Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3665

Plattform- und sprachunabhängige Serialisierung mit SKilL

Fabian Harth

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Dipl.Inf. Timm Felden
Beginn am:	5. Mai 2014
Beendet am:	4. November 2014
CR-Nummer:	D1.5,D.3.3,D.3.4,E.2

Kurzfassung

Die Diplomarbeit beschreibt die Entwicklung einer Sprachanbindung an die Programmiersprache C für die Serialisierungssprache SKilL, um nachzuweisen, dass die Anbindung an eine nicht objektorientierte Sprache möglich ist. Im Besonderen wird die Darstellung von Daten behandelt, die in einer Vererbungshierarchie organisiert sind. Wir beschreiben die Gestaltung einer intuitiv zu bedienenden Benutzungsschnittstelle, die Möglichkeiten einer objektorientierten Sprache nachstellt, und veranschaulichen die Verwendung anhand von Beispielen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einführung

1.1 Gliederung

Dieses Kapitel stellt bisherige Arbeiten zu SKilL vor und gibt eine Einführung in die Sprache. Außerdem geben wir die Aufgabenstellung für diese Diplomarbeit an und besprechen Begriffe, die in der Arbeit verwendet werden.

Programmcode zum Lesen und Schreiben von Daten wird für verschiedene SKilL Spezifikationen jeweils neu erzeugt. Das zweite Kapitel beschreibt den Aufbau und die Bedienung des Codegenerators.

Das dritte Kapitel behandelt die Benutzungsschnittstelle des Bindings und veranschaulicht die Verwendung anhand von Beispielen. Wir führen im Detail aus, wie die Aspekte der Objektorientierung der zu serialisierenden Daten abgebildet wurden.

Im vierten Kapitel beschreiben wir die Implementierung der Schnittstelle. Wir zeigen die Architektur des generierten Codes, erklären, wie die zu serialisierenden Daten im Speicher organisiert sind, und behandeln das Lesen und Schreiben von Binärdateien.

Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen und stellt Möglichkeiten zur Erweiterung der Implementierung vor.

1.2 Bisherige Anbindungen

Die Spezifikation von Skill ist in [Fel13] gegeben, teilweise mit Anleitungen oder Empfehlungen zur Implementierung. Der folgende Abschnitt gibt eine Einführung in die Sprache.

Im Rahmen der Diplomarbeit [Ung14] ist die Anbindung an Java entstanden. Die Arbeit untersucht auch die generelle Tauglichkeit der Sprache und wie die Bedienungsschnittstelle gestaltet werden kann. Die Bachelorarbeit [Prz14] untersucht Performance anhand der Implementierung der Anbindung an Ada. Es existiert außerdem eine Anbindung an Scala [ski].

1.3 Die Seriaisierungssprache SKilL

Die Arbeit [Fel13] definiert erstens eine Beschreibungssprache und zweitens das Format für die Serialisierung. In der Beschreibungssprache wird die Struktur der zu speichernden Daten definiert. Der Anwender muss eine Spezifikationsdatei in der Beschreibungssprache erstellen, um ein Binding generieren zu können. Das Format für die Serialisierung beschreibt den Aufbau der Binärdateien, die serialisierte Daten speichern. Die Sprachanbindungen müssen dieses Format lesen und schreiben können. Der Anwender programmiert dann gegen die generierte Schnittstelle, kommt mit den Daten in serialisierter Form also nicht in Berührung.

Im Folgenden werden Beschreibungssprache und Serialisierungsformat kurz vorgestellt. Die Beschreibungen sind nicht vollständig, sondern sollen nur jeweils die wichtigsten Konzepte und Besonderheiten erklären.

1.3.1 Beschreibungssprache

Mit der Beschreibungssprache [Fel13, §2 ff] wird die Struktur der zu serialisierenden Daten beschrieben. So genannte *Benutzertypen* bilden die Einheiten, deren Instanzen später von den Sprachanbindungen geschrieben und gelesen werden können. Ein *Benutzertyp* hat eine Menge von Feldern, die über ihren Namen identifiziert werden. Die möglichen Typen für diese Felder werden in den nächsten Unterpunkten behandelt.

Eingebaute Typen

Eingebaute Typen (*Built-In Types*[Fel13, §4.1]) beinhalten

- Typen für ganze Zahlen mit verschiedenen Wertebereichen
- Typen für Kommazahlen mit verschiedener Genauigkeit
- Strings
- Booleans
- Annotations, also Zeiger auf beliebige Benutzertypen

Zusammengesetzte Typen

Zusammengesetzte Typen werden verwendet, um Daten gleichen Typs zu gruppieren. Die Spezifikation verbietet, dass die enthaltenen Typen wiederum zusammengesetzte Typen sind. Sie definiert:

- Arrays fester Länge
- Arrays variabler Länge
- Listen

- Mengen
- Maps

Für Arrays fester Länge ist die Anzahl ihrer Elemente in der Typdefinition gegeben. Für Arrays variabler Länge, Listen, und Mengen kann die Anzahl der Elemente In jeder Instanz unterschiedlich sein. Mengen dürfen jedes Element nur ein mal beinhalten. Maps dürfen beliebig tief verschachtelt werden.

Vererbung

Die Beschreibungssprache, erlaubt es, zwischen Benutzertypen eine einfache Vererbungshierarchie zu definieren. Der Untertyp enthält alle Felder des Obertyps, definiert aber ggf. weitere.

Zeiger auf Benutzertypen

Die Beschreibungssprache erlaubt es, Zeiger auf Benutzertypen zu definieren. Ein Benutzertyp kann Zeiger auf andere Benutzertypen als eines seiner Felder enthalten. Jeder Untertyp des angegebenen Typen ist dann als Ziel der Referenz ebenfalls erlaubt.

1.3.2 Serialisierungsformat

Das Serialisierungsformat beschreibt, wie Instanzen von Benutzertypen serialisiert werden müssen, also den Aufbau der Binärdateien [Fel13, §6 f]. Wir gehen in diesem Abschnitt auf die wichtigsten Eigenschaften des Formates ein.

Typinformationen

Das Format speichert Typinformationen zusammen mit den Daten selbst [Fel13, §6.2]. Um den Speicherverbrauch der Binärdateien gering zu halten, werden Typinformationen aber nicht für jede Instanz, sondern nur ein mal pro Benutzertyp gespeichert.

Listing 1.1: Aufbau des Serialisierungsformates XML

```
[..] // Schemadefinition hier weggelassen
<Buch>
    <Titel>Am Anfang</Titel>
    <Autor>Amanda Abele</Autor>
    <Preis>9.90</Preis>
</Buch>
<Buch>
    <Titel>Mittig</Titel>
    <Autor>Doran Daumen</Autor>
    <Preis>19.90</Preis>
</Buch>
<Buch>
    <Titel>Schlussendlich</Titel>
    <Autor>Emil Ende</Autor>
    <Preis>29.90</Preis>
</Buch>
```

Listing 1.2: Aufbau des Serialisierungsformates SKilL

```
// Metadaten
Buch {
  Felder {
    string Titel;
    string Autor;
    f32 Preis;
  }
  Anzahl: 3
}

// Daten der Felder
1 -> "Am Anfang"  // Strings sind über IDs referenziert
2 -> "Mittig"
3 -> "Schlussendlich"
4 -> "Amanda Abele"
5 -> "Doran Daumen"
6 -> "Emil Ende"
9.90
19.90
29.90
```

Listing 1.1 und Listing 1.2 zeigt die Darstellung der gleichen Daten in XML und SKilL. Es sind für drei Buch-Instanzen jeweils Titel, Autor, und Preis hinterlegt. In XML gibt jede Instanz die Namen ihrer Felder mit an. In SKilL sind diese Metainformation nur einmal gespeichert. Für eine große Menge gleichartiger Daten wird die Größe der Metadaten vernachlässigbar, und das Format verbraucht insgesamt weniger Speicher. Die Darstellung ist vereinfacht und soll hier nur das Prinzip erklären.

Das Format ist *Aufwärtskompatibel* [Fel13, §1]. Das bedeutet, dass Unbekannte Typen (*unknown types* [Fel13, Glossary]), also Benutzertypen, die das Binding nicht kennt, genau so wie unbekannte Felder, beim Lesen übersprungen werden können.

Um den Speicherbedarf weiter zu senken, werden Identifikationsnummern für Benutzertypen, Instanzen, und Strings in der Binärdatei nicht explizit angegeben, sondern implizit über die Reihenfolge dieser Elemente bestimmt [Fel13, §6.3 f].

Write und Append

Das Format ist dafür ausgelegt, neue Instanzen an bestehende Dateien anzuhängen (*append*), ohne die Datei komplett neu schreiben zu müssen. Es ist sogar möglich, an bestehende Instanzen neue Felder anzuhängen. Die Operation *write* schreibt alle erzeugten Instanzen in eine Binärdatei, die Operation *append* schreibt nur die Instanzen, die seit dem letzten Schreiben oder Lesen hinzugekommen sind.

1.4 Aufgabenstellung

Aufgabenstellung dieser Arbeit war

- Entwickeln einer Sprachanbindung an die Programmiersprache *C*
- Enwickeln von Tests um die Kompatibilität zu bisherigen Sprachanbindungen nachzuweisen

Es wurde eine formale Spezifikation der Beschreibungssprache und des Serialisierungsformates zur Verfügung gestellt, außerdem die Implementierung der Anbindung an die Programmiersprache Scala inklusive Tests. Es existierte bereits ein Codegenerator, der Beschreibungssprache in eine Zwischendarstellung in Form von Javaklassen übersetzt. Dieser Generator durfte ebenfalls verwendet werden.

Eine weitere, optionale Aufgabenstelung war die Entwicklung einer Sprachanbindung an eine nicht statisch typisierte Sprache. Diese Aufgabe wurde nicht bearbeitet.

1.5 Begriffe

Beschreibungssprache, Spezifikationsdatei Die Beschreibungssprache ist in [Fel13, §2 f] definiert und wurde bereits in 1.3.1 eingeführt. Wir bezeichnen eine in der Beschreibungssprache verfasste Datei als *Spezifikationsdatei* oder kurz *Spezifikation*.

Serialisierungsformat, Binärdatei Das Serialisierungsformat bezeichnet die Form der serialisierten Daten, die in [Fel13, §6] ausgeführt ist. In dieser Arbeit bezeichnen wir eine Datei, die Daten dieser Form enthält als *Binärdatei*.

Generator, Sprachanbindung Als Generator bezeichnen wir ein Programm, das bei Eingabe einer Spezifikationsdatei Programmcode erzeugt, der die spezifizierten Daten lesen, bearbeiten und schreiben kann. Eine *Sprachanbindung*, oder kurz *Anbindung*, an eine Programmiersprache ist der Generator, der Programmcode in dieser Sprache erzeugt. Ist die Programmiersprache nicht angegeben, meinen wir in dieser Arbeit die Anbindung an die Programmiersprache *C*.

Binding Als *Binding* bezeichnen wir ein Programm, das durch eine Ausführung des Generators erzeugt wurde. Das Binding ist aus einer Spezifikation erzeugt, und kennt deren Benutzertypen.

Benutzertyp, Instanz Ein *user type* ??Glossary]Fel13 ist in einer Spezifikation durch eine *type declaration* beschrieben. Wir übernehmen dafür den deutschen Begriff *Benutzertyp* [Ung14, §1.3.1], oder nur *Typ* und verwenden ihn im Kontext der Spezifikation, der Ausführung des Generators, des Bindings, und der Binärdatei.

Eine *Instanz* bezeichnet die Instanz eines Benutzertyps im Kontext einer Binärdatei oder der Ausführung eines Bindings.

Feld Ein *Feld* bezeichnet sowohl eine Einheit eines Benutzertypes im Sinne von [Fel13, §3.4], als auch die entsprechenden Daten von Instanzen im Kontext einer Binärdatei oder Binding-Ausführung.

Storage Pool Wir übernehmen den Begriff *storage pool* [Fel13, §6.3], oder kurz *pool* als Konstrukt im Binding zum Speichern von Instanzen. Wir speichern hier außerdem Informationen über die Organisation dieser Instanzen in der Binärdatei, wie die Reihenfolge der Felder.

Obertyp, Untertyp, Basistyp Um Vererbungshierarchien zwischen Benutzertypen zu beschreiben, verwenden wir die Begriffe *Obertyp* für den vererbenden Typen, *Untertyp* für den erbenden Typen, und *Basistyp* für den Typ an oberster Stelle in der Vererbungshierarchie [Fel13, vgl. Glossary]. SKilL erlaubt nur einfache Vererbung, deswegen hat jeder Typ einen eindeutigen Basistypen.

Bekannte und unbekannte Typen Wir nennen einen Benutzertypen *dem Binding bekannt*, falls er in der Spezifikation vorkommt, aus der das Binding generiert wurde. Der Typ heißt *der Binärdatei bekannt*, falls er in der Binärdatei definiert ist. Ist ein Typ nicht *bekannt*, heißt er *unbekannt* (vgl. *unknown type* [Fel13, Glossary]).

Analog nennen wir ein Feld *bekannt*, falls sein Typ in der Spezifikation, bzw. in der Binärdatei dieses Feld enthält. Andernfalls heißt es *unbekannt*.

Typblock, Stringblock Daten in Binärdateien sind in *type blocks* und *string blocks* organisiert [Fel13, §6.2]. Wir verwenden dafür die Begriffe *Typblock* , bzw. *Stringblock*, oder kurz *Block*.

2 Generator

Der Generator erzeugt aus einer Spezifikationsdatei ein Binding, welches die in der Spezifikation definierte Benutzertypen lesen, erzeugen, bearbeiten, und schreiben kann. Dieses Kapitel behandelt die Benutzungsschnittstelle und die Funktionsweise des Generators.

2.1 Bedienung

Die Benutzungsschnittstelle des Generators für die Programmiersprache C orientiert sich stark an den Benutzungsschnittstellen für bereits existierende Generatoren für Scala und Ada.

Listing 2.1: Hilfetext für die Benutzung des Generators

```
usage:
 [options] skillPath outPath
Opitions:
-p prefix           Set a prefix for emitted code.
                    This is used for identifier names
                    in the generated code.
--unsafe            If this option is set, the generated binding
                    will not execute any type checks when modifying instances.
                    This improves performance.
```

Listing 2.1 zeigt die Benutzung des Generators. Als Parameter *skillPath* muss der Dateipfad zu einer Spezifikationsdatei übergeben weren, und als *outPath* der Pfad zum Verzeichnis, in dem das Binding erzeugt werden soll. Falls der Parameter *prefix* gesetzt wird, so wird dieser Wert jedem Bezeichner im generierten Code vorangestellt, sodass ein Programm kompiliert werden kann, das mehrere Bindings enthält. Mit dem Parameter *unsafe* werden Typüberprüfungen im Binding ausgeschaltet. Diese Überprüfungen sind in Abschnitt 4.2.1 ausgeführt. Diese Einstellung verbessert die Performance.

Die Anbindung unterstüzt für die Namen von Typen und Feldern nur ASCII Zeichen, weil die Namen im Binding als Bezeichner verwendet werden.

2.2 Funktionsweise

Der generierte Code muss sich je nach Spezifikation unterscheiden, weil wir die Benutzertypen aus der Spezifikation in Datentypen abbilden wollen. Teile des Codes werden sich also abhängig vom Aufbau der Benutzertypen unterscheiden. Aus diesem Grund ist die Verwendung von *Freemarker* [fre]

sinnvoll. Wir verwenden pro Datei, die generiert werden soll ein so genanntes *Template*, also eine Vorlage, in der Teile bei der Ausführung von *Freemarker* aus einem Datenmodell geladen werden. Die Arbeit zur Java Anbindung führt die Funktionsweise von Freemarker im Detail aus [Ung14, §4.2].

3 Benutzungsschnittstelle

Dieser Abschnitt behandelt die Benutzungsschnittstelle des Bindings. Die Schnittstelle wurde strikt von ihrer Implementierung getrennt. In diesem Abschnitt werden nur Typdefinitionen und Funktionen beschrieben, die für den Benutzer sichtbar sind.

3.1 Systemanforderungen

- Der Compiler, mit dem das Binding übersetzt wird, muss den C-Standard [c9903] einhalten. Wir verwenden daraus Zusicherungen über die Darstellung von *structs* im Speicher in Abschnitt 4.2.
- Wir verlangen, dass die Darstellung von float und double dem IEEE Standard entspricht [iee08] für das Lesen und Schreiben dieser Datentypen.
- Die *glib* Bibliothek [gli14] muss vorhanden sein.

3.2 SkillState

Das Binding muss folgende Funktionalität anbieten:

- Instanzen erzeugen
- Instanzen bearbeiten
- Instanzen aus einer Binärdatei lesen
- Instanzen in eine Binärdatei schreiben oder anhängen (append)

Es soll grundsätzlich möglich sein, mit mehreren Binärdateien parallel zu arbeiten, bzw. Instanzen verschiedenen Kontexten zuzuordnen. Deswegen definiert die Schnittstelle einen *SkillState*, dem Instanzen zugeordnet sind. Ein *SkillState* kann entweder leer oder aus einer Binärdatei erzeugt werden. Ein *SkillState*, der aus einer Datei erzeugt wurde, enthält alle Instanzen bekannter Typen aus der Datei.

Listing 3.1: Benutzungsschnittstelle - SkillState

```
typedef struct skill_state_struct *skill_state;

skill_state empty_skill_state ();
void delete_skill_state ( skill_state state );
skill_state skill_state_from_file ( char *file_path );
```

Nachdem ein *SkillState* in eine Binärdatei geschrieben wurde, wird die Datei Typdefinitionen zu allen bekannten Typen und Feldern enthalten, außerdem alle Instanzen aus dem *SkillState*. Ist ein *SkillState* aus einer Binärdatei erzeugt oder bereits in eine Binärdatei geschrieben, bleibt der Zustand dieser Datei im *SkillState* gespeichert. Nur in diesem Fall ist die Operation *append* möglich, um neue Instanzen hinzuzufügen, oder bestehenden Instanzen neue Felder hinzuzufügen.

Listing 3.2: Benutzungsschnittstelle - write und append

```
void write_to_file ( skill_state state, char *file_path );

// Benötigt keinen Dateipfad, weil aus Datei erzeugt oder schon in Datei geschrieben
void append_to_file ( skill_state state );
```

3.3 Benutzertypen

Für jeden Benutzertypen in einer Spezifikation wird das daraus erzeugte Binding einen entsprechenden Typen mit gleichem Namen anbieten. Um Instanzen erzeugen zu können, dient ein Konstruktor, der für jeden Typen generiert wird.

Listing 3.3: Spezifikation des Benutzertyp Person

```
Person {
    string name;
}
```

Listing 3.4 zeigt die Typdefinition und den Konstruktor für die Spezifikation aus Listing 3.3.

Listing 3.4: Benutzungsschnittstelle - Benutzertypen

```
typedef struct person_struct *person;

// Die Instanz wird dem übergebenen skill_state zugeordnet
person create_person ( skill_state this, char *name );
```

Wir verstecken die Felder der Benutzertypen und bieten für den Zugriff *getter* und *setter* für jedes Feld aus der Spezifikation. Weil es nicht möglich ist, die Methoden per Punktschreibweise direkt auf Instanzen aufzurufen (*person.get_name()*), übergeben wir die Instanz als Parameter.

Listing 3.5: Benutzungsschnittstelle - Getter und Setter

```
char *person_get_name ( person instance );
void person_set_name ( person instance, char *name );
```

Um alle Instanzen eines Typs zurückzugeben, verwenden wir die *GList* aus der Bibliothek *glib* als Implementierung einer Liste.

Listing 3.6: Benutzungsschnittstelle - Listen

```
// enthält 'person'-Instanzen
GList *get_person_instances ( skill_state this );
```

Listing 3.7 zeigt, wie mithilfe einer *for-Schleife* über alle in einer Liste enthaltenen Elemente iteriert werden kann.

Listing 3.7: Iterieren über alle Instanzen eines Typs

```
GList *persons = get_person_instances ( state );
GList *iterator;
for ( iterator = persons; iterator; iterator = iterator->next ) {
  person_get_name ( (person) iterator->data );
}
```

3.4 Vererbung

SKilL kann einen Benutzertypen als *Untertypen* eines anderen definieren. Der Untertyp ist eine Spezialisierung seines Obertyps und erbt dessen Felder, kann aber weitere hinzufügen.

Listing 3.8: Benutzertypen mit Vererbung - Spezifikation

```
Person {
    string name;
    Parent mother;
    Parent father;
}

Parent : Person {
    list<Person> children;
}
```

Wir erklären die Funktionsweise der Schnittstelle für Benutzertypen mit Vererbung anhand des Beispiels aus Listing 3.8, das den Typ *Parent* als Untertyp von *Person* einführt. Wir verwenden den abstrakten Typ *skill_type* in der Schnittstelle als Basistyp für alle Benutzertypen.

Listing 3.9: Benutzertypen mit Vererbung - Typdefinitionen

```
typedef struct skill_type_struct *skill_type;

// Erbt von 'skill_type'
typedef struct person_struct *person;

// Erbt von 'person'
typedef struct parent_struct *parent;
```

Listing 3.10 zeigt die *Getter* für Benutzertypen mit Vererbung. Als *person* Instanz kann jeweils auch eine *parent* Instanz übergeben werden. Die *Setter* sind nach dem gleichen Muster aufgebaut.

Listing 3.10: Benutzertypen mit Vererbung - Getter

```
// Kann auf jedem Untertypen von 'person' aufgerufen werden.
char *person_get_name ( person instance );
parent person_get_father ( person instance );
parent person_get_mother ( person instance );

// Kann nur auf 'parent' Instanzen aufgerufen werden.
GList *parent_get_children ( parent instance );
```

Um dem Benutzer Typkonvertierungen zu ersparen, erhält jeder Typ auch *getter* und *setter* für alle geerbten Felder, wie in Listing 3.11 angegeben.

Listing 3.11: Benutzertypen mit Vererbung - Getter für Felder von Obertypen

```
// Duplizierte Getter für Felder vom Obertyp 'person'.
char *parent_get_name ( parent instance );
parent parent_get_father ( parent instance );
parent parent_get_mother ( parent instance );
```

Wir bieten eine Liste der Instanzen aller bekannten Typen im *skill_state* über die funktion *get_all_instances*. Unbekannte Typen werden nicht unterstützt. Die zurückgegebene Liste von *person* Instanzen enthält auch alle Instanzen vom Untertyp *parent*. Um den Typ abfragen zu können, bietet die Schnittstelle für jeden Benutzertypen eine *instanceof* Funktion. Die Funktion gibt *true* zurück, falls die übergebene Instanz vom gefragten Typen oder einem seiner Untertypen ist.

Listing 3.12: Benutzertypen mit Vererbung - Listen und Instanceof-Funktionen

```
// Enthält alle Instanzen bekannter Typen.
GList *get_all_instances ( skill_state this );

// Enthält auch 'parent'-Instanzen
GList *get_person_instances ( skill_state this );
GList *get_parent_instances ( skill_state this );

bool instanceof_parent ( skill_type instance );
// 'true' auch für alle Untertypen von 'person'
bool instanceof_person ( skill_type instance );
```

Das Binding enthält nur einen Destruktor, dem Instanzen jeden Typs zu übergeben werden können. Konstruktoren sind für jeden Typen vorhanden, weil Untertypen ggf. neue Felder hinzufügen und damit andere Parameter benötigen.

Listing 3.13: Benutzertypen mit Vererbung - Konstruktoren und Destruktoren

```
person create_person ( skill_state this, char *name, parent father, parent mother );
parent create_parent ( skill_state this, GList *children, char *name, parent father, parent
    mother );

// Hier kann jeder Benutzertyp übergeben werden.
void delete_skill_type ( skill_type instance );
```

Wir zeigen in Listing 3.14, wie die *instanceof*-Funktionen verwendet werden können. Für den Zugriff auf das Feld *name* kann jeder Untertyp von *person* übergeben werden. Falls die Instanz vom Typ *parent* ist, gibt *instanceof_parent true* zurück, und wir können nach Typ *parent* konvertieren, um auf das Feld *children* zuzugreifen.

Listing 3.14: Benutzertypen mit Vererbung - Verwendung des Bindings

```
// Aus Skillstate laden
person p = [..]

// Jeder Untertyp von 'person' erlaubt
char *name = person_get_name ( p );

// Iterieren über die Kinder, nur falls vom Typ 'parent'
if ( instanceof_parent ( p ) ) {
    GList *children = parent_get_children ( (parent) p );
    GList *iter;
    for (iter = children; iter; iter=iter->next) {
        person child = (person) iter->data;
    }
}
```

Skill Typ	In C
i8	int8_t
i16	int16_t
i32	int32_t
i64	int64_t
v64	int64_t
f32	float
f64	double
bool	bool
string	char*
Annotation	skill_type
array (feste Länge)	GArray*
array (variable Länge)	GArray*
list	GList*
set	GHashTable*
map	GHashTable*

Tabelle 3.1: Darstellung der SKilL Datentypen in C

3.5 SKilL Datentypen

In [Fel13, §4] wird das Typsystem für Spezifikationsdateien erklärt. Diese Typen sind für Felder von Benutzertypen verwendet. Tabelle 3.1 gibt jeweils den verwendeten Datentyp im Binding an.

3.5.1 Zahlentypen und Boolean

Für die Datentypen für ganze Zahlen *i8* bis *i64* und *v64* verwenden wir *int8_t* bis *int64_t*[c9903, §7.18], die genau die geforderten Wertebereiche haben. Der Standard liefert auch einen Datentyp *bool* [c9903, §7.16].

3.5.2 Annotation

Eine *Annotation* ist ein Zeiger auf einen beliebigen Benutzertyp. Im Binding verwenden wir dafür den abstrakten Obertyp *skill_type*. Mithilfe der *instanceof*-Funktionen kann der genaue Typ bestimmt werden. Es ist nicht möglich, Referenzen auf unbekannte Typen darzustellen.

3.6 Zusammengesetzte Typen

Zusammengesetzte Typen sind *array*, *list*, *set* und *map*. Wir verwenden für die Darstellung Datentypen aus der *glib* [gli14]. Das Binding gibt den Speicher für diese Datentypen beim Löschen des *skill_state* frei, wenn sie in seinen Instanzen verwendet sind.

Listing 3.15: Zusammengesetzte Typen - Spezifikation

```
// v64 als Elemente dieser zusammengesetzten Typen
Int_Container {
  v64[3] const_arr;       // Array mit fester Länge.
  v64[] var_arr;          // Array mit variabler Länge.
                          // Zugriff auf Array-Elemente in konstanter Zeit.
  list<v64> list;         // Liste, Elemente sind geordnet
  set<v64> set;           // Set, enthält keine Elemente doppelt
  map<v64, v64, v64> map; // Map mit drei Basistypen
}
```

3.6.1 Array

Listing 3.16: Zusammengesetzte Typen - Arrays

```
// Parameter zum Erzeuge des Arrays:
//   -Null-Element am Ende?
//   -Elemente 0-initialisiert?
//   -Größe der Elemente in Bytes
GArray *new_array = g_array_new ( false, true, sizeof (int64_t) );

// Vergrößert ggf. das Array
g_array_append_val ( new_array, 1 );
g_array_append_val ( new_array, -1 );

// Wert an beliebiger Position überschreiben:
// Zugriff über Array-Name, Datentyp, Index
// Erstes Element hat den Index 0
g_array_nth ( new_array, int64_t, 0 ) = 2;
```

Der Datentyp *GArray* [gli14, Arrays] bietet konstante Zugriffszeiten auf beliebige Elemente und passt seine Größe flexibel an. Beim Erzeugen muss die Größe der Elemente in Bytes übergeben werden. Wird der Datentyp für ein Feld mit konstanter Länge verwendet, muss die Anzahl der Elemente übereinstimmen.

3.6.2 List

Für Listen (*GList*, [gli14, Doubly-Linked Lists]) müssen deren Elemente als Zeiger übergeben werden. Die Funktion *g_list_nth_data* erlaubt zwar Zugriff auf beliebige Elemente, ihre Laufzeit ist aber linear

in der Anzahl der Listenelemente. Eine Besonderheit des Datentyps ist, dass eine leere Liste als *null* Zeiger dargestellt wird.

Listing 3.17: Zusammengesetzte Typen - Listen

```
// Verwendet '0' als leere Liste
GList *list = 0;
int64_t value1 = 1;
int64_t value2 = -1;

// Elemente per Zeiger übergeben, Rückgabewert wieder zuweisen
list = g_list_append ( list, &value1 ); // hinten anhängen
list = g_list_prepend ( list, &value2 ); // vor dem ersten Element einfügen
```

3.6.3 Set und Map

Die *GHashTable* dient als Datentyp sowohl für *set* als auch für *map*. Der Datentyp speichert eine Schlüssel-Wert Zuordnung, kann aber auch als Menge verwendet werden, indem für Schlüssel und Wert immer die gleiche Variable übergeben wird [gli14, Hash Tables,Description]. Auf Schlüssel wird über eine Hashfunktion zugegriffen, die beim Erzeugen des Datentyps definiert werden muss. Wir verwenden die von der Bibliothek angebotenen Funktionen

- *g_int64_hash* und *g_int64_equal* für die *SKilL* Datentypen *i8* bis *i64*, *v64* und *bool*
- *g_str_hash* und *g_str_equal* für Strings
- *g_direct_hash* und *g_direct_equal* für Benutzertypen und *Annotation*

Wie bei Listen müssen Schlüssel und Werte als Zeiger übergeben werden.

Listing 3.18: Verwendung der GHashTable

```
// Verwendet als Menge
GHashTable *set = g_hash_table_new ( g_int64_hash, g_int64_equal );
int64_t value1 = 1;
int64_t value2 = -1;

// Elemente per Zeiger übergeben mit Schlüssel = Wert
g_hash_table_insert ( set, &value1, &value1 );
g_hash_table_insert ( set, &value2, &value2 );

// Kein Effekt, weil schon vorhanden
g_hash_table_insert ( set, &value1, &value1 );

// Verwendet als Map
GHashTable *map = g_hash_table_new ( g_int64_hash, g_int64_equal );
int64_t value1 = 1;
int64_t value2 = -1;

// Dem Schlüssel '1' den Wert '-1' zuordnen
g_hash_table_insert ( set, &value1, &value2 );
```

Der Datentyp kann Schlüsselmenge und Wertemenge als *GList* zurückgeben um über alle Elemente zu iterieren.

Listing 3.19: Iterieren über die Wertemenge einer Map

```
int_container container = [..] // Aus Skillstate laden
GHashTable *set = int_container_get_set ( container );

// Zurückgeben der Wertemenge als GList
GList *value_list = g_hash_table_get_values ( set );
GList *iter;
for ( iter = value_list; iter; iter = iter->next ) {
  int64_t *current_value = (int64_t*) iter->data;
}
g_list_free ( value_list );
```

Der Datentyp *map* muss im Binding nach einem induktiven Prinzip aufgebaut werden, falls er mit mehr als zwei Basistypen verwendet wird. Eine *map* mit genau zwei Basistypen $map < typ_1, typ_2 >$ wird dargestellt als *GHashTable*, die typ_1 Variablen typ_2 Variablen zuordnet (Schlüssel-Wert Paare). Eine *map* mit den Basistypen $typ_1, \ldots, typ_{n+1}$ wird zur *GHashTable*, die typ_1 Variablen eine Map mit den Basistypen $typ_2, \ldots, typ_{n+1}$ zuordnet. Die Datentypen müssen denen aus der Spezifikation entsprechen.

Listing 3.20: Verwendung von Maps mit mehr als zwei Basistypen

```
// drei Basistypen: map<v64,v64,v64>

GHashTable *map = g_hash_table_new ( g_int64_hash, g_int64_equal );
GHashTable *nested_map = g_hash_table_new ( g_int64_hash, g_int64_equal );

int64_t one = 1;
int64_t two = 2;
int64_t three = 3;

// Geschachtelte Map als Wert für den Schlüssel '1'
g_hash_table_insert ( map, &one, nested_map );

// Schlüssel '2' -> Wert '3'
g_hash_table_insert ( nested_map, &two, &three );
```

4 Implementierung

4.1 Architektur

Der Programmcode des Bindings gliedert sich in Einheiten mit Attributen und Methoden, die wir im Sinne der Objektorientierung als *Klassen* bezeichnen. Die Programmiersprache C unterstützt die Objektorientierung nicht, deswegen verwenden wir folgende Konventionen:

- Jede Klasse wird im Binding repräsentiert in Form einer *header*-Datei (Dateiendung *.h*) und einer *source*-Datei (Dateiendung *.c*). Die *header*-Datei enthält ein *struct* mit den öffentlichen Attributen, das wie die Klasse benannt ist, und Funktionsdeklarationen für die öffentlichen Methoden. Die *source*-Datei enthält die Implementierung dieser Funktionen. Sie darf in anderen Klassen nicht eingebunden werden.
- Öffentliche Methoden einer Klasse sind Funktionen mit dem Namensschema *<Klassenname>_<Methodenname>*. Die Instanz wird immer als erster Parameter übergeben. Wir haben bewusst darauf verzichtet, einfache Datenzugriffe in *getter* und *setter* zu kapseln, um die Lesbarkeit des Codes zu verbessern. Wir schreiben z.B. *pool->declaration->super_type* statt *type_declaration_get_super_type (storage_pool_get_declaration (pool))*
- Jede Klasse hat einen Konstruktor und einen Destruktor *<Klassenname>_new* und *<Klassenname>_destroy*. Der Konstruktor allokiert Speicher auf dem Heap und initialisiert alle Felder des *struct*. Der Destruktor gibt den Speicher der übergebenen Instanz frei.

Abbildung 4.1 zeigt die Klassen zum Lesen und Schreiben von Binärdateien. Die Klassen *binary_reader* und *binary_writer* lesen und schreiben auf unterster Ebene, also einfache Datentypen wie Integer, Floats und Strings. Die Klassen *reader* und *writer* lesen, bzw. schreiben ganze Binärdateien, also Stringblöcke, Typinformationen und Felddaten. Die Abschnitte 4.4 und 4.5 behandeln Details zum Lesen und Schreiben.

Abbildung 4.2 enthält die Modellklassen im Binding in UML-Darstellung. Bekannte Benutzertypen sind im Binding als Klassen vorhanden, die von *skill_type* erben, sind aber im Diagramm nicht dargestellt.

4.1.1 Skill_State

Die Klasse *skill_state* dient als Container für Typinformationen, Instanzen und ggf. Informationen zur Binärdatei. Falls der *skill_state* aus einer Binärdatei erzeugt wurde oder bereits geschrieben wurde, enthält das Attribut *filename* den Namen dieser Datei. Der *skill_state* hält außerdem eine Referenz auf den *string_access* und einen *storage_pool* für jeden bekannten Benutzertypen.

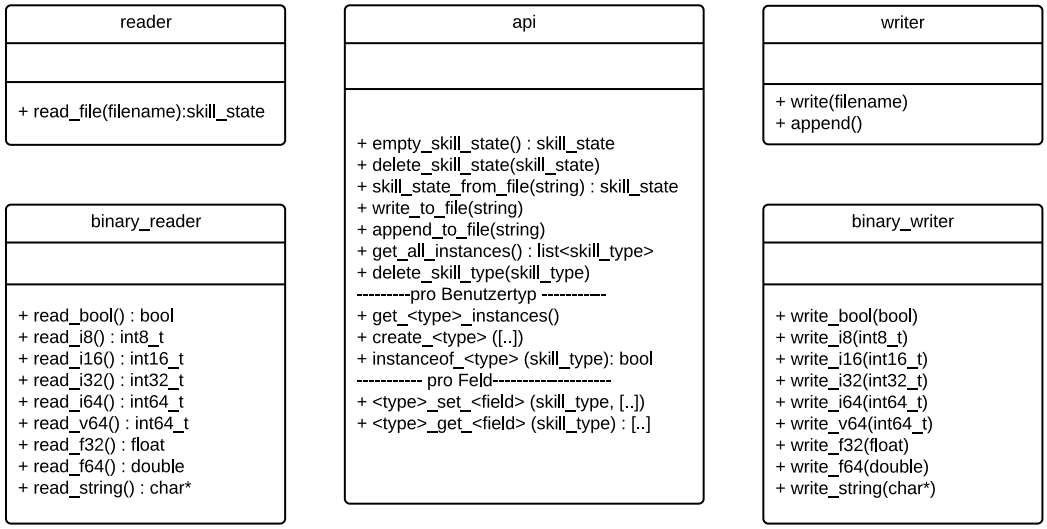

Abbildung 4.1: Klassen zum Lesen und Schreiben im Binding

4.1.2 String_Access

Strings sind in Binärdateien in Form von *Stringblöcken* organisiert [Fel13, §6.2]. Sie sind über eine Id identifiziert. Deswegen brauchen wir beim Schreiben der Datei eine Zuordnung $String \rightarrow Id$, da wir an dieser Stelle nicht den String selbst einfügen, und beim Lesen eine Zurdnung $Id \rightarrow String$. Die Klasse *string_access* hält deswegen zwei *maps*, *id_by_string* und *string_by_id* als Attribute. Wir kapseln den Zugriff auf die *maps* in Methoden, wie in Listing 4.1 gezeigt.

Listing 4.1: Methoden der Klasse String_Access

```
int64_t get_id_by_string ( string_access this, char *string );
char *get_string_by_id ( string_access this, int64_t id );
GList *get_all_strings ( string_access this );
void add_string ( string_access this, char *string );
```

Somit stellen wir sicher, dass beide *maps* zueinander passen, also die Schlüsselmenge der einen immer gleich der Wertemenge der anderen ist. Die Methode *add_string* fügt den String nur dann hinzu, falls er nicht schon vorhanden ist.

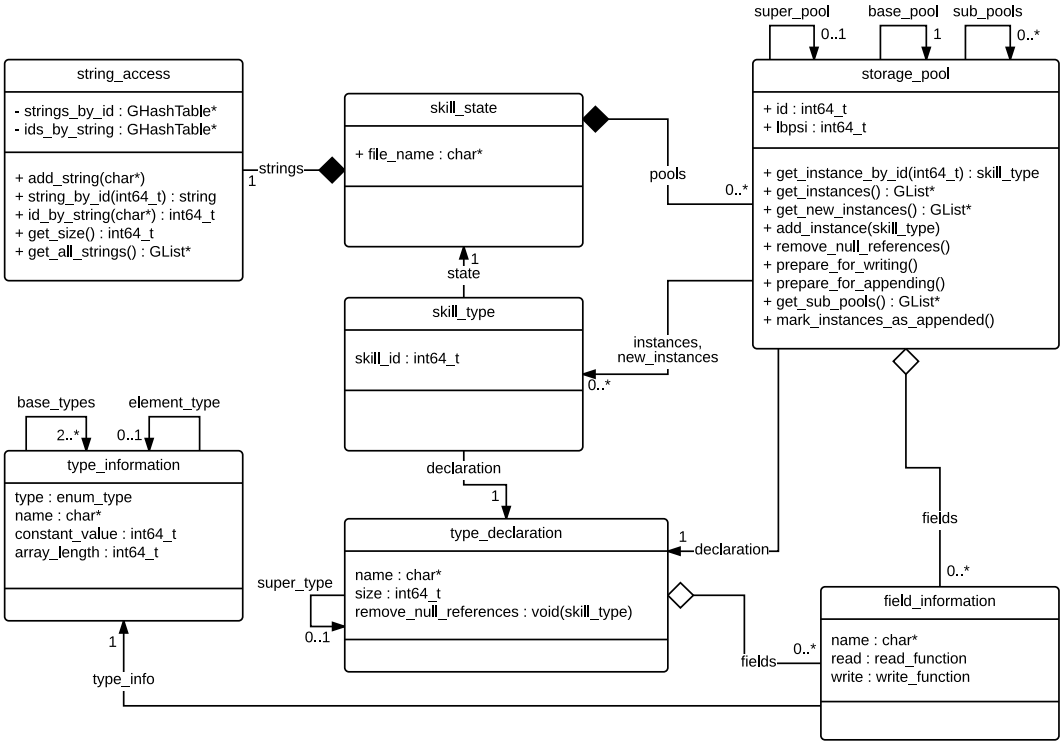

Abbildung 4.2: Klassendiagramm der Modellklassen im Binding

4.1.3 Skill_Type

Die Klasse *skill_type* dient als Basisklasse für alle Benutzertypen. Jede Instanz hält eine Referenz auf den *skill_state*, dem sie zugeordnet ist und ggf. die Id, die sie in der Binärdatei hat. Außerdem ist die *type_declaration* referenziert, die den Benutzertyp beschreibt.

4.1.4 Type_Declaration

Die Klasse *type_declaration* beschreibt die Struktur eines Benutzertypen. Name und Obertyp sind direkt aus der Spezifikation entnommen. Außerdem speichert die Klasse die Anzahl Bytes, die eine Instanz des Typs im Speicher belegt. Die Felder des Benutzertypen sind als *field_information* referenziert.

4.1.5 Field_Information

Die Klasse *field_information* beschreibt ein Feld eines Benutzertypen. Sie hält den Namen und ggf. zusätzliche Informationen in Form einer *type_information*. Außerdem Funktionen um Felddaten für dieses Feld zu lesen und zu schreiben.

4.1.6 Type_Information

Die Klasse *type_information* beschreibt den Datentyp eines Feldes. Sie hat die Attribute:

- Name des Benutzertyps, falls das Feld einen Zeiger auf einen Benutzertypen enthält
- Den konstanten Wert, falls das Feld eine Konstante ist.
- Den Datentyp der Elemente für *array*, *set* oder *list*
- Die Länge des Arrays, falls das Feld ein *array* fester Länge enthält
- Eine Liste von Basistypen, falls das Feld eine *map* enthält

Nicht zutreffende Attribute werden jeweils auf *null* gesetzt.

4.1.7 Storage_Pool

Für jeden bekannten Benutzertypen im Binding hält der *skill_state* eine *storage_pool* Instanz. Hier sind alle Instanzen dieses Typs referenziert, außerdem die *type_declaration* für diesen Typ.

4.2 Darstellung von Benutzertypen

Wir erklären in diesem Abschnitt den Aufbau der *structs* für Benutzertypen, zeigen, wie der Zugriff auf deren Daten funktioniert (*getter* und *setter*), und begründen, warum Typkonvertierungen möglich sind. Die Klasse *skill_type* ist der abstrakte Obertyp für alle Benutzertypen. Sie definiert eine Id und Referenzen auf *skill_state* und *type_declaration*.

Listing 4.2: Der abstrakte Typ Skill_Type

```
typedef struct skill_type_struct {
    int64_t skill_id;
    type_declaration declaration;
    skill_state state;
} skill_type_struct;
```

Wir erklären den Aufbau der *structs* für Benutzertypen anhand der Spezifikation aus Listing 3.8. Die Vererbungshierarchie ihrer Benutzertypen ist in Abbildung 4.3 verdeutlicht.

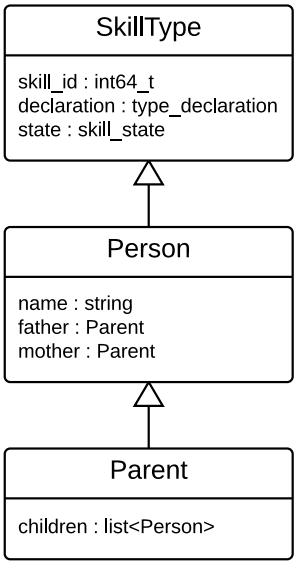

Abbildung 4.3: Vererbungshierarchie der Benutzertypen Person und Parent

Wir stellen Untertypen als *structs* dar mit dem Obertypen als ersten Eintrag. Dabei ist entscheidend, dass kein Zeiger auf den Obertyp verwendet wird, sondern das entsprechende *struct*. Listing 4.3 zeigt den Aufbau der Typen im Binding nach diesem Prinzip.

Listing 4.3: Interne Darstellung der Benutzertypen

```
// Benutzertyp 'person' mit Obertyp 'skill_type'
typedef struct person_struct {
    skill_type_struct _super_type; // Struct des Obertyp
    char *_name;
    parent _father;
    parent _mother;
} person_struct;

// Benutzertyp 'parent' mit Obertyp 'person'
typedef struct parent_struct {
    person_struct _super_type; // Struct des Obertyp
    GList *_children;
} parent_struct;
```

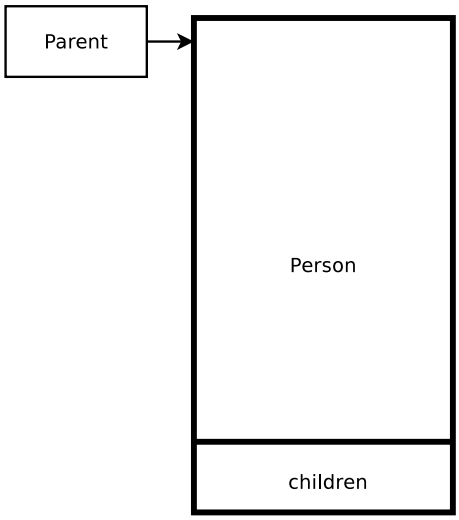

Abbildung 4.4: Eine Parent Instanz im Speicher

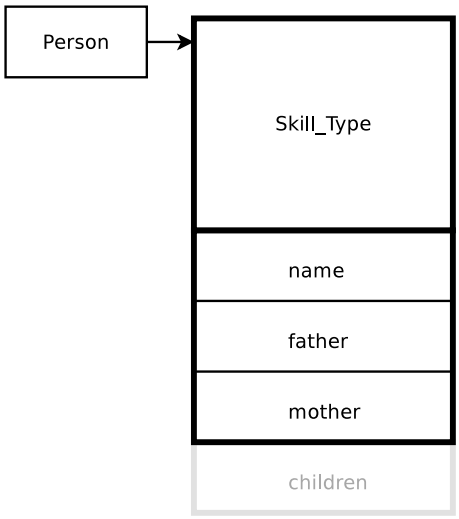

Abbildung 4.5: Typkonvertierung von Parent zu Person

Der Standard garantiert, dass ein Zeiger auf ein *struct* nach passender Konvertierung immer zu einem Zeiger auf den ersten Eintrag des *struct* wird[c9903, §6.7.2.1.13]. Nach unserer Konstruktion ist der erste Eintrag immer die Darstellung des Obertyps.

Abbildung 4.4 veranschaulicht die Darstellung einer *Parent* Instanz im Speicher. Nach einer Typkonvertierung zu *Person* bleibt die Adresse gleich, nur die Sicht auf die Daten (Offsets der Felder) ist angepasst. Das *Person* Feld hat die gleiche Adresse wie die *Parent*-Instanz, deswegen sind die Zugriffe auf die Felder auch nach der Typkonvertierung nach *Person* korrekt (Abbildung 4.5). Die Sicht nach der Typkonvertierung nach *skill_type* ist schließlich in Abbildung 4.6 verdeutlicht.

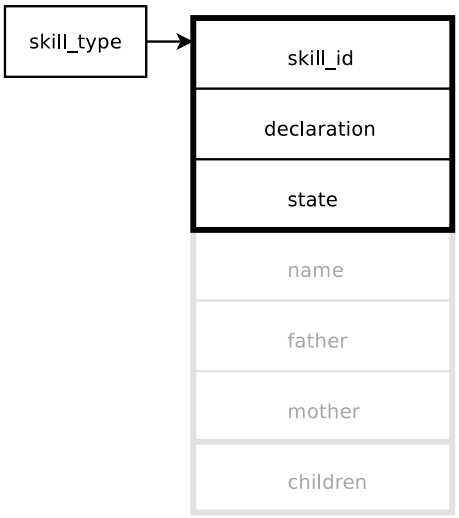

Abbildung 4.6: Typkonvertierung von Parent zu SkillType

Wir halten während der Ausführung des Bindings *storage pools* für alle Benutzertypen im Speicher. Jeder Pool referenziert den Pool seines Obertyps und eine Liste von Pools für Untertypen. Referenzen auf Instanzen sind in der Binärdatei über die Id der Instanz angegeben, deswegen benötigen wir beim Lesen von Binärdateien eine effiziente Zuordnung von der Id zur Instanz. Ids sind eindeutig innerhalb eines Basistyps [Fel13, vgl.§6.3], deswegen speichern wir alle Instanzen des Basistyps, inklusive Instanzen erbender Typen in einem Array im Pool des Basistyps. Wir können dann eine Id als Index im Array interpretieren, um die zugehörige Instanz zu finden.

Andererseits benötigen wir eine schnelle Möglichkeit, alle Instanzen eines Untertypen zu finden. Deswegen hält jeder Untertyp ein Array, das nur Instanzen dieses Typs enthält. Abbildung 4.7 zeigt die Organisation von Pools und Instanzen im Speicher anhand der Spezifikation aus Listing 4.4

Listing 4.4: Spezifikation der Benutzertypen Person, Parent und Other

```
Person {
    string name;
    Parent mother;
    Parent father;
}

Parent : Person {
    list<Person> children;
}

// Weiterer Typ 'Other' ohne Felder
Other {
}
```

Im Beispiel wurden Instanzen in folgender Reihenfolge erzeugt:

- Person
- Parent
- Other
- Parent
- Other

Instanzen von Untertypen werden immer von mehreren Pools referenziert, auch von allen Pools von Obertypen. Die Darstellung erlaubt das effiziente Iterieren über Instanzen eines Typs und Zugriff auf eine Instanz über ihre Id.

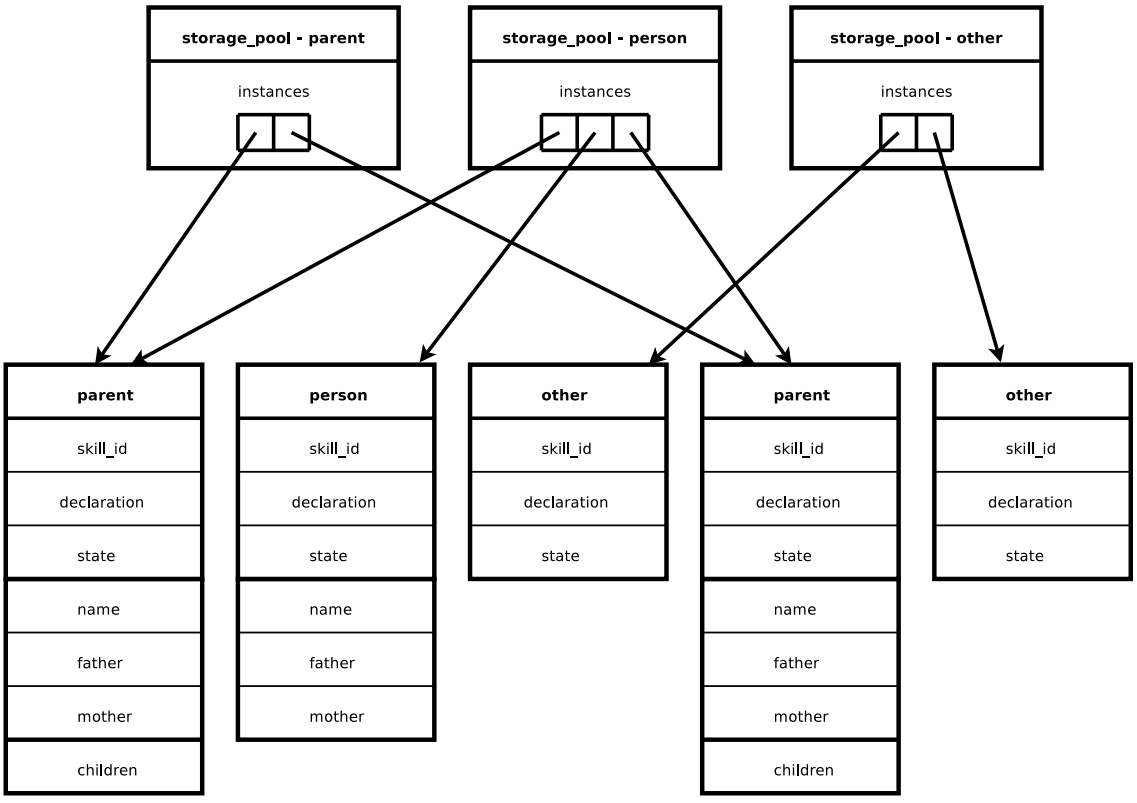

Abbildung 4.7: Storage Pools und Instanzen im Speicher

4.2.1 Instanceof Abfragen

Jede Instanz hält eine Referenz auf die *declaration*, die ihren Typ bestimmt. Die Implementierung der *instanceof* Funktionen kann deswegen diese *declaration* mit dem gefragten Typen abgleichen. Die Funktion muss auch *true* zurückgeben, falls irgendeiner der Obertypen übereinstimmt. Listing 4.5 zeigt die Implementierung anhand der Spezifikation aus Listing 3.8.

Listing 4.5: Instanceof-Funktionen

```
bool instanceof_person ( skill_type instance ) {
    type_declaration declaration = instance->state->person->declaration;

    // Jeder Obertyp der übergebenen Instanz muss überprüft werden
    type_declaration super_declaration = instance->declaration;
    while ( super_declaration ) { // Abbruch, wenn der supertyp 'null' ist
        if ( super_declaration == declaration ) {
            return true;
        }
        super_declaration = super_declaration->super_type;
    }
    return false;
}
```

4.3 Konventionen

Im verbleibenden Teil dieses Kapitels verwenden wir abkürzende Schreibweisen um Programmcode lesbarer und kompakter zu gestalten:

- Wir verwenden eine Punktnotation für den Aufruf von Methoden und den Zugriff auf Attribute, wie sie manche höhere Programmiersprachen anbieten (*instanz.attribut* oder *instanz.methode()*).
- Wir verwenden eine vereinfachte Darstellung, um über Listen zu iterieren (*GList* aus der glib [gli14, Doubly-Linked Lists]). Die Darstellung orientiert sich an erweiterten for-Schleifen, wie sie *Java* und *C#* anbieten. Listing 4.6 zeigt die Verwendung für eine Liste *pools* mit Elementen vom Typ *storage_pool*.

Listing 4.6: Vereinfachte Darstellung von for-Schleifen

```
// Vereinfachte Schreibweise:
for ( storage_pool pool in pools ) {
    [..] // Verwenden der Laufvariablen 'pool'
}

// Tatsächliche Syntax im Code:
GList *iterator;
storage_pool pool;
for ( iterator = pools; iterator; iterator = iterator->next ) {
    pool = (storage_pool) iterator->data;
    [..] // Verwenden der Laufvariablen 'pool'
}
```

- Für Arrays verwendet das Binding den Datentyp *GArray*. Wir schreiben kurz *array[i]* für den Zugriff auf das *i-te* Element statt *g_array_index (array, type, i)*.
- Für den Datentyp *GHashTable* schreiben wir verkürzt *map.get_values()* und *map.get_keys()* für die Funktionen *g_hash_table_get_values* und *g_hash_table_get_keys*, die eine Liste der Schlüssel, bzw. Werte zurückgeben. Die Funktion, um den hinterlegten Wert für einen Schlüssel abzurufen, ist *g_hash_table_lookup*. Wir schreiben dafür verkürzt *map[key]*.
- Typkonvertierungen sind in vielen Fällen nicht angegeben
- Funktionsparameter werden manchmal weggelassen, wenn sie für das Verständnis nicht relevant sind.
- Das Prüfen von Werten auf *null* ist manchmal weggelassen.

4.4 Lesen

Das grundlegende Vorgehen beim Lesen von Binärdateien ist in [Fel13, §7] beschrieben. Das Binding unterstützt kein *lazy loading* [Fel13, vgl.§6.2.2], Es werden also alle Strings und Instanzen bekannter Typen im Speicher erzeugt. Das Lesen besteht aus den Schritten:

- Stringblock lesen
- Typinformation lesen
- Instanzen allokieren
- Felddaten lesen und den Instanzen zuweisen

Diese Schritte werden wiederholt bis das Ende der Datei erreicht wird.

4.4.1 Stringblöcke lesen

Der Aufbau eines *Stringblocks* ist in [Fel13, Abb.2] veranschaulicht. Um Strings zu lesen, muss das Binding die Anzahl der Strings im Block lesen, anschließend entsprechend viele Offsetwerte und schließlich die Strings selbst, deren Längen anhand der Offsets bestimmt werden:

Listing 4.7: Lesen eines Stringblocks

```
int number_of_strings = read_v64 ();
int offsets[number_of_strings];
for ( int i = 0; i < number_of_strings; i++ ) {
    offsets[i] = read_i32 ();
}

// Länge des Strings ist jeweils Offset minus vorherigem Offset.
int previous_offset = 0;
for ( int i = 0; i < number_of_strings; i++ ) {
    char *new_string = read_string ( offsets[i] - previous_offset );
    strings.add_string(new_string);
    previous_offset = offsets[i];
}
```

An dieser Stelle erzeugt das Binding eine *string_access* Instanz, die im *skill_state* referenziert wird. Für folgende Typblöcke wird der *string_access* Strings per Id zurückgeben können.

4.4.2 Typinformation überprüfen

Ein Typblock beginnt jeweils mit Typinformation zu jedem instanziierten Benutzertypen im Block. Die Benutzertypen werden über ihren Namen identifiziert. Typinformationen sind in zwei Kategorien einzuteilen.

- Erstens den Namen des Obertyps und Namen und Datentypen von Feldern. Diese Informationen sind bereits in der Spezifikation festgelegt, also zur Zeit der Erstellung des Bindings bekannt und werden beim Lesen lediglich mit den Daten aus der Binärdatei abgeglichen. Werden hier Unterschiede festgestellt, werden sie auf der Konsole ausgegeben und die Programmausführung wird abgebrochen.

  Unbekannte Benutzertypen werden übersprungen, genau so wie unbekannte Felder bekannter Benutzertypen.

- Zweitens Informationen spezifisch für diese Binärdatei:
  - Anzahl der Instanzen
  - Reihenfolge der Felder
  - ggf. der *local base pool start index* [Fel13, vgl.§6.2.2]

Wir verwenden eine einfache Datenstruktur um diese Informationen zwischenzuspeichern, da sie fürs Lesen der Felddaten gebraucht werden. Die Reihenfolge, in der die Felder in der Spezifikationsdatei aufgelistet sind, ist nicht bindend, kann zwischen Binärdateien variieren und von der Reihenfolge aus der Spezifikation abweichen.

4.4.3 Instanzen erzeugen

Um neue Instanzen zu allokieren, muss für jede Instanz der exakte Typ bestimmt werden, damit die Größe einer Instanz in Bytes bekannt ist. Die *instance*-Arrays der Pools enthalten ggf. bereits Instanzen aus zuvor gelesenen Typblöcken. In diesem Fall werden sie vergrößert. Listing **??** zeigt die Implementierung im Binding. Wir sortieren Instanzen von Untertypen hinter Instanzen des exakten Typs ein.

Listing 4.8: Instanzen erzeugen

```
int count = read_information.count; // Information aus der Binärdatei
int sub_instance_count = 0; // Bestimme Anzahl Instanzen von Subtypen
for ( storage_pool sub_pool in pool.sub_pools ) {
    sub_instance_count += sub_pool.number_of_instances;
}

int old_count = pool.instances.size;
pool.instances.size += count; // Array des Pools vergrö ß ern
for ( int i = old_count; i < old_count + count - sub_instance_count; i++ ) {
  // Speicher für neue Instanzen einzeln allokieren,
  // sodass Instanzen einzeln gelöscht werden können.
    pool.instances[i] = malloc ( pool.declaration.size );
}

// Verwende für Pools von Untertypen die Reihenfolge, in der sie
// In der Binärdatei vorkommen
for ( storage_pool sub_pool in read_info.subtype_order ) {
  // Rekursiver Aufruf für Untertypen
    create_sub_pool_instances ( state, pool.instances );
}
```

4.4.4 Felddaten lesen

Beim Erzeugen des Bindings ist der Datentyp für jedes Feld bereits bekannt. Das Binding stellt für jedes Feld eine Funktion bereit, die Felddaten dieses Feldes liest, und den gelesenen Wert in einer Instanz setzt. Diese *read*-Funktion hat die Parameter

- Der *skill_state*, dem die erzeugt Instanz zugeordnet wird, um Zeiger auf Benutzertypen auflösen zu können
- Den *string_access*, um Strings über ihre Id zu bekommen
- Die Instanz selbst, in der der gelesene Wert gesetzt werden soll

Listing 4.9: Deklaration der read-Funktion

```
typedef void read_function ( skill_state, string_access, skill_type );
```

Diese Funktionen werden als Attribute der Klasse *field_information* gespeichert.

Die Implementierung der *read*-Funktion muss den zum Feld passenden Datentypen lesen und der übergebenen Instanz zuweisen. Listing 4.10 veranschaulicht die Funktionsweise anhand der Spezifikation von *Person* aus Listing 3.3.

Listing 4.10: Funktionsweise der read-Funktionen

```
void person_read_name ( skill_state state, string_access strings, skill_type instance ) {
  // Strings sind über ihre Id angegeben
  int64_t string_id = binary_reader.read_v64 ();
    ( (person) instance )->name = strings.string_by_id ( string_id );
}
```

Für Felder, die Zahlen oder *boolean* enthalten, wird die passende Funktion aus dem *binary_reader* aufgerufen:

- *read_i8* bis *read_i64*
- *read_v64*
- *read_f32* und *read_f64*
- *read_bool*

Listing 4.11: Spezifikation mit Annotation und Zeiger auf Benutzertyp

```
Test {
    annotation f; // Kann jeden bekannten Benutzertypen referenzieren
    person p;     // Benutzertyp bekannt bei der Erstellung des Bindings
}
```

Für *Annotations* muss zuerst der Typ der referenzierten Instanz bestimmt werden, um die Instanz über ihre Id zu bestimmen. Listing 4.12 zeigt die Implementierung am Beispiel der Spezifikation aus Listing 4.11.

Listing 4.12: Lesen von Annotations

```
void test_read_f ( skill_state state, string_access strings, skill_type instance ) {
  // Zieltyp ist als String angegeben
    int64_t type_name = strings.string_by_id ( binary_reader.read_v64 () );
    storage_pool target_pool = state.pools[type_name];

    // Ziel-Instanz über ihre Id bestimmen
    int64_t target_id = binary_reader.read_v64 ();
    instance.f = target_pool.get_instance_by_id ( target_id );
}
```

Für Zeiger auf Benutzertypen ist der Typ des Ziels beim Erstellen des Bindings bereits bekannt und sein Pool ist in der Funktion schon angegeben. Listing 4.13 zeigt die Implementierung.

Listing 4.13: Lesen von Referenzen auf Benutzertypen

```
void test_read_p ( skill_state state, string_access strings, skill_type instance ) {
    storage_pool target_pool = state.person;

    // Ziel-Instanz über ihre Id bestimmen
    int64_t target_id = binary_reader.read_v64 ();
    instance.p = target_pool.get_instance_by_id ( target_id );
}
```

Wir beschreiben das Lesen von zusammengesetzten Typen anhand der Spezifikation aus Listing 3.15. Elemente der zusammengesetzten Typen sind Zahlen, *boolean*, Strings, *annotation* oder Zeiger auf Benutzertypen. Sie werden wie oben beschrieben gelesen.

Listing 4.14: Lesen von zusammengesetzten Typen

```
// Für Arrays mit variabler Länge: zuerst Länge bestimmen
int64_t length = binary_reader.read_v64 ();
GArray *var_array = g_array_new ( length );
[..] // Einzelne Elemente lesen

// Für Arrays konstanter Länge ist diese hart in die read-Funktion kodiert.
GArray *var_array = g_array_new ( 3 );
[..] // Einzelne Elemente lesen

// Lesen von Listen
int64_t length = binary_reader.read_v64 ();
GList *list = 0; // Leere Liste entspricht null-Zeiger
for ( int i = 0; i < length; i++ ) {
  list = g_list_append ( list, binary_reader.read_v64 () );
}

// Lesen von Mengen
int64_t size = binary_reader.read_v64 ();
GHashTable *set = g_hash_table_new ( g_int64_hash, g_int64_equal );
for ( int i = 0; i < size; i++ ) {
    current_value = malloc ( sizeof ( int64_t ) );
        *current_value = read_i64 ( buffer );
        g_hash_table_insert ( set, current_value, current_value );
    }
}
```

Ganzzahlen, Kommazahlen, und Boolean Werte müssen für *list*, *set* und *map* auf dem Heap allokiert werden, da die dafür vorgesehenen Typen *GList* und *GHashTable* Zeiger speichern. Der zusammengesetzte Typ *map* muss gesondert behandelt werden, da er mehr als zwei Basistypen haben kann. Hat die *map* genau zwei Basistypen, können Schlüssel-Wert Paare nacheinander gelesen werden. Für *maps* mit drei und mehr Basistypen muss ggf. zu einem Schlüssel eine neue *map* erzeugt werden. Der Generator erzeugt eine Funktion für jeden Basistypen der *map*, außer dem letzten.

Listing 4.15: Lesen von Maps

```
// Liest Map mit einfachen Schlüssel-Wert Paaren (v64->v64)
GHashTable container_read_map_nested_1 ([..]) {
    GHashTable *result = g_hash_table_new ([..]);
    int64_t length = read_v64 ();
    int64_t *key;
    int64_t *value;
    for ( int i = 0; i < length; i++ ) {
        *key = binary_reader.read_v64 ();
        *value = binary_reader.read_v64 ();
        result.insert ( key, value );
    }
    return result;
}

// Liest Map mit geschachtelten Maps als Werten
GHashTable container_read_map_nested_0 ([..]) {
    GHashTable *result = g_hash_table_new ([..]);
    int64_t length = read_v64 ();
    int64_t *key;
    for ( int i = 0; i < length; i++ ) {
        *key = binary_reader.read_v64 ();
        // Eigener Funktionsaufruf, um geschachtelte Map zu lesen
        result.insert ( key, container_read_map_nested_1 ([..]) );
    }
    return result;
}
```

Bevor Felddaten gelesen werden, sind bereits alle Instanzen allokiert wie in Abschnitt 4.4.3 beschrieben.

Ein Typblock kann Felder definieren, die bereits in früheren Typblöcken vorkamen. Diese Felder sind in einer Liste im *storage pool* dieses Typs gespeichert. Es ist möglich, dass ein Typblock keine neuen Instanzen, sondern nur neue Felder für bestehende Instanzen hinzufügt. Gibt es neue Instanzen, so muss der Block mindestens alle Felder aus früheren Blöcken angeben, kann aber weitere hinzufügen. Bereits zuvor definierte Felder müssen in der Binärdatei vor neuen Feldern stehen, und müssen die zuvor verwendete Reihenfolge einhalten, was wir beim Lesen überprüfen.

Falls der Typblock sowohl neue Felder, als auch neue Instanzen hinzufügt, stehen die Daten von bereits früher definierten Feldern neuer Instanzen vor den Daten von neuen Feldern [Fel13, vgl.§6.2.2 Effects of Appending]. Daten neuer Felder müssen ggf. auch Instanzen aus früheren Typblöcken abdecken. Das Vorgehen, um Felddaten zu lesen, ist:

- Lesen der Felddaten für alle bereits früher definierten Felder, nur für neue Instanzen
- Lesen der Felddaten für alle neuen Felder für alle Instanzen (aus früheren Blöcken und aus diesem Block)

Listing 4.16 zeigt, wie die *read*-Funktionen aufgerufen werden, um Felddaten zu lesen und den Instanzen zuzuordnen.

Listing 4.16: Felddaten lesen

```
// 1. Lese Daten von früher definierten Feldern
// Die Liste 'pool.fields' enthält nur Felder aus früheren Blöcken
for ( field_information field_info in pool.fields ) {

    // Index der ersten neuen Instanz.
    // Im Array des Pools sind neue Instanzen bereits allokiert.
    int64_t start_index = pool.instances.len - field_info.count;
    for ( int i = start; i < pool.instances.len; i++ ) {
        field_info.read_method ( [..], pool.instances[i] );
    }
}

// 2. Lese Daten von neuen Feldern
// für alte und neue Instanzen
// Die Liste 'read_information.new_fields' enthält nur neue Felder
for ( field_info in read_information.new_fields ) {
    for ( int i = 0; i < pool.instances.len; i++ ) {
        field_info.read_method ( [..], pool.instances[i] );
    }
}
```

4.5 Schreiben

Das Vorgehen beim Schreiben von Binärdateien wird in [Fel13, §6] erklärt. Die erzeugte Binärdatei wird genau einen Stringblock und einen Typblock enthalten.

4.5.1 Gelöschte Instanzen

Wird eine Instanz vom Benutzer gelöscht, so wird ihr Speicher intern nicht direkt freigegeben, sondern sie wird als *gelöscht* markiert, indem ihre Id auf null gesetzt wird. Damit diese Instanzen nicht in Binärdateien geschrieben werden, müssen sie vor dem Schreiben aussortiert werden, und der von ihnen belegte Speicher muss freigegeben werden. Das Vorgehen ist in Abschnitt 4.7 beschrieben.

4.5.2 Instanzen umsortieren

Das Serialisierungsformat verlangt, dass Instanzen vom gleichen Untertyp innerhalb eines Blocks nacheinander stehen. Wir müssen also Instanzen eines Basistyps nach Untertyp sortieren. Untertypen können wiederum Untertypen besitzen, deswegen verwenden wir eine rekursive Funktion.

Listing 4.17: Umsortieren von Instanzen innerhalb von Pools

```
for ( storage_pool sub_pool in this.sub_pools ) {
  // Rekursiver Aufruf auf den Untertypen
    reorder_instances ( sub_pool );
}

// Untertypen sind jetzt bereits sortiert
GArray *new_array = g_array_new ( this.instances.len );

int64_t number_of_sub_instances = 0;
for ( storage_pool sub_pool in this.sub_pools ) {
    number_of_sub_instances += sub_pool.count;
}

// Einsortieren der Instanzen mit genau dem Typ des Pools
int64_t index = 0;
for ( skill_type current_instance in this.instances ) {
    // Einsortieren nur, falls kein Untertyp
    if ( current_instance.declaration == this.declaration ) {
        new_array[index] = current_instance;
        index++;
    }
}

// Untertypen dahinter einsortieren
storage_pool sub_pool;
for ( storage_pool sub_pool in this.sub_pools ) {
    for ( skill_type current_instance in sub_pool.instances ) {
        new_array[index] = current_instance;
        index++;
    }
}
[..] // Speicher der Instanzen freigeben
this.instances = new_array;
```

4.5.3 Local Base Pool Start Index

Der *local base pool start index*(*lbpsi*,[Fel13, vgl.Glossary,LBPSI]) ist der Index der ersten Instanz eines Untertyps im Array des Basispools. Wir schreiben Untertypen in der Reihenfolge, in der sie in der *sub_pools* Liste im Pool des Obertyps vorkommen. Der *lbpsi* für den ersten Untertyp ist die Anzahl der Instanzen des Obertyps minus die Anzahl aller Instanzen von Untertypen.

Listing 4.18: Bestimmen des Local Base Pool Start Index

```
// lbpsi als Zeiger übergeben, damit der Wert überschrieben werden kann
void calculate_lbpsi ( storage_pool pool, int64_t *lbpsi ) {
    pool.lbpsi = *lbpsi;
    int sub_instance_count = 0;
    for ( storage_pool sub_pool: pool.sub_pools ) {
        sub_instance_count += sub_pool.instances.len;
    }
    *lbpsi += pool.instances.len - sub_instance_count;
    for ( storage_pool sub_pool: pool.sub_pools ) {
        calculate_lbpsi ( sub_pool, lbpsi );
    }
}
```

Listing 4.18 veranschaulicht die Berechnung der *lbpsi*. Die Funktion wird für alle Pools von Basistypen aufgerufen.

4.5.4 Storage Pool Ids

Pools haben innerhalb einer Binärdatei eine Id, die über die Reihenfolge gegeben ist, in der sie in der Datei auftreten. Für die *write* Operation müssen wir diese Reihenfolge festlegen, wie in Listing 4.19 gezeigt.

Listing 4.19: Pool Ids zuweisen

```
int64_t pool_id = 0; // Pool-Ids fangen bei 0 an
for ( storage_pool pool in state.storage_pools ) {
    pool.id = pool_id;
    pool_id++;
}
```

4.5.5 Strings schreiben

Alle Stings, die in der Binärdatei vorkommen, stehen im Stringblock vor dem Typblock. Wir müssen also Strings aus Typinformationen und aus Felddaten sammeln, um den Stringblock zu schreiben Listing 4.20 zeigt das Vorgehen.

Listing 4.20: Sammeln von Strings zum Serialisieren

```
// Sammeln von Strings aus Typinformationen
for ( storage_pool pool in all_pools ) {
    string_access.add_string ( pool.declaration.name );
    // Enthält alle bekannten Felder
    for ( field_information field in pool.declaration.fields ) {
        string_access.add_string ( field.name );
    }
}

[..] // Sammeln von Strings aus Felddaten
     // Ggf. auch aus zusammengesetzten Typen
```

Die Strings in einen Stringblock zu schreiben funktioniert dann mit den Schritten:

- Schreibe die Anzahl der Strings
- Schreibe für jeden String das Offset (Offset des vorherigen Strings plus die Länge des Strings)
- Schreibe die Strings byteweise

4.5.6 Typinformation schreiben

Der Aufbau der Typinformation ist in [Fel13, Figure 2] dargestellt und in [Fel13, §6.4] im Detail ausgeführt. Die Implementierung schreibt für jeden Benutzertyp die Typinformation:

- Name des Typs
- Ggf. Name des Obertyps
- Ggf. *lbpsi*
- Anzahl der Instanzen
- *Restrictions* des Typs. Wird nicht unterstützt, deswegen immer 0.
- Anzahl der Felder
- Für jedes Feld
  - Restrictions. Wird nicht unterstützt, deswegen immer 0.
  - Typ
  - Namen
  - Offset der Felddaten

Falls der Benutzertyp ein Basistyp ist, schreiben wir für den Namen des Obertyps 0 und lassen den *lbpsi* weg. Für konstante Felder wird der Wert direkt nach dem Typindex eingefügt.

Listing 4.21: Typinformation für Konstanten schreiben

```
if ( type_info.type.is_constant () ) {
    if ( type_info.type == ConstantI8 ) {
        write_i8 ( type_info.constant_value );
    } else if ( type_info.type == ConstantI16 ) {
        [..] Andere Konstante Datentypen analog behandeln
    }
}
```

Für Felder, die Zeiger auf Benutzertypen enthalten, muss die Id des Pools, der die referenzierten Instanzen enthält, auf den Typindex für Zeiger addiert werden.

Listing 4.22: Referenzen auf Benutzertypen

```
int8_t type_index = type_info.type.get_index ();
if ( type_info.type == UserType ) {
  storage_pool target_pool = storage_pools[type_info.name];
  type_index += target_pool.pool_index;
}
write_i8 ( type_index );
```

Listing 4.22 zeigt, wie Referenzen auf Benutzertypen in der Typinformation dargestelt werden. die Funktion *type_info.type.get_index()* ist nach [Fel13, Appendix,E] implementiert. Für Zusammengesetzte Typen wird zusätzlich der Typ der Elemente und ggf. deren Anzahl angegeben. Für *Maps* die Anzahl der Basistypen, gefolgt von den Indizes für jeden Basistyp. Das wird in Listing **??** ausgeführt.

Listing 4.23: Typinformation für zusammengesetzte Typen

```
if ( type_info.type.is_container_type () ) ) {
    if ( type_info.type == MapType ) {
        write_v64 ( type_info.base_types.length );
        for ( type in type_info.base_types ) {
            // Schreibe IDs aller Basistypen
            write_i8 ( type.get_index () );
        }
    } else {
        if ( type_info.type == ConstantLengthArray ) {
            write_v64 ( type_info.array_length );
        }
        // Schreibe Id des Typs der Elemente
        write_i8 ( type_info.base_type.get_index () ) );
    }
}
```

4.5.7 Felddaten schreiben

Der Generator erzeugt beim Erstellen des Bindings für alle bekannten Felder Funktionen, um Daten aus diesem Feld zu schreiben. Die Funktion hat den *skill_state* als Parameter, um für Zeiger auf Benutzertypen die Id bestimmen zu können, den *string_access*, um für Strings die Id bestimmen zu können, außerdem die Instanz. Der Rückgabewert ist die Anzahl an Bytes, die geschrieben wurden.

Listing 4.24: Deklaration der write-Funktion

```
typedef int64_t write_function ( skill_state, string_access, skill_type );
```

Instanzen der Klasse *field_information* halten eine Referenz auf eine dieser Funktionen. Die *write*-Funktionen sind nach dem gleichen Muster implementiert wie die *read*-Funktionen in Abschnitt 4.4.4. Listing 4.25 zeigt den Aufruf der Funktionen.

Listing 4.25: Felddaten schreiben

```
for ( field_information current_field = pool.declaration.fields ) {
    for ( skill_type instance in pool.get_instances() ) {
        current_field->write ( state, strings, instance, out );
    }
}
```

4.5.8 Append

Voraussetzung für die Funktion *append* ist ein *skill_state*, der aus einer Binärdatei erzeugt wurde oder der bereits zu einem früheren Zeitpunkt in eine Binärdatei geschrieben wurde. Daten aus der Binärdatei dürfen im *skill_state* nicht verändert sein. Das bedeutet erstens, dass Instanzen aus der Binärdatei nicht gelöscht werden dürfen und zweitens, dass Felddaten aus der Binärdatei nicht verändert werden dürfen. Es ist möglich, dass die Binärdatei von einem Binding geschrieben wurde, welches aus einer anderen Spezifikation erzeugt wurde. In diesem Fall kann ein Benutzertyp in der Binärdatei andere Felder haben als im Binding [Fel13, vgl.§6.2.3]. *Append* ist nur möglich, falls die Binärdatei für jeden Benutzertypen eine Teilmenge der Felder oder genau die Felder definiert, die das Binding kennt.

Die Implementierung von *append* überschneidet sich stark mit *write* (Abschnitt 4.5), und verwendet größtenteils die gleichen Funktionen. Wir erklären in diesem Abschnitt nur die Unterschiede.

Das Binding initialisiert Pools mit der Id -1. Beim Lesen oder Schreiben einer Binärdatei bekommen Pools die Id gesetzt, die ihrer Reihenfolge in der Datei entspricht. Die Id eines Pools ist also gleich -1 gdw. der Pool noch nicht in der Binärdatei vorkommt. Beim Lesen oder Schreiben einer Binärdatei wird außerdem für jeden Pool die Liste von Feldern gesetzt $(storage\_pool.fields)$, die für diesen Typ in der Datei vorkommen. Bei *append* schreiben wir einen Pool in die Binärdatei, falls er

- Noch nicht in der Binärdatei vorkommt,

- Neue Instanzen enthält, oder
- Neue Felder enthält.

Das Umsortieren der Instanzen und Bestimmen der *lbpsi* funktioniert wie bei *write* (Abschnitte 4.5.2 und 4.5.3), für *append* werden aber nur neue Instanzen betrachtet. Pools, denen bereits eine Id ungleich -1 zugewiesen ist, behalten diese Id. Pools mit Id *-1* bekommen jetzt wie in Abschnitt 4.5.4 neue Ids zugewiesen, aber startend mit dem Maximum der bisher vergebenen Ids plus eins. Listing 4.26 zeigt das Vorgehen.

Listing 4.26: Pool Ids zuweisen bei Append

```
int64_t max_id = 0;
// Bisher größte Id finden
for ( storage_pool pool in all_pools ) {
    if ( pool.id > max_id ) {
        max_id = pool.id;
    }
}

// Nur Pools mit id '-1' sind noch nicht in der Binärdatei definiert
int64_t pool_id = max_id + 1;
for ( storage_pool pool in all_pools ) {
    if ( pool.id == -1 ) {
        pool.id = pool_id;
        pool_id++;
    }
}
```

Die *string_access* Instanz, die beim letzten *read* oder *write* erzeugt wurde, enthält alle Strings, die in der Binärdatei vorkommen. Wir sortieren wie in Abschnitt 4.5.5 Strings in den *string_access* ein, schreiben aber nur die Strings, deren Id größer ist als die größte Id in der Binärdatei. Strings, die bereits in der Binärdatei vorkommen, müssen nicht erneut geschrieben werden.

4.6 Fehlerbehandlung

Wenn fehlerhafte Binärdateien gelesen werden [Fel13, Appendix,B], bricht das Binding die Programmausführung ab. Der Fehler wird auf der Konsole ausgegeben. Die Programmiersprache C unterstützt keine intelligentere Behandlung von Fehlern, aber es ist eine Implementierung denkbar, die Ausnahmen anhand von Rückgabewerten darstellt, und Fehlermeldungen in eine globale Variable schreibt. Aus Zeitgründen haben wir eine Lösung dieser Art nicht implementieren können.

4.7 Speicherfreigabe

Um den Speicher freizugeben, der für eine Instanz auf dem Heap allokiert ist, reicht ein Aufruf von *free*. Auf diese Weise kann aber undefiniertes Verhalten des Programmes entstehen, wenn Zeiger auf diese Instanz existieren. Deswegen verwenden wir Funktionen, um solche Zeiger auf *null* zu setzen:

Listing 4.27: Definition der Cleanup-Funktion

```
typedef void cleanup_function ( skill_type instance );
```

Wir referenzieren eine solche Funktion in der Klasse *type_declaration*. Vor dem Schreiben in eine Binärdatei setzen wir mit Hilfe dieser Funktionen Referenzen auf gelöschte Instanzen auf *null*, und geben dann den Speicher für gelöschte Instanzen frei.

4.8 Tests

Mithilfe von Tests soll sichergestellt werden, dass die Implementierung korrekt ist. Die Tests orientieren sich an den Beispielen in [Fel13] und an den Tests aus der Anbindung für Scala [ski] und verwenden die dort bereitgestellten Binärdateien. Die Tests lesen die folgenden Binärdateien ein und gleichen die erzeugten Instanzen mit den erwarteten Werten ab:

- Die Binärdatei mit zwei *date*-Instanzen [Fel13, §6.6]. Damit wird grundlegend das Parsen von Binärdateien getestet.
- Eine Binärdatei mit einem Benutzertyp, der Felder aller Zahlentypen, Strings, und Boolean enthält.
- Eine Binärdatei mit einem Benutzertyp, in dessen Felder alle zusammengesetzten Typen vorkommen.
- Binärdateien zu den Beispielen aus [Fel13, §6.2.3]. Wir testen damit das Lesen von Instanzen, die über mehrere Blöcke verteilt sind, und das Lesen von Typblöcken, die neue Felder hinzufügen. Wir testen auch die Kombination aus beidem, neue Instanzen und neue Felder in einem Block.
- Eine Binärdatei mit leeren Blöcken, und eine mit Instanzen ohne Felder. Damit testen wir, dass diese Fälle nicht zu Fehlern führen.
- Eine Binärdatei zu [Fel13, §6.3.2], die Benutzertypen mit Vererbungshierarchie enthält, deren Instanzen über mehrere Blöcke verteilt sind. Wir testen damit auch die *instanceof*-Funktionen und gleichen die Typen der Instanzen mit den erwarteten Werten ab.
- Eine Binärdatei, die *Annotations* enthält, um zu testen, dass der Zeiger richtig ausgewertet wird.
- Eine Binärdatei, die konstante Felder definiert.

Alle Tests schreiben zusätzlich selbst Binärdateien, lesen sie wieder ein, und gleichen die Instanzen ab, um das Schreiben zu testen.

Außerdem werden Binärdateien mit unbekannten Typen gelesen, um zu testen, dass sie nicht zum Fehler führen.

5 Zusammenfassung und Ausblick

In der Diplomarbeit ist die Entwicklung einer SKilL-Sprachanbindung an die Programmiersprache C beschrieben. Damit ist nachgewiesen, dass die Anbindung an eine Sprache ohne Objektorientierung möglich ist. Wir haben beschrieben, wie Benutzertypen, die in einer Vererbungshierarchie organisiert sind, in C abgebildet werden können. Die Implementierung erlaubt Typkonvertierungen zwischen Benutzertypen, und bietet Typsicherheit.

Die Anbindung könnte hinsichtlich Performance weiter verbessert werden, indem *lazy loading* unterstützt wird. Es könnte auch eine Fehlerbehandlung durch Ausnahmen wie aus höheren Programmiersprachen nachgebildet werden. Außerdem könnte die Anbindung erweitert werden, sodass sie die volle SKilL Unterstützung bietet.

Naturgemäß ist die Verwendung der generierten Bindings syntaktisch umständlicher als bei einem Binding für eine höhere Programmiersprache. Der Benutzer muss zusätzliche Typkonvertierungen einfügen, weil es nicht möglich ist, Methoden zu überladen. Bei zusammengesetzten Typen wird der Datentyp ihrer Elemente nicht überprüft. Die Anbindung bietet trotzdem den gleichen Funktionsumfang wie Anbindungen an höhere Programmiersprachen und ist dank intelligenter Gestaltung der Benutzungsschnittstelle sinnvoll einsetzbar.

Literaturverzeichnis

[c9903] *The C standard: incorporating technical corrigendum 1; BS ISO/IEC 9899:1999; [includes the C rationale].* Wiley, Chichester [u.a.], 2003. URL http://swbplus.bsz-bw.de/bsz107279258cov.htm. (Zitiert auf den Seiten 17, 22 und 32)

[Fel13] T. Felden. The SKilL Language. Technischer Bericht Informatik 2013/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2013. (Zitiert auf den Seiten 9, 10, 11, 13, 14, 22, 28, 33, 36, 37, 42, 43, 44, 46, 47, 48, 49 und 50)

[fre] FreeMarker Java Template Engine. http://freemarker.org/. (Zitiert auf Seite 15)

[gli14] GLib Reference Manual. https://developer.gnome.org/glib/2.42/, 2014. (Zitiert auf den Seiten 17, 23, 24 und 35)

[iee08] *IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society.* 3 Park Avenue, New York, NY 10016-5997, USA, 2008. (Zitiert auf Seite 17)

[Prz14] D. Przytarski. Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau, 2014. (Zitiert auf Seite 9)

[ski] SKilL auf Github. https://github.com/skill-lang/skill. (Zitiert auf den Seiten 9 und 50)

[Ung14] W. Ungur. *Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2014. (Zitiert auf den Seiten 9, 14 und 16)

Alle URLs wurden zuletzt am 03. 11. 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