HAW HAMBURG

BACHELOR THESIS
Matthias Marschalk

Entwurf und Implementierung eines Mechanismus zur Erhöhung der Modularität und Wiederverwendbarkeit in einem Modellierungstool

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

Matthias Marschalk

Entwurf und Implementierung eines Mechanismus zur Erhöhung der Modularität und Wiederverwendbarkeit in einem Modellierungstool

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
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: 17.06.2025

Matthias Marschalk

Thema der Arbeit

Entwurf und Implementierung eines Mechanismus zur Erhöhung der Modularität und Wiederverwendbarkeit in einem Modellierungstool

Stichworte

Importmechanismen, Import-Statements, Modellierungssprachen, UML, USE, ANTLR

Kurzzusammenfassung

Modularität ist ein ein entscheidendes Prinzip zur Reduktion von Komplexität und zur Förderung der Wiederverwendbarkeit in der Softwareentwicklung wie auch der Modellierung. Das Open-Source Modellierungswerkzeug USE ermöglicht die textuelle Spezifikation und Validierung von Systemen, bietet bislang jedoch keine Unterstützung für modulare Strukturen. In dieser Arbeit wird ein Importmechanismus für USE entwickelt, der es erlaubt, externe Modelle in bestehende Spezifikationen einzubinden. Dadurch können wiederverwendbare Modellbestandteile, wie etwa benutzerdefinierte Datentypen, zentral definiert und projektübergreifend genutzt werden. Dies verringert Redundanz, vereinfacht die Wartung und verbessert die Konsistenz modellbasierter Spezifikationen.

Matthias Marschalk

Title of Thesis

Design and implementation of a mechanism to increase modularity and reusability in a modelling tool

Keywords

import mechanisms, import statements, modelling languages, UML, USE, ANTLR

Abstract

Modularity is a key principle to reduce complexity and promote reusability in software development and modeling. The open-source modeling tool USE enables the textual

specification and validation of systems but currently lacks support for modular structures. This work introduces an import mechanism for USE that allows external models to be integrated into existing specifications. As a result, reusable model components, such as user defined data types, can be defined once and then utilized across multiple models. This reduces redundancy, facilitates maintenance and improves the consistency of model-based specifications.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ANTLR Another Tool for Language Recognition.

AST Abstract Syntax Tree.

EBNF Erweiterte Backus-Naur-Form.

GUI Graphical User Interface.

OCL Object Constraint Language.

UML Unified Modeling Language.

USE UML-based Specification Environment.

XMI XML Metadata Interchange.

Listings

1 Einleitung

1.1 Motivation

Modularität ist ein zentrales Gestaltungsprinzip, um die Verständlichkeit, Wartbarkeit und Wiederverwendbarkeit von Softwaresystemen zu gewährleisten. Sie ermöglicht es, komplexe Systeme in überschaubare, logisch zusammenhängende Einheiten zu gliedern, die unabhängig voneinander entwickelt und genutzt werden können. Dieses Prinzip hat sich nicht nur in der Softwareentwicklung etabliert, sondern gewinnt auch in der Modellierung zunehmend an Bedeutung. Modellierungswerkzeuge sollten daher Konzepte bereitstellen, die den Aufbau modularer und wiederverwendbarer Modelle unterstützen.

Gerade mit steigendem Projektumfang ist es essenziell, Modelle so zu strukturieren, dass wiederverwendbare Komponenten nur einmal definiert und in verschiedenen Kontexten eingebunden werden können. Dies vermeidet redundante Strukturen, reduziert den Pflegeaufwand und verringert die Fehleranfälligkeit. Um dies zu unterstützen, braucht es Mechanismen, mit denen sich externe Modelle problemlos in bestehende Spezifikationen integrieren lassen.

Das *UML-based Specification Environment* ist ein Open-Source-Werkzeug zur textuellen Modellierung auf Basis einer Teilmenge der UML und der OCL. Es erlaubt die Beschreibung von Systemen sowie deren Validierung anhand formal definierter Integritätsbedingungen. Bislang bietet USE jedoch keine Unterstützung für modulare Strukturen durch die Wiederverwendung von Spezifikationen. Häufig genutzte Elemente müssen daher in jeder Spezifikation neu definiert werden, was zu unnötigen Doppelungen und erhöhtem Modellierungsaufwand führt. Besonders deutlich zeigt sich der Bedarf im Zuge der Erweiterung von USE um benutzerdefinierte Datentypen. Da solche Typen in vielen Modellen wiederverwendet werden, bietet sich hier ein typischer Anwendungsfall für ein Konzept zur Modularisierung.

Im Rahmen dieser Ausarbeitung wird also das Ziel verfolgt, USE um einen Importmechanismus zu erweitern, der die Modularität und Wiederverwendbarkeit von Modellen und ihrer Bestandteile verbessert. Zu diesem Zweck wird ein neues Sprachelement entwickelt und in die Grammatik der USE-Modellierungssprache integriert, sodass Modelle aus externen Spezifikationen eingebunden und verwendet werden können.

1.2 Aufbau

Diese Arbeit gliedert sich in die Kapitel Grundlagen, Spezifikation, Entwurf und Implementierung, Evaluation, Diskussion und Ausblick sowie Fazit.

Im Kapitel Grundlagen wird zunächst ein Überblick über die Anwendungsdomäne und theoretischen Hintergründe von USE gegeben, bevor Importmechanismen definiert und deren Eigenschaften beleuchtet werden.

Danach wird im Kapitel Spezifikation die Zielsetzung präzisiert und eine Abgrenzung dahingend vorgenommen, was nicht als Teil der Anforderungen behandelt wird. Es folgt die Definition der funktionalen und nichtfunktionalen Anforderungen.

Das Kapitel Entwurf und Implementierung beinhaltet das Design der Umsetzung und die getroffenen Entwurfsentscheidungen. Die Konzeption des Importmechanismus wird anhand der Grammatik, dem abstrakten Syntaxbaum, der Verarbeitung von Import-Deklarationen und dem Metamodell erklärt. Außerdem werden das Testkonzept und die Dokumentation vorgestellt.

Darauf folgt im Kapitel Evaluation die Untersuchung des Entwurfs und der Implementierung dahingegend, ob die funktionalen und nichtfunktionalen Anforderungen erfüllt werden.

Anschließend werden im Kapitel Diskussion und Ausblick ausgewählte Entwurfsentscheidungen und ihre Alternativen dargestellt, bevor ein Ausblick darauf gegeben wird, welche möglichen Weiterentwicklungen sich aus der Umsetzung des Importmechanismus ergeben.

Zuletzt wird im Kapitel Fazit das Ergebnis der Umsetzung des Importmechanismus präsentiert.

2 Grundlagen

2.1 UML

Die *Unified Modeling Language (UML)* ist ein weit verbreiteter Standard zur visuellen Modellierung von Software- und anderen Systemen. Sie wurde von der *Object Management Group* entwickelt und stellt eine einheitliche Modellierungssprache zur Verfügung, um die Struktur und das Verhalten komplexer Systeme zu dokumentieren und zu analysieren. UML wird im Software- und Systems-Engineering eingesetzt, um Anforderungen in Systemarchitekturen zu überführen und über Entwurfsentscheidungen zu kommunizieren.

Ein zentrales Element der UML ist das Klassendiagramm[9, 21], das die statischen Strukturen des Systems, wie die enthaltenen Klassen mit ihren Attributen und Methoden sowie deren Beziehungen untereinander beschreibt. Darauf aufbauend können dynamische Aspekte im System mit Sequenzdiagrammen[9, 565] beschrieben werden. Diese stellen exemplarische Interaktionen zwischen Objekten dar und dienen der Validierung von Szenarien. Objektdiagramme[9, 285] hingegen veranschaulichen den Zustand von konkreten Objekten und deren Beziehungen zu einem bestimmten Zeitpunkt. Dies erlaubt, die Korrektheit und Konsistenz der statischen Modellstruktur zu überprüfen.

Im Folgenden werden die wichtigsten Elemente der UML-Modellierung vorgestellt, die auch in der *UML Specification language* des USE-Projektes zu finden sind. Im Rahmen der Implementierung des Importmechanismus in USE ist die Betrachtung der Bestandteile der UML-Spezifikation eine wichtige Grundlage dafür, dass die korrekte Funktionsweise der importierten Modelle gewährleistet wird. Alle Begriffe und Definitionen stützen sich auf die aktuelle Version 2.5.1 der offiziellen UML-Spezifikation.[9]

2.1.1 Modell

Ein Modell *(engl. model)* stellt eine vollständige Beschreibung des Systems dar, in der alle Elemente zusammengefasst werden, die dieses System bilden. Es dient also als Container für die verschiedenen Bestandteile eines UML-Diagramms, wie Klassen und Assoziationen. Dabei repräsentiert es typischerweise das oberste Element einer Modellstruktur.[9, 247-248]

2.1.2 Classifier

Ein Classifier ist ein abstraktes Modellierungselement, das Instanzen mit gemeinsamen strukturellen und funktionalen Merkmalen zusammenfasst. Diese Merkmale umfassen unter anderem Attribute, Assoziationen und Operationen. Classifier können abstrakt oder konkret sein, d.h., sie können entweder direkt instanziierbar sein oder nur über Spezialisierung. Sie dienen als Grundlage von Modellelementen wie Klassen, Datentypen oder Signalen.[9, 129-132]

2.1.3 Class

Eine Klasse *(engl. class)* ist ein spezieller Typ von Classifier, der Attribute und Operationen definiert und somit sowohl strukturelle als auch funktionale Merkmale aufweist. Klassen sind zentrale Bausteine im Klassendiagramm und können in Generalisierungsbeziehungen zueinander stehen.[9, 194-198]

2.1.4 Assoziation

Mit einer Assoziation *(engl. association)* wird eine strukturelle Beziehung zwischen zwei oder mehr Classifiern, typischerweise Klassen, beschrieben. Assoziationen können mit Rollen, Multiplizitäten und Navigierbarkeit näher spezifiziert werden. Multiplizitäten bestimmen, wie viele Objekte an einer Beziehung teilnehmen können, und Navigierbarkeit gibt an, von welcher Seite aus auf die zugehörigen Objekte zugegriffen werden kann. Mit einer Assoziationsklasse kann die Assoziation mit einer eigenen Klasse verbunden werden, sodass die Beziehung zwischen Objekten zusätzliche Attribute und Operationen enthalten kann.[9, 199-208]

2.1.5 Datentyp

Ein Datentyp *(engl. datatype)* ist ein Classifier, der zur Beschreibung von Werten verwendet wird. Im Unterschied zu Klassen verfügen Datentypen über keine eigene Identität und werden nur über ihre Werte unterschieden. Beispiele sind einfache Typen wie *Integer*, *String* oder benutzerdefinierte Typen wie *Date*. Datentypen können nicht an Assoziationen teilnehmen.[9, 167-169]

2.2 OCL

Die *Object Constraint Language (OCL)* ist eine formale Sprache, die zur Beschreibung von Ausdrücken in UML-Modellen dient. OCL ergänzt die modellbasierte Darstellung der UML um eine Spezifikationssprache, beispielsweise um Einschränkungen *(engl. constraints)* für Attribute, Beziehungen oder Operationen zu definieren. Die Sprache ist streng typisiert und wird vor allem in Form von Invarianten, Vor- und Nachbedingungen eingesetzt, um valide Systemzustände zu beschreiben. OCL-Ausdrücke können auf einer konkreten Systeminstanz ausgewertet werden, um vorher definierte Integritätsbedingungen zu überprüfen. Die Auswertung ist seiteneffektfrei, kann also den Systemzustand nicht verändern.[7, 5]

Ein genaues Verständnis zentraler OCL-Konzepte ist im Hinblick auf die Implementierung eines Importmechanismus wichtig, da importierte Modellelemente in bestehende Systemzustände eingebettet werden und die korrekte Prüfung von Integritätsbedingungen auch über Modellgrenzen hinweg gewährleistet werden muss. Daher werden im Folgenden kurz die wichtigsten Elemente der OCL im Kontext von USE erläutert.

2.2.1 Invarianten

Eine Invariante ist eine Bedingung, die für alle Instanzen einer Klasse über ihre gesamte Lebensdauer gültig sein muss. Sie wird üblicherweise im Kontext von Classifiern formuliert und kann sich auf Attribute, Assoziationen oder zusammengesetzte Bedingungen beziehen.[7, 8]

2.2.2 Vor- und Nachbedingungen

Vorbedingungen *(engl. preconditions)* spezifizieren, unter welchen Bedingungen eine Operation aufgerufen werden darf. Sie beschreiben Anforderungen an den Zustand einer Instanz vor der Ausführung der Operation. Demgegenüber definieren Nachbedingungen *(engl. postconditions)* den Zustand einer Instanz nach erfolgreicher Ausführung der Operation. Gemeinsam legen sie das erwartete Verhalten von Methoden fest und können zur Verifikation der Systemlogik verwendet werden.[7, 8-9]

2.3 UML-based Specification Environment

Das *UML-based Specification Environment (USE)*[4] ist eine Open-Source Software zur Beschreibung von Softwaresystemen. In USE können Systemkomponenten modelliert und Simulationen konkreter Instanzen durchgeführt werden.

Mithilfe der *USE specification language*, die auf einer Teilmenge der *Unified Modeling Language (UML)*[9] und der *Object Constraint Language (OCL)*[7] basiert, lassen sich textuell Modelle definieren (siehe Listing 2.1). Diese Modelle enthalten Eigenschaften aus UML-Klassendiagrammen wie beispielsweise Klassen und Assoziationen. Darüber hinaus lassen sich mit OCL-Ausdrücken Integritätsbedingungen für die Modellelemente definieren.[9]

Diese Spezifikationen können kompiliert werden, woraufhin sich konkrete Systemzustände erzeugen und Anwendungsszenarien durchlaufen lassen. Dabei können Integritätsbedingungen mithilfe von Ausdrücken in OCL geprüft werden. Außerdem werden verschiedene Sichten und Diagramme zur Einsicht von Informationen über das System zur Verfügung gestellt.

```
model Company

class Employee
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end

constraints

context Employee::raiseSalary(rate : Real) : Real
  post raiseSalaryPost:
    salary = salary@pre * (1.0 + rate)
  post resultPost:
    result = salary
```

Listing 2.1: USE-Modell mit einer Klasse sowie Vor- und Nachbedingungen

Die Programmiersprache des Projektes ist Java und die GUI ist eine Swing-Anwendung. Die Grammatik der *USE specification language* und deren Parserkomponenten werden mithilfe des Parsergenerators *ANTLR (v3)* erzeugt. Neben der visuellen Benutzeroberfläche wird eine Kommandozeilen bereitgestellt, mit der die Anwendung über textuelle Befehle gesteuert werden kann.

2.4 Konstruktion und Erkennung von Grammatiken mit ANTLR

Another Tool for Language Recognition (ANTLR) v3 ist ein Parsergenerator zur Definition und Erkennung von Grammatiken. Es wird insbesondere zur Entwicklung von Domänenspezifischen Sprachen (DSLs), Compilern, Interpretern und Werkzeuge zur Sprachanalyse eingesetzt.[13, 21] In Vorbereitung auf die Erweiterung der Grammatik von USE durch ein Import-Statement werden in diesem Kapitel die wesentlichen Konzepte von ANTLR vorgestellt.

2.4.1 Arten von Grammatiken in ANTLR

ANTLR verarbeitet kontextfreie Grammatiken und basiert intern auf dem *LL(*)*-Parsing. Das bedeutet, die Eingabe wird von links nach rechts gelesen und eine Linksableitung berechnet. Im Gegensatz zu klassischen *LL(k)-Parsern*, die eine feste Anzahl von k Symbolen vorausschauen, ist bei einem *LL(*)-Parser* das *Lookahead* nicht fest, sondern wird von einem deterministischen endlichen Automaten (DFA) bestimmt, der bestimmte Schlüsselwörter auswertet. Dies erlaubt dem *LL(*)-Parser* eine größere Menge an natürlichen Sprachen zu erkennen als einem klassischen *LL(k)-Parser*.[13, 268-269]

ANTLR ist zwar nicht für kontextabhängige Grammatiken geeignet, bietet über sogenannte *semantische Prädikate* eine Möglichkeit, um mit eingebettetem Java-Code zusätzliche Regeln festzulegen, wodurch die Menge der erkennbaren Sprachen erweitert wird.[13, 294]

Grammatiken in ANTLR bestehen aus Regeln, die in einer abgewandelten *erweiterten Backus-Naur-Form (EBNF)* spezifiziert werden. Ihr Aufbau ist modular und umfasst Lexer-Regeln, die Zeichenketten in Tokens umwandeln, Parser-Regeln, die Token in syntaktisch gültige Konstrukte strukturieren und semantische Aktionen, die die Einbettung von Java-Code zur Laufzeit ermöglichen.[13, 107]

2.4.2 Semantische Aktionen

ANTLR erlaubt die direkte Einbettung von Java-Code innerhalb der Grammatik und verbindet somit die Grammatikdefinition mit dem Laufzeitverhalten. Dies ermöglicht beispielsweise die Erzeugung von Knoten des abstrakten Syntaxbaums (AST) oder das Sammeln von Symbolnamen.[13, 130]

2.4.3 Rückgabewerte und Attribute

Parser-Regeln können Rückgabewerte besitzen und benannte Attribute definieren, wodurch komplexe Datenstrukturen während des Parsens aufgebaut werden können. Die Rückgabe erfolgt über benannte Variablen und erlaubt die Übergabe von Objekten an übergeordnete Regeln.[13, 101-102]

2.4.4 Abstrakter Syntaxbaum (AST)

ANTLR bietet die Möglichkeit, Eingaben in einen abstrakten Syntaxbaum AST als Zwischenrepräsentation zu überführen. Es wird eine Standard-Implementierung für Knoten des AST bereitgestellt. Es können jedoch auch eigene Datenstrukturen verwendet werden. Die Konstruktion des Baumes erfolgt im Java-Code der semantischen Aktionen. Aus dem AST lässt sich anschließend die endgültige Repräsentation der Eingabe ableiten.[13, 168]

2.5 Importmechanismen

2.5.1 Definition

Das Prinzip der Modularisierung beschreibt die Aufteilung von Systemen in logische Einheiten, die sich an ihren Verantwortlichkeiten orientieren und ist seit Langem ein essenzielles Konzept, etwa in Programmiersprachen. Diese Strukturierung trägt dazu bei, die Flexibilität eines Produkts und dessen Verständlichkeit zu erhöhen.[12, 1053-1054] Auch auf der Ebene von Modellierungssprachen ist die Organisation von Systemspezifikationen in einzelne Modellartefakte mittlerweile eine gängige Funktionalität. So werden beispielsweise in SysML *namespaces* und *packages* spezifiziert, die Sprachelemente mit der Absicht kapseln, dass diese Einheiten an anderer Stelle wiederverwendbar sind.[10, 23-24]

Importmechanismen unterstützen dieses Prinzip, indem sie die notwendigen Sprachkonzepte und die technische Implementierung zum Einbinden dieser Strukturen bereitstellen. So definiert die Spezifikation des XMI-Dateiformats, das unter anderem zum Austausch von UML-Klassendiagrammen verwendet wird, ein Sprachelement zum Import externer XMI-Schemata.[8, 45-46] Die technische Umsetzung zur Verarbeitung dieser Importe muss allerdings von den Werkzeugen bereitgestellt werden, die mit diesem Dateiformat umgehen. In der SysML-Spezifikation findet sich neben einer Syntax für Importe auch eine umfangreiche Standardbibliothek für gängige Modellelemente.[10, 28-29] Außerdem stellt die Object Modeling Group (OMG) eine Pilotimplementierung für den SysMLv2-Standard bereit.[5]

2.5.2 Eigenschaften von Importmechanismen

Für die Konzeption einer Import-Funktionalität ist eine ausführliche Auseinandersetzung mit der gängigen Praxis und den wesentlichen Eigenschaften von Importmechanismen erforderlich. In ihrem Paper „*A synopsis on Import Statements in Modeling Languages*“ analysieren Jansen, Rumpe und Schmalzing[11] verschiedene Gestaltungsansätze zur Umsetzung von Importmechanismen in Modellierungssprachen mit dem Ziel, typische Designentscheidungen zu formalisieren und zu vereinfachen. Es werden relevante Charakteristika identifiziert und in Hinblick auf ihre Vor- und Nachteile analysiert. Die gewonnenen Erkenntnisse bilden eine wesentliche theoretische Grundlage für den in dieser Arbeit ausgearbeiteten Entwurf des Importmechanismus und werden im Folgenden näher vorgestellt.

Die Autoren kategorisieren die Eigenschaften von Importmechanismen in drei Gruppen. Intrinsische Charakteristika betreffen vor allem die interne Umsetzung und die technische Infrastruktur, die für eine Modellierungssprache bereitgestellt werden muss. Demgegenüber stehen explizite Eigenschaften, die sich unmittelbar auf den syntaktischen Aufbau des Import-Statements auswirken. Darüber hinaus werden Mischformen dieser Eigenschaften identifiziert, die sowohl die infrastrukturelle Aspekte als auch den syntaktischen Aufbau betreffen. Diese Mischformen beruhen häufig darauf, dass entweder interne Designentscheidungen Auswirkungen durch zusätzliche Modellierungseigenschaften unterstützt werden, oder umgekehrt, dass syntaktische Konstrukte sich auf die technische Infrastruktur auswirken.[11, 1164]

Auswertungsstrategie

Jansen et al. (2024) beschreiben die Auswertungsstrategie als eine intrinsische Eigenschaft von Importen, die bestimmt, wie und zu welchem Zeitpunkt Import-Statements ausgewertet, also die referenzierten Ressourcen tatsächlich geladen werden. Es wird die Gegenüberstellung gemacht zwischen *instant* und *lazy loading*. Dabei beschreibt das *instant loading* die sofortige Auswertung der referenzierten Ressource. Beim *lazy loading* hingegen werden externe Ressourcen nur dann geladen, wenn sie auch tatsächlich benötigt werden. Die Wahl der Auswertungsstrategie hat in der Regel zwar keinen unmittelbaren Einfluss auf die Struktur des Modells, wirkt sich aber auf die Effizienz des Mechanismus aus.[11, 1164]

In der Bewertung wird die Strategie des *lazy loading* vor allem aufgrund ihrer höheren Effizienz favorisiert. Externe Ressourcen müssen nur bei Bedarf geladen werden, was den Initialaufwand verringert und eine Pufferung für Mehrfachzugriffe ermöglicht. Nicht genutzte Importe haben darüber hinaus keinen negativen Einfluss auf die Laufzeit des Mechanismus. Demgegenüber steht in Hinblick auf die Umsetzung eines Lazy-Loading-Ansatzes ein höherer Implementierungsaufwand als beim *instant loading*, da eine komplexere Logik zur Verwaltung und Auflösung von Ressourcen erforderlich ist. Dennoch wird als Auswertungsstrategie das *lazy loading* empfohlen, insbesondere in komplexeren Modellierungssprachen mit umfangreichen Modellen.[11, 1165]

Importausführung

Eine weitere intrinsische Eigenschaft betrifft die Ausführung von Importen und bezieht sich auf die Art und Weise, wie die Integration der referenzierten Elemente in das importierende Modell umgesetzt ist. Es werden hier zwei gegensätzliche Ansätze identifiziert, die zum einen als vollständige Inklusion und zum anderen als lose Kopplung bezeichnet werden. Bei vollständiger Inklusion werden externe Elemente bereitgestellt, indem sie direkt in das betreffende Modell kopiert werden. Dagegen werden bei der losen Kopplung referenzierte Elemente zwar in den Speicher geladen, bleiben aber auf technischer Ebene getrennte Ressourcen.[11, 1164]

Im Hinblick auf die Bewertung wird eine eindeutige Empfehlung für das Prinzip der losen Kopplung ausgesprochen. Eine vollständige Inklusion wird als problematisch angesehen, da sie in ihrer Funktionsweise einem automatisierten Copy-and-Paste-Mechanismus entspricht. Obwohl das Modellierungswerkzeug die Einbindung der externen Ressource übernimmt, wird hier die Zielsetzung der Wiederverwendbarkeit verletzt. Dagegen verfolgen nahezu alle modernen Programmiersprachen den Ansatz der losen Kopplung bei ihren Importmechanismen, indem lediglich Verweise auf externe Ressourcen bestehen. Dadurch wird eine unnötige Vermischung von Artefakten vermieden und eine klare Struktur beibehalten.[11, 1165]

Transitivität

Als letzte intrinsische Charakteristik wird die Transitivität benannt, also der Umstand, dass Import-Statements Artefakte referenzieren, die wiederum selbst Import-Statements enthalten, wodurch sich indirekte Abhängigkeiten ergeben. In diesem Zusammenhang

müssen Entscheidungen darüber getroffen werden, wie diese transitiv referenzierten Elemente eingebunden werden. Eine Möglichkeit ist hier, dass diese transitiven Abhängigkeiten ebenfalls geladen werden, um die Funktionsfähigkeit der direkt importierten Elemente zu gewährleisten, selbst wenn sie nicht explizit importiert wurden. Als Beispiel für diesen Ansatz wird die Programmiersprache Java angeführt. Im Gegensatz dazu steht der flache Import, bei dem transitive Abhängigkeiten explizit mit zusätzlichen Import-Statements importiert werden müssen. Dies bedeutet einen erhöhten Modellierungsaufwand, ermöglicht aber auch ein höheres Maß an Kontrolle und Flexibilität. Dieses Prinzip wird beispielsweise in der Programmiersprache R angeboten.[11, 1164]

Die Entscheidung über den Umgang mit transitiven Abhängigkeiten hängt aus Sicht der Autoren stark vom Anwendungsbereich ab. Demnach ist es denkbar, dass in stark technisch geprägten Domänen die maximale Kontrolle durch den expliziten flachen Ansatz erwünscht sein kann. Typischerweise gibt es diese Anforderung aber nicht im Kontext der Modellierung, weshalb die Vorteile eines transitiven Importmechanismus überwiegen, insbesondere mit Blick auf die Benutzerfreundlichkeit.[11, 1165-1166]

Wildcards

Mit der expliziten Eigenschaft der sogenannten Wildcard wird ein syntaktisches Mittel von Import-Statements beschrieben, das es erlaubt, mehrere externe Elemente durch einen Platzhalter gemeinsam zu importieren, ohne diese einzeln benennen zu müssen. Wildcards, welche typischerweise durch * dargestellt werden, ermöglichen somit Importe zu bündeln und die Lesbarkeit der Modelle zu verbessern.[11, 1164]

Platzhalter sind ein etabliertes Konstrukt in vielen Programmier- und Modellierungssprachen, da sich mit ihnen Redundanz reduzieren und die Benutzerfreundlichkeit erhöhen lässt. Trotzdem können auch durch Wildcards unerwartete Mehrdeutigkeiten in der Referenzierung entstehen, welche zu Fehlern im Modell führen können. Dies lässt sich allerdings durch entsprechendes Tooling vermeiden, welches die Wohlgeformtheit während der Modellierung überprüft. Außerdem bedeutet die Semantik des Platzhalters grundsätzlich erst einmal, dass alle Elemente einer referenzierten Ressource geladen werden sollen. Dem könnte zum Beispiel mit der Umsetzung von *Lazy Loading* entgegengewirkt werden, da sichergestellt wird, dass auch nur die benötigten Elemente tatsächlich importiert werden. Daher empfehlen Jansen et al. (2024) die Unterstützung eines Platzhalter-Konstrukts als Teil der Modellierungssprache.[11, 1166]

Aliasing

Eine Problematik, die sich durch die Verteilung von Elementen über mehrere Ressourcen ergibt, ist, dass identische Namen vergeben werden können, die beim gemeinsamen Import Namenskonflikte verursachen. Dies führt dazu, dass Elemente nicht mehr eindeutig identifiziert werden können. Die explizite Eigenschaft des *Aliasing* wirkt dem entgegen, indem die Umbenennung von Elementen während ihres Imports erlaubt wird. So wird gewährleistet, dass Elemente mit identischen Bezeichnern aus externen Namensräumen einen einzigartigen Namen im importierenden Kontext haben.[11, 1164]

In der Bewertung der Aliasing-Mechanik ist hervorzuheben, dass eine wichtige Problematik gelöst wird, ohne die sonst gewährleistet werden müsste, dass alle Artefakte von beispielsweise Bibliotheken global eindeutige Namen verwenden müssen. Außerdem lassen sich so möglicherweise passendere Namen im Kontext des importierenden Modells vergeben. Als einziger Nachteil könnte hier benannt werden, dass es nicht unbedingt eine häufig benutzte Funktionalität ist. Somit wird die Unterstützung von *Aliasing* als eine wertvolle Eigenschaft empfohlen.[11, 1166]

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
```

Abbildung 2.1: Aliasing in JavaScript

Sichtbarkeit

Das Konzept unterschiedlicher Sichtbarkeiten ist sowohl in Programmier- als auch in Modellierungssprachen etabliert, um den äußeren Zugriff auf Elemente zu kontrollieren. Übertragen auf Import-Statements bezeichnet die Sichtbarkeit die externe Eigenschaft, dass für importierte Elemente festgelegt werden kann, ob diese auch wieder exportiert werden können. So kann der Zugriff auf Elemente eingeschränkt werden, die nur für den internen Gebrauch bestimmt sind.[11, 1164]

Zwar ist es denkbar, dass Szenarien existieren, in denen die Sichtbarkeit von Importen relevant für die Modellierung ist, allerdings lässt sich kein genereller Nutzen aus diesem Konzept ableiten. Insbesondere im Kontext der Modellierung wird eher eine Abstraktion

von technischen Details angestrebt wodurch der Mehrwert einer solchen Sichtbarkeitsfunktionalität fraglich erscheint. Im Gegenteil könnte eine solche Sichtbarkeitsregelung eher für unnötige Komplexität sorgen.[11, 1166]

Position der Importe

Die Position von Import-Statements innerhalb eines Modells kann sowohl semantische als auch technische Auswirkungen haben. In vielen Programmiersprachen wie Java sind Importe an einer bestimmten Position am Anfang des Artefaktes angesiedelt und gelten für den gesamten darunterliegenden Bereich. Demgegenüber gibt es auch Modellierungssprachen, in denen Importe auch in Unterbereichen deklariert werden können, wodurch ein verteilter Import von externen Elementen ermöglicht wird.[11, 1164-1165]

Der dezentrale Ansatz bietet auf der einen Seite höhere Flexibilität, kann aber auch die Übersichtlichkeit eines Modells beeinträchtigen, da Importe an beliebigen Stellen im Modell definiert werden können. Ein zentraler Ansatz dagegen kann mehr Klarheit bedeuten, da alle Importinformationen an einer Stelle gebündelt sind, wodurch auch die technische Umsetzung einfacher ist. Für diese beiden Ansätze wird keine eindeutige Präferenz vorgegeben, allerdings sollte abgewogen werden, ob eine dezentrale Positionierung für den jeweiligen Anwendungsfall sinnvoll ist.[11, 1166]

Importziele

Import-Statements haben grundsätzlich zunächst die Aufgabe, externe Elemente in einen anderen Kontext zu überführen und sie dort zur Verwendung bereitzustellen. Das konkrete Ziel, also die Ressource, die eingebunden wird, kann allerdings unterschiedlich sein und entweder das Artefakt, Sprachelemente aus dem Artefakt oder eine Kombination von beidem umfassen. Beim puren Artefakt-Import wird lediglich eine Referenz auf das Artefakt im Import-Statement angegeben und somit alle enthaltenen Elemente und Funktionalitäten geladen. Dagegen wird beim puren Element-Import das umgebende Artefakt ignoriert und das enthaltene Element direkt eingebunden. Beim Ansatz der Kombination der beiden Ansätze können die Zielressourcen sowohl Artefakte als auch konkrete Elemente sein. Dieser Ansatz wird beispielsweise in Java verfolgt, wobei primär die Artefakte als Ziel dienen, die denselben Namen wie Klassen, Interfaces, etc. haben. Es ist aber auch möglich innere Klassen, oder statische Methoden direkt zu referenzieren.[11, 1165]

Der Ansatz des reinen Artefakt-Imports wird als eher ungeeignet angesehen, da er die Möglichkeiten der Referenzierung sehr einschränkt und Gegebenheiten der Domänensprache, wie Elemente, die sich in eingebetteten Sichtbarkeitsbereichen befinden, nicht abgebildet werden können. Wird hingegen der Ansatz des reinen Element-Imports verfolgt, so lassen sich zwar Elemente feingranularer einbinden, allerdings entsteht der große technische Nachteil, dass die Verbindung zwischen einem importierten Element und dem Artefakt, in dem es sich befindet, verloren geht. Dies kann zu einem erhöhten Aufwand in der Lokalisierung der Elemente führen und wird daher bei der Konzeption einer Modellierungssprache eher nicht empfohlen. Die Kombination der beiden Ansätze kann diese Nachteile kompensieren, da sowohl das gesamte Artefakt, als auch interne Modellelemente als Ziel des Import-Statements dienen können. Eine Problematik, mit der an dieser Stelle umgegangen werden muss, ist die potenzielle Mehrdeutigkeit von Artefakt- und Elementbezeichnern, da die Grenzen zwischen ihnen unter Umständen in der Syntax fließend verlaufen. Hier kann eine systematische Trennung hilfreich sein, allerdings ergibt sich auch ein leicht erhöhter Modellierungsaufwand. Grundsätzlich wird aber der integrierte Ansatz favorisiert, da er neben Benutzerfreundlichkeit in der Modellierung auch ein hohes Maß an Präzision bei der Referenzierung des Importziels ermöglicht.[11, 1166-1167]

```
import java.util.*;
import java.util.stream.Stream;
import static java.lang.String.format;
```

Abbildung 2.2: Verbindung von Artefakt- und Elementbezeichnern in Java

Zyklikalität

Zyklikalität ist eine zentrale Eigenschaft von Importmechanismen, die sich daraus ergibt, dass importierte Elemente ihrerseits wiederum andere Elemente referenzieren können und direkte oder indirekte zyklische Abhängigkeiten entstehen können. Diese Zyklen werden zwar aus architektonischer Perspektive meist als negativ bewertet, in bestimmten Szenarien ist es allerdings denkbar, dass der bewusste Einsatz von zyklischen Abhängigkeiten sich als praktikabel erweist. Im Bereich der Konzeption von Domänensprachen haben sich drei Ansätze etabliert. So können Zyklen grundsätzlich unterbunden werden, wodurch zwar alle negativen Auswirkungen vermieden, aber auch die Modellierungsfreiheit eingeschränkt wird. Andersherum können Zyklen erlaubt, aber die manuelle Auflösung durch

Modellierer*innen, etwa durch ein zusätzliches syntaktisches Konstrukt eingefordert werden. Ein fortgeschrittener Ansatz ist, dass Zyklen erlaubt werden und bei der Verarbeitung eine automatische Zyklenextraktion im Abhängigkeitsgraphen durchgeführt wird, die dafür sorgt, dass alle eingebundenen Artefakte nur einmal durchlaufen werden.[11, 1166]

Für die Bewertung ist zunächst eine Abwägung erforderlich, ob die Berücksichtigung von zyklischen Abhängigkeiten im Kontext der Modellierungssprache sinnvoll ist. Generell ist ein Umgang mit Zyklen aber vor dem Hintergrund der möglichen Anwendungsszenarien sinnvoll und sollte beispielsweise über Validierungsregeln unterbunden werden können. Eine manuelle Auflösung von Zyklen ist aber auf der höheren Abstraktionsebene von Modellierungssprachen zu vermeiden, da technische Details in dem Kontext eher uninteressant sind. Daher wird die automatische Zyklenextraktion als empfohlenes Mittel betrachtet, um eine möglichst große Modellierungsfreiheit zu gewährleisten. Allerdings ist auch zu beachten, dass dieser Ansatz einen höheren Implementierungsaufwand bedeutet.[11, 1167]

Relative Importe

Üblicherweise spezifizieren Import-Statements den absoluten Namen des Ziels, dass eingebunden werden soll, häufig ausgehend von der Wurzel des Projektes. Es kann aber auch die Möglichkeit angeboten werden, dass Zielressourcen relativ gesehen zum importierenden Artefakt qualifiziert werden können.[11, 1166]

Die absolute Qualifikation des Importziels wird als der Standardfall angesehen und sollte in jedem Fall unterstützt werden. Darüber hinaus können sich aber auch Vorteile aus der relativen Qualifikation ergeben, denn sie kann für kürzere Import-Statements sorgen, wenn sich die betreffenden Ressourcen im Namensraum in unmittelbarer Nähe zueinander befinden. Ein Nachteil der in diesem Zusammenhang allerdings berücksichtigt werden muss, ist, dass diese Referenzen auf zwei Arten ungültig werden können, nämlich dann wenn sich die Position der importierten oder der importierenden Ressource verändert.[11, 1167]

3 Spezifikation

3.1 Einleitung

Im Folgenden werden die Anforderungen an die Implementierung der Import-Funktionalität dargelegt und erläutert. Die Struktur orientiert sich dabei am Standard ISO/IEC/IEEE 29148:2018[2] zur Anforderungsanalyse, insofern die darin definierten inhaltlichen Vorgaben auf die vorliegende Problemstellung anwendbar sind. Zu Beginn werden Zielsetzung und Umfang der Implementierung beschrieben. Daraufhin wird die Anwendung im Kontext ihres Einsatzbereiches betrachtet, einschließlich ihrer bestehenden Funktionalitäten und der Eigenschaften der Nutzer*innen. Dabei werden auch Einschränkungen, Annahmen und Abhängigkeiten identifiziert. Auf Grundlage dieser beschriebenen Rahmenbedingungen werden die konkreten funktionalen und nicht-funktionalen Anforderungen an die Implementierung ausgeführt.

3.2 Zielsetzung

Die Zielsetzung für diese Spezifikation ist die Erweiterung des USE-Frameworks um eine Import-Funktionalität, die es Nutzer*innen erlaubt, Elemente aus externen USE-Spezifikationen in das eigene Modell zu integrieren, ohne dass sie erneut definiert werden müssen. Zu diesem Zweck wird ein neues Sprachelement in der Grammatik der USE-Modellierungssprache eingeführt, mit dem Importe deklariert werden können, sodass die importierten Elemente im Scope des Modells sichtbar gemacht und verwendet werden können. Diese neue Funktionalität löst das Problem, dass bisher identische Elemente, wie beispielsweise benutzerdefinierte Datentypen, in jeder Spezifikation, in der sie verwendet werden sollen, mehrfach definiert werden mussten. Dies erhöht den Modellierungsaufwand, da duplizierte Strukturen erzeugt und über mehrere Spezifikationen hinweg konsistent gehalten werden müssen. Die Funktionalität kann also dazu beitragen, den Zeitaufwand für die Modellierung und die Fehleranfälligkeit zu verringern.

3.3 Abgrenzung

Bisher erlaubt USE das Öffnen einer Spezifikation von einem beliebigen Ort auf dem Dateisystem, was insofern bislang unproblematisch war, da ein Modell alle Sprachelemente, die es verwendet, selbst deklariert hat. Mit der Implementierung von Importen stellt sich allerdings die Frage, wie referenzierte Artefakte auf dem Dateisystem zuverlässig lokalisiert werden können. In Programmiersprachen wie z.B. Java gibt es das Sprachkonzept der Packages[6, 209], die eine logische Strukturierung in Namensräume ermöglichen. Die physische Ordnerstruktur auf dem Dateisystem muss diese virtuelle Struktur abbilden, damit bei der Kompilierung die importierten Elemente gefunden und geladen werden können.

Zwar kann man ein Modell in USE auch als eine Art Package oder Namensraum betrachten, allerdings gibt es keine Struktur über einem Modell, die zur Lokalisierung eines anderen Modells genutzt werden könnte. Außerdem wird USE nicht von einem Tooling wie in Java begleitet, um die Einhaltung bestimmter Konventionen zu erzwingen, die die Pfadauflösung ermöglichen. Die Entwicklung einer solchen Struktur könnte zwar langfristig durchaus sinnvoll sein, wird im Rahmen dieser Spezifikation allerdings nicht als Teil der Anforderungen betrachtet, da der Fokus auf der grundlegenden Implementierung einer Import-Funktionalität liegt.

3.4 Funktionale Anforderungen

Aus der Definition der Zielsetzung und der Analyse des Systems und seiner Funktionen und Eigenschaften ergeben sich funktionale Anforderungen, die sich nach drei Aspekten des Systems strukturieren lassen:

- Anforderungen an die Sprachdefinition und Kompilierungslogik
- Anforderungen an die grafische Benutzeroberfläche (GUI) und ihre Funktionalitäten
- Anforderungen an die USE-Shell und ihre Befehle

3.4.1 Anforderungen an die Sprachdefinition und Kompilierungslogik

Im Folgenden werden die Anforderungen dargestellt, welche die Erweiterung der USE-Modellierungssprache sowie die daraus folgenden Anpassungen des Parsers und des Kompilierungsprozesses betreffen.

ID	Anforderung
FR-01	Unterstützung der USE-Grammatik und des Parsers für ein Sprachelement zur Deklaration von Import-Statements
FR-02	Verwendbarkeit importierter Elemente im USE-Modell
FR-03	Importierbarkeit der Classifier eines USE-Modells
FR-04	Berücksichtigung von Invarianten sowie Vor- und Nachbedingungen der importierten Classifier

Tabelle 3.1: Funktionale Anforderungen an Sprachdefinition und Kompilierungslogik

Zentral ist hierbei die Anforderung, in USE-Spezifikationen die Verwendung eines Sprachelements zu ermöglichen, das den Import von Elementen aus einer anderen Spezifikation repräsentiert (FR-01). Dafür muss das Import-Statement so konzipiert sein, dass alle notwendigen Informationen enthalten sind, um die grundlegende Funktionsfähigkeit sicherzustellen. Darüber hinaus sollen zusätzliche Eigenschaften integriert werden, die in Hinblick der Bedienbarkeit und Flexibilität als sinnvoll gelten. Diese Eigenschaften leiten sich aus der Analyse gängiger Praktiken im Bereich der Programmier- und Modellierungssprachen ab. Dieses Sprachelement soll dann vom USE-Parser verarbeitet werden können, sodass es während des Kompilierungsprozesses berücksichtigt werden kann.

Im Rahmen der Kompilierung muss gewährleistet sein, dass die durch das Import-Statement referenzierten Modelle korrekt auf Dateiebene lokalisiert und verarbeitet werden, sodass die importierten Sprachelemente im aktuellen Modell verfügbar gemacht werden (FR-02).

Im Fokus steht dabei, dass die Elemente integrierbar sein müssen, die als Bestandteile anderer Elemente eines USE-Modells verwendet werden können (FR-03). In der USE-Modellierungssprache ist das gemäß UML-Konventionen die Menge der Classifier, also Klassen, benutzerdefinierte Datentypen, Enumerationen und Assoziationen. So wäre ein typischer Anwendungsfall, dass ein benutzerdefinierter Datentyp oder ein Enum aus einer externen Spezifikation als Attributtyp einer Klasse verwendet wird. Diese Nutzung ist in Listing 3.1 skizziert, dabei dient der importierte Datentyp `Date` als Typ des Attributs *birthday* in der Klasse *User*. Ebenso ist die Verwendung einer externen Klasse als Teil

einer Assoziation denkbar, wie etwa bei der Assoziation *ParticipatesIn*, die die Beziehung zwischen der Klasse *User* und der importierten Klasse *Meeting* darstellt.

```
import { Meeting } from "Meetings.use"
import { Date } from "Dates.use"

model User

class User
    attributes
        name: String
        birthday: Date
        num: Integer
end

association ParticipatesIn
    between
        User[*] role participant
        Meeting[*] role meeting
    end
```

Listing 3.1: USE-Modell mit Importen

Ergänzend müssen auch begleitende Modellbestandteile wie Invarianten sowie die Vor- und Nachbedingungen von Operationen berücksichtigt werden, welche die validen Zustände beschreiben, in denen sich Instanzen importierter Classifier befinden dürfen (FR-04). Sie sind zwar bei der Modellierung nicht als aktive Bestandteile importierbar und verwendbar, müssen allerdings auch bei der Kompilierung importierter Modelle erzeugt und zur Laufzeit zur Validierung verfügbar gemacht werden.

3.4.2 Anforderungen an die grafische Benutzeroberfläche (GUI) und ihre Funktionalitäten

Die Implementierung der Import-Funktionalität erfordert eine Erweiterung der grafischen Benutzeroberfläche von USE, um sowohl Darstellung als auch die Interaktion mit importierten Elementen innerhalb der bestehenden Funktionalitäten zu ermöglichen.

Beim Öffnen einer USE-Spezifikation wird nach erfolgreicher Kompilierung das geladene Modell mit seinen Elementen im Modellbrowser angezeigt. Dabei muss für Spezifikationen

ID	Anforderung
FR-05	Unterstützung des Öffnens von Spezifikationen mit Importen
FR-06	Instanziierung importierter Klassen und Assoziationen
FR-07	Zuweisung von Attributwerten für importierte Datentypen und Enums
FR-08	OCL-Auswertung auf importierten Elementen
FR-09	Berücksichtigung importierter Elemente bei Informationen über den Status des Systems

Tabelle 3.2: Funktionale Anforderungen an die grafische Benutzeroberfläche

mit Importen sichergestellt sein, dass sowohl das Hauptmodell als auch alle referenzierten Modelle im Modellbrowser angezeigt werden (FR-05). Dies ermöglicht es Nutzer*innen, einen vollständigen Überblick über das System zu erhalten.

Mithilfe der Objekterzeugung bietet USE eine Funktionalität zur Simulation von Systemzuständen. Es lassen sich Objekte von Klassen erzeugen und mittels Assoziationen verknüpfen. Da referenzierte Elemente nach dem Import Teil des Systems sind, muss diese Funktionalität auch auf importierte Klassen und Assoziationen erweitert werden (FR-06). Dazu gehört, dass Objekte dieser Elemente dann im *Object Diagram View* angezeigt werden.

Im Fenster *Object Properties* lassen sich für Attribute von Objekten Werte festlegen. Diese Funktion muss um die Fähigkeit ergänzt werden, auch für importierte Datentypen und Enumerationen Werte zu deklarieren (FR-07).

Darüber hinaus bietet USE die Möglichkeit OCL-Ausdrücke sowohl zustandslos als auch gegen die aktuelle Systeminstanz auszuführen, etwa zur Überprüfung von Bedingungen oder zum Testen von Operationen für Objekte. Bei der Auswertung dieser Ausdrücke muss sichergestellt werden, dass diese Funktionalität auch für importierte Elemente und ihre Instanzen verfügbar ist (FR-08). Insbesondere muss darauf geachtet werden, dass die Auswertungslogik konsistent bleibt, beispielsweise wenn zwei unterschiedliche Modelle denselben Datentyp importieren, dann muss dieser folgerichtig auch vergleichbar sein.

Ergänzend zu den Mechanismen zur Interaktion mit dem Modell stellt die USE-GUI verschiedene Darstellungen zur Verfügung, die den aktuellen Zustand des Systems widerspiegeln. Dies umfasst unter anderem den *Class Invariant View*, der die Erfüllung von Invarianten des Systems anzeigt, sowie Fenster, welche einen Überblick über die Anzahl instanziierter Objekte und Assoziationen geben. Auch bei diesen Statusanzeigen müssen

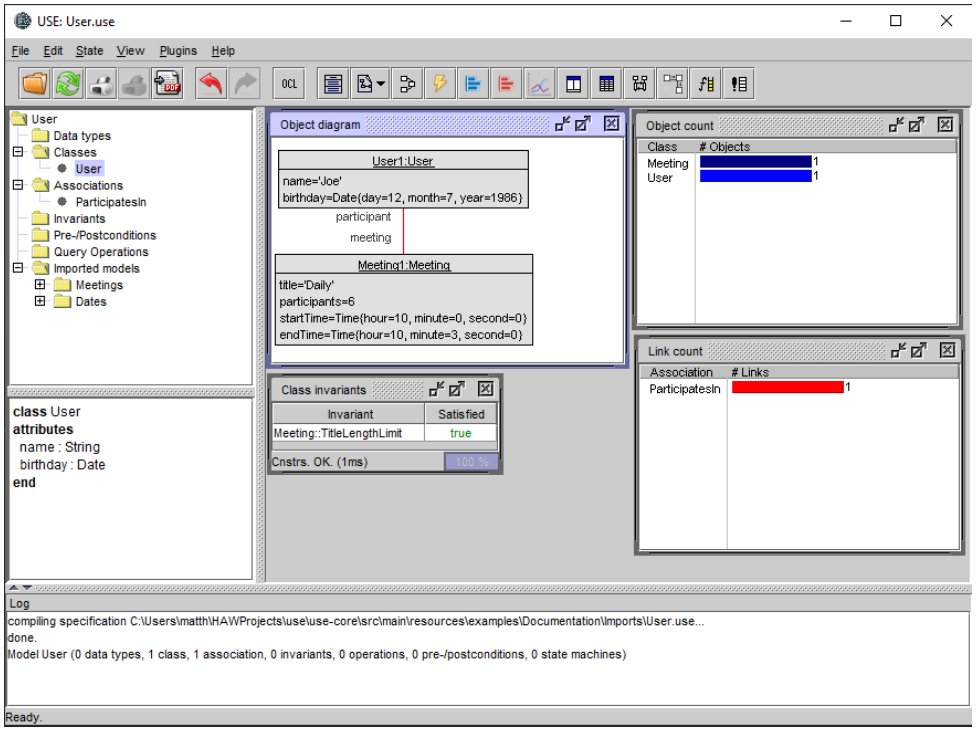

Abbildung 3.1: Funktionalitäten der grafischen Benutzeroberfläche.

referenzierte Modelle integriert werden, sodass Informationen zu importierten Klassen, Objekten und Assoziationen sichtbar sind (FR-09).

3.4.3 Anforderungen an die USE-Shell und ihre ausführbaren Befehle

Die Kommandozeilenoberfläche von USE bietet alternativ zur grafischen Oberfläche eine textbasierte Schnittstelle zur Interaktion mit Modellen und Systeminstanzen. Daher müssen auch auf dieser Ebene Anpassungen vorgenommen werden, sodass die bereitgestellten textuellen Befehle auch mit importierten Elementen umgehen können.

Über die Shell lässt sich mithilfe des Befehls `!open` unter Angabe des Dateipfads eine USE-Spezifikation öffnen, sodass analog zur Anforderung an die GUI auch hier Spezifikationen mit Importen korrekt geladen werden müssen (FR-10).

Nach dem Laden eines Modells ermöglichen die Shell-Befehle `!create`, `!insert` und `!set` das Erzeugen von Objekten, die Verknüpfung von Objekten über Assoziationen

ID	Anforderung
FR-10	Unterstützung von Importen beim Befehl zum Öffnen von Spezifikationen (`!open`)
FR-11	Unterstützung importierter Elemente bei Befehlen zur Erzeugung von Objekten und Assoziationen, sowie zum Setzen von Attributen (`!create`, `!set`, `!insert`)
FR-12	Validierung von Invarianten und Kardinalitäten importierter Modelle bei Befehlen zur Statusprüfung (`!check`)
FR-13	Berücksichtigung importierter Elemente bei der Auswertung von OCL-Ausdrücken und Operationsaufrufen (`!openter`, `!opexit`)

Tabelle 3.3: Funktionale Anforderungen an die USE-Shell

sowie zum Zuweisen von Attributen. Diese Funktionalitäten müssen ebenfalls auf importierte Klassen, Assoziationen und Datentypen anwendbar sein (FR-11). Ein beispielhafter Ablauf ist in Listing 3.2 dargestellt.

```
use> !create joe : User
use> !create daily : Meeting
use> !set meeting.startTime := Time(10, 0, 0)
use> !insert (joe, daily) into ParticipatesIn
```

Listing 3.2: Befehle in der USE-Shell

Zur Analyse des Systemzustands stellt der Befehl *check* die Funktion bereit, Invarianten und Kardinalitäten zu überprüfen. Die Validierung muss dabei alle Elemente des Systems umfassen, sowohl aus dem Hauptmodell als auch aus importierten Modellen (FR-12).

Darüber hinaus bietet die USE-Shell verschiedene Befehle zur Untersuchung des Systemverhaltens. Dazu zählen zum einen die Auswertung von OCL-Ausdrücken und zum anderen die Ausführung von Operationen mit den Befehlen *!openter* und *!opexit*, über die sich Abläufe im System simulieren lassen. Dabei muss gewährleistet sein, dass auch importierte Elemente und ihre Instanzen korrekt referenziert und in die Ausführung einbezogen werden können (FR-13).

3.5 Nichtfunktionale Anforderungen

Über die funktionalen Aspekte hinaus ergeben sich weitere Anforderungen an die Implementierung, die nicht das Verhalten des Systems betreffen, sondern dessen Qualitätsmerkmale. Solche nichtfunktionalen Anforderungen stellen sicher, dass neben der Umsetzung der Funktionalität auch übergeordnete Qualitätsziele verfolgt werden, wie etwa eine hohe Benutzerfreundlichkeit oder eine gute Wartbarkeit. Der Standard ISO/IEC 25010:2011[1] beschreibt eine Einteilung in acht verschiedene Qualitätsmerkmale mit jeweils mehreren Submerkmalen. Im Folgenden liegt der Fokus auf den für die Umsetzung der Import-Funktionalität besonders relevanten Merkmalen Benutzerfreundlichkeit, Wartbarkeit und Kompatibilität.

3.5.1 Benutzerfreundlichkeit

Die Erweiterung des USE-Frameworks um eine Import-Funktionalität muss so gestaltet sein, dass sie den Nutzer*innen einen klaren Mehrwert liefert, indem der Modellierungsaufwand effektiv reduziert wird. Die Import-Funktionalität in USE zielt darauf ab, den Modellierungsaufwand für Nutzer*innen zu reduzieren, indem Wiederverwendbarkeit ermöglicht und redundante Strukturen vermieden werden. Daher muss diese Erweiterung des Systems so gestaltet sein, dass die Hürden für ihre Benutzung möglichst niedrig sind. Dazu trägt eine gute Erlernbarkeit bei. Es sollte intuitiv verständlich sein, wie ein Import-Statement formuliert und verwendet werden muss. Es ist außerdem wichtig, dass Anwendungsfehler, wie beispielsweise syntaktische und semantische Fehler in der Spezifikation, welche die Kompilierung fehlschlagen lassen, transparent und verständlich den Nutzer*innen gegenüber kommuniziert werden. Fehlermeldungen müssen aussagekräftig sein, damit Probleme selbstständig behoben werden können.

3.5.2 Wartbarkeit

Angesichts der Tatsache, dass es sich beim USE-Framework um eine Open-Source Software handelt, kommt der Wartbarkeit besondere Aufmerksamkeit zu. Entwickler*innen, die am Projekt mitarbeiten möchten, sollten mit vertretbarem Zeitaufwand in der Lage sein, sich in die bestehende Codebasis einzuarbeiten. Eine Voraussetzung dafür ist, dass der Quellcode entweder gut dokumentiert oder durch eine konsistente Struktur weitgehend

selbsterklärend ist. Dies ist besonders vor dem Hintergrund relevant, dass die ursprünglichen Autor*innen noch im Projekt mitarbeiten und für Rückfragen zur Verfügung stehen. Ein zentraler Aspekt ist in diesem Zusammenhang die Modularität, neue Funktionalitäten sollten möglichst so gekapselt entwickelt werden, dass die Zuständigkeiten im System klar erkennbar sind und unerwartete Seiteneffekte vermieden werden. Eine gut konzipierte Architektur erleichtert auch die Anpassbarkeit des Systems, da sich Anpassungen und Erweiterungen weniger risikobehaftet und zielgerichteter umsetzen lassen. Letztlich spielt auch die Testbarkeit eine wichtige Rolle. Implementierungen sollten so konzipiert werden, dass sich ihre Bestandteile automatisiert und isoliert testen lassen. Dadurch kann langfristig bei Anpassungen und Neuentwicklungen sichergestellt werden, dass die bestehende Funktionalität nicht beeinträchtigt wird und Fehler frühzeitig erkannt und behoben werden können.

3.5.3 Kompatibilität

Der Aspekt der Kompatibilität ist vor allem im Kontext der Plugin-Funktionalität des USE-Frameworks relevant. Da Plugins teilweise direkt auf Methoden der Kernanwendung zugreifen, etwa zur Kompilierung von Spezifikationen, muss bei der Entwicklung neuer Funktionalitäten darauf geachtet werden, dass bestehende Schnittstellen, insbesondere die Methodensignaturen möglichst unverändert bleiben. Falls eine Erweiterung der Signatur zwingend zur Umsetzung der Anforderungen notwendig ist, sollte die ursprüngliche Methode erhalten bleiben und dokumentiert werden, dass eventuelle neue Funktionen wie der Import-Mechanismus nur über die erweiterte Methode verfügbar sind. So kann die Rückwärtskompatibilität gewährleistet werden, ohne die Erweiterbarkeit des Systems einzuschränken.

4 Entwurf und Implementierung

Auf Grundlage der zuvor definierten funktionalen und nichtfunktionalen Anforderungen können nun die wesentlichen Anpassungen und Erweiterungen definiert werden, die im Zuge einer erfolgreichen Umsetzung der Import-Funktionalität für Modelle im USE-System vorgenommen werden müssen.

Im Folgenden wird die Konzeption des Importmechanismus mit den getroffenen Entwurfsentscheidungen vorgestellt. Dies umfasst zunächst die Erweiterung der Grammatik der *UML based specification language* um ein Sprachkonstrukt, welches die textuelle Beschreibung des Einbindens von Elementen externer Modelle repräsentiert. Dann wird auf die Anpassungen an der Konstruktion des abstrakten Syntaxbaums und des Metamodells zur Transformation des Sprachkonstruktes in eine interne Repräsentation der referenzierten Ressource eingegangen.

Im Anschluss wird das Testkonzept erläutert, das auf verschiedenen Ebenen die funktionale Korrektheit der implementierten Erweiterungen und des Gesamtsystems sicherstellt. Abschließend erfolgt eine Übersicht über die Ergänzungen der Dokumentation, die dazu dienen sollen, die Benutzung der neuen Funktionalität zu erläutern.

4.1 Konzeption des Importmechanismus

4.1.1 Grammatik

Im Rahmen des Kompilierungsprozesses wird der Zeichenstrom der USE-Spezifikation über den durch ANTLR generierten Parser verarbeitet.

Zur Unterstützung von Import-Deklarationen wird die Grammatik des USE-Parsers um eine entsprechende neue Produktionsregel erweitert. Diese ist in Listing 4.1 in erweiterter Backus-Naur-Form dargestellt:

```
importStatement ::= "import" importClause "from" artifact
importClause ::= elementIdent
                | "*"
                | "{" elementIdent ("," elementIdent)* "}"
elementIdent ::= [id "#"] id
artifact ::= STRING
```

Listing 4.1: Grammatik des Import-Statements in EBNF

Mithilfe des Schlüsselwortes *import* wird das Import-Statement eingeleitet, darauf folgt die Importklausel, mit der die zu implementierenden Elemente in verschiedener Form spezifiziert werden können. Diese Formen umfassen die Angabe eines einzelnen Bezeichners *(elementIdent)* oder mehrerer Bezeichner, die durch geschweifte Klammern gruppiert werden. Die Parser-Regel *elementIdent* beschreibt den Bezeichner des Elements, wobei dieser entweder ein einfacher Bezeichner *id* ist, oder sich aus zwei Bezeichnern zusammensetzt, die vom Token # getrennt werden. Bei der einfachen Variante ist nur der Name des referenzierten Elements zu spezifizieren, wohingegen bei der zusammengesetzten Variante zusätzlich der Name des Modells angegeben werden muss, in dem sich das Element befindet. Der Token *id* steht hier für eine Kurzbezeichnung, die sich in den *EBNF*-Beschreibungen im USE-Umfeld etabliert hat und die Menge an validen Bezeichnern repräsentiert. Alternativ zur expliziten Definition der referenzierten Elemente kann auch ein Wildcard-Token * angegeben werden, wodurch signalisiert wird, dass alle relevanten Bestandteile des referenzierten Modells importiert werden sollen. Es folgt die Angabe des Schlüsselwortes *from* und des Artefakts in dem sich die Modellelemente befinden, ausgedrückt durch eine Zeichenkette.

In Listing 4.2 ist die Regel *model* zu sehen, die den Startpunkt für die Verarbeitung einer USE-Spezifikation bildet. Die Produktionsregel für ein Import-Statement wird in der Parserhierarchie so angeordnet, dass sie zwangsläufig am Anfang einer USE-Spezifikation stehen muss und sowohl gar nicht als auch mehrmals vorhanden sein kann.

```
model ::= { importStatement } "model" id { modelElement }
```

Listing 4.2: Integration der importStatement-Regel in die model-Regel (EBNF)

Die Form des Import-Statements wurde bewusst so gestaltet, dass sie der Syntax gängiger Programmiersprachen ähnelt, insbesondere ist eine starke Nähe zur Notation in JavaS-

cript erkennbar[3]. Diese Entscheidung bringt mehrere Vorteile mit sich. Zum einen tragen die gewählten Schlüsselwörter zueiner natürlichen und selbsterklärenden Ausdrucksweise bei. So signalisiert das Schlüsselwort *import*, dass bestimmte Elemente eingebunden werden sollen, während *from* die Quelle dieser Elemente angibt. Zum anderen ist zu erwarten, dass die große Mehrheit der Nutzer*innen von USE einen Hintergrund in der Softwareentwicklung oder verwandten Bereichen hat und dadurch mit der Form der Notation grundsätzlich vertraut ist. Dies fördert eine intuitive Nutzung der Import-Funktionalität und erleichtert ihre Erlernbarkeit. Darüber hinaus integriert sich das Import-Statement nathlos in die bestehende Notation von USE-Spezifikationen ein, ohne stilistische Brüche zu erzeugen.

Die unterschiedlichen Ausprägungen der Regel *importClause* decken die verschiedenen Anwendungsfälle im Umgang mit Importen ab. Nutzer*innen können zum einen gezielt steuern, ob ein einzelnes oder mehrere Elemente importiert werden sollen. Die Verwendung geschweifter Klammern beim Import mehrerer Elemente erfüllt hier zwar keinen technisch notwendigen Zweck, tragen aber zu einer besseren Strukturierung und Lesbarkeit bei. Darüber hinaus wird hier die externe Eigenschaft der Wildcard * umgesetzt, die es erlaubt, alle in einer externen Ressource definierten Elemente mit einem Ausdruck zu importieren. Diese Entscheidung ist dadurch zu begründen, dass sich aus der Verwendung auch für diese Implementierung der wesentliche Nutzen ergibt, der von Jansen et al. (2024)[11] festgestellt wird. Zwar lässt sich zu diesem Zeitpunkt nicht genau feststellen, wie genau Nutzer*innen mit der Import-Funktionlität in Zukunft interagieren und wie viele Elemente generell importiert werden. Trotzdem trägt das Konstrukt dazu bei, dass sich Redundanz im Modell verringern und die Übersichtlichekeit erhöhen lassen. Außerdem verringert sich der manuelle Aufwand beim Erstellen von Modellen, was zu einer höheren Benutzerfreundlichkeit beiträgt.

Die Eigenschaft des *Aliasing* ist in der Grammatik mit Einschränkungen berücksichtigt. Zwar ist es nicht möglich, importierten Elementen vollständig frei wählbare Namen zuzuweisen, wie es in vielen anderen Sprachen üblich ist. Stattdessen kann jedoch der ursprüngliche Bezeichner des Elements durch die Qualifikation mit dem Namen des Modells, aus dem es importiert wird, anders identifiziert werden. Dadurch lassen sich Namenskonflikte auflösen etwa wenn zwei Modelle Elemente mit identischen Bezeichnern definieren. Bei der Verwendung im Modell kann dann das gewünschte Element referenziert werden, indem der qualifizierte Name (z.B. *Modellname#Elementname*) angegeben wird. Dieses eingeschränkte Aliasing stellt eine praktibable Lösung für potenzielle Namenskonflikte dar, und ist in der aktuellen Implementierung auf systembedingte Einschränkungen

zurückzuführen, auf die in den folgenden Abschnitten näher eingegangen wird. Ein vollständiger Verzicht auf diese Funktionalität hätte die Flexibilität in der Modellierung deutlich eingeschränkt.

Die Einführung von Sichtbarkeiten ist in der Syntax des Import-Statements bewusst nicht umgesetzt. In der Betrachtung des Systems und seiner Funktionalitäten ist es nicht ersichtlich, inwiefern eine Umsetzung dieser Eigenschaft sinnvoll wäre. Diese Entscheidung deckt sich mit der Einschätzung von Jansen et al. (2024)[11], dass sich im Kontext von Modellierungssprachen als wenig vorteilhaft erweisen.

In Hinblick auf die Eigenschaften des Importziels ist festzustellen, dass hier sowohl das Artefakt als Container referenziert wird, als auch die Elemente, die das Artefakt beinhaltet. Dadurch lassen sich gezielt ausschließlich die gewünschten Elemente des Modells referenzieren, wodurch ein hohes Maß an Kontrolle über die Modellierung gewährleistet wird.

4.1.2 Abstrakter Syntaxbaum

Bei der Verarbeitung einer USE-Spezifikation wird der Quelltext durch den USE-Parser zur internen Verarbeitung in einen abstrakten SyntaxbaumAST übersetzt. Unter Anwendung der neuen Produktionsregel für Import-Statements wird mit Hilfe von *semantischen Aktionen* des Parsers eine neue Art Knoten des abstrakten Syntaxbaums erzeugt.

Die zentrale Java-Klasse ist hierfür `ASTImportStatement`. Sie speichert sämtliche Informationen, die beim Parsen eines Import-Statements gesammelt werden. Dies umfasst zum einen die Bezeichner der importierten Elemente, eine Referenz darauf, ob das Wildcard-Attribut verwendet wurde, und zusätzlich die Zeichenkette, welches die Struktur identifiziert, aus der die Elemente importiert werden.

Dieser Knoten ist in der Hierarchie des abstrakten Syntaxbaums direkt unter dem Wurzelknoten `ASTModel` neben den anderen Bestandteile einer USE-Spezifikation angeordnet. (siehe Listing 4.1)

Die AST-Knoten eines Modells übernehmen dabei nicht nur die Rolle eines Strukturknotens im Syntaxbaum, sondern stellen auch Methoden bereit, die die Übersetzung in ihre endgültige Repräsentation, den sogenannten Metamodell-Klassen ermöglichen. Die Klasse `ASTImportStatement` übernimmt dabei die zentrale Aufgabe, externe Modelle

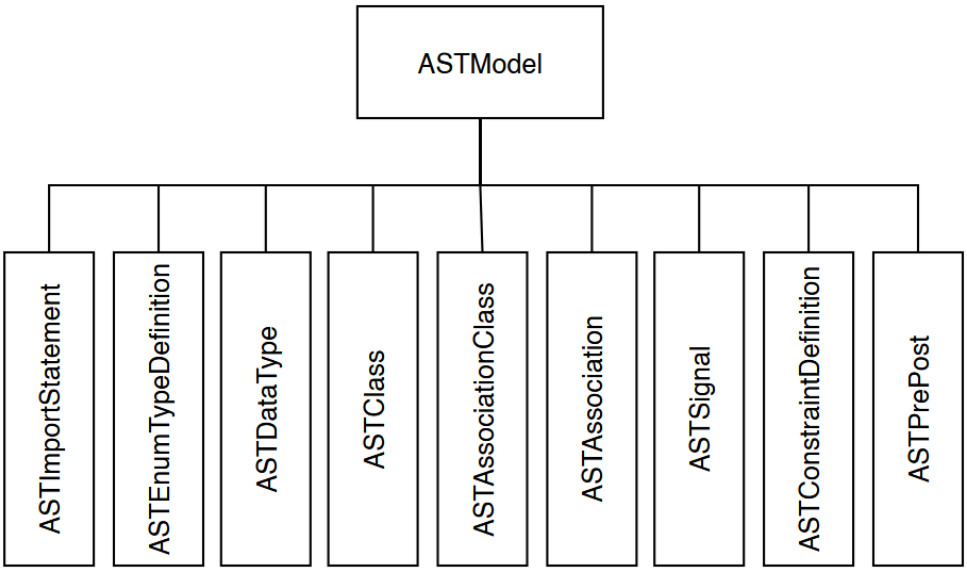

Abbildung 4.1: Der abstrakte Syntaxbaum eines Modells in USE.

einzubinden und deren Elemente dem aktuellen Modell verfügbar zu machen. Die Details dieses Vorgangs werden im folgenden Abschnitt beschrieben.

4.1.3 Verarbeitung von Import-Deklarationen

Die Konzeption des Algorithmus zum Einbinden externer Modelle enthält einige zentrale Entwurfsentscheidungen, die im Folgenen näher erläutert werden. In Listing 4.2 wird der Ablauf der Verarbeitung eines Import-Statements in verkürztem Umfang dargestellt.

Pfadauflösung

Die Ausführung eines Import-Statements in bedeutet in USE den Zugriff auf eine weitere Spezifikationsdatei. Eine besondere Herausforderung besteht darin, dass die Funktionalität des Öffnens bisher ausschließlich auf die Hauptspezifikation beschränkt war, wobei der physische Speicherort beliebig sein konnte. Anders als in vielen Programmiersprachen gab es in USE nicht die Notwendigkeit einer Paketstruktur, welche eine virtuelle Abbildung des Dateisystems darstellt. Eine Spezifikationsdatei enthält in der Regel die

vollständige Modellbeschreibung und war bisland unabhängig von einer festgelegten Verzeichnisstruktur. Dies hat wesentliche Konsequenzen für die Auflösung von Pfaden beim Import externer Spezifikationen.

Ein zentrales Problem dabei ist, dass sich auch importierte Spezifikationen auf weitere Modelle beziehen können, wodurch keine eindeutige Wurzelstruktur im Dateisystem festgelegt werden kann. Daraus ergibt sich, dass das Import-Statement in Abwesenheit einer Paketstruktur zwingend eine Pfadangabe zur Zieldatei enthalten muss, damit während der Kompilierung eine korrekte Auflösung erfolgen kann. Diese Notwendigkeit motiviert auch die klare Trennung zwischen den zu importierenden Bezeichnern und der Angabe des Artefakts, denn eine semantische Verbindung zwischen diesen beiden Teilen ergibt sich nur in Sprachen mit Paket- oder Namensraumkonzepten.

Die Pfadauflösung unterstützt sowohl absolute als auch relative Pfadangaben. Aufgrund der fehlenden übergeordneten Struktur ist eine absolute Pfadangabe allerdings wenig portabel, da sie auf ein konkretes Dateisystem verweist und somit nicht zwischen unterschiedlichen Umgebungen übertragbar ist. Relative Pfade bieten hier die deutlich praktikablere Lösung, da sie in Relation zur Lage der importierten Datei aufgelöst werden und sich so beispielsweise Modelle in einem gemeinsamen Projektverzeichnis organisieren lassen. Allerdings muss für eine korrekte Funktionsweise diese relative Lage zu einander beibehalten werden.

Kompilierung

Nach der erfolgreichen Pfadauflösung kann die externe Spezifikationsdatei geöffnet werden. Infolgedessen, dass mit der Einführung des Importmechanismus mehrere Modelle während der Kompilierung erzeugt werden können, wird eine globale Kontextstruktur eingeführt, die unter anderem die bereits kompilierten Modelle in einer Datenstruktur sammelt. So kann überprüft werden, ob die geöffnete Spezifikation bereits kompiliert wurde, und die Mehrfachkompilierung vermieden werden.

Wird ein Modell importiert, kann es seinerseits weitere Modelle importieren, wodurch eine rekursive Kompilierung ausgelöst wird. Auf diese Weise entstehen transitive Abhängigkeiten zwischen mehreren Modellen. Für den Kompilierungsprozess stellt dies zunächst kein Problem dar, da die importierten Modelle alle relevanten Informationen in Form von konkreter Objekte von Modellbestandteilen bereitstellen. Diese enthalten sämtliche semantisch notwendigen Daten, um in übergeordneten Modellen korrekt eingebunden zu

werden. Dadurch müssen sich Modellierer*innen während der Kompilierung nicht explizit mit transitiven Importabhängigkeiten auseinandersetzen.

Eine direkte Folge der transitiven Abhängigkeiten zwischen Modellen ist die Möglichkeit zyklischer Importe. Auch wenn es laut Jansen et al. (2024)[11] denkbare Anwendungsfälle gibt, in denen zyklische Modellbeziehungen sinnvoll sein können, wird in der vorliegenden Implementierung bewusst darauf verzichtet, solche Zyklen zuzulassen. Ein eleganterer Umgang mit diesen Abhängigkeiten zur Kompilierzeit wäre grundsätzlich möglich, wird allerdings angesichts in Abwägung zwischen dem technischen Aufwand und dem eher geringen erwartbaren Nutzen nicht umgesetzt.

Nach der Kompilierung wird das erzeugte Modell in ein neues Objekt des Metamodells überführt, das ein importiertes Modell repräsentiert, und anschließend dem importierenden Modell als Attribut hinzugefügt. Dieser Schritt ist zwar für die erfolgreiche Kompilierung nicht zwingend erforderlich, jedoch essenziell für die korrekte Funktionalität der kompilierten Modelle zur Laufzeit. Der Zusammenhang wird in Abschnitt 4.1.4 näher erläutert.

Zusätzlich werden verschiedene Prüfungen zur semantischen Validierung der Importklausel durchgeführt. So wird bei explizit angegebenen Elementen überprüft, ob diese tatsächlich im Modell vorhanden sind. Ebenso wird bei der Verwendung von qualifizierten Elementnamen kontrolliert, ob der angegebene Modellname mit dem tatsächlichen Modellnamen übereinstimmt. Diese Validierungen sollen Fehler in der Modellierung des Import-Statements, etwa falsche Benennungen von Elementen explizit sichtbar machen, anstatt sie zu ignorieren und dadurch unerwartetes Verhalten zu verursachen.

Integration in Symboltabelle

Während der Kompilierung einer einzelnen Spezifikation wird ein Objekt der Klasse `Context` mitgeführt, das zentrale Informationen enthält, die von den AST-Knoten genutzt werden, um die entsprechenden *Metaklassen*-Objekte zu erzeugen. Dieses Kontextobjekt umfasst unter anderem eine Symboltabelle, in der alle komplexen Typen, also alle Modellelemente, die als Werte eines Attributs zulässig sind, verwaltet werden.

Da das Kontextobjekt modellbezogene Informationen enthält, wird es bei der Kompilierung importierter Modelle nicht wiederverwendet. Stattdessen wird für jedes importierte

Modell ein eigenes Kontextobjekt erzeugt. Im Rahmen der Verarbeitung von Import-Statements wird dieses Kontextobjekt jedoch verfügbar gemacht, sodass die darin enthaltenen Elemente insbesondere die importierten Classifier in die Symboltabelle des importierenden Modells aufgenommen werden können. Dadurch ist es möglich, diese importierten Typen im übergeordneten Modell zu referenzieren, etwa als Attributtypen.

Zusätzlich dient die Symboltabelle dazu, potenzielle Namenskonflikte beim Import von Elementen zu erkennen. Durch die gezielte Bereitstellung des Kontextobjekts zur Kompilierungszeit wird sichergestellt, dass innerhalb eines Modells und seiner direkten Importe keine Namenskonflikte auftreten können. Für transitiv importierte Modelle kann eine solche Eindeutigkeit der Bezeichner allerdings nicht garantiert werden - und wäre auch nicht sinnvoll, da dies bedeuten würde, dass alle Namen modellübergreifend eindeutig sein müssten.

Diese Einschränkung hat Auswirkungen auf den Umgang mit transitiven Importen: Sollen auch deren Inhalte zur Laufzeit instanziierbar sein, so muss ein Mechanismus vorhanden sein, der eine eindeutige Zuordnung von Bezeichnern trotz möglicher Namensüberschneidungen erlaubt.

4.1.4 Metamodell

Neben der Bereitstellung von importierten Elementen in der Symboltabelle für die Kompilierung muss der Importmechanismus auch eine konkrete Repräsentation der importierten Modelle zur Laufzeit zur Verfügung stellen. Dazu wurde die Klasse `MImportedModel` als Teil des Metamodells von USE konzipiert. Sie kapselt ein kompiliertes externes Modell (`MModel`) und sorgt dafür, dass nur die durch das Import-Statement erlaubten Elemente im importierten Modell verfügbar sind. Diese Semantik kommt insbesondere in der grafischen Benutzeroberfläche und der interaktiven Shell zur Anwendung, etwa beim Erzeugen von Objekten, dem Setzen von Attributen oder der Simulation von Systemzuständen.

Neben der Referenz auf das importierte Modell enthält die Klasse alle relevanten Informationen aus dem Import-Statement, sodass die Semantik des Imports korrekt abgebildet werden kann. Demnach werden nur diejenigen Classifier nach außen sichtbar gemacht, die entweder per Wildcard oder explizit importiert wurden. Ebenso gelten nur die Invarianten sowie Vor- und Nachbedingungen als verfügbar, die diese Classifier betreffen. Für

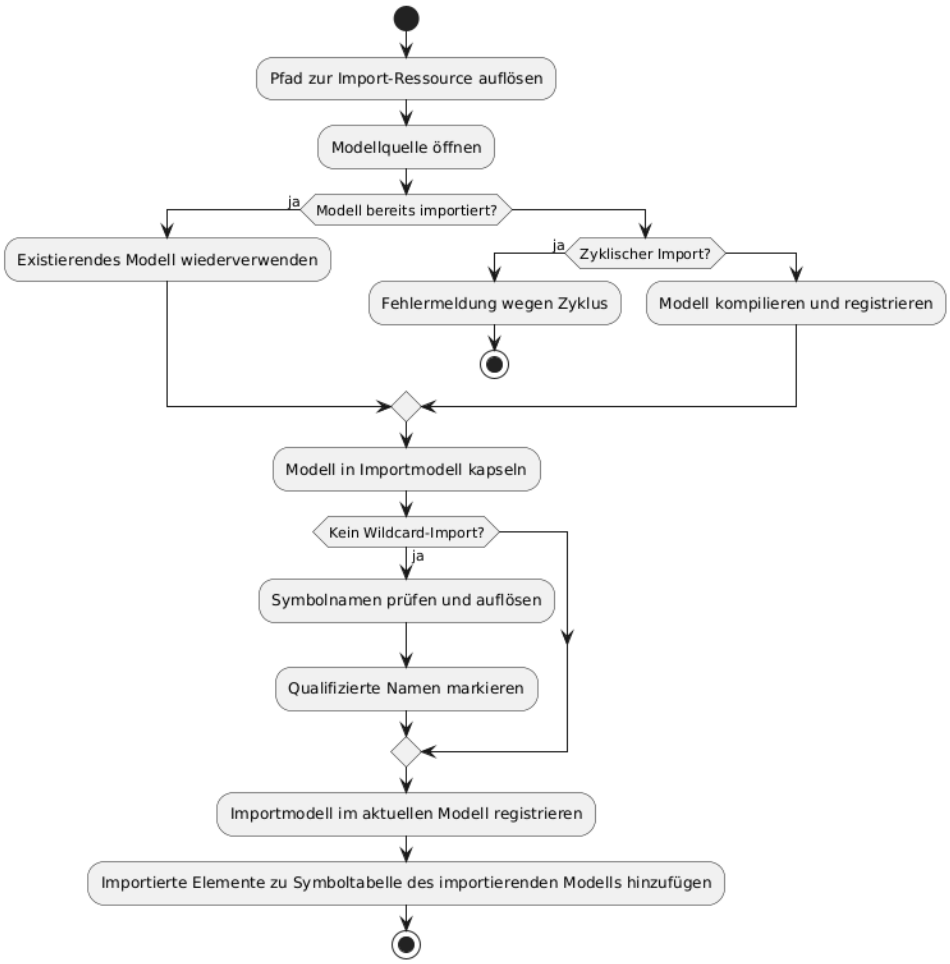

Abbildung 4.2: Ablauf des Imports eines Modells in USE.

komplexe Elemente wie Assoziationen, Assoziationsklassen und Invarianten wird zusätzlich geprüft, ob auch alle beteiligten Klassen importiert wurden. Nur wenn dies der Fall ist, sind diese Elemente zur Laufzeit tatsächlich verfügbar, da sie sonst weder erzeugt noch ihre Gültigkeit als Einschränkung im System berücksichtigt werden können.

Die Zugriffsmethoden (*Getter*) orientieren sich im Wesentlichen an den entsprechenden Methoden der `MModel`-Klasse. Dies ist vor dem Hintergrund zu erklären, wie der Zugriff auf Modellelemente in der Benutzeroberfläche organisiert ist. Die Methode zur Kompilierung von Spezifikationen gibt immer genau ein Modell zurück. Dieses Hauptmodell entspricht dem Modell, das über die Öffnen-Funktionalität der GUI oder der Shell geladen wird, und stellt den zentralen Einstiegspunkt für alle Zugriffe auf Modellelemente

und deren Funktionalitäten dar. Das bedeutet beispielsweise, dass beim Erzeugen eines neuen Objekts einer Klasse immer zuerst das Hauptmodell aufgerufen wird, um den Klassennamen aufzulösen.

Um die importierten Modelle ebenfalls zugänglich zu machen, werden die `MImported-Model` -Objekte, die von einem Modell importiert werden, als Klassenattribut hinzugefügt. Dadurch ist ein direkter Zugriff auf die Importe möglich. Elemente werden zuerst im Hauptmodell gesucht. Falls sie dort nicht gefunden werden, erfolgt die Suche in den importierten Modellen. Die Umsetzung in dieser Form hat den Vorteil, dass sie ohne große Veränderungen an den Aufrufen in der GUI und der Shell auskommen. Die wesentlichen Anpassungen befinden sich in den Methoden von `MModel` und `MImportedModel`. Eine Ausnahme bilden die Methoden, die Sammlungen von Modellelementen zurückgeben. Hier muss unterschieden werden, ob vom Aufrufer alle Elemente des Modells mit den importierten Elementen gewünscht werden, oder ob alle tatsächlich lokal zum Modell gehörenden Elemente gemeint sind. Zu diesem Zweck werden in `MModel` Methoden für beide Fälle zur Verfügung gestellt. Ein Anwendungsfall davon wäre der Modellbrowser, der zur Anzeige der geöffneten Modelle dient (siehe Listing 4.3). Hier soll gezielt ein Überblick darüber gegeben werden, aus welchen Modellen die jeweiligen Elemente stammen. Andersherum sind für Informationen über den Systemzustand auch die importierten Elemente relevant.

Eine Herausforderung ist an dieser Stelle der Umgang mit transitiv importierten Elementen. Für diese Elemente ist keine Einzigartigkeit der Bezeichner gegeben. Daher werden für die Aufrufe bestimmter Elemente anhand ihres einfachen Namens nur das Hauptmodell und seine direkt importierten Modelle durchsucht. Somit werden die transitiv importierten Elemente in der aktuellen Umsetzung nicht unmittelbar als Teil des Systems betrachtet und beispielsweise auch nicht im Modellbrowser angezeigt. Man könnte allerdings argumentieren, dass es doch möglich sein sollte, wenn ein Element importiert wird und dieses Element Attribute eines importierten Typs hat, dass für dieses Attribut auch ein Wert instanziierbar sein sollte. Daher wird diese Funktionalität durch Methoden in `MImportedModel` zur qualifizierten Namensauflösung realisiert. Mit ihnen lassen sich Namen der Form `ModelName#ClassName` auflösen, indem eine zusätzliche Qualifizierung mithilfe des Modellnamens durchgeführt wird. Die Umsetzung dieser Funktionalität erfordert kleine Anpassungen an den Parsern für OCL-Ausdrücke und Shell-Befehle, damit diese Form der Bezeichner erkannt wird.

Die Organisation der Modelle verfolgt den Ansatz der losen Kopplung[11], statt dass Elemente direkt in das Hauptmodell kopiert werden, werden die kompilierten Modellstrukturen intakt behalten und importierte Elemente über diese Struktur referenziert und zur Verfügung gestellt.

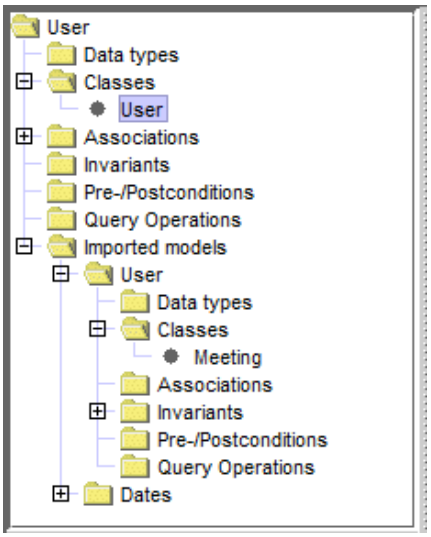

Abbildung 4.3: Der Modellbrowser mit importierten Modellen.

4.2 Tests

Die funktionale Korrektheit des Importmechanismus wird über eine robuste Teststrategie sichergestellt. Neben Unit Tests für isolierte Einheiten des Systems, wird der Compiler-Test und der Integrationstest der USE-Shell um weitere Testfälle erweitert.

4.2.1 Unit Tests

Im Rahmen der Unit Tests wird insbesondere die Metaklasse `MImportedModel` vor allem dahingehend getestet, ob der Zugriff auf importierte Elemente wie erwartet funktioniert. Dabei werden unterschiedliche Szenarien berücksichtigt.

Bei Wildcard-Importen wird getestet, ob sämtliche Elemente des importierten Modells dem Hauptmodell verfügbar gemacht werden. Im Gegensatz dazu wird bei Importen mit spezifizierten Elementen geprüft, dass ausschließlich auf die angegebenen Elemente

zugegriffen werden kann. Zudem wird überprüft, ob Namenskonflikte durch die Verwendung qualifizierter Namen korrekt aufgelöst werden. Auch transitive Importbeziehungen werden hinsichtlich ihrer korrekten Auflösung getestet.

Die Tests decken dabei alle Modellelemente ab, für die gemäß der Zugriffsregeln in `MImportedModel` ein bestimmtes Verhalten erwartet wird.

4.2.2 Compiler Tests

Der Compiler-Test in USE bietet die Möglichkeit, Testspezifikationen zu definieren, die kompiliert und auf die erwartete Ausgabe geprüft werden können. Dabei können sowohl positive als auch negative Testfälle erstellt werden. Entweder handelt es sich um syntaktisch korrekte Spezifikationen, die fehlerfrei kompiliert werden sollen, oder um Spezifikationen mit Fehlern, bei denen eine bestimmte Fehlermeldung erwartet wird, die mit der tatsächlichen Ausgabe übereinstimmen muss.

In Hinblick auf die Implementierung des Importmechanismus ist die Überprüfung der Compiler-Funktionalität von zentraler Bedeutung, da der Import externer Spezifikationen den Kompilierungsprozess erheblich komplexer macht. Außerdem können neue Arten von Fehlern in der Kompilierung entstehen. Daher ist es wichtig, dass die im Importmechanismus vorgesehenen Fehlermeldungen unter den korrekten Bedingungen ausgelöst und entsprechend zurückgegeben werden.

In diesem Zusammenhang ist der Compiler-Test sehr hilfreich, um semantische Fehler in der Deklaration von Importen sowie deren Fehlerbehandlung zu überprüfen. Dazu zählen unter anderem referenzierte Elemente, die im importierten Modell nicht vorhanden sind, Namenskonflikte durch den Import von Elementen mit identischen Bezeichnern wie lokale Elemente, oder fehlerhaft angegebene Artefaktpfade. Auch die Erkennung von zyklischen Importen sowie die Überprüfung der Qualifizierung von Elementen können getestet werden, insbesondere die Übereinstimmung zwischen dem spezifizierten und dem tatsächlichen Namen des zugehörigen Modells.

Dies umfasst unter anderem spezifizierte Elemente, die sich nicht im importierten Modell befinden, Namenskonflikte beim Import eines Elements mit dem gleichen Bezeichner wie ein lokales Element, oder falsch spezifizierte Artefaktpfade. Außerdem kann die Zyklendetektion überprüft werdenn sowie die für die Qualifizierung von Elementen, dass der Modellname mit dem betreffenden Modell übereinstimmt.

4.2.3 Integrationstests

Der Integrationstest für die USE-Shell ermöglicht die Überprüfung der Interaktion mit der Kommandozeilenoberfläche. Dabei können Testspezifikationen definiert werden, die sowohl konkrete Systemabläufe als auch die erwarteten Ausgaben auf der Konsole festlegen. Diese Abläufe werden in Form von Skriptdateien umgesetzt, die eine Abfolge von Shell-Befehlen enthalten.

Im Rahmen der Tests werden verschiedene Szenarien abgedeckt um das korrekte Laufzeitverhalten von Modellen mit Importen zu sicherzustellen. Dies umfasst beispielsweise Tests, die überprüfen, ob sich Objekte aus importierten Modellbestandteilen wie erwartet erzeugen lassen und ob für Objektattribute Werte für importierte Typen gesetzt werden können. Ein weiterer Aspekt betrifft die Erweiterung der Eingabegrammatik für Shell-Befehle und OCL-Ausdrücke, welche die Verwendung qualifizierter Bezeichner durch Angabe des Modellnamens ermöglicht. Mit Hilfe des Integrationstests kann verifiziert werden, dass diese Bezeichner korrekt aufgelöst und den entsprechenden Modellelementen zugeordnet werden.

4.3 Dokumentation

Im USE-Projekt wird ein Handbuch gepflegt, welches den Funktionsumfang und die Bedienung erläutert. Als Teil der Umsetzung des Importmechanismus muss auch die Dokumentation der Funktionalität und seine Bedienung berücksichtig werden. Daher ergänzt die Implementierung auch das Handbuch um ein Kapitel zur Spezifikation von Modellen mit Importen sowie den Umgang mit importierten Elementen zur Lauftzeit. Außerdem werden Regeln und Einschränkungen beschrieben, die für die Implementierung gelten. (siehe Appendix A.1)

5 Evaluation

Im Folgenden wird die Implementierung hinsichtlich der Umsetzung der funktionalen Anforderungen unter Berücksichtigung der nichtfunktionalen Anforderungen ausgewertet.

5.1 Umsetzung der funktionalen Anforderungen

5.1.1 Sprachdefinition und Kompilierungslogik

Zur Erfüllung der funktionalen Anforderung *FR-01* wurde die USE-Grammatik um eine Regel zur Deklaration von Import-Statements erweitert. Diese Regel führt ein neues Sprachelement ein, welches die Grundlage für die Einbindung externer Modelle bildet. Dabei berücksichtigt die Konzeption sowohl funktionale als auch ergonomische Aspekte. Funktional wird eine Trennung zwischen den zu importierenden Elementen und dem referenzierten Artefakt vorgenommen, was die für die Verarbeitung relevante semantische Trennung zwischen der externen Modellresource und den zu importierenden Bezeichnern gewährleistet. Vor dem Hintergrund einer intuitiven Erlernbarkeit orientiert sich die Syntax an gängigen Programmiersprachen und ermöglicht durch Merkmale wie das Wildcard-Konstrukt eine flexible und benutzerfreundliche Bedienung.

Durch die Integration des Import-Statements in den Parsing-Prozess einer Modellspezifikation und die Erzeugung eines entsprechenden Knotens im abstrakten Syntaxbaum, wird sichergestellt, dass importierte Modellelemente im weiteren Ablauf der Kompilierung berücksichtigt und in das importierende Modell integriert werden können (FR-02). Die Klasse `ASTImportStatement` dient hier als Brücke zwischen dem grammatischen Konstrukt und seiner semantischen Umsetzung.

Der Importmechanismus wurde so umgesetzt, dass während des Kompilierungsprozesses alle importierten Classifier über die betreffende Symboltabelle dem importierenden

Modell zur Verfügung gestellt werden (FR-03). Dies ermöglicht beispielsweise die Referenzierung externer Typen bei der Erzeugung von Attributtypen. Ergänzend stellt die neue Metaklasse `MImportedModel` sicher, dass auch zugehörige Invarianten sowie Vor- und Nachbedingungen zur Laufzeit abrufbar und validierbar sind (FR-04).

Die korrekte Umsetzung dieser Anforderungen wird einerseits durch spezifizierte Testfälle im Compiler-Test validiert, welche verschiedene Szenarien für Modelle mit Importen durchlaufen. Andererseits wird über den Unit-Test von `MImportedModel` sichergestellt, dass die gemäß der Importsemantik importierbaren Classifier und Integritätsbedingungen zur Laufzeit zur Verfügung stehen.

5.1.2 Grafische Benutzeroberfläche (GUI) und ihre Funktionalitäten

Nach der Kompilierung wird wie bisher das zur geöffneten Spezifikation gehörende Modell der Benutzeroberfläche zur Verfügung gestellt. Dieses Modell enthält nun auch Referenzen auf seine importierten Modelle, sodass sie für die GUI zugänglich sind. Mithilfe einer Anpassung in der Konstruktion des Modellbrowsers werden sowohl für das Hauptmodell, als auch für seine importierten Modelle Repräsentationen erzeugt, sodass sie sichtbar und interagierbar werden (FR-05).

Die Erweiterungen an der Metaklasse `MModel` sowie die Einführung der neuen Metaklasse `MImportedModel` sorgen dafür, dass sich importierte Elemente aus Sicht der GUI-Anwendung genauso verhalten wie Elemente des Hauptmodells. Dadurch lassen sich importierte Klassen und Assoziationen instanziieren (FR-06) und Attributwerte für importierte Datentypen und Enums setzen (FR-07). Anpassungen an der Programmlogik der GUI waren hierfür nicht erforderlich. Um Attributwerte für transitiv importierte Typen zu setzen, ist jedoch eine Qualifikation der Konstruktor-Operation mit dem Modellnamen erforderlich, um Namenskonflikte zu vermeiden.

Auch OCL-Auswertungen auf importierten Elementen funktionieren wie gewohnt über das Hauptmodell (FR-08). Für den Zugriff auf transitiv importierte Elemente muss ebenfalls die qualifizierte Schreibweise *ModelName#ElementName* verwendet werden. Hierfür wurden Anpassungen am OCL-Parser vorgenommen, um Bezeichner dieser Art zu verarbeiten.

Zur Erweiterung der Visualisierung des Systemzustands wurden in `MModel` neue Methoden implementiert, die lokale und importierte Elemente einer gemeinsamen Metaklasse

gesammelt zurückgeben. Dies ermöglicht die konsistente Anzeige auch importierter Inhalte in allen betroffenen GUI-Komponenten (FR-09).

5.1.3 USE-Shell und ihre ausführbaren Befehle

Das Öffnen einer Spezifikation mit Importen über die USE-Shell wird unterstützt, da die zugrundeliegende Programmlogik derjenigen entspricht, die auch von der GUI aufgerufen wird (FR-10).

Die Shell-Befehle zum Instanziieren von Objekten und Assoziationen sowie zum Setzen von Attributwerten funktionieren ebenfalls wie gewohnt, da sie auf dieselbe Modellstruktur zurückgreifen, welche die Referenzierung importierter Elemente ermöglicht (FR-11). Ergänzend wurde der Shell-Parser erweitert, um qualifizierte Bezeichner zu erkennen. Dies ermöglicht die Referenzierung transitiv importierter Elemente sowie solcher, die zur Auflösung von Namenskonflikten mit qualifiziertem Namen importiert wurden.

Zur vollständigen Überprüfung des Systemzustands wurde die Klasse `MModel` um eine Methode ergänzt, die zusätzlich zu den lokalen auch die Invarianten importierter Elemente zurückliefert, sodass der Befehl `!check` alle relevanten Bedingungen berücksichtigt (FR-12).

Auch die Shell-Befehle `!openter` und `!opexit` funktionieren für importierte Elemente unverändert, da sich deren Referenzierung über das Hauptmodell abbildet.

Die korrekte Umsetzung der Anforderungen an die USE-Shell lässt sich über den Shell-Integrationstest überprüfen. Diesem Integrationstest wurden Testfälle hinzugefügt, die das Verhalten bei der Ausführung von Befehlen auf Modellen mit Importen überprüfen.

5.2 Berücksichtigung der nichtfunktionalen Anforderungen

Die Benutzerfreundlichkeit der neuen Funktionalität lässt sich an verschiedenen Stellen beobachten. Der Umgang mit dem entworfenen Import-Statement ist leicht erlernbar und wird von einer umfangreichen Dokumentation unterstützt. Darüber hinaus wird die technische Umsetzung des Kompilierens und Einbindens der importierten Modelle an vielen Stellen durch eine robuste Fehlerbehandlung begleitet, sodass für Nutzer*innen

erkennbar ist, warum Fehler auftreten. Importierte Modelle werden so in der Anwendung eingebunden, dass sich an der Nutzerinteraktion kaum etwas verändert.

Wartbarkeit wird bei der Umsetzung dahingehend erfüllt, dass die Kernfunktionalität im Wesentlichen in den neu erstellten Klassen `ASTImportStatement` und `MImportedModel` umgesetzt wird und somit klare Zuständigkeiten für die Funktionalitäten des Importmechanismus erkennbar sind. Außerdem trägt die umfangreiche Dokumentation der Klassen und Methoden zu einer klaren Verständlichkeit bei.

Um die Kompatibilität der Implementierung, beispielsweise mit dem Plugin-System von USE zu gewährleisten, wurden bestehende Methodensignaturen weitgehend beibehalten. So ist die Rückgabe der Methode zur Kompilierung von Spezifikationen nach wie vor ein Modell, obwohl während der Kompilierung mit Importen mehrere Modelle kompiliert werden. Dies liegt vor allem an der Strukturierung der Modelle und ihrer importierten Modelle. Außerdem hat diese Modellstruktur nach außen hin ein nahezu identisches Verhalten wie vorher, egal ob sie Importe enthält oder nicht. So ändert sich dadurch die Verwendung der Modelle kaum. Eine Anpassung, die die Kompatibilität einschränkt, ist die Erweiterung der Methodensignatur zur Kompilierung einer Spezifikation. Infolge der relativen Pfadauflösung ist es notwendig, den Pfad des importierenden Artefakts verfügbar zu machen. Die alte Methodensignatur wurde allerdings beibehalten, sie funktioniert nur nicht für Spezifikationen mit Importen.

6 Diskussion und Ausblick

In diesem Kapitel werden bestimmte Entwurfsentscheidungen und ihre Alternativen besprochen sowie ein Ausblick auf mögliche zukünftige Erweiterungen gegeben, die sich aus der Implementierung des Importmechanismus ergeben.

6.0.1 Modellstruktur

Eine zentrale Entwurfsentscheidung der Implementierung betrifft die Strukturierung der Modelle und wie diese aufeinander zugreifen können. Die gewählte Lösung ist eine rekursive Struktur, bei der ein Modell mehrere importierte Modelle haben kann und diese wiederum weitere Modelle importieren können. Mehrere Kriterien haben diese Entscheidung motiviert. Zum einen ist die Rückgabe der Methode zur Kompilierung von Modellen genau ein Modell und zum anderen sind alle Komponenten des Systems darauf ausgelegt, dass es ein Modell gibt, auf dem die Aufrufe durchgeführt werden. Das ist verständlich, da dies vor der Umsetzung des Importmechanismus auch immer der Fall war, schränkt aber die Menge an praktikablen Lösungen etwas ein. Ein Nachteil der gewählten Modellstruktur ist, dass die Suche nach Elementen nicht besonders effizient ist, da erst das Hauptmodell und dann iterativ alle importierten Modelle befragt werden müssen, ob sie das Element enthalten.

Eine Alternative zur rekursiven Struktur wäre eine Komponente zur zentralen Verwaltung aller Modelle. Diese hätte mehr Möglichkeiten, um die Suche nach Elementen möglichst effizient zu gestalten. Sie müsste aber potenziell auch eine komplexere Datenstruktur aufbauen, um die Importbeziehungen zueinander abzubilden. Außerdem könnte sie die Kompatibilität mit Blick auf externe Schnittstellen einschränken.

6.0.2 Transitive Importe

Ein bislang nicht abschließend geklärter Aspekt betrifft die Behandlung transitiver Importe dahingehend, inwiefern sie ebenfalls in ihrer Semantik als Teil des Systems und seines Zustands betrachtet werden müssen.

In der aktuellen Umsetzung werden transitive Importe vor allem berücksichtigt, um eine fehlerfreie Kompilierung zu gewährleisten. In der grafischen Benutzeroberfläche werden sie jedoch bislang nur eingeschränkt dargestellt, hier werden zum System nur die lokalen sowie die direkt importierten Elemente gezählt.

Ein nun mögliches Szenario ist, dass das Hauptmodell eine Klasse importiert, die Attribute besitzt, deren Werte einem importierten Datentyp entsprechen.. Die Klasse wird als Teil des Systems betrachtet und zur Bestimmung des Systemzustandes müssen die Instanzen dieser Klassen und ihr Zustand validiert werden. Zu dieser Zustandsbeschreibung müsste streng genommen auch die Validierung der Attributwerte für den transitiv importierten Datentyp zählen. Daher ist es denkbar, dass Modellierer*innen für diese Attribute einen Wert setzen wollen, um einen realistischen Systemzustand nachzubilden. In der Semantik von Java wäre es allerdings nicht möglich, für dieses Attribut einen Wert zu setzen, weil der Datentyp nicht direkt importiert wurde. Die Klasse wurde aber bewusst importiert, deshalb könnte man als Anwender*in fragen, warum sie nicht vollumfänglich nutzbar ist.

In der Implementierung wird mit diesen möglichen Anforderungen an die Modellierung umgegangen, indem ermöglicht wird, dass über eine qualifizierte Namensauflösung transitive Elemente referenziert werden können. Dadurch wird im Endeffekt explizit signalisiert, dass die Nutzung dieses transitiv importierten Typs bewusst geschieht.

Ein anderer möglicher Umgang mit dieser Problematik wäre den direkten Import des Datentyps in das Hauptmodell vorauszusetzen. Die Verantwortlichkeit läge hier also deutlich mehr bei dem oder der Modellierer*in, die gewünschte Semantik des Systemzustandes herzustellen. Dies bedeutet aber einen höheren Modellierungsaufwand.

Hierbei handelt es sich aber um eine Überlegung, inwiefern die theoretische Freiheit in der Modellierung durch die Umsetzung von Importen erweitert wird und welche neuen Anforderungen sich theoretisch daraus ergeben könnten. Ob dieser Anwendungsfall überhaupt besteht, muss sich im praktischen Umgang mit der Anwendung zeigen. Es

werden jedenfalls beide Ansätze von der Anwendung unterstützt, sollte sich ein Bedarf ergeben.

6.0.3 Namespaces

Eine Problematik, die sich in der Umsetzung des Importmechanismus ergeben hat, ist die Pfadauflösung beim Einbinden von externen Spezifikationen. Bisher wurde beim Öffnen einer Spezifikation immer genau ein Modell kompiliert und es gab keine Abhängigkeiten zu externen Spezifikationen und somit hatte der tatsächliche Ort auf dem Dateisystem keine Relevanz. Infolge der Import-Funktionalität gibt es nun aber eine logische Verbindung zwischen Modellen, die auch eine örtliche Relation auf dem Dateisystem voraussetzen kann.

Es wird sich also wahrscheinlich etablieren, zusammengehörende Spezifikationen, die einander referenzieren in einer Ordnerstruktur zusammenzufassen, die ihre Zusammengehörigkeit signalisiert, Dies könnte man als einen gemeinsamen Namensraum oder sogar eine Art Projektstruktur bezeichnen.

Eine mögliche Überlegung, die man in diesem Zusammenhang verfolgen könnte, ist, diese physische Struktur mit einer logischen Struktur auf der Modellierungsebene zu unterstützen. So könnte ein Packaging-System entwickelt werden, welches einen gemeinsamen Namensraum für Modelle beschreibt und Konventionen etabliert, die die örtliche Relation auf dem Dateisystem zueinander formalisieren und algorithmisch validieren. Infolgedessen könnte der Importmechanismus um eine Funktionalität zur Auflösung des Pfades über den gemeinsamen Namensraum erweitert werden. Dies würde die Implementierung robuster machen, eine modulare Strukturierung und die Portabilität einer Gruppierung von Modellsammlungen fördern.

Darüber hinaus könnte diese Struktur auch als Basis für eine neue Sicht auf ein System in USE verwendet werden, denn durch die Umsetzung des Importmechanismus könnte man den Begriff des Systems auch als eine Sammlung von Komponenten begreifen, die durch die Modelle repräsentiert werden. Daher wäre es denkbar, dass Modellierer*innen in Zukunft auch Systeme auf dieser Abstraktionsebene beschreiben möchten.

7 Fazit

Als Zielsetzung dieser Arbeit wurde die Erweiterung des USE-Projekts um einen Importmechanismus umgesetzt, der das Einbinden externer Modelle ermöglicht und so zur Erhöhung der Modularisierung und Wiederverwendbarkeit in der Modellierung beiträgt.

Dafür wurden zuerst die theoretischen Grundlagen etabliert, die zur Erfassung der Anforderungen und zum Entwurf der Implementierung notwendig waren. Es wurde die Anwendungsdomäne von USE analysiert, wobei die Modellierungssprachen UML und OCL sowie das System selbst und seine Funktionsweise erläutert wurden. Darunter fällt auch die Betrachtung des Parsergenerators ANTLR, mit dem in USE die Grammatik der Modellierungssprache definiert wird. Es wurde zudem das Konzept eines Importmechanismus vorgestellt und die relevanten Eigenschaften erläutert, die bei der Konzeption dieser Funktionalität berücksichtigt werden müssen.

Aus diesen Erkenntnissen konnten die funktionalen und nichtfunktionalen Anforderungen an eine erfolgreiche Umsetzung des Importmechanismus definiert werden. Es wurden hier unter anderem Anforderungen an die Erweiterung der Sprachdefinition und Kompilierungslogik sowie die GUI und die Shell identifiziert. Darüber hinaus wurden Benutzerfreundlichkeit, Wartbarkeit und Kompatibilität als zentrale Qualitätsziele definiert, die bei der Umsetzung beachtet werden sollten.

Auf dieser Basis wurde ein Entwurf der Funktionalität konzipiert und implementiert. Dabei wurde eine Grammatik für ein neues Sprachkonzept entworfen, das ermöglicht, bei der Spezifikation von Modellen ein Import-Statement zu formulieren, welches die Einbindung externer Modelle und ihrer Elemente beschreibt. Die Verarbeitung wurde auf der Ebene des Parsers und des Compilers umgesetzt und eine neue Metaklasse definiert, die eine Datenstruktur für importierte Modelle bereitstellt und zur Laufzeit den Zugriff auf importierte Elemente durch die GUI und die Shell ermöglicht.

USE verfügt jetzt also über einen Importmechanismus, der Modellierer*innen die Möglichkeit gibt, modulare Strukturen aufzubauen, die gemeinsames Verhalten definieren und

über mehrere Spezifikationen hinweg wiederverwendbar sind. Die Implementierung erfüllt dabei die definierten funktionalen Anforderungen. Es können auf der Modellierungsebene externe Modelle importiert und ihre Elemente als Bestandteile der eigenen Spezifikation verwendet werden. Nach der Kompilierung einer Spezifikation sind diese Elemente auf der Benutzeroberfläche sichtbar und für die Verwendung von GUI-Aktionen oder Shell-Befehlen verfügbar. Dadurch lassen sich Systemzustände simulieren, die Instanzen importierter Elemente einschließen und bei der Validierung von Integritätsbedingungen berücksichtigen. Die Korrektheit der Umsetzung wird durch ein solides Testkonzept sichergestellt, das die implementierten Funktionalitäten auf verschiedenen Ebenen durch relevante Testfälle validiert. Außerdem wird durch eine Erweiterung der Dokumentation für Nutzer*innen die Bedienung erläutert.

In einer kritischen Auseinandersetzung wurden bestimmte Entwurfsentscheidungen mit möglichen Alternativen gegenübergestellt. Dabei wird auf die rekursive Modellstruktur eingegangen, die im Entwurf konzipiert wurde. Als Alternative wurde eine zentrale Struktur aufgezeigt, die zwar einige potenzielle Vorteile bietet, im Rahmen der Integration in die bestehende Anwendung aber als nicht praktikabel bewertet wurde. Es werden Problematiken im Umgang mit transitiven Importen thematisiert und darauf hingewiesen, dass eine weitere Beobachtung erforderlich ist, um zu beurteilen, ob die Implementierung dem gewünschten Verhalten entspricht.

Abschließend wird ein Ausblick auf ein mögliches Konzept für Namespaces gegeben, welches auf der Import-Funktionalität aufbauen könnte und die neu entstandene Abhängigkeit zwischen Modellen systematisiert und das Konzept der Modularisierung weiter vorantreibt.

Literaturverzeichnis

[1] ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. In: *ISO/IEC/IEEE 25010:2011(E)* (2011), S. 1–33

[2] ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering. In: *ISO/IEC/IEEE 29148:2018(E)* (2018), S. 1–104

[3] ECMAScript® 2024 Language Specification (ECMA-262, 15th Edition) / Ecma International. Geneva, Switzerland, Juni 2024 (ECMA-262 15th). – Technical Report. – URL https://www.ecma-international.org/wp-content/uploads/ECMA-262_15th_edition_june_2024.pdf

[4] *USE - UML-Based Specification Environment*. 2025. – URL https://github.com/useocl/use. – GitHub repository; accessed 2025-06-14

[5] Community, Systems M.: *SysML v2 Pilot Implementation*. – URL https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation. – Accessed on June 11, 2025

[6] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckley, Alex ; Smith, Daniel ; Bierman, Gavin: *The Java Language Specification, Java SE 24 Edition*. Santa Clara, CA : Oracle, Februar 2025

[7] Group, Object M.: *Object Constraint Language (OCL), Version 2.4*. February 2014. – URL https://www.omg.org/spec/OCL/2.4. – Zugriffsdatum: 10.06.2025. – Formal Specification

[8] Group, Object M.: *XML Metadata Interchange (XMI), Version 2.5.1*. June 2015. – URL https://www.omg.org/spec/XMI/2.5.1. – Zugriffsdatum: 11.06.2025. – Formal Specification

[9] Group, Object M.: *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. December 2017. – URL https://www.omg.org/spec/UML/2.5.1. – Zugriffsdatum: 10.06.2025. – Formal Specification

[10] Group, Object M.: *OMG Systems Modeling Language (SysML), Version 1.6*. December 2019. – URL https://www.omg.org/spec/SysML/1.6. – Zugriffsdatum: 11.06.2025. – Formal Specification

[11] Jansen, Nico ; Rumpe, Bernhard ; Schmalzing, David: A Synopsis on Import Statements in Modeling Languages. In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. New York, NY, USA : Association for Computing Machinery, 2024 (MODELS Companion '24), S. 1161–1169. – URL https://doi.org/10.1145/3652620.3688347. – ISBN 9798400706226

[12] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Dezember, Nr. 12, S. 1053–1058. – URL https://doi.org/10.1145/361598.361623. – ISSN 0001-0782

[13] Parr, Terence: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. – ISBN 0978739256

A Anhang

A.1 Dokumentation

A.1.1 Importing Elements from External Models

In a USE specification, `import` statements can be used to include model elements (e.g., data types, classes) defined in external models. An `import` statement specifies both the elements to include and the source model file from which they are imported.

To avoid naming conflicts between identically named elements from different models, elements can be qualified using the notation `modelName#elementName`. This mechanism is currently implemented for classes, data types, and enumerations.

Syntax

```
<importstatement> ::= import ( <elementname> | '{' <elementname> { , <
    elementname> } '}' | * ) from <artifactpath>
<artifactpath>    ::= <stringliteral>
```

Examples

Importing single or multiple elements:

```
import Date from "Dates.use"
import { Apple, Banana } from "Fruits.use"
```

Importing all elements using the wildcard operator:

```
import * from "Shapes.use"
```

Importing qualified elements:

```
import { Dates#Date } from "Dates.use"
import { Date } from "OtherDates.use"
```

Example model:

The following example shows a `User` model that imports the data type `Date` from an external model located at `imports/Dates.use`, and uses it in the `User` class.

```
import Date from "imports/Dates.use"

model User

class User
attributes
    name: String
    birthday: Date
end
```

Rules and Restrictions

- `import` statements must appear at the beginning of a USE file, before the model declaration.
- Each element listed in an `import` statement must be defined in the referenced model file (except when using the wildcard operator).
- Imported element identifiers must be unique within the importing model.

- Circular imports (`Model A` → `Model B` → `Model A`) are not permitted.
- The referenced file path may be absolute or relative to the importing model file.
- Identifier uniqueness is only enforced between a model and its direct imports. Therefore, access to transitively imported elements (i.e., elements imported by imported models) is only possible via qualified identifiers (`modelName#elementName`).
- If elements are listed with a qualified name in an `import` statement, they must also be referenced using that qualified name.

Transitive Import Example

The following example shows a `User` model importing the class `Meeting` from the model `Meetings`, and the data type `Date` from the model `Dates`. The `Meetings` model itself imports the data type `Time` from another model `Time` and uses it in the `Meeting` class.

```
import { Meeting } from "Meetings.use"
import { Date } from "Dates.use"

model User

class User
attributes
    name: String
    birthday: Dates#Date
    num: Integer
end
```

```
import { Time } from "Time.use"

model Meetings

class Meeting
attributes
    title: String
    participants: Integer
    startTime: Time
    endTime: Time
end
```

In order to assign values to the `startTime` and `endTime` attributes of a `Meeting` instance, the constructor must be invoked using a qualified name with the model and a # separator:

```
use> !set meeting.startTime := Time#Time(12,12,12)
```

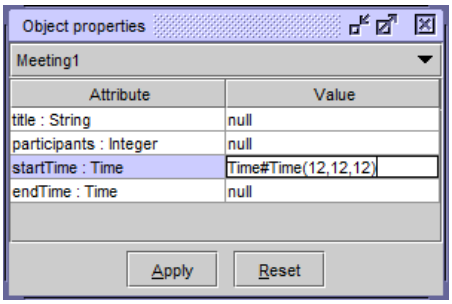

Abbildung A.1: Object properties dialog with transitive type usage

Erklärung zur selbständigen 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