Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3603

Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache

Wladislaw Ungur

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden

Beginn am:	2. Januar 2014
Beendet am:	4. Juli 2014
CR-Nummer:	D.2, D.3.3, E.2, E.5

Kurzfassung

In dieser Diplomarbeit wird die Nutzbarkeit und die praktische Tauglichkeit einer Schnittstelle und des passenden Codegenerators für die Serialisierungssprache SKilL untersucht. Der Schwerpunkt liegt bei der Entwicklung einer API und eines Codegenerators in der Programmiersprache Java, wobei laut Anforderung die Schnittstelle sich möglichst gut und natürlich in die Sprache integrieren muss. Besondere Beachtung finden auch die Benutzerfreundlichkeit und die Einfachheit, nicht nur die Schnittstelle sondern auch der Gnerator sollen unkompliziert und intuitiv zu bedienen sein. Der Entwickler, der sich für Skill entschieden hat, muss sich nicht gründlich mit dem Serialisierungsformat beschäftigen, um es nutzen zu können. Die Diplomarbeit zeigt unter anderem, wie die Schnittstelle und der Codegenerator in Java implementiert wurden, und welche Entscheidungen vor allem im Bezug auf die Nutzbarkeit hinter der Implementierung stehen.

Inhaltsverzeichnis

1 Einführung **7**
1.1 Gliederung . . . 7
1.2 Aufgabenstellung . . . 7
1.3 SKilL . . . 9
1.4 Schnittstelle und Codegenerator . . . 13
1.5 Ähnliche Arbeiten . . . 14
1.6 Abgrenzung bei Begriffen . . . 14

2 Schnittstelle **15**
2.1 Grundlegende Anforderungen an die Schnittstelle . . . 15
2.2 Analyse der Anforderungen . . . 17
2.3 SKilL-Schnittstelle für Java . . . 21
2.4 Nutzungsbeispiele . . . 32

3 Nutzbarkeitsevaluation der Schnittstelle **35**
3.1 Bedeutung einer guten Schnittstelle . . . 35
3.2 Eigenschaften einer guten Schnittstelle . . . 36
3.3 Erlernbarkeit . . . 36
3.4 Einfachheit . . . 39
3.5 Falsche Verwendung soll nicht möglich sein . . . 47
3.6 Gewährleistung eines lesbaren Nutzer-Codes . . . 47
3.7 Erweiterbarkeit . . . 48
3.8 Weitere Merkmale einer guten Schnittstelle . . . 49

4 Codegenerator **51**
4.1 Anforderungen an den Codegenerator . . . 51
4.2 Architektur und Implementierung des Codegenerators . . . 53
4.3 Implementierung des GUI-Prototypen . . . 56
4.4 Nutzbarkeitevaluation des Codegenerators . . . 57

5 Zusammenfassung und Ausblick **63**

Literaturverzeichnis **65**

1 Einführung

Dieses Kapitel stellt die Aufgabe dieser Diplomarbeit vor. Es wird außerdem die Serialisierungssprache und das passende Format in Ansätzen erklärt.

1.1 Gliederung

In der Einleitung befinden sich, neben der Aufgabe und der Serialisierungssprache die Gliederung der Diplomarbeit, ähnliche Arbeiten und Begriffsabgrenzungen.

Im Kapitel 2 wird die Schnittstelle vorgestellt und es werden zwei Beispiele für die Nutzung gezeigt. Im Kapitel 3 wird die Schnittstelle im Bezug auf die Nutzbarkeit evaluiert.

Im Kapitel 4 befinden sich die Beschreibungen und Erläuterungen zu dem Codegenerator sowie zu dem grafischen Generator. Beide werden, ähnlich zu der Schnittstelle, untersucht, um zu prüfen, in wie weit sie benutzerfreundlich und einfach zu bedienen sind.

Das Kapitel 5 gibt abschließend eine Zusammenfassung der Arbeit und einen kurzen Ausblick.

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit sollen mindestens drei Punkte bearbeitet werden.

- Implementierung einer SKilL-Anbindung für die Programmiersprache Java, die die SKilL-Kernsprache vollständig abdeckt. Zudem soll noch ein Codegenerator entwickelt werden, der die Schnittstelle bzw. Teile davon erzeugen kann.
- Es muss analysiert werden, wie die generierte Schnittstelle aussehen muss, um sich möglichst gut in die Programmiersprache Java zu integrieren.
- Es sollen Möglichkeiten evaluiert werden, den Entwurf von Datenformaten und die Erzeugung der dazugehörigen Sprachanbindungen benutzerfreundlicher zu gestalten.

Optionale Punkte in der Aufgabenstellung...

- Eine Sprachanbindung und ein Codegenerator für eine weitere Programmiersprache, z.B. C++.
- Vorhersagen über die Robustheit des Serialiserungsformats gegenüber üblichen Änderungen machen.
- Vorschläge für weitere Sprachkonstrukte.

Ein besonderer Schwerpunkt bei der Umsetzung der Schnittstelle und des Codegenerators war die Benutzerfreundlichkeit und die Integration in Java. Die Schnittstelle darf z.B. nicht überladen wirken, transparent in der Nutzung sein und sich bekannten Java-Konstrukten bedienen, sodass Entwickler sich damit schnell zurecht finden. Das Wissen, wie das SKilL-Serialisierungsformat aussieht oder funktioniert, wird nicht vorausgesetzt. Es sollte ausreichen, dass man sich nur mit dem Beschreibungsformat beschäftigt, um die Typen und deren Beziehungen zu definieren. Mit Hilfe dieser Definitionen und dem Codegenerator soll der Nutzer auf eine einfache Art und Weise eine Schnittstelle bzw. Teile davon generieren und in eigene Projekte einbinden können.

Eine konsistente Nutzung sowie gleiches Verhalten der Codegeneratoren für andere Sprachen soll - wenn möglich - gegeben sein, beispielsweise sollte der Codegenerator für Scala sich gleich bedienen wie der Codegenerator für Java. Dadurch müssen die Entwickler, die vielleicht für mehrere Sprachen entwickeln, nicht die Bedienung der einzelnen Generatoren erneut lernen.

Eine SKilL-Anbindung für eine weitere Programmiersprache konnte aus zeitlichen Gründen nicht implementiert werden. Somit hat sich die ursprüngliche Aufgabe leicht geändert. Statt der Kernsprache enthält die Sprachanbindung für Java mehrere weitere Konzepte, etwa die korrekte Behandlung von reservierten Java-Wörtern, sowie eine Sammlung an Tests, die sich gut eignen, um z.B. Regressionsfehler zu finden. Außerdem wurde ein grafischer Codegenerator entwickelt, um die Nutzung der Generatoren zu vereinfachen.

Da eine weitere Schnittstelle nicht realisiert werden konnte, wird die Arbeit unter anderem zeigen, wie die Java-Anbindung implementiert wurde. Ein großes Problem während der Anfertigung dieser Diplomarbeit war, dass nur eine SKilL-Anbindung existierte, nämlich die Schnittstelle für die Programmiersprache Scala [ski14]. Bis auf wenige Vorschläge in der SKilL-Spezifikation [Fel13] bezüglich der Implementierung konnte man nur das Scala-Binding als Referenz nehmen. Das war oft schwierig, da die Schnittstelle viele Scala-Konzepte verwendet, die man nur über Umwege in anderen Sprachen realisieren kann.

Die Implementierung enthält zwar einige Java-spezifische Konzepte, z.B. die Reflexion, allerdings kann man diese spezifischen Teile leicht durch andere Konstrukte ersetzen, sodass die Anbindung in ähnlicher Form für andere Programmiersprachen entwickelt werden kann. Diese Arbeit kann man daher als Hilfestellung für Implementierungen weiterer Anbindungen an SKilL nutzen.

Listing 1.1 Einfacher Typ “Date“ mit SKilL definiert

```
Date {
   v64 date;
}
```

1.3 SKilL

SKilL [Fel13] ist eine Serialisierungsspache, die an der Universität Stuttgart entwickelt wurde. Sie besteht aus der Beschreibungssprache und dem Serialisierungsformat. Die Sprache ist auf große Datenmengen ausgelegt und soll im Vergleich zu gängigen Formaten, wie etwa XML, deutlich schneller in der Serialisierung und beim Lesen sein. Die hohe Performance soll vor allem durch die Struktur sowie das Speichern im binären Format erreicht werden. Das Format erlaubt zudem eine kompakte Serialisierung.

Die Sprache ist plattform- und sprachunabhängig. Oft kommt es vor, dass Daten über viele Phasen gesammelt und verarbeitet werden. Laut Aufgabe ist es möglich, dass die einzelnen Phasen zudem in unterschiedlichen Sprachen implementiert werden. Sollten Anbindungen für diese mehrere Programmiersprachen existieren, ist es mit SKilL möglich, die serialisierten Daten leicht zwischen Tools und Phasen auszutauschen.

Ein weiteres Merkmal von SKilL ist die Einfachheit. Viele alternative Beschreibungsformate sind kompliziert und erfordern eine gewisse Einarbeitung. Ein Beispiel dafür wäre XML-Schema-Definition (XSD), das selbst für einfache Typen überladen und kompliziert sein kann. Das SKilL-Beschreibungsformat ist im Vergleich zu XSD intuitiver, nicht nur weil es keine besondere Struktur aufzwingt, wie z.B. die Baum-Struktur einer XML-Datei, sondern auch weil es einfachen Typdefinitionen in objektorientierten Sprachen ähnelt.

1.3.1 Beschreibungssprache

Die Beschreibungssprache kann verwendet werden, um Typen zu definieren.

Listing 1.1 ist ein Beispiel für die Definition eines einfachen Typen. Zuerst kommt der Name des Typen, danach innerhalb der geschweiften Klammern die Auflistung der Felder. Felder besitzen einen Typen sowie einen Namen.

Man unterscheidet zwischen eingebauten, zusammengesetzten und Benutzertypen, siehe Tabelle 1.1. Zudem können die Typen *i8*, *i16*, *i32*, *i64* und *v64* konstant sein. Der v64-Typ ist eine Sonderform von i64. Beide haben den gleichen Wertebereich, werden aber unterschiedlich serialisiert. *i64* hat immer die Länge von acht Bytes, *v64* kann je nach Wert zwischen einem und neun Bytes lang sein. In Java entsprechen beide Typen dem primitiven Typ long bzw. Long in Listen, Sets und Maps.

Alle Felder können die Attribute “auto“ besitzen. Solche Felder werden zwar im generierten Code sichtbar und nutzbar sein, allerdings werden sie bei der Serialisierung nicht berücksichtigt. Somit bedeutet “auto“ nicht das Gleiche wie in der Programmiersprache C++ oder C.

Kategorie	Typ	Java-Typ
Eingebaute Typen	annotation	SkillAnnotation
	bool	boolean
	i8	byte
	i16	short
	i32	int
	i64	long
	v64	long
	f32	float
	f64	double
	string	String
Zusammengesetzte Typen	T[i]	T[]
	T[]	T[]
	list<T>	java.util.List<T>
	set<T>	java.util.Set<T>
	map<T_1, ..., T_n>	java.util.Map<T_1, T_2>
Benutzertypen	T	T

Tabelle 1.1: SKilL-Typen und ihre äquivalenten Typen in der Java-Schnittstelle

Typen können Untertypen sein, sodass die Felder des Elterntypen vererbt werden. Listing 1.2 zeigt zwei Typen, mit *DatedMessage* als Untertyp von *Message*. Man kann nicht alle Typen erweitern, beispielsweise ein Untertyp von *string* ist nicht erlaubt.

Der Typ der Elemente in zusammengesetzten Typen wie Arrays oder Listen kann nur ein eingebauter oder Benutzertyp sein. Das bedeutet, dass z.B. Arrays keine Arrays oder Listen als Elemente enthalten können.

Es ist möglich zu kommentieren. Kommentare können - ähnlich zu Programmiersprachen - den Code verständlicher machen, indem sie ihn erklären oder beschreiben. Im generierten Java-Code bleiben sie als Javadoc-Kommentare in set- und get-Methoden erhalten.

Alle anderen SKilL-Konstrukte wie Beschränkungen (restrictions) oder Hinweise (hints) sind im Rahmen dieser Diplomarbeit irrelevant und werden nicht berücksichtigt, da sie zu den optionalen Anforderungen gehören.

1.3.2 Serialisierungsformat

Daten werden im binären Format serialisiert. Das Format besteht aus einer Folge von Blöcken. Es gibt zwei Arten von Blöcken, die String- und die Typblöcke.

Zuerst kommt ein Stringblock. Ein Stringblock ist immer vorhanden, selbst wenn er leer ist. Der Stringblock enthält am Anfang die Anzahl der Strings. Wenn keine neuen Strings existieren, wird 0

Listing 1.2 Untertypen

```
Message {
   string content;
}

DatedMessage extends Message {
   v64 date;
}
```

bzw. *null* als Byte-Wert - **0x00** - geschrieben und der Block ist damit am Ende. Wenn neue Strings (von der Schnittstelle) gefunden werden, folgen der Anzahl die sogenannten End-Offsets. Ein Offset beschreibt, in welchem Bereich sich ein einzelner String befindet. Danach kommen die eigentlichen Stringdaten im UTF-8-Format.

Alle Stringblöcke zusammen funktionieren ähnlich dem String-Pool in Java, in dem Strings nur einmal vorkommen, zumindest solange der Java-Entwickler nicht mit *new* neue Strings instanziiert. Alle Strings besitzen eine ID, die dem Index des Strings in den Stringblöcken entspricht. Der Index fängt bei 1 an. Strings in den Typblöcken werden mit diesen IDs referenziert und nicht direkt serialisiert.

Nach jedem Stringblock kommt ein Typblock, der im Header die Typinformation zu den Daten sowie anschließend die Felddaten enthält.

Als erster Eintrag steht - ähnlich wie in einem Stringblock - die Anzahl der Typen in dem Block. Danach folgt eine Liste von Definitionen zu jedem Typ. Jede Definition enthält am Anfang den Namen als Referenz (Index bzw. ID des Strings), der Name wird in jedem Typblock eingetragen. Dann kommt - wenn vorhanden - der Name des Basistypen. Falls der Typ ein Untertyp ist, wird noch der (lokale) Startindex im Pool (LBPSI) eingetragen. Danach kommt die Anzahl der Objekte von diesem Typ, die Beschränkungen und die Anzahl der Felder.

Anschließend werden alle Felder des Typen definiert. Jede Feldinformation enthält die Beschränkungen, den Feldtypen, den Namen des Feldes und das End-Offset, das ähnlich zu den End-Offsets in Stringblöcken ist und beschreibt, wo die Daten eines Feldes enden.

Nach den Definitionen der Typen und ihren Feldern kommen die eigentlichen serialisierten Daten nach Feldern gruppiert.

Abbildung 1.1 zeigt das typische Aussehen des binären Formats.

Ebenso wie Strings auch werden die Instanzen der Benutzertypen nicht direkt als Felddaten serialisiert. Stattdessen werden nur die Referenzen als *v64*-Wert eingetragen. Eine Referenz ist nichts anderes als der Index bzw. die ID des Objektes im jeweiligen Pool.

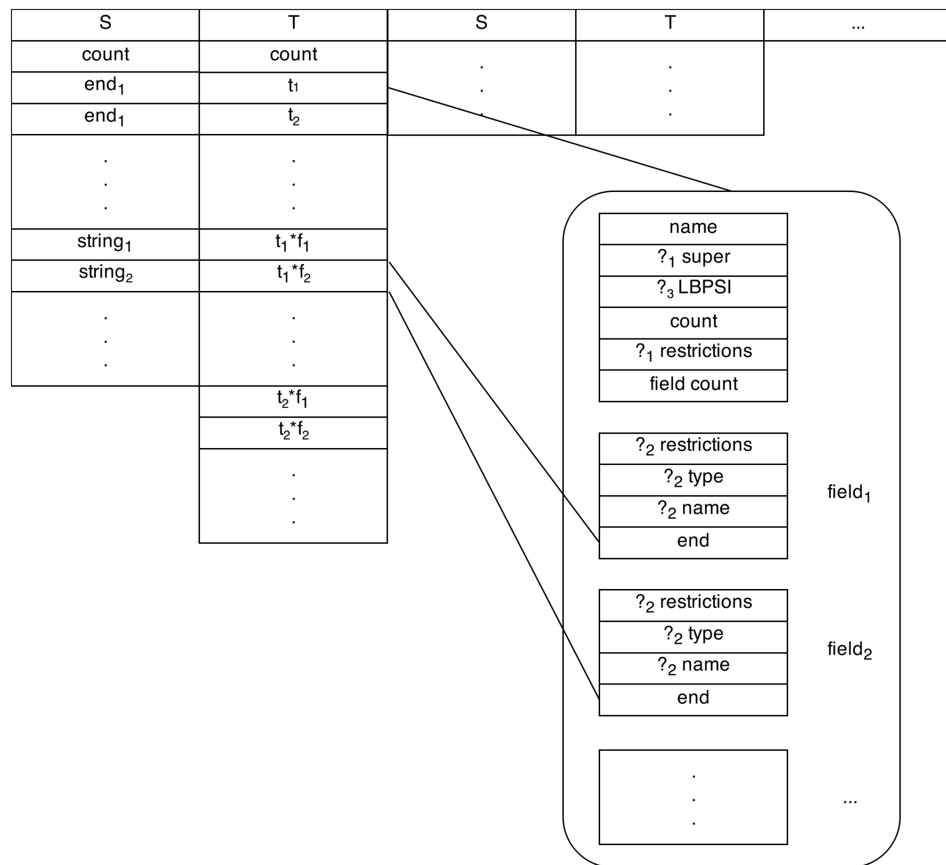

Abbildung 1.1: SKilL-Blöcke mit S (Stringblock) und T (Typblock) dargestellt. Daten, die mit $?_1$ markiert sind, werden nur beim ersten Vorkommen eines Typen eingetragen. $?_3$ ist nur da, wenn ein Elterntyp exisiert, und $?_2$ muss vorhanden sein, wenn ein Feld zum ersten Mal definiert wird.

1.3.3 Pools

Pools, manchmal auch Storage Pools genannt, sind Container, die nur Daten vom gleichen Typ enthalten. Alle Instanzen befinden sich in passenden Pools. Falls ein Typ ein Untertyp ist, befinden sich dessen Instanzen in dem Pool des Basistypen und es wird der Startindex (BPSI bzw. Base Pool Start Index) in der Typinformation eingetragen, ab dem die Instanzen des Untertypen im Pool vorkommen. Falls die Daten über mehrere Blöcke serialisiert werden, entspricht der BPSI-Wert dem lokalen Index, auch als LBPSI (Local Base Pool Start Index) bekannt.

Ein typischer Pool würde wie in der Abbildung 1.2 aussehen. Solange der Typ kein Untertyp ist, wird für jeden solchen Basistypen ein eigener Pool gebildet. Dadurch kann man Felder der Elterntypen an einer Stelle für alle Instanzen der Untertypen abspeichern. Das ermöglicht das Lesen von unbekannten Untertypen, sodass diese Untertypen immer noch als Basistypen instanziiert werden können.

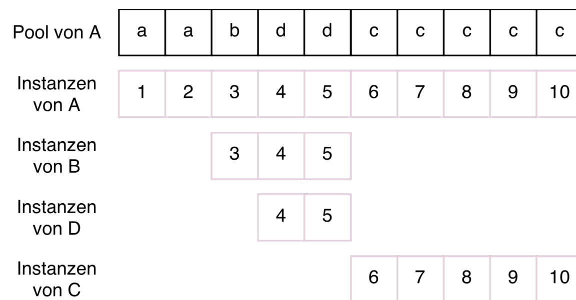

Abbildung 1.2: Dieses Bild zeigt einen typischen Storage Pool eines Basistypen A, der mehrere Untertypen, B und C hat, wobei der Untertyp B auch einen Untertypen D hat. Die obere Folge ist der Pool von A. Die unteren und grauen Folgen repräsentieren die Instanzen der jeweiligen Typen sowie ihre Indizes in dem Pool, die hilfreich bei der Suche nach Instanzen der jeweiligen Typen sind. Der Startindex (BPSI) von B ist 3, der von D ist 4, da man immer von dem Basistypen ausgeht, und der BPSI-Wert von C ist 6.

Wenn z.B. der Typ C in der Abbildung 1.2 der Schnittstelle unbekannt wäre, könnte die Schnittstelle die bekannten Teile der Felddaten von C-Instanzen auslesen und sie dem Nutzer in Form von A-Instanzen präsentieren. Das ermöglicht im geringen Maße das Arbeiten mit unbekannten Typen.

1.4 Schnittstelle und Codegenerator

Im Rahmen der Diplomarbeit sollen zwei Artefakte entstehen, die eigentliche Schnittstelle und der Codegenerator, der die Schnittstelle erzeugen soll. Diese Arbeit beschreibt, wie beide Artefakte entstanden sind und welche Entscheidungen hinter der Implementierung stehen.

Ein besonderes Merkmal der Schnittstelle soll vor allem die Benutzerfreundlichkeit, die Nutzbarkeit und die Integration in Java sein. Um das zu erreichen, wurde ein spezielles Design geschaffen, nach dem die Schnittstelle in zwei Teile getrennt wurde.

Ein Teil der Schnittstelle soll generisch sein und wenn möglich soll er als ein JAR-Paket ausgeliefert werden, das man in eigene Projekte als Abhängigkeit einbinden kann. Damit können bestimmte Funktionalitäten der Schnittstelle versteckt werden, sodass sie einer Blackbox ähnelt. Der Nutzer kann die meisten Klassen, die sich dort befinden, ignorieren. Es gibt außerdem weitere Vorteile eines generischen Schnittstellen-Teils, die im Kapitel 2 der Arbeit gezeigt werden.

Der zweite Teil der Schnittstelle wird mit dem Codegenerator generiert und beinhaltet Java-Klassen, die genau typisiert werden müssen. Java ist zwar statisch typisiert, allerdings kann man um die Typisierung in bestimmten Bereichen herum arbeiten und einen sehr generischen Code erzeugen, der sich so etwas wie Reflexion oder generischen Klassen wie *Object* und Interfaces bedient. Der generierte Teil beinhaltet unter anderem die Typklassen und Rückgabetypen in Methoden, die strikt

typisiert werden, damit der Nutzer sowie Entwicklungsumgebungen mit Autovervollständigung diese Typen leicht erkennen können.

1.5 Ähnliche Arbeiten

Diese Diplomarbeit bedient sich der SKilL-Spezifikation als Hauptquelle [Fel13], da zu dem Zeitpunkt, als diese Arbeit entstanden ist, keine weiteren Arbeiten zum Thema SKilL existierten.

Eine weitere Quelle ist außerdem die erste Schnittstellen- und Codegenerator-Implementierung für die Programmiersprache Scala [ski14]. Die Java-Schnittstelle ist zwar nicht der Scala-Schnittstelle nachgebildet, orientiert sich aber in Ansätzen daran.

Da SKilL relativ neu ist und sich immer noch in der Entwicklung befindet, konnte im Rahmen der Diplomarbeit Bezug auf ähnliche Formate und Schnittstellen genommen werden. Ein sehr bekanntes Beispiel ist XML/XSD mit der beliebten XML-Schnittstelle JAXB (Java Architecture for XML Binding) [jax14] und dem Codegenerator xjc (Java Architecture for XML Binding Compiler). JAXB ist vor allem wegen der Einfachheit populär geworden und zumindest in der Nutzung und Benutzerfreundlichkeit wurde es als eine Art Vorbild für die Implementierung der SKilL-Schnittstelle für Java genommen.

Für die Implementierung werden außerdem Erfahrungen anderer Entwickler eingeschlossen, um dem subjektiven Aspekt des Themas im Bezug auf Benutzerfreundlichkeit und Nutzbarkeit gegenüber zu treten. Benutzerfreundlichkeit ist ein beliebtes Gesprächsthema. Vor allem große Unternehmen beschäftigen sich damit intensiv, da sie oft Services in Form von Schnittstellen den Nutzern zur Verfügung stellen.

Zudem wurde eine Reihe an Studien als Quellen für einige Experimente genommen, um zu zeigen, was Nutzer als gut, benutzerfreundlich oder einfach empfinden, siehe [SGB+], [ESM], [RD10] usw. Manche dieser Studien waren sehr detailliert, mit Ergebnissen, die in quantitative (Tests, Messungen) und qualitative (Gespräche, Umfragen) unterteilt sind, z.B. [RD10].

1.6 Abgrenzung bei Begriffen

Im Bereich des Software Engineering gibt es viele Begriffe, die teilweise Synonyme sind oder ähnliche Dinge beschreiben. In diesem Abschnitt werden Begriffe aufgelistet und eventuell abgegrenzt.

Schnittstelle wird in dieser Arbeit oft im Zusammenhang mit der SKilL-Schnittstelle erwähnt. Das passiert allerdings nicht exklusiv. Der Begriff kann auch im allgemeinen Zusammenhang erwähnt werden und die Bedeutung ergibt sich aus dem Kontext. Verwendete Synonyme oder Spezifizierungen: *Programmierschnittstelle*, *API*, *Anbindung*.

Mit *Entwickler* ist meistens der Entwickler der Schnittstelle gemeint, nicht der Entwickler, der die Schnittstelle für seine Zwecke verwendet. Diese Gruppe ist unter dem Begriff *Nutzer* zusammengefasst. Allerdings ist auch hier keine Exklusivität gegeben, da es vorkommen kann, dass für eine bessere Beschreibung *Entwickler* als Synonym für *Nutzer* dient. Die Bedeutung ergibt sich dann aus dem Zusammenhang oder wird extra vermerkt.

2 Schnittstelle

Im Rahmen der Diplomarbeit sind zwei große Artefakte entstanden, die Schnittstelle und der Codegenerator. Die wichtigste der beiden Komponenten ist die Schnittstelle, die Nutzern erlaubt, Daten im SKilL-Format zu serialisieren. Dieses Kapitel geht auf die Schnittstelle genauer ein. Es wird das Design des Codes beschrieben, um einen Einstieg in das Thema zu geben. Im Kapitel 3 wird anschließend die API im Bezug auf die Nutzbarkeit untersucht.

2.1 Grundlegende Anforderungen an die Schnittstelle

Vor jeder Implementierung muss sich jeder Entwickler fragen, was er überhaupt entwickeln muss. Dafür wird in der Regel eine Liste mit Anforderungen an die Software erstellt, die z.B. die Features der fertigen Software beschreiben.

Die Anforderungen sind ein wichtiger Bestandteil im Entwicklungsprozess jeder Software. Sie sind Teil der Spezifikation und definieren Vorbedingungen, Abläufe und Nachbedingungen. Sie werden während der Analysephase, also im ersten Schritt des Entwicklungsprozesses, aufgestellt, und dienen als Vorlage für den Entwurf der Schnittstelle im nächsten Schritt.

Im Fall der SKilL-Schnittstelle gab es bereits eine Liste an Anforderungen in der SKilL-Spezifikation. Diese Anforderungen wurden in zwei Teile getrennt: Das "core language", auch Kernsprache genannt, und optionale Funktionalität [Fel13].

Kernsprache, die unterstützt werden muss...

- Integer-Typen: i8 bis i64 sowie v64
- string, bool und Annotationen
- Zusammengesetzte Typen
- Benutzertypen und Untertypen
- Konstante und auto-Felder
- Reflexion

Optionale Features, die implementiert werden können aber nicht müssen.

- Float-Formate (f32 und f64)
- Beschränkungen (restrictions)
- Hinweise (hints)
- Sprachabhängige Behandlung von Kommentaren, z.B. Integration in Javadoc
- Bezeichner-Behandlung, falls Bezeichner reservierte Wörter in Programmiersprachen sind, z.B. "super" in Java
- Reflektives Interface, welches das Lesen von Instanzen unbekannter Typen erlaubt

Die Java-Schnittstelle unterstützt die Kernsprache komplett sowie Teile der erweiterten Funktionalität.

Von den optionalen Features unterstützt die umgesetzte Schnittstelle die Float-Typen, die Behandlung von in Java reservierten Wörtern, die Integration der Kommentare sowie zum Teil das Auslesen von unbekannten Typen, allerdings nur, wenn die Typen Untertypen sind und der Basis- bzw. der Elterntyp bekannt ist. Solche Typen werden als Instanzen der Eltertypen erkannt, während unbekannte Felder übersprungen werden. Komplett unbekannte Typen, z.B. Basistypen, werden ebenso übersprungen. Falls ein Typ Felder von unbekannten Benutzertypen enthält, werden diese Felder nicht ausgelesen.

Beschränkungen sind immer leer und werden als *null* serialisiert.

Weitere Features, die nicht in der SKilL-Spezifikation erwähnt werden und sich nur in der Aufgabenbeschreibung befinden:

- Benutzerfreundlichkeit
- Einfachheit
- Integration in Java

Das sind alles sogenannte nichtfunktionale Anforderungen, die im Vergleich zu funktionalen Anforderungen in der Regel nicht im Vordergrund stehen. In dieser Arbeit ist das nicht der Fall. Die Arbeit soll untersuchen, ob es möglich ist, eine einfache Schnittstelle für Java zu entwickeln, um SKilL attraktiver in der Nutzung zu machen.

Während funktionale Anforderungen sich präzise erfassen lassen, sind die nichtfunktionalen Anforderungen vage und müssen unter Umständen besser definiert werden. Das ist oft schwierig, da solche Anforderungen subjektiver Natur sind, daher muss man als Entwickler die Zielgruppe einbeziehen. Eine Schnittstelle soll z.B. nicht zu einfach sein, sodass die Funktionalität leidet, dennoch sollte sie nicht zu kompliziert sein und den Nutzer nicht überfordern.

Die nichtfunktionalen Anforderungen muss man zudem in viele kleine Anforderungen aufteilen. Benutzerfreundlichkeit ist ein Oberbegriff, der viel erfassen kann. Deswegen ist es ratsam, weitere Anforderungen zu definieren, die man mit solchen Obergriffen gruppiert. Im Bezug auf die

Benutzerfreundlichkeit können z.B. Anforderungen wie die Wahl der passenden Bezeichner, Dokumentation, Lesbarkeit, Strukturiertheit usw. sammeln. Die Arbeit geht im Abschnitt 2.2 genauer auf diese Teilanforderungen ein.

2.2 Analyse der Anforderungen

In diesem Unterkapitel werden die Aktionen der Nutzer analysiert und zusammengefasst. Damit wird die Zielgruppe genauer definiert und weitere genaue Anforderungen an die Schnittstelle spezifiziert, die über die groben Anforderungen der Aufgabenbeschreibung hinausgehen.

Diese Analyse behandelt nicht die Nutzung des Codegenerators, auf diesen wird im Kapitel 4 eingegangen.

Ein interessanter Ansatz, um Anforderungen an die Schnittstelle zu finden, wurde in der Fallstudie "A Case Study of API Redesign for Improved Usability" [SGB+] gezeigt. In der Fallstudie ging es um die Vereinfachung einer vorhandenen API in dem Softwareunternehmen SAP, da diese Schnittstelle laut Anwendern zu umständlich in der Bedienung war. Um genaue Anforderungen und Anwendungsfälle während der Analysephase zu definieren, haben SAP-Mitarbeiter für die Nutzer der Schnittstelle eine neue Pseudofunktionalität der API erfunden. Danach mussten die Nutzer mit Pseudocode diese Funktionalität verwenden. Anschließend wurde der erzeugte Nutzer-Code von den Schnittstellen-Entwicklern analysiert und die Schnittstelle entsprechend erweitert, sodass sie den Erwartungen der Anwender entsprach.

Im Rahmen dieser Arbeit ist eine ähnliche Durchführung der Analyse unrealistisch, da nicht genug SKilL-Nutzer vorhanden sind, allerdings könnte man es für kommende Versionen der Schnittstelle versuchen. Stattdessen wird hier versucht, eine Reihe an Anwendungsfällen (Use-Cases) zu finden, die von der Schnittstelle auf eine einfache Art und Weise unterstützt werden müssen. Der Ansatz ist einfach und erfordert nicht viel Zeit.

2.2.1 Anwendungsfälle

Angenommen es gibt einen Entwickler, der Daten mit SKilL serialisieren möchte. Man muss herausfinden, was er von einer Schnittstelle erwartet, sodass diese Schnittstelle passend entworfen werden kann. Die Schnittstelle soll von Entwicklern nicht zu viel Wissen im Bezug auf SKilL verlangen.

Für Anforderungen sollen Vor- und Nachbedingungen aufgestellt werden. Das wird empfohlen, um z.B. einen bestimmten Einstiegspunkt bei einer Aktion oder Operation zu geben.

Es gibt mehrere Anwendungsfälle, die bestimmte Handlungen erfordern. Diese Abläufe gehen auch auf die Zustände ein, da je nach Zustand unterschiedliche Abläufe nötig sind.

Am Anfang jedes Ablaufs stehen die Daten, die gesammelt wurden und serialisiert werden müssen. Diese Daten befinden zunächst im Arbeitsspeicher. Es kann sein, dass diese Daten außerdem in eigenen bzw. internen Datenformaten untergebracht sind. Hier sind mit Datenformaten Instanzen eigener Klassen gemeint. Das ist eine übliche Praxis, um die Software modular und von der Persistenzschicht bzw. dem Persistenzmodul getrennt zu halten. Für die Arbeit und die kommenden Anwendungsfälle

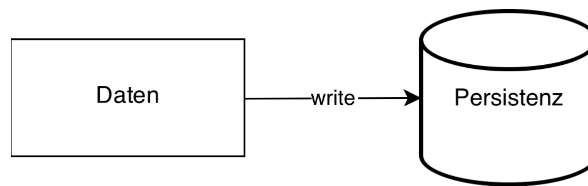

Abbildung 2.1: *write*-Operation, die von jeder SKilL-Schnittstelle unterstützt werden muss.

wird angenommen, dass der Nutzer sich für SKilL entschieden hat. Weiter sei anzunehmen, dass er mit dem Codegenerator die Schnittstelle generiert hat und die Daten bereits in Instanzen der Typklassen der Schnittstelle für die Serialisierung bereit stehen.

write

Der erste Anwendungsfall behandelt die *write*-Operation. Abbilding 2.1 zeigt den einfachen Ablauf, den die Schnittstelle unterstützen muss. Diese Operation kann der Nutzer aufrufen, wenn die Daten zum ersten Mal oder wenn sie in einer neuen Datei serialisiert werden.

Für diese und weitere Operationen empfiehlt es sich, eine Klasse zu definieren, die solche Befehle ausführen kann. Eine Instanz der Klasse soll in der Lage sein, den Zustand der Daten im Speicher in die Persistenz-Schicht zu übertragen. Da für SKilL bereits eine Schnittstelle zum Anfang dieser Diplomarbeit zur Verfügung stand, konnte aus Konsistenz-Gründen eine ähnliche Klasse aus der Scala-Schnittstelle im Ansatz übernommen werden. Die Klasse wird "SkillState" genannt, auch bekannt als State- oder Zustandsklasse.

Die Zustandsklasse kann Daten verwalten und bietet Methoden an, um diese Daten zu serialisieren. Dazu gehört auch die *write*-Operation. Um den Nutzer nicht zu überfordern, sollte die write-Methode nur einen Parameter akzeptieren, den Pfad bzw. Namen der Datei, in welche die serialisierten Daten gespeichert werden.

append

Die *append*-Operation ist eine Möglichkeit Daten schneller zu serialisieren. Während *write* alle Daten serialisiert, versucht *append* nur die Daten zu serialisieren, die seit dem Lesen hinzugefügt wurden und sich somit noch nicht in der Datei befinden. Diese neuen Daten werden in Form eines neuen String- und Typblocks an das Ende der Datei angehängt.

Abbildung 2.2 zeigt den typischen Ablauf, bei dem *append* verwenden wird. Eine wichtige Bemerkung für diese Operation ist, dass *append* nicht nur neue Instanzen hinzufügen kann. Es ist auch möglich, neue Felddaten von bereits vorhandenen Instanzen zu serialisieren, z.B. wenn ein Typ neue Felder bekommt. Das wird von SKilL erlaubt und ist manchmal auch erwünscht.

Die Zustandsklasse soll möglichst einfach diesen Anwendungsfall behandeln. Es wird nicht geraten, zwei Methoden für *write* und *append* zu implementieren. Stattdessen empfiehlt es sich eine einzige

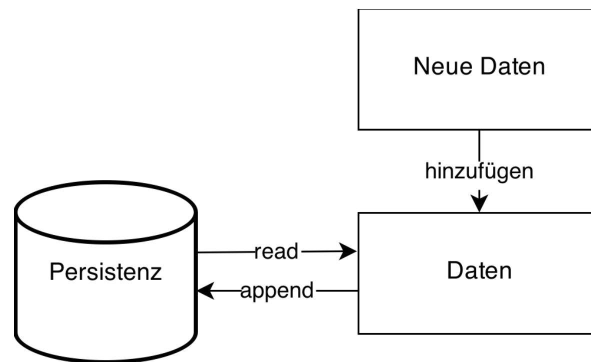

Abbildung 2.2: *append*-Operation, die neue Daten effizient in eine Datei hinzufügen kann.

Methode zu haben, die beide Fälle abdeckt. Die Methode soll automatisch erkennen, welche Operation ausgeführt werden soll.

Das ist leicht machbar. Es reicht aus, wenn beim Lesen einer Datei die Zustandsklasse sich diese Datei merkt und beim nächsten Aufruf der Serialisierungsmethode eine *append*-Operation ausführt, zumindest solange der Name der Datei sich nicht geändert hat. Der Name kann vom Nutzer als String-Parameter übergeben werden. Es reicht ein einfacher Vergleich des Namen bzw. Pfades, um zu entscheiden, ob *write* oder *append* ausgeführt wird.

Verwaltung der Daten

In SKilL werden die Daten in Pools verwaltet. Dieses Prinzip kann man auch in die Schnittstelle übertragen. Die Schnittstelle soll Klassen zur Verfügung stellen, die Pools repräsentieren. In Java empfiehlt es sich, bereits vorhandene Interfaces zu verwenden, um die Pool-Klasse zu implementieren. Es wird nicht empfohlen, vorhandene Collection-Implementierungen wie etwa Listen, Sets oder direkt Arrays zu verwenden, da Pools mehr als einfache Container für Daten sind.

Pro Typ soll es einen Pool geben. Die Pools sollen eine Möglichkeit anbieten, neue Instanzen des Pool-Typs hinzuzufügen. Unter Umständen können Pools auch als Objektfabriken agieren. Es wird empfohlen, für Untertypen getrennte Pools zu erzeugen, sodass die Daten leichter gefiltert werden können. Dennoch soll man über den Pool des Elterntypen auch die Instanzen der Untertypen aufrufen können.

Die Pools sollen nicht nur die Objekte sondern auch die Felder, zumindest indirekt, verwalten können. Es ist vor allem für die *append*-Operation wichtig, bei der erkannt werden muss, ob neue Felddaten existieren. In diesem Fall können z.B. nach dem Lesen einer Datei die bekannten Felder vermerkt und später zum Vergleich herangezogen werden, um zu erkennen, ob ein Feld neu oder bereits bekannt ist.

Pools sollen Möglichkeiten zur Verfügung stellen, leicht auf die Instanzen zuzugreifen. Dazu gehört z.B. ein Iterator, der genutzt werden kann, um durch den Pool effizient zu iterieren. Optional sollte man Objekte über die Indizes aufrufen können. In Java exisiert dafür das Interface *Collection*, das

das Interface *Iterable* erweitert. Allerdings definiert das Interface Methoden, die in Pools teilweise unerwünscht sind, z.B. *remove(E element)*, da das Entfernen von Objekten nicht direkt vorgesehen ist. Ohne spezielle Vorkehrungen, wie z.B. das Erzwingen einer *write*-Operation nach dem Entfernen von Objekten, können fehlerhafte Zustände entstehen.

Aus diesem Grund ist es ratsam, in Java nur das Interface *Iterable* zu implementieren und die Pool-Klasse nur um die Methoden ergänzen, die benötigt werden, wie etwa *add(E element)* oder *size()*. Es ist leichter etwas hinzuzufügen als etwas zu entfernen, vor allem wenn man eine Schnittstelle entwirft. Dazu mehr im Kapitel 3.

Wie die Pool-Klasse letztendlich implementiert wird, bleibt dem Schnittstellen-Entwickler überlassen. In Java ist es von Vorteil ein Interface oder eine abstrakte Klasse zu erstellen, die für unterschiedliche Einsatzzwecke unterschiedlich implementiert wird. Der Zugriff auf die Pools erfolgt dann über dieses Interface oder die abstrakte Klasse. So kann man schnell und unkompliziert die Implementierung ändern oder gar ersetzen, ohne die Schnittstelle zu verändern.

Pools sind Datenbehälter, die nicht nur für den Nutzer sondern auch für die Serialisierungsfunktionen wichtig sind. Während der Serialisierung braucht man teilweise spezielle Formen von Daten, etwa statische Typdaten, die nur Daten des Basis- und nicht der Untertypen sind. Dafür können z.B. Methoden bzw. Funktionen wie *getStaticData()* implementiert werden. Allerdings sind solche Methoden für den Nutzer sehr wahrscheinlich nicht wichtig und sollten nicht Bestandteil der öffentlichen Schnittstelle sein.

Behandlung von Typen

Ein besonderes Merkmal von SKilL sind die Typen. Einige von diesen Typen, z.B. *v64* oder *annotation*, können nicht direkt in die passenden Typen einer Programmiersprache abgebildet werden. Die Schnittstelle soll in der Lage sein, Typen in der Anwendung zu erkennen und sie korrekt nach SKilL-Definition zu serialisieren. Ein Feld vom Typ *v64* entspricht etwa dem Typ *long* in Java. Ohne spezielle Vorkehrungen kann die Schnittstelle nicht unterscheiden, ob es sich um *i64* oder *v64* handelt.

In Java entspricht ein konstantes SKilL-Feld einem Feld, das mit *final static* markiert wurde.

Arrays mit konstanter Länge für alle Instanzen eines Typen gibt es nicht in Java. In diesem Fall muss die Schnittstelle prüfen, ob ein Array-Feld der mit SKilL definierten Länge entspricht.

SKilL-Annotation ist ein einzigartiger Typ, der außerdem keine spezielle Abwandlung eines bekannten Typen ist. In objektorientierten Programmiersprachen kann man eine Klasse definieren, die der Generalisierung aller Benutzertypen entspricht. Ein Basistyp ist demnach ein Untertyp der SKilL-Annotation-Klasse. Alternativ kann man in Java zur Vereinfachung ein Interface nutzen, da Annotationen nur Zeiger auf beliebige Benutzertypen repräsentieren und keine Implementierung enthalten.

Tabelle 1.1 zeigt einige SKilL-Typen und wie sie in Java abgebildet werden.

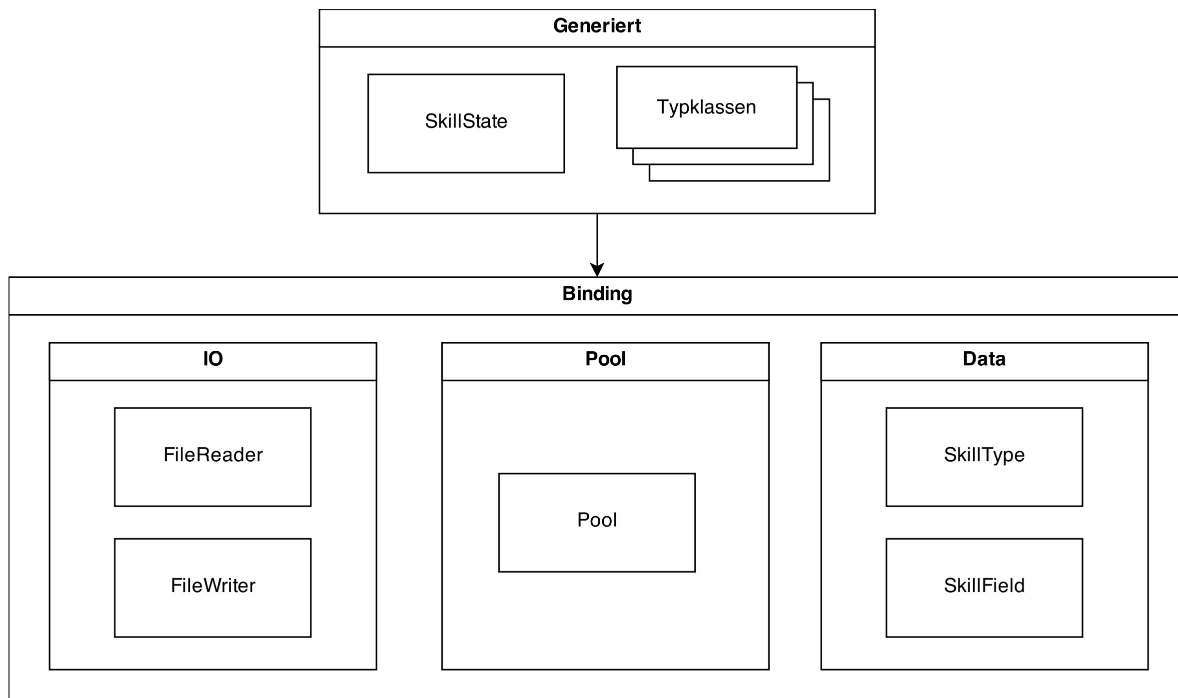

Abbildung 2.3: Die grobe Architektur der SKilL-Schnittstelle für Java.

2.3 SKilL-Schnittstelle für Java

Ein großer Teil dieser Diplomarbeit war die Implementierung der SKilL-Schnittstelle für die Programmiersprache Java. Die Schnittstelle sollte nicht nur einfach die Kernsprache von SKilL abdecken, sondern auch auf die Nutzung und somit Benutzerfreundlichkeit und Einfachheit eingehen.

2.3.1 Architektur der Schnittstelle

Abbildung 2.3 zeigt die Architektur der SKilL-Schnittstelle für Java. Der Ansatz beinhaltet zwei Teile, einen generischen Teil und einen generierten. Der generische Teil heißt "Binding" und wird auch so in dieser Arbeit und im Code genannt, z.B. als Paket- und Projektname. Der generierte Teil wird von dem Codegenerator erzeugt.

Binding ist oft das, was der Nutzer nicht kennen muss. Die Idee hinter einem generischen Teil der Schnittstelle war bestimmte Funktionalität von dem Nutzer zu verstecken. Viele Klassen, die sich in *Binding* befinden, sind für die Nutzung der Schnittstelle unwichtig. Sie sind Teil der Implementierung aber nicht Teil der Schnittstelle.

Der einzige für den Nutzer interessante Teil des *Bindings* sind die Pools, wobei dort nur die abstrakte *Pool*-Klasse von Bedeutung ist. Nur sie wird vom Nutzer verwendet. Es ist nicht von Bedeutung, ob die

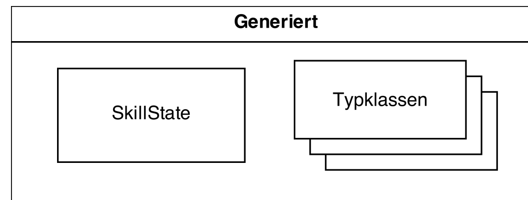

Abbildung 2.4: Der generierte Teil der Schnittstelle

verwendete *Pool*-Klasse in Wirklichkeit eine Instanz von *BasePool* oder *SubPool* ist, die Erweiterungen von *Pool* darstellen, siehe Abschnitt 2.3.3.

2.3.2 Generierter Teil der Schnittstelle

Der generische Teil wird von dem Codegenerator erzeugt. Er nimmt die Typbeschreibungen, die mit SKilL definiert wurden und erzeugt für jeden Typen eine passende Typklasse, die nicht nur zum Instanziieren gut ist, sondern auch als Typinformation während der Serialisierung agiert. Außerdem wird vom Codegenerator eine Zustandsklasse erzeugt, die Zugriff auf die Pools ermöglicht und Methoden zum Serialisieren von Daten bzw. zum Lesen von Dateien zur Verfügung stellt.

Typklassen

Typklassen sind ähnlich zu POJO (Plain Old Java Objects) [poj14]. Sie enthalten eine Liste privater Felder, die den mit SKilL definierten Feldern entsprechen. Danach folgt eine Liste mit passenden und öffentlichen set- und get-Methoden. Solange alle Felder keine SKilL-spezifischen Typen haben, wie etwa *v64*, ist eine Typklasse ein POJO. Wenn das nicht der Fall ist, bedient sich eine Typklasse Java-Annotationen, die die Klasse um zusätzliche Informationen erweitern. Mittels Reflexion können diese Informationen aus den Annotationen ausgelesen und beim Serialisieren verwendet werden.

Listing 2.1 zeigt die generierte Typklasse für den einfachen Date-Typ aus dem Listing 1.1.

Die Schnittstelle bildet die Felder vom Typ long standardmäßig nach i64 ab. Falls ein SKilL-Feld vom Typ *v64* ist, wird eine Java-Annotation verwendet, die den long-Typen präzisiert. Beim Serialisieren prüft die Schnittstelle, ob die Annotation existiert und nimmt dann den Typen, der dort gesetzt wurde. Auf diese Weise können unter anderem neue Feldtypen, die Konflikte verursachen, in Java definiert werden.

Das Annotation-Interface *FieldAnnotation* enthält eine Reihe weiterer Felder, die gesetzt werden können, um den Typen des Feldes zu beschreiben. Dazu gehört so etwas wie *collectionType*, das gesetzt wird, wenn der Elementtyp in einem Array oder in einer Liste vom Typ *v64* ist, oder *auto*, wenn das Feld als ein auto-Feld in SKilL definiert wurde.

Zudem befindet sich dort ein Platzhalter für die Beschränkungen, die im Rahmen dieser Diplomarbeit nicht implementiert wurden, aber dennoch später zu der Schnittstelle hinzugefügt werden können.

Listing 2.1 Beispiel für eine Typklasse

```java
// package name and imports
...

public class Date implements SkillAnnotation {

   @FieldAnnotation(type="v64")
   private long date;

   public long getDate() {
      return date;
   }

   public void setDate(long date) {
      this.date = date;
   }
}
```

Neben dem Annotation-Interface *FieldAnnotation*, die nur bei Feldern gesetzt werden kann, existiert eine weitere Annotation für Typklassen, *TypeAnnotation*, die zwar nicht verwendet wird, allerdings später für Typ-Beschränkungen einfach ergänzt werden kann.

Die Namen der Felder können am Anfang Unterstriche enthalten, was Teil der erweiterten Anforderungen an die Schnittstelle war, um in Typklassen Namenskollisionen mit geschützten Java-Wörtern zu vermeiden. Die Klassennamen werden immer groß geschrieben, somit können dort keine Kollisionen entstehen, da Java case-sensitive ist und alle geschützten Java-Wörter mit kleinen Buchstaben anfangen. Die Namen der Pools - als Felder in der Zustandsklasse - enthalten immer einen Unterstrich am Anfang, um die Handhabung zu erleichtern.

Zustandsklasse

Neben den Typklassen wird zusätzlich eine Zustandsklasse generiert. Der Zweck dieser Klasse ist die Vermeidung der Typumwandlung (Cast). Die Zustandsklasse agiert ähnlich der *JAXBContext*-Klasse in der XML-Schnittstelle JAXB. Dort kann der Nutzer Marshalling und Unmarshalling betreiben, um Daten zu serialisieren bzw. zu lesen. Die Context-Klasse ist generisch, dadurch kann beim Unmarshalling das Wurzelelement nur als Wert vom Typ *Object* zurückgegeben werden und der Nutzer muss anschließend das Wurzelelement umwandeln (casten), um es nutzen zu können.

Um dem Nutzer diesen nichtintuitiven Cast zu ersparen, ist es von Vorteil die Klasse mit statischer Typisierung zu generieren. Im Fall der SKilL-Schnittstelle für Java können nach dem Lesen Pools mit korrekten Parametertypen zur Verfügung gestellt werden.

Um das zu gewährleisten, muss die Zustandsklasse die Schreiben- und Lesen-Methode (*saveToFile(String fileName)* und *readFile(String fileName)*) aus *AbstractSkillState* implementieren, die sich im generischen Teil der Schnittstelle (*Binding*) befindet. In *saveToFile(String fileName)* werden nach dem Lesen statisch typisierte Pools instantiiert. In *readFile(String fileName)* werden die Typklassen und

Listing 2.2 Beispiel für eine Zustandsklasse, als Basis dient die Typdefinition 1.1

```
// package name and imports
...

public class SkillState extends AbstractSkillState {

   private Pool<Date> _date;

   public SkillState() {
      this._date = new BasePool<>();
   }

   public void readFile(String fileName) throws SkillSerializationException {
      Map<String, Object[]> data;

      data = this.readFile(fileName, getTypeClasses());
      Date[] dateData = this.getData(data.get("date"), Date[].class);
      this._date = new BasePool<>(dateData);
      this.setKnownFieldsForPool(this._date, "date");
   }

   public void saveToFile(String fileName) throws SkillSerializationException {
      this.saveToFile(fileName, getPools(), getTypeClasses());
   }

   private Pool<?>[] getPools() {
      Pool<?>[] pools = new Pool<?>[1];
      pools[0] = this._date;
      return pools;
   }

   private Class<?>[] getTypeClasses() {
      Class<?>[] typeClasses = new Class<?>[1];
      typeClasses[0] = Date.class;
      return typeClasses;
   }

   public Pool<Date> getDate() {
      return this._date;
   }
}
```

die Pools benötigt. Beides muss bekannt sein und sollte nicht vom Nutzer als Parameter übergeben werden, um die Benutzerfreundlichkeit der Schnittstelle zu verbessern.

Listing 2.2 zeigt die Zustandsklasse, die erzeugt wird, wenn der Nutzer dem Codegenerator nur den Typen *Date* aus dem Listing 1.1 übergibt.

Alles fängt klein an, so auch die Zustandsklasse beim Instantiieren. Im Konstruktor werden leere Pools erzeugt. Diese leeren Pools können als Container für initiale Daten dienen, die noch nicht serialisiert wurden. Alle Pool-Felder haben eine get-Methode. Setter für Pools sind unerwünscht, da sie unter Umständen zu negativen Effekten führen können, z.B. wenn der Nutzer entscheidet, dass

ein Pool auch *null* sein kann. Die Zustandsklasse ist damit eine Fabrikklasse für Pools, um korrektes Verhalten der Schnittstelle zu garantieren.

Anders verhält sich die Zustandsklasse, wenn der Nutzer eine Datei auslesen möchte. Beim Lesen einer Datei wird im generischen Teil der Schnittstelle eine Map erstellt, die Strings als Schlüssel nach Arrays als Wert abbildet. Die Strings sind die Namen der Basistypen, während die Arrays entsprechende Daten sind.

In der *readFile*-Methode wird zunächst diese Map referenziert und dann je nach Art des Typen wird eine passende Pool-Klasse instanziiert. Neue BasePool-Instanzen bekommen nur das konvertierte Array (mit richtigem Elementtypen statt *Object*) als Konstruktor-Parameter. Instanzen von SubPool bekommen als Parameter den Pool des Elterntypen, den sogenannten Superpool, sowie die Klasse des Typen, um im Superpool nach korrekten Instanzen des Typen zu suchen und um eine Referenz auf die kommenden neuen Instanzen des Typen zu setzen, die sich in einer ArrayList befindet.

Anschließend müssen in jedem Pool bekannte Felder gesetzt werden, sodass bei der nächsten *append*-Operation die fehlenden Felddaten serialisiert werden. Auch der Name bzw. der Pfad der Datei wird vermerkt.

Um Daten zu serialisieren, ruft der Nutzer die Methode *saveToFile(String fileName)* auf. Diese sammelt die zu serialisierenden Pools sowie passende Typklassen und ruft anschließend eine geschützte (*protected*) Methode mit dem gleichen Namen in *AbstractSkillState* auf, der sie die Pools und die Typklassen als Parameter übergibt.

Die Methode in *AbstractSkillState* entscheidet dann anhand der zur Verfügung stehenden Informationen, ob eine *write*- oder *append*-Operation ausgeführt wird. In jedem der beiden Fälle merkt sich die Schnittstelle den Namen der Datei, in die Daten geschrieben wurden, sodass beim nächsten Serialisieren in die gleiche Datei *append* ausgelöst wird.

2.3.3 Binding

Binding ist der Kern der Schnittstelle. Es beinhaltet alle Klassen, die für die Serialisierung der Daten oder das Lesen von Dateien nötig sind. Dabei bedient es sich einer Abstraktion mit einer schrittweisen (De-)Serialisierung.

Abstrakte Zustandsklasse

Die Eingangsklasse ist die abstrakte Klasse *AbstractSkillState*, die erweitert werden muss, um statische Typisierung in die Schnittstelle zu bekommen. Diese Erweiterungen sind die Zustandsklassen aus den generierten Teil der Schnittstelle, die im vorigen Abschnitt 2.3.2 beschrieben wurden.

AbstractSkillState definiert Teile der Schnittstelle der Zustandsklasse, z.B. die Methoden *readFile* und *writeToFile*. Andere Teile der Zustandsklasse wie die Getter der Pools können nicht vorgesetzt werden, da die Klasse generisch ist und somit statische Typen nicht kennt.

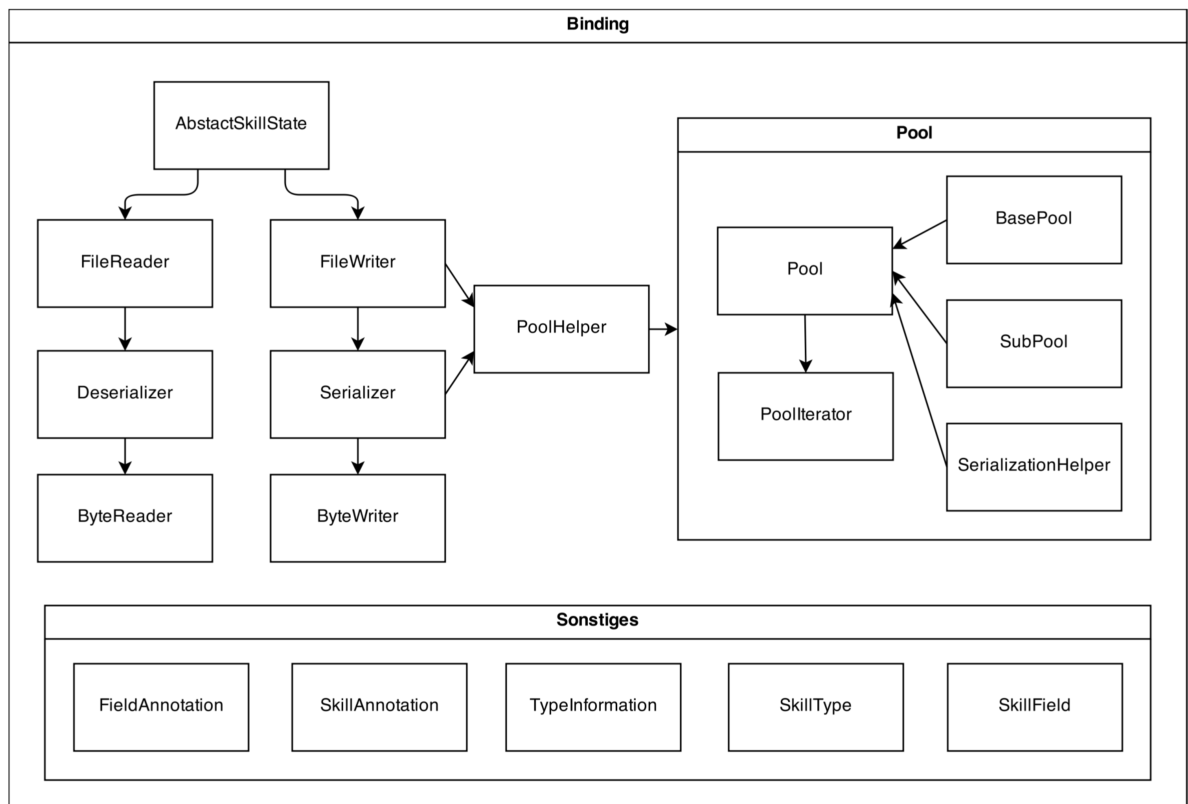

Abbildung 2.5: Der generierte Teil der Schnittstelle

Außerdem enthält die Klasse alle Methoden, die nicht sichtbar sein müssen, um die generierten Zustandsklassen kompakter zu halten, wie die Methode zur Konvertierung von *Object*- nach Typ-Arrays oder die Methode zum Setzen der bekannten bzw. erkannten Felder nach dem Lesen. Sie sind zudem nicht Teil der Schnittstelle und sollen für den Nutzer unsichtbar sein.

Pools

Alle für die Pools relevanten Klassen befinden sich in dem Paket *de.ust.skill.binding.pools.* Abbildung 2.6 ist ein Ausschnitt aus dem Entwurf der Schnittstelle und zeigt nur den Teil mit den Pools. Für den Nutzer sind nur zwei Klassen davon interessant, die abstrakte Klasse *Pool<T>* und der passende Iterator *PoolIterator<T>*. Die Klasse wird von *BasePool* und *SubPool* erweitert und je nach Art des Typen wird die entsprechende Pool-Unterklasse benutzt.

Pool ist eine Implementierung des *Iterable<T>*-Interfaces aus Java, die einen Iterator für Elemente erfordert. *PoolIterator* ist die Implementierung des *Iterator<T>*-Interfaces.

Ursprünglich wurde statt *Iterable* das *Collection<T>*-Interface von *Pool* implementiert. Das Problem mit *Collection* waren die teilweise unnötigen Methoden wie *remove(T element)* oder *clear()*, die mit SKilL Probleme bereiten können. Stattdessen wurde die Implementierung auf *Iterable* reduziert. Es

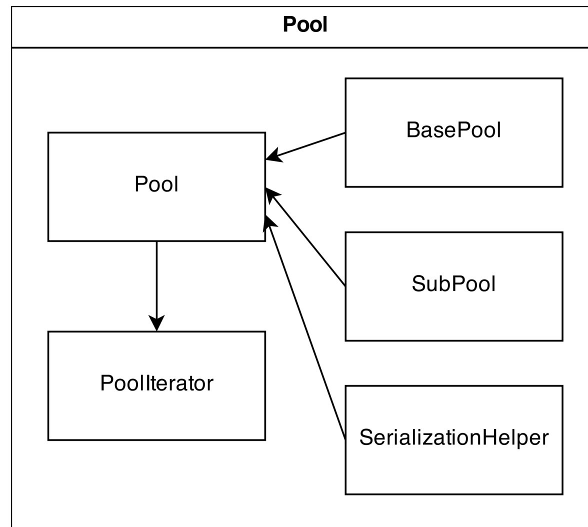

Abbildung 2.6: Pool-Klassen in der Schnittstelle

ist einfacher eine Schnittstelle später um neue Methoden zu ergänzen als vorhandene Methoden zu entfernen.

Neben einem Iterator stellt die Pool-Klasse dem Nutzer die Methoden *get(int index)*, *add(T element)* und *size()* zur Verfügung.

get(int index) gibt ein Element aus dem Pool zurück, das sich unter dem angegebenen Index befindet. Die Zugriffszeit der Methode ist konstant für Basistypen, linear für Typen mit Untertypen. Wobei die lineare Zugriffszeit nur das Worst-Case-Szenario darstellt, in der Praxis sollte es keine bedeutende Rolle spielen. Die lineare Zugriffszeit tritt nur dann ein, wenn ein Typ viele Untertypen enthält. Für jeden Untertypen existiert in den Pools eine Liste (als eine Instanz der ArrayList-Klasse) mit Objekten. Um auf ein Objekt aus so einer Liste zuzugreifen, muss der Reihe nach geprüft werden, ob der Index auf die richtige Liste zeigt. Ein kurzes Beispiel: Ein Typ hat zwei Untertypen, es gibt somit zwei Listen mit neuen Instanzen. Angenommen, die erste Liste enthält 20 Objekte, die zweite 30. Um z.B. das Objekt hinter dem Index 40 zu bekommen, muss auch die erste Liste untersucht werden. Es wird geprüft, ob die Länge der Liste kleiner als der Index ist. Wenn das der Fall ist, wie in diesem Beispiel, passiert ein Sprung in die nächste Liste. Somit muss man die Länge aller Listen abfragen, bis die richtige Liste erreicht ist. Das Worst-Case-Szenario wäre ein Typ mit vielen Untertypen, deren Pool-Listen maximal nur ein Objekt enthalten.

size() zählt alle Instanzen eines Typen im Pool, während *add(T element)* eine neue Instanz des Typen *T* in die Liste *newInstances* (als *ArrayList*-Feld in der Pool-Klasse) hinzufügt. Dieses *newInstances*-Feld wird unter anderem beim Serialisieren verwendet, um leichter zwischen neuen und bereits serialisierten Daten zu unterscheiden. Somit müssen für die *append*-Operation nur die Instanzen aus der *newInstances*-Liste berücksichtigt werden, da anzunehmen sei, dass die bereits serialisierten Daten in einer Datei sich nicht zwischenzeitlich geändert haben. Es existiert allerdings kein Sperr-Mechanismus für Dateien, sodass nicht garantiert werden kann, ob eine Datei sich nicht geändert

hat. Als alternativen Weg zu der Annahme, dass die Datei gleich ist, können von der Schnittstelle Hashing-Funktionen auf die Dateien angewendet werden, die das Risiko korrupte Daten zu schreiben, deutlich minimieren. Eine andere Alternative wäre die Anwendung der Hashing-Funktionen auf die Daten im Arbeitsspeicher und in den Dateien, was allerdings kompliziert und zu langsam ist, um daraus einen Vorteil für die Datenkonsistenz zu gewinnen.

Es gibt außerdem eine Reihe zusätzlicher privater und geschützter (*protected*) Methoden, die allerdings nur für die Serialisierung von Bedeutung und daher nicht Teil der öffentlichen Schnittstelle sind. Um die protected-Methoden aufzurufen, wurde die *SerializationHelper*-Klasse im gleichen Paket implementiert. Im Code wird sie *PoolSerializationHelper* genannt. Diese Klasse ist, grob ausgedrückt, eine Schnittstelle für die Schnittstelle, die Pools und Klassen zum Schreiben von Dateien verbindet. Sie enthält unter anderem Methoden, um statische Daten aus den Pools zu laden oder neue Instanzen zu holen, oder ganz einfach Methoden für Größenangaben für diverse Zwecke, wie etwa die Anzahl der neuen Instanzen eines Typen für die *append*-Operation.

Anzumerken sei noch, dass in bestimmten Fällen die Instanzen nicht in der Reihenfolge in den Pools liegen, in der sie dort eingefügt wurden. Falls Untertypen existieren, werden die Instanzen des Elterntypen vor den Instanzen der Untertypen eingefügt. Das ergibt sich aus dem SKilL-Serialisierungsformat, in dem zuerst die Objekte des Basis- bzw. Elterntypes serialisiert werden.

Lesen von Dateien

Abbildung 2.7 zeigt den ungefähren Ablauf beim Lesen einer Datei.

Das Lesen von Daten erfolgt in einer Abstraktion. Damit beschäftigen sich hauptsächlich vier Klassen, *AbstractSkillState*, *FileReader*, *Deserializer* und *ByteReader*. Wobei noch weitere Klassen für Typinformationen (*SkillType* und *SkillField*) sowie SKilL-Vorgaben (*TypeInformation*) verwendet werden.

ByteReader repräsentiert die unterste Ebene und bewegt sich im Byte-Bereich. In dieser Klasse werden Bytes aus einer Datei gelesen und zu primitiven Typen zusammengefasst. Ein serialisierter Integer-Wert hat z.B. die Länge 32 Bit. D.h., dass in so einem Fall ByteReader vier Bytes liest und diese dann in einen Java-Integer-Wert konvertiert und zurückgibt. Das passiert, wenn die Methode *ByteReader.i32()* aufgerufen wird.

Somit werden solche primitiven Typen wie *Byte*, *Short*, *Integer*, *Long* usw. aber auch *v64* aus einer Datei ausgelesen und mit dem richtigen Typ zurückgegeben. Auf dieser Ebene findet man nichts zu Pools oder gar Objekten bzw. Instanzen von Typklassen. Die einzigen Vorgänge hier sind nur das Öffnen bzw. Schließen von Dateien und Lesen von Byte-Folgen, die in primitive Typen konvertiert werden.

ByteReader bedient sich der Java-Klasse *FileChannel*, die Bereiche von Dateien auf physischen Datenträgern im Speicher abbildet und es erlaubt, durch die Datei Byte für Byte zu iterieren. Weiterhin ist es möglich, Bereiche zu überspringen. Zurzeit wird die Funktion zum Überspringen von unbekannten Felddaten und Typen benutzt, kann aber später für *Lazy Handling* von Dateien verwendet werden.

In der Schicht über dem *ByteReader* befindet sich die Klasse *Deserializer*, die *ByteReader* gezielt steuert, um etwas komplexere Daten zu holen, wie Arrays oder Listen, wobei sie auch primitive Typen

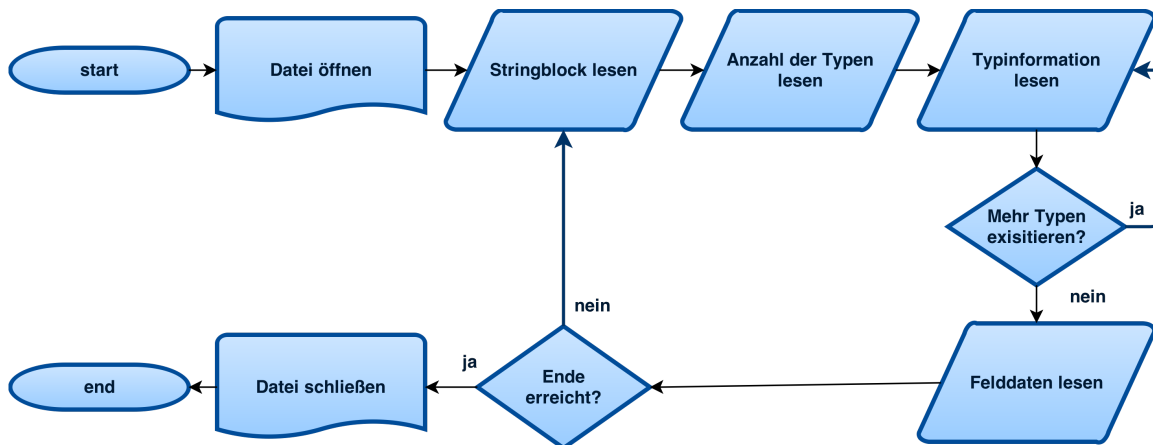

Abbildung 2.7: Ablaufdiagramm, das den Leseprozess darstellt

durchlässt. Ein Array ist nichts anderes als die Anzahl der Elemente am Anfang gefolgt von einer Folge an Elementen. So etwas ist logischerweise ein zusammengesetzter Typ, bei dem die Anzahl der Elemente einer Zahl vom Typ *v64* entspricht, gefolgt von eigentlichen Elementen eines Typen. In diesem Fall liest *Deserializer* zuerst eine *v64*-Zahl mit *ByteReader* und dann werden mit einer for-Schleife die Elemente als primitive oder Benutzertypen ausgelesen. Dazu erzeugt *Deserializer* ein neues Array-Objekt mit der entsprechenden Länge und setzt die Elemente, das anschließend an FileReader übergeben wird, um es im passenden Feld einer Typ-Instanz zu setzen.

Ähnlich funktioniert das auch für andere Typen wie Listen, Sets oder Maps, aber auch für Strings und Benutzertypen. Obwohl die Benutzertypen-Objekte eigentlich in der Schicht darüber erzeugt werden - in der Klasse *FileReader* - werden sie dennoch im Deserializer referenziert, sodass diese Objekte als Elemente in z.B. Listen gesetzt werden können.

Deserializer enthält auch Funktionen zum Serialisieren von Feldtypen, die wiederum nichts anderes als zusammengesetzte Informationen sind. Dazu gehört etwa die Referenz auf den Feldnamen im String-Pool oder das End-Offset für Felddaten als *v64*-Zahl.

Kurz zusammengefasst enthält *Deserializer* Funktionen bzw. Methoden, die beliebige Felddaten mit Hilfe von *ByteReader* auslesen können.

In der Schicht darüber, in der Klasse *FileReader*, werden diese mit *Deserializer* ausgelesenen Felddaten den Objekten oder Typinformationen zugeordnet. Pro String- und Typblock-Paar werden zuerst die Strings aus dem Stringblock gelesen und in ein Array hinzugefügt, anschließend werden die Typinformationen aus dem Header des Typblocks gelesen. Mit diesen Typinformationen werden neue Objekte instanziiert, anschließend werden die Felddaten ausgelesen und den Objekten zugeordnet.

FileReader verteilt alle Objekte in Arrays. Für jeden Basistypen existiert ein Array. Diese Arrays befindet sich in einer Map, die Strings nach Object-Arrays abbildet. Strings als Schlüssel entsprechen den Typnamen. Die Arrays werden später von Pools referenziert und werden dort als *data*-Feld verwendet, um auf die bereits serialisierten Objekte zuzugreifen.

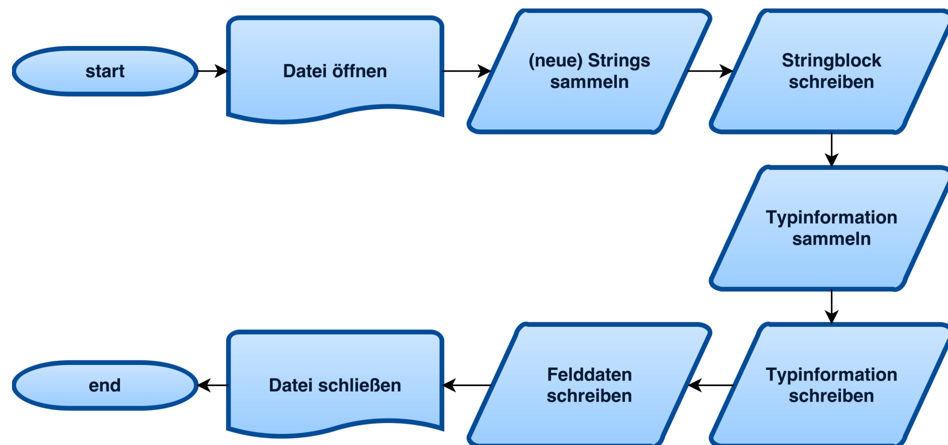

Abbildung 2.8: Ablaufdiagramm für den Serialisierungsprozess

Die letzte Schicht in der Abstraktion ist die Zustandsklasse, wobei hier die generierte Zustandsklasse sowie *AbstractSkillState* als Einheit verstanden werden. In der Zustandsklasse werden die Pools erzeugt und die Objekte, die im *data*-Array liegen, den Pools zugeordnet. Außerdem merkt sich die Zustandsklasse die Strings, um für *append* die bekannten Strings auszuschließen.

Serialisierung

Abbildung 2.8 zeigt den Ablauf der Serialisierung in vereinfachter Darstellung.

Ähnlich wie das Lesen von Dateien wird auch die Serialisierung abstrahiert. Der einzige grobe Unterschied zum Lesen ist die Richtung der Abstraktion. Während beim Lesen die Daten von Bytes zu Pools werden, ist es bei der Serialisierung genau umgekehrt.

Die Daten werden von der Zustandsklasse und den Klassen *FileWriter*, *PoolHelper*, *Serializer* und *ByteWriter* serialisiert. Die Rolle der Zustandsklasse ist simpel: Sammeln von Pools und passenden Typklassen, die als Typinformation dienen. Diese werden an *FileWriter* übergeben. Dort wird auch festgelegt, ob *write* oder *append* ausgeführt wird. Für *append* wird zusätzlich das Array mit bekannten Strings übergeben.

FileWriter durchsucht zuerst alle Pools nach neuen Strings, sammelt diese in einem Set und serialisiert sie danach in einem neuen Stringblock. Das Set wird behalten, um später nach den passenden Indizes zu suchen, wenn ein String referenziert wird.

Nachdem der Stringblock geschrieben wurde, sucht *FileWriter* nach Informationen zu den Typen. Die Typklassen sind hierbei die erste Anlaufstelle. Mittels *PoolHandler* werden die Klassen und die passenden Pools nach relevanten Informationen durchgesucht. Dafür werden Instanzen von *SkillType* und *SkillField* angelegt. *SkillType* enthält solche Informationen wie den Namen des Typen, Anzahl der Objekte usw. Die Klasse enthält außerdem eine Liste an *SkillField*-Instanzen, die Typfelder beschreiben oder End-Offsets enthalten.

Diese Offsets werden für jeden Feldtypen explizit berechnet. Solche Berechnungen sind oft sehr einfach und schnell. Beispielsweise beträgt die Länge von 10 Byte-Feldern in serialisierter Form 10 Bytes, für 10 Integer-Felder 40 Bytes usw. Für komplexere Feldtypen wie *v64*-Referenzen (für Objekte oder Strings) muss die Schnittstelle aber jeden Wert für die Berechnung berücksichtigen, womit das Ganze langsamer wird.

Die Scala-Schnittstelle [ski14] funktioniert anders. Dort werden die Daten zuerst serialisiert und die End-Offsets ergeben sich aus den Größen der serialisierten Daten, die vorerst im Speicher liegen müssen. Dieser Ansatz ist zwar schneller, benötigt aber mehr Arbeitsspeicher. Der Ansatz der Java-Schnittstelle ist für manche Typen schnell, z.B. Bytes, für andere wiederum allerdings langsam, z.B. *v64*, da Felder doppelt aufgerufen werden müssen, einmal um End-Offsets zu berechnen und dann um die Daten zu serialisieren. Dafür ist der Speicherverbrauch minimal.

Alle diese Typinformationen werden anschließend im Header eines neuen Typblocks gespeichert. Die Informationen sind nach Typ aufgeteilt und werden an eine Instanz von *Serializer* übergeben, die die Daten weiter aufteilt und mit dem *ByteWriter* in eine Datei schreibt.

ByteWriter ist ähnlich zu *ByteReader* und kennt nur primitive Typen wie Byte, Integer, *v64* usw. Zum Schreiben verwendet es die Java-Klasse *BufferedOutputStream*. Es existiert ein Byte-Buffer, dessen Größe standardmäßig 1MB beträgt. Die Größe kommt von einfachen Experimenten und Tests. Zu kleine Buffer führten zu langsamen Schreibvorgängen, während große Buffer mit 4-8MB oder mehr keinen deutlichen Anstieg in der Geschwindigkeiten zeigten.

ByteWriter füllt den Buffer, solange dort Platz für neuen Daten vorhanden ist. Wenn der Buffer zu voll ist, also wenn die Größe des zur Verfügung stehenden Platzes kleiner als die Anzahl der zu serialisierenden Bytes ist, wird der Buffer mit *BufferedOutputStream* in die Datei geschrieben und anschließend geleert. Eine Ausnahme dabei sind die Strings, die direkt in die Datei geschrieben werden, da von allen primitiven Typen Strings deutlich größer als der Buffer sein können und die Aufteilung von Strings bzw. entsprechende Byte-Folgen in kleinere Stücke zu zeitintensiv und unnötig kompliziert wäre.

Serializer ähnelt im Prinzip *Deserializer*, nur mit umgekehrter Funktionalität. Es enthält Serialisierungsfunktionen für Felddaten beliebiger Typen. *Serializer* akzeptiert nicht nur primitive Typen sondern auch zusammengesetzte wie Arrays, Listen oder Feldtypen, die für *ByteWriter* in Daten primitiver Typen aufgeteilt werden. Für Strings oder Objekte von Benutzertypen sucht es passende Indizes, die als *v64* serialisiert werden. Für *annotation*-Felder sucht es nach dem Namen des Basistypen als *v64*-Referenz.

Listing 2.3 Einfaches Beispiel für die Nutzung der Schnittstelle

```java
package com.company.special.project;

import de.ust.skill.binding.pools.Pool;

import de.ust.skill.generated.date.Date;
import de.ust.skill.generated.date.SkillState;

public class SkillAPIUsageExample1 {

   public static void main(String[] args) {

      SkillState state = new SkillState();
      String fileName = "/path/to/dates.sf";

      state.readFile(fileName);
      Pool<Date> dates = state.getDate();

      for (Date date : dates) {
         System.out.println(date.getDate());
      }

      Date newDate = new Date();
      newDate.setDate(3);
      dates.add(newDate);

      state.saveToFile(fileName);
   }
}
```

2.4 Nutzungsbeispiele

Bis jetzt wurde die Architektur und die Implementierung der Schnittstelle erklärt. Dieser Abschnitt zeigt zwei einfache Beispiele für die Nutzung der Schnittstelle, um sie aus der Sicht des Nutzers darzustellen.

2.4.1 Date

Dieses Beispiel zeigt das Initialisieren, Lesen und Serialisieren anhand eines einfachen Typen.

Dafür wird wieder das einfache *Date*-Beispiel aus dem Listing 1.1 als Grundlage genommen. Außerdem werden die beiden generierten Klassen *Date.java* (Listing 2.1) und *StateClass.java* (Listing 2.2) verwendet, die mit Hilfe des Codegenerators erzeugt wurden.

Listing 2.3 zeigt die Nutzung der Schnittstelle. Zuerst benötigt man eine Instanz der Zustandsklasse, die im Konstruktor vorerst leere Pools erzeugt. Diese leeren Pools können als initiale Container für Daten dienen. Optional kann der Nutzer eine Datei auslesen. In diesem Fall werden neue Pools mit den in der Datei vorhandenen Daten gefüllt.

Listing 2.4 Node als SKilL-Typ

```
Node {
   i8 id;
   /** color was added later */
   string color;
}
```

Listing 2.5 Beispiel: Neue Felddaten hinzufügen

```
// package and imports
...

public class SkillAPIUsageExample2 {

   public static void main(String[] args) {

      SkillState state = new SkillState();
      String fileName = "/path/to/nodes.sf";

      // file only contains data for the field id
      state.readFile(fileName);
      Pool<Node> nodes = state.getNode();

      for (Node node : nodes) {
         System.out.println(node.getId());
         node.setColor(ColorTool.randomColor());
      }

      // only new field data will be saved
      state.saveToFile(fileName);
   }
}
```

Für jeden Typen bietet die Zustandsklasse einen separaten Pool an. Für diese existiert jeweils eine Getter-Methode. Da die Pool-Klasse das *Iterable*-Interface aus Java implementiert, kann man auf einen Iterator zugreifen oder wie in dem Listing 2.3 eine vereinfachte For-Schleife einsetzen, um die Daten aufzurufen.

Auf die Daten zugreifen ist nur ein Anwendungsfall. Ein anderer wäre, neue Daten zu erzeugen und zu serialisieren. Dafür reicht es, wie im Beispiel zu sehen, neue Typklassen-Objekte in die Pools hinzuzufügen und mit Hilfe der Zustandsklasse diese Daten in eine Datei zu schreiben. Abhängig von der Datei wird von der Schnittstelle ohne zusätzliche Parameter entschieden, ob *append* oder *write* ausgeführt wird.

Für *append* wird nicht garantiert, dass die erzeugte Datei fehlerfrei ist, da kein Locking der Datei über die Dauer der Arbeit mit Daten unterstützt wird. Sollte ein anderes Tool zwischenzeitlich die Datei um neue Daten ergänzen, kann es passieren, dass Indizes als Referenzen auf mehrere unterschiedliche Objekte zeigen oder Strings mehrmals vorkommen.

2.4.2 Neue Felddaten hinzufügen

Während das erste Beispiel einen einfachen Anwendungsfall zeigt, ist dieses Beispiel etwas anders. Hier werden neue Felddaten hinzugefügt und serialisiert. Auf diese einfache Weise kann man Typen um weitere Felder erweitern.

Listing 2.4 zeigt den Typen *Node*, der ursprünglich nur das Feld *id* enthielt und später um das Feld *color* ergänzt wurde. Anzunehmen sei, dass bereits eine Datei existiert, die *id*-Felddaten aber keine Daten für das Feld *color* beinhaltet.

Listing 2.5 zeigt beispielhaften Quellcode, in dem die alten *Node*-Daten aus einer Datei ausgelesen und dann um Daten für das Feld *color* ergänzt werden. Es werden keine neuen Instanzen erzeugt. Beim Serialisieren wird die *append*-Operation ausgeführt, die einen entsprechenden String- und Typblock an die Datei anhängt. Wobei der Typblock keine neuen Objekte anlegt und nur Felddaten für das *color*-Feld enthält. Bei der nächsten Deserialisierung werden Daten für alle Felder ausgelesen.

3 Nutzbarkeitsevaluation der Schnittstelle

Dieser Kapitel geht auf die Nutzbarkeit der Schnittstelle ein. Hierbei werden die Benutzerfreundlichkeit, Einfachheit, Integration in Java usw. berücksichtigt. Zuerst wird ausführlich dargestellt, was wichtig ist, dann wird anhand der Beispiele gezeigt, wie die Implementierung der SKilL-Schnittstelle für Java die relevanten Punkte umsetzt. Dies kann, in diesem subjektiven Thema, als eine Form der Evaluation angesehen werden.

3.1 Bedeutung einer guten Schnittstelle

Der Entwurf bzw. das Aussehen einer Schnittstelle ist ein sehr wichtiger Punkt, den viele Entwickler unterschätzen. Wenn ein Nutzer sich für eine Schnittstelle entscheidet, muss er viel Zeit investieren, um die API kennenzulernen, sodass er sie produktiv einsetzen kann. Das ist vor allem im praktischen Umfeld von Bedeutung, in dem Zeit oft den Kosten gleichgesetzt wird. Außerdem können zeitliche Begrenzungen eine große Rolle spielen, z.B. Meilensteine oder Auslieferungstermine.

Eine Schnittstelle sollte daher schnell zu erlernen und einfach in der Nutzung sein. Ein Nutzer soll sie möglichst ohne viel Aufwand verstehen und verwenden. Eine gut entworfene Schnittstelle erspart außerdem den Entwicklern eine Menge an Support- und Wartungsaufwand. Vor allem im Bezug auf die Wartung kann viel falsch umgesetzt werden. Öffentliche Schnittstellen, wie etwa Schnittstellen für offene Serialisierungsformate, sind "für die Ewigkeit". Ein guter Entwurf ist daher schon von Anfang an wichtig, um die ersten Nutzer nicht zu überfordern oder auch ihnen später den Aufwand zu ersparen, wenn an der Schnittstelle sich etwas ändert.

Einfache Schnittstellen sind außerdem Werbung für die Serialisierungsformate oder generell für alles, was sie dem Nutzer an Möglichkeiten zur Verfügung stellen. Je einfacher eine Schnittstelle, desto wahrscheinlicher ist es, dass ein Nutzer sich für sie entscheidet. Dies könnte vor allem bei neuen Formaten wie SKilL relevant sein, um die ersten "Kunden" für sich zu gewinnen, die dann das Format sowie die Schnittstelle nach den ersten Erfahrungen eventuell an weitere Entwickler empfehlen werden.

3.2 Eigenschaften einer guten Schnittstelle

Eine gute Schnittstelle assoziiert man meistens mit solchen allgemeinen Begriffen wie Benutzerfreundlichkeit oder Einfachheit. Das ist für die Umsetzung zu generell und jeder Entwickler muss zuerst die Eigenschaften präzisieren oder ergänzen.

Nutzbarkeitsevaluation ist außerdem von subjektiver Natur und kann nur mit Hilfe der Nutzer und Entwickler erfasst werden. Es gibt zwar Stolpersteine, die die Wahrnehmung der Nutzer und Entwickler im Bezug auf die Einfachheit, Benutzerfreundlichkeit, Intuition usw. beeinflussen, z.B. vorhandene Kenntnisse oder gesammelte Erfahrungen, dennoch kann man einiges bewerten und zum Vergleich heranziehen.

Als Grundlage für die folgende Liste an Eigenschaften dienten in der Regel Erfahrungen und Empfehlungen der Entwickler, die entweder in Fallstudien, wissenschaftlichen Arbeiten oder Präsentationen gefunden wurden.

- Schnell erlernbar
- Einfach nutzbar (auch ohne Dokumentation)
- Nur mit Schwierigkeiten falsch verwendbar
- Gewährleistung eines einfachen und lesbaren Nutzer-Codes
- Informativ
- Erfüllung der Anforderungen bzw. Vollständigkeit
- Berücksichtigung der Zielgruppe
- Robust
- Einfach erweiterbar
- Konsistenz
- Verwendung bekannter Konzepte, z.B. aus anderen Schnittstellen
- Integration in die Technologie (SKilL)

3.3 Erlernbarkeit

Erlernbarkeit ist ein Begriff, der oft in wissenschaftlichen Arbeiten oder Artikeln vorkommt, z.B. [RD10], [SM]. Dabei ist es ein sehr allgemeiner Begriff, der eventuell eine weitere Präzisierung benötigt. Wichtige Punkte bei der Erlernbarkeit sind unter anderem Dokumentation sowie deren Formatierung und Präsentation, Beispiele, Komplexität usw.

Die folgende Evaluation stützt sich besonders auf die Studie "A field study of API learning obstacles" [RD10], die im Unternehmen Microsoft durchgeführt wurde. Der Schwerpunkt der Studie war die Dokumentation der Schnittstelle. Es gab außerdem eine weitere Analyse zu dem Thema, die [RD10]

vorausging und es ergänzte. Für den Artikel "What Makes APIs Hard to Learn? Answers from Developers" [Rob09] wurde eine Umfrage durchgeführt, um bestimmte Punkte, die sich als besondere Stolpersteine erwiesen haben, herauszufiltern und darzustellen. Dafür wurde eine Gruppe an Entwicklern mit unterschiedlicher Erfahrung bei Microsoft befragt und die Ergebnisse kategorisiert.

Für [Rob09] wurden insgesamt 13 Fragen überlegt, wobei die Fragen offener Natur waren und zudem sich teilweise auf die Erfahrung des Teilnehmers bezogen. Insgesamt haben 80 Entwickler mit einer breiten Schicht an Kenntnissen und Erfahrungen an der Umfrage teilgenommen. Ein Beispiel für Erfahrungen: Die Teilnehmer haben in der Studie angegeben, dass sie sich mit 54 Schnittstellen in ihrer Karriere als Softwareentwickler befasst haben. Der Wert ist vermutlich ein Durchschnitt aus dem Antworten von allen Entwicklern, es gibt aber keine genaue Erklärung dazu.

78% der Entwickler meinten, dass sie zum Erlernen einer Schnittstelle die Dokumentation lesen. 55% schauen sich Code- und Nutzungsbeispiele an. 34% experimentieren mit den Schnittstellen, 30% lesen weitere Hilfestellungen, z.B. Artikeln im Internet, und 28% fragen Kollegen. Es gab außerdem eine Menge an weiteren Punkten, wie das Lesen von Büchern oder das Tracing des Codes mit einem Debugger.

Aus den Antworten ergab sich zudem, dass für das Erlernen der Entwurf einer Schnittstelle von Bedeutung ist. Testbarkeit und Debuggen spielen zwar keine große Rolle, wurden aber von einigen Teilnehmern als erwähnenswert betrachtet.

An die Umfrage angeschlossen gab es Interviews mit 12 Entwicklern, um das Thema nicht nur quantitativ sondern auch qualitativ zu erfassen. Dabei wurden weitere interessante Punkte aufgestellt. Ein Entwickler meinte, dass er manchmal das Problem hat, die richtige Funktion aus vielen ähnlichen Alternativen auszuwählen. Im Bezug auf den Entwurf waren einige Entwickler von den Schnittstellen enttäuscht, da diese keine gute Architektur vorzuweisen hatten. Wie die Schnittstelle genau implementiert wurde, war aber ihnen größtenteils egal. Dabei ging es ihnen mehr um das Verständnis zum Geschehen im Hintergrund... "Was macht die Funktion XYZ?" oder "Welche Funktion ist die richtige für meine Zwecke?" An dieser Stelle spielt auch die Effizienz der Schnittstelle eine Rolle. Nutzer einer API wollen nichts verwenden, was vielleicht mit alternativen Wegen effizienter oder eleganter funktionieren könnte.

Code-Beispiele waren ein weiteres wichtiges Thema. Die Beispiele wurden in drei Kategorien aufgeteilt: Snippets, Tutorials und Code in Anwendungen. Snippets sind kurze Beispiele, nicht länger als 30 Zeilen. Tutorials sind längere Texte, die mehrere Beispiele beinhalten können. Code in Anwendungen sind meistens Ausschnitte aus realisierten Projekten, z.B. Open-Source-Projekten. Relevant bei den Beispielen sind Empfehlungen ("best practices"), Aktualität und Design-Entscheidungen. Verwendete Funktionen sollen außerdem Sinn im Zusammenhang ergeben, z.B. nach dem Lesen einer Datei werden gelesene Daten erwartet. Der Nutzer sollte keine weiteren Funktionsaufrufe zwischendurch oder danach machen. Nicht interessante Beispiele sind eher schädlich als nützlich, weil Nutzer den Eindruck bekommen können, dass die Schnittstelle funktional nicht das leistet, was sie verspricht.

Überraschendes Verhalten ist oft ein Problem für Nutzer, z.B. wenn eine Funktion nicht das zurückgibt oder macht, was der Nutzer erwartet. Das kann zu fehlerhafter Anwendung der Schnittstelle führen. Hier wird *information hiding* eher als negativ empfunden, obwohl man intuitiv das Gegenteil vermutet. Als Beispiel kann man *null* als Rückgabewert erwähnen. Falls Collections wie Arrays oder Listen

zurückgegeben werden, sollen sie im schlimmsten Fall leer sein, z.B. ein *null*-Pool als Rückgabewert in der Zustandsklasse ist kontraproduktiv. So etwas erfordert eine Ausnahmebehandlung und verkompliziert den Code des Schnittstellen-Nutzers.

Die Fallstudie "A field study of API learning obstacles" [RD10] geht, wie erwähnt, weiter auf das Thema ein. Dafür wurden über 300 Microsoft-Mitarbeiter befragt, wobei die Fachstudie sich eher auf die Dokumentation bezogen hat, da dies ähnlich wie in der ersten Studie [Rob09] die am meisten erwähnte Antwort war. Zusammengefasst hat diese zweite Analyse nur das bestätigt, was die erste aussagte.

Manche Punkte wurden ausführlicher erläutert, wie etwa Code-Beispiele, die z.B. nicht zu kurz aber auch nicht zu lang sein dürften. Einzeilige Beispiele sind in der Regel nutzlos, mehrere Zeilen mit Kontext helfen im Lernprozess. Viele gaben außerdem zu, dass sie Beispiele kopieren.

Ergänzend zu den Studien gibt es weitere Merkmale einer Schnittstelle, die wichtig für die Erlernbarkeit sind, z.B. ein leichter Einstieg in die Nutzung. Damit sind nicht Beispiele, Dokumentation oder Ähnliches gemeint. Ein Nutzer kann eine API schneller erlernen, wenn er z.B. nicht mit vielen Methodenaufrufen oder sonstigen Instruktionen direkt am Anfang überfordert wird.

3.3.1 Umsetzung in der SKilL-Schnittstelle für Java

Die SKilL-Schnittstelle für Java ist relativ klein. Es gibt nur wenige Funktionen, die von einem Nutzer verwendet werden können. Alle von ihnen befinden sich entweder in den Typklassen, in der Zustandsklasse oder in der Pool-Klasse. Die Typklassen enthalten Getter und Setter, die in der Regel keine Kommentare enthalten, da dies technisch nicht immer realisierbar ist. Der Codegenerator kann nicht die Felder beschreiben, zumindest solange der Nutzer in den SKilL-Beschreibungsdateien keine Kommentare setzt. Getter und Setter sind außerdem selbsterklärend und benötigen normalerweise keine zusätzliche Erläuterung. Sie sind Teil des Benutzercodes, wenn auch generiert, bei dem der Nutzer am besten weiß, wofür die passenden Felder da sind. Die Zustandsklasse und die Pool-Klassen sind nach den Javadoc-Regeln kommentiert.

Beispiele existieren nicht direkt, allerdings gibt es eine Reihe von Testfällen, die die meisten Funktionen der Schnittstelle testen. Solche Testfälle können als Code-Snippets verwendet werden.

Um den Nutzer nicht zu überfordern und die Schnittstelle überschaubar zu halten, wurde absichtlich *information hiding* betrieben. Die Schnittstelle ist in zwei Teile gespalten. Der generische Teil *Binding* versteckt die Funktionalität, die für den Nutzer wenig interessant ist. Das ist vor allem wegen der Komplexität und der Integration in Java der Fall. Es werden allerdings passende Exceptions ausgelöst, falls während der Serialisierung etwas nicht stimmt, sodass im Endeffekt nicht alles versteckt wird und der Nutzer immer noch in einem gewissen Rahmen debuggen kann.

Unerwartetes Verhalten kann zwar jederzeit passieren, da die Schnittstelle von dem System, der Java-API und der JVM abhängig ist, z.B. beim Schreiben der Dateien. Allerdings sollten in der Schnittstelle selbst keine überraschenden Resultate erzeugt werden. Wenn eine Datei nicht gelesen werden kann, wird eine Exception ausgelöst und der Nutzer kann entscheiden, was in der Software passiert. Korrupte oder fehlerhafte Daten können dabei nicht entstehen.

3.4 Einfachheit

Dokumentation ist nicht alles. Wenn die Schnittstelle umständlich in der Nutzung ist, helfen selbst die besten Beispiele und API-Dokumentationen nicht, die sowieso meistens am Anfang in der Nutzung angeschaut werden. Ab einem bestimmten Zeitpunkt wird die API endlich benutzt.

Eine Schnittstelle sollte, wenn möglich, so einfach und verständlich sein, dass ein Nutzer nach einer kurzen Einarbeitung oder vielleicht auch sofort produktiv ist. Dieser Abschnitt behandelt diesen Aspekt der Schnittstelle.

Wie auch bei der Erlernbarkeit handelt es sich um einen sehr allgemeinen Begriff, der viel bedeuten kann. Im Bezug auf Schnittstellen kann man mehrere Teileigenschaften finden, die eine API besitzen muss, um für den Nutzer als einfach zu wirken.

3.4.1 Bezeichner

Ein Punkt, der oft erwähnt wird, ist die Wahl der passenden Bezeichner, z.B. für Parameter, Methoden, Klassen, Pakete usw. Eine Schnittstelle oder Teile davon müssen selbsterklärend sein. Dafür ist es ungünstig, wenn die verwendeten Bezeichner Abkürzungen, Akronyme, Sonderzeichen, kryptische Namen etc. sind. Abkürzungen sind nur dann als Bezeichner in Ordnung, wenn sie bekannte Wörter und somit allgemein verständlich sind, z.B. XML, HTML usw.

Der Name einer Methode sollte genau das beschreiben, was die Methode macht. Der Name einer Klasse soll genau das sein, was die Klasse repräsentiert. Die Bezeichner sollen außerdem konsistent sein, unterschiedliche Bezeichner für ähnliche Methoden oder gleiche Parameter sind verwirrend.

In der Studie "Influencing Factors on the Usability of API Classes and Methods" [SK12] wurde unter anderem untersucht, in wie weit die Namen die Benutzbarkeit beeinflussen. Es stellte sich heraus, dass die Hälfte der API-Nutzer in Entwicklungsumgebungen wie Eclipse den Anfang eines möglichen Bezeichners eingeben und dann die Vorschläge anschauen. Ein Beispiel wäre die Hinzufügen-Operation. Für diese Aktion gaben die Nutzer *add* in die Entwicklungsumgebung ein und erwarteten eine entsprechende Liste an Methoden zur Auswahl. Zumindest für solche Nutzer-Gruppen ist es empfehlenswert, beschreibende Bezeichner zu verwenden, die leicht gefunden werden können.

Die zweite Hälfte der Nutzer in der Studie schaute alle zur Verfügung stehenden Klassen und Methoden an und versuchte passend auszuwählen. Auch für diese Gruppe ist es ratsam gute Bezeichner zu wählen, wenn auch hier die korrekte Benennung nicht kritisch wie für die erste Gruppe ist. Auf die zweite Weise wurden Methoden und Klassen übrigens langsamer als mit der ersten Möglichkeit gefunden.

Die SKilL-Schnittstelle für Java versucht das umzusetzen. Ein "Pool" repräsentiert einen *Storage Pool* in SKilL, der Begriff ist zudem allgemein und kann als Container für Daten verstanden werden. Auch die Methoden-Bezeichner versuchen selbsterklärend zu sein, z.B. die Methoden *Pool.add(T element)*, *SkillState.writeToFile(String fileName)* oder *SkillState.readFile(String fileName)* können ohne Dokumentation benutzt werden. Der Parametername *fileName* ist konsistent über die beiden letzten Methoden.

Eine gute Praxis ist die Verwendung von Namenskonventionen, um die Bezeichner in der Schnittstelle, über viele Schnittstellen hinweg oder generell Software konsistent zu halten. Ein Wort sollte nicht mehrere Bedeutungen haben. Eine weitere Empfehlung wäre, die Bezeichner so zu wählen, dass der Quellcode wie ein gewöhnlicher Text lesbar wird.

Die Bezeichner sollen außerdem symmetrisch sein. Für eine *get*-Methode sollte logischerweise eine *set*-Methode, für *add* eine *remove*-Methode usw. existieren. Zumindest wenn es sinnvoll ist.

3.4.2 Bezug auf andere Schnittstellen

Nicht jeder Entwickler ist ein Anfänger, der bei null anfängt. Viele Entwickler haben bereits Erfahrungen mit anderen Schnittstellen und versuchen, wenn sie eine neue API verwenden, das vorhandene Wissen in irgendeine Weise einzubeziehen. SKilL ist neu, hier ist eigentlich jeder Nutzer ein Anfänger. Dennoch kann der Entwickler der SKilL-Schnittstelle sich an anderen und ähnlichen Schnittstellen orientieren. Daher sollte man als Schnittstellenentwickler versuchen, eine konsistente Nutzung über mehrere unterschiedliche Schnittstellen hinweg zu erreichen.

Eine zu der SKilL-Schnittstelle für Java ähnliche API ist JAXB [jax14]. JAXB ist vor allem dadurch beliebt geworden, weil es einfach in der Nutzung ist. Es kann beliebige Typen annehmen, dann einen Kontext erzeugen, der serialisiert werden kann. Der Code dafür ist in der Regel nur ein paar Zeilen lang. Man braucht eine *JAXBContext*-Instanz, welche benutzt werden kann, um Daten aus einer Datei zu lesen (Unmarshalling) oder sie in eine Datei zu serialisieren (Marshalling).

Listing 3.1 zeigt die zu Listing 2.3 funktional äquivalente Implementierung der Serialisierung. Dafür wurden mit Hilfe von XSD (XML Schema Definition) zwei Typen definiert, Dates und Date. Dates repräsentiert das Wurzelelement in einer XML-Datei, die eine Liste von Date-Objekten enthält. Man braucht somit zwei Typen statt nur einem wie in dem SKilL-Beispiel 2.3. Die Dates-Klasse kann verwendet werden, um einen JAXB-Kontext zu generieren, der die Struktur einer XML-Datei festgelegt. Anschließend kann eine Dates-Instanz als Container, ähnlich einem SKilL-Pool, für Date-Objekte verwendet werden. Diese Instanz kann zum Schluss als Parameter der *Marshall.marshall(Dates dates, File file)* zum Serialisieren übergeben werden.

Man sieht an dem Beispiel eine Ähnlichkeit der beiden Schnittstellen JAXB und SKilL. Eine *SkillState*-Instanz entspricht einer Mischung aus *JAXBContext*-, *Unmarshaller*-, *Marshaller*-Instanzen und dem Wurzelelement. Die Typklassen sind auch sehr ähnlich. Sie entsprechen POJOs und enthalten jeweils neben einer Auflistung der Feldern die passenden Getter- und Setter-Methoden.

Jemand, der bereits Erfahrung mit JAXB hat, muss sich nicht besonders gründlich in die SKilL-Schnittstelle für Java einarbeiten. Beide funktionieren ähnlich und auch das Verhalten ist vergleichbar. Das ist allerdings nicht immer der Fall. Bei komplizierten Typen, die Referenzen auf andere Typen enthalten kann die JAXB-Implementierung einfacher sein, da Instanzen von Typeklassen automatisch mit der *JAXBContext*-Klasse aufgelöst werden. Die SKilL-Schnittstelle hat diese Funktionalität nicht, der Nutzer muss selbst die Objekte in passende Pools einfügen. Die Funktionalität kann aber nachgerüstet werden.

Listing 3.1 JAXB-Beispiel, funktional äquivalente Implementierung zu Listing 2.3

```
package com.company.special.project;

import java.io.File;
import java.io.IOException;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

import de.ust.jaxb.generated.date.Dates;
import de.ust.jaxb.generated.date.Date;

public class JAXBUsageExample {

   public static void main(String[] args) throws JAXBException, IOException {

      JAXBContext context = JAXBContext.newInstance(Dates.class);
      File xmlFile = new File("path/to/dates.xml");

      Unmarshaller unmarshaller = context.createUnmarshaller();
      Dates datesElement = (Dates) unmarshaller.unmarshal(xmlFile);

      List<Date> dates = datesElement.getDate();

      for (Date date : dates) {
         System.out.println(date.getDate());
      }

      Date newDate = new Date();
      newDate.setDate(3);
      dates.add(newDate);

      Marshaller narshaller = context.createMarshaller();
      marshaller.marshal(datesElement, xmlFile);
   }
}
```

3.4.3 Anzahl der Pakete, Klassen, Methoden und Parameter

Die bereits erwähnte Studie “Influencing Factors on the Usability of API Classes and Methods“ [SK12] untersuchte außerdem, ob die Anzahl der Pakete, Klassen und Methoden eine Rolle in der Nutzung spielen.

Dafür wurden zwei Schnittstellen mit der gleichen Funktionalität entwickelt. Die erste Schnittstelle (A) hatte nur ein Paket mit 33 Klassen. Die wichtigste Klasse, die *ZipFile* genannt wurde, hatte 33 Methoden mit 50 Methoden-Überladungen. Die zweite Schnittstelle (B) hatte mehrere Pakete, mit acht Klassen im obersten Paket und weiteren Klassen in Unterpaketen. Die Klasse *ZipFile* hatte hier nur zehn Methoden und 20 Überladungen.

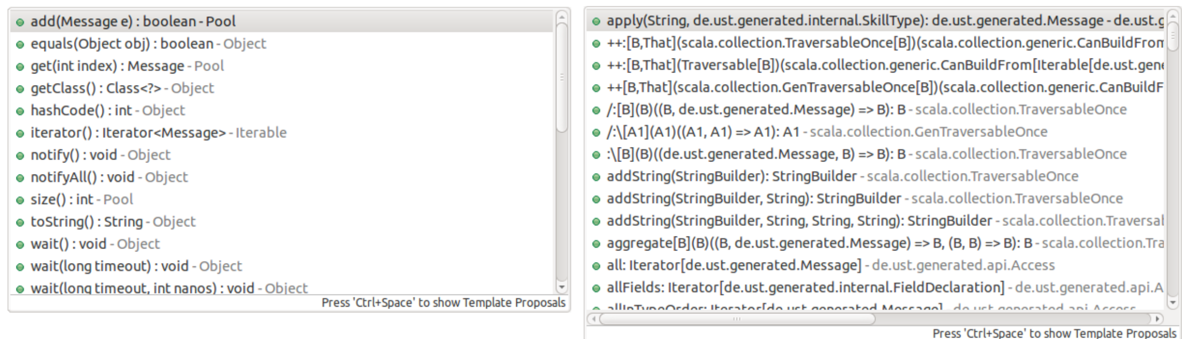

Abbildung 3.1: Der Screenshot zeigt Methoden bzw. Funktionen-Vorschläge aus Eclipse für die SKilL-Schnittstellen in Java (links) und Scala (rechts). Aufgerufen werden alle Methoden aus dem *Message*-Pool (siehe Listing 1.2). Die Anzahl der Funktionen in der Scala-Schnittstelle ist ca. 12 Mal höher als in der Schnittstelle für Java.

Für die Studie wurden 20 Entwickler mit zwei bis zehn Jahren Erfahrung ausgewählt, die acht Aufgaben erledigen mussten. Die Entwickler wurden in zwei Gruppen geteilt, die ersten Gruppe (A) musste mit der ersten Schnittstelle die Aufgaben erledigen, die zweite (B) mit der zweiten.

Es stellte sich heraus, dass Gruppe B im Durchschnitt sowie im Median teilweise deutlich schneller in der Aufgabenerledigung war. Wichtiger Faktor war dabei die Suche nach Methoden und Klassen. Für die Suche nach der Klasse *ZipFile* benötigte Gruppe A 45 Sekunden, Gruppe B benötigte nur 26. Beide sind Medianwerte. Es wurde außerdem ein Lerneffekt erzielt, in späteren Aufgaben war die Suche um 75% schneller. Auch bei Methoden gab es ähnliche Beobachtungen, obwohl hier die Anzahl der Methoden eine weniger wichtige Rolle spielte. Die Nutzer gaben meistens die Namen der Methoden in die Suche ein, um die Methodenliste zu filtern. Dadurch konnten sie schnell die Anzahl der relevanten Methoden verkleinern.

Abbildung 3.1 zeigt Eclipse-Vorschläge, wenn der Nutzer bei der Suche nach Methoden bzw. Funktionen in den SKilL-Schnittstellen für Java und Scala alles auflisten lässt. Dies könnte ein Hindernis darstellen, da wegen der Menge an angebotenen Funktionen die Suche verlangsamt wird.

Um zu messen, wie sich die Anzahl der Parameter auswirkt, wurden für die Studie drei Typen an Methoden definiert, jeweils mit einem, zwei und vier Parametern. Danach wurde gemessen, wie lange die Nutzer für die Verwendung der Methoden brauchen. Nicht überraschend waren die Methoden mit nur einem Parametern am schnellsten in der Nutzung, dafür brauchten die Entwickler nur 7 Sekunden (Median). Für Methoden mit zwei Parametern benötigten sie 12 Sekunden, und mit vier Parametern 28 Sekunden. Wobei das Ergebnis womöglich verfälscht ist. Die Entwickler tendierten zur Nutzung von Methoden mit nur einem Parameter. Je mehr Parameter eine Methode hatte, desto weniger wurde sie benutzt. Methoden mit vier Parametern wurden nur zwei Mal aufgerufen, Methoden mit einem Parameter 48 Mal. Auch hier sieht man, dass wenig Parameter hilfreich sind.

Wenig Parameter ist nicht immer eine Lösung. Deswegen wurde untersucht, in wie weit Methodenüberladungen die Verwendung einer Schnittstelle beeinflussen. Das Ergebnis war, dass Nutzer einfache

Methoden bevorzugen, die oft überladen werden. Methoden mit wenigen Überladungen aber dafür mit mehr Parametern wurden weniger benutzt, weil manche Parameter als unnötig erschienen.

Für eine Schnittstelle muss man daher abwägen, ob man Methoden mit vielen Überladungen aber mit weniger Parametern möchte, oder ob man weniger Methoden zur Verfügung stellen soll, die dafür mehr Parameter erfordern. Alternativ können Methoden gespalten werden, sodass mehrere Methoden aufgerufen werden müssen, die, einzeln betrachtet, weniger Parameter erfordern.

Mengen spielen in der SKilL-Schnittstelle für Java nur eine untergeordnete Rolle. Es gibt nur drei Typen vom Klassen, die verwendet werden. Dazu gehören die Typklassen, die Zustandsklasse sowie die *Pool*-Klasse. Die Typklassen liegen außerdem außerhalb des Zuständigkeitsbereichs des Schnittstellenentwicklers, er kann also die Methoden dort nicht beeinflussen. Somit beträgt die Anzahl der relevanten Klassen 2. Das ist für die Einfachheit vernachlässigbar.

Anders ist es bei Methoden. Die Zustandsklasse kann optimiert werden. Dafür wurde für die Java-Schnittstelle die Funktionalität von der eigentlichen Zustandsklasse in die abstrakte ausgelagert, sodass der Nutzer nur die Methoden sieht, die er wirklich verwenden kann. Dadurch reduzierte sich die Anzahl der Methoden auf 2 (zum Lesen und Schreiben) + n (mit n = Anzahl der Typen). Die *Pool*-Klasse enthält nur vier deklarierte öffentliche Methoden. Auch das ist vernachlässigbar und macht die Nutzung der Pools einfach und schnell.

Anzumerken sei auch noch die Sichtbarkeit von Methoden. Um die Suche zu erleichtern, sollen alle irrelevanten Methoden privat sein. Alternativ kann man die Sichtbarkeit auf *default* oder *protected* setzen und dann zusätzliche Klassen im gleichen Paket implementieren, die wie Proxys für diese Methoden agieren. Diese zusätzlichen Klassen können optional benutzt werden, um auf mehr Funktionen zuzugreifen. In der SKilL-Schnittstelle gibt im Pool-Paket die Klasse *PoolSerializationHelper*, welche ein Proxy für Pools dient, um nichtöffentliche Methoden zur Serialisierungszeit aufzurufen.

3.4.4 Fabrikmethoden

Fabrikmethoden sind Funktionen, mit den der Nutzer Instanzen einer Klasse erzeugen kann. Das ist ein relativ bekanntes Muster in der Softwareentwicklung und wird daher oft verwendet, z.B. in der SKilL-Schnittstelle für Scala zum Instanziieren der Typklassen mit Hilfe der Zustandsklasse. In der Schnittstelle für Java werden sie allerdings aus einem Grund nicht eingesetzt. Sie sind manchmal kontraproduktiv, wie die Studie "The Factory Pattern in API Design: A Usability Evaluation" [ESM] das gezeigt hat. Es gibt zwar noch weitere Gründe, aber sie sind anderer Natur (z.B. Konsistenz).

In der Studie mussten 12 Java-Entwickler (mit unterschiedlicher Erfahrung) fünf Aufgaben erledigen. Alle Aufgaben konnten mit Fabrikmethoden erledigt werden, oder erforderten sie in der Nutzung. In der ersten Aufgabe mussten die Entwickler eine Klasse implementieren, die zwei Typen an E-Mails repräsentierte. Der Typ wurde anhand eines Parameters festgelegt. Niemand von den Entwicklern ist auf die Idee gekommen, dafür eine Fabrikmethode zu nutzen. Stattdessen wurden entweder unterschiedliche Konstruktoren, Parameter oder Setter verwendet.

Für die zweite Aufgabe gab es eine API, die E-Mail-Instanzen mit einer Fabrikmethode erzeugte. Es wurde beobachtet, wie die Entwickler darauf reagieren. Die meisten versuchten zuerst mit einem Konstruktor eine Instanz zu erzeugen, obwohl ein Konstruktor in der Dokumentation gar nicht

erwähnt wurde. Manche versuchten sogar eine Unterklasse zu implementieren. Schließlich haben alle die Fabrikmethode gefunden und benutzt.

In der dritten Aufgabe mussten die Entwickler Instanzen von zwei Klassen benutzen. Eine Klasse konnte mit einem Konstruktor instanziiert werden (genannt "Flarn"), die andere mit einer Fabrikklasse ("Squark"). Für Squark-Instanziierung brauchten die Entwickler 7 Minuten und 10 Sekunden (Median-Wert), für die der Flarn-Klasse nur eine Minute und 20 Sekunden.

Die fünfte Aufgabe ähnelte der dritten und entsprach einem reellen Fall. Dafür mussten Instanzen der Klassen *SSLSocket* und *MulticastSocket* aus der Java-API erzeugt werden. Für die erste Klasse kann nur eine Fabrikklasse verwendet werden, für die zweite gibt es mehrere Konstruktoren. Das Besondere an dieser Aufgabe war, dass die Entwickler die offizielle Java-API-Dokumentation lesen mussten, um zu sehen, wie man die Instanzen der beiden Klassen erzeugt. Für die *SSLSocket*-Instanziierung brauchten sie über 20 Minuten, für *MulticastSocket* nur 9 Minuten und 31 Sekunden.

Fabrikmethoden sind gut, wenn beim Erzeugen von Instanzen z.B. Abhängigkeiten aufgelöst werden. So etwas dem Nutzer zu überlassen, wäre weniger benutzerfreundlich als der Fabrikmethoden-Ansatz selbst. Daher muss man abwägen, in welchen Situationen sich diese Art der Instanziierung lohnt. Wie die Studie zeigt, sind sie in manchen Fällen kontraproduktiv.

In der SKilL-Schnittstelle für Java wurde darauf verzichtet, um z.B. die Nutzung der Pools mit *Collection*-Implementierungen konsistent zu halten. Pools agieren somit nicht als Fabriken für Objekte eines Typen wie in der Schnittstelle für Scala. In der Zustandsklasse werden zwar die Pools jedes Mal erzeugt, wenn die Klasse instanziiert oder wenn eine Datei gelesen wird, aber das entspricht nicht dem klassischen Fabrikmethoden-Muster, da eine Zustandsklasse, die einen Zustand repräsentiert, auch als Container für Pools bzw. Daten allgemein verstanden werden kann.

Die Bevorzugung der Konstruktoren den Fabrikmethoden wird nicht von allen Entwicklern empfohlen. Im Buch "Practical API Design. Confessions of a Java Framework Architect" [Tul08] wird genau das Gegenteil behauptet. Allerdings ist das Buch älter als die Studie, somit könnte die Behauptung nicht mehr besonders relevant sein.

Im Buch werden Vorteile von Fabrikmethoden aufgezählt. Dazu gehören z.B. Instanziierung von Unterklassen aus Elternklassen heraus, bessere Synchronisation etc. Außerdem erzeugen Konstruktoren immer neue Instanzen, während Fabrikmethoden auch bereits vorhandene Instanzen zurückgeben können, wie z.B. im Singleton-Entwurfsmuster.

3.4.5 Methodenplatzierung

Methodenplatzierung ist ein Thema, das nicht oft in Empfehlungen zum Schnittstellenentwurf vorkommt. Dabei spielt es eine wichtige Rolle, um die Schnittstellen benutzerfreundlicher zu machen. Diesen Punkt könnte man genau so im Abschnitt 3.3 unterbringen, da es auch in der Erlernbarkeit einer API von Bedeutung ist.

In diesem Thema geht es um die korrekte Platzierung der Methoden, also der Unterbringung in passende Klassen. Um es für die SKilL-Schnittstelle relevant zu machen, könnte man sich überlegen, ob solche Funktionen wie *SkillState.writeToFile(String fileName)* nicht in einer aus der Benutzersicht besseren Klasse bzw. Methode, wie z.B. *SkillWriter.write(SkillState state, File target)*, untergebracht wären. JAXB verfolgt genau diesen Ansatz mit den Klassen *Marshaller* und *Unmarshaller*, die zum Serialisieren bzw. Lesen verwendet werden.

Die Studie "The Implications of Method Placement on API Learnability" [SM] untersucht dieses Thema. Dort wurden zunächst drei Hypothesen aufgestellt: 1. Entwickler suchen zuerst benötigte Methoden in den Klassen, mit denen sie gerade arbeiten. 2. Sie suchen nach weiteren Klassen über die Verweise in der gegenwärtigen Klasse. 3. Angenommen es gibt eine Anfangsklasse. Die Entwickler sind deutlich schneller bei der Suche und Implementierung, wenn die Anfangsklasse Methoden oder Verweise auf andere Klassen (wie Hilfsklassen) enthält. Anschließend wurden die drei Hypothesen untersucht.

Um die Hypothesen zu testen, wurden drei Schnittstellen entwickelt. Zwei Schnittstellen basierten auf echten APIs. Die dritte Schnittstelle war eine komplette Neuentwicklung, um zu sehen, wie Nutzer sich mit einer neuen API zurechtfinden. Anschließend mussten zehn Entwickler drei Aufgaben erledigen.

Für jede der Aufgaben konnte jeweils die passende Schnittstelle benutzt werden, wobei jede Schnittstelle zwei Varianten hatte. In der ersten Variante (A) enthielte eine Hilfsklasse die benötigte Methode, in der zweiten (B) war sie in der Anfangsklasse. Die Entwickler wurden in zwei Gruppen für jede Variante geteilt.

Ergebnisse der Studie zusammengefasst: Mit der zweiten Variante (B) waren die Entwickler, je nach Aufgabe, 2,4 bis 11,2 Mal schneller in der Implementierung als mit der ersten. Ein deutlicher Unterschied in der Geschwindigkeit, obwohl der Unterschied in der API-Architektur sehr klein war. Dafür gibt es mehrere Begründungen: 1. Die Entwickler denken, dass sie die falsche Anfangsklasse benutzen, wenn sie die gesuchte Methode nicht finden. 2. Manche realisieren nicht, dass eine zusätzliche Klasse erforderlich ist. 3. Und wenn sie es realisieren, müssen sie diese Hilfsklasse lokalisieren, was auch etwas Zeit kostet.

Um auf die SKilL-Schnittstelle Rückschlüsse zu ziehen, kann man sagen, dass die Zustandsklasse, die wie eine Anfangsklasse der API agiert, tatsächlich zusätzliche Methoden enthalten kann, obwohl diese Methoden vielleicht in anderen Klassen sinnvoller untergebracht wären, wie z.B. die write-Methode.

3.4.6 Konstruktorparameter

Eine Praxis, die oft vorkommt, sind Konstruktoren mit Parametern, z.B. erfordert die SKilL-Schnittstelle für Scala Instanziierung der Typklassen mit Hilfe der Konstruktorparameter. Die Schnittstelle für Java funktioniert anders, hier werden Setter-Methoden und keine Konstruktoren mit Parametern benutzt.

Der übliche Ansatz, wie in der Schnittstelle für Java, ist "create-set-call". Um den Code vermeintlich einfacher zu halten und Dateneingaben erforderlich zu machen, entscheiden sich Schnittstellenentwickler manchmal für Konstruktoren mit Parametern. Das Arbeiten mit Objekten vereinfacht sich dabei zu "create-call". Allerdings ist dieser Ansatz für Nutzer weniger intuitiv und erschwert die Nutzung der API, wie es die Studie [SC] herausgefunden hat.

Dafür wurden 30 Entwickler ausgewählt, die eine Reihe an Aufgaben erledigen mussten. Währenddessen wurde beobachtet, wie die Entwickler auf die unterschiedlichen Ansätze in Schnittstellen im Bezug auf Instanziierung mit Kontruktorparametern reagieren. Den Entwicklern wurde nicht gesagt, was genau getestet wird, um sie möglichst wenig zu beeinflussen.

Die Aufgaben waren unterschiedlicher Natur. In der ersten Aufgabe musste ein Teil der Entwickler eine Schnittstelle in einem einfachen Texteditor schreiben, um zu sehen, was sie von einer Schnittstelle erwarten. Weitere Aufgaben erforderten die Nutzung unterschiedlicher APIs, manche mit Parametern in Konstruktoren, andere ohne. Die letzte Aufgabe war das Lesen eines ausgedruckten Code-Abschnittes, um zu prüfen, wie gut die Entwickler den Code verstehen. Hier wurde vor allem getestet, ob sie den Zweck eines Parameters bzw. Setters verstehen können. Außerdem gab es Gesprächsrunden, um qualitativ zu erfassen, ob die Nutzung von Konstruktorparametern gut ist.

Die Ergebnisse der Studie sahen folgendermaßen aus: Die meisten Entwickler erwarteten keine Konstruktoren mit Parametern. Sie haben in der Regel Konstruktoren ohne Parameter aufgeschrieben und erst später gemerkt, dass der Code so nicht kompiliert werden kann. Wenn der fehlerhafte Code von der Entwicklungsumgebung markiert wurde, nahmen sie an, dass sie einen gewöhnlichen Syntaxfehler gemacht haben. Dieses Verhalten änderte sich nicht im Verlauf der Tests, somit gab es keinen Lerneffekt. Eine andere Beobachtung war die fehlende Einsicht im Bezug auf die Funktionalität. Oft wurde z.B. ein null-Objekt als Parameter übergeben, ohne zu hinterfragen, ob das schädlich ist.

In den anschließenden Interviews erzählten die Entwickler außerdem, warum sie parameterlose Konstruktoren bevorzugen: Flexible Instanziierung, weniger beschränkend, Konsistenz, mehr Kontrolle. Erstaunlicherweise wurden optionale Konstruktoren nicht negativ aufgenommen.

Listing 3.2 Konstruktoren mit Parametern im Vergleich zu Methoden

```
...
// Instancing with costructors and parameters
Book book1 = new Book("Wie entwerfe ich gute Schnittstellen?", "978-3-86680-192-9", "Max
    Mustermann", "Springer", 2014, 276, 29.99);
...
// Alternative way of instancing
Book book2 = new Book();
book2.setTitle("Wie entwerfe ich gute Schnittstellen?");
book2.setISBN("978-3-86680-192-9");
book2.setAuthor("Max Mustermann");
book2.setPublisher("Springer");
book2.setYear(2014);
book2.setPageCount(276);
book2.setPriceInEuro(29.99);
...
```

3.5 Falsche Verwendung soll nicht möglich sein

Eine gut entworfene Schnittstelle muss vermeiden, dass der Nutzer einen falschen Code schreibt. Ein Beispiel dazu ist die Reihenfolge, in der Methoden aufgerufen werden.

Um auf die SKilL-Schnittstelle zurückzukommen, könnte der Entwickler *append*- und *write*-Operationen als separate Methoden implementieren. *append* erfordert allerdings ein vorausgehendes Lesen einer Datei. Wenn das Lesen nicht ausgeführt wird, wären die Ergebnisse der *append*-Operation unter Umständen nicht korrekt. Um das vermeiden empfiehlt es sich, eine gemeinsame Methode zu implementieren, die je nach Zustand die passende Operation ausführt.

Ein weiteres Beispiel ist das Zuweisen aller Objekte den richtigen Pools. Angenommen es existieren zwei Typen A und B. A enthält ein Feld vom Typ B, das nicht *null* sein darf. Um A zu serialisieren benötigt man somit auch B bzw. dessen Pool. Es wäre für den Nutzer einfacher, dass sobald ein Objekt vom Typ A in den A-Pool hinzugefügt wird, auch das Objekt vom Typ B, das als Feldwert beim A-Objekt gesetzt wurde, automatisch in den B-Pool hinzugefügt wird. Alternativ kann das zur Serialisierungszeit passieren.

3.6 Gewährleistung eines lesbaren Nutzer-Codes

Dieser kurze Abschnitt fasst das zusammen, was in 3.4 in Ansätzen angesprochen wurde, und gibt Richtlinien für einen lesbaren Nutzer-Code vor.

Es wurde außerdem gezeigt, dass Konstruktoren mit Parametern weniger intuitiv sind, da sie unter anderem Bezeichner verstecken können. Zudem können sie die Lesbarkeit des Codes deutlich verschlechtern, wenn viele Parameter erforderlich sind.

Listing 3.2 zeigt ein Beispiel für die schlechte Lesbarkeit des Codes, weil eine Schnittstelle Instanziierung von Typklassen mit Konstruktoren erlauben könnte, die viele Parameter enthalten. Während einige Parameter leicht erkannt werden können, wie etwa der Buchtitel, sind andere Parameter wie

die Seitenanzahl oder der Preis in € nicht mehr so einfach zu identifizieren. Der darauf folgende Code benutzt *set*-Methoden und ist besser lesbar, da hier die Felder benannt werden. Somit muss der Nutzer, der den Code vielleicht wartet, nicht die Dokumentation lesen, um den Code schnell zu verstehen. Auch die Wahl der selbstbeschreibenden Bezeichner spielt hier eine Rolle.

Das Beispiel enthält einen Typen mit nur sieben Feldern, die aus dem Zusammenhang erkannt werden können. Es kann allerdings deutlich komplizierter aussehen, wenn der Typ z.B. 20 oder mehr Felder hätte.

Ein anderer negativer Effekt betrifft die Reihenfolge. Methoden setzen keine Reihenfolge voraus, der Nutzer kann sie zu beliebigen Zeiten aufrufen und muss die Reihenfolge nicht aus dem Gedächtnis kennen. Die gilt, solange keine Entwicklungsumgebung oder ein Editor mit automatischer Vervollständigung zur Verfügung steht, was in vielen anderen Sprachen immer noch der Fall ist.

3.7 Erweiterbarkeit

Schnittstellen, wie jede andere Software, können mit der Zeit wachsen, z.B. indem neue Funktionen dazukommen. Sie müssen dann um neue Klassen und Methoden erweitert werden. Es kann auch möglich sein, dass die Funktionalität bzw. Implementierung geändert wird. Erweiterbarkeit ist somit eine der wichtigsten Eigenschaften. Die erste Version der API ist möglicherweise schnell implementiert und ist nicht perfekt. Die API kann daher fehlerhaft, unvollständig, schlecht entworfen oder dokumentiert sein. Aus diesem Grund muss sie einfach zu erweitern sein, sodass spätere Versionen Mängel korrigieren, ohne die Schnittstelle aus der Benutzersicht zu ändern bzw. sodass die Anzahl der benötigten Änderungen an dem Nutzer-Code so klein wie möglich gehalten wird.

Während der Entwicklung einer API muss man besonders auf diesen Punkt achten. Es sollte nichts implementiert werden, was zu viel sein könnte. Beispielsweise sollte eine zusätzliche Methode, die vielleicht nicht wichtig ist, nicht dem Nutzer zur Verfügung stehen. Dadurch wird das Entfernen solcher Methoden zu einem Problem, da das den Nutzer-Code unbrauchbar und nicht mehr kompilierbar machen kann. Eine Schnittstelle sollte daher so klein wie möglich aber dennoch vollständig sein.

Eine knappe Schnittstelle hat auch den Vorteil, dass sie leicht erweitert werden kann. Eine neue Methode oder Klasse hinzuzufügen ist leichter als etwas zu ändern oder zu entfernen. Es empfiehlt sich auch Interfaces oder abstrakte Klassen zu verwenden, die eine Schnittstelle definieren und entweder keine oder nur wenig Implementierung enthalten. Dadurch kann eine gewisse Konsistenz über einen längeren Zeitraum gewährleistet werden, da z.B. nicht Klassen sondern Interfaces als Typen im Nutzer-Code verwendet werden. In der SKilL-Schnittstelle wird z.B. eine abstrakte Pool-Klasse als Typ für alle Pools verwendet. Die eigentliche Implementierung ist in den Klassen *BasePool* und *SubPool*, beide können leicht in der Implementierung geändert oder um Funktionalität erweitert werden, ohne dass der Nutzer etwas merkt.

Als alternative Lösung für das Problem unterstützt Java z.B. *Deprecated*-Annotationen, mit den Methoden als veraltet markiert werden, um die Schnittstelle rückwärtskompatibel zu halten, zumindest für einige Versionssprünge. Die Nutzer werden in Entwicklungsumgebungen oder in Dokumentationen über den Status der markierten Methoden informiert, beispielsweise mit Warnungen, und

können eigene Software an die neue Version der API anpassen. Einige Versionen später können solche Methoden entfernt werden, um den Code der Schnittstelle zu säubern.

3.8 Weitere Merkmale einer guten Schnittstelle

- Parameterordnung soll konsistent sein. Falls z.B. eine Methode die Reihenfolge der Parameter setzt, sollte diese Reihenfolge über andere Methoden hinweg gleich sein.
- Methoden werden Feldern bevorzugt. Methoden haben den Vorteil, dass sie zusätzliche Funktionalität enthalten können, z.B. Validierung eines Wertes in der dazugehörigen Setter-Methode. Zudem lassen sich Felder schwerer entfernen, wenn sie nicht mehr benötigt werden. Implementierung in einer Methode lässt sich leicht ändern, da nur die Signatur für den Nutzer relevant ist. Ein öffentliches Feld ist Teil der Schnittstellen- sowie Nutzer-Implementierung.
- “When in doubt, leave it out“ ist eine bekannte und ungeschriebene Regel. Sie ist vor allem im Bezug auf die Erweiterbarkeit wichtig. Oft ist ein Schnittstellen-Entwickler nicht sicher, ob gewisse Funktionalitäten erwünscht sind. In diesem Fall sollte man sie auslassen und eventuell auf die Reaktionen vieler Nutzer warten. Nachrüsten geht immer, Entfernen dagegen nicht. In [Bla08] wird als Faustregel erwähnt, dass man auf mindestens drei Nutzer warten muss, die sich ein neues Feature wünschen.
- Zu clever sein ist nicht gut. Falls die Schnittstelle clevere Nebeneffekte in der Funktionalität aufweist, können sie für den Nutzer unverständlich sein. Als Folge können sie die API falsch verwenden. Außerdem können Tricks die Lesbarkeit verschlechtern.
- Ein einziger Einstiegspunkt macht den Anfang einfacher. In der SKilL-Schnittstelle erfolgt dieser Einstieg über die Zustandsklasse, in JAXB über die *Context*-Klasse. Solche Klassen sollen deutlich markiert werden, z.B. in der Dokumentation. Zudem sollen sie über Methoden oder sonstige Verweise dem Nutzer in der weiteren Verwendung der Schnittstelle behilflich sein.

4 Codegenerator

In diesem Kapitel geht es um das zweite Artefakt, den Codegenerator, der während der Anfertigung der Diplomarbeit entstanden ist. Der Generator erzeugt Teile der Schnittstelle. Dazu gehören die Typklassen sowie die Zustandsklasse. Eine besondere Eigenschaft des Generators soll laut Aufgabenstellung die Benutzerfreundlichkeit sein. Um eine einfache Nutzung zu gewährleisten, wurde unter anderem ein grafischer Prototyp entwickelt.

4.1 Anforderungen an den Codegenerator

Laut Aufgabenstellung gab es nur nichtfunktionale Anforderungen. Der Codegenerator soll benutzerfreundlich und einfach zu bedienen sein. Eine indirekte Anforderung war auch Konsistenz in der Bedienung, da bereits weitere Codegeneratoren für SKilL-Schnittstellen existieren. Daher wurde zuerst der Generator für die Scala-Schnittstelle analysiert, um herauszufinden, was dort umgesetzt wurde.

Der Codegenerator für Scala ist eine Konsolenanwendung, die bestimmte optionale und Pflichtparameter akzeptiert.

- Erforderlich: SKilL-Beschreibungsdatei mit Typdefinitionen
- Erforderlich: Ausgabepfad für generierte Dateien
- Erforderlich: Paketname für Klassen
- Optional: Headerzeilen
- Optional: Benutzername
- Optional: Datum

Um die Konsistenz zu gewährleisten ist der Generator für die Java-Schnittstelle auch eine Konsolenanwendung, die die erforderlichen Parameter des Scala-Codegenerators akzeptiert. Allerdings ist der Paketname als Parameter für den Java-Codegenerator nicht mehr Pflicht. Falls der Paketname nicht eingegeben wurde, wird es ausgelassen. Das wird zwar nicht empfohlen, ist aber von Java unterstützt und stört die SKilL-Schnittstelle nicht.

Eine Konsolenanwendung ist nichts, was sich intuitiv bedienen lässt. Es fehlt zunächst ein visuelles Feedback oder Metaphern, die dem Nutzer erlauben, Funktionalität mit passenden Elementen zu assoziieren. Eine Metapher kann z.B. ein Button als GUI-Element sein, der wie eine Taste, Schalter,

"(Druck-)Knopf" oder Ähnliches aussieht. Über die Assoziation mit etwas Echtem weiß der Nutzer, dass er auf einen Knopf drücken muss, um eine Aktion auszuführen.

In Konsolenanwendungen fehlt so etwas. Hier muss man gezielt dem Nutzer helfen, z.B. mit der Dokumentation, die erscheint, wenn falsche Eingaben erfolgen. Ebenso funktioniert der Codegenerator für die Scala-Schnittstelle, der einen Hilfetext anzeigt, wenn nichts eingegeben wurde. Dieses Konzept wurde in den Codegenerator für die Java-Schnittstelle übertragen. Somit ähneln sich beide Generatoren in der Bedienung und Funktionalität, was die Nutzung etwas erleichtert, wenn bereits einer von beiden Generatoren dem Nutzer bekannt ist.

Außerdem braucht eine Konsolenanwendung etwas, um Feedback während der Laufzeit darzustellen. Hier sind die Möglichkeiten stark reduziert. Es können letztendlich nur Ausgaben in die Konsole gemacht werden. Diese sollen natürlich so informativ wie möglich sein, aber dennoch den Nutzer nicht überfordern. Schlecht sind beispielsweise kryptische Informationen, wohingegen beschreibende Statusmeldungen gut sind.

Die Bedienung ist außerdem selbst mit Hilfen, die automatisch erscheinen, relativ nachteilig. Ein Nutzer kann sich z.B. während der Parametereingabe leicht vertippen oder falsche Angaben machen. Ebenso wie in der Schnittstelle sollte eine Konsolenanwendung nicht zu viele Parameter erfordern, das verschlechtert nicht nur die Benutzerfreundlichkeit sondern auch die Erlernbarkeit.

4.1.1 GUI-Codegenerator

Um Neulingen den Einstieg und generell die Nutzung zu erleichtern, wurden weitere Konzepte und Möglichkeiten analysiert, z.B. ein GUI-Codegenerator.

In Entwicklungsumgebungen wie Eclipse oder Netbeans existieren bereits grafische Codegeneratoren für diverse Serialisierungsschnittstellen wie JAXB. Im Rahmen der Diplomarbeit orientierte sich ein Ansatz an bereits vorhandenen Lösungen. Der umgesetzte GUI-Prototyp für den SKilL-Codegenerator orientiert sich an dem JAXB-Generator aus Eclipse, siehe Abbildung 4.1.

Der GUI-Codegenerator in Eclipse ist ein Wizard, also eine Folge von Dialogen, und begleitet den Nutzer durch die Eingabe der benötigten Optionen, um anschließend die Generierung der Klassen auszulösen. Am Anfang wählt dort der Nutzer eine XSD-Datei mit Typ- bzw. Schema-Definitionen, anschließend können der Ausgabeorder, Paketname und Sonstiges angegeben werden. Am Ende des Dialogs führt Eclipse automatisch xjc aus, den eigentlichen Codegenerator für JAXB.

Ein ähnliches Konzept verfolgt der GUI-Codegenerator für die SKilL-Schnittstelle. Zuerst kann der Nutzer einen passenden Codegenerator als Konsolenanwendung auswählen, z.B. den Codegenerator für die Java-Schnittstelle, da mehrere unterstützt werden. Danach wählt er die passende SKilL-Beschreibungsdatei. Im nächsten Schritt wird der Ausgabepfad sowie der Paketname ausgewählt. Anschließend können die Typklassen sowie die Zustandsklasse mit Hilfe des eigentlichen Codegenerators erzeugt werden.

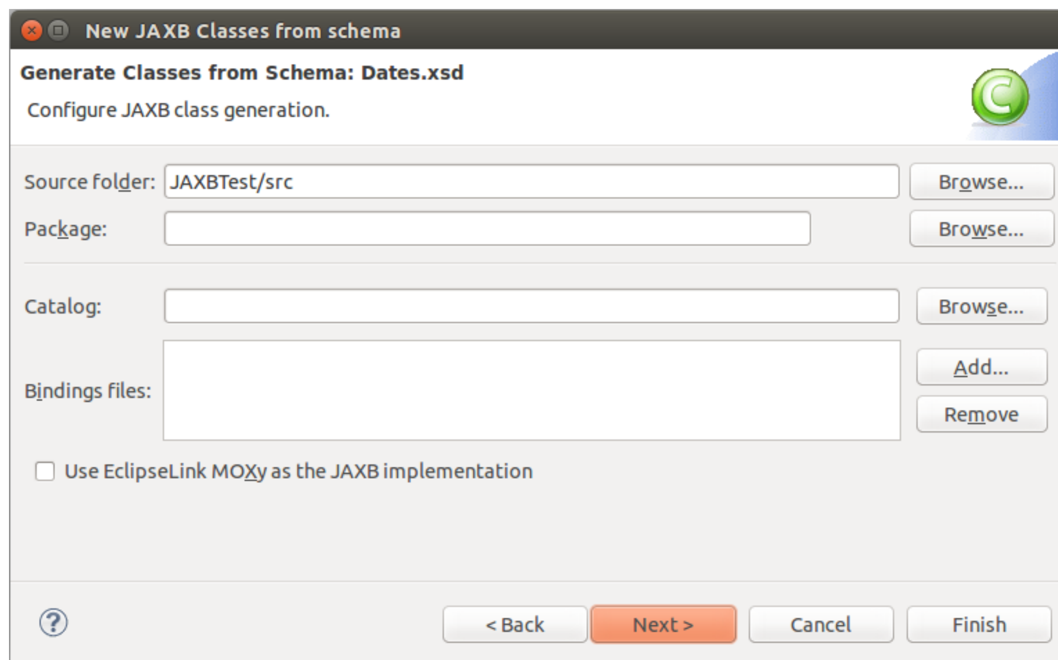

Abbildung 4.1: Ein Screenshot aus Eclipse, auf dem ein Teil des Wizards zur Generierung der JAXB-Klassen abgebildet ist.

4.2 Architektur und Implementierung des Codegenerators

In diesem Abschnitt wird kurz auf die Implementierung des Generators, als Kommandozeile, für die SKilL-Schnittstelle für Java eingegangen. Die Implementierungsdetails sowie Evaluation des GUI-Prototypen befinden sich im nächsten Abschnitt.

Abbildung 4.2 zeigt die Architektur des Codegenerators. *CodeGenerator* ist die Hauptklasse, die Benutzereingaben empfängt, weitere Klassen instanziiert, die Methoden in Instanzen aufruft und anschließend das JAR-Paket mit dem generischen Bindung zu den generierten Klassendateien im Ausgabeverzeichnis kopiert.

Die Parameter werden an eine Instanz der Klasse *ArgumentsHandler* übergeben, die sie analysiert und je nach Ergebnis der Analyse entweder einen Hilfetext in die Konsole ausgibt oder entsprechende Werte, z.B. den Paketnamen, über Getter-Methoden zur Verfügung stellt.

Danach wird mit einer Instanz der Klasse *SkillReader* die ausgewählte SKilL-Beschreibungsdatei, die Typdefinitionen enthält, ausgelesen und die Daten als Werte in die Instanzen der Klassen *SkillType* und *SkillField* gesetzt. Um Beschreibungsdateien zu lesen, benutzt *SkillReader* das SKilL-Frontend, auch Parser genannt, aus dem SKilL-Projekt für die Programmiersprache Scala [ski14]. Das stand bereits vor Beginn dieser Arbeit zur Verfügung und konnte somit als Abhängigkeit verwendet werden. *SkillType* und *SkillField* enthalten ausgelesene Typ- und Felddefinitionen, die über Getter-Methoden aufgerufen werden können. Eine Liste mit *SkillField*-Objekten ist ein Feld in *SkillType*. *Skillfield*-Objekte haben

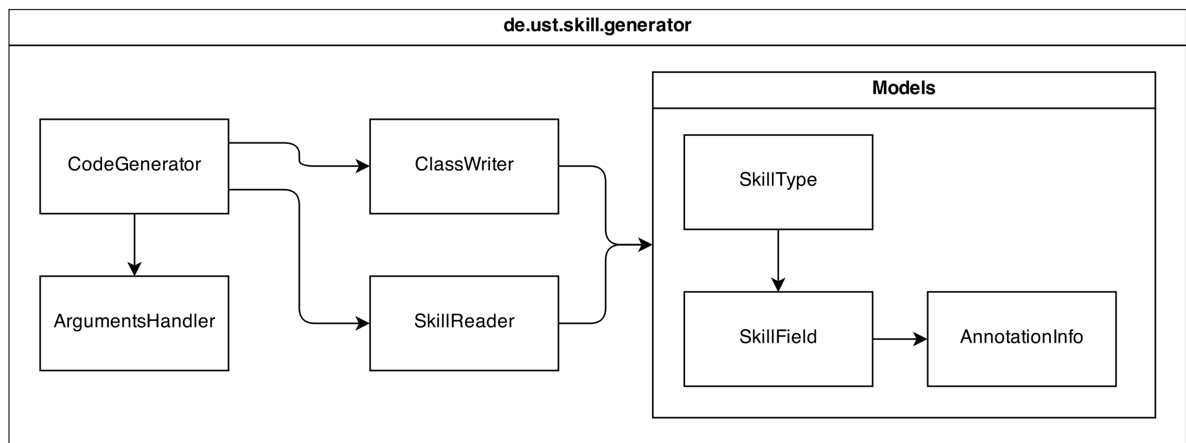

Abbildung 4.2: Entwurf des Codegenerators

außerdem ein Feld vom Typ *AnnotationInfo*, dessen Instanzen zusätzliche Informationen für die Klassengenerierung enthalten, z.B. Markierung eines Feldes als auto.

Nach dem Parsing wird eine Liste mit erkannten SKilL-Typen an die Instanz der Klasse *ClassWriter* übergeben. Diese Klasse verwendet Freemarker [fre14], eine Template-Engine. Freemarker generiert Textdateien aus Vorlagen und ist (Turing-)mächtig. Die Vorlagen erlauben Scripting sowie diverse andere Konstrukte und Möglichkeiten. Außerdem kann man POJO-Instanzen an die Engine übergeben, sodass man auf ihre Felder in den Templates zugreifen kann. Es können z.B. Instanzen der *SkillType*-Klasse an Freemarker übergeben werden, um auf ihre Felder von Templates aus zuzugreifen.

Dadurch vereinfacht sich die Generierung der Klassen enorm, da somit nicht ganze Klassenabschnitte in Maker-Klassen oder anderen Ressourcen liegen müssen. Es reicht aus, für jeden Klassentypen eine Freemarker-Vorlage zu definieren und dann Freemarker die Daten, als POJOs oder über eine Map, sowie die Vorlagen zu übergeben.

Listing 4.1 zeigt die vereinfachte Darstellung einer Vorlage. Hierbei handelt es sich um das Template für die Typklassen, die zuerst eine Liste an Feldern und dann die passenden Getter- und Setter-Methoden enthalten. Ein doppeltes Fragezeichen ist eine Prüfung des gesetzten Wertes, so kann man erkennen, ob der Nutzer einen Paketnamen eingegeben hat oder ob ein Elterntyp existiert. Die Syntax erlaubt aber auch normale if-Abfragen.

Durch die Nutzung von Freemarker wird der Code relativ einfach gehalten, die Wartung erleichtert sich enorm. Um etwas an den generierten Dateien zu ändern reicht es die POJO-Klassen anzupassen sowie anschließend entsprechende Änderungen in den Templates vorzunehmen.

Da vom Codegenerator nur ein Teil der Schnittstelle generiert wird, muss der Nutzer auf irgendeine Art und Weise von dem generischen Binding erfahren. Dazu nimmt der Generator das bereits vorkompilierte Binding als JAR-Paket und kopiert es in das Ausgabeverzeichnis zu den generierten Klassen. Der Nutzer kann anschließend das Paket als Abhängigkeit in das eigene Projekt einbinden. Falls das Projekt als ein Maven-Projekt [mav14] umgesetzt wird, kann der Nutzer das Binding kompilieren und dann die *deploy*-Operation mit Maven ausführen, die diesen generischen Teil der Schnittstelle

Listing 4.1 Vereinfachte Darstellung der Typklassen-Vorlage für FreeMarker

```
<#if package??>package ${package};</#if>

<#list imports as import>
import ${import};
</#list>

public class ${typeName} <#if supertypeName??> extends ${supertypeName}</#if> {

// List of fields
<#list fields as field>
   private ${field.returnType} ${field.escapedName?lower_case};
</#list>

<#list fields as field>

   // Getter for the field
   public ${field.returnType} get${field.name?capitalize}() {
      return ${field.escapedName?lower_case};
   }

   // Setter for the field if it is not a constant
   <#if !field.constant>
   public void set${field.name?capitalize}(${field.returnType} ${field.escapedName?lower_case}) {
      this.${field.escapedName?lower_case} = ${field.escapedName?lower_case};
   }
   </#if>

</#list>
}
```

in die lokale oder eine entfernte Repository einfügt. Der Vorteil dadurch wäre, dass der Nutzer nur noch in *pom.xml* des Projektes auf das Binding verweisen müsste. Die Handhabung des Bindings als Abhängigkeit würde dann Maven übernehmen. Auch die Versionsverwaltung kann von Maven automatisch erledigt werden, z.B. wenn neue Versionen des Bindings erscheinen.

Während der Erzeugung der Klassen zeigt der Codegenerator passende Statusmeldungen in der Konsole, beispielsweise welche Klassen mit dem Pfad generiert wurden. Hier ist die Umsetzung limitiert, da nur eine Ausgabe in die Konsole gemacht wird. Alternativ könnte ein Logger verwendet werden, der in mehrere Ziele den Status der Erzeugung ausgeben kann, z.B. in die Konsole und in eine Datei. Dies wurde allerdings im Rahmen dieser Arbeit nicht umgesetzt, da ein Logger eventuell Abhängigkeiten mit sich bringen könnte, welche die Wartung erschwert und unter Umständen Lizenzkonflikte verursacht.

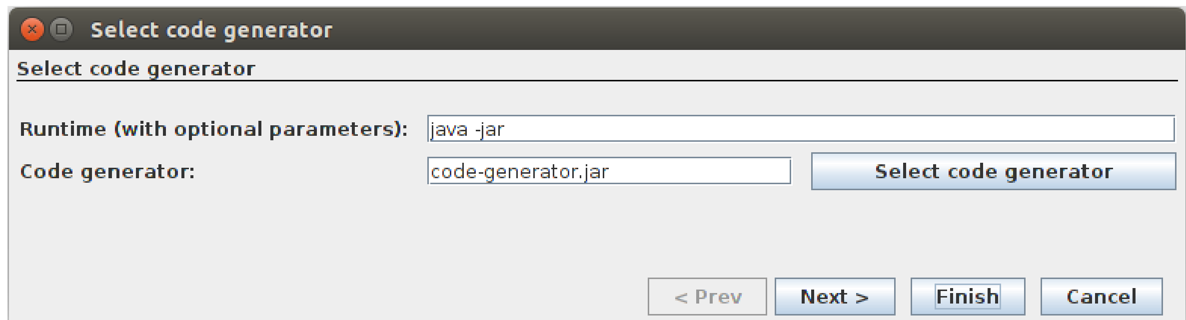

Abbildung 4.3: Die erste Ansicht im GUI-Prototyp, der die Nutzung des Codegenerators erleichtern soll

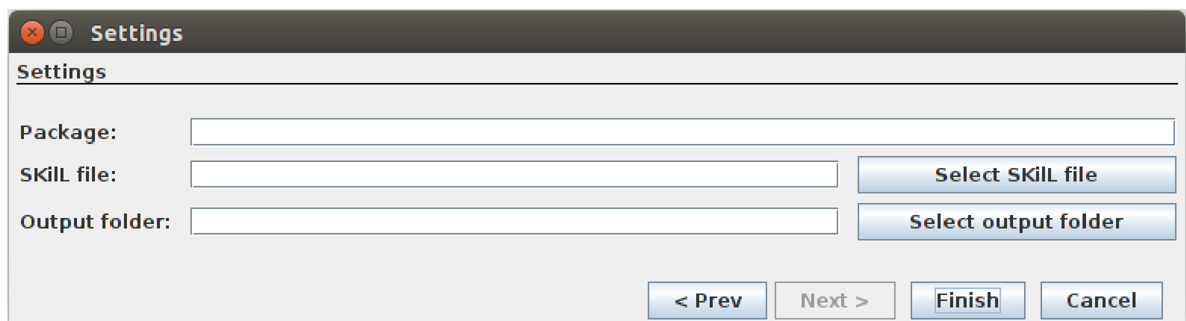

Abbildung 4.4: Die zweite Ansicht des GUI-Prototypen

4.3 Implementierung des GUI-Prototypen

Eine Konsolenanwendung ist nicht sonderlich intuitiv oder einfach nutzbar. Aus diesem Grund wurde ein Prototyp mit einer grafischen Oberfläche als eine Art Frontend für den Codegenerator umgesetzt. Es empfiehlt sich, so etwas als ein Wizard-Dialog umzusetzen, der den Nutzer während der Generierung Schritt für Schritt begleitet und ihm dabei assistiert.

In Java gibt es zwei bekannte Technologien, um grafischen Oberflächen zu erzeugen: Swing [swi14] und JavaFX [jav14b]. JavaFX ist moderner als Swing und erzeugt auch ansehnliche Oberflächen. Es ist allerdings schwer in der Wartung, da JavaFX sich nicht im Klassenpfad von Java 7 befindet. Der Nutzer oder jemand, der den GUI-Prototypen kompiliert, müsste somit manuell die passenden JavaFX-Abhängigkeiten in den Klassenpfad einfügen. Das Problem wurde zwar in Java 8 behoben, allerdings wurde mit dieser Diplomarbeit vor dem Release-Termin von Java 8 begonnen. Um den Nutzer damit nicht zu überfordern, wurde daher auf die Verwendung von JavaFX verzichtet und die Oberfläche mit Swing umgesetzt.

Der GUI-Prototyp besteht aus nur drei Klassen, einer Hauptklasse, die eine fremde Bibliothek zum Erzeugen von Wizard-Dialogen [swi] verwendet, und je einer Klasse für einen Teil des Prototypen.

Der Prototyp bzw. der Wizard-Dialog zeigt nur zwei Ansichten an, siehe Abbildungen 4.3 und 4.4. Für jede Ansicht gibt es eine passende Definition als Klasse.

Um die beiden Ansichten ansprechend zu gestalten, wurde ein spezielle Bibliothek [des14] verwendet, die ein Layout zur Verfügung stellt, das für Formulare gedacht ist. Das ist ein wichtiger Punkt, da es viele Richtlinien gibt, die darauf besonders eingeben.

Wenn der Nutzer auf "Finish" drückt, ruft der GUI-Prototyp den eigentlichen Codegenerator auf und übergibt die Eingaben aus dem Dialog als Parameter. Optional, das wurde aus Zeitgründen nicht umgesetzt, können am Ende der Ausführung Ausgaben des Generators im GUI-Prototypen ausgelesen und dem Nutzer in passender Form präsentiert werden..

4.4 Nutzbarkeitevaluation des Codegenerators

Dieser Abschnitt zeigt, wie man gut zu bedienende Anwendungen implementiert. Es wird gezeigt, warum eine Konsolenanwendung nicht optimal ist und wie man es besser machen kann. Es wird allerdings nicht in Detail darauf eingegangen, da der Schwerpunkt hier der Generator ist und nur dafür relevante Punkte angesprochen werden.

4.4.1 Kommandozeile

Konsolenanwendungen oder Kommandozeilen sind nicht per se schlechter als Anwendungen mit grafischen Oberflächen. Sie sind oft mächtiger und erfordern weniger Abhängigkeiten, z.B. ein grafisches System zur Anzeige. Sie sind allerdings weniger intuitiv in der Bedienung, da am Anfang visuell nichts präsentiert wird, woran der Nutzer sich zum weiteren Vorgehen orientieren kann. Man sieht nur ein Eingabefeld, was oft auch nicht als ein Feld dargestellt wird. Um weiterzukommen muss der Nutzer aus dem Gedächtnis oder mit Hilfe der Dokumentation teilweise kryptische Befehle eingeben.

Das größte Problem mit Kommandozeilen sind die Erforschungsmöglichkeiten, da oft nichts auf die Nutzung und Funktionalität deutet. Der Nutzer könnte ratlos vor Eingabe stehen, meist vor einem blinkenden Zeichen, und nicht vorankommen. Es gibt keine Menüs, keine Buttons oder visuelle Hilfe. Ein erfahrener Nutzer kann versuchen, zur Verfügung stehende Eingabeparameter anzeigen zu lassen. Dafür kann er sich an bekannten Parametern orientieren, die häufig in anderen Anwendungen vorkommen, z.B. "-h" für die Hilfe. Die Hilfe kann wiederum weitere Optionen und Beschreibungen zur Nutzung enthalten.

Ein weiteres Problem ist die Syntax. Es gibt für einzelne Umgebungen Richtlinien im Bezug auf Parametereingabe in Kommandozeilen und teilweise auch Implementierungen, z.B. *getopt* für GNU [get14]. An diesen kann der Entwickler sich entweder orientieren oder entsprechende Implementierungen verwenden. Es existieren aber auch Normen, z.B. IEEE Std 1003.1 [iee13]. Diese Norm gibt bestimmte Richtlinien für Parameter in Konsolenanwendungen vor. Anbei eine Zusammenfassung dieser Richtlinien.

- Aufruf einer Anwendung soll wie folgt passieren:
  utility_name[-a][-b][-c option_argument][operand...]
  Auf den Namen der Anwendung folgt eine Liste mit Optionen, die jeweils mit einem Strich anfangen, sowie zusätzlichen Argumenten, und eine Liste mit Operanden, die immer notwendig sind.
- Namen der Anwendungen sollen zwischen 2 und 9 Zeichen lang sein. Sie müssen klein geschrieben und in den Namen dürfen nur bekannte Zeichen verwendet werden.
- Optionen können zusammengefasst werden, wenn sie keine Argumente enthalten, z.B.:
  utility_name[-ab][operand...]
- Optionen werden in alphabetischer Ordnung aufgelistet, allerdings sollen sie in einer beliebigen Reihenfolge aufrufbar sein. Die Ordnung der Operanden ist fest.
- Optionen sollen nur aus einem Zeichen bestehen, großgeschriebene Zeichen sollen vermieden werden.
- Zahlen in Parametern werden als dezimale Integer intepretiert

Wissenschaftliche Untersuchungen zur Nutzbarkeit von Kommandozeilen sowie ihren funktional ähnlichen grafischen Benutzeroberflächen gab es schon vor langer Zeit. In 1987 untersuchten Margono und Shneiderman [MS87], wie schnell Benutzer eine einfache Aufgabe mit der Kommandozeile sowie einer grafischen Oberfläche, als direkte Manipulation an repräsentativen Modellen, erledigen. Die Aufgabe beinhaltete das Kopieren von zwei Dateien. 12 Personen nahmen an dem Test teil. Das Ergebnis: Mit der Kommandozeile konnten die Tester die Aufgabe in 5,8 Minuten (Median-Wert) erledigen. Dabei machten sie 2,4 Fehler. Mit der grafischen Oberflächen sanken die Zahlen respektive auf 4,8 Minuten und 0,8 Fehler. Somit waren die Tester nicht nur etwas schneller, sie machten auch deutlich weniger Fehler. Es gab zudem eine qualitative und subjektive Umfrage, in der die Nutzer die grafische Oberfläche besser bewerteten.

Ein ähnliches Experiment wurde auch von Matthias Rauterberg in 1992 [Rau92] gemacht. In seinem Test mussten Nutzer eine Datenbank über eine zeichenbasierten sowie mit einer grafischen Oberfläche bedienen. Das Resultat war eindeutiger als in der oben erwähnten Studie [MS87]. Die Nutzer wurden hier in zwei Gruppen aufgeteilt, Neulinge und Experten. Mit der zeichenbasierten Oberfläche konnten Neulinge fünf Aufgaben in 82 Minuten erledigen, mit der grafischen hingegen in nur 49 Minuten. Die Experten waren auch schneller, allerdings ist das Ergebnis hier noch deutlicher. Sie erledigten die Aufgabe in 25 und 10 Minuten.

Beide Studien sind sehr spezifisch, sie geben eine grobe Vorstellung davon, was besser in der Bedienung ist. Sie sind jedoch nicht geeignet, um eine generelle Aussage über die Nutzbarkeit der verschiedenen Benutzerschnittstellen zu machen. Oft ist es so, dass je nach Einsatzzweck Kommandozeilen besser sind und die Nutzbarkeit entweder gar nicht oder nur wenig erforderlich ist. Sie erlauben z.B. Scripting bzw. Automatisierung, sie laufen in minimalen Umgebungen usw.

In jedem Fall sollte die Konsolenanwendung dem Nutzer Informationen bereitstellen, sodass er erstens ein visuelles Feedback bekommt, wenn etwas falsch ausgeführt wird, und zweitens um ihm das Erlernen der Bedienung zu erleichtern. Folgende Punkte helfen bei der Gestaltung von Kommandozeilen-Anwendungen:

- Einfache und verständliche Syntax.
- Hilfen in Textform sollen immer erscheinen, wenn sie sinnvoll sind, nicht nur wenn Benutzer sie explizit aufrufen, z.B. wenn der Nutzer falsche Eingaben macht.
- Unterstützung von Manpages. Das ist vielleicht nur für solche Systeme wie Linux interessant, Manpages sind aber dort ein beliebter Startpunkt in der Nutzung einer Anwendung.
- Visuelles Feedback. Wenn etwas Wichtiges passiert, sollen Meldungen mit ansprechender Formatierung angezeigt werden.

4.4.2 Grafischer Codegenerator

Für SKilL sowie passende Schnittstellen und Generatoren ist es vermutlich besser je nach Zweck unterschiedliche Ansätze zu verwenden. Angenommen, SKilL soll für eine breite Masse an Entwicklern attraktiv gemacht werden. Eine solche Menge an Leuten umfasst verschiedene Erfahrungen und Anforderungen. Für weniger Erfahrene oder für solche, die sich nicht zu sehr in das Thema einarbeiten wollen, ist es tatsächlich von Vorteil etwas zu haben, das sich leicht bedienen lässt, z.B. in Form eines grafischen "Wrappers" um den Codegenerator. Als allgemeine Rückfall-Lösung kann immer noch die Kommandozeile verwendet werden. Sollte SKilL jemals in eine Entwicklungsumgebung integriert werden, ähnlich zu JAXB in Eclipse, ist ein grafischer Codegenerator auf jeden Fall notwendig. Daher umfasst diese Arbeit auch einen grafischen GUI-Prototypen, damit sich kommende Lösungen eventuell daran orientieren können.

Nicht jede grafische Oberfläche ist gut. Auch wenn Grafikbibliotheken wie Swing oder JavaFX viel vorgeben, z.B. das Aussehen von Buttons im Standarddesign, der Entwickler kann bei der Umsetzung der GUI immer noch einiges falsch machen. Solche Bibliotheken lassen offenen Spielraum zu und erlauben in den meisten Fällen sogar das Ändern von Standardkomponenten bzw. deren Design. Allerdings endet dies nicht immer mit guten Ergebnissen.

Um vor allem unerfahrenen Entwicklern zu helfen, wurden viele Richtlinien aufgestellt. Es gibt auch Normen und Standards, die allerdings sehr allgemein ausgerichtet sind, z.B. EN ISO 9241. Konkreter sind Richtlinien, die von Unternehmen oder Organisationen wie Microsoft [mic01], Apple [osx14], GNOME [gno14] etc. vorgegeben werden. Um alles zu erfassen, bräuchte man vermutlich mehrere Bücher - in dieser Arbeit wird nur auf die für den Codegenerator relevanten Aspekte eingegangen, d.h. Dialoggestaltung.

Grunsätzliche Richtlinien für Dialoggestaltung, die aus den oben genannten Quellen kommen und ergänzende Informationen enthalten...

- Pro Ansicht soll nur eine Aufgabe bearbeitet werden. Um auf GUI-Prototypen zurückzukommen, kann man hier die Ansichten des Wizard-Dialogs für passende Teilaufgaben trennen, beispielsweise wählt der Nutzer im ersten Schritt nur den Codegenerator als Konsolenanwendung, wie in der Abbildung 4.3 zu sehen. Im zweiten setzt er Optionen wie den Paketnamen oder wählt eine SKilL-Beschreibungsdatei aus.
- Die Aufgaben sollen effizient erledigt werden, nichts Unnötiges soll dem Nutzer im Weg stehen. Besonders wichtige Stellen der Ansichten sollen markiert werden, damit sie schneller erfasst werden können, z.B. erforderliche Eingabefelder.
- Aufgaben beschreiben, z.B. indem man für jede Ansicht einen passenden und kurzen Titel wählt, "Auswahl des Codegenerators" oder Ähnliches. Die Ansichten können außerdem kurze Hilfestellungen enthalten oder auf weitere Hilfen verweisen. Man sollte es allerdings nicht übertreiben und sich knapp halten, da die Nutzer dazu tendieren, viel Text nicht zu lesen oder nur zu überfliegen.
- Es soll möglich sein, aus dem Kontext die Aufgabe leicht zu erkennen, d.h., dass z.B. Eingabefelder oder Buttons passend benannt werden müssen.
- Feedback geben. Der Nutzer soll wissen, was passiert, wenn er eine Aktion ausführt. Zudem können während des Feedbacks weitere Aktionen vorgeschlagen werden, z.B. etwas korrigieren oder abbrechen.
- Selbstbeschreibende Elemente. Symbole, die dem Nutzer auf den ersten Blick nicht ersichtlich sind, sind schlecht. Oft reicht beispielsweise der Platz nicht für ein größeres und besseres Symbol aus, daher nimmt man Icons, die nicht intuitiv erkennbar sind. In solchen Fällen kann man auf zusätzliche Elemente wie Tooltips zurückgreifen.
- Konsistente Anzeige. Dialoge sollen so gestaltet werden, dass sie sich ähneln. Beispielsweise sollen Buttons die gleiche Größe haben, sie sollen konsistent angeordnet sein, usw. Für Dialoge wichtig sind außerdem Fluchtlinien, zu viele Linien führen zu falschen Zuordnungen, was die Nutzer irritiert.
- Die Anzahl gleicher Elemente soll überschaubar sein, also z.B. nicht zu viele Buttons oder Eingabefelder darstellen. So etwas verschlechtert den Überblick. Stattdessen sollte die Ansicht in mehrere Ansichten unterteilt werden.
- Einfache Führung. In Wizard-Dialogen braucht man Buttons zum Blättern, "Weiter" und "Zurück". Außerdem sollen Möglichkeiten vorhanden sein, um eine Aufgabe abzuschließen, z.B. den Codegenerator nach vollständiger Eingabe ausführen, bzw. die Aufgabe abzubrechen.
- Konsistente Beschriftungen. Nicht nur innerhalb der Anwendung selbst sollte man allgemein bekannte Begriffe verwenden, wie etwa "Beenden" oder "Abbrechen" für Buttons.
- Hässliche Oberflächen vermeiden. Es gibt Farben, die nicht zueinander passen, z.B. Blau und Rot. Einer der Gründe ist die Wellenlänge. Diese Farben befinden sich an den gegenüberliegenden "Enden" des sichtbaren Lichtspektrums, somit haben sie deutliche Unterschiede in der Wellenlänge. Das Auge kann sich daher nur schlecht auf beide Farben gleichzeitig konzentrieren, wenn sie sich zu nah aneinander befinden. Dezente und weniger kontrastreiche Farben sind besser.

Der GUI-Prototyp erfüllt die meisten dieser Punkte. Dort wurde die Aufgabe, Generierung von Teilen der Schnittstelle, in Unteraufgaben getrennt. Die Buttons sind klar markiert und sind beschreibend. Es gibt für beide Ansichten Titel, die kurz die Teilaufgabe dem Nutzer erklären. Weitere kleinere Aufgaben, wie die Auswahl von Dateien, befinden in zusätzlichen Dialogen und sind über Buttons als Verweise oder Weiterleitungen erreichbar.

Es fehlt allerdings Feedback. Ursprünglich wurde noch eine dritte Ansicht geplant, die die Ausgabe des Kommandozeilen-Generators darstellen sollte, um dem Nutzer zu zeigen, was generiert wurde und wo die generierten Dateien liegen. Das wurde aber verworfen, da während der kurzen Testphase der Codegenerator so schnell war, sodass die Anzeige nur ganz kurz erschien und der GUI-Prototyp nach einer erfolgreichen Ausführung schnell geschlossen wurde. Der JAXB-Generator in Eclipse funktioniert etwas anders. In Eclipse gibt es eine zusätzliche Ansicht für die Konsole. Während der Generierung zeigt Eclipse alle Ausgaben in dieser zusätzlichen Ansicht, die sich nicht im Wizard-Dialog befindet. Somit bleibt dort die Anzeige auch nach dem Beenden des Generierungsvorgangs. Auf eine ähnliche Weise könnte man das auch für einen potentiellen GUI-Generator für SKilL in Eclipse oder einer anderen Entwicklungsumgebung umsetzen.

5 Zusammenfassung und Ausblick

Es wurde untersucht, in wie weit die Schnittstelle den Vorstellungen der Nutzer entspricht. Dafür wurde sie und ihre Teile aus verschiedenen Perspektiven betrachtet und analysiert. Es wurde Bezug zu vielen relevanten Studien genommen, um herauszufinden, was Nutzer als gut und benutzerfreundlich empfinden. Das Thema ist jedoch subjektiv, da jeder unter Benutzerfreundlichkeit jeder etwas Anderes versteht. Hier spielen Erfahrungen - von Nutzern und Entwicklern - Kenntnisse, Geschmack usw. eine Rolle, beispielsweise kann man im Bezug auf Erfahrung die Nutzer in Neulinge und Experten aufteilen. Daher muss man ganz genau die Anforderungen und Bedingungen definieren. Eine Schnittstelle kann für erfahrene Nutzer anders gestaltet werden als für Neulinge. Die Studien waren ein sehr guter Anhaltspunkt, da sie grob erfasst haben, was man von einer Schnittstelle erwartet. Es wurde unter anderem herausgefunden, dass Fabrikmethoden als unfreundlich angesehen werden, da sie nicht der klassischen Instanziierung von Klassen entsprechen. Es wurde auch gezeigt, dass viele Klassen und Methoden sich kontraproduktiv auswirken können, da sie die Suche nach benötigten Klassen/Methoden verlängern.

Während der Arbeit sind zwei Artefakte entstanden, der Codegenerator und die Schnittstelle, wobei ein Teil der Schnittstelle generisch ist und der andere von dem Codegenerator erzeugt wird.

Durch die Aufteilung der Schnittstelle wurde nicht nur die Implementierung einfacher, es gibt auch Vorteile für den Endnutzer. Der generische Teil kann leicht als Abhängigkeit in Projekte eingebunden werden. Erstens wird ein Projekt nicht mit für den Nutzer uninteressanten Klassen gefüllt. Zweitens ermöglicht es eine leichte Austauschbarkeit der Schnittstelle bei einer Versionsänderung, ohne dass der generierte Teil nochmal generiert werden muss, z.B. wenn die Performance der API verbessert wird. Vergleichbar ähnlich funktionieren andere und beliebte Schnittstellen wie JAXB.

Ähnliches wurde auch für den Codegenerator untersucht. Der Generator wurde in zwei unterschiedlichen Formen umgesetzt, als eine Kommandozeilenanwendung und als grafische Oberfläche. Obwohl Kommandozeilen oft Vorteile bieten, sind sie nicht benutzerfreundlich. Es wurde zuerst gezeigt, wie man Konsolenanwendungen so gestalten kann, dass ihre Bedienung verbessert wird. Die Nutzer sollen z.B. leicht Hilfen erreichen können, da sie anfänglich nur vor einem blinkenden Zeichen sitzen und ohne zusätzliche Informationen nicht auf weiteres Vorgehen schlussfolgern können.

Dann wurde ein alternativer Vorschlag für eine andere Art des Codegenerators gemacht, als GUI-Anwendung. Da es sich lediglich um einen Prototypen handelt, ist es nicht ratsam ihn in produktiven Umgebungen einzusetzen, obwohl er funktional ist. Daher gab es Vorschläge für potentielle Lösungen, die zeigen, wie man einen Codegenerator als grafische Oberfläche am besten umsetzt. Hier spielt die Dialoggestaltung eine wichtige Rolle. Die Dialoge sollen die Aufgabe in kleine Teilaufgaben trennen und den Nutzer schrittweise bei der Erledigung führen.

Die Diplomarbeit zeigt, dass es möglich ist, eine Schnittstelle und einen Codegenerator für SKilL zu entwickeln, die sich leicht bedienen lassen, sich gut in die Programmiersprache integrieren und einen einfachen Einstieg in die Nutzung ermöglichen.

Es wird auch gezeigt, dass um SKilL-Schnittstellen verwenden zu können, der Nutzer sich nicht gut mit SKilL auskennen muss. Es reicht, wenn er in dem SKilL-Beschreibungsformat Typen definieren kann. Anschließend kann er einen einfachen Codegenerator verwenden, um die Schnittstelle bzw. Teile davon zu generieren. Die Sprache und die Java-Anbindung stehen somit professionellen Lösungen wie XML/XSD und JAXB zumindest in der Benutzerfreundlichkeit nicht nach.

Trotz der Fortschritte, die in die Richtung der Nutzbarkeit gemacht wurden, sind die Schnittstelle und der Codegenerator nicht optimal. Es können noch Funktionen umgesetzt werden, die die Benutzerfreundlichkeit weiter verbessern. Dazu gehört z.B. das automatische Einfügen von Instanzen in passende Pools, die zwar irgendwo als Felddaten gesetzt wurden, aber sich nicht in den entsprechenden Pools befinden. Die Schnittstelle könnte solche Abhängigkeiten automatisch auflösen und unbekannte Objekte in die Pools zum Serialisieren einfügen.

Auch der grafische Codegenerator kann besser gestaltet sein, es fehlt z.B. gutes visuelles Feedback nach dem Generieren. Die Oberfläche wirkt wegen Java Swing zudem veraltet und kann mit JavaFX 8 oder anderen modernen Frameworks deutlich besser gestaltet werden. Alternativ kann der GUI-Generator als Erweiterung für Eclipse oder Netbeans implementiert werden.

Literaturverzeichnis

[Bla08] J. Blanchette. The Little Manual of API Design. 2008. (Zitiert auf Seite 49)

[des14] DesignGridLayout, 2014. URL https://designgridlayout.java.net/. (Zitiert auf Seite 57)

[ESM] B. Ellis, J. Stylos, B. Myers. The Factory Pattern in API Design: A Usability Evaluation. (Zitiert auf den Seiten 14 und 43)

[Fel13] T. Felden. The SKilL Language. Technischer Bericht Informatik 2013/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2013-06&engl=0. (Zitiert auf den Seiten 8, 9, 14 und 15)

[fre14] FreeMarker Java Template Engine, 2014. URL http://freemarker.org/. (Zitiert auf Seite 54)

[get14] Getopt - The GNU C Library, 2014. URL http://www.gnu.org/software/libc/manual/html_node/Getopt.html. (Zitiert auf Seite 57)

[gno14] GNOME Richtlinien für Benutzeroberflächen, 2014. URL https://developer.gnome.org/hig-book/stable/. (Zitiert auf Seite 59)

[iee13] The Open Group Base Specifications Issue 7, 2013. URL http://pubs.opengroup.org/onlinepubs/9699919799/. (Zitiert auf Seite 57)

[jav14a] ArrayList, 2014. URL http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

[jav14b] JavaFX - The Rich Client Platform, 2014. URL http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. (Zitiert auf Seite 56)

[jax14] JAXB Reference Implementation, 2014. URL https://jaxb.java.net/. (Zitiert auf den Seiten 14 und 40)

[mav14] Apache Maven, 2014. URL http://maven.apache.org/. (Zitiert auf Seite 54)

[mic01] Microsoft Inductive User Interface Guidelines, 2001. URL http://msdn.microsoft.com/en-us/library/ms997506.aspx. (Zitiert auf Seite 59)

[MS87] S. Margono, B. Shneiderman. A Study of File Manipulation by Novices Using Commands vs. Direct Manipulation. 1987. (Zitiert auf Seite 58)

[osx14] OS X Human Interface Guidelines, 2014. URL https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/AppleHIGuidelines/Intro/Intro.html. (Zitiert auf Seite 59)

[poj14] POJO, 2014. URL http://www.martinfowler.com/bliki/POJO.html. (Zitiert auf Seite 22)

[Rau92] M. Rauterberg. An empirical comparison of menu-selection (CUI) and desktop (GUI) computer programs carried out by beginners and experts. Behaviour and Information Technology. 1992. (Zitiert auf Seite 58)

[RD10] M. P. Robillard, R. DeLine. A field study of API learning obstacles. In P. T. Devanbu, Herausgeber, *Empirical Software Engineering*. Springer, 2010. (Zitiert auf den Seiten 14, 36 und 38)

[Rob09] M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 2009. (Zitiert auf den Seiten 37 und 38)

[SC] J. Stylos, S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. (Zitiert auf Seite 46)

[SGB+] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, J. Karstens. A Case Study of API Redesign for Improved Usability. (Zitiert auf den Seiten 14 und 17)

[SK12] T. Scheller, E. Kühn. Influencing Factors on the Usability of API Classes and Methods. *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, 2012. (Zitiert auf den Seiten 39 und 41)

[ski14] Cross platform, cross language, easy-to-use serialization interface generator., 2014. URL https://github.com/skill-lang/skill. (Zitiert auf den Seiten 8, 14, 31 und 53)

[SM] J. Stylos, B. A. Myers. The Implications of Method Placement on API Learnability. (Zitiert auf den Seiten 36 und 45)

[swi] An API for creating step-by-step wizards with Java. URL https://github.com/creswick/cjwizard. (Zitiert auf Seite 56)

[swi14] A Swing Architecture Overview, 2014. URL http://www.oracle.com/technetwork/java/architecture-142923.html. (Zitiert auf Seite 56)

[Tul08] J. Tulach. *Practical API Design. Confessions of a Java Framework Architect.* Apress, 2008. (Zitiert auf Seite 44)

Alle URLs wurden zuletzt am 03. 07. 2014 geprüft.

Erklärung

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

Ort, Datum, Unterschrift