richtig lesen kann. Ohne Ersetzung für `enum`-Typen kann die Klasse `EnumSubstitution` gelöscht werden. Infolgedessen kann die Methode `addTypes` aus `Substitution`

und allen übrigen Subklassen von `Substitution` entfernt werden. Die Methode `addTypes` erfüllt nur für die Ersetzung von `enum`-Typen einen Zweck, wird also für die übrigen Ersetzungen nicht mehr gebraucht. Die Methode `removeEnums` kann ebenfalls gelöscht werden. Wenn `addTypes` und `removeEnums` nicht mehr existieren, müssen auch alle Aufrufe dieser Methoden entfernt werden. Für `addTypes` betrifft das die Klasse `TypeContext` und für `removeEnums` die Klasse `TypeContext` und die Generatoren für Java, Scala, SIDL und jForeign.

Bisher hat die Ersetzung der `enum`-Typen durch Nutzertypen auch dafür gesorgt, dass `enum`-Konstanten innerhalb eines `enum`-Typs nicht identisch benannt sein dürfen. Ohne die Ersetzung muss dies an anderer Stelle sichergestellt werden. Eine entsprechende Überprüfung der Namen von `enum`-Konstanten wird in der Klasse `TypeCheck` ergänzt.

5.2 Java Quellcode-Generator

Dieser Abschnitt ist nach den Bestandteilen des Java-Codegenerators gegliedert. Es wird erläutert, wie die Bestandteile verändert und erweitert werden müssen, damit Anbindungen wie in Abschnitt 3.4 beschrieben generiert werden. Die Traits `InterfaceMaker`, `DependenciesMaker` und `api.VistorsMaker` werden nicht behandelt, da sie unverändert bleiben.

`GeneralOutputMaker` Als Erstes wird der Aufruf von `removeEnums` entfernt und ein neues Feld `enums` hinzugefügt. In `enums` werden die `enum`-Typen gespeichert. `removeEnums` wird entfernt, da es die Methode nach den im vorhergehenden Abschnitt beschrieben Anpassungen nicht mehr gibt.

Die Methoden `appendDefaultEnumArguments`, `appendEnumArguments`, `knownInstance` und das Feld `instanceNameStore` werden hinzugefügt. Die ersten Beiden werden von `TypesMaker` und `AccessMaker` genutzt. Ihre Implementierung ist in `Main` zu finden. Die letzten Beiden verwalten die Namen der Feldrepräsentanten von `enum`-Konstanten. Sie funktionieren wie `knownField` und `fieldNameStore`, die für die Namen normaler Feldrepräsentanten verantwortlich sind. Die Namen können nicht zusammen verwaltet werden, da Felder in der Zwischendarstellung vom Typ `FieldLike` und `enum`-Konstanten vom Typ `Name` sind. Da `enum`-Konstanten `Name` sind, wissen sie nicht, zu welchem `enum`-Typ sie gehören, sodass auch `enum`-Typ an `knownInstance` übergeben werden muss, um sie eindeutig zu identifizieren.

`Main` Für `enum`-Typen gibt es die Methoden `appendDefaultEnumArguments` und `appendEnumArguments`. Sie werden verwendet, wenn dem Konstruktor eines Nutzertyps `enum`-Proxys für seine `enum`-Felder übergeben werden sollen. `appendDefaultEnumArguments` erzeugt Code um die Defaultwerte zu erhalten und `appendEnumArguments` erzeugt Code um beliebige `enum`-Proxys zu erhalten. `appendEnumArguments` erzeugt außerdem die Argumente, wie sie für `enum`-Felder im Kopf eines Nutzertypkonstruktors stehen. Damit sich die Argumente der Konstruktoren nicht doppeln, müssen `enum`-Typen aus `appendConstructorArguments` entfernt werden.

Die Methode `mapType` wird um einen Fall für `enum`-Typen erweitert. Da für einen `enum`-Typ je nach Situation entweder der tatsächliche `enum`-Typ oder `EnumProxy` als Typ generiert werden muss, wird der Boolean-Parameter `boxed` verwendet, um zwischen den Beiden zu wechseln. In den meisten Fällen wird so der richtige Typ generiert. In den übrigen muss händisch zwischen `enum`-Typen und andere Typen unterschieden werden.

EnumsMaker Dieser Trait ist ein neuer Bestandteil des Quellcode-Generators. Er generiert die Enumerationen. Er erbt von `GeneralOutputMaker` und wird von `Main` implementiert.

FieldDeclarationMaker Dieser Trait bekommt eine neue Methoden `makeEnumFields` um Feldrepräsentanten für `enum`-Konstanten zu generieren. Außerdem muss in `makeFields` die `switch`-Anweisung zum Generieren der Überprüfung des Feldtyp um einen Fall für `enum`-Typen erweitert werden. Die Methoden `prelude`, `readCodeInner`, `offsetCode`, `offsetCodeInner` und `writeCode(t : Type, fieldAccess : String)` brauchen ebenfalls zusätzlichen Fälle für `enum`-Typen. Durch die Erweiterung von `mapType` in `Main` werden Feldrepräsentanten von `enum`-Feldern schon mit den passenden Feldtypen generiert. Nur in `writeCode` ist der Feldtyp für die Fälle `ConstantLengthArrayType` und `SingleBaseTypeContainer` nicht richtig. Im Falle eines `enum`-Typs darf `mapType` nicht aufgerufen werden. Stattdessen wird direkt `EnumProxy` generiert.

AccesMaker Dieser Trait bekommt eine neue Methode `makeEnumPools`, welche die in Abschnitt 3.4.3 beschriebenen Subklassen von `EnumPool` generiert. Die Methode `makePools` wird angepasst, damit Pools von Nutzertypen mit `enum`-Felder wie in Abschnitt 3.4.4 beschrieben generiert werden. Dazu werden die Methoden `appendDefaultEnumArguments` und `appendEnumArguments` verwendet.

TypesMaker Dieser Trait wird so erweitert, dass Nutzertypklassen mit `enum`-Feldern wie in Abschnitt 3.4.2 beschrieben generiert werden. Dazu werden `appendEnumArguments` und `mapType` verwendet.

internal.FileParserMaker Dieser Trait wird so erweitert, dass der Parser wie in Abschnitt 3.4.7 beschrieben generiert wird. Die Gesamtzahl aller spezifizierten Typen ist nicht mehr die Anzahl der Nutzertypen, sondern die Summe aus Nutzer- und `enum`-Typen.

internal.StateMaker und api.SkillFileMaker Diese Traits werden so erweitert, dass sie Code wie in Abschnitt 3.4.8 beschrieben generieren. Dazu muss an mehreren Stellen zusätzlich zu den Nutzertypen auch über die `enum`-Typen iteriert werden.

InternalMaker Dieser Trait wird um Aufrufe der Methoden `makeEnumPools` und `makeEnumFields` erweitert, damit die `enum`-Pools und die Feldrepräsentanten der `enum`-Konstanten generiert werden.

5.3 C++ Quellcode-Generator

In diesem Abschnitt werden die Veränderungen des C++-Quellcode-Generators beschrieben, die vorgenommen wurden, um Anbindungen wie in Abschnitt 4.4 beschrieben zu generieren. Die Veränderungen sind nach den Bestandteilen des Quellcode-Generators geordnet. `DependenciesMaker` wird nicht behandelt, da sich an diesem Bestandteil nichts ändert.

EnumsMaker Der Trait `EnumsMaker` erbt von `GeneralOutputMaker` und generiert die Enumerationen.

GeneralOutputMaker Dieser Trait wird um ein Feld enums zum Speichern der enum-Typen und um die Methoden knownInstance und name erweitert. knownInstance ist das Äquivalent zu knownField für enum-Konstanten, und aus denselben Gründen notwendig, wie im Java-Generator (siehe Abschnitt 5.2). name gibt den Namen einer enum-Konstanten in CamelCase zurück und wird von EnumsMaker verwendet. Außerdem muss allStrings zu einem Tripel erweitert werden. Bisher enthielt allStrings an erster Stelle die Namen der Typen und an zweiter Stelle die Namen der Felder, die keine Namen von Typen sind. Für enum-Typen braucht allStrings eine dritte Stelle, die die Namen der enum-Typen enthält. Die Namen der Nutzertypen und der enum-Typen sind disjunkt, daher müssen für diese Mengen keine zusätzlichen Vorkehrungen getroffen werden. Zu den Namen der Felder müssen die Namen der enum-Konstanten hinzugefügt und die Namen der enum-Typen abgezogen werden.

Main Damit die Enumerationen generiert werden, muss EnumsMaker zu den von Main implementierten Traits hinzugefügt werden. mapType wird um einen Fall für enum-Typen erweitert. Für enum-Typen wird ::skill::api::EnumProxy * als Typ verwendet. Für FieldDeclarationsMaker, makeConstructorArguments, appendConstructorArguments und Teile von TypesMaker ist das der passende Typ. Dort, wo nicht EnumProxy, sondern der tatsächliche enum-Typ benötigt wird, muss händisch zwischen enum-Typen und anderen Typen unterschieden werden.

FieldsDeclarationsMaker Um die Header- und Source-Dateien von Feldrepräsentanten von enum-Konstanten zu generieren, gibt es die neue Methoden makeEnumHeader und makeEnumSource. Sie generieren Feldrepräsentanten wie in Abschnitt 4.4.6 beschrieben. Der Code, der die Überprüfung des Feldtyps generiert, hat zuvor vollständig gefehlt. Die Überprüfung des Feldtyps wird nur für Feldrepräsentanten von enum-Konstanten und enum-Feldern mit Funktion generiert. Für alle anderen Repräsentanten wird if (false) throw ::skill::SkillException("..."); generiert. Außerdem wird setR im Falle eines enum-Feldes wie in Abschnitt 4.4.5 beschrieben generiert.

PoolsMaker Um die Header- und Source-Dateien von enum-Pools zu generieren wird PoolsMaker um die beiden Methoden makeEnumSource und makeEnumHeader erweitert. Sie generieren Subklassen von EnumPool wie in Abschnitt 4.4.3 beschrieben. Sie verwenden knownInstance um die Namen der Feldrepräsentanten zu erfahren. Die Methoden makeHeader und makeSource werden so erweitert, dass Pools von Nutzertypen mit enum-Feldern wie in Abschnitt 4.4.4 beschrieben generiert werden. Dazu braucht mapFieldDefinition einen zusätzlichen Fall für enum-Typen.

SkillFileMaker Dieser Trait wird so erweitert, dass er Code wie in Abschnitt 4.4.9 generiert. Dazu wird an mehreren Stellen zusätzlich zu den Nutzertypen auch über die enum-Typen iteriert und zusätzlich zu den Nutzertypen überprüft, ob enum-Typen existieren.

StringKeeperMaker In StringKeeperMaker muss neben der ersten und zweiten auch noch über die dritte Komponente von allStrings iteriert werden.

6 Container

In SKilL gibt es Containertypen (siehe [Fel17b] §3.5.1). Es gibt Arrays mit konstanter oder variabler Länge, Listen, Sets und mehrstellige Maps. Sie können auch `enum`-Konstanten enthalten.

Um herauszufinden, wie häufig dies vorkommt, wurden die von Raza et al. [RVP06] und Pfister [Pfi18] verwendeten Spezifikationen untersucht. In keiner davon gibt es `enum`-Typen in Containern. In Anbetracht dessen ist es vertretbar, dass die hier beschriebene Implementierung noch Mängel hat.

Aus den gleichen Gründen, aus denen `enum`-Typen nicht als Feldtypen von einzelnen `enum`-Feldern verwendet werden können, können sie auch nicht als Grundtyp von Containern verwendet werden. Wie auch in der bisherigen Implementierung von `enum`-Typen dieser Arbeit, wird `EnumProxy` als Grundtyp von Containern verwendet. In der Java Implementierung gibt es für eine bekannte Liste mit `enum`-Konstanten dann ein Feld vom Typ `java.util.LinkedList<EnumProxy>` und einen Feldrepräsentanten mit Feldtyp `ListType<EnumProxy>`. In der C++-Implemenetierung gibt es für eben jene Liste ein Feld vom Typ `::skill::api::Array<::skill::api::EnumProxy*>*` und einen Feldrepräsentanten mit Feldtyp `::skill::fieldTypes::ListType`. In beiden Implementierungen kennt der Feldtyp des Feldrepräsentanten den Feldtyp des Grundtyps. Für die obige Liste ist der Feldtyp des Grundtyps `EnumPool<EnumProxy>`.

Die Feldrepräsentanten der Containerfelder rufen die Felddatenserialiserungsmethoden des Feldtyps oder des Grundtyps auf. Die Felddaten rufen auch die Methoden des Grundtyps auf, sodass für `enum`-Typ-Container immer die Felddatenserialisierungsmethoden von `EnumPool` aufgerufen werden. Davon ausgenommen sind die Feldrepräsentanten von bekannten Listen, Arrays und Sets in Java-Anbindungen. Sie verarbeiten ihre Felddaten selbst. Mit `v.add(p.getProxy((int) in.v64()))` werden sie eingelesen und zum Container hinzugefügt, mit `for (EnumProxy x : v) out.v64(e.getSkillID())` in eine Austauschdatei geschrieben und mit `for (EnumProxy x : v) result += null==x?1:V64.singleV64Offset(x.getSkillID())` wird ihr Offset berechnet. `p` ist der `enum`-Pool und `v` der Container.

Um Container mit `enum`-Typen wie eben beschrieben zu realisieren, müssen am Quellcode-Generator der C++-Anbindung keine zusätzlichen Änderungen vorgenommen werden. Für Java muss `offsetCodeInner` in `FieldDeclarationMaker` um einen Fall für `enum`-Typen erweitert werden.

Container mit `enum`-Proxys zu befüllen hat Nachteile. Zum einen gewährt ein Container nur Zugriff auf seinen tatsächlichen Inhalt, sodass der Werkzeugbauer auch dann mit `enum`-Proxys arbeiten muss, wenn diese eigentlich zu bekannten `enum`-Konstanten gehören. Außerdem können zu einem `enum`-Typ-Container `enum`-Proxys von beliebigen `enum`-Typen oder gar die Werte `null` beziehungsweise `nullptr` hinzugefügt werden. Man könnte sowohl das erste als auch das zweite Problem lösen, indem man Subklassen der Containertypen macht und in diesen die Zugriffsmethoden so überschreibt, dass die Getter, falls vorhanden, `enum`-Konstanten zurückgeben und die Setter unpassende Werte zurückweisen.

7 Anpassungen an der Sprachspezifikation

Dieses Kapitel bezieht sich auf die in [Fel17b] beschrieben Sprachspezifikation von SKilL. Die im Folgenden verwendeten Kapitelnummern beziehen sich ebenfalls auf [Fel17b].

4.4.2 Enums Die Beschreibung der enum-Typen muss verändert werden. Sie haben keine Felder mehr und ihre Konstanten werden nicht mehr zu Subtypen mit Singleton-Restriktion.

4.4.4 Views Der Satz „Views should not be used in enum declarations" muss gelöscht werden, da enum-Typen keine Views mehr haben können.

4.5 Default Values In diesem Abschnitt definiert Felden [Fel17b] die erste Konstante eines enum-Typs als Defaultwert diese Typs. Da nur bekannte enum-Typen eine erste Konstante haben, muss diese Definition um den Defaultwert für unbekannte enum-Typen erweitert werden.

5.1.1.2 Singleton enum-Typen dürfen in diesem Abschnitt nicht mehr erwähnt werden, da sie nichts mehr mit der Singleton-Restriktion zu tun haben.

7.4 Serialization of Field Data / Appendix B Die Felddaten von enum-Feldern werden anders serialisiert als die von Feldern, deren Feldtyp ein Nutzertyp ist. Wenn E die Menge aller enum-Typen ist, dann müssen die Felddatenserialisierungsfunktionen um Gleichung (7.1) ergänzt werden.

$$\forall t \in E.[\![f]\!]_t = \begin{cases} [\![t.defaut.fieldID]\!]_{v64} & f = \texttt{NULL} \\ [\![t.fieldByName(f.name).fieldID]\!]_{v64} & else \end{cases} \tag{7.1}$$

Außerdem müssen die enum-Typen in der Serialisierungsfunktion von Nutzertypen ausgeschlossen werden: $\forall t \in U \backslash E \cup \{\textbf{string}\}.[\![f]\!]_t$

Appendix A In der Grammatik der Spezifikationssprache müssen Felder aus der Produktionsregel für enum-Typen entfernt werden. Die korrigierte Regel lautet:

```
enum-type ::= comment? enum ID '{' ID (',' ID)* ';' '}'
```

Appendix G Die Tabelle der Typ-Restriktionen muss um einen Eintrag für enum-Restriktionen ergänzt werden. Die ID ist 7 und die Serialisierung ε.

8 Tests

Dieses Kapitel befasst sich mit den Tests der neuen enum-Typ-Implementierung. Es werden sowohl die von SKilL automatisch generierten Tests, als auch manuell geschriebene Tests verwendet. Im ersten Abschnitt werden die verwendeten Spezifikationen und Austauschdateien vorgestellt. In den weiteren Abschnitten folgen Erläuterungen zu den einzelnen Tests. Der letzte Abschnitt befasst sich mit der Testabdeckung der Java-Implementierung.

Die hier getestete Version von SKilL ist `546260c7451495f236737ffb231256c48fb02aa9`, die der Bibliothek `java.common` ist `04db4ec20a18333a900b1b11c0432ff976577082` und die der Bibliothek `cpp.common` ist `7384ab379bb54eb8fbbb487293196c7f2f1ba8aa`. Die Versionen der Testsuite `javaTest` ist `e4973aa4d229100af4d851d1e834286376e0c1a3` und die der Testsuite `cppTest` ist `562be58c7aaf614f26bf6fb6322efd3533b1cbc7`.

8.1 Verwendete Spezifikationen und Austauschdateien

Die neu geschriebenen Testfälle verwenden die Spezifikationen aus Listing 8.1 bis 8.6. Um Platz zu sparen fehlt in jeder Spezifikation die erste Zeile. Ihr Inhalt ist nur für den Test-Generator relevant.

Die Spezifikationen `completeEnum.skill` und `incompleteEnum.skill` aus Listing 8.1 und 8.2 sind so entworfen, dass die Typen aus `incompleteEnum.skill` eine Teilmenge der Typen aus `completeEnum.skill` sind. Damit sind diese Spezifikationen geeignet, um zu überprüfen ob hinzugefügte oder gelöschte enum-Felder, -Typen und -Konstanten richtig gehandhabt werden. Die Spezifikationen `enumTypeFarbe.skill`, `userTypeFarbe.skill` und `subTypeFarbe.skill` aus Listing 8.3, 8.4 und 8.5 enthalten alle einen Typ `Farbe`. In `enumTypeFarbe.skill` ist er ein enum-Typ, und in `userTypeFarbe.skill` und `subTypeFarbe.skill` ein Nutzertyp. In `subTypeFarbe.skill` erbt er von einem anderen Nutzertyp, in `userTypeFarbe.skill` nicht. `containedEnums.skill` aus Listing 8.6 enthält für jeden der fünf in SKilL vorhandenen Containertypen ein Feld und für Maps sogar zwei, eines für eine zweistellige Map und eines für eine dreistellige Map. Die Felder sind so spezifiziert, dass alle Container enum-Typen enthalten.

Listing 8.1 completeEnum.skill

```
1 enum Farbe { rot, blau, gelb, weiss; }
2 enum Brot { weiss, roggen; }
3 Typ { Farbe farb; Brot brot; }
4 Ding { Brot brot; }
```

Listing 8.2 incompleteEnum.skill

```
1 enum Farbe { blau, rot; }
2 Typ { Farbe farb; }
```

Listing 8.3 enumTypeFarbe.skill

```
1 enum Farbe { rot; }
```

Listing 8.4 userTypeFarbe.skill

```
1 Farbe { i8 rot; }
```

Listing 8.5 subTypeFarbe.skill

```
1 GeneralFarbe { i8 rot; }
2 Farbe extends GeneralFarbe { i8 gelb; }
```

Listing 8.6 containedEnums.skill

```
1 enum Farbe { rot; }
2 claTyp { Farbe[1] claFarb; }
3 arrayTyp{ Farbe[] arrayFarb; }
4 listTyp{ list<Farbe> listFarb; }
5 setTyp { Set<Farbe> setFarb; }
6 map2Typ { map<Farbe,Farbe> map2Farb; }
7 map3Typ { map<Farbe,Farbe,Farbe> map3Farb; }
```

Spezifikation	Austauschdatei	Mod	Pfad
completeEnum.skill	completeEnum.sf		`[[empty]]/accept/`
completeEnum.skill	incompleteEnum.sf		`[[empty]]/accept/`
incompleteEnum.skill	incorrectIdEnumField.sf	×	`enums/fail`
incompleteEnum.skill	incorrectIdEnumConst.sf	×	`enums/fail`
enumTypeFarbe.skill	enumtypeFarbe.sf		`enumrestrictionEnumtype/accept`
userTypeFarbe.skill	usertypeFarbe.sf		`enumrestrictionUsertype/accept`
subTypeFarbe.skill	subtypeFarbe.sf		`enumrestrictionSubtype/accept`
subTypeFarbe.skill	subtypeEnumrest.sf	×	`[[all]]/fail`
–	supertypeEnumrest.sf	×	`[[all]]/fail`

Tabelle 8.1: Zusammengehörigkeit von Spezifikationen und Austauschdateien. Der Pfad ist relativ zu `skill/test/resources/genbinary` angegeben. Die in der Spalte **Mod** markierten Austauschdateien sind modifiziert.

Außerdem verwenden die Tests mehrere Austauschdateien, die mit Anbindungen der beschriebenen Spezifikationen geschrieben wurden. Tabelle 8.1 zeigt welche Austauschdatei zu welcher Spezifikation gehört, ob sie modifiziert wurde und auf welchem Pfad sie liegt. Die Spezifikation, zu der die letzte Austauschdatei gehört ist nicht mit angegeben, da sie in den Tests nicht weiterverwendet wird.

Die Austauschdateien `supertypeEnumrest.sf`, `subtypeEnumrest.sf`, `incorrectIdEnumField.sf` und `incorrectIdEnumConst.sf`, wurden von Hand verändert. In den Dateien `incorrectIdEnumField.sf` und `incorrectIdEnumConst.sf` wurde die Feld-ID von `farb` beziehungsweise `blau` verfälscht. Die Dateien `supertypeEnumrest.sf` und `subtypeEnumrest.sf` enthalten beide zwei Nutzertypen, wobei der eine ein Subtyp des anderen ist. In `supertypeEnumrest.sf` wurde der Supertyp zu einem `enum`-Typ gemacht und in `subtypeEnumrest.sf` wurde der Subtyp zu einem `enum`-Typ gemacht. Damit sind diese beide Dateien so fehlerhaft, dass sie von keiner Anbindung akzeptiert werden dürfen.

8.2 Generierte Tests

Diese Tests werden für alle Spezifikationen, die im Ordner `gentest` liegen, generiert. Es werden Tests für den Lesevorgang und Tests für das Application Programming Interface (API) generiert.

8.2.1 API Tests

Die generierten API-Tests können für `enum`-Typen nur bedingt eingesetzt werden, da der Test-Generator nicht mit `enum`-Konstanten in `.json`-Dateien umgehen kann. Der Generator müsste dazu so angepasst werden, dass er den Zugriff auf `enum`-Konstanten zur Zielsprache passend generiert, also `Farbe.ROT` für Java und `::Farbe::rot` für C++. Es können für Spezifikationen mit `enum`-Typen im Moment nur API-Tests generiert werden, bei denen die Werte von `enum`-Feldern nicht neu gesetzt werden.

8.2.2 Lesetests

Die für Java generierten Lesetests enthalten die Testfälle `writeGeneric` und `writeGenericCheck`, in denen unter anderem für jeden Typ der Anbindung eine Instanz erstellt wird. Da es nicht erlaubt ist Instanzen von `enum`-Typen zu erstellen und die Tests nicht überprüfen, ob ein Typ ein `enum`-Typ ist, schlagen sie für alle Spezifikationen, die `enum`-Typen enthalten, fehl. Diese Verhalten ist korrekt. Die C++-Tests enthalten keine derartigen Testfälle. Es darf keine Fehlschläge geben.

8.3 Manuell geschriebene Tests

Die folgenden Tests existieren, um sicherzustellen, dass die Implementierungen der `enum`-Typen die an sie gestellten Anforderungen erfüllen.

No.	Spezifikation	Austauschdatei	Fehler
1	enumTypeFarbe.skill	enumtypeFarbe.sf	–
2	userTypeFarbe.skill	usertypeFarbe.sf	–
3	subTypeFarbe.skill	subtypeFarbe	–
4	enumTypeFarbe.skill	usertypeFarbe.sf	`SkillException`
5	userTypeFarbe.skill	enumtypeFarbe.sf	`SkillException`
6	subTypeFarbe.skill	subtypeEnumrest.sf	`SkillException`
7	enumTypeFarbe.skill	supertypeEnumrest.sf	`SkillException`

Tabelle 8.2: Verhalten der Tests der `enum`-Restriktionen

Die `enum`-Typen müssen änderungstolerant sein. Die verschiedenen Aspekte der Änderungstoleranz wurden in Abschnitt 1.3 wiedergegeben. Für Veränderung am Feldtyp ist `FieldTypeTest` (Abschnitt 8.3.7) zuständig und für Veränderungen einer Supertypbeziehung ist `EnumRestrictionTest` (Abschnitt 8.3.1) zuständig. `EnumRestrictionTest` stellt außerdem sicher, dass illegale `enum`-Restriktionen erkannt werden. Für das Hinzufügen und Entfernen von Typen und Feldern gibt es die Tests `ReadTest`, `WriteTest` und `AppendTest` (Abschnitt 8.3.2, 8.3.3 und 8.3.4).

Aus der Aufgabenstellung geht hervor, dass `enum`-Typen weder Subtypen noch Instanzen haben dürfen. Diese Eigenschaften werden von `EnumPoolTest` überprüft.

Aus dem in Kapitel 2 beschriebenen Entwurf ergibt sich die Existenz von `EnumProxy` und Anforderungen an die in den Nutzertypen und Feldrepräsentanten implementierten Zugriffsmethoden. Es gibt `ProxyTest` (Abschnitt 8.3.6), um sicherzustellen, dass sich `enum`-Proxys wie gewünscht verhalten und es gibt `AccessTest`, um sicherzustellen, dass sich die Zugriffsmethoden wie gewünscht verhalten.

Da es möglich ist, `enum`-Konstanten in Container zu stecken, gibt es `ContainedEnumsTest` (Abschnitt 8.3.9), um das Verhalten von Containern mit `enum`-Typen zu überprüfen.

8.3.1 EnumRestriction Tests

In Tabelle 8.2 ist dargestellt, welche Austauschdateien von den Anbindungen welcher Spezifikationen eingelesen werden, und welche Fehler erwartet werden.

Die ersten fünf Tests prüfen, dass kompatible und inkompatible Typen richtig erkannt werden. Im vierten und fünften Test wird ein Fehler erwartet, da die Anbindung und die Austauschdatei beide einen Typ `Farbe` enthalten, diese aber nicht miteinander kompatibel sind. Mit den letzten beiden Tests wird geprüft, ob illegale `enum`-Restriktionen an bekannten Typen erkannt werden. Der sechste Test überprüft dies für bekannte Subtypen und der siebte Test für bekannte Supertypen. Dass illegale `enum`-Restriktionen an unbekannten Typen erkannt werden, wird durch die generierten Lesetests geprüft, da die fehlerhaften Austauschdateien in `[[all]]/fail` liegen und die bereits vorhandenen Tests ausreichend Spezifikationen ohne `enum`-Typen enthalten.

8.3.2 Read Tests

Beim Lesen werden drei Fälle betrachtet. Der erste Fall ist, dass die Anbindung genau die in der Austauschdatei enthaltenen Typen und Felder kennt. Der zweite Fall ist, dass sie mehr Typen und Felder kennt als die Austauschdatei und der dritte, dass sie weniger kennt. Um zu testen, ob der Zustand in allen drei Fälle richtig erzeugt wird, werden die Spezifikationen `completeEnum.skill` und `incompleteEnum.skill` und die Austauschdateien `completeEnum.sf` und `incompleteEnum.sf` verwendet. Für den ersten Fall wird `completeEnum.sf` von der zu `completeEnum.skill` generierten Anbindung eingelesen. Für den zweiten Fall wird `incompleteEnum.sf` von der zu `completeEnum.skill` generierten Anbindung eingelesen. Für den dritten Fall wird `completeEnum.sf` von der zu `incompleteEnum.skill` generierten Anbindung eingelesen. Für jeden Fall wird nach der Zustandserstellung überprüft, ob die Anzahl der Typen, Felder und Typinstanzen stimmt und ob für jedes Feld der richtige Wert eingelesen wurde. Zur Überprüfung der Felddaten wird mit den verschiedenen `get`-Methoden auf die Felddaten zugegriffen. Wenn `ReadTest` erfolgreich ist, dann funktionieren auch die Getter.

8.3.3 Write Tests

Mit diesen Tests wird überprüft, ob bekannte und unbekannte `enum`-Typen, `enum`-Konstanten und `enum`-Felder fehlerfrei serialisiert werden. Für bekannte `enum`-Felder, -Konstanten und -Typen wird mit der zu `completeEnum.skill` generierten Anbindung eine temporäre Datei geschrieben und wieder eingelesen. Für unbekannte `enum`-Felder, -Konstanten und -Typen wird mit der zu `completeEnum.skill` generierten Anbindung eine temporäre Datei geschrieben, die mit der Anbindung zu `incompleteEnum.skill` eingelesen und nochmal geschrieben und dann von der ersten Anbindung wieder gelesen wird. Dann wird in beiden Fällen überprüft, ob der erste und letzte Zustand jeweils dieselben Typen, Felder und `enum`-Konstanten haben. Wenn dem nicht so ist, schlagen die Tests fehl. `WriteTest` kann so nur implementiert werden, weil mit `ReadTest` sichergestellt wird, dass das Lesen fehlerfrei ist.

8.3.4 Append Tests

Da der `append`-Modus bisher nur für Java implementiert ist, gibt es `AppendTest` nur in Java. Mit diesem Test wird überprüft, ob bekannte `enum`-Typen, -Konstanten und -Felder und neue Instanzen von Nutzertypen mit unbekannten `enum`-Felder korrekt an eine Austauschdatei angehängt werden. Für den ersten Teil wird mit der zu `incompleteEnum.skill` generierten Anbindung eine temporäre Datei geschrieben, von der zu `completeEnum.skill` generierten Anbindung gelesen, um Instanzen erweitert und im `append`-Modus erneut geschrieben. Für den zweiten Teil wird mit der zu `completeEnum.skill` generierten Anbindung eine temporäre Datei geschrieben. Die Datei wird von der zu `incompleteEnum.skill` generierten Anbindung gelesen, um Instanzen erweitert und im `append`-Modus erneut geschrieben. Dann werden die zuletzt geschriebenen Dateien in beiden Fällen mit der Anbindung zu `completeEnum.skill` nochmal eingelesen und es wird überprüft, ob alle hinzugefügten Instanzen vorhanden sind und ihre Felder die richtigen Werte haben.

8.3.5 Access Tests

Für diese Tests wird die Austauschdatei `completeEnum.sf` von der zu `incompleteEnum.skill` generierten Anbindung eingelesen. Der so erstellte Zustand enthält bekannte und unbekannte `enum`-Typen und -Konstanten und bekannte und unbekannte Nutzertypen mit bekannten und unbekannten `enum`-Feldern. `AccessTest` testet weder die in den Nutzertypen noch die in den Feldrepräsentanten implementierten `get`-Methoden, da diese implizit in `ReadTest` getestet werden.

Wenn sowohl der Nutzertyp als auch das `enum`-Feld bekannt sind, kann das Feld über die `set`-Methoden des Nutzertyps neu gesetzt werden. Es wird überprüft, dass die `set`-Methode keine illegalen Parameterwerte akzeptiert und bei einem legalen Wert das Feld richtig setzt. Für Java ist `null` der einzig mögliche illegale Wert. In C++ können Integerwerte mit `static_cast` in `enum`-Konstanten konvertiert werden. Alle so erzeugten `enum`-Konstanten, die aus Integerwerten konvertiert wurden, die nicht im Intervall des `enum`-Typs liegen, sind illegale Werte.

Unabhängig davon, ob ein Nutzertyp und seine `enum`-Felder bekannt oder unbekannt sind, können die Felddaten mit den Settern des Feldrepräsentanten verändert werden. Für bekannte `enum`-Felder muss überprüft werden, ob der Setter nur legale Werte entgegen nimmt und dass er sie richtig setzt. Für Java sind `null` und `enum`-Proxys, die zu einem falschen `enum`-Typ gehören, illegale Werte. In C++ ist der Parameter vom Typ `skill::api::Box`. Damit sind alle Boxen, die keine `enum`-Proxys enthalten und alle Boxen, die `enum`-Proxys enthalten, die zu einem falschen `enum`-Typ gehören, illegale Werte. Für unbekannten `enum`-Felder wird nur überprüft, ob der Setter das Feld richtig setzt, da die Setter der verteilten Felder nicht überprüfen, ob ein übergebener Wert legal ist. `AccessTest` ruft die Setter also mit legalen und illegalen Werten auf und stellt sicher, dass sie sich wie erwartet verhalten.

Die Zugriffsmethoden der Feldrepräsentanten gibt es auch für `enum`-Konstanten. Beim Aufruf dieser Methoden erwarten die Tests, dass ein Fehler geworfen wird.

8.3.6 EnumProxy Tests

Mit `enum`-Proxys befassen sich die Klassen `EnumProxyIterator` und `EnumPool`. Für `EnumProxyIterator` wird geprüft, dass die vom Iterator zurückgegebenen `enum`-Proxys denen des richtigen Pools entsprechen. Für `EnumPool` wird das Verhalten von `getProxy`, `getDefault` und `hasProxy` geprüft. Die Felddatenserialisierungsmethoden in `EnumPool` befassen sich auch mit `enum`-Proxys, werden aber schon von `ReadTest` und `WriteTest` abgedeckt. Außerdem werden die `enum`-Proxys selbst getestet.

Um `enum`-Proxys und `enum`-Pools sowohl von bekannten als auch von unbekannten `enum`-Typen und `enum`-Konstanten testen zu können, werden für `ProxyTest` die zu `incompleteEnum.skill` generierte Anbindung und die Austauschdatei `compeletEnum.sf` verwendet.

Für jeden `enum`-Proxy wird überprüft, ob, wenn seine SKilL-ID mit der Feld-ID eines Feldrepräsentanten übereinstimmt, auch sein SKilL-Name mit dem Namen dieses Feldrepräsentanten übereinstimmt. Ferner wird überprüft, ob der dem Proxy als `owner` bekannte `enum`-Pool auch den Proxy kennt. Für die Java-Implementierung wird noch überprüft, ob die SKilL-Namen der `enum`-Proxys mit den Namen der generierten `enum`-Konstanten übereinstimmen.

Für jeden `enum`-Pool wird geprüft, dass er genauso viele `enum`-Proxys wie Feldrepräsentanten hat. Falls alle Konstanten bekannt sind, wird auch noch überprüft, ob die Anzahl der `enum`-Proxys gleich der Anzahl der `enum`-Konstanten ist. Außerdem wird überprüft, dass `getProxy` und `getDefault` die

richtigen enum-Proxys zurückgeben. In der Java-Implementierung gibt es neben getProxy(int id) noch die Methode getProxy(String Name), für die auch geprüft wird, dass sie den richtigen Proxy zurückgibt. Alle getProxy-Methoden werden mit passenden Werten und mit Werten, zu denen kein Proxy existiert, aufgerufen. In letzteren Fällen wird ein Fehler erwartet.

hasProxy wird mit Proxys aufgerufen, die zum Pool gehören und solchen, die es nicht tun. Letztere haben dieselbe SKilL-ID und denselben SKilL-Namen, wie einer der Proxys aus dem richtigen Pool, um sicherzustellen, dass nicht einfach nur der Name oder die ID überprüft wird. Für die Java-Implementierung von EnumPool wird noch die Methode isKnownEnum getestet. Dabei werden dieselben enum-Proxys verwendet wie bei hasProxy. Für die C++-Implementierung wird noch überprüft, dass die Abbildung zwischen den SKilL-IDs der enum-Proxys und den Werten der bekannten enum-Konstanten korrekt ist.

8.3.7 FieldType Tests

Für diese Tests werden incorrectIdEnumConst.sf und incorrectIdEnumField.sf mit der zu incompleteEnum.skill generierten Anbindung eingelesen und jeweils eine SkillException erwartet. Wenn keine SkillException geworfen wird, schlagen die Tests fehl.

Dabei wurde, wie in Abschnitt 3.4.5 erwähnt, festgestellt, dass die Überprüfung des Feldtyps in der Java-Implementierung fehlerhaft ist.

8.3.8 EnumPool Tests

Die Java Variante von EnumPoolTest ruft die Methoden add, getByID, make und makeSubPool auf und erwartet den Fehler NoSuchMethodError. Die C++ Variante von EnumPoolTest ruft die Funktionen get, getAsAnnotation und makeSubPool auf und erwartet den Fehler std::bad_function_call. Wenn die erwarteten Fehler nicht geworfen werden, schlagen die Tests fehl.

8.3.9 ContainedEnums Tests

ContainedEnumsTests überprüft, ob enum-Konstanten richtig gelesen und geschrieben werden, wenn sie in einem Container enthalten sind. Der Test deckt bekannte und unbekannte Container-Felder ab. Mit der zu containedEnum.skill generierten Anbindung wird ein Zustand erstellt, alle Container instanziiert und zu jedem ein Element hinzugefügt. Dieser Zustand wird in eine temporäre Datei geschrieben. Um bekannte Container-Felder zu testen, wird die temporäre Datei mit derselben Anbindung wieder eingelesen. Um unbekannte Container-Felder zu testen, wird sie mit der zu incompleteEnum.skill generierten Anbindung eingelesen. In beiden Fällen wird nach dem Lesen geprüft, ob alle Container die vor dem Schreiben gesetzten Werte enthalten.

8.4 Codecoverage der Java Tests

Die Codecoverage der Java Tests wurde mit EclEmma (siehe [Mar]) getestet. Die Klassen `EnumProxy`, `EnumProxyIterator` und `EnumRestriction` und der Code zur Serialisierung von `enum`-Restriktionen in `FileParser` und `SerializationFunctions` werden vollständig abgedeckt. Die Klasse `EnumPool` wird bis auf die Methoden `delete` und `singleOffset` und Teile der Methode `calculateOffset` vollständig abgedeckt. `delete` kann nicht getestet werden, da sie nicht `public` ist und daher in den Tests nicht aufgerufen werden kann. Dass die Methoden `singleOffset` und `calculateOffset` nicht vollständig abgedeckt werden, ist vertretbar, da sie, bis auf die Wahl der zu serialisierenden ID, identisch zu den von ihnen überschriebenen Methoden in `StoragePool` implementiert sind. Wenn sie in `StoragePool` funktionieren, sollten sie also auch in `EnumPool` funktionieren.

Die zu `completeEnum.skill` und `incompleteEnum.skill` generierten Anbindungen decken zusammen große Teile der generierten Pools und Feldrepräsentanten ab. Nicht abgedeckt werden die innere Klasse `UnknownSubPool`, die Methode `makeSubPool` und Teile der inneren Erbauerklasse der für Nutzertypen generierten Pools.

Alle anderen Instruktionen werden in mindestens einer der generierten Klassen dieser beiden Anbindungen mindestens einmal ausgeführt. Diese beiden Anbindungen decken fernen die relevanten Teile der Nutzertypklassen, also die Zugriffsmethoden und Konstruktoren ab.

Die Methoden `newPool` und `newEnumPool` der Klasse `Parser` werden vollständig abgedeckt. `newPool` wird durch die Anbindung zu `enumTypeFarbe.skill` abgedeckt. Um alle Verzweigungen in `newEnumPool` abzudecken bedarf es mehrerer Anbindungen, zum Beispiel der Anbindung zu `userTypeFarbe.skill` oder `subTypeFarbe.skill` und einer beliebige Anbindung, die keinen Typ `Farbe` kennt.

9 Zusammenfassung

In dieser Arbeit wurde gezeigt, dass es möglich ist enum-Typen in SKilL so zu implementieren, dass sie auf Enumerationen der Zielsprache abgebildet werden und trotzdem änderungstolerant sind. Der Kern dieses Erfolgs ist die Klasse EnumProxy. Indem sie als Feldtyp von enum-Feldern und als Typ von enum-Pools eingesetzt wird, ermöglicht sie es, dass enum-Typen änderungstolerant sind.

Neben der Lösung des Problems der nicht änderungstoleranten Zielsprachenenumerationen, wurde auch ein Weg gefunden, um enum-Typen und -Konstanten zur Laufzeit zu repräsentieren und mit möglichst geringem Mehraufwand zu serialisieren. Dazu bedarf es der Pool-Klasse EnumPool und der bereits vorhandenen Feldrepräsentanten. Instanzen der Klasse EnumPool repräsentieren die enum-Typen und die Feldrepräsentanten repräsentieren die enum-Konstanten.

Die entworfene Lösung wurde in Java und C++ implementiert. Die notwendigen Klassen wurden hinzugefügt und die SKilL-Bibliotheken und Codegeneratoren für Java und C++ wurden angepasst. Die fertigen Implementierungen wurden getestet und haben die beschriebenen Tests bestanden.

Ausblick

Die Implementierung der enum-Typen selbst lässt sich insbesondere bezüglich der Klasse EnumProxy verbessern. In Abschnitt 2.2.3 und Kapitel 6 ist man damit konfrontiert, dass anhand des Feldtyps eines enum-Feldes nicht bestimmt werden kann, von welchem enum-Typ das Feld ist. Dieses Problem lässt sich beseitigen, indem für jeden bekannten enum-Typ eine eigen Subklasse von EnumProxy generiert wird. Leider ist diese Idee erst gegen Ende der Arbeit aufgekommen, sonst wäre sie vermutlich schon umgesetzt.

Außerdem sollten der Generator der Tests und die noch unveränderten Quellcode-Generatoren, also alle außer die Quellcode-Generatoren von C++ und Java, angepasst werden. Der Generator der API-Tests sollte so erweitert werden, dass er den Zugriff auf enum-Konstanten zur Zielsprache passend generiert. Die Methode reflectiveInit der Klasse CommonTest der Jave-Testsuite sollte so geändert werden, dass sie keine Instanzen von enum-Typen mehr erstellen will. Die noch unveränderten Quellcode-Generatoren sollten so angepasst werden, dass sie mit enum-Typen umgehen können. Im Moment können sie dies nicht, da enum-Typen in der Zwischendarstellung der Quellcode-Generatoren bisher durch Nutzertypen ersetzt wurden. Die betroffenen Quellcode-Generatoren werfen einen Fehler, wenn sie Code zu einer Spezifikation generieren sollen, die enum-Typen enthält.

Einige der in Kapitel 8 beschriebenen Austauschdateien liegen in den Ordnern genbinary/[[empty]] und genbinary/[[all]]. Auf diese Ordner greifen die Tests aller Sprachen zu. Einige Sprachen, können mit Austauschdateien, die enum-Typen enthalten, nicht umgehen. Die generierten API- und Lesetest dieser Sprachen werfen Fehler. Um die Fehler zu vermeiden, sollte die Implementierung der enum-Typen der betroffenen Sprachen angepasst werden.

Glossar

enum-Feld Ein Feld in einem Nutzertyp, dessen Feldtyp ein enum-Typ ist. 5, 9–18, 23–26, 33, 34, 36, 37, 41–45, 47, 49, 51, 53, 54, 57

enum-Konstante Eine in einem enum-Typ deklarierte Konstante einer Spezifikation oder Anbindung. Kann auch unbekannt sein. 5, 9–17, 20, 21, 23, 24, 26, 30, 31, 33, 35–38, 42–45, 51–55, 57

enum-Typ Eine Enumeration in einer Spezifikation oder Anbindung. Kann auch unbekannt sein. 3, 5, 7, 9–20, 22–27, 29–35, 37–39, 41–45, 47, 49, 51–54, 57, 58

Anbindung Besteht aus einer SKilL-Bibliothek und dem vom Quellcode-Generator generierten Code. 7, 8, 10–12, 17, 18, 23, 26, 36, 42, 43, 45, 51–56

Austauschdatei Eine Datei, die mittels SKilL serialisierte Daten enthält. 6–8, 10–15, 17, 18, 21, 22, 25, 26, 33, 35–37, 39, 45, 49–54, 57

Nutzertyp Ein Typ, der keiner der von SKilL vordefinierten Typen und auch kein enum-Typ ist. Also alle nicht-enum-Typen, die der Werkzeugbauer spezifiziert hat. 5, 9–11, 14, 15, 17, 23–26, 31–39, 41–44, 49, 51–54

Quellcode-Generator Generiert Code für den spezifikationsspezifischen Teil einer Anbindung. 6–8, 35, 42, 43, 45, 57

Spezifikation Beschreibung einer Zwischendarstellung in der SKilL-Spezifikationssprache. 6–12, 14, 16, 17, 23, 26, 33, 35, 49–53, 57

Akronyme

API Application Programming Interface. 51

SKiIL Serialization Killer Language. 7–9, 12–15, 17, 18, 20–22, 25, 29, 31–33, 35, 36, 45, 47, 49, 54, 55, 57, 58

Literaturverzeichnis

[BRJ05] G. Booch, J. Rumbaugh, I. Jacobson. *The unified modeling language user guide*. Englisch. 2. ed. The Addison-Wesley object technology series. Upper Saddle River, NJ: Addison-Wesley, 2005, XVIII, 475 Seiten. ISBN: 0-321-26797-4 (zitiert auf S. 19, 29).

[Fel17a] T. Felden. „Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen“. Dissertation. Universität Stuttgart, 2017. URL: http://dx.doi.org/10.18419/opus-9661 (zitiert auf S. 7, 8, 10, 22, 32).

[Fel17b] T. Felden. *The SKilL Language V1.0*. Technischer Bericht Informatik 2017/01. Universität Stuttgart, 2017 (zitiert auf S. 17, 33, 45, 47).

[GHJV15] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. Deutsch. Hrsg. von M. Feilen, K. Lorenzen. 1. Aufl. Frechen: mitp, 2015, 472 Seiten. ISBN: 978-3-8266-9700-5 (zitiert auf S. 12, 14, 16, 19, 29).

[GJS+14] J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014 (zitiert auf S. 7, 23).

[ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372 (zitiert auf S. 7, 29, 33).

[KND+97] P. King, P. Naughton, M. DeMoney, J. Kanerva, K. Walrath, S. Hommel. *Java Code Conventions*. California: Sun Microsystems, Inc., 1997. URL: http://www.oracle.com/technetwork/java/codeconventions-150003.pdf (zitiert auf S. 23).

[Mar] E. M. Marc R. Hoffmann Brock Janiczak. *EclEmma 3.1.0 Java Code Coverage for Eclipse*. Accessed July 19, 2018. URL: https://www.eclemma.org/ (zitiert auf S. 56).

[Ora] Oracle. *Java^{TM} Platform, Standard Edition 8 API Specification*. Accessed July 19, 2018. URL: https://docs.oracle.com/javase/8/docs/api/index.html (zitiert auf S. 15, 19).

[Pfi18] D. Pfister. „Skilled LLVM“. Masterarbeit. Universität Stuttgart, 2018 (zitiert auf S. 45).

[RVP06] A. Raza, G. Vogel, E. Plödereder. „Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering“. In: *Reliable Software Technologies – Ada-Europe 2006*. Bd. 4006. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 71–82 (zitiert auf S. 45).

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