Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Plattformunabhängiges Abhängigkeitsmanagement am Beispiel von SKilL/C#

Simon Glaub

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dr. rer. nat. Timm Felden
Beginn am:	5. Juni 2018
Beendet am:	5. Dezember 2018

Kurzfassung

Die Serialization Killer Language (SKilL) ist eine Plattform, die entwickelt wurde um die Serialisierung von großen Datenmengen innerhalb von Werkzeugketten zu ermöglichen. Dabei ist sie vor allem auf Werkzeugketten ausgelegt, in denen unterschiedliche Programmiersprachen zum Einsatz kommen. In dieser Arbeit wird mithilfe einer Implementierung von SKilL für die Programmiersprache C# untersucht, wie ein plattformunabhängiges Abhängigkeitsmanagement für gemeinsam genutzten Code einzelner SKilL-Implementierungen effektiv realisiert werden kann. Dazu wird zum einen eine SKilL-Anbindung für C# erstellt, die an die bestehende SKilL/Java-Implementierung angelehnt ist. Zum anderen wird eine Strategie entwickelt und implementiert, um die gemeinsam genutzten Implementierungsbestandteile dem Benutzer zu Verfügung zu stellen. Diese neu geschaffene Anbindung wird abschließend ausführlich getestet.

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel wird zunächst die Motivation der Arbeit erläutert. Danach werden die Ziele dieser Arbeit und die dafür zu erledigenden Aufgaben beschrieben. Den Abschluss bildet die Gliederung, welche einen Überblick über den gesamten Inhalt dieser Arbeit liefert.

1.1. Motivation

Im Rahmen von industriellen oder wissenschaftlichen Softwareprojekten fallen oft große Datenmengen an, die zwischen der Weiterverarbeitung durch verschiedenen Werkzeugen serialisiert werden müssen. Dabei kann es häufig vorkommen, dass einzelne Werkzeuge in unterschiedlichen Programmiersprachen programmiert sind. Durch die Verwendung unterschiedlicher Programmiersprachen sind diese Werkzeuge aber, ohne eine sprachunabhängige Struktur der Kerndaten, nicht ohne weiteres miteinander kompatibel. Um dieser Problematik zu begegnen, wurde die Serialization Killer Language (SKilL) entworfen, die es erlaubt Datenformate zu beschreiben. Diese eignen sich zur plattform- und sprachunabhängigen Serialisierung und sind, im Gegensatz zu existierenden Lösungen, auf sehr große Datenmengen ausgelegt [Fel17].

C# ist eine objektorientierte Allzweck-Programmiersprache, die von Microsoft im Rahmen ihrer .NET Framework Initiative entwickelt wird [Ecm17]. Sie ist stark verbreitet und wird aktiv von Microsoft weiterentwickelt. Es wäre deshalb sinnvoll eine SKilL-Anbindung für diese Sprache zu entwickeln. So wäre es möglich auch Werkzeuge, die mithilfe von C# programmiert wurden, in eine SKilL-Werkzeugkette einzubinden.

1.2. Ziele

In dieser Arbeit soll mithilfe einer Implementierung von SKilL für C# untersucht werden, wie ein plattformunabhängiges Abhängigkeitsmanagement für gemeinsam genutzten Code einzelner SKilL-Implementierungen effektiv realisiert werden kann.

Dazu muss zum einen ein Codegenerator für die Programmiersprache C# entwickelt werden, mit dessen Hilfe aus einer mit SKilL geschriebenen Spezifikation eine entsprechende Implementierung in C# generiert werden kann. Der SKilL/C#-Codegenerator muss dabei mindestens die SKilL-Kernsprache sowie die Features *auto*, *custom*, *documented* und *escaped* unterstützen. Welche Anforderungen genau zum Erfüllen dieser Features erfüllt sein müssen, wird in Kapitel 2.1.1 genauer erläutert. Da sich die Programmiersprachen Java und C# syntaktisch und semantisch stark ähneln, kann als Grundlage des Generators die bestehende SKilL/Java-Implementierung verwendet werden. Es muss außerdem anhand geeigneter Tests gezeigt werden, dass die neue Implementierung plattformunabhängig und kompatibel zu bereits existierenden SKilL-Anbindungen ist.

Weiterhin muss eine Strategie entwickelt und implementiert werden, die es dem SKilL/C#--Codegenerator ermöglicht gemeinsam genutzte Implementierungsbestandteile bereitzustellen, sowie es bereits für SKilL/Java der Fall ist. Bei den gemeinsam genutzten Implementierungsbestandteilen handelt es sich um Teile des Programmcodes, die unabhängig von einer SKilL-Spezifikation sind. Diese müssen deshalb nicht jedes mal neu generiert werden, sondern sind als Bibliothek abgespeichert. Im Rahmen dieser Arbeit soll diese Bibliothek für die Programmiersprache C# implementiert und bei der Generierung des Codes zu einer SKilL-Spezifikation so ausgegeben werden, dass sie direkt eingebunden werden kann. Auch hiervon muss die plattformübergreifende Funktionsfähigkeit demonstriert werden.

1.3. Gliederung

Diese Bachelorarbeit gliedert sich in verschiedene Teile. Zunächst werden die Grundlagen erläutert. Dazu wird darauf eingegangen, was SKilL ist und welche Features es beinhaltet. Dabei werden insbesondere die Teile von SKilL näher erklärt, die im Rahmen dieser Bachelorarbeit besonders wichtig sind. Außerdem werden die Grundlagen der Programmiersprache C# beschrieben, wobei insbesondere die Unterschiede zu Java näher erläutert werden.

Der zweite Teil der Arbeit beschäftigt sich mit der praktischen Implementierung des Codegenerators und den gemeinsam genutzten Implementierungsbestandteilen für die Programmiersprache C#. Dazu wird für jede dieser Aufgaben zuerst die SKilL/Java-Implementierung beschrieben, die als Grundlage verwendet wird. Danach werden die nötigen Anpassungen des Codes an C# erläutert. Zuletzt werden bei der Implementierung aufgetretene Probleme und die entsprechenden Lösungen dafür beschrieben.

Der dritte Teil der Arbeit beschäftigt sich damit eine Strategie zu finden und zu implementieren, um die gemeinsam genutzten Implementierungsbestandteile durch den SKilL/C#-Codegenerator bereitzustellen. Im vierten Teil der Arbeit wird näher erläutert welche Tests vorgenommen und wie sie erstellt werden. Dazu zählen sowohl Tests zum Implementierungs-Teil, zur Bereitstellung der gemeinsamen Implementierungsbestandteile, als auch zur Interoperabilität mit anderen SKilL-Anbindungen. Außerdem werden Tests zur Verwendung von Dokumentationskommentaren und zur Leistung der SKilL/C#-Implementierung durchgeführt. Im letzten Teil der Bachelorarbeit findet sich eine Zusammenfassung dieser Arbeit, sowie ein Ausblick, wie die in dieser Arbeit behandelten Themen weiter verfolgt werden können.

2. Grundlagen

In diesem Kapitel werden die Grundlagen näher erläutert, die zum besseren Verständnis dieser Arbeit nötig sind. Zunächst wird dazu das Projekt SKilL und die enthaltenen Features vorgestellt. Danach folgt ein Teil der sich näher mit der Programmiersprache C# beschäftigt. Sowohl bei SKilL, als auch bei C# wird dabei besonders auf die Bestandteile eingegangen, die für diese Bachelorarbeit wichtig sind.

2.1. Serialization Killer Language (SKilL)

Die Serialization Killer Language (SKilL) ist eine Softwareplattform für Werkzeugketten, die eine möglichst wartungsfreie und effiziente Kommunikation zwischen den einzelnen Werkzeugen ermöglichen soll. Insbesondere die Kommunikation zwischen Werkzeugen, die in unterschiedlichen Programmiersprachen implementiert wurden, soll damit möglich gemacht werden.

Dazu definiert SKilL zum einen eine leicht benutzbare Spezifikationssprache, mit deren Hilfe Datenstrukturen dargestellt werden können. Diese Sprache unterstützt dabei einfache Datentypen (zum Beispiel Integers und Strings), Container-Typen (zum Beispiel Sets und Maps), typsichere Zeiger und Einfachvererbung. Außerdem ist sie modular aufgebaut, um so größere Spezifikationen besser lesbar zu machen [Fel17].

Zum anderen stellt SKilL ein formal spezifiziertes, auf Änderungstoleranz und Ausleseeffizienz optimiertes Binärformat bereit, das es ermöglicht die spezifizierten Typen auf eine bitweise Repräsentation von Objekten abzubilden. Diese Abbildung ist sehr kompakt gehalten und deshalb leicht skalierbar. Außerdem ist ihre Struktur möglichst simpel gestaltet, weshalb sie leicht für eine neue Programmiersprache umgesetzt werden kann [Fel17].

SKilL stellt außerdem eine Schnittstelle zur Erstellung und Änderung der serialisierten Dateien zur Verfügung. Diese kann mithilfe des Codegenerators von SKilL aus einer, mit der Spezifikationssprache erstellten Spezifikation, spezifisch für eine von SKilL hierfür unterstützte Programmiersprache erzeugt werden. Diese Schnittstelle gewährleistet Typsicherheit und stellt eine komfortable Einbindung in die Zielsprache bereit [Fel17].

2.1.1. Level der Sprachunterstützung

Da die Implementierung der kompletten SKilL-Plattform sehr zeitaufwändig sein kann, wurde die Sprache in zwei Teile aufgeteilt, die Kernsprache und zusätzliche Features. Die Kernsprache unterstützt die wichtigsten Funktionen von SKilL und kann relativ schnell implementiert werden. Die Features enthalten dagegen zusätzliche Funktionen, die in den vielen Szenarien nicht benötigt

werden. Im folgenden werden die Kernsprache und die Features anhand der Spezifikation von SKilL [Fel17, S.59+60] näher beschrieben. Dabei werden nur Features vorgestellt, die auch implementiert werden.

Kernsprache: Damit die Kernsprache als umgesetzt gilt müssen mehrere Funktionen implementiert sein. Als erstes müssen alle SKilL-Typen den entsprechenden Typen der Zielsprache zugeordnet sein und korrekt verarbeitet werden können. Weiterhin muss der aktuelle SKilL-Zustand korrekt dargestellt und verwaltet werden können. Außerdem muss das Schreiben eines SKilL-Zustands in eine binäre SKilL-Datei und das Einlesen einer solchen Datei möglich sein. Auch müssen die Singleton-, Default- und NonNull-Einschränkungen unterstützt werden. Bei der Singleton-Einschränkung darf es für den entsprechenden Typ nur eine Instanz geben. Bei der NonNull-Einschränkung darf das entsprechende Feld nicht den Wert `null` enthalten. Bei der Default-Einschränkung kann der Standardwert für den entsprechenden Typ beziehungsweise das entsprechende Feld festgelegt werden. Zuletzt muss ein öffentliches Interface für die entsprechende Zielsprache erzeugt werden können. Über dieses muss man Zugriff auf SKilL-Binärdateien erhalten und diese darüber verändern können.

Auto: Damit das Feature *auto* als umgesetzt gilt, muss die Verwendung von sogenannten *Auto-Feldern* unterstützt werden. Diese werden zwar vom Codegenerator erzeugt, werden allerdings bei der Serialisierung nicht berücksichtigt. Sie können dafür verwendet werden um interne Berechnungen leichter durchzuführen.

Append: Damit das Feature *append* als umgesetzt gilt, muss es möglich sein neue Daten einer bereits bestehenden Datei hinzuzufügen, ohne diese komplett neu schreiben zu müssen.

Customs: Damit das Feature *customs* als umgesetzt gilt, muss die Verwendung von sogenannten *Custom-Feldern* unterstützt werden. Mithilfe dieser ist es möglich den Codegenerator einer bestimmten Programmiersprache für dieses Feld zu verwenden und dabei spezielle Operationen dafür anzugeben (zum Beispiel Importe). Dadurch kann die Integration einer generierten Schnittstelle in die Werkzeugumgebung verbessert werden.

Documented: Damit das Feature *documented* als umgesetzt gilt, muss die sprachabhängige Behandlung von Kommentaren unterstützt werden. Wenn also Kommentare in der Spezifikations-Datei angegeben werden, müssen diese bei der Generierung so umgewandelt werden, dass sie der Syntax der Zielsprache entsprechen.

Escaped: Damit das Feature *escaped* als umgesetzt gilt, muss es möglich sein Schlüsselwörter der Zielsprache und illegale Zeichen in den Spezifikationsdateien zu verwenden. Dazu müssen in der Spezifikation angegebene Bezeichner bei der Codegenerierung verändert werden.

Interfaces: Damit das Feature *interfaces* als umgesetzt gilt, muss die Verwendung von Interfaces unterstützt werden, falls diese von der Zielsprache unterstützt werden.

Lazy: Damit das Feature *lazy* als umgesetzt gilt, muss die Verwendung der sogenannten *On-Demand Deserialization* unterstützt werden. Damit ist möglich bei der Deserialisierung manche Daten zu überspringen.

View: Damit das Feature *view* als umgesetzt gilt, muss die Verwendung von sogenannten *Views* unterstützt werden. Diese ermöglichen es Felder nachträglich umzubenennen oder ihren Typ zu verändern.

2.2. Programmiersprache C#

C# ist eine objektorientierte Allzweck-Programmiersprache, die von Anders Hejlsberg, Scott Wiltamuth und Peter Golde im Auftrag von Microsoft entwickelt wurde. Die erste Implementation von C# wurde im Juli 2000 von Microsoft im Rahmen ihrer .NET Framework Initiative veröffentlicht [Ecm17]. C# ist zwar plattformunabhängig, wird aber meistens im Rahmen des .NET Frameworks verwendet, da die Sprache darauf besonders optimiert ist. Die Verwendung des .NET Frameworks ist dabei auf Windows beschränkt. Inzwischen wird C# aber auch auf Linux und MacOS mithilfe von Mono oder .NET Core unterstützt [AA17].

Ein Hauptziel beim Design von C# war es Prinzipien der Softwaretechnik zu unterstützen. Dazu gehören beispielsweise eine starke Typsicherheit, Überprüfung der Grenzen von Arrays, Erkennung des Versuchs uninitialisierte Variablen zu verwenden und eine automatische *Garbage Collection*. Außerdem sollte die Sprache dafür geeignet sein Programme sowohl für gehostete als auch für eingebettete Systeme zu entwickeln [Ecm17].

Code, der in C# geschrieben wird, kann sowohl direkt in Maschinencode (zum Beispiel mit .NET Native) [Micb] oder alternativ, ähnlich wie in Java, in eine Zwischensprache, die Common Intermediate Language (CIL), kompiliert werden [NMRW17]. Der CIL-Code kann dann in einer entsprechenden Laufzeitumgebung, in diesem Fall zum Beispiel die sogenannte Common Language Runtime (CLR), ausgeführt werden. Dadurch kann C# auf jeder Plattform verwendet werden, die eine solche Laufzeitumgebung unterstützt.

C#-Code kann in Form von Dynamic Link Library (DLL)-Dateien gespeichert werden. Diese Dateien können als Programmbibliotheken in andere C#-Programme eingebunden werden. So kann derselbe Code von mehreren Programmen verwendet werden. Für C# werden dabei aber spezielle .NET-DLLs verwendet. Während normale DLL-Dateien direkt ausführbaren Maschinencode enthalten, enthält eine .NET-DLL-Datei eine sogenannte *Assembly* [Mica]. Eine Assembly besteht aus Code im CIL-Format und aus einem sogenannten *Manifest*. Ein Manifest enthält unter anderem Informationen zu den enthaltenen Klassen, zur Anwendung selbst und zu Abhängigkeiten von anderen Assemblies.

2.2.1. Unterschiede zu Java

Die Programmiersprache C# weist eine große Ähnlichkeit zu Java auf und ist in vielen Punkten nahezu identisch. Im Rahmen dieser Bachelorarbeit wird deshalb für die Implementierung von C#-Code die bestehende SKilL/Java-Implementierung als Vorlage verwendet und entsprechend

angepasst. Um ein besseres Verständnis dafür zu bieten, welche Anpassungen dabei vorgenommen werden müssen, werden im folgenden die wichtigsten Unterschiede zwischen C# und Java näher erläutert.

Namespace

In Java werden, um Klassen und Interfaces besser organisieren zu können, sogenannte *Pakete* verwendet [GJS+18, S.177-186]. Diese bestimmen unter anderem die Zugriffsmöglichkeiten auf die darin enthaltenen Klassen und können in anderen Klassen importiert werden. In C# existiert ein ähnliches Konzept, die sogenannten *Namespaces* [Ecm17, S.247+248]. Das Konzept der Namespaces wird auch in der Programmiersprache C++ verwendet [Sta13, S.157-161]. Während allerdings in Java die Paket-Struktur mit der Ordner-Struktur im Dateisystem übereinstimmen muss, ist dies in C# nicht nötig. Hier ist die Struktur der Namespaces vollkommen unabhängig von der Ordner-Struktur, wobei natürlich auch hier ein logischer Zusammenhang der Strukturen für eine bessere Übersichtlichkeit sorgen kann.

Using-Anweisung

In Java werden, um Zugriff auf andere Klassen und Interfaces zu erhalten, sogenannte Import-Anweisungen verwendet. Damit können entweder einzelne Klassen und Interfaces oder ganze Pakete zur Verwendung in der aktuellen Klasse verfügbar gemacht werden [GJS+18, S.186-192]. Das entsprechende Äquivalent in C# dazu sind die sogenannten Using-Anweisungen [Ecm17, S.249-255]. Das Konzept der Using-Anweisungen wird auch in der Programmiersprache C++ verwendet [Sta13, S.161-169]. Mit einer normalen Using-Anweisung können dabei, im Gegensatz zu Java, nur ganze Namespaces und keine einzelnen Klassen oder Interfaces importiert werden. Um einzelne Klassen zu importieren muss eine sogenannte *Using-Aliasdirektive* verwendet werden. Hierbei wird der Klasse, die importiert wird, ein neuer Name für die Verwendung in der importierenden Klasse zugewiesen. Dieser Alias darf sich vom definierten Namen der Klassen unterscheiden. Using-Aliasdirektiven können auch verwendet werden um für ganze Namespaces ein Alias festzulegen. Diese Funktion, ein Alias bei einem Import zu vergeben, existiert in Java nicht.

Vererbung

Um in C# die Methode einer Superklasse in einer entsprechenden Subklasse überschreiben zu können, reicht es nicht aus, wie in Java, einfach die entsprechende Methode in der Subklasse neu zu implementieren [GJS+18, S.257-267]. In C# muss zum einen in der Superklasse die zu überschreibende Methode mit dem Schlüsselwort *virtual* gekennzeichnet werden, zum anderen muss die überschriebene Methode in der Subklasse mit dem Schlüsselwort *override* gekennzeichnet werden [Ecm17, S.297-301]. Das Schlüsselwort *override* muss dabei auch beim Überschreiben von abstrakten Methoden verwendet werden.

Zusätzlich gibt es in C# die Möglichkeit, Methoden und Attribute der Superklasse auszublenden und neue Versionen davon in der Subklasse zu erstellen. Dazu muss die entsprechende Methode beziehungsweise das entsprechende Attribut mit dem Schlüsselwort *new* gekennzeichnet werden [Ecm17, S.274+275]. Diese Möglichkeit existiert in Java nicht.

Generische Typen

Wie in Java ist es auch in C# möglich generische Klassen und Interfaces zu erstellen und zu verwenden. Dabei stehen in C# allerdings einige der Funktionen nicht zur Verfügung, die Java in diesem Zusammenhang bietet. In Java gibt es die Möglichkeit bei der Deklaration eines generischen Typs als Typargument, statt eines expliziten Typs, eine sogenannte *Wildcard*, im Programmcode dargestellt als Fragezeichen, mitzugeben [GJS+18, S.59-64].

Der Typ, den die Wildcard annehmen kann, kann dabei nach oben oder nach unten beschränkt werden. Um den Typ einer Wildcard nach oben hin zu beschränken, wird die Notation `<? extends Supertyp>` verwendet. Dadurch muss der Typ der Wildcard ein Subtyp des angegebenen Supertyps sein. Um den Typ einer Wildcard nach unten hin zu beschränken, wird die Notation `<? super Subtyp>` verwendet. Dadurch muss der Typ der Wildcard ein Supertyp des angegebenen Subtyps sein. Diese Funktionen stehen in C# nicht zur Verfügung und können nur über Umwege realisiert werden. Eine solche Möglichkeit diese Funktionen zu realisieren, wird in Kapitel 3.2.2 beschrieben.

Property

Eine häufig verwendete Vorgehensweise beim Erstellen von Klassen ist es, die dafür definierten Attribute so zu implementieren, dass man von anderen Klassen nicht direkt darauf zugreifen kann. Stattdessen werden Attributzugriffe in Java über sogenannte Getter- und Setter-Methoden vorgenommen. In C# gibt es, als Alternative dazu, sogenannte *Properties*. Ein Property ist eine Komponente einer Klasse, die Zugriff auf ein Attribut dieser Klasse ermöglicht [Ecm17, S.308-315]. Der Aufbau ähnelt dabei der einer Methode, mit dem Unterschied, dass kein Parameter übergeben wird, wogegen die Syntax zum Verwenden, der von Feldern entspricht. Ein Property kann dabei eine Zugriffsfunktion zum Abrufen und eine zum Festlegen des entsprechenden Attributs enthalten. Die Zugriffsfunktion zum Abrufen wird dabei über das Schlüsselwort *get* und die zum Festlegen über *set* gekennzeichnet. Für beide Zugriffsfunktionen kann ein beliebiger auszuführender Code definiert werden. Dabei muss allerdings im Teil zum Abrufen ein Wert zurückgegeben werden. Im Teil zum Festlegen kann, mithilfe des Parameters *value*, auf den entsprechenden übergebenen Wert zugegriffen werden. Ein Property kann dabei auch als abstrakt gekennzeichnet oder für ein Interface festgelegt werden. In diesem Fall wird nur der Kopf des Properties und der Zugriffsfunktionen, die verwendet werden sollen, und nicht die eigentliche Implementierung angegeben.

Methoden-Parameter

Im Gegensatz zu Java gibt es in C# die Möglichkeit sogenannte Referenz- und Ausgabeparameter in Methoden zu verwenden [Ecm17, S.74+75]. Beide Parametertypen teilen sich dabei die Eigenschaft, dass sie im Gegensatz zu normalen Parametern keinen neuen Speicherort erstellen. Stattdessen enthalten sie eine Referenz zu dem Speicherort der übergebenen Variable. Ein Referenzparameter wird mit dem Schlüsselwort *ref*, sowohl in der Methodendeklaration als auch beim Methodenaufruf, gekennzeichnet. Innerhalb einer Methode wird der Parameter als initialisiert betrachtet, weshalb eine Variable vor der Übergabe als Referenzparameter auch zwingend initialisiert worden sein muss. Ein Ausgabeparameter wird mit dem Schlüsselwort *out*, sowohl in der Methodendeklaration als auch beim Methodenaufruf, gekennzeichnet. Innerhalb einer Methode wird der Parameter als

uninitialisiert betrachtet, weshalb eine Variable vor der Übergabe als Ausgabeparameter auch nicht zwingend initialisiert worden sein muss. Stattdessen muss der Ausgabeparameter zwingend initialisiert worden sein, bevor die Methode beendet wird.

Collections

Unter *Collections* versteht man, sowohl in Java als auch in C#, Datentypen mit Hilfe derer man mehrere Werte abspeichern und organisieren kann. Dazu zählen unter anderem Listen, Maps und Sets. Dabei unterscheidet sich in Java und C# zum einen die konkrete Bezeichnung der entsprechenden Klassen und zum anderen die Methoden, die zum Zugriff darauf verwendet werden. Besonders zu beachten ist dabei, dass in C# auf Elemente einer Collection über einen sogenannten *Indexer* zugegriffen werden kann. Die Syntax dieses Zugriffs ist dabei identisch zu der des Zugriffs auf Elemente eines Arrays [Ecm17, S.322-325]. Als Beispiel: Will man das Element an der Position zwei einer Liste namens `list` abrufen, so kann das in C# mit der Codezeile `list[2]` erreicht werden. Zusätzlich ist es aber auch in C# möglich Elemente einer Collection mithilfe einer Methode abzurufen.

Vor allem bei Dictionaries, dem Äquivalent zu Maps aus Java, kann das sinnvoll sein, da hier bei Verwendung des Indexers eine Exception geworfen wird, falls das gesuchte Element nicht vorhanden ist [Micc]. Wird dagegen die Methode `TryGetValue` dafür verwendet erhält man als Rückgabewert `null`. Auch ein Unterschied ist, dass Dictionaries keinen `null`-Schlüssel enthalten können. Für die Maps aus Java ist das dagegen möglich.

Es besteht außerdem ein Unterschied zwischen der `ArrayList`-Klasse von Java und C#. Während in Java die `ArrayList`-Klasse generisch ist, ist sie in C# nicht-generisch. Ein weiterer Unterschied ist, dass auch bei der Initialisierung einer Collection zwingend die entsprechenden Typargumente explizit angegeben werden müssen. In Java dagegen reicht es aus diese bei der Deklarierung anzugeben.

Kommentare

In C# gibt es, wie auch in Java, drei unterschiedliche Arten von Kommentaren: einzeilige Kommentare, mehrzeilige Kommentare und Dokumentationskommentare. Dokumentationskommentare werden dabei direkt vor einem Element des Codes angegeben, das sie näher beschreiben sollen. Während einzeilige und mehrzeilige Kommentare in Java und C# identisch dargestellt werden, also mit den Zeichen `//` beziehungsweise mit `/*` und `*/` [GJS+18, S.21+22] [Ecm17, S.16-18], unterscheidet sich die Darstellung der Dokumentationskommentare.

In Java werden Dokumentationskommentare mit den Zeichen `/**` begonnen, mit `*` fortgesetzt und mit `*/` beendet [Orab]. Mithilfe des Zeichens `@` können verschiedene Tags angegeben werden. Mit dem Tag `@author` zum Beispiel kann der Name des Autors des entsprechenden Code-Elements angegeben werden. Aus diesen angegebenen Informationen kann mithilfe des Werkzeugs *javadoc* eine HTML-Dokumentation erzeugt werden.

In C# dagegen werden stattdessen sogenannte XML-Dokumentationskommentare verwendet, in denen jede Zeile mit den Zeichen `///` beginnt [Ecm17, S.475-484]. Hier können XML-Elemente eingefügt werden, die zusätzliche Informationen zum entsprechenden Code-Element enthalten.

Für allgemeine Informationen zum Code-Element wird das XML-Element `<sumary> </summary>` verwendet. Aus den XML-Dokumentationskommentare kann dann beim Kompilieren des Codes mithilfe der Option */doc* eine XML-Dokumentationsdatei erstellt werden.

Bezeichner

In Java bestehen Bezeichner aus einer beliebig langen Sequenz aus Buchstaben, Zahlen, dem Sonderzeichen `$` und Unterstrichen [GJS+18, S.22-24]. Ein Bezeichner kann dabei nicht aus einem einzelnem Unterstrich bestehen, da das als Schlüsselwort gilt. Außerdem darf das erste Zeichen keine Zahl sein. In C# besteht ein Bezeichner aus den selben Zeichen, die auch in Java zugelassen sind. Allerdings ist das Sonderzeichen `$` nicht erlaubt. Ein weiterer Unterschied ist, dass das Zeichen `@` am Anfang eines Bezeichners verwendet werden kann [Ecm17, S.19-21]. Üblicherweise wird es verwendet um C#-Schlüsselwörter als Bezeichner verwenden zu können.

In C# ist es außerdem nicht möglich für verschiedene Elemente innerhalb eines Typs denselben Bezeichner oder den Bezeichner dieses Typs zu verwenden. In einer Klasse dürfen also mehrere Attribute, Methoden und Properties weder denselben Bezeichner, noch den der Klasse verwenden. Wenn eine Klasse also zum Beispiel ein Attribut mit dem Bezeichner `age` enthält, darf weder eine Methode noch ein Property in dieser Klasse den Bezeichner `age` ebenfalls verwenden. In Java ist diese Einschränkung nur teilweise vorhanden. Hier dürfen Attribute und Methoden nur jeweils untereinander nicht denselben Bezeichner verwenden. Wenn eine Klasse also zum Beispiel ein Attribut mit dem Bezeichner `age` enthält, darf eine Methode in dieser Klasse den Bezeichner `age` ebenfalls verwenden. Alle anderen für C# beschriebenen Einschränkungen gelten für Java nicht.

3. Implementierung

In diesem Kapitel wird die Implementierung des Codegenerators und der gemeinsam genutzten Implementierungsbestandteile für die Programmiersprache C# erläutert. Dazu wird jeweils zuerst die bestehende SKilL/Java-Implementierung beschrieben und dann die nötigen Anpassungen an C#. Dabei wird vor allem auf Probleme und Besonderheiten bei der Implementierung eingegangen.

3.1. Codegenerator

Das Hauptziel dieser Bachelorarbeit ist einen Codegenerator für die Programmiersprache C# zu erstellen, der an die bereits vorhandene SKilL/Java-Implementierung angelehnt ist. Dazu wird die Funktionsweise der SKilL/Java-Implementierung genau analysiert und anschließend bezüglich der Unterschiede zu C# (siehe Kapitel 2.2.1) angepasst. Der bestehende SKilL/Java-Codegenerator ist mithilfe der Programmiersprachen Scala und Java implementiert worden. Da der SKilL/Java-Codegenerator als Grundlage für den SKilL/C#-Codegenerator verwendet wird, werden hierfür die selben Programmiersprachen verwendet. Der SKilL/Java-Codegenerator besteht insgesamt aus 1860 Codezeilen.

3.1.1. Erforderlichen Anpassungen für die Übertragung auf C#

Als Grundlage der Implementierung des SKilL/C#-Codegenerators wird der bestehende SKilL/Java-Codegenerator übernommen. Dieser generiert dabei jeweils eine Datei für jeden spezifizierten Typ und die Dateien `internal`, `SkillFile` und falls gewünscht `Visitor`. Der SKilL/C#-Codegenerator besteht insgesamt aus 1852 Codezeilen. Durch ihn werden ein Teil der Kernsprache und die verpflichtenden Features *auto*, *custom*, *documented* und *escaped* implementiert. Außerdem werden hier zusätzlich die Features *interfaces* und *view* implementiert. Der Rest der Kernsprache und weitere zusätzliche Features werden durch die gemeinsam genutzten Implementierungsbestandteile implementiert (siehe Kapitel 3.2).

Allgemeine Anpassungen

Als erstes wird der interne Name für den SKilL/C#-Codegenerator auf *csharp* festgelegt. Er wird dabei so eingebunden, dass es möglich ist über die Kommandozeile Code für diese Programmiersprache zu generieren. Als nächster Schritt wird der komplette zu generierende Code so angepasst, dass er im Allgemeinen eine für C# gültige Syntax aufweist. In Listing 3.1 wird anhand von Beispielcode dargestellt welche konkreten Änderungen unter anderem vorgenommen werden. Dazu zählen die Anpassung von Schlüsselwörtern, den Importanweisungen und das Ersetzen von Paketen durch Namespaces. Weiterhin wird die Kennzeichnung von Vererbung sowohl im Klassenkopf,

Listing 3.1 Beispielcode zum Vergleich von Java (links) und C# (rechts)

```java
package Test;

import java.util.LinkedList;

public final class TestKlasse
  extends SuperTestKlasse
  implements TestInterface {

    final LinkedList<Integer> list;

    public TestKlasse() {
        super();
        list = new LinkedList<>();
    }

    @Override
    public final int testMethode() {
        return list.get(0);
    }
}


public class SuperTestKlasse {

    public int testMethode() {
        return 0;
    }
}
```

```csharp
using System.Collections.Generic;

namespace Test
{
    public sealed class TestKlasse
      : SuperTestKlasse, TestInterface
    {
        readonly List<int> list;

        public TestKlasse() : base()
        {
            list = new List<int>();
        }

        public sealed override int testMethode()
        {
            return list[0];
        }
    }


    public class SupertestKlasse
    {
        public virtual int testMethode()
        {
            return 0;
        }
    }
}
```

als auch in den entsprechenden Methodenköpfen angepasst. Außerdem muss die Verwendung von Collections, entsprechend der in Kapitel 2.2.1 erläuterten Unterschiede, angepasst werden. Zuletzt werden alle *throws*-Klauseln aus den Methodenköpfen entfernt, da weder diese noch eine äquivalente Funktionalität in C# zur Verfügung steht.

Typzuordnung

Im bestehenden SKilL/Java-Codegenerator werden die SKilL-Typen den entsprechenden Java-Typen zugeordnet. Für den SKilL/C#-Codegenerator müssen die Java-Typen mit den entsprechenden C#-Typen ersetzt werden. Dadurch wird eine Bedingung erfüllt, die nötig ist um die SKilL-Kernsprache zu unterstützen (siehe Kapitel 2.1.1). In Tabelle 3.1 wird für jeden SKilL-Typ dargestellt zu welchem Java-Typ er ursprünglich zugeordnet war und zu welchem C#-Typ er in der SKilL/C#-Implementierung zugeordnet ist. Dabei befindet sich der Typ *SkillObject* in den gemeinsam genutzten Implementierungsbestandteilen der entsprechenden Programmiersprache. Es fällt auf, dass für die SKilL-Typen `ConstantLengthArray` und `VariableLengthArray` für C# ein

SKilL-Typ	Java-Typ	C#-Typ
annotation	internal.SkillObject	@internal.SkillObject
bool	java.lang.Boolean	System.Boolean
i8	java.lang.Byte	System.SByte
i16	java.lang.Short	System.Int16
i32	java.lang.Integer	System.Int32
i64	java.lang.Long	System.Int64
v64	java.lang.Long	System.Int64
f32	java.lang.Float	System.Single
f64	java.lang.Double	System.Double
string	java.lang.String	System.String
ConstantLengthArray	java.util.ArrayList	System.Collections.ArrayList
VariableLengthArray	java.util.ArrayList	System.Collections.ArrayList
ListType	java.util.LinkedList	System.Collections.Generic.List
SetType	java.util.HashSet	System.Collections.Generic.HashSet
MapType	java.util.HashMap	System.Collections.Generic.Dictionary

Tabelle 3.1.: Zuordnung der SKilL-Typen zu den entsprechenden Typen der Programmiersprachen Java und C#

nicht-generischer Typ verwendet wird, während für Java ein generischer Typ verwendet wird. Der Grund hierfür liegt darin, dass der für Java verwendete Typ `ArrayList` in C# nur als nicht-generischer Typ vorhanden ist (siehe Kapitel 2.2.1).

Kommentare

Ein weiterer Punkt ist die Anpassung der Generierung von Kommentaren. Dabei werden die Dokumentationskommentare von Java durch die in C# verwendeten XML-Dokumentationskommentare ersetzt (siehe Kapitel 2.2.1). In der SKilL/C#-Implementierung wird allerdings nur das XML-Element `summary` zur Angabe von allgemeinen Informationen des entsprechenden Code-Elements verwendet. Durch diese Implementierung wird das Feature *documented* unterstützt (siehe Kapitel 2.1.1).

Escaping

Der bestehende SKilL/Java-Codegenerator enthält bereits eine Escaping-Strategie. Falls ein Java-Schlüsselwort erkannt wird, wird es am Anfang um den Buchstaben `Z` ergänzt. Dazu ist im Codegenerator eine Liste mit allen Java-Schlüsselwörtern vorhanden. Falls ein normaler Name den Buchstaben `Z` an einer Stelle enthält, wird dieser durch die Zeichenfolge `ZZ` ersetzt. So werden Überschneidungen mit bereits veränderten Schlüsselwörtern vermieden. Außerdem werden Zeichen, die nicht für Java-Bezeichner zugelassen sind, durch den Buchstaben *Z* gefolgt von ihrer hexadezimalen Representation ersetzt. Um alle zugelassenen Zeichen zu erkennen wird die Java-Methode `isJavaIdentifierPart` verwendet. Außerdem wird das Zeichen `:` durch das Zeichen `$` ersetzt.

Diese Escaping-Strategie wird größtenteils für den SKilL/C#-Codegenerator übernommen und nur an einigen Stellen angepasst. Dazu wird zum einen die Liste von Java-Schlüsselwörtern durch eine Liste aller C#-Schlüsselwörter ersetzt. Weiterhin wird das Zeichen : durch einen Unterstrich ersetzt, da das Zeichen `$` in C#-Bezeichnern nicht zugelassen ist (siehe Kapitel 2.2.1). Bereits im Namen vorhandene Unterstriche werden durch zwei Unterstriche hintereinander ersetzt, um Überschneidungen von Namen zu vermeiden. Da eine äquivalente und für C# geeignete Methode zu `isJavaIdentifierPart` weder in Java noch in Scala vorhanden ist, wird stattdessen die Methode `isLetterOrDigit` verwendet Diese erkennt Zeichen, die Buchstaben oder Zahlen sind. Da in C#-Bezeichnern nur Buchstaben, Zahlen und Unterstriche zugelassen sind (siehe Kapitel 2.2.1), wird so eine äquivalente Funktionsweise erreicht. Durch diese Implementierung wird das Feature *escaped* unterstützt (siehe Kapitel 2.1.1).

Typ

Der bestehende SKilL/Java-Codegenerator generiert für jeden spezifizierten Typen eine Klasse. Diese enthält die spezifizierten Attribute und die entsprechenden Getter- und Setter-Methoden. Weiterhin werden verschiedene Konstruktoren und eine innere Subtyp-Klasse generiert. Diese fungiert als generischer Platzhalter für unbekannte Subtypen.

Für den SKilL/C#-Codegenerator werden zum einen allgemeine Anpassungen am Code vorgenommen (siehe Kapitel 3.1.1). Zum anderen werden alle Getter- und Setter-Methoden durch Properties (siehe Kapitel 2.2.1) ersetzt. Normalerweise wird als Name eines Properties der Bezeichner des entsprechenden Attributs mit einem großen Anfangsbuchstaben verwendet. In C# ist es aber nicht zulässig, dass der Bezeichner der Klasse mit dem eines Elements innerhalb der Klasse übereinstimmt (siehe Kapitel 2.2.1). Deshalb wären Properties zu Attributen nicht möglich, wenn diese denselben Namen wie die Klasse hätten. Um solche Konflikte zu vermeiden, wird für Properties ein kleiner Anfangsbuchstabe verwendet. Allerdings würde so ein Konflikt mit dem Bezeichner des Attributs entstehen. Deshalb wird an den Anfang des Attribut-Bezeichners ein Unterstrich angefügt. Diese Taktik wird auch in der bestehenden SKilL/Scala-Implementierung angewendet. Wenn beispielsweise ein Attribut mit dem Namen `age` spezifiziert wird, würde es im generierten Code den Bezeichner `_age` verwenden. Das entsprechende Property würde in diesem Fall den Bezeichner `age` verwenden. In Listing 3.2 sieht man links die Implementierung einer Getter- und einer Setter-Methode in Java und rechts die Implementierung eines entsprechenden Properties in C#. Dabei werden Codeausschnitte verwendet, die mit dem jeweiligen Codegenerator erstellt wurden. Durch diese Implementierung werden die Features *auto* und *custom* unterstützt (siehe Kapitel 2.1.1).

Interface

Der bestehende SKilL/Java-Codegenerator generiert für jedes spezifizierte Interface ein Java-Interface. Dieses kann Methodenköpfe von Getter- und Setter-Methoden für ein Attribut enthalten. So kann deutlich gemacht werden, dass Klassen, die von diesem Interface erben, das entsprechende Attribut implementieren sollen.

Listing 3.2 Vergleich von Getter-/Setter-Methoden in Java (links) und einem Property in C# (rechts)

```
protected long age = 0;

public long getAge() {
    return age;
}

public void setAge(long age) {
    this.age = age;
}
```

```
protected long _age = 0;

public long age {
    get {return _age;}
    set {_age = value;}
}
```

Für den SKilL/C#-Codegenerator werden zum einen allgemeine Anpassungen am Code vorgenommen (siehe Kapitel 3.1.1). Zum anderen werden alle Getter- und Setter-Methoden durch Properties ersetzt, wie in Kapitel 3.1.1 beschrieben. Für diese Properties wird allerdings nur der Kopf implementiert. Dieser enthält dabei, je nachdem welche Zugriffsfunktionen verwendet werden sollen, die Schlüsselwörter `get` und `set`. Durch diese Implementierung wird das Feature *interfaces* unterstützt (siehe Kapitel 2.1.1).

Visitor

Der bestehende SKilL/Java-Codegenerator kann die Datei `Visitor` erzeugen. Diese enthält eine abstrakte Basisklasse, die zum Implementieren des Visitor-Patterns verwendet werden kann. Diese Funktion wird zusätzlich für den SKilL/C#-Codegenerator implementiert, da damit nahezu kein Aufwand verbunden ist.

SkillFile

Der bestehende SKilL/Java-Codegenerator erzeugt die Datei `SkillFile`. Diese enthält eine öffentliche Schnittstelle über die serialisierte SKilL-Dateien erstellt, eingelesen und verändert werden können. Für den SKilL/C#-Codegenerator muss dieses Interfaces zu einer abstrakten Klasse geändert werden. Dieses Änderung ist nötig, da es in C#, im Gegensatz zu Java, nicht möglich ist, statische Methoden in Interfaces zu implementieren. Durch diese Änderung muss zusätzlich auch die Vererbungsstruktur angepasst werden. In der bestehenden SKilL/Java-Implementierung erbt die Klasse `SkillState` aus der Datei `internal` sowohl von der Klasse `SkillState` aus den gemeinsam genutzten Implementierungsbestandteilen als auch von dem Interface `SkillFile` aus der Datei `SkillFile`. Da es in C# nicht möglich ist von mehr als einer Klasse zu erben, macht die Änderung des Interfaces `SkillFile` zu einer abstrakten Klasse die bisherige Vererbungsstruktur unmöglich. Um dieses Problem zu umgehen, erbt die Klasse `SkillState` nur noch von der abstrakten Klasse `SkillFile`. Die abstrakte Klasse `SkillFile` wiederum erbt von der Klasse `SkillState` aus den gemeinsam genutzten Implementierungsbestandteilen, wodurch auch die Klasse `SkillState` indirekt davon erbt. So konnte eine Vererbungsstruktur erstellt werden, die, bis auf die neue Superklasse von `SkillFile`, äquivalent zur ursprünglichen ist. Durch diese Implementierung wird eine Bedingung erfüllt, die nötig ist um die SKilL-Kernsprache zu unterstützen (siehe Kapitel 2.1.1).

Listing 3.3 Darstellung der Hilfsmethode `transformMapTypes`

```
def transformMapTypes(baseTypes: java.util.List[Type], count: Integer) : String = {
    if((baseTypes.size() - 1) == count) {
        mainCount -= 1;
        return s"${mapType(baseTypes.get(count))}"
    }
    else {
        return s"System.Collections.Generic.Dictionary<${mapType(baseTypes.get(count))}," +
          s"${transformMapTypes(baseTypes, count + 1)}>"
    }
}
```

Internal

Der bestehende SKilL/Java-Codegenerator erzeugt die Datei `internal`. Diese fungiert als Verbindungsstück zwischen den generierten Typen, der öffentlichen Schnittstelle und den gemeinsam genutzten Implementierungsbestandteilen. Für den SKilL/C#-Codegenerator muss die komplette Datei an die Änderungen der gemeinsam genutzten Implementierungsbestandteile (siehe Kapitel 3.2.2) angepasst werden. Dazu gehört unter anderem die Ersetzung von Wildcards verwendenden generischen Typen durch den entsprechenden nicht-generische Supertyp. Außerdem wird die neue Hilfsmethode `cast` an den entsprechenden Stellen verwendet. Weiterhin muss die Funktionsweise bezüglich geschachtelter Dictionaries angepasst werden. Da in C# die konkreten Typargumente von Collections auch bei deren Initialisierung angegeben werden müssen (siehe Kapitel 2.2.1), wird eine Hilfsmethode namens `transformMapTypes` im SKilL/C#-Codegenerator erstellt. In Listing 3.3 wird diese Methode dargestellt. Es kann erkannt werden, dass sie rekursiv aufgebaut ist und anhand einer Liste von Typen ein geschachteltes Dictionary erstellt.

3.2. Gemeinsam genutzte Implementierungsbestandteile

Ein großer Teil von SKilL besteht aus den sogenannten gemeinsam genutzten Implementierungsbestandteilen. Diese enthalten allen Code, der unabhängig von erstellten SKilL-Spezifikationen ist und müssen für jede Programmiersprache separat erstellt werden. Deshalb muss im Rahmen dieser Arbeit ebenfalls eine entsprechende Implementierung für C# erstellt werden. Dazu wird die bestehende SKilL/Java-Implementierung als Grundlage verwendet und anschließend bezüglich der Unterschiede zu C# (siehe Kapitel 2.2.1) angepasst. Insgesamt besteht die SKilL/Java-Implementierung der gemeinsam genutzten Implementierungsbestandteile aus 5904 Codezeilen.

3.2.1. Architektur

Die bestehende SKilL/Java-Implementierung besteht aus zwei Klassenbibliotheken mit den Namen `skill.jvm.common` und `skill.java.common`, die im Folgenden als JVM- und Java-Bibliothek bezeichnet werden. Dabei enthält die JVM-Bibliothek Code der das Lesen und Schreiben von Dateien ermöglicht Diese Bibliothek kann auch für andere Programmiersprachen, die die Java Virtual

Machine (JVM) verwenden, wie zum Beispiel Scala, verwendet werden. Die Java-Bibliothek enthält den restlichen benötigten Code und kann nur für Java verwendet werden. Für C# wird nur eine Klassenbibliothek namens `skill.csharp.common` erstellt. Diese vereint beide Klassenbibliotheken der SKilL/Java-Implementierungen in sich, da eine Auskoppelung der JVM-Bestandteile hier nicht nötig ist.

In Abbildung A.1 in Anhang A wird die Architektur der SKilL/C#-Implementierung der gemeinsam genutzten Implementierungsbestandteile in Form eines Klassendiagramms dargestellt. Dabei werden die Klassen entsprechend der Namespaces angeordnet, in denen sie jeweils enthalten sind. Die Namespace-Struktur der SKilL/C#-Implementierung ist dabei nahezu identisch zur Paket-Struktur der bestehenden SKilL/Java-Implementierung. Nur der Namespace `internal` muss umbenannt werden, da `internal` ein Schlüsselwort von C# ist und deshalb nicht als Namespace verwendet werden kann. Der Namespace wird in `@internal` umbenannt, wie in C# üblich (siehe Kapitel 2.2.1). Klassen, die für die SKilL/C#-Implementierung neu erstellt werden, sind in der Abbildung rot markiert.

Alle dargestellten Namespaces befinden sich im Namespace `de.ust.skill.common.csharp`. Im Namespace `api` sind Interfaces und abstrakte Klassen enthalten, die allgemeinen Zugriff auf Typen und Objekte gewähren. Der Namespace `restrictions` enthält zwei Interfaces, die als Obertyp für Feld- und Typrestriktionen fungieren und außerdem Klassen, die konkrete Restriktionen repräsentieren. Der Namespace `streams` enthält Klassen zum Einlesen und Schreiben von Daten aus Binärdateien. Der Namespace `@internal` enthält unter anderem verschiedene Iteratoren, Klassen zum Einlesen und Schreiben von SKilL-Zuständen und allgemeine Oberklassen für Feldtypen und -deklarationen. Außerdem enthält er die Klasse `StoragePool` und davon abgeleitete Klassen, die das Grundgerüst für den Serialisierungsprozess in SKilL bilden. Weiterhin enthält der Namespace selbst die Namespaces `exceptions`, `fieldDeclarations`, `fieldTypes` und `parts`. Der Namespace `exceptions` enthält verschiedene Typen von Ausnahmen, die geworfen werden können. Der Namespace `fieldDeclarations` enthält Klassen, die verschiedene konkrete Felddeklarationen repräsentieren. Der Namespace `fieldTypes` enthält Klassen, die verschiedene konkrete Feldtypen repräsentieren. Der Namespace `parts` enthält Klassen, die die Verwaltung von Felddaten übernehmen.

3.2.2. Erforderlichen Anpassungen für die Übertragung auf C#

Als Grundlage für die SKilL/C#-Implementierung der gemeinsam genutzten Implementierungsbestandteile wird die bestehende SKilL/Java-Implementierung übernommen. Insgesamt besteht die SKilL/C#-Implementierung der gemeinsam genutzten Implementierungsbestandteile aus 6854 Codezeilen. Durch sie werden ein Teil der Kernsprache und die zusätzlichen Features *append* und *lazy* implementiert (siehe Kapitel 2.2.1). Der Rest der Kernsprache und weitere Features werden durch den SKilL/C#-Codegenerator implementiert (siehe Kapitel 3.1). Als erstes wird der komplette Code so angepasst, dass er im Allgemeinen eine für C# gültige Syntax aufweist. In Kapitel 3.1.1 wird dieser Vorgang näher erläutert.

Listing 3.4 Darstellung von Wildcards in Java (links) im Vergleich zu einer Taktik um eine äquivalente Funktionsweise in C# zu erzeugen (rechts)

```
public class StoragePool<T, B> {

    public void Method1() {
        StoragePool<?, ?> pool;
        pool=new StoragePool<String, Integer>();
    }
}
```

```
public class StoragePool<T, B>
  : AbstractStoragePool
{
    public void Method1()
    {
        AbstractStoragePool pool;
        pool=new StoragePool<string, int>();
    }
}
```

Generische Typen und Wildcards

Als nächstes müssen die häufig in der bestehenden SKilL/Java-Implementierung vorkommenden Wildcards ersetzt werden, da diese in C# nicht zur Verfügung stehen (siehe Kapitel 2.2.1). Dazu wird sich an der SKilL/C++-Implementierung orientiert und die dort verwendete Lösung dieses Problems für die SKilL/C#-Implementierung übernommen und leicht angepasst. Die Idee ist es für einen generischen Typ, der Wildcards verwendet, einen nicht generischen Supertyp zu definieren. Dieser wird dann an allen Codestellen in denen Wildcards verwendet werden statt des generischen Typs eingefügt. Dadurch kann später, bei der eigentlichen Verwendung der entsprechenden Variablen, der generische Typ mit beliebigen Typargumenten verwendet werden. In Listing 3.4 wird anhand von einem Beispiel die eben beschriebene Taktik im Vergleich zu einer Java-Implementierung dargestellt.

Am leichtesten ist es als Supertyp ein Interface zu verwenden, da damit Mehrfachvererbung möglich ist und so die Vererbungshierarchie nicht verändert werden muss. Der Nachteil ist, das in einem Interface weder Attribute noch statische Methoden definiert werden können. Deshalb werden in der SKilL/C#-Implementierung für alle Fälle, bei denen eine dieser beiden Eigenschaften zwingend gegeben sein muss, abstrakte Klassen verwendet und sonst Interfaces. In Abbildung A.1 sind diese Klassen und Interfaces rot markiert.

Alle Methoden, die keine Typargumente der generischen Klasse verwenden, werden als Methodenköpfe im Supertyp deklariert. Die Implementierung der Methoden bleibt dabei in der generischen Klasse, nur statische Methoden werden im Supertyp implementiert. Falls der Supertyp zwingend auf Methoden, die generische Typargumente verwenden, zugreifen muss, werden diese dort ebenfalls als Methodenköpfe deklariert. Da im Supertyp aber keine generischen Typargumente verfügbar sind, werden diese durch den Typ `object` ersetzt, von dem sämtliche andere Typen erben. Dadurch können an diesen Stellen beliebige Typen eingesetzt werden. Wird der Typ `object` als Rückgabetyp verwendet, muss eventuell beim Aufruf der Methode eine Konvertierung des Rückgabewertes zum gewünschten Typ vorgenommen werden.

Namespace	generischer Typ	nicht-generischer Supertyp
api	Access<T>	IAccess
api	GeneralAccess<T>	IGeneralAccess
internal	BasePool<T>	IBasePool
internal	DynamicDataIterator<T, B>	IDynamicDataIterator
internal	FieldDeclaration<T, Obj>	AbstractFieldDeclaration
internal	InterfacePool<T, B>	IInterfacePool
internal	LazyField<T, Obj>	ILazyField
internal	StoragePool<T, B>	AbstractStoragePool
internal	UnrootedInterfacePool<T>	IUnrootedInterfacePool
internal.fieldDeclarations	AutoField<T, Obj>	IAutoField
internal.fieldTypes	ConstantIntegerType<T>	IConstantIntegerType
internal.fieldTypes	ConstantLengthArray<T>	IConstantLengthArray
internal.fieldTypes	ListType<T>	IListType
internal.fieldTypes	MapType<K, V>	IMapType
internal.fieldTypes	SetType<T>	ISetType
internal.fieldTypes	SingleArgumentType<T, Base>	ISingleArgumentType
internal.fieldTypes	VariableLengthArray<T>	IVariableLengthArray
restrictions	FieldRestriction<T>	IFieldRestriction

Tabelle 3.2.: Zuordnung der generischen Typen, die Wildcards verwenden, zu den entsprechenden nicht-generischen Supertypen

In Tabelle 3.2 sind alle generischen Typen, die an mindestens einer Stelle im Code Wildcards verwenden und die entsprechenden Supertypen abgebildet. Dabei wird zusätzlich angegeben in welchem Namespace sich die entsprechenden Typen befinden. Die Namen aller Supertypen, die Interfaces sind, beginnen mit dem großen Buchstaben `I`, abstrakte Klassen beginnen mit dem Wort `Abstract`.

AbstractStoragePool und FieldType

Auch für die Klasse `StoragePool` muss, wie oben beschrieben, ein nicht generischer Supertyp namens `AbstractStoragePool` erstellt werden. Die Klasse `StoragePool` erbt ursprünglich von der generischen Klasse `FieldType<T>` aus dem Namespace `internal` und gibt dabei einen ihrer Typparameter an `FieldType` weiter. Da `StoragePool` aber bereits von `AbstractStoragePool` erben muss, erbt stattdessen `AbstractStoragePool` von `FieldType`, wodurch auch `StoragePool` indirekt davon erbt. Allerdings kann so kein Typparameter mehr an `FieldType` weitergegeben werden, da `AbstractStoragePool` bewusst nicht generisch ist. Um dieses Problem zu umgehen, wird der generische Typparameter aus der Klasse `FieldType` und ihrer Superklasse, die ebenfalls `FieldType` heißt, sich allerdings im Namespace `api` befindet, entfernt. Alle Verwendungen des Typparameters innerhalb dieser Klassen und in allen überschriebenen Methoden von Subklassen werden zu dem Typ `object` abgeändert.

Weiterhin wird eine generische abstrakte Methode namens `cast<K,V>` in der Klasse `api.FieldType` deklariert. Diese wird für alle Klassen, die konkrete Container-Feldtypen darstellen, implementiert und ermöglicht eine Umwandlung von deren Typargumenten zu den in der Methode angegebenen

Typparametern. Dabei werden nur in der Klasse `MapType` beide Typparameter benötigt, da nur sie auch zwei Typargumente besitzt. In allen anderen Klassen wird der zweite Typparameter ignoriert und kann beim Methodenaufruf beliebig angegeben werden.

Iteratoren

In den gemeinsam genutzten Implementierungsbestandteilen existieren einige Klassen, die es ermöglichen über bestimmte Objekte der Implementierung zu iterieren. In der bestehenden SKilL/Java-Implementierung wird dies ermöglicht, indem die entsprechenden Klassen von der Klasse `Iterator` erben. Dort werden dann die zum Iterieren nötigen Methoden `hasNext` und `next` mit einer eigenen Implementierung überschrieben. In C# kann für eine äquivalente Funktionsweise die Klasse `System.Collections.IEnumerator` verwendet werden. Klassen die von `IEnumerator` erben, müssen dabei die Methoden `MoveNext`, `Reset` und das Property `Current` implementieren. `MoveNext` prüft dabei ob das Ende des Enumerators erreicht ist. Falls nicht wird der Enumerator auf das nächste Element gesetzt und `true` zurückgegeben, ansonsten wird `false` zurückgegeben. `Current` gibt das aktuelle Element zurück und `Reset` setzt den Enumerator auf die anfängliche Position zurück.

Für die SKilL/C#-Implementierung wird in jeder entsprechenden Iterator-Klasse eine private Klasse namens `Enumerator` deklariert, die von `IEnumerator` erbt. In Listing 3.5 wird sie am Beispiel der Iterator-Klasse `DynamicDataIterator` dargestellt. Dabei wird eine Instanz der Iterator-Klasse im Attribut `container` und das aktuelle Element im Attribut `current` gespeichert. Die Methode `MoveNext` und das Property `Current` wird dabei mithilfe der Methoden `hasNext` und `next` der Iterator-Klasse verwirklicht. In der Methode `Reset` wird, der als Methode ausgekoppelte, Konstruktor der Iterator-Klasse aufgerufen und das Attribut `current` auf `null` gesetzt. Außerdem wird die Methode `GetEnumerator` in jeder Iterator-Klasse implementiert. Mithilfe der aktuellen Instanz der Iterator-Klasse kann dadurch der entsprechende Enumerator erstellt und zurückgegeben werden. Dieser kann dann zum Beispiel zur Iteration in `foreach-Schleifen` verwendet werden.

ThreadPools und Sempahore

In der bestehenden SKilL/Java-Implementierung wird die Klasse `ThreadPoolExecutor` verwendet um asynchron Daten einzulesen und zu schreiben. Die Klasse baut eine Sammlung von Threads auf, den sogenannten ThreadPool [Oraa]. Objekte vom Typ `Runnable` können dann eine Ausführung ihrer `run`-Methode beim `ThreadPoolExecutor` beantragen. Diese wird dann von einem freien Thread des `ThreadPools` ausgeführt.

Für die SKilL/C#-Implementierung wird hierfür die statische Klasse `ThreadPool` verwendet, die einen ThreadPool zur Verfügung stellt. Rechts in Listing 3.6 kann man erkennen, wie mithilfe der Methode `QueueUserWorkItem` diesem ThreadPool ein Objekt vom Typ `WaitCallback` zugeweisen wird, das die auszuführenden Methoden enthält. Links in der Abbildung wird die bestehende SKilL/Java-Implementierung dargestellt, wobei `SkillState.pool` einen `ThreadPoolExecutor` enthält.

Auch die Verwendung von Semaphoren muss in diesem Zusammenhang leicht angepasst werden. In C# muss beim Erstellen eines Semaphors, zusätzlich zum anfänglichen Zählerstand, auch der maximal mögliche Zählerstand angegeben werden. Da diese Möglichkeit in Java nicht zur Verfügung steht und dort deshalb kein maximaler Zählerstand festgelegt wird, wird in der SKilL/C#Implementierung

Listing 3.5 Darstellung der privaten Klasse `Enumerator` am Beispiel der Iterator-Klasse `DynamicDataIterator`

```
private class Enumerator : IEnumerator
{
    DynamicDataIterator<T, B> container;

    T current;

    public Enumerator(DynamicDataIterator<T, B> container)
    {
        this.container = container;
    }

    public object Current
    {
        get
        {
            return current;
        }
    }

    public bool MoveNext()
    {
        bool hasNext = container.hasNext();
        if (hasNext)
        {
            current = (T)container.next();
        }
        return hasNext;
    }

    public void Reset()
    {
        container.Constructor((StoragePool<T,B>)container.p);
        current = null;
    }
}
```

Listing 3.6 Vergleich der Verwendung eines `ThreadPools` in der Java-Implementierung (links) und der C#-Implementierung (rechts)

```
SkillState.pool.execute(new Runnable() {
    @Override
    public void run() {
        s.allocateInstances(b);
        barrier.release();
    }
});
```

```
ThreadPool.QueueUserWorkItem(new WaitCallback(
  (Object stateInfo) => {
    s.allocateInstances(b);
    barrier.Release();
}));
```

Listing 3.7 Darstellung der Methode `clone` der Klasse `MappedOutStream` im Vergleich zwischen der Java-Implementierung (links) und der C#-Implementierung (rechts)

```
public MappedOutStream clone(int begin,
  int end) {
    ByteBuffer b = buffer.duplicate();
    b.position(begin);
    b.limit(end);
    return new MappedOutStream(b);
}
```

```
public MappedOutStream clone(int begin,
  int end) {
    MemoryStream ms = new MemoryStream(end);
    BinaryWriter b = new BinaryWriter(ms);
    MappedOutStream clonedMap =
      new MappedOutStream(b, file, begin);
    clonedMaps.Add(clonedMap);
    return clonedMap;
}
```

dafür die größte positive Integer-Zahl verwendet. Weiterhin ist es in C#, im Gegensatz zu Java, nicht möglich mehr als eine Erlaubnis auf einmal von einem Semaphor zu erhalten. Deshalb wird diese Funktionalität mithilfe einer entsprechend langen `for-Schleife` abgebildet.

Lesen und Schreiben von Daten

Im Paket `stream` der bestehenden SKilL/Java-Implementierung werden verschiedene Klassen aus Java für Lese- und Schreib-Operationen verwendet. Als Ersatz für diese müssen äquivalente Klassen aus C# gesucht werden müssen. Als Buffer wird in der SKilL/Java-Implementierung die Klasse `ByteBuffer` verwendet. In der SKilL/C#-Implementierung verwenden die Klassen `OutStream`, `FileOutputStream` und `MappedOutStream` dafür die Klasse `BinaryWriter` und die Klassen `InStream`, `FileInputStream` und `MappedInStream` die Klasse `BinaryReader`. `BinaryReader` baut dabei in der SKilL/C#-Implementierung auf einem `FileStream` und `BinaryWriter` auf einem `MemoryStream` auf. Die beiden Klassen stellen außerdem Methoden bereit mit denen es leichter ist spezifische Daten aus einer Datei einzulesen und zu schreiben. Außerdem wird die, in der SKilL/Java-Implementierung verwendete, Klasse `FileChannel` durch die C#-Klasse `FileStream` ersetzt.

Weiterhin muss vor allem die Klasse `MappedOutStream` stark angepasst werden, da in C# kein Äquivalent zur dort verwendeten Klasse `MappedByteBuffer` und der darin enthaltenen Methode `duplicate` vorhanden ist. Diese Methode sorgt dafür, dass ein Duplikat des entsprechenden Buffers erzeugt wird. Die aktuelle Position der Buffer ist zwar unabhängig voneinander, sie greifen aber trotzdem auf dieselben Daten zu. So können verschiedene Bereiche des Buffers zeitgleich bearbeitet werden. In der SKilL/Java-Implementierung wird diese Funktion verwendet, um große Datenmengen asynchron in verschiedene Teile eines Buffers und anschließend in eine Datei zu schreiben. Für die SKilL/C#-Implementierung wird diese Funktionalität umgesetzt, indem neue Instanzen der Klasse `MappedOutStream` erstellt werden, die jeweils die unterschiedlichen Teile des Buffers repräsentieren. Diese werden im ursprünglichen `MappedOutStream` als Liste gespeichert und können so am Ende alle zusammen in eine Datei geschrieben werden. In Listing 3.7 wird die Methode `clone` der Klasse `MappedOutStream`, in der die oben beschriebene Funktionalität umgesetzt wird, im Vergleich zwischen der Java- und der C#-Implementierung dargestellt.

Listing 3.8 Umwandlung von Big- zu Little-Endian beim Einlesen von Dateien am Beispiel des Integer-Datentyps

```
public int i32()
{
    byte[] bytes = input.ReadBytes(4);
    Array.Reverse(bytes);
    return BitConverter.ToInt32(bytes, 0);
}
```

Listing 3.9 Umwandlung von Little- zu Big-Endian beim Schreiben von Dateien am Beispiel des Integer-Datentyps

```
public void i32(int data)
{
    byte[] bytes = BitConverter.GetBytes(data);
    Array.Reverse(bytes);
    buffer.Write(bytes);
}
```

Eine weitere Anpassung die vorgenommen werden muss, bezieht sich auf die Unterschiede zwischen Java und C# bezüglich des Einlesens und Schreibens von Bytes in eine serialisierten Datei. Während in der bestehenden SKilL/Java-Implementierung bei der Verarbeitung von Bytes Big-Endian verwendet wird, wird in C# standardmäßig Little-Endian verwendet. Dadurch werden also die einzelnen Bytes in genau umgekehrter Reihenfolge eingelesen. Dies führt bei der Umwandlung zu konkreten Datentypen zu falschen Werten. Um dieses Problem zu beheben, werden, je nach Datentyp entsprechend viele, Bytes einzeln aus einer Datei eingelesen und in einem Array gespeichert. Die Reihenfolge des Arrays wird dann umgekehrt und es wird mithilfe der C#-Klasse `BitConverter` zum entsprechenden Datentyp konvertiert. Beim Schreiben von einem Datenwert wird dieser mithilfe des `BitConverters` zu einem Byte-Array konvertiert. Die Reihenfolge dieses Arrays wird dann umgekehrt und die Bytes einzeln in eine Datei geschrieben. In Listing 3.8 und 3.9 wird dieses Vorgehen anhand des Integer-Datentyps gezeigt.

Konvertierung von Strings

In der bestehenden SKilL/Java-Implementierung wird die Konvertierung von einem String in eine entsprechende Byterepräsentation mithilfe der Methode `getBytes` der Klasse String realisiert. Dabei wird die Kodierung, in diesem Fall `UTF-8`, mithilfe der Klasse `Charset` angegeben. In der SKilL/C#-Implementierung wird hierfür die Klasse `Encoding`, die die Kodierung festlegt, und deren Methode `GetBytes` verwendet. In Listing 3.10 werden diese Funktionalitäten anhand von Beispielcode abgebildet.

Für die Konvertierung von einem Byte-Array zu einem String wird in der SKilL/Java-Implementierung ein neuer String erstellt. Dabei werden als Konstruktorargumente das Byte-Array und die gewünschte Kodierung angegeben. In der SKilL/C#-Implementierung wird hierfür zunächst

Listing 3.10 Konvertierung von String zu Bytes im Vergleich zwischen der Java-Implementierung (links) und der C#-Implementierung (rechts)

```java
public String BytesToString(byte[] bytes)
{
    String s = null;
    try
    {
        s = new String(bytes, "UTF-8");
    }
    catch (UnsupportedEncodingException e)
    {
        e.printStackTrace();
    }
    return s;
}
```

```csharp
public string BytesToString(byte[] bytes)
{
    Encoding utf8 = Encoding.UTF8;
    char[] chars = utf8.GetChars(bytes);
    string s = null;
    try
    {
        s = new String(chars);
    }
    catch (InvalidOperationException e)
    {
        Console.Write(e.StackTrace);
    }
    return s;
}
```

Listing 3.11 Konvertierung von Bytes zu String im Vergleich zwischen der Java-Implementierung (links) und der C#-Implementierung (rechts)

```java
public byte[] StringToBytes(String s)
{
    Charset utf8 = Charset.forName("UTF-8");
    byte[] bytes = s.getBytes(utf8);
    return bytes;
}
```

```csharp
public byte[] StringToBytes(string s)
{
    Encoding utf8 = Encoding.UTF8;
    byte[] bytes = utf8.GetBytes(s);
    return bytes;
}
```

das Byte-Array, mithilfe der Klasse `Encoding`, die auch hier wieder die Kodierung festlegt, und deren Methode `GetChars` zu einem Char-Array umgewandelt. Mithilfe dieses neuen Arrays wird ein neuer String erstellt. In Listing 3.11 werden diese Funktionalitäten anhand von Beispielcode abgebildet.

4. Bereitstellung gemeinsam genutzter Implementierungsbestandteile

Ein wichtiger Bestandteil der meisten SKilL-Anbindung sind gemeinsam genutzte Implementierungsbestandteile. Um diese Implementierungsbestandteile möglichst leicht mit einer, vom entsprechenden Codegenerator erzeugten, Implementierung einer Spezifikation verwenden zu können, sollten sie dem Benutzer in einer Form zur Verfügung gestellt werden, die auf der verwendeten Plattform leicht eingebunden werden kann. In diesem Kapitel soll deshalb eine Strategie entwickelt und implementiert werden, um Benutzern von SKilL die gemeinsam genutzten Implementierungsbestandteile durch den SKilL/C#-Codegenerator zur Verfügung zu stellen.

Als erster Schritt wird die bestehende Strategie und Implementierung für die Programmiersprache Java betrachtet. Hier werden die gemeinsam genutzten Implementierungsbestandteile in zwei Java Archive (JAR)-Dateien gespeichert. Da Java-Code zuerst in einen Bytecode kompiliert wird, der dann mithilfe der JVM ausgeführt wird, können diese Bibliotheken unabhängig von der verwendeten Plattform verwendet werden [LYB+18]. Die Implementierung dazu befindet sich im bestehenden SKilL/Java-Codegenerator. Hier werden die bereits erstellten JAR-Dateien dann in einen Ausgabeordner kopiert, der bei der Codegenerierung einer Spezifikation festgelegt werden kann. Von dort können die JAR-Dateien in den restlichen Programmcode eingebunden werden.

Für die SKilL/C#-Implementierung konnte diese Taktik tatsächlich nahezu identisch übernommen werden. Statt JAR-Dateien werden in C# allerdings DLL-Dateien verwendet (siehe Kapitel 2.2). Zuerst war es nicht sicher, ob diese ohne weiteres auf Plattformen außer Windows eingebunden werden können. C#-Code wird, ähnlich zu Java, zuerst in eine Zwischensprache kompiliert und dann mithilfe einer entsprechenden Laufzeitumgebung ausgeführt (siehe Kapitel 2.2). Eine solche Laufzeitumgebung war aber ursprünglich mithilfe von .NET Framework nur auf Windows verfügbar. Inzwischen sind solche Laufzeitumgebungen, durch die Einführung von .NET Core und Mono, allerdings auch auf Linux und MacOS verfügbar. Dadurch stellt auch die Verwendung von entsprechenden DLL-Dateien kein Problem dar. Über die Entwicklungsumgebung *Visual Studio 2017* wird eine DLL-Datei erzeugt, die die gemeinsam genutzten Implementierungsbestandteile der Programmiersprache C# enthält. Diese DLL-Datei wird dann in einen Ausgabeordner kopiert, der bei der Codegenerierung einer Spezifikation festgelegt werden kann. Die Implementierung diese Vorgangs befindet sich im skill/C#-Codegenerator.

Ursprünglich sollte die entwickelte Strategie zum Bereitstellen von gemeinsam genutzten Implementierungsbestandteilen auch mit einer andere Programmiersprache kompatibel sein, die bereits eine Anbindung an SKilL, aber noch keine entsprechende Strategie besitzt. Die Strategie der SKilL/Java-Implementierung, die für die SKilL/C#-Implementierung verwendet wird, ist allerdings auf keine solche Programmiersprache übertragbar. Da die genutzte Strategie aber bereits optimal für die SKilL/C#-Implementierung benutzt werden kann, wird im Rahmen dieser Bachelorarbeit auf die Entwicklung einer anderen Strategie verzichtet.

5. Tests

Ein großer Teil dieser Bachelorarbeit besteht aus der konkreten Implementierung von Code. Um zu erreichen, dass dieser Code möglichst fehlerfrei funktioniert, wird er ausführlich getestet. In diesem Kapitel sollen deshalb die verschiedenen vorgenommenen Tests und deren Erstellung näher erläutert werden. Dazu wird zuerst näher auf die automatisch generierten Tests eingegangen. Danach wird beschrieben wie die Bereitstellung der gemeinsam genutzten Implementierungsbestandteile getestet wird. Außerdem wird die korrekte Funktionsweise der XML-Dokumentationskommentare getestet. Weiterhin werden die Tests zur Interoperabilität der SKilL/C#-Implementierung mit anderen SKilL-Anbindungen näher erläutert. Zuletzt werden noch Tests zur Leistung der SKilL/C#-Anbindung durchgeführt.

Dabei werden alle Tests zum einen auf dem Betriebssystem *Windows 10 Education x64 Version 1709* ausgeführt und zum anderen auf *Linux Ubuntu 18.04.1 LTS x86-64*. Das Linux-Betriebsystem läuft dabei in einer virtuellen Maschine, die mithilfe des Programms *Oracle VM VirtualBox* erstellt wird.

5.1. Automatisch generierte Tests

Automatisch generierte Tests sind eine gute Möglichkeit um ohne großen Aufwand viele Funktionen zu testen. Dazu muss ein Testgenerator erstellt werden, der es ermöglicht, aus zuvor bereitgestellten Testdaten, Tests für die entsprechende Programmiersprache zu erstellen. Dadurch kann viel repetitive Programmierarbeit eingespart werden, da der Testgenerator nur einmalig implementiert werden muss. Außerdem können so fehlende Testfälle durch die Bereitstellung der entsprechenden Testdaten unkompliziert hinzugefügt werden. Für SKilL sind die Testdaten im Verzeichnis `skill/src/test/resources` hinterlegt und bestehen aus Binärdateien, Spezifikationsdateien und JavaScript Object Notation (JSON)-Dateien, welche zur Erstellung von konkreten Objekten verwendet werden [HZM17].

5.1.1. Implementierung des Testgenerators

Zur Implementierung eines Testgenerators für die SKilL/C#-Implementierung wird als Grundlage der bereits existierende Testgenerator der SKilL/Java-Implementierung verwendet und entsprechend an C# angepasst. Zuerst erstellt der Testgenerator dabei mithilfe des entsprechenden Codegenerators für jeden Testfall eine Implementierung zu den Spezifikationen, die in den Testdaten bereitgestellt werden. Die erste Anpassung ist es also dem Testgenerator den SKilL/C#-Codegenerator, statt den SKilL/Java-Codegenerator zuzuweisen.

Während für Java das Framework *JUnit* zum Erstellen der Tests verwendet wird, wird für C# das Framework *NUnit* verwendet. NUnit ist dabei, bis auf minimale Unterschiede, nahezu identisch zu JUnit. Entsprechend wird die Syntax der zu generierenden Tests so angepasst, dass sie für NUnit gültig ist.

Die Implementierung des Testgenerators selbst ist in zwei Klassen aufgeteilt, die unterschiedliche Arten von Test erzeugen. Die Klasse `GenericTests` erzeugt dabei pro Spezifikation in den Testdaten eine Datei namens `GenericReadTest`, die mehrere Testfälle enthält. In Listing 5.1 wird ganz oben der Testfall `writeGeneric` dargestellt, der mithilfe der Methode `reflectiveInit` Beispieldaten erstellt und in eine temporäre Datei schreibt. Direkt darunter kann man den Testfall `writeGenericChecked` erkennen, der zusätzlich noch überprüft ob die richtige Anzahl an Beispieldaten in die temporäre Datei geschrieben wurde. Weiterhin wird für jede Binärdatei in den Testdaten, ein Testfall erstellt, der entweder einen Erfolg oder eine Exception beim Einlesen der Binärdatei erwartet. Die zwei Testfälle, die unten in der Abbildung dargestellt werden sind Beispiele dafür. Ein Erfolg bedeutet dabei vorerst nur, dass es keine Fehler beim Einlesen der Datei gab und nicht, dass die Struktur der Datei wirklich vollkommen korrekt ist, auch wenn das in diesem Fall wahrscheinlich ist. In der Klasse `GenericTests` muss, außer den oben bereits genannten Anpassung, nichts verändert werden.

Die Klasse `APITests` erzeugt pro Spezifikation in den Testdaten eine Datei namens `GenericAPITest`. Diese enthält Testfälle, die überprüfen ob konkrete Objekte erfolgreich in eine Binärdatei geschrieben und wieder ausgelesen werden können. Ein Beispiel eines solchen Testfalls wird in Listing 5.2 dargestellt. Dazu werden die JSON-Dateien aus den Testdaten verwendet, die angeben welche Objekte geschrieben werden sollen. Außerdem wird hier angegeben, ob der Testfall eine Exception erwartet oder nicht. In dieser Klasse muss, zusätzlich zu den oben bereits genannten Anpassungen, der Vergleich von Container-Objekten beim Einlesen geändert werden. In der SKilL/Java-Implementierung wird hierfür die Methode `Equals` verwendet. Da diese aber in C# nicht für jeden Fall korrekte Ergebnisse liefert, wird stattdessen die Methode `SequenceEqual` aus der C#-Klasse `Enumerable` verwendet.

In den generierten Tests werden zusätzlich auch Hilfsmethoden aus den Klassen `CommonTest` und `ForceLazyFields` verwendet. Da diese Klassen speziell für die SKilL/Java-Implementierung erstellt wurden, mussten sie zuerst an C# angepasst werden, wobei keine Besonderheiten oder Probleme aufgetreten sind. In der Klasse `CommonTest` wird allerdings die Variable `basePath` hinzugefügt, die den Pfad des SKilL-Verzeichnisses enthält. Dadurch ist es möglich diesen Pfad jederzeit unkompliziert an die aktuelle Arbeitsumgebung anzupassen.

5.1.2. Testergebnisse

Die Tests werden unter Windows in der Entwicklungsumgebung *Visual Studio 2017* mithilfe der Erweiterung *NUnit3TestAdapter* und unter Linux in der Entwicklungsumgebung *MonoDevelop* ausgeführt. Es werden dabei alle Testdaten verwendet, die auch für die Tests der bestehenden SKilL/Java-Implementierung verwendet werden. Unter Windows werden dabei die API-Tests zu zwei Spezifikationen allerdings nicht korrekt generiert. Zu der Spezifikation `escaping` werden überhaupt keine API-Tests erstellt. Stattdessen wird nur eine leere Datei generiert. Zu der Spezifikation `graph` werden fälschlicherweise dieselben API-Tests wie zu der Spezifikation `graphInterface` erzeugt. Die Ursachen zu diesen beiden Fehlern konnten auch nach längerer Suche nicht ermittelt werden.

Listing 5.1 Ein Ausschnitt der automatisch generierten ReadTests

```
[Test]
public void writeGeneric() {
    string path = tmpFile("write.generic");
    SkillFile sf = SkillFile.open(path);
    reflectiveInit(sf);
    sf.close();
    File.Delete(path);
}

[Test]
public void writeGenericChecked() {
    string path = tmpFile("write.generic.checked");
    SkillFile sf = SkillFile.open(path);
    reflectiveInit(sf);
    // write file
    sf.flush();

    // create a name -> type map
    Dictionary<string, IAccess> types = new Dictionary<string, IAccess>();
    foreach (IAccess t in sf.allTypes())
        types[t.Name] = t;
    // read file and check skill IDs
    SkillFile sf2 = SkillFile.open(path, Mode.Read);
    foreach (IAccess t in sf2.allTypes()) {
        IEnumerator os = types[t.Name].GetEnumerator();
        foreach (SkillObject o in t) {
            Assert.IsTrue(os.MoveNext(), "to few instances in read state");
            Assert.AreEqual(o.SkillID, ((SkillObject)os.Current).SkillID);
        }
        Assert.IsFalse(os.MoveNext(), "to many instances in read state");
    }
    File.Delete(path);
}

[Test]
public void test_age_read_accept_age_example_sf() {
    Assert.IsNotNull(read("src\\test\\resources\\genbinary" +
    "\\[[empty]]\\accept\\age-example.sf"));
}

[Test]
public void test_age_read_reject_duplicateDefinition_sf() {
    try {
        ForceLazyFields.forceFullCheck(read("src\\test\\resources\\genbinary" +
          "\\[[all]]\\fail\\duplicateDefinition.sf"));
        Assert.Fail("Expected ParseException to be thrown");
    } catch (SkillException e) {
        return; // success
    }
}
```

Listing 5.2 Darstellung eines automatisch generierter APITests

```
[Test]
public void APITest_core_age_acc_one() {
    string path = tmpFile("one");
    SkillFile sf = SkillFile.open(path, Mode.Create, Mode.Write);

    // create objects
    age.Age one = (age.Age)sf.Ages().make();
    // set fields
    one.AgeProp = (long)30L;
    sf.close();

    { // read back and assert correctness
        SkillFile sf2 = SkillFile.open(sf.currentPath(), Mode.Read, Mode.ReadOnly);
        // check count per Type
        Assert.AreEqual(1, sf.Ages().staticSize());
        // create objects from file
        age.Age one_2 = (age.Age)sf2.Ages().getByID(one.SkillID);
        // assert fields
        Assert.IsTrue(one_2.AgeProp == 30L);
    }
    File.Delete(path);
}
```

Da die API-Tests zu den beiden Spezifikationen allerdings unter Linux korrekt generiert werden, werden stattdessen diese auch unter Windows zum Testen verwendet werden. Da unter Windows und Linux normalerweise dieselben Tests generiert werden sollten, können so auch unter Windows alle vorgesehenen Tests ausgeführt werden.

Insgesamt wurden 804 Tests zu 23 Spezifikationen ausgeführt, von denen 800 erfolgreich waren und vier fehlgeschlagen sind. Im Folgenden wird auf die fehlgeschlagenen Tests und die entsprechenden Fehlerursachen eingegangen. Die Tests sind dabei anhand der Fehlerursache zu Gruppen zusammengefasst.

Konvertierung von geschachtelten Dictionaries: In den Testfällen `APITest_core_container_acc_make` und `test_container_read_accept_container_sf` werden unter anderem geschachtelte Map-Typen verwendet, die in C# als Dictionaries implementiert sind. Dabei muss an einer Stelle im Code eine Konvertierung an einem geschachtelten Dictionary vorgenommen werden um die Typparameter zu verändern. Die für die SKilL/C#-Implementierung erstellte Methode, um Dictionaries zu konvertieren, funktioniert aber nicht mit geschachtelten Dictionaries. Da die Implementierung dieser Funktion relativ aufwändig wäre und angenommen wird das geschachtelte Map-Typen in realen Anwendungen nur selten verwendet werden, wird dieser Fehler im Rahmen dieser Bachelorarbeit nicht behoben.

Null-Schlüssel in einer Map: Im Testfall `APITest_core_map3_acc_simple` wird unter anderem ein `null`-Schlüssel in einem Map-Type verwendet. Während die Maps aus Java `null` als Schlüsselwert ohne Probleme akzeptieren, unterstützen die Dictionaries aus C# diesen Schlüssel nicht (siehe Kapitel 2.2.1). Da eine Umgehung dieser Beschränkung relativ aufwändig wäre, wird dieser Fehler im Rahmen dieser Bachelorarbeit nicht behoben.

EndOfStreamException: Bei der Ausführung des Testfalls `APITest_interfaces_graphInterface_acc_succ__1` wird eine *EndOfStreamException* mit der folgenden Fehlermeldung ausgegeben: *Über das Ende des Datenstroms hinaus kann nicht gelesen werden.* Dieser Fehler ist zwar im Verlauf des Testprozesses bereits häufiger aufgetreten, allerdings konnten diese alle behoben werden. Die Ursache für diesen speziellen Fehler konnte auch nach längerer Suche nicht ermittelt werden, weshalb er nicht behoben werden konnte.

5.2. Tests zur Bereitstellung der gemeinsam genutzten Implementierungsbestandteile

In Kapitel 4 wird eine Strategie zur Bereitstellung der gemeinsam genutzten Implementierungsbestandteile durch den SKilL/C#-Codegenerator implementiert. Diese muss ebenfalls getestet werden um ihre Funktionalität zu überprüfen. Dazu wird zuerst die entsprechende DLL-Datei erstellt und dann versucht diese manuell in ein C#-Projekt einzubinden, das die automatisch generierten Tests enthält. Unter Windows wird dafür die Entwicklungsumgebung *Visual Studio 2017* und unter Linux *MonoDevelop* verwendet. Der Prozess des Einbindens konnte dabei ohne Probleme durchgeführt werden. Anschließend werden dann die im Projekt enthaltenen automatisch generierten Tests ausgeführt, um die Funktionalität der DLL-Datei zu verifizieren. Unter Windows wird hierfür .NET Framework und unter Linux .NET Core verwendet. Dabei wurden die gleichen Ergebnisse wie bereits in Kapitel 5.1.2 beobachtet. Dadurch wurde verifiziert, dass die implementierte Strategie zur Bereitstellung der gemeinsam genutzten Implementierungsbestandteile fehlerfrei funktioniert.

5.3. Tests zu Dokumentationskommentaren

In einer SKilL-Spezifikation können Kommentare angegeben werden, die dann vom SKilL/C#-Codegenerator zu XML-Dokumentationskommentaren umgewandelt werden (siehe Kapitel 3.1.1). Aus diesen Kommentaren kann dann eine XML-Dokumentationsdatei erstellt werden. Im Folgenden wird getestet, ob XML-Dokumentationskommentare fehlerfrei funktioniert. In der Entwicklungsumgebung *Visual Studio 2017* werden XML-Dokumentationskommentare als Overlay bei den entsprechenden Code-Elementen angezeigt. In Abbildung 5.1 kann man am Beispiel der Klasse `Date` erkennen, dass diese Funktion funktioniert. Der Text, der im XML-Dokumentationskommentar zwischen den `summary`-Tags steht, wird genauso im Overlay angezeigt. Weiterhin wird getestet ob eine XML-Dokumentationsdatei erstellt werden kann. Dazu wird zu den in Kapitel 5.1 generierten Spezifikationen eine XML-Dokumentationsdatei erstellt. Diese wird dann anhand von Stichproben mit denen im generierten Code vorhandenen Kommentaren verglichen. Da hierbei keine Fehler gefunden wurden, konnte verifiziert werden, dass die XML-Dokumentationskommentare fehlerfrei funktionieren.

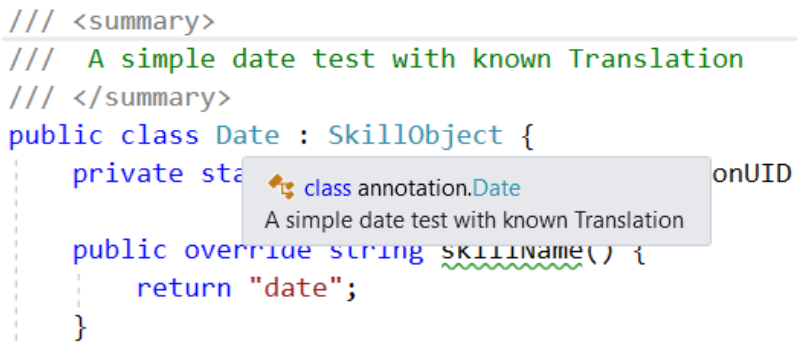

Abbildung 5.1.: Overlay in Visual Studio 2017 für einen XML-Dokumentationskommentar

Listing 5.3 Erzeugung einer Binärdatei durch die SKilL/Java-Implementierung

```
@Test
public void APITest_core_age_acc_one() throws Exception {
    String path = basePath + "age_one.sf";
    SkillFile sf = SkillFile.open(path, Mode.Create, Mode.Write);

    // create objects
    age.Age one = sf.Ages().make();
    // set fields
    one.setAge(30L);
    sf.close();
}
```

5.4. Tests zur Interoperabilität

Ein weiterer Punkt, der getestet werden sollte, ist die Interoperabilität der neu geschaffenen SKilL/C#-Implementierung mit den bereits existierenden SKilL-Anbindungen für andere Programmiersprachen. Konkret wird im folgenden die Interoperabilität zwischen der C#- und der Java-Implementierung getestet. Dabei muss sowohl das Einlesen von durch die SKilL/Java-Implementierung generierten Binärdateien durch die SKilL/C#-Implementierung, als auch die umgekehrte Richtung getestet werden.

Als Grundlage dafür werden die automatisch generierten API-Tests der beiden Sprachen verwendet und angepasst. In Listing 5.3 kann man den Teil eines API-Tests der SKilL/Java-Implementierung erkennen, der für die Generierung einer Binärdatei und das Schreiben von Objekten in diese verantwortlich ist. Dieser Teil wird allerdings noch so angepasst, dass keine temporäre Datei mehr erzeugt wird. Stattdessen wird die erzeugte Datei in einem Verzeichnis abgelegt, das über das Attribut `basePath` der Klasse `CommonTest` festgelegt wird.

In Listing 5.4 ist der Teil eines API-Tests der SKilL/C#-Implementierung abgebildet, der für das Einlesen einer Binärdatei und das Vergleichen der Objekte verantwortlich ist. Dieser Teil wird dabei noch so angepasst, dass die Datei aus dem Verzeichnis eingelesen wird, das im Attribut `basePath` der Klasse `CommonTest` festgelegt wird. Außerdem wird am Anfang eine temporäre Binärdatei erzeugt und Objekte darin gespeichert. Dieses Vorgehen ist nötig um Objekte mit denselben *SkillIDs*,

Listing 5.4 Einlesen einer Binärdatei durch die SKilL/C#-Implementierung

```
[Test]
public void APITest_core_age_acc_one() {
    string tmpPath = tmpFile("one");
    SkillFile sf = SkillFile.open(tmpPath, Mode.Create, Mode.Write);
    // create objects
    age.Age one = (age.Age)sf.Ages().make();
    // set fields
    one.AgeProp = (long)30L;
    sf.close();
    File.Delete(tmpPath);

    string path = basePath + "age_one.sf";
    SkillFile sf2 = SkillFile.open(path, Mode.Read, Mode.ReadOnly);
    //create objects from file
    age.Age one_2 = (age.Age)sf2.Ages().getByID(one.SkillID);
    //assert fields
    Assert.IsTrue(one_2.AgeProp == 30L);
    sf2.close();
}
```

wie sie in Listing 5.3 erzeugt werden, zu erhalten. Lässt man nun die von Listing 5.3 generierte Binärdatei von Listing 5.4 einlesen, ist eine Richtung der benötigten Tests abgebildet. Für die andere Richtung tauscht man einfach die jeweils verwendete Implementierung der beiden Teile aus. Das heißt zuerst wird mithilfe der SKilL/C#-Implementierung eine Binärdatei erzeugt, die dann mithilfe der SKilL/Java-Implementierung eingelesen wird.

Dieses Vorgehen wird für alle automatisch generierten API-Tests durchgeführt, die auch in Kapitel 5.1.2 verwendet werden und bei denen keine Fehler aufgetreten sind. Insgesamt werden so 24 Tests pro Testrichtung ausgeführt. Da diese Tests alle erfolgreich waren, ist die fehlerfreie Interoperabilität der neu geschaffenen SKilL/C#-Implementierung mit der bereits bestehenden SKilL/Java-Implementierung bestätigt.

5.5. Tests zur Leistung

Ein weiterer Punkt der getestet wird, ist die Leistung der neu geschaffenen SKilL/C#-Implementierung im Vergleich zur bestehenden SKilL/Java-Implementierung. Dazu werden Spezifikationen und Binärdateien aus der Masterarbeit *Skilled LLVM* von Pfister [Pfi18] verwendet. Zum Testen wird für fünf Binärdateien in verschiedenen Größen die Zeit gemessen, die für das Einlesen und Schreiben benötigt wird. Pro Datei werden zehn Messungen durchgeführt und dann der Mittelwert gebildet. So kann eine Verfälschung der Messwerte, durch eventuell auftretende Schwankungen der Leistung, ausgeglichen werden. In Tabelle 5.1 werden die gemessen Zeiten für das Einlesen und Schreiben der einzelnen Dateien dargestellt. Dabei werden die Zeiten der SKilL/C#-Implementierung denen der SKilL/Java-Implementierung gegenübergestellt. Die gemessenen Zeiten werden auf Zehntelsekunden gerundet angegeben.

Dateiname	Dateigröße	Lesezeit		Schreibzeit	
		Java	C#	Java	C#
make.sf	1128 Kb	00.02 s	00.59 s	00.12 s	00.10 s
bash.sf	6176 Kb	00.05 s	03.56 s	00.29 s	00.60 s
git.sf	15870 Kb	00.28 s	08.32 s	00.70 s	01.29 s
llvm-dis.sf	24417 Kb	00.55 s	25.14 s	01.50 s	02.13 s
llvm-split.sf	52652 Kb	02.02 s	52.00 s	03.37 s	05.23 s

Tabelle 5.1.: Lese- und Schreibzeit im Vergleich zwischen Java und C#

Bei den ersten Versuchen die Dateien einzulesen traten allerdings Fehler auf. Die Ursache konnte schnell auf den Vorgang des asynchronen Einlesens eingegrenzt werden. Da aber die genaue Ursache auch nach längerer Suche nicht ermittelt werden konnte, wurden diese Fehler nicht behoben. Stattdessen wird die Eigenschaft Daten asynchron einzulesen entfernt. Die grundsätzliche Funktionalität der SKilL/C#-Implementierung wird dabei nicht verändert. Allerdings wird dadurch die Leistung beim Einlesen gesenkt. Das kann gut in Tabelle 5.1 beobachtet werden. Die Lesezeit ist hier bei der SKilL/C#-Implementierung wesentlich höher als bei der SKilL/Java-Implementierung. Die Schreibzeit dagegen ist bei beiden Implementierungen ungefähr gleich groß.

6. Zusammenfassung und Ausblick

SKilL ist eine Plattform, die entwickelt wurde, um die Serialisierung von großen Datenmengen innerhalb von Werkzeugketten, in denen unterschiedliche Programmiersprachen zum Einsatz kommen, zu ermöglichen. Die Programmiersprache C# ist stark verbreitet und wird aktiv von Microsoft weiterentwickelt. Das Ziel dieser Bachelorarbeit ist es deshalb eine entsprechende SKilL-Anbindung für C# zu implementieren und zu testen.

Dazu wird der Codegenerator und die gemeinsam genutzten Implementierungsbestandteile, auf der Grundlage der SKilL/Java-Anbindung, implementiert und an C# angepasst. Bei der Implementierung des Codegenerators werden so die Zuordnung der Typen, die Escaping-Strategie und die Behandlung von Kommentaren angepasst. Weiterhin werden Properties eingeführt und die generierten Dateien `SkillFile`, `internal` und `Visitor` angepasst. Bei der Implementierung der gemeinsam genutzten Implementierungsbestandteile wird die Verwendung von Wildcards bei generischen Typen durch entsprechende nicht-generische Supertypen ersetzt. Außerdem werden einige Anpassung für das Lesen und Schreiben von Dateien vorgenommen, die Konvertierung von Strings und die Verwendung von Semaphoren angepasst. Zuletzt werden Iteratoren und die Verwendung eines ThreadPools durch äquivalente Funktionalitäten aus C# ersetzt. Insgesamt werden so die SKilL-Kernsprache und alle verpflichtenden Features implementiert. Zusätzlich werden auch die Features *append, interfaces, lazy* und *view* implementiert.

Weiterhin wird eine Strategie vorgestellt und implementiert, um die gemeinsam genutzten Implementierungsbestandteile durch den SKilL/C#-Codegenerator bereitzustellen. Bei der, in der bestehenden SKilL/Java-Implementierung angewandten, Strategie werden die gemeinsam genutzten Implementierungsbestandteile als JAR-Datei gespeichert und in das entsprechende Ausgabeverzeichnis kopiert. Diese Strategie kann so auch für die SKilL/C#-Implementierung angewendet werden, und wird deshalb übernommen. Dabei werden die gemeinsam genutzten Implementierungsbestandteile allerdings als DLL-Datei gespeichert.

Zuletzt werden alle implementierten Funktionalitäten ausführlich getestet. Dazu wird zuerst die Implementierung eines Testgenerators beschrieben. Die damit automatisch generierten Tests werden dann ausgeführt und waren, bis auf wenige Fehlschläge, erfolgreich. Weiterhin werden Tests durchgeführt, um die fehlerfreie Funktionalität, der in Kapitel 4 implementierten Strategie und die Interoperabilität der neu geschaffenen SKilL/C#-Implementierungen mit der bestehenden SKilL/Java-Implementierung zu bestätigen. Diese Tests waren dabei alle erfolgreich. Zuletzt wird noch die Leistung der SKilL/C#-Implementierung im Vergleich zur SKilL/Java-Implementierung getestet.

Ausblick

In dieser Bachelorarbeit wurde eine SKilL-Anbindung für C# entwickelt, die alle Features unterstützt, die auch von der bestehenden SKilL/Java-Implementierung unterstützt werden. Das sind allerdings noch nicht alle Features, die für eine SKilL-Anbindung implementiert werden können. Als nächstes könnten also die noch fehlenden Features implementiert werden. Dabei handelt es sich um *conversion, enums, hint, limits, multi-state, restricted, safe* und *typedefs*. Außerdem waren die, im Rahmen dieser Arbeit, durchgeführten Tests nicht alle erfolgreich. Im Folgenden sollten also die Ursachen der aufgetretenen Fehler ermittelt und anschließend behoben werden. Weiterhin musste die Eigenschaft Daten asynchron einzulesen entfernt werden, da hierbei Fehler auftraten. Da so aber die Leistung beim Einlesen erheblich gesenkt wird, sollten die entsprechenden Fehler behoben und die Funktionalität wieder hinzugefügt werden.

A. Architektur der gemeinsam genutzen Implementierungsbestandteile

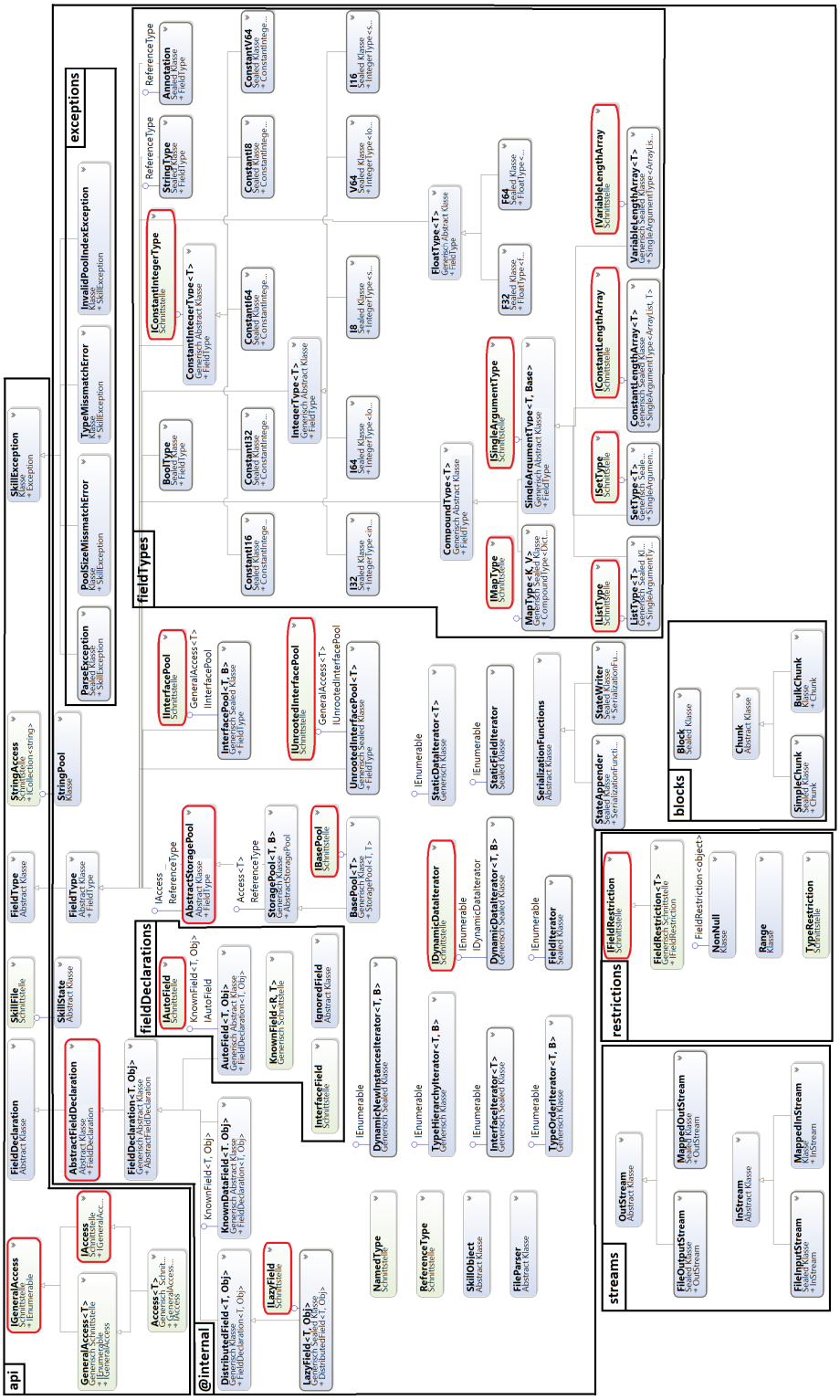

Abbildung A.1.: Klassendiagramm zu den gemeinsam genutzten Implementierungsbestandteilen von C#

Literaturverzeichnis

[AA17] J. Albahari, B. Albahari. *C# 7.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, Inc., 2017. ISBN: 1491987626 (zitiert auf S. 11).

[Ecm17] Ecma International. *ECMA-334 – C# Language Specification*. 2017 (zitiert auf S. 7, 11–15).

[Fel17] T. Felden. *The SKilL Language V1.0*. 2017 (zitiert auf S. 7, 9, 10).

[GJS+18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith. *The Java® Language Specification – Java SE 11 Edition*. 2018 (zitiert auf S. 12–15).

[HZM17] M. Hermann, P. von Zameck Glyscinski, P. Martis. *Sprachunabhängiges Testen sprachunabhängiger Serialisierung*. 2017 (zitiert auf S. 33).

[LYB+18] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, D. Smith. *The Java® Virtual Machine Specification – Java SE 11 Edition*. 2018 (zitiert auf S. 31).

[Mica] Microsoft. *MSDN Library – Assemblies kurz und knapp*. https://msdn.microsoft.com/de-de/library/bb978926.aspx (zitiert auf S. 11).

[Micb] Microsoft. *.NET-Documentation – Compiling Apps with .NET Native*. https://docs.microsoft.com/de-de/dotnet/framework/net-native/ (zitiert auf S. 11).

[Micc] Microsoft. *.NET-Documentation – Dictionary<TKey,TValue> Class*. https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.dictionary-2?view=netframework-4.7.2 (zitiert auf S. 14).

[NMRW17] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. O'Reilly Media, Inc., 2017. ISBN: 1491987626 (zitiert auf S. 11).

[Oraa] Oracle. *Java® Platform, Standard Edition and Java Development Kit Version 10 API Specification – Class ThreadPoolExecutor*. https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ThreadPoolExecutor.html (zitiert auf S. 26).

[Orab] Oracle. *Java SE Documentation – How to Write Doc Comments for the Javadoc Tool*. https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#format (zitiert auf S. 14).

[Pfi18] D. Pfister. *Skilled LLVM*. 2018 (zitiert auf S. 39).

[Sta13] I. O. for Standardization. *ISO/IEC JTC1 SC22 WG21 N 3690*. 2013 (zitiert auf S. 12).

Alle URLs wurden zuletzt am 28. 11. 2018 geprüft.

Verzeichnis der Listings

Abkürzungsverzeichnis

CIL Common Intermediate Language. 11

CLR Common Language Runtime. 11

DLL Dynamic Link Library. 11

JAR Java Archive. 31

JSON JavaScript Object Notation. 33

JVM Java Virtual Machine. 22

SKiIL Serialization Killer Language. 3

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