HAW HAMBURG

BACHELOR THESIS
Till Vincent Aul

# Design und Integration einer Autovervollständigung für partielle OCL-Ausdrücke in einem UML-Modellierungswerkzeug

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Till Vincent Aul

# Design und Integration einer Autovervollständigung für partielle OCL-Ausdrücke in einem UML-Modellierungswerkzeug

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Lars Hamann
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 30.04.2024

**Till Vincent Aul**

**Thema der Arbeit**

Design und Integration einer Autovervollständigung für partielle OCL-Ausdrücke in einem UML-Modellierungswerkzeug

**Stichworte**

intellisense, uml, ocl, integration

**Kurzzusammenfassung**

Da Modellierung ein wichtiger Teil der Softwareentwicklung ist, ist eine effiziente Verwendung des zur Modellierung verwendeten Werkzeugs essenziell, um effizient Software entwickeln zu können. Diese Modellierungen werden häufig in der Unified Modeling Language (UML) bzw. in der Object Constraint Language (OCL) beschrieben. Da die Einbindung einer Autovervollständigung die Nutzung eines Modellierungswerkzeugs effizienter machen kann und dies in dem für OCL gängigen Modellierungstool UML-based Specification Environment (USE) noch nicht integriert ist, wird in dieser Arbeit eine Autovervollständigung für OCL entworfen und anschließend in USE integriert.
Die Architektur basiert dabei auf der Grundlage der Archtiektur von Compilern und verwendet das Model-View-Controller-Pattern (MVC-Pattern) zur Festlegung des Informationsflusses. Die entscheidende Komponente ist der Parser, bei welchem durch Iterationen und Interpretationen von Teilergebnissen des in USE verwendeten OCL-Parsers auch partielle OCL-Ausdrücke erkannt und von der Autovervollständigung unterstützt werden können.

**Till Vincent Aul**

**Title of Thesis**

Design and integration of an auto completion for partial OCL-Statements in a UML modelling tool

**Keywords**

intellisense, uml, ocl, integration

**Abstract**

As modelling is an integral part of software development, efficient use of modeling tools is essential for effective software development. These models are often described in the UML or in the OCL. Since the integration of autocomplete functionality can enhance the usability of a modeling tool, and this feature is not yet integrated into the widely used modeling tool USE for OCL, this work proposes the design and integration of an autocomplete feature for OCL in USE.

The architecture is based on compiler architecture principles and utilizes the MVC-Pattern to define the flow of information. The key component is the parser, which, through iterations and interpretations of partial results from the USE OCL parser, can recognize partial OCL expressions and support them with autocomplete functionality.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# 1 Einleitung

## 1.1 Motivation

Eine Art der Softwareentwicklung ist das Model-Driven-Development (MDD). Die hierbei verwendeten Modelle werden häufig mit UML beschrieben. Die Syntax der UML ist aber an einigen Stellen nicht eindeutig, wodurch Mehrdeutigkeiten entstehen können, wenn verschiedene Menschen mit den gleichen Modellen arbeiten. Da diese Modelle sowohl als erste Skizze als auch zur Dokumentation eines vollständigen Systems verwendet werden, ist es besonders wichtig, dass diese Modelle in sich schlüssig sind. Um die Modelle eindeutiger zu beschreiben, kann die OCL, eine Erweiterung der UML, verwendet werden. Hierbei werden Constraints beschrieben, die über das gesamte Modell hinweg gelten müssen, wodurch Mehrdeutigkeiten schnell gefunden werden können. Da beim MDD die Modelle über den gesamten Entwicklungsprozess hinweg verwendet werden, ist eine effiziente Nutzung des Modellierungswerkzeugs essenziell, um effizient Software zu entwickeln. Eine Autovervollständigung unterstützt diese, da weniger konkretes Wissen über verwendete Teile des Modells benötigt wird und gesuchte Operationen schneller gefunden und eingegeben werden können.

## 1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Architektur für eine Autovervollständigung für (partielle) OCL-Ausdrücke zu erarbeiten. Dabei liegt der Fokus dieser Architektur insbesondere auf leichter Erweiterbarkeit und leichter Wartbarkeit. Anschließend wird diese in das bereits bestehende Modellierungswerkzeug USE integriert.

## 1.3 Aufbau der Arbeit

Zu Beginn der Arbeit wird in Kapitel 2 der wissenschaftliche Hintergrund zu den Themen Modellierung (Abschnitt 2.1), UML (Abschnitt 2.2), OCL (Abschnitt 2.3), USE (Abschnitt 2.4) und Autovervollständigung (Abschnitt 2.5) dargestellt. Am Ende des Kapitels werden zu dieser Arbeit verwandte Arbeiten vorgestellt.

In Kapitel 3 wird die Anforderungsanalyse der Autovervollständigung durchgeführt. Das Ergebnis dieser ist eine Tabelle an Funktionen, die umgesetzt werden müssen (vgl. Tabelle 3.4). Anschließend wird die Architektur der Autovervollständigung entworfen (Abschnitt 3.1.2) und die Funktionsweise der einzelnen Komponenten erläutert (Abschnitte 3.1.3, 3.1.4, 3.1.5 und 3.1.6). In Abschnitt 3.1.7 wird erläutert wie die korrekte Funktionsweise der Autovervollständigung überprüft wird. Zum Abschluss dieses Kapitels wird an einem Beispiel erläutert, was getan werden muss, damit eine neue Operation von der Autovervollständigung unterstützt wird (Abschnitt 3.2).

In Kapitel 4 wird ein Fazit über die Durchführung dieser Arbeit gezogen. Anschließend werden in Abschnitt 4.2 Themen für mögliche weiterführende Arbeiten dargestellt.

# 2 Hintergrund

In diesem Kapitel wird zuerst der wissenschaftliche Hintergrund bezüglich Modellierung in der Softwareentwicklung dargestellt. Hierbei werden UML, OCL und USE erläutert. Anschließend wird der wissenschaftliche Hintergrund zum Thema Autovervollständigung beschrieben. Abschließend werden zu dieser Arbeit verwandte Arbeiten vorgestellt.

## 2.1 Relevanz der Modellierung von Softwaresystemen

Bei der Entwicklung eines Softwaresystems müssen (fast) immer Informationen ausgetauscht werden. Dies liegt unter anderem daran, dass Software insbesondere bei größeren Systemen, selten von nur einer Person alleine entwickelt wird. Sowohl mit anderen Entwicklern als auch mit anderen Stakeholdern müssen Informationen über das System ausgetauscht werden, um eine Integrität des Systems zu ermöglichen. Eine gängige Methode hierfür sind beispielsweise Modelle des zu entwickelnden Systems. Diesen Modellen wird insbesondere beim MDD eine zentrale Rolle bei der Entwicklung zugewiesen, da sie nicht nur zum Informationsaustausch, sondern auch zur Spezifikation genutzt werden. Des Weiteren können Modelle genutzt werden, um das System über die anfängliche Entwicklungsphase hinaus zu dokumentieren. Diese Dokumentation ist für alle am Entwicklungsprozess beteiligten Personen sowie für zukünftige Nutzer des Systems von Bedeutung, um das System verwenden, erweitern und warten zu können.

## 2.2 Unified Modeling Language

Zum Teilen von Informationen über Systeme eignen sich Modelle dieser Systeme. Damit mit diesen Informationen ausgetauscht werden können, bedarf es eine einheitliche Sprache. Diese muss das Definieren und Lesen von Modellen ermöglichen. Die prominenteste Modellierungssprache ist die UML. Im Folgenden wird zunächst darauf eingegangen, was

allgemein unter UML verstanden wird. Dies ist der Teil der UML, der ehemals unter der UML-Superstructure bekannt war [19]. Diese umfasst die typischen UML-Diagramme (Klassen-, Komponenten-, Aktivitäts- etc. ... Diagramme).

### 2.2.1 Hintergrund der Unified Modeling Language

Die UML ist die wohl verbreitetste Modellierungssprache. Sie wird von der Object Management Group (OMG) entwickelt und ist ebenfalls durch die ISO standardisiert [6]. Sie ist eine Notation, um Softwaresysteme zu modellieren, dokumentieren, spezifizieren und visualisieren [17]. Die UML besteht aus Strukturdiagrammen (Klassendiagramme, Objektdiagramme, Komponentendiagramme, etc. ...) und aus Verhaltensdiagrammen (Use-Case-Diagramme, Aktivitätsdiagramme, Sequenzdiagramme etc. ...) [17]. Erstere werden dazu verwendet, Komponenten des Systems zu modellieren, bei denen die Struktur der Elemente gleich bleibt und sich nur die Daten verändern. Verhaltensdiagramme visualisieren Veränderungen einzelner Aspekte des Systems.

### 2.2.2 Probleme der UML

Trotz der stetigen Updates [15], welche die UML verbessern, ist die UML noch nicht perfekt. Ein Problem ist, dass die Syntax in einigen Details nicht eindeutig ist. Ein weiteres Problem ist, dass oft eine vereinfachte (oder veraltete) Version der UML verwendet wird. Trotzdem verwendeten 60% der 32 in [9] untersuchten Arbeiten eine vereinfachte Version der UML. Durch die Verwendung dieser nicht standardisierten Versionen verliert die UML an Verwendbarkeit, da Probleme, die in der neuesten UML-Version bereits gelöst sind, weiterhin vorhanden sind, oder Modelle aufgrund der vereinfachten Modellierung an Aussagekraft verlieren.

Neben diesen syntaktischen Problemen hat die UML aber teilweise auch Probleme mit der Beschreibung von Systemen. Dies soll in folgendem Beispiel veranschaulicht werden:

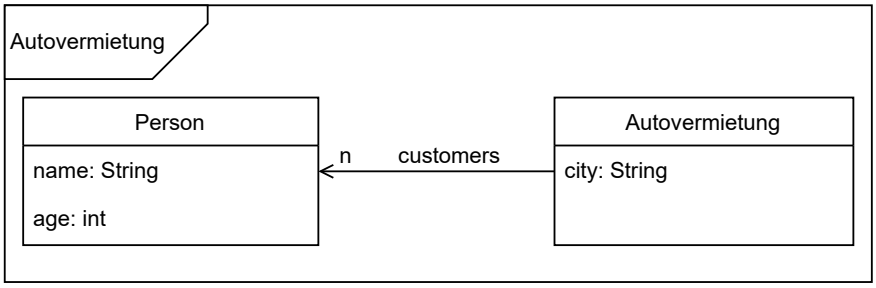


Abbildung 2.1: Ein einfaches Klassendiagramm einer Autovermietung

Die UML kann hier den Wertebereich des Alters nicht einschränken. Dies führt dazu, dass nach diesem Diagramm ein Minderjähriger Kunde bei der Autovermietung sein könnte. Ein noch allgemeineres Problem, welches sich auf (fast) jedes System anwenden lässt, in dem eine Komponente ein Alter als Attribut hat, sind negative Werte.

## 2.3 Object-Constraint-Language

### 2.3.1 Was ist OCL

Wie in 2.2 bereits erwähnt wurde, ist die OCL inzwischen Teil der UML. Mit ihr können Regeln definiert werden, die innerhalb des beschriebenen Modells gelten müssen. Diese Regeln heißen OCL-Constraints. Zu Beginn eines OCL-Constraints kann der Kontext durch eine Kontext-Deklaration angegeben werden [14]. In dieser kann auch angegeben werden, ob es sich um eine Invariante handelt. Invarianten müssen zu jeder Zeit für alle Instanzen eines Typen wahr sein [14]. Dies bedeutet, dass durch eine Invariante das in Abschnitt 2.2.2 beschriebene Problem gelöst werden kann (siehe Abbildung 2.2).

```
context Autovermietung inv CustomersOver18:
self.customers->forAll(c | c.age >= 18)
```

Abbildung 2.2: OCL-Constraint nach welcher Kunden einer Autovermietung mindestens 18 Jahre alt sein müssen.

Modellierungswerkzeuge, die OCL unterstützen, prüfen die Einhaltung der OCL-Constraints, sodass schnell bemerkt werden kann, wenn etwas modelliert wird, was eine OCL-Constraint verletzt [4].

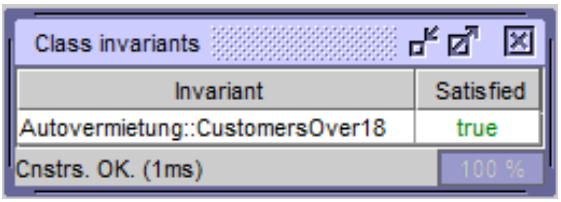


Abbildung 2.3: Klassen-Invarianten in einem Modellierungswerkzeug

### 2.3.2 Wozu kann OCL verwendet werden

Die OCL ermöglicht nicht nur das in Abschnitt 2.2.2 beschriebene Beispiel besser zu beschreiben, sondern kann auch verwendet werden, um zu garantieren, dass ein Modell wie geplant implementiert werden kann. Soll das Modell beispielsweise in Java implementiert werden, so müssen alle Member eines Interfaces public sein und Klassen dürfen nur von maximal einer anderen Klasse erben [5].

Abbildung 2.4 zeigt eine OCL-Constraint die garantiert, dass Interfaces nur public Member besitzen und Abbildung 2.5 zeigt eine OCL-Constraint welche garantiert, dass Klassen nur von maximal einer anderen Klasse erben [1].

```
context Interface
self.feature->forAll(f | f.visibility = #public)
```

Abbildung 2.4: OCL Statement nach welchem alle Member eines Interfaces public sein müssen.

```
context Classifier inv SingleInheritance:
self.generalization->size()<=1
```

Abbildung 2.5: OCL Statement nach welchem Klassen nur von maximal einer Klasse erben können

## 2.4 UML-based Specification Environment

USE ist ein Modellierungswerkzeug, welches, neben den in Abschnitt 2.2.1 erwähnten gängigen UML-Diagrammen, auch OCL unterstützt. Zugang zu OCL wird dem User in dem GUI durch die Auflistung der Klasseninvarianten (siehe Abbildung 2.3) ermöglicht. Des Weiteren können OCL-Ausdrücke auch direkt ausgewertet werden. Dafür können der

`Evaluate OCL expression`-Dialog (siehe Abbildung 2.6) und die Konsole verwendet werden.

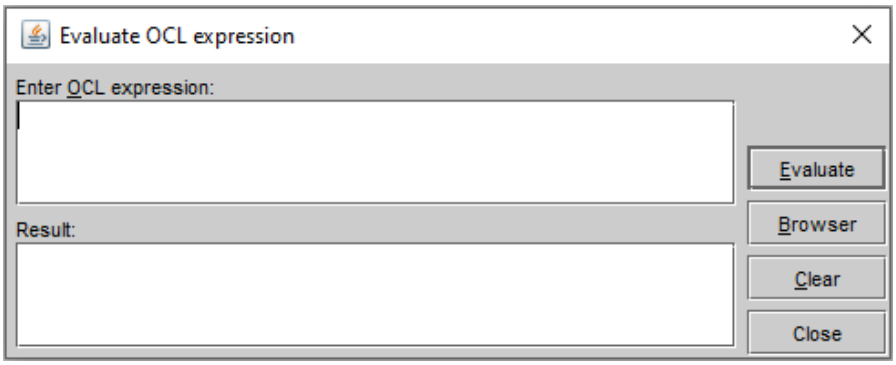


Abbildung 2.6: Der `Evaluate OCL expression`-Dialog in USE

Dies ist der Dialog, in dem in Abschnitt 3.1 Autovervollständigung für OCL eingebunden werden soll.

## 2.5 Autovervollständigung

### 2.5.1 Was ist Autovervollständigung

Autovervollständigung ist der Vorschlag von möglichen Fortsetzungen eines Inputstrings in einer Vorschlagsliste. Im Folgenden werden neben möglichen Features einer Autovervollständigung auch eine grundlegende Architektur sowie mögliche verschiedene Varianten der Komponenten dieser Architektur erarbeitet und vorgestellt. Da es in dieser Arbeit um Autovervollständigung für partielle OCL-Ausdrücke geht (siehe Abschnitt 1.3), wird im Folgenden auch nur dieser Anwendungsfall genauer betrachtet. Dies bedeutet, dass die im in dieser Arbeit betrachtete Autovervollständigung eine Syntax beachten muss (anders als bei Suchvorschlägen in einem Browser). Dadurch, dass OCL statisch typisiert ist, können die Vorschläge direkt auf den Typen basieren. Dies ermöglicht eine simplerere Architektur als bei Autovervollständigung bei dynamisch typisierten Sprachen (vergleiche [12]).

### 2.5.2 Möglichkeiten von Autovervollständigung

#### Was kann vorgeschlagen werden

Die Vorschläge in typisierten Sprachen basieren auf dem Typ des Inputstrings. Das bedeutet zum Beispiel, dass bei einem Objekt im Inputstring die Member dieses Objekts vorgeschlagen werden. Außerdem können Teilausdrücke eines Ausdrucks bereits vor der eigentlichen Auswertung ausgewertet werden. Dies kann beispielsweise dazu verwendet werden, dass bei einem Vergleichsausdruck im Inputstring auf der rechten Seite des Ausdrucks priorisiert Typen vorgeschlagen werden, die zum Typen auf der linken Seite des Ausdrucks passen.

#### Was kann mit den Vorschlägen gemacht werden

Zusätzlich zu den bloßen Vorschlägen von Fortsetzungen des Inputstrings kann auch eine Quickinfo zu den Vorschlägen gegeben werden. Dies wäre beispielsweise bei Membern eines Objektes der Typ oder eine ggf. vorhandene Dokumentation. Des Weiteren kann ein zwischen Inputstring und Vorschlag übereinstimmendes Präfix hervorgehoben (siehe Abschnitt 3.1.4) oder die Liste nach bestimmten Kriterien sortiert werden (siehe Abschnitt 2.5.6).

### 2.5.3 Architektur

Damit Vorschläge abhängig vom Inputstring gemacht werden können, muss dieser lexikalisch, syntaktisch und semantisch analysiert werden. Da dies ebenfalls die Schritte bei der Textanalyse eines Compilers sind [18] kann ein Teil der Architektur eines Compilers als Grundlage der Architektur einer Autovervollständigung verwendet werden (vgl. Abbildung 2.7 [8]). Die Analyse wird dabei von den grün markierten Komponenten durchgeführt. Diese Komponenten lassen sich auch in Abbildung 2.8 wiederfinden.

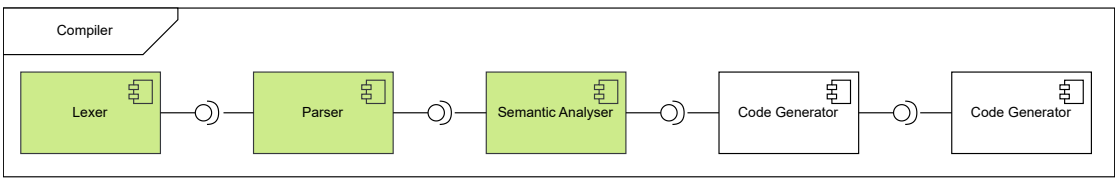


Abbildung 2.7: Architektur eines Compilers

Da bei Autovervollständigung eine Liste an Vorschlägen anstelle von einem Sourcecode generiert werden soll, tritt anstelle der Komponente `Code Generator` die Komponente `Suggester`.

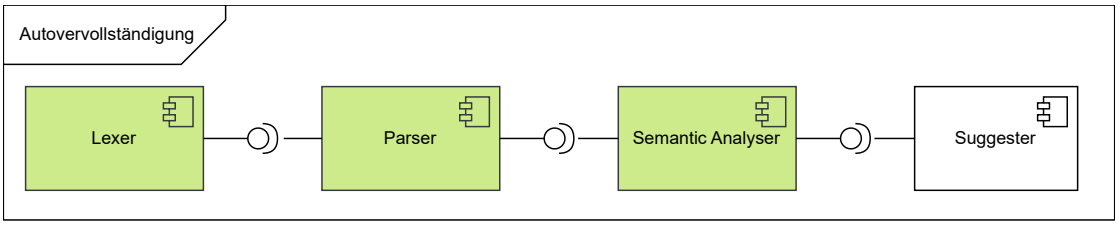


Abbildung 2.8: Architektur einer Autovervollständigung

Der Lexer hat die Aufgabe, den eingegebenen Inputstring in Tokens aufzuteilen [8]. Die Tokens werden anschließend an den Parser gegeben, der aus diesen einen Abstract Syntax Tree (AST) erstellt. Hier bei wird ebenfalls die Syntax des Inputstrings überprüft [8]. Der Semantic Analyser erhält den AST und prüft die Semantik des Inputstrings. Anschließend stellt der Suggester aus den erhaltenen Informationen eine Liste von Vorschlägen zusammen.

### 2.5.4 Analyse

Der Begriff des Analysers fasst im folgenden die Tätigkeiten der Komponenten `Lexer`, `Parser` und `Semantic Analyser` zusammen. Es bestehen zwei Möglichkeiten, welcher Analyser von der Autovervollständigung genutzt werden kann. Die eine ist, dass die Autovervollständigung einen eigenen, extra für diese erstellten Analyser verwendet. Die andere verwendet den Analyser der Sprache, mit welcher die Autovervollständigung verbunden ist.

#### Zusätzlicher Analyser

Die Laufzeit eines zusätzlichen Analysers kann besser sein als die des bereits vorhandenen. Dies liegt daran, dass der zusätzliche Analyser für seinen tatsächlichen Zweck verwendet wird. Ein Nachteil bei der Verwendung eines zusätzlichen Analysers ist, dass dadurch zwei Analyser existieren, die beide auf der gleichen Sprache arbeiten. Wird die Sprache nun verändert, so müssen zwei Analyser parallel angepasst werden. Die Verwendung eines zusätzlichen Analysers bedeutet also zusätzlichen Wartungsaufwand.

#### Vorhandener Analyser der Sprache

Wie zuvor in Abschnitt 2.5.4 erwähnt, wird die Laufzeit im Regelfall bei der Verwendung des bereits vorhandenen Analysers der Sprache schlechter sein als bei einem nur für Autovervollständigung vorgesehenen Analysers. Dies liegt daran, dass der Analyser nicht so verwendet wird, wie er ursprünglich verwendet werden sollte. Dies kann dazu führen, dass der Analyser für eine Analyse des Inputstrings mehrmals ausgeführt werden muss (siehe auch Abschnitt 3.1.3). Ein Vorteil bei der Verwendung des bereits vorhandenen Analysers ist neben des in Abschnitt 2.5.4 erwähnten besseren Laufzeitverhaltens auch, dass die Auswertung von Teilausdrücken leichter ist. Dies liegt daran, dass der bereits vorhandene Analyser als Ziel die Auswertung der Ausdrücke hat. Somit wird dieser bei der Auswertung von Teilausdrücken wieder so verwendet, wie er es ursprünglich sollte. Der zusätzliche Analyser hätte hier die gleichen Probleme, die der bereits vorhandene Parser bei dem Parsen für die Autovervollständigung hätte.

### 2.5.5 Datenhaltung des Suggesters

Damit der Suggester passende Vorschläge, die über primitive Datentypen hinaus gehen, geben kann, benötigt er Daten über den aktuellen Kontext. Hier gibt es ebenfalls zwei Möglichkeiten. Zum einen können die Daten redundant gespeichert werden. Zum anderen können, wenn diese vorhanden sind, Datenstrukturen verwendet werden, die bereits intern von dem Werkzeug verwendet werden, in dem die Autovervollständigung stattfindet.

#### Redundante Datenhaltung

Redundante Datenhaltung ermöglicht konstante Antwortzeiten des Suggesters, da die Daten so aufbereitet werden können, dass diese nach Analyse des Analysers lediglich abgefragt werden müssen. Wenn allerdings Daten im Werkzeug verändert werden, führt die redundante Datenhaltung zu einer längeren Antwortzeit, da mehr Datensätze aktualisiert werden müssen.

#### Keine zusätzliche Datenhaltung

Die Verwendung der bereits im Werkzeug gespeicherten Daten führt zu inkonstanten, längeren Antwortzeiten der Autovervollständigung. Dies liegt daran, dass die vorhandenen

Daten bei jeder Anfrage erst durchsucht und aufbereitet werden müssen. Die Laufzeitkomplexität kann je nach Begebenheit der Organisation der Daten und den benötigten Informationen zwischen konstant (O(1)) und faktoriell (O(n!)) liegen. Ein Vorteil der Verwendung der bereits vorhanden Daten ist, dass die Laufzeit bei Veränderung der Daten unverändert ist.

### 2.5.6 Sortierung der Vorschlagsliste

Bei den viel verwendeten Implementierungen von Autovervollständigungen in Visual Studio und IntelliJ wird das Sortieren der Vorschläge durch KI durchgeführt [11] [7]. Diese sorgt dafür, dass Vorschläge, deren Wahrscheinlichkeit ausgewählt zu werden, am höchsten ist, ganz oben in der Liste angezeigt werden. Dadurch wird die Autovervollständigung effizienter, da der User weniger Zeit mit Suchen bzw. Scrollen verbringen muss. Die Sortierungen basieren dabei auf open-source Projekten auf GitHub [11]. Außerdem passen die neuronalen Netze die Sortierung der Vorschläge an Aktionen des Users an [7]. Das Sortieren der Vorschläge wird aber aufgrund des Umfangs des Themas in dieser Arbeit lediglich angerissen und soll im weiteren Verlauf kein Bestandteil mehr sein.

## 2.6 Verwandte Arbeiten

#### OCL2PSQL: An OCL-to-SQL Code-Generator for Model-Driven Engineering

In [13] wird eine Übersetzungseinheit zwischen OCL und Structured Query Language (SQL) (konkreter PostgreSQL) entworfen, implementiert und Benchmarkvergleiche mit ähnlichen Arbeiten durchgeführt. Andere Arbeiten, in denen ebenfalls von OCL zu SQL übersetzt wird, sind beispielsweise die Arbeiten [2] und [16]. Arbeit [13] hat diesen Arbeiten voraus, dass die Übersetzung effizienter abläuft bzw., dass auch (geschachtelte) Iteratoren übersetzt werden können [13]. Die Übersetzungseinheit von [13] erstellt für jede Klasse und jede Beziehung zwischen den Klassen eines Modells eine SQL-Tabelle. Diesen werden anschließend die Instanzen hinzugefügt. OCL-Ausdrücke werden dann mithilfe von den in dieser Arbeit beschriebenen Regeln in SQL-Queries übersetzt und ausgeführt. Dies bedeutet, dass die SQL-Datenbank zur Auswertung vollständiger OCL-Ausdrücke verwendet wird.

Da die Autovervollständigung größtenteils partielle OCL-Ausdrücke verwendet, müssten die in [13] vorgestellten Regeln angepasst werden, damit diese Übersetzungseinheit für Autovervollständigung von OCL genutzt werden kann. Des Weiteren wird eine redundante Speicherung des Modells in einer Datenbank in diesem Anwendungsfall als nicht notwendig erachtet, da die Daten in USE bereits so aufbereitet sind, dass diese lediglich abgefragt werden müssen. Aus diesen Gründen wird dieser Ansatz in dieser Arbeit nicht weiter verfolgt.

#### Efficient and Effective Query Auto-Completion

In [3] wird eine Autovervollständigung für Suchen im Internet (konkreter eBay) beschrieben. Die Eingabe wird bei jedem Leerzeichen getrennt und in Tokens aufgeteilt. Anschließend werden diese in einer Datenstruktur gesucht und auf skalare Werte übersetzt. Diese skalaren Werte referenzieren alle Einträge in der Datenstruktur, die im Kontext zu diesem Token stehen, also als Vorschlag relevant sind. Die Vorschläge an sich sind dabei als Trie organisiert, damit der Zugriff möglichst effizient ist (da zusätzlich auch nach Präfixen gesucht wird).

Der Anwendungsfall von [3] unterscheidet sich von dem in dieser Arbeit dadurch, dass sowohl der Inputstring als auch die Vorschläge keine Grammatik befolgen müssen, dies ist bei OCL-Ausdrücken allerdings der Fall. Um dies umzusetzen, müsste der vorgestellte Algorithmus stark angepasst werden, weshalb diese Methode nicht weiter verfolgt wurde.

#### IntelliSense implementation of a dynamic language

In [12] wird eine Autovervollständigung für PHP entwickelt und in Microsoft Visual Studio integriert. Da PHP dynamisch ist, speichert die Autovervollständigung alle deklarierten Variablen einer Datei in einem AST. Von diesem wird eine gecachete Version gespeichert, damit nicht alle Zeilen einer Datei nach einer neuen Eingabe neu geparst werden müssen.

Da OCL statisch, im Regelfall weniger umfangreich als Programmiersprachen ist und seltener neue Instanzen erstellt werden, wird keine zusätzliche Datenstruktur benötigt, um

alle deklarierten Variablen zu verfolgen. Dies bedeutet, dass es für eine Autovervollständigung für OCL unkompliziertere Lösungsmöglichkeiten gibt. Daher wird der in Arbeit [12] beschriebene Ansatz in dieser Arbeit nicht weiter verfolgt.

# 3 Design und Integration

In diesem Kapitel wird zu Beginn die Anforderungsanalyse der Autovervollständigung durchgeführt. Anschließend wird die Architektur der Autovervollständigung entworfen und die Funktionalität der einzelnen Komponenten erläutert. Abschließend wird anhand eines Beispiels gezeigt, welche Änderungen in USE durchgeführt werden müssen, wenn eine neue Operation einer Collection von der Autovervollständigung unterstützt werden soll.

## 3.1 Einbindung von Autovervollständigung in USE

### 3.1.1 Anforderungsanalyse

#### Anforderungen

In den folgenden Tabellen 3.1, 3.2 und 3.3 sind die Anforderungen an eine Autovervollständigung in USE dargestellt. Alle Anforderungen beziehen sich auf die GUI-Darstellung, die Unterstützung der Konsole wird hier nicht weiter beachtet.

| ID | Beschreibung |
|---|---|
| BE-1 | Die Vorschläge werden automatisch vorgeschlagen.<br>Es ist kein Hotkey notwendig. Dies beinhaltet die automatische Aktualisierung bei einer Veränderung des Textfeldes. |
| BE-2 | Die Auswahl eines Vorschlags erfolgt durch die ENTER-Taste. |
| BE-3 | Die Navigation durch die Vorschlagsliste erfolgt durch die Pfeiltasten HOCH und RUNTER. |

Tabelle 3.1: Bedienungsanforderungen an eine Autovervollständigung in USE

Das automatische Vorschlagen vereinfacht die Bedienung, da zum einen kein Wissen über das Vorhandensein einer Autovervollständigung nötig ist, um diese nutzen zu können und zum anderen, da weniger Tastatureingaben benötigt werden.

| ID | Beschreibung |
|---|---|
| DAR-1 | Die Vorschläge werden unter dem Cursor so dargestellt, dass übereinstimmende Präfixe untereinander und Vorschläge hinter dem Cursor stehen. |
| DAR-2 | Übereinstimmende Präfixe werden in blau hervorgehoben. |
| DAR-3 | Parameter werden in orange hervorgehoben. |
| DAR-4 | Gibt es keine Vorschläge, so wird das Fenster zum Darstellen der Vorschläge nicht angezeigt. |
| DAR-5 | Die Vorschläge werden alphabetisch in absteigender Reihenfogle sortiert. |

Tabelle 3.2: Darstellungsanforderungen an eine Autovervollständigung in USE

Die farbliche Hervorhebung von Parametern und übereinstimmenden Präfixen (vgl. DAR-2 und DAR-3) ist wichtig, da diese nach Auswahl eines Vorschlags nicht mit in das Textfeld geschrieben werden. Dies soll an folgendem Beispiel verdeutlicht werden:

Gegeben sei folgendes Objekt:

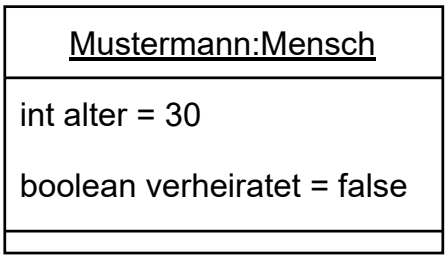


Abbildung 3.1: Objektdiagrammbeispiel zur Erläuterung der Darstellungsanforderungen

Wird nun bei dem Inputstring `Mustermann.alt` der Vorschlag `alter` ausgewählt, so wird nur `er` dem Inpustring hinzugefügt, sodass, nach Auswahl des Vorschlags, der Text in dem Textfeld `Mustermann.alter` entspricht.

| ID | Beschreibung |
|---|---|
| FUNK-1 | Besteht der Inputstring aus einem Objekt mit anschließender Punktnotation (`objekt.`), werden alle Member der (Super-)Klasse(n) des Objektes vorgeschlagen. |
| FUNK-2 | Wird hinter einem Objekt mit anschließender Punktnotation noch ein Präfix angegeben, werden nur die Member der (Super-)Klasse(n) vorgeschlagen, die den gleichen Präfix besitzen. |
| FUNK-3 | Besteht der Inputstring aus einer Collection mit anschließender Pfeilnotation (`Collectiontyp{collectionElements}->`), so werden alle möglichen Operationen dieses Collectiontyps vorgeschlagen. |
| FUNK-4 | Wird hinter einer Collection mit Pfeilnotation noch ein Präfix angegeben, werden nur die Operationen des Collectiontyps vorgeschlagen, die den gleichen Präfix besitzen. |
| FUNK-5 | Bei einem Inputstring der einen Vergleich durchführt mit einem Objekt auf der rechten Seite des = (argument1 = objekt.) besteht die Vorschlagsliste aus zwei Teillisten die jeweils sortiert sind. Zu Beginn stehen die Attribute und Operationen, die dem Typen auf der linken Seite des = entsprechen, anschließend folgen alle anderen Attribute und Operationen. |

Tabelle 3.3: Funktionalitätsanforderungen an eine Autovervollständigung in USE

FUNK-5 wird an folgendem Beispiel genauer erläutert:

Gegeben sei folgendes Objekt:

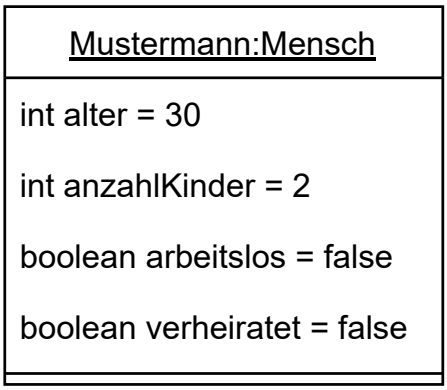


Abbildung 3.2: Objektdiagrammbeispiel zur Erläuterung der funktionalen Anforderungen

Wird nun der Inputstring `True=Mustermann.` eingegeben, so wird folgende Liste vorgeschlagen:

- `arbeitslos`
- `verheiratet`
- `alter`
- `anzahlKinder`

Der erste Teil der Liste besteht aus den Attributen des Typen `Boolean`, da diese dem Typen auf der linken Seite des = entsprechen. Dahinter folgen die anderen Attribute ( `alter` und `anzahlKinder`).

### Funktionen

Aus den in Abschnitt 3.1.1 beschriebenen Anforderungen leiten sich die in Tabelle 3.4 beschriebenen Funktionen ab.

| | |
|---|---|
| Name | getInputstring |
| Parameter | void |
| Beschreibung | Bereitstellung des aktuellen Inputstrings. |
| Rückgabewert | Inputstring |
| Name | parse |
| Parameter | inputstring |
| Beschreibung | Schnittstelle des Parsers der Autovervollständigung. |
| Rückgabewert | Liste aller im Inputstring gefundenen Typen. |
| Name | processFoundTypes |
| Parameter | foundTypes |
| Beschreibung | Klassifizierung des Inputstrings auf Grundlage der gefundenen Typen. |
| Rückgabewert | Inputstringtype |
| Name | getSuggestions |
| Parameter | inputstring, OCLOnly |
| Beschreibung | Schnittstelle der Autovervollständigung. USE-Befehle werden abhängig davon unterstützt, ob OCLOnly gesetzt ist oder nicht. |
| Rückgabewert | Vorschlagsliste zum inputstring |
| Name | displayResults |
| Parameter | suggestions |
| Beschreibung | Darstellung (und Formatierung) der Vorschläge in der GUI. |
| Rückgabewert | void |

Tabelle 3.4: Funktionen der USE-Autovervollständigung als Ergebnis der Anforderungsanalyse

### 3.1.2 Architektur

Auf Grundlage der in Tabelle 3.4 dargestellten Funktionen ergibt sich die Notwendigkeit für drei Komponenten. Diese werden im weiteren Verlauf dieser Arbeit als Suggester, Par-

ser und View bezeichnet. Im Folgenden werden die Funktionen dieser drei Komponenten sowie der Informationsfluss zwischen diesen kurz erläutert.

#### Parser

Der Parser hat die Aufgaben der Funktionen `parse` und `processFoundTypes`. Die Funktionsweise des Parsers sowie die möglichen Klassifizierungen werden in Abschnitt 3.1.3 noch weiter vertieft.

#### Suggester

Der Suggester hat die Aufgabe der Funktion `getSuggestions`. Die Funktionsweise des Suggesters wird in Abschnitt 3.1.5 weiter erläutert.

#### View

Die View hat die Aufgabe der Funktionen `getInputstring` und `displayResults`. Bei der Auswahl der View gibt es zwei Möglichkeiten. Die eine ist die Konsole, die andere die GUI. In Abschnitt 3.1.4 werden die für die Autovervollständigung relevanten Unterschiede in Tabelle 3.5 aufgeführt.

#### Informationsfluss

Die Grundlage des Informationsflusses der Autovervollständigung die in USE integriert wird, bildet das MVC-Pattern. In Abbildung 3.3 ist zu sehen, dass dieses um die Komponente Parser erweitert wurde.

Diese wird benötigt, da ein Vorschlagszyklus immer mit dem Inputstring in der View beginnt und dieser, bevor er vom Suggester verarbeitet wird, an den Parser weitergegeben wird. Nachdem der Parser den Inputstring geparst hat, wertet der Suggester das Ergebnis aus und holt sich vom Model die benötigten Daten. Danach verarbeitet der Controller diese je nach festgestelltem Parserergebnis und gibt eine Liste an Suggestionstrings an die View zurück. Diese stellt die Suggestionstrings dem User zur Verfügung.

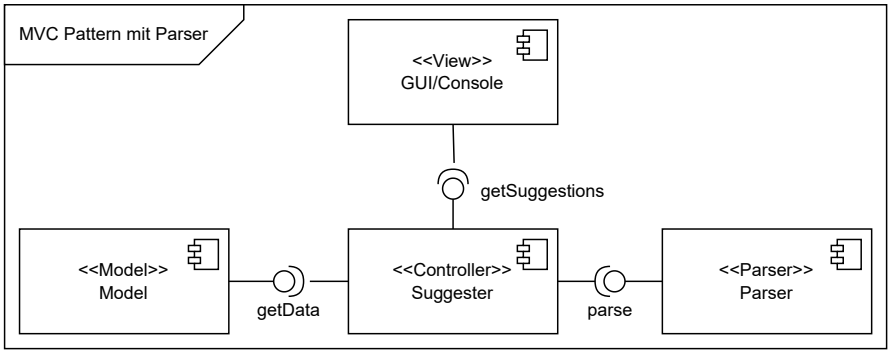


Abbildung 3.3: Model-View-Controller Pattern mit Parser

#### Model

Das Model ist das in USE durch UML und OCL beschriebene Modell. Das Model ist also keine direkte Komponente der Autovervollständigung. Im Model befindet sich die Datenhaltung. Diese bestimmt:

- Welche Collection Operationen existieren
- Welche Klassen existieren
- Welche Objekte existieren

In Abschnitt 3.1.6 wird genauer erläutert, welche Daten gespeichert werden, wie auf diese zugegriffen wird und welche Anpassungen für eine leichtere Integration der Autovervollständigung gemacht worden.

### 3.1.3 Parser

#### Architektur des Parsers

Zum Parsen der (partiellen) OCL-Ausdrücke für die Autovervollständigung wird der in USE bereits vorhandene Parser verwendet und über die USE-Schnittstelle angesprochen, die normalerweise zum Kompilieren von OCL-Ausdrücken dient. Dass die Komponente Parser der Autovervollständigung neben dem Parser auch den Lexer und den Semantic-Analyser umfasst liegt also daran, dass der Parser intern einen Compiler verwendet (vgl. Abbildung 3.4).

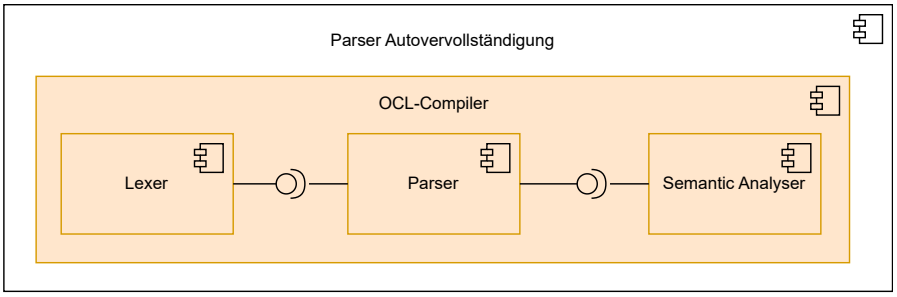


Abbildung 3.4: Architektur der Komponente Parser

Bei den in Abbildung 3.4 in orange dargestellten Komponenten handelt es sich um Komponenten, die Teil von USE und nicht nur Teil der Autovervollständigung sind.

### Funktionsweise des Parsers

Wie in Abschnitt 2.5.4 bereits erwähnt wurde, wird der Parser hierbei zweckentfremdet. Dies liegt daran, dass die Inputstrings bei Autovervollständigung im Regelfall partielle OCL-Ausdrücke sind, der Parser aber vollständige OCL-Ausdrücke erwartet. Dies führt zu der in den Codebeispielen 3.1 und 3.2 dargestellten Verwendung des USE-OCL-Parsers. Auf einen Auszug aus dem tatsächlichen Sourcecode wird an dieser Stelle verzichtet, da die vereinfachte Darstellung durch Pseudocode für den Leser besser nachvollziehbar und dadurch leichter verständlich ist. Der interessierte Leser kann die Auszüge des tatsächlichen Sourcecodes im Anhang (vgl. Abbildungen A.1 und A.2) finden.

Listing 3.1: Pseudocode der Verwendung des USE-OCL-Parsers I

```
def newParser(input):
    types = parseInputString(input)
    process(types)
```

Zu sehen ist der Pseudocode der `newParser`-Funktion der Autovervollständigung. `newParser(input)` sammelt dabei zu erst alle Typen, die sich im Inputstring befinden (vgl. Codebeispiel 3.2) und verarbeitet diese anschließend in der `process`-Funktion.

Listing 3.2: Pseudocode der Verwendung des USE-OCL-Parsers II

```
def parseInputString(input):
    type = parse(input)
    if type is None:
        tokens = input.split()
        for token in tokens:
            parseInputString(token)
    else:
        handleAddingCollectionTypes(type)
        handleAddingPrimitiveTypes(type)
```

Zu Beginn wird in der `parseInputString`-Funktion der USE-OCL-Parser aufgerufen, welcher den OCL-Typen des Arguments zurückgibt. Sollte der Inputstring nicht zu einem Typen kompilieren, so wird vom USE-OCL-Parser `null` zurückgegeben. Ist dies der Fall, so wird der Inputstring gelext (`split`-Aufruf) und anschließend mit jedem Token als Argument erneut die `parseInputString`-Funktion aufgerufen. Kompiliert das Argument der `parseInputString`-Funktion, so wird der Typ über die Funktionen `handleAddingCollectionTypes` und `handleAddingPrimitiveTypes` gespeichert. Gegebenenfalls werden noch weitere Informationen zu diesem Typen gespeichert. Welche weiteren Informationen aus dem Inputstring erkannt und gespeichert werden können, ist in Abbildung 3.5 zu sehen.

In der `process`-Funktion werden die gesammelten Typen ausgewertet und (bei OCL-Ausdrücken) ein Objekt eines der in Abbildung 3.5 beschriebenen Typen erstellt.

#### Welche Klassifizierungen möglich sind

In Abbildung 3.5 sind die möglichen Klassifizierungen des Parsers der Autovervollständigung für (partielle) OCL-Ausdrücke zu sehen.

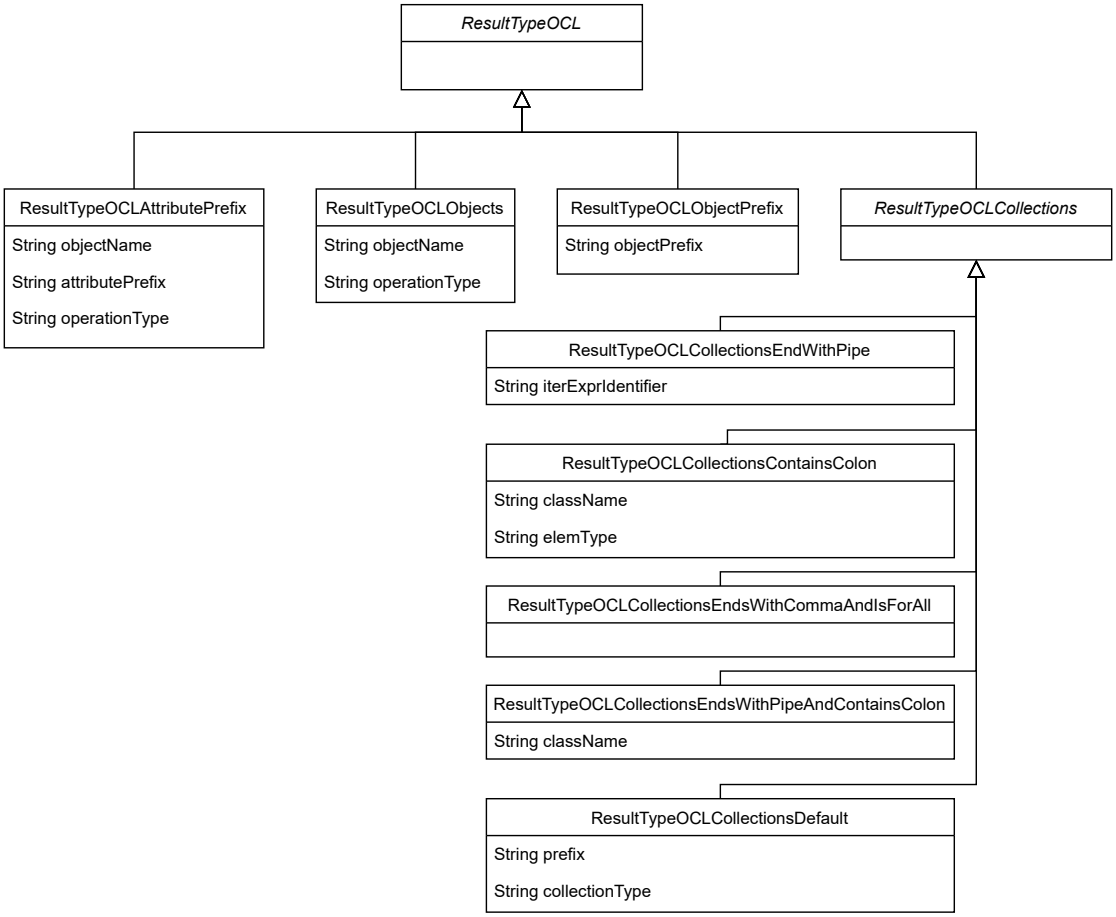


Abbildung 3.5: Klassifizierungen des Parsers der Autovervollständigung für (partielle) OCL-Ausdrücke

Hervorzuheben ist dabei, dass es sich bei dem root-Typ `ResultTypeOCL` sowie bei `ResultTypeOClCollections` um Interfaces handelt, diese also keine tatsächlich möglichen Klassifizierungen des Parser sind. Des Weiteren sind die Felder der Typen zusätzliche Informationen, die aus dem Inputstring gewonnen werden können, welche die Vorschläge des Suggesters beeinflussen. Dies wird in Abschnitt 3.1.5 weiter erläutert.

### Warum diese Art des Parsens gewählt wurde

Zu Beginn wurde ein eigener Parser entwickelt, welcher simple Inputstrings sehr elegant verarbeiten konnte. Bei einigen Edgecases ist er allerdings sehr komplex und an vielen

Stellen hardcoded geworden. Dies führte zu schlechter Wartbarkeit und schlechter Erweiterbarkeit. Daraufhin wurde ein Parser entwickelt, der intern den USE-OCL-Parser verwendet. Dadurch ist in USE nur ein OCL-Parser vorhanden, was die Wartung deutlich vereinfacht. Die Erweiterbarkeit der Autovervollständigung um neue Unterstützungen wird dadurch ebenfalls einfacher. Dies wird in Abschnitt 3.2 gezeigt.

### 3.1.4 View

Zur Darstellung wird Swing verwendet. Das Verfolgen und Bereitstellen der aktuellen Inputstrings erfolgen über ein Objekt der Swingklasse DocumentListener. Dieser verfolgt das Textfeld des `Evaluate OCL expression` Dialogs und ruft bei jedem neuen Zeichen die `getSuggestions`-Funktion des Suggesters auf. Hiervon ausgenommen sind die Pfeiltasten HOCH und RUNTER sowie die ENTER-Taste. Die Eingaben dieser Tasten werden abgefangen und an das Fenster zur Darstellung der Vorschläge weitergeleitet.

Zur Darstellung der Vorschläge wird eine Kombination aus den Swingklassen PopupMenu und JList verwendet. Zur nutzerfreundlicheren Navigation durch die Vorschläge überschreibt die JList das Verhalten der Pfeiltasten HOCH und RUNTER, da standardmäßig eine Navigation über das letzte Element hinaus nicht zur Auswahl des ersten Elementes führt (und entsprechend umgekehrt beim ersten Element). Des Weiteren wird das Verhalten der ENTER-Taste überschrieben. Hier werden die Parameter des ausgewählten Vorschlags entfernt, bevor dieser an den im Textfeld vorhandenen Text angehängt wird.

Wie bereits in Tabelle 3.2 mit den IDs DAR-2 und DAR-3 beschrieben, soll eine farbliche Hervorhebung von Parametern und Präfixen eine leichtere Verwendung ermöglichen. In Abbildung 3.6 ist die farbliche Hervorhebung zu sehen.

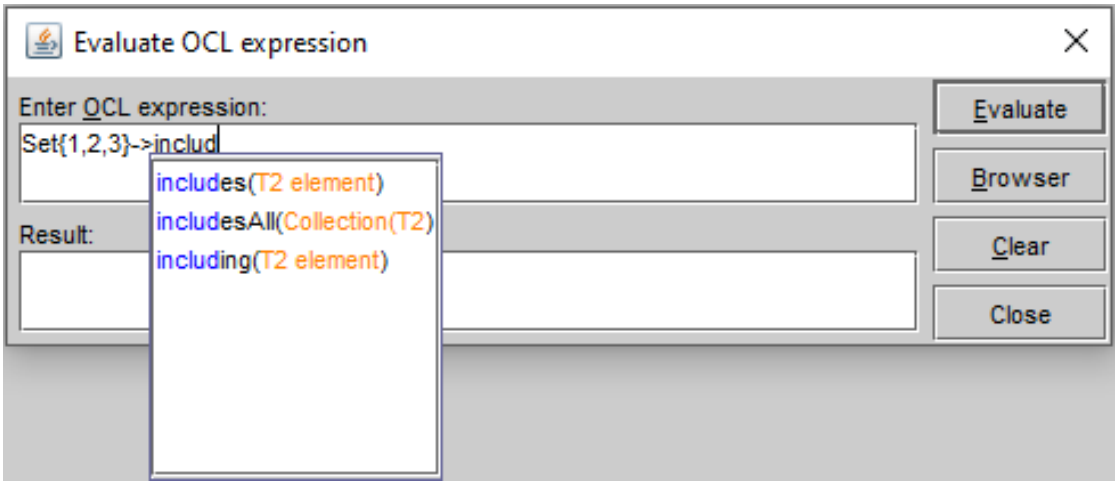


Abbildung 3.6: Darstellung der Vorschläge der Autovervollständigung in der USE-GUI

Da es, wie bereits in Abschnitt 3.1.2 erwähnt, zwei mögliche Views (GUI und Konsole) gibt, sind in Tabelle 3.5 die für die Autovervollständigung relevanten Unterschiede zwischen diesen zwei Möglichkeiten dargestellt.

| | GUI | Konsole |
|---|---|---|
| Darstellung | Anzeigen aller Vorschläge als Liste<br>Wechseln über Pfeiltasten | Anzeigen eines Vorschlages,<br>Wechseln über TAB |
| OCL-Statements | Direkte Eingabe in den `Evaluate OCL expression` Dialog | Beginnen mit `?` oder `??` |
| USE-Befehle | Werden nicht unterstützt | Werden unterstützt |

Tabelle 3.5: Unterschiede der Views `GUI` und `Konsole`

Die Vorschläge des Suggesters sind abhängig von der verwendeten View. Dies ist zwar eine Verletzung des klassischen MVC-Patterns, hier aber notwendig, da wie in Tabelle 3.5 unter dem Punkt `USE-Befehle` zu sehen ist, in der Konsole Vorschläge für USE-Befehle unterstützt werden und in der GUI, konkreter im `Evaluate OCL expression` Dialog, nicht. Dies Unterscheidung wird, wie bereits in Tabelle 3.4 beschrieben, über das Flag OCLOnly gemacht. Ist es gesetzt, so berücksicht der Parser keine USE-Befehle.

### 3.1.5 Suggester

Abhängig davon, welchen der in Abbildung 3.5 dargestellten Typen der Suggester vom Parser erhält, ruft er die entsprechende Funktion auf. In diesen Funktionen werden die benötigten Daten am Model abgefragt und entsprechend der Felder der erhaltenen Klassifizierung gefiltert. So werden beispielsweise nicht alle möglichen Operationen, sondern nur diejenigen mit übereinstimmendem Präfix vorgeschlagen.

### 3.1.6 Model

Das von der Autovervollständigung verwendete Model ist dasselbe Model, das bereits von USE verwendet wird, um existierende Klassen und Objekte zu speichern. Dies bedeutet, dass keine redundante Datenspeicherung vorhanden ist. Die in Abschnitt 2.5.5 beschriebene längere Laufzeit aufgrund schlechter Datenorganisation ist zu vernachlässigen, da die in USE verwendeten Daten zur Autovervollständigung passend organisiert sind.

#### Collectionoperationen

Für die Operationen von Collections gibt es in USE eine Datenstruktur, bei der diese hierarchisch nach dem Schema `collectionoperation.collectiontyp.operation` organisiert sind. Dies ermöglicht eine schnelle Abfrage der möglichen Operationen, da in OCL bei Ausdrücken mit Collections und partiellem Operationsnamen der Collectiontyp immer bekannt ist (vgl. Abbildung 3.7).

```
Set{1,2,3}->includ
```

Abbildung 3.7: OCL-Ausdruck mit Collectiontyp `Set` und partiellem Operationsnamen `includ`

Des Weiteren wurde die root-Klasse zur Beschreibung von Operationen von Collections (OpGeneric) um ein Feld des Typen ParamInfo erweitert. Dieses speichert unter anderem die Parameter der jeweiligen Operation und lässt einen Zugriff auf diese über eine public `toString`-Methode zu (vgl. Abbildung 3.8). Dadurch ist es der Autovervollständigung möglich, bei Operationen von Collections die Parameter mit anzugeben.

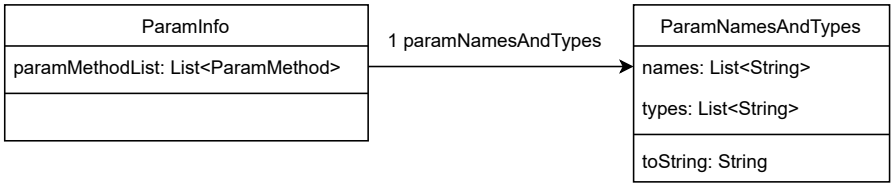


Abbildung 3.8: Klassendiagramm zur `ParamInfo`-Klasse

ParamMethod ist dabei ein funktionales Interface und wird USE-intern verwendet, um zu überprüfen, welche Typen die Parameter bei einem tatsächlichen Aufruf der Operation haben. Für die Autovervollständigung wird bei Instanziierung des jeweiligen ParamInfo Objektes neben den ParamMethod-Methoden auch eine Stringrepräsentation der entsprechenden Typen mit angegeben. Dies wird in Abschnitt 3.2 an einem Beispiel erläutert.

### Objekte und Klassen

Für die im USE-Model existierenden Objekte gibt es die Möglichkeit, abhängig eines Objektnamens Informationen über das Objekt zu erhalten. Von diesem kann anschließend die Klasse abgefragt werden. Die Felder und Operationen der Klasse können über die Methoden `allAttributes` und `allOperations` erhalten werden (vgl. Abbildung 3.9).

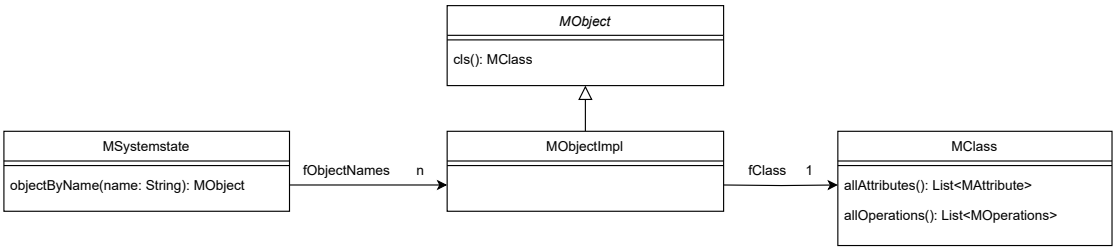


Abbildung 3.9: stark vereinfachtes Klassendiagramm zu den USE internen Klassen zur Speicherung der Klassen und Objekte

Dies bedeutet, dass die Liste der Operationen bzw. der Attribute ein Mal vollständig durchlaufen werden muss, um alle passenden Member zu finden. Somit entsteht eine Laufzeitkomplexität von O(n), die abhängig von der Anzahl der Operationen bzw. der Attribute einer Klasse ist. Dies Laufzeitkomplexität ist so gut, dass zur besseren Wartbarkeit darauf verzichtet wird, eine redundante Datenhaltung hinzuzufügen. Erkennbar

ist die gute Laufzeitkomplexität daran, dass, wie in Abschnitt 2.5.5 beschrieben, eine Laufzeitkomplexität von O(n!) möglich ist.

### 3.1.7 Verifikation

Zur Verifikation werden Unittests verwendet. Dabei gibt es zwei Arten von Tests. Die eine verifiziert das Ergebnis des Parsers. Hier werden sowohl die im Inputstring gefunden Typen und Werte als auch das Gesamtergebnis überprüft. Die andere überprüft die Gesamtfunktionalität. Dies bedeutet, dass ein Inputstring über die `getSuggestions`-Schnittstelle (vgl. Abbildung 3.3) an den Suggester gegeben wird und dieser anschließend unter Verwendung des Models und des Parsers eine Liste an Vorschlägen zurückgibt.

Zum Testen der Funktionalitäten, die in Bezug mit Objekten bzw. Klassen stehen, wird ein Modell verwendet, welches die in in Abbildung 3.10 beschriebenen Klassen verwendet. Dieses Testmodell stellt ein einfaches Beispiel eines möglichen Models dar.

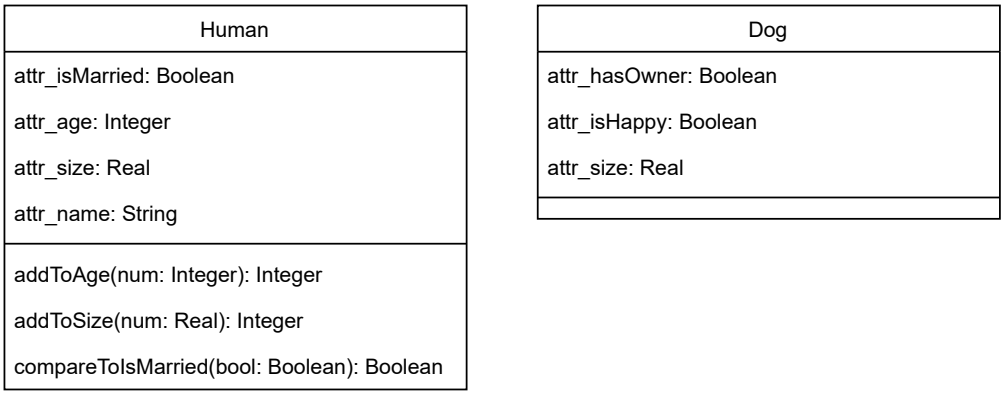


Abbildung 3.10: Klassen eines einfachen Testmodels zur Verifikation

Um zu garantieren, dass die Autovervollständigung vollständig korrekt arbeitet, wird mit zwei verschiedenen Klassen getestet. Die Typisierung der Attribute hat das Ziel, das Verhalten der Autovervollständigung bei Attributen mit verschiedenen Typen zu testen, wobei einige Typen doppelt innerhalb einer Klasse vorkommen, um zu überprüfen, dass die Autovervollständigung alle Attribute berücksichtigt. So gibt es in `Human` zwei numerische Typen, damit überprüft werden kann, ob, wenn ein numerischer Wert erwartet wird, auch beide vorgeschlagen werden. Selbiges gilt auch für die Typisierung der Rückgabewerte der Operationen `addToAge` und `addtoSize`. Die anderen Typen sind

ausgewählt worden, um zu verifizieren, dass die Autovervollständigung alle Funktionalitäten auch mit nicht-numerischen Typen erfüllen kann. Dies ist auch der Grund, warum es in `Dog` zwei Attribute vom Typen `Boolean` gibt.

Zum Testen der Funktionalitäten, die in Bezug zu Collections stehen, wird das gleiche Testmodel verwendet, da USE die Verwendung der Collections und ihrer Operationen bei Initialisierung eines Models ermöglicht und daher kein zusätzliches Model benötigt wird. In diesen Tests werden die Funktionalitäten an allen in USE vorhanden Collections (Bag, Set, OrderedSet, Sequence) getestet.

## 3.2 Beispiel: Hinzufügen einer neuen OCL-Operation

In diesem Abschnitt wird anhand eines Beispiels die Integration der Autovervollständigung in USE veranschaulicht. Dabei soll eine neue Operation für den Collectiontypen `Set` zu USE hinzugefügt und von der Autovervollständigung unterstützt werden.

Diese Beispieloperation ist in Tabelle 3.6 dargestellt.

| | |
|---|---|
| Name | CartesianProduct |
| Parameter | Set, Set |
| Beschreibung | Bildet das kartesisches Produkt der beiden Mengen. |
| Rückgabewert | Menge von Bags |

Tabelle 3.6: Beispieloperation `CartesianProduct`

Innerhalb der direkten Komponenten der Autovervollständig (d.h. Parser, Suggester, Model und View) muss zur Unterstützung der `CartesianProduct`-Operation keine Anpassung vorgenommen werden. In Tabelle 3.7 ist dargestellt, warum innerhalb der einzelnen Komponenten keine Anpassung erfolgen muss, um die `CartesianProduct`-Operation durch Autovervollständigung zu unterstützen.

| Komponente | Erläuterung |
|---|---|
| Parser | Da zum (Lexen und) Parsen der USE-Parser verwendet wird, ist keine weitere Anpassung notwendig. |
| Suggester und Model | Erhält der Suggester eine `ResultTypeOClCollectionDefault`-Klassifizierung, wird eine Methode aufgerufen, die abhängig von dem festgestellten Collectiontypen alle Operationen (mit übereinstimmendem Präfix) dieses Collectiontypen sammelt und vorschlägt. Da hier lediglich das von USE verwendete Model benutzt wird, ist keine weitere Anpassung notwendig. |
| View | In der View könnte lediglich die farbliche Hervorhebung nicht mehr funktionieren. Da diese allerdings für Parameter alles zwischen `(` und `)` hervorhebt und bei übereinstimmenden Präfixen sich nur an `->` bzw. `.` orientiert, ist hier keine weitere Anpassung notwendig. |

Tabelle 3.7: Erläuterungen, warum innerhalb der einzelnen Komponenten der Autovervollständigung keine Anpassung erfolgen muss

Dies bedeutet, damit die Autovervollständigung die `CartesianProduct`-Operation unterstützen kann, muss diese lediglich zu USE hinzugefügt werden. Dies funktioniert (fast) so wie vor der Einbindung der Autovervollständigung in USE. Die Schritte sind in Auflistung 3.2 dargestellt.

- In `StandardOperationsSet.java` eine neue Subklasse `CartesianProduct` von OpGeneric erstellen.
- Folgende Funktionen implementieren damit USE die `CartesianProduct`-Operation verwenden kann:
  - `matches`
  - `name`
  - `kind`

  - `isInfixOrPrefix`
  - `eval`

- Als Veränderung zu vorher muss des Weiteren noch ein Konstruktor implementiert werden, in welchem die `setParamInfo`-Methode (welche von OpGeneric vererbt wird) aufgerufen wird. Hier muss zum einen eine Liste von Funktionsreferenzen (Type::isTypeOfSet, Type::isTypeOfSet) übergeben werden (welche sowohl von USE als auch von der Autovervollständigung zum Überprüfen der Parametertypen verwendet wird) und zum anderen je eine Liste mit Stringrepräsentationen der Parametertypen(`Set(T1)`, `Set(T2)`) und Parameternamen(`self`, `other`). Diese werden lediglich zur Darstellung der Operationen in der Vorschlagsliste in der View verwendet.
- Die Operation muss in der `registerTypeOperations`-Methode der `StandardOperationsSet`-Klasse zur `opmap` hinzugefügt werden.
- In `SetValue.java` muss das Verhalten der Operation definiert werden.

Dies bedeutet, dass sobald die Operation zu USE hinzugefügt wird, die Autovervollständigung diese sofort unterstützt.

# 4 Fazit und Ausblick

In diesem Kapitel wird zu Beginn ein Fazit über die Methodiken und den Ablauf dieser Arbeit gezogen. Anschließend werden Themen für nachfolgende, auf dieser Arbeit aufbauende Arbeiten dargestellt.

## 4.1 Fazit

Wie in Abschnitt 3.1.3 erwähnt, wurde zu Beginn ein eigener Parser entwickelt, welcher aber anschließend verworfen wurde, da er nur einfache Inputstrings verarbeiten konnte. Wäre vor der Entwicklung des eigenen Parsers der in USE vorhandene Parser besser analysiert worden, so hätte früher festgestellt werden können, dass dieser, durch leichte Anpassungen, für das Parsen partieller OCL-Ausdrücke verwendet werden kann. Dies hätte viel zeitlichen Aufwand verhindert.

Des Weiteren könnten die Tests zur Verifikation der Autovervollständigung sauberer getrennt werden. Während es für den Parser Tests gibt, die diesen einzeln prüfen, gibt es keine, die die Funktionsweise des Suggesters einzeln prüfen. Die einzigen Tests, die die Funktionsweise des Suggesters verifzieren, sind die Integrationstests, die die Zusammenarbeit der beiden Komponenten überprüfen. Einzelne Tests für die Suggester Komponente sollten keine bestehenden Tests ersetzen, sondern zusätzlich zu den vorhandenen Tests hinzugefügt werden. Durch die Gestaltung der Schnittstellen (vgl. Abbildung 3.3) ist es mit dem aktuellen Design nicht möglich, die Suggester Komponente zu verwenden ohne, dass diese den Parser verwendet. Um das einzelne Testen des Suggesters zu ermöglichen, müsste ein Adapter erstellt werden. Dieser könnte eine fertige Parserklassifizierung erhalten und anschließend die entsprechende Methode des Suggesters aufrufen. Durch Verwendung des Adapters müsste bei den Tests nicht mit der Parser Komponente interagiert werden.

Auch wenn die verwendete Architektur und die Schnittstellen der einzelnen Komponenten aktuell kein einzelnes Testen des Suggesters zulassen, so ist die verwendete Architektur (vgl. 3.3) dennoch gut geeignet, um die Aufgaben einer Autovervollständigung zu erfüllen.

## 4.2 Ausblick

### 4.2.1 Teil-Auswertung von OCL-Ausdrücken

Die Idee ist, dass bei (partiellen) OCL-Ausdrücken Teilausdrücke bereits ausgewertet werden, bevor der Nutzer einen vollständigen OCL-Ausdruck geschrieben und auf `evaluate` gedrückt hat. Dies bedeutet beispielsweise, dass bei dem Audruck `Set{1,2,3}->select(e | e > 1)->select(e | e >` der erste select-Aufruf bereits ausgewertet und das Ergebnis (in diesem Fall Set{2,3}) dem Nutzer (beispielsweise) in einem Fenster unterhalb des select-Aufrufs dargestellt wird. Dies kann insbesondere bei komplexen OCL-Ausdrücken die Nutzererfahrung verbessern.

Die Umsetzung der Teilauswertung erfordert eine Anpassung der aktuellen Variante, wie der USE-OCL-Parser verwendet wird. Dies liegt daran, dass, wenn es sich um einen partiellen OCL-Ausdruck handelt, noch nach Teilausdrücken gesucht werden muss, die bereits auswertbar sind.

### 4.2.2 Bessere Suchfunktionen

Ein Beispiel für bessere Suchfunktionen wäre die Suche nach Akronymen. Dies bedeutet beispielsweise, dass wenn im Modell ein Objekt mit dem Namen `MeinObjekt` existiert, `MeinObjekt` nicht nur bei übereinstimmendem Präfix, sondern auch bei übereinstimmenden Akronymen (in diesem Fall MO) vorgeschlagen wird. Da die Suche nach Akronymen in einem Trie gut umsetzbar ist, würde es sich anbieten, die vorhanden Objekte (und ggf. Operationen, Funktionen etc. ...) bei Initialisierung (des Modells bzw. der einzelnen Komponenten) in einem Trie zu organisieren. Hierfür könnten entweder die von USE verwendeten Datenstrukturen angepasst oder neue geschaffen werden.

### 4.2.3 Sortierung nach Nutzungswahrscheinlichkeit

Momentan werden die Vorschläge alphabetisch sortiert. Dies ist eine naheliegende, einfache Variante, die Vorschläge zu organisieren. Besser wäre es, diese auf Grundlage der Nutzungswahrscheinlichkeit zu organisieren. Hierzu könnten allgemeine oder/und nutzerspezifische Daten verwendet werden. Ein Datensatz, um die Nutzungswahrscheinlichkeiten zuzuordnen, könnte auf GitHub-Repositories basieren, so wie es beispielsweise in [10] gemacht wurde. Die Integration in die aktuell vorhandene Autovervollständigung ist einfach, da bereits abhängig von der Parserklassifizierung der jeweilige Sortieralgorithmus aufgerufen wird (nur, dass es sich aktuell immer um denselben Aufruf handelt). Dies ermöglicht verschiedene Sortieralgorithmen, falls bestimmte Teile nicht aufgrund der Nutzungswahrscheinlichkeit sortiert werden sollen.

# Literaturverzeichnis


[1] Demuth, Birgit ; Wilke, Claas: Model and object verification by using Dresden OCL. In: *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*, 2009, S. 687–690

[2] Egea, Marina ; Dania, Carolina: SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. In: *Software & Systems Modeling* 18 (2017), Nr. 1, S. 769–791. – URL https://doi.org/10.1007/s10270-017-0597-6. – ISSN 1619-1374

[3] Gog, Simon ; Pibiri, Giulio E. ; Venturini, Rossano: Efficient and Effective Query Auto-Completion. In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA : Association for Computing Machinery, 2020 (SIGIR '20), S. 2271–2280. – URL https://doi.org/10.1145/3397271.3401432. – ISBN 9781450380164

[4] Gogolla, Martin ; Büttner, Fabian ; Richters, Mark: USE: A UML-based specification environment for validating UML and OCL. In: *Science of Computer Programming* 69 (2007), Nr. 1, S. 27–34. – URL https://www.sciencedirect.com/science/article/pii/S0167642307001608. – Special issue on Experimental Software and Toolkits. – ISSN 0167-6423

[5] Gosling, James ; Joy, Bill ; Steele, Guy L. ; Bracha, Gilad ; Buckley, Alex ; Smith, Daniel ; Bierman, Gavin: *The Java® Language SpecificationJava SE 21 Edition*. 2023. – URL https://docs.oracle.com/javase/specs/. – Zugriffsdatum: 2023-12-12

[6] International Organization for Standardization: *Information technology – Object Management Group Unified Modeling Language (OMG UML)*. ISO 19505:1:2012. Vernier, Geneva, Switzerland : International Organization for Standardization, 2012. – URL https://www.iso.org/standard/32624.html

[7] JetBrains: *IntelliJ IDEA: Code completion*. 2023. – URL https://www.jetbrains.com/help/idea/auto-completing-code.html#ml_completion. – Zugriffsdatum: 2023-12-21

[8] Jordan, Willy ; Bejo, Agus ; Persada, Anugerah G.: The Development of Lexer and Parser as Parts of Compiler for GAMA32 Processor's Instruction-set using Python. In: *2019 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, Dec 2019, S. 450–455

[9] Lucas, Francisco J. ; Molina, Fernando ; Toval, Ambrosio: A systematic review of UML model consistency management. In: *Information and Software Technology* 51 (2009), Nr. 12, S. 1631–1645. – URL https://www.sciencedirect.com/science/article/pii/S0950584909000433. – Quality of UML Models. – ISSN 0950-5849

[10] Mengerink, Josh G. M. ; Noten, Jeroen ; Serebrenik, Alexander: Empowering OCL research: a large-scale corpus of open-source data from GitHub. In: *Empirical Software Engineering* 24 (2019), Nr. 3, S. 1574–1609. – URL https://doi.org/10.1007/s10664-018-9641-6. – ISSN 1573-7616

[11] Microsoft: *IntelliCode: AI-assisted code development in Visual Studio*. 2023. – URL https://learn.microsoft.com/en-us/visualstudio/ide/intellicode-visual-studio?view=vs-2022. – Zugriffsdatum: 2023-12-21

[12] Míšek, Jakub: *IntelliSense implementation of a dynamic language*. 2017

[13] Nguyen Phuoc Bao, Hoang ; Clavel, Manuel: OCL2PSQL: An OCL-to-SQL Code-Generator for Model-Driven Engineering. In: Dang, Tran K. (Hrsg.) ; Küng, Josef (Hrsg.) ; Takizawa, Makoto (Hrsg.) ; Bui, Son H. (Hrsg.): *Future Data and Security Engineering*. Cham : Springer International Publishing, 2019, S. 185–203. – ISBN 978-3-030-35653-8

[14] Object Management Group: *Unified Modeling Language: Object Constraint Language Version 2.4*. 2014. – URL https://www.omg.org/spec/OCL/2.4/PDF. – Zugriffsdatum: 2023-11-27

[15] Object Management Group: *OMG® Unified Modeling Language® (OMG UML®) Versions formal*. 2017. – URL https://www.omg.org/spec/UML/#spec-versions-formal. – Zugriffsdatum: 2023-12-10

[16] Oriol, Xavier ; Teniente, Ernest: Incremental checking of OCL constraints through SQL queries. In: Brucker, Achim D. (Hrsg.) ; Dania, Carolina (Hrsg.) ; Gottlob, Georg (Hrsg.) ; Gogolla, Martin (Hrsg.): *OCL@MoDELS* Bd. 1285, CEUR-WS.org, 2014, S. 23–32. – URL http://ceur-ws.org/Vol-1285/

[17] Rupp, Chris ; Queins, Stefan: *UML 2 glasklar: Praxiswissen für die UML-Modellierung, 4. Auflage.* Hanser Verlag, 2012

[18] Wilhelm, Reinhard ; Seidl, Helmut ; Hack, Sebastian: *Compiler design: syntactic and semantic analysis.* Springer Science & Business Media, 2013

[19] Zehetner, Christoph: *An adaptable OCL engine for validating models in different tool environments*, Technische Universität Wien, Dissertation, 2013

# A Anhang

## A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

| Tool | Verwendung |
|---|---|
| LaTeX | Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments |
| ChatGPT | KI-Werkzeug verwendet zur Erstellung von Programmcode |
| Diagrams.net | Diagramm- und Zeichenprogramm verwendet zur Erstellung von Abbildungen |

## A.2 Sourcecodebeispiele

In den Abbildungen A.1 und A.2 sind die die Sourcecodebeispiele der Verwendung des USE-OCL-Parsers dargestellt.

```
1    public AutoCompletionParser(MModel model, String input) {
2       PrintWriter err = new PrintWriter(new StringWriter());
3       parserResult = new ParserResult();
4
5       parse(model, input, err);
6
7       processResult();
8    }
```

Abbildung A.1: Sourcecodebeispiel der Verwendung des USE-OCL-Parsers I

```
1   private void parseOCLOnly(MModel model, String input, PrintWriter err) {
2       //remove leading ? and possible spaces following these
3       input = input.trim().replaceFirst("^(\\?\\?|\\?)\\s*", "");
4
5
6       String trailingDelimiter = getTrailingDelimiter(input);
7       String type = typeOfInput(model, input, err, trailingDelimiter);
8
9       if (type == null) {//input doesnt compile as type
10          List<String> tokens = splitInput(input);
11
12          if (tokens.size() == 1) {
13              handle(input, trailingDelimiter);
14          } else {
15              for (String token : tokens) {
16                  token = handleParenthesis(token.trim());
17
18                  parseOCLOnly(model, token.trim(), err); //could remove something ->
                    recursion
19              }
20          }
21      } else { //input compiles as type
22          handleAddingType(type, trailingDelimiter);
23      }
24  }
```

Abbildung A.2: Sourcecodebeispiel der Verwendung des USE-OCL-Parsers I

## Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| Ort | Datum | Unterschrift im Original |
|---|---|---|