Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

API-Nutzungsanalyse eines state-of-the-art Modeling Frameworks im modellgetriebenen Engineering - eine Fallstudie

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Pascal Tim Alexander Jochmann
geboren am: 29.06.1996
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Abstract

Das Eclipse-Plugin Henshin ist ein führendes Tool zur Transformation von Modellen und Metamodellen. Henshin ist eine state-of-the-art Modelltransformationssprache zur Übersetzung von formalisierten Texten in Modelle, und deren Transformation durch Änderungsoperationen, wie das Hinzufügen oder Löschen von Kanten und Knoten. Die technische Realisierung von Henshin basiert auf dem Eclipse Modeling Framework (EMF). Viele Klassen und Methoden, die EMF bereitstellt, werden von Henshin aber nicht benutzt. In dieser Arbeit führen wir eine quantitative Analyse des Henshin-Codes durch. Durch diese soll geklärt werden, ob eine Verbesserung von Henshins in Hinsicht auf Laufzeit- und Speicherkapazitäten oder Wartbarkeit und Erweiterung des Codes möglich wäre, oder ob dieses Vorhaben von vornherein nicht lohnenswert ist. Dazu untersuchen wir Henshins Basis, das Meta-Modell Ecore, und versuchen die relevanten Komponenten zu quantifizieren. Im Ergebnis konnten wir so nachweisen, dass ein großer Teil von Ecore für Henshin irrelevant ist und extrahiert werden könnte. In zukünftigen Arbeiten könnte somit eine Neuimplementation dieser Ecore-Teile vorgenommen werden, um Henshin möglicherweise qualitativ zu verbessern.

Inhaltsverzeichnis

1 Einführung

Durch die immer stärkere Ausbreitung von Software in immer mehr Lebensbereiche und das stetige Wachstum des Bedarfs von Industrien an immer besserer Software, wird diese immer komplexer [4]. Aufgrund dieser wachsenden Komplexität fällt es Entwicklern immer schwerer, alle Teile und Funktionen ihres Programmes im Auge zu behalten. Eine Abhilfe dazu ist schon lange das Nutzen von Modellen wie UML [9]. Im *model-driven Development* ziehen Entwickler den Fokus der Softwareeinwicklung noch stärker auf die Modellebene, indem aus Modellen automatisiert lauffähige Programme erzeugt werden [10]. Diese Herangehensweise bietet viele Vorteile, vor allem kann Software in leichterer Sprache beschrieben werden, ohne die konkreten Implementation selbst durchführen zu müssen. Der Entwickler kann sich mehr auf das Zusammenspiel der Komponenten und „das große Ganze“ fokussieren. Diese Funktion, ein Modell in Code umzuwandeln, wird durch ein *Modeling-Framework* ermöglicht [6, 4].

Das Eclipse Modeling Framework (EMF) [11, 16] ist ein mächtiges, in die Eclipse IDE integriertes Modeling-Framework. EMF bietet viele Funktionen, aber im Kern kann es zu gegebenen Modellen lauffähigen Java-Code erzeugen. Die Modelle für EMF werden in *Ecore* beschrieben. Ecore ist ein (Meta-)Modell und das Herzstück von EMF. In Ecore können Modelle gebaut werden, die entweder direkt Software beschreiben, oder andere Modelle darstellen. Eine tiefgreifendere Beschreibung von Ecore findet in Kapitel 3.1 statt.

Ein weiterer Bestandteil von EMF ist die Modelltransformationssprache Henshin [1, 19, 12, 3], die wie EMF auf Ecore basiert. EMF erlaubt einem nicht nur Modelle zu erzeugen, sondern auch diese zu transformieren. Dazu ist jedoch eine formale Sprache nötig und eine Software, die diese Sprache interpretiert und auf das Model anwendet. Genau diese Aufgabe übernimmt Henshin. Neben Henshin gibt es noch viele weitere Modelltransformationssprachen [3] wie EWL [5] oder Kermeta [18], jedoch hat Henshin den Anspruch auch komplexere Transformationen zu erlauben [1]. Dadurch ist Henshin ein sehr komplexes System, das unter anderem versucht ein algorithmisch schweres Problem zu lösen [1, 2, 13]. Diese Komplexität bringt einige Nachteile mit sich, wie mögliche Defizite in der Laufzeit, dem Speicherverbrauch und der Wartung/Erweiterung des Codes. In dieser Arbeit wollen wir beleuchten, ob sich einige dieser Nachteile, insbesondere die Wartbarkeit/Erweiterbarkeit ausgleichen lassen, indem Ecore durch eine leichtgewichtigere Implementation ersetzt wird.

Bevor dieses Ersetzten der Implementation stattfindet versuchen wir in dieser Arbeit zu ergründen, ob dieses Vorhaben überhaupt von Erfolg geprägt sein kann. Dazu identifizieren wir die Teile von Ecore, die Henshin tatsächlich benötigt. Dazu machen wir uns einige Tools wie die Java-Reflection Libary (zur Analyse von Klassen und Methoden zu Laufzeit) und den JavaParser (zur statischen Codeanalyse) zu nutze. Die so identifizierten Teile vergleichen wir anschließend in ihrem Umfang und einigen ausgewählten Attributen mit dem original Ecore.

In dieser Analyse haben wir festgestellt, dass die überwältigende Mehrheit der Klassen und Methoden von Ecore von Henshin nicht benötigt werden. Daher kann davon ausgegangen werden, dass das „Ausbauen“ dieser Teile zumindest eine realistische

Chance hat, die Speicher- oder Laufzeitbedürfnisse Henshins oder dessen Wartbarkeit zu verbessern.

2 Hintergrund

Das Eclipse Modeling Framework (EMF) ist ein auf der IDE Eclipse basierendes Modeling-Framework, das Applikationen und Code aus bereitgestellten Datenmodellen erzeugen kann [16]. Auf der Basis von Spezifikationen, die im XML Metadata Interchange (XMI) Format vorliegen, kann EMF Javaklassen erzeugen. Zusätzlich können diese Klassen in EMF annotiert und editiert werden. EMF erlaubt damit seinen Nutzern, Modelle direkt in lauffähigen Code zu übersetzten; eine abstrakte und vereinfachte Art der Softwareentwicklung.

Die Basis von EMF bildet dabei *Ecore*. Ecore ist ein (Meta-)modell, dass die Modelle in EMF beschreibt und abbildet. Zusätzlich ist Ecore sein eigenes Metamodell, also Ecore ist selbst als Ecore-Modell spezifiziert. Ecore unterteilt sich in drei große Java-Pakete: emf.common, emf.ecore und emf.ecore.xmi. Jedes dieser Pakete enthält mehrere Subpakete die jeweils eigene Klassen und Subklassen implementieren. Für eine volle Visualisierung des Aufbaues der Pakete siehe Abb. 1.

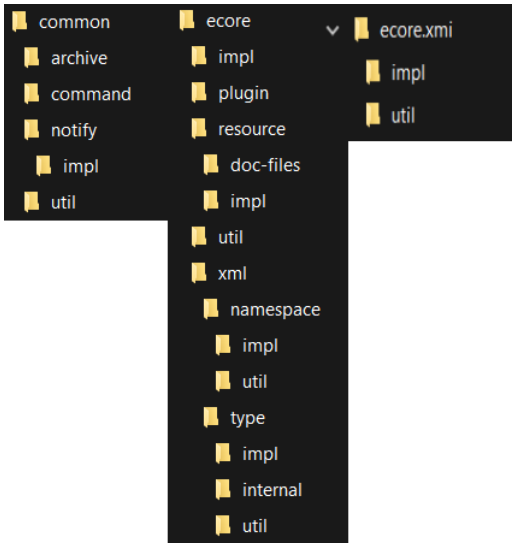


Abbildung 1: Aufbau von Ecore

Ein Bestandteil von EMF ist dabei „Henshin“ [19, 1], eine state-of-the-art Modelltransformationssprache. Basierend auf (vom Nutzer bereitgestellten) Grammatiken kann Henshin die EMF-Modelle (beschrieben als Ecore-Modell) transformieren.

Henshin leidet dabei unter seiner eigenen Komplexität. Die Transformationsregeln von Henshin sind vergleichsweise wenig Beschränkungen unterworfen, wodurch diese sehr aussagekräftig werden können. Gleichzeitig bringt diese Komplexität Laufzeitprobleme mit sich [14, 13]. Daher kann die Performance über bessere Algorithmen nur begrenzt verbessert werden, eine Verbesserung an anderen Stellen wäre vorteilhaft.

Eine Möglichkeit wäre die Basis von Henshin zu verbessern. Wie oben erwähnt ist Henshin Teil von EMF und basiert damit auf Ecore. Jedoch ist Ecore die Basis für gesamt EMF und Henshin ist nur eine Funktion von EMF. Da viele andere EMF-Funktionen auch Ecore benutzen, können wir davon ausgehen, dass Henshin nicht jede Funktionalität

von Ecore benötigt [16]. Daher kann Henshin möglicherweise verbessert werden (z.B. niedrigere Laufzeit, niedrigeren Speicherverbrauch oder einfachere Wartbarkeit) indem die Ecore-Basis soweit möglich durch leichtgewichtigere Objekte ersetzt wird.

Diese Bachelorarbeit legt die Grundlage für diese Verbesserung. Dabei soll festgestellt werden, ob das Ziel überhaupt realistisch ist. Dazu wird Henshin im Hinblick auf seine Ecore-Abhängigkeit analysiert. Dazu soll überprüft werden, welche Klassen und Methoden von Ecore Henshin tatsächlich benötigt, und wie groß dieser Anteil ist. Sollte sich dieser Anteil als sehr groß herausstellen (also sollte Henshin fast komplett Ecore benötigen) kann davon ausgegangen werden, dass eine Ersetzung durch schlankere Objekte nicht möglich ist.

3 Methodik

Das Ziel dieser Arbeit ist es, diejenigen Teile von Ecore zu identifizieren, die von Henshin benötigt werden, um später zu messen, wie groß dieser Anteil ist. Für die so identifizierten Klassen und Methoden wollen wir herausfinden, ob eine Ersetzung von Ecore (in Henshin) zu einer Verbesserung hinsichtlich Wartbarkeit oder Laufzeit/Speicherbedarf führen könnte.

Als erstes muss das Subset von Ecores Klassen und Methoden gefunden werden, das Henshin tatsächlich benötigt (Kapitel 3.1). Jede Ecore-Klasse wird dabei durch eine neue Klasse, in Form eines Interfaces, ersetzt. Dieses Interface wird als Stub benutzt. Da die eigentliche Funktionalität der Klassen für uns nicht relevant ist (wir sind nur an den Abhängigkeiten interessiert), verlieren wir durch die Stubs keine Information, gewinnen aber einfachere Konstrukte. Der einzige Nachteil ist der Verlust einiger Methodenaufrufe, wie später in Kapitel 3.4 beschrieben.

Im nächsten Schritt, in Kapitel 3.2, wird Henshin durchsucht und jede Methode, die nun nicht mehr gefunden werden kann (weil sie aus Ecore stammt und dieses entfernt wurde), wird in den Interfaces ebenfalls als Stub implementiert. Diese Funktionen werden von Henshin zwingend benötigt und müssen daher von uns gesammelt werden. Außerdem müssen einige Methodenaufrufe gecasted werden, um eine Typenübereinstimmung unterschiedlicher Variablen zu erreichen.

Für die anschließende Analyse in Kapitel 3.3 von Henshin nutzen wir Java-Reflection. Dies erlaubt es zur Laufzeit Klassen und Methoden auszulesen. Damit können in Java die Klassen und Methoden verglichen und ausgewertet werden, etwa indem für jede Methode zur Laufzeit ihre Sichtbarkeit oder für jede Klasse die Anzahl ihrer implementierten Methoden ausgelesen wird.

Neben den Methodenaufrufen aus Henshin gibt es auch die Möglichkeit, dass Ecore-Methoden andere Ecore-Methoden aufrufen. Solche Aufrufe wären indirekt ebenfalls nötig für Henshin. Dies lässt sich nicht in den Stubs nachvollziehen, da diese keine Implementierung der Funktion mehr haben, und muss daher in den original Java-Dateien erfolgen. Dazu haben wir in Kapitel 3.4 statische Code-Analyse genutzt.

Diese Auswertungen muss anschließend nur noch aufbereitet und ausgegeben werden (Kapitel 4). In dieser Auswertung können wir überprüfen, welche Teile von Ecore tatsächlich von Henshin benötigt werden. So können wir einschätzen, ob eine zukünftige Verbesserung von Henshin (Wartbarkeit/Laufzeit/Speicherbedarf) möglich ist.

3.1 Grundlagen und Aufbau von Ecore

Für eine umfangreiche Analyse interessieren uns neben den Namen, auch die Attribute (etwa die Sichtbarkeit), der Klassen/Methoden. Daher sammeln wir alle Informationen über die Klassen/Methoden in zwei Listen, in jeder sind jeweils Klassen und ihre zugehörigen Methoden. Wie genau diese befüllt werden besprechen wir in Kapitel 3.2. Um zwischen diesen beiden Listen unterscheiden zu können geben wir ihnen unterschiedliche Präfixe, die sich dann auch auf Klassen und Methoden dieser Liste beziehen:

Die α-Liste (auch kurz nur α) enthält alle Ecore-Klassen und Methoden, unabhängig davon ob Henshin diese benötigt oder nicht. Die Klassen werden entsprechend α-Klassen genannt, die Methoden α-Methoden.

Analog sind β-Klassen und β-Methoden diejenigen, die von Henshin benötigt werden. Sie liegen in der entsprechenden β-Liste (auch kurz nur β). Diese ist zu Beginn leer und wird nach und nach befüllt.

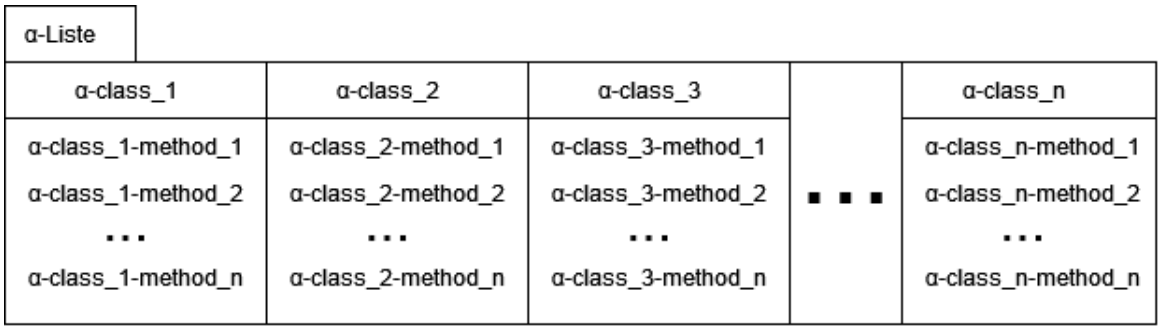


Abbildung 2: Aufbau der α-Liste

Diese Listen sind zum einen abstrakte Konzepte, um Diskussionen in dieser Arbeit zu vereinfachen. Allerdings repräsentieren sie später (Kapitel 3.3) auch konkrete Implementierung (in Formen von zwei Arrays), um Analysen zur Laufzeit zu ermöglichen.

Wie oben erwähnt, interessieren wir uns auch für Methodenaufrufe von Ecore nach Ecore. Der Grund, dass diese ebenfalls berücksichtigt werden sollten, ist die bereits erwähnte Komplexität von Ecore. Jede von Henshin aufgerufene Ecore-Funktion könnte intern (und damit unsichtbar für Henshin) weitere Ecore-Funktionen aufrufen. Im schlimmsten Fall könnten alle Ecore-Methoden intern durch Aufrufe verknüpft sein; dann könnte kein Teil von Ecore entfernt werden, ohne Funktionalität zu verlieren. In Abb. 3 sehen wir einen Abhängigkeitsgraph, der Ecore veranschaulicht. Wir sehen einen hohen Grad der Interoperabilität und eine tiefe Vererbungshierarchie. Daher müssen auch die internen Vorgänge von Ecore berücksichtigt werden. Aufgrund dieses Aufbaus können wir davon ausgehen, dass Ecore sich an sehr vielen Stellen selbst aufruft. Auch diese Aufrufe werden wir später analysieren.

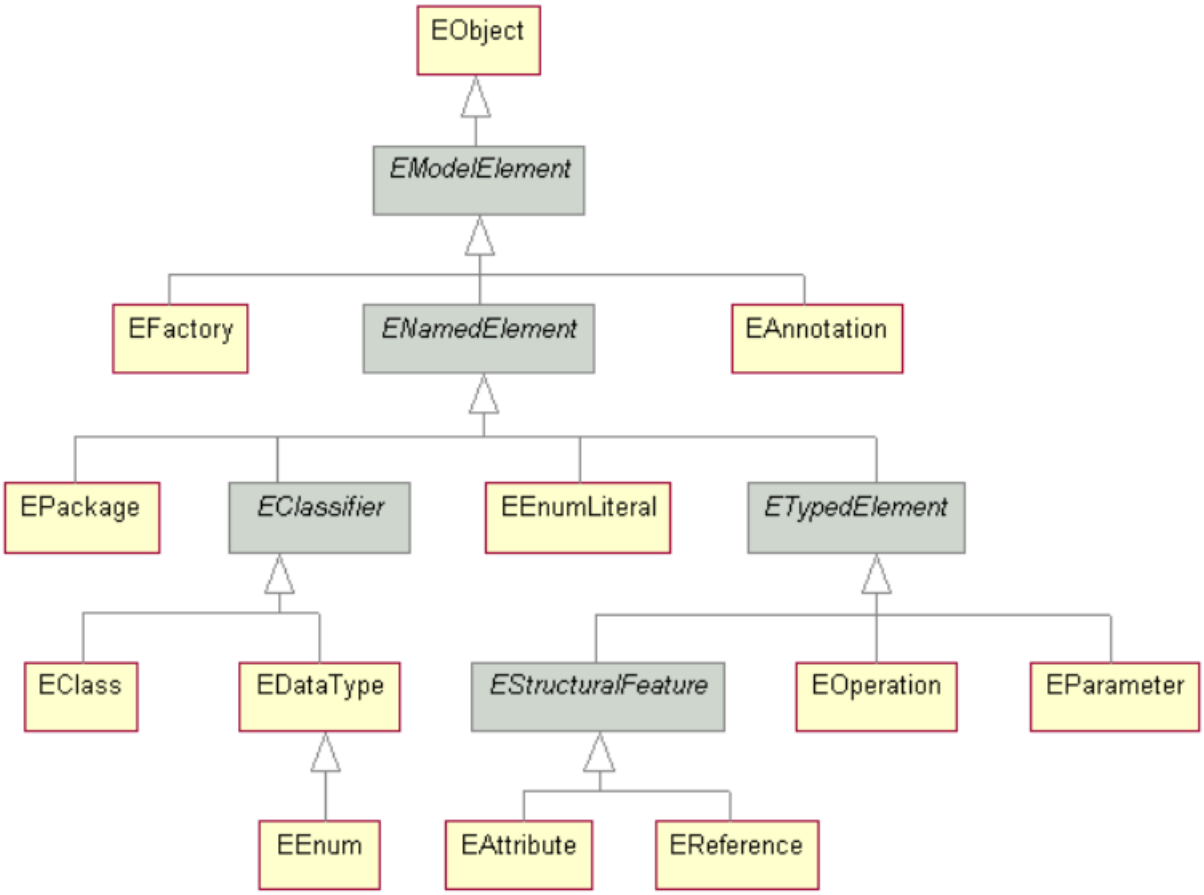


Abbildung 3: UML-Diagramm von Ecore [17]

3.2 Relevante Klassen und Methoden: Die β-Liste

Ziel ist es, alle Klassen und Methoden zu identifizieren, die Henshin benötigt und diese in die β-Liste zu schreiben. Dazu müssen wir Henshin untersuchen und identifizieren, welche Ecore-Klasse und Methoden benutzt werden.

Zunächst stellen wir dabei fest, dass nicht ganz Henshin von Ecore abgekoppelt werden kann. Wie erwähnt ist Ecore eine Metamodell-Sprache und spezifiziert nicht nur EMF sondern auch sich selbst. Und Ecore spezifiziert auch teilweise Henshin. Einige Komponenten von Ecore werden also nicht Teil dieser Analyse sein. Glücklicherweise ist diese Einteilung im Quellcode recht einfach. Henshin hat einen „Model“-Teil und einen „Interpreter“-Teil, letzterer ist für die Grammatiktransformation verantwortlich. Der Model-Teil kann nicht von Ecore getrennt werden, das Metamodell ist einfach zu tief integriert. Jedoch kann der Interpreter von Ecore getrennt werden, und das ist auch genau der Teil, der für uns von Interesse ist. Der Interpreter übernimmt das Finden und Transformieren der Regeln, wie in Kapitel 2 beschrieben. Dies entspricht genau dem Teil, den wir verbessern wollen.

Da der Model-Teil für die weitere Arbeit keine Rolle spielt, beziehen wir uns ab jetzt mit „Henshin“ stets nur auf den Interpreter.

Ziel ist es nun, die β-Liste zu befüllen. Dazu müssen alle α-Klassen und Methoden gefunden werden, die von Henshin benötigt werden. Dazu kann die grundlegende Java-Struktur verwendet werden. Wie in Java üblich müssen alle Ecore-Klassen, auf die Henshin zugreifen will, importiert werden (über das „import “-Statement). Indem wir diese Imports manipulieren, können wir die relevanten Klassen extrahieren; eine beispielhafte Extraktion ist in Abb. 4 zu sehen.

Sukzessive gehen wir alle Henshin Java-Klassen durch. Für jede importierte α-Klasse erzeugen wir eine eigene β-Klasse. Diese hat den identischen Namen. Auch die (Ver)erbung wird übernommen (jedoch werden auch diese α-Klassen durch β-Klassen ersetzt). Um die Implementierung zu erleichtern, werden jedoch alle Klassen durch Interfaces ersetzt. Da diese Analyse keine funktionsfähigen Prototypen zum Ziel hat, ist dies keine Einschränkung.

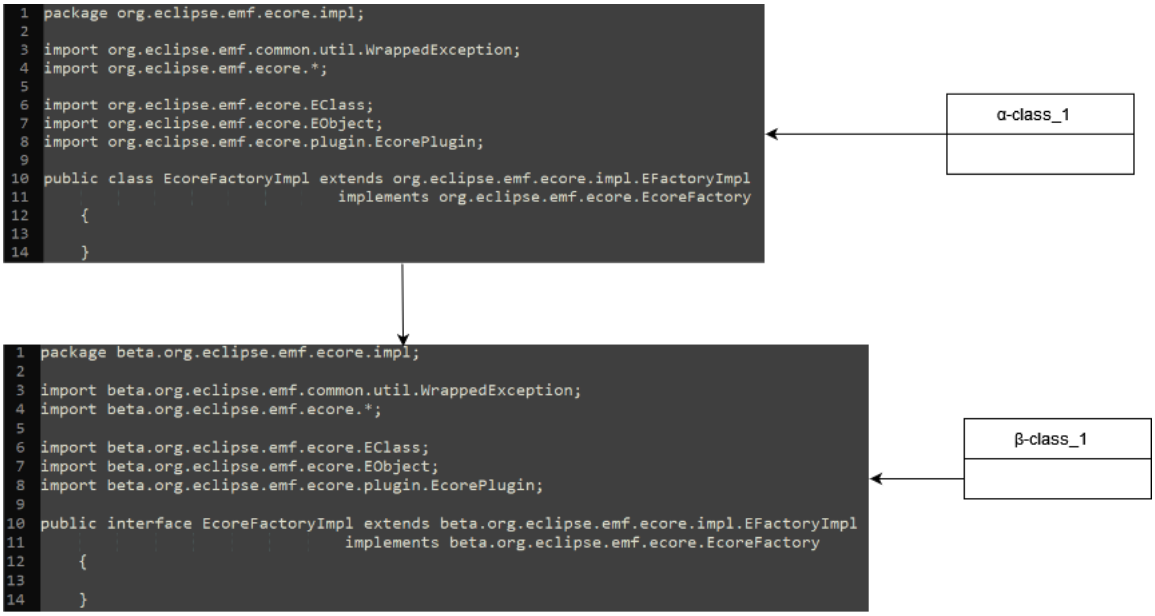


Abbildung 4: Transformation α zu β Klasse

Nachdem alle Imports ersetzt wurden, haben wir eine Liste aller Ecore-Klassen, die Henshin benötigt. Das sind unsere β-Klassen. Im nächsten Schritt müssen diese Klassen nun mit den relevanten Methoden gefüllt werden (Siehe Abb. 5). Da diese Methoden keine Funktion bieten müssen, implementieren wir sie nur als Stub. Das heißt, die Methoden bekommen die gleiche Sichtbarkeit, den gleichen Namen, den gleichen Rückgabewert und die gleiche Parameterliste (wobei wie zuvor Verweise auf α-Klassen/Methoden durch β-Klassen/Methoden ersetzt werden). Der Methodenkörper hingegen wird nicht übernommen und bleibt leer (bzw. gibt einen Standardwert zurück, wenn nötig).

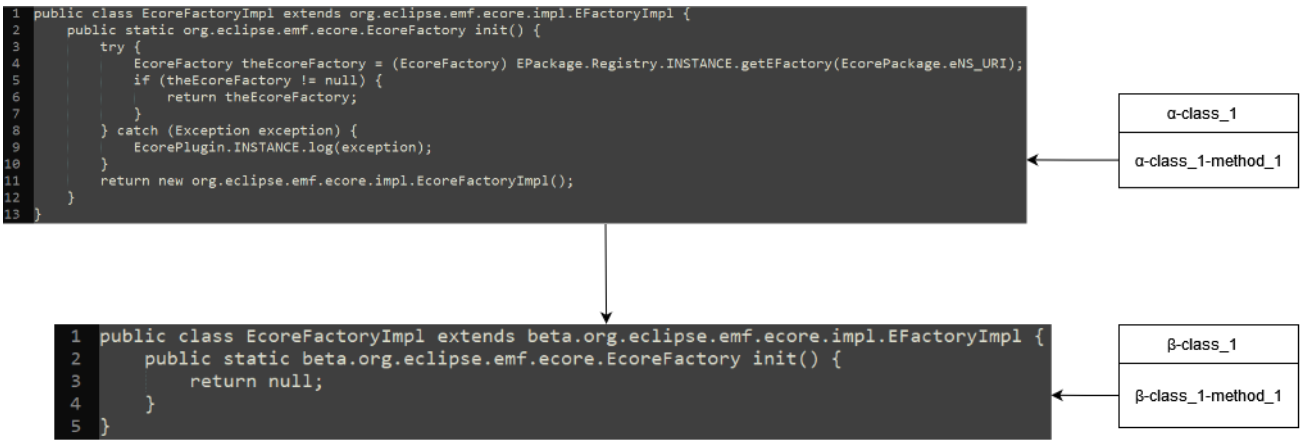


Abbildung 5: Transformation α zu β Methode

Das Finden, welche Methoden deklariert werden müssen, übernimmt der Java-Compiler für uns. Da die Imports bereits ersetzt wurden, können die relevanten Methoden jetzt nicht mehr aufgelöst werden. Entsprechend werden sie beim Kompilieren mit einem Fehler zurückgegeben.

Bevor wir sicherstellen, dass wir alle notwendigen Methoden deklariert haben und keine weiteren, ist es jedoch nötig, einige Typcasts in Henshin zu ergänzen. Da an mehreren Stellen der Interpreter mit dem Model interagiert (und der Model-Teil von Henshin nicht von Ecore gelöst werden kann, siehe Kapitel 3.2) ist es nötig, an diesen Stellen die Typen zu casten. Insgesamt sind 40 solche Casts nötig (für ein volle Liste siehe Anhang 8.1, für ein Beispiel siehe Abb. 6).

```
original Zeile:
    EClass nodeType = node.getType();
Zeile nach dem Cast:
    EClass nodeType = (EClass) node.getType();
```

Abbildung 6: Beispiel eines Methoden casts zur Auflösung von Typproblemen

Nachdem die Typprobleme gelöst wurden, können wir sicherstellen, dass wir die korrekten Methoden übernommen haben. Der Java-Compiler hilft uns dabei sicherzustellen, dass wir *alle* Methoden ersetzt haben, die Henshin benötigt und auch keine *zusätzlich*. Dazu ist es schlicht nötig, dass das Henshin Java-Projekt korrekt gebaut werden kann. Sollte eine Klasse oder Methode nicht aufgelöst werden können, würde der Compiler einen Fehler zurückgeben. Gleichzeitig würden ungenutzte β-Klasse oder β-Methoden eine Warnung ausgeben.

Das heißt, wenn Henshin erfolgreich und ohne Warnung baut, wissen wir, dass wir *alle* Klassen und Methoden, die benötigt werden, korrekt abbilden und keine Klassen oder Methoden *darüber hinaus*. Es muss noch sichergestellt werden, dass wir keine α-Klassen-Imports übersehen haben, aber das kann durch ein temporäres Entfernen

von Ecore erreicht werden. Das Projekt kann dann zwar nicht mehr gebaut werden (da der Model-Teil Ecore benötigt), aber wir können überprüfen, ob im Interpreter auch ein Fehler vorliegt. Falls nicht, haben wir alle Imports ersetzt. Die Zuweisung erfolgt dabei über den Namen, da dieser in beiden Listen gleich ist.

3.3 Von Abstraktion zu Implementation

Bisher haben wir zwei Ecore-Pakete erzeugt. Einmal das Original (das wir in unserer abstrakten α-Liste zusammengefasst haben) und die reduzierte Version, bestehend aus Interfaces und Stubs, die nur jene Klassen und Methoden deklariert, die Henshin benötigt (zusammengefasst in der abstrakten β-Liste). Das heißt, wir können eine surjektive Abbildung zwischen den Listen vornehmen, mit der α-Liste als Definitionsmenge. Jede β-Klasse und β-Methode hat ein Pendant in der α-Liste. Anders herum ist das nicht zwingend so; die Liste ist möglicherweise nicht injektiv. Das wäre sie nur, wenn absolut alle Ecore-Methoden von Henshin benötigt werden.

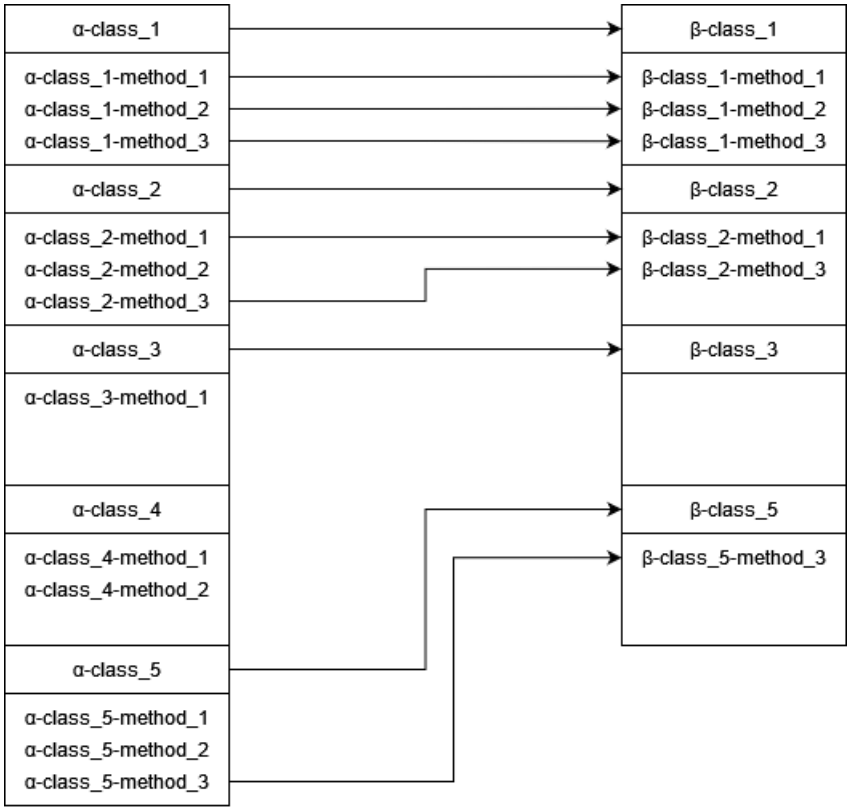


Abbildung 7: Abbildung von $\alpha \mapsto \beta$. Die Nummerierung in β entspricht jeweils dem Pendant in α

Da die β-Objekte aber als tatsächliche Implementation von Java-Klassen/Methoden vorliegen (und die α-Objekte sowieso), können wir unsere Liste auch konkret implementieren (Abb. 7). Das erlaubt uns zur Laufzeit die beiden Sammlungen von Java-Klassen und Java-Methoden zu analysieren und zu vergleichen.

Konkret erzeugen wir zwei Arrays, in denen jedes Feld ein Objekt ist. Dieses Objekt repräsentiert eine Klasse, identifiziert über den vollen Namen. Außerdem hat jedes Feld eine Liste von Methoden, die diese Klasse implementiert. Subklassen, die von anderen Klassen implementiert sind, werden als normale Klasse betrachtet und erhalten ihr eigenes Feld. In die Liste der Methoden kommen nur die genau in dieser Klasse implementierten Methoden. Geerbte Methoden, oder von Subklassen überbrückte Methoden, werden nicht hinzugefügt.

Um diese Liste zu erzeugen, verwenden wir Java-Reflection [8]. Dabei handelt es sich um eine Java-Library, die es einem Programm erlaubt, zur Laufzeit Informationen über Objekte, Klassen und Felder zu gewinnen, etwa alle Member zurückzugeben. Für ein Beispiel siehe Abb. 8. Konkret kann beispielsweise eine Java-Klasse mit einem bestimmten Namen (inklusive Package) ausgelesen werden und als Class-Objekt abgelegt werden. Anschließend können wir uns alle deklarierten Methoden dieser Klasse zurückgeben lassen.

```
1 String className = "org.eclipse.emf.ecore.impl.EClassImpl";
2 Class<?> cls = Class.forName(className);
3 System.out.println(cls.getPackageName());
4
5  > org.eclipse.emf.ecore.impl
```

Abbildung 8: Beispiel Java-Reflection

Damit verfügen wir über die Mittel unsere abstrakten α- und β-Listen zur Laufzeit konkret zu implementieren, dafür müssen lediglich die Klassennamen vorliegen. Diese können wir entweder in einer Textdatei ablegen (sie werden sich nicht mehr ändern), oder zur Laufzeit aus den Dateinamen auslesen.

3.4 Ecore-interne Methodenaufrufe

Bisher haben wir uns nur Henshin angesehen und die dortigen Klassen- und Methodenaufrufe extrahiert. Streng genommen ist das nicht ausreichend. So wie Henshin-Methoden Ecore aufrufen können, können auch Ecore-Methoden andere Ecore-Methoden aufrufen. In der Implementation der β-Liste würden diese Aufrufe wegfallen, da wie oben beschrieben die β-Methoden Stubs mit leeren Funktionskörpern sind (siehe Abb. 9). Wir haben vorhin gesehen (siehe Kapitel 3.3), dass Ecore sehr verzweigt und stark verwoben ist. Wir können also davon ausgehen, dass sich viele Ecore-Methoden gegenseitig aufrufen. Würden wir diese Aufrufe nicht betrachten, riskieren wir ein irreführendes Ergebnis, in dem viele Methoden, die Ecore selbst verwendet, wegfallen (weil Henshin sie nicht direkt aufruft).

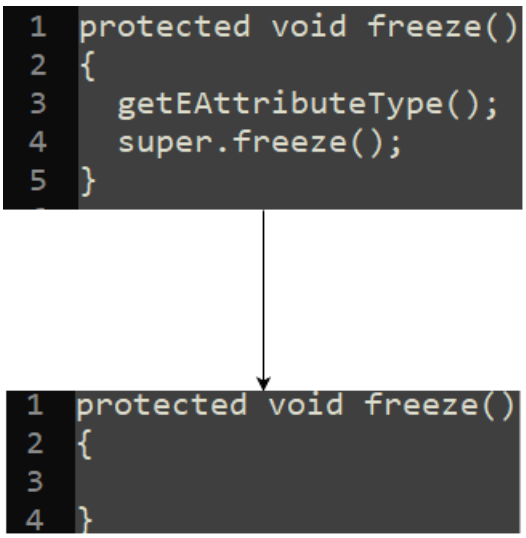


Abbildung 9: Verlust von Methodenaufrufen bei der Überführung in β

Um eine übermäßige Doppelung von Definitionen zu vermeiden, werden wir von jetzt an diese Methodenaufrufe mit „mcxpr“ (MethodCallExpression, der Name aus dem JavaParser) bezeichnen. Konkret wollen wir: Für alle α-Methoden, die ein β-Methoden-Äquivalent haben, für alle mcxpr überprüfen, ob diese ebenfalls ein β-Pendant haben (und wenn nicht wollen wir eines erzeugen).

Im Gegensatz zu den bisherigen Methoden-Aufrufen können die mcxpr nicht länger von Hand überprüft werden, dafür sind es zu viele ($\sim$ 30.000). Zur Laufzeit ist dieses Problem aber auch nicht direkt lösbar. Java-Reflection bietet hierfür nicht die benötigte Funktionalität. Wir wollen für jede α-Methode eine Liste aller Methoden, die diese aufruft. Java-Reflection kann eine Klasse oder Methode aber nur lesen, wenn diese aufgerufen wird. Das heißt, um alle mcxpr über Reflection zu finden, müssten sie alle auslösen, also so lange Testdaten erzeugen, bis wir alle mcxpr ausgelöst hätten (Testdaten, um eine Code-coverage von annähernd 100 % zu erreichen). Ein Vorhaben, das den Umfang dieser Arbeit sprengen würde.

Stattdessen können wir die mcxpr mittels statischer Codeanalyse auslesen. Mithilfe des AST des Programmes und einem Symbol-Solver ist es möglich, die mcxpr aufzulösen und damit in ecore zu verorten. Wir wählen zur Analyse den *JavaParser* [15].

Der JavaParser baut den AST einer gegebenen Java-Datei. Danach erlaubt er das Auslesen der Blätter des AST oder deren Manipulation. Zur Auflösung von Methodenaufrufen ist der AST allein nicht mächtig genug. JavaParser hat dafür einen Symbolsolver, der mittels des Aufbaus der Java-Klasse (imports, Vererbung, etc.) versucht, die korrekte Java-Klasse (und deren Knoten im AST) zu finden. Wir lassen also den JavaParser in allen Klassen alle MethodCallExpressions finden und versuchen diese aufzulösen, um an den Fully-Qualified-Name zu kommen.

Leider hat sich dabei rausgestellt, dass der JavaParser inadäquat ist, um das Ecore-Objekt auszulesen. Trotz anderslautender Beschreibung konnte der JavaParser in unserem Fall einen großen Teil der mcxpr nicht auflösen. Daher haben wir einen rudimentären Symbolsolver implementiert, der selbstständig versucht, den mcxpr aufzulösen,

falls der JavaParser fehlschlägt.

Zur Erinnerung, ein mcxpr ist ein Methodenaufruf. Jeder mcxpr besteht aus drei Teilen: 1) dem Namen, 2) der Anzahl der Argumente, und 3) der 'Range' (Zeilennummer und Zeichennummer). Die Range dient lediglich der einzigartigen Identifikation und ist für die spätere Analyse nicht von Belangen.

Um die mcxpr später korrekt in die β-Liste einfügen zu können, müssen wir uns noch weitere Informationen über sie speichern (für eine grafische Beschreibung siehe Abb. 10.):

1) MethodCaller: Der Name der Methode, die die mcxpr aufruft.
2) ClassCaller: Der Name der Klasse, in der MethodCaller liegt.
3) ClassDeclarer: Die Klasse, die die mcxpr implementiert (der fully qualified name).

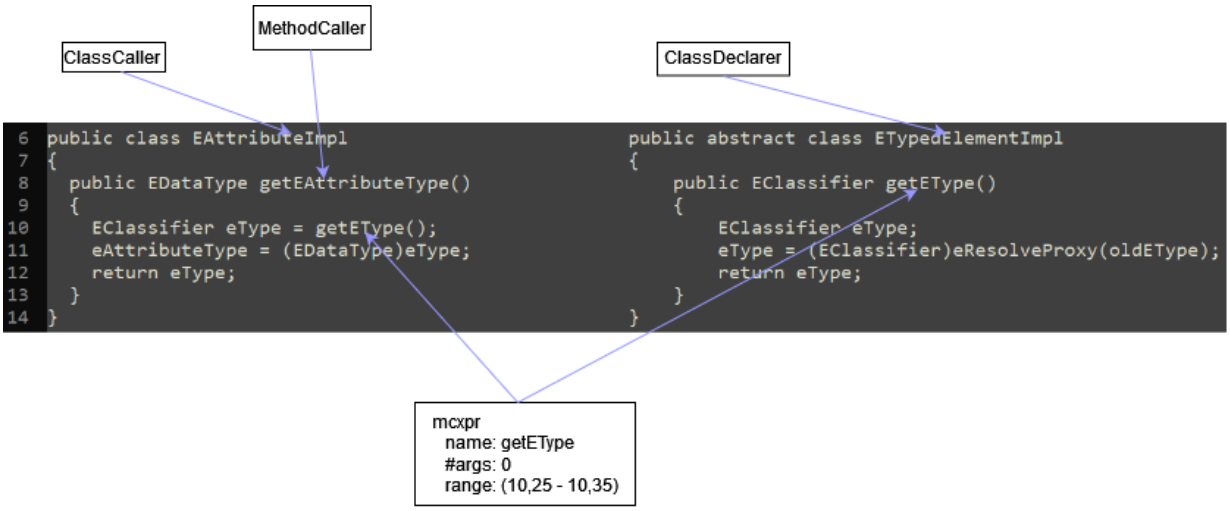


Abbildung 10: Erklärung mcxpr Begriffe

Das Parsing läuft dann in folgenden Schritten ab:

1. Überprüfe, ob die Kombination ClassCaller + mcxpr überhaupt in α existiert. Andernfalls ist es keine Ecore-Methode und für uns nicht interessant.
2. Den JavaParser versuchen lassen, die mcxpr aufzulösen.
3. Überprüfe, ob die mcxpr einzigartig in α ist. Falls ja kommt nur diese Ecore-Funktion in Frage.
4. Überprüfe alle Klassen, die von der ClassCaller Klasse importiert werden. Falls die mcxpr in diesen genau ein mal auftritt, löse zu dieser auf.
5. Überprüfe die Klasse, die von der ClassCaller Klasse 'extended' wird. Kommt die mcxpr dort vor, löse zu dieser auf.

6. Lies die Klassen der Argumente der MethodCaller Methode aus. Ist die Kombination mcxpr in diesen Klassen einzigartig, löse zu dieser auf.

7. Überprüfe alle anderen Methoden in der ClassCaller Klasse, ob eine in Name und #args mit mcxpr übereinstimmt. Falls ja zu dieser auflösen (ClassCaller = ClassDeclarer).

8. Nur falls die mcxpr immer noch nicht aufgelöst werden konnte als nicht auflösbar hinnehmen und mit der Nächsten fortfahren.

Für eine Aufschlüsselung, was diese Art des Parsings für die Ergebnisse der Arbeit bedeutet, siehe Kapitel 5.1.

Damit ist das Parsing allerdings noch nicht abgeschlossen. Bisher wurde nur eine „Karte“ der Methodenaufrufe gebaut. Es ist jetzt noch erforderlich, die mcxpr, wenn nötig, in die β-Liste einzufügen. Das heißt vor allem, relevant sind nicht alle mcxpr, sondern lediglich solche, die in Methoden von β liegen.

Um diese einzusortieren, müssen wir über alle β-Klassen und ihre Methoden iterieren. Für jede Methode überprüfen wir, ob sie eine mcxpr enthält und falls ja, fügen wir die mcxpr in die entsprechende β-Klasse ein. Liegt sie bereits darin, muss nichts weiter getan werden, existiert die Klasse in β noch nicht, muss sie hinzugefügt werden. Da dadurch neue Methoden hinzugefügt werden, könnten auch weitere mcxpr relevant geworden sein. Daher muss dieser Prozess solange wiederholt werden, bis es keine Änderung in β mehr gibt (Pseudocode siehe 3.4).

Listing 1: Pseudocode für das Einfügen der mcxpr

```
1  Repeat until no change:
2  for each class in beta
3     for each method in class
4        for each mcxpr
5           if mcxpr.resolved() == true AND
6           if mcxpr.MethodCaller == method.name AND
7           if mcxpr NOT IN beta
8              if mcxpr.classDeclarer NOT IN beta
9                 insert mcxpr.classDeclarer into beta
10                insert mcxpr into beta
11             else mcxpr.class IN beta
12                insert mcxpr into beta
```

Sobald der Prozess keine weiteren mcxpr mehr in β einfügt, ist die Liste vollständig. Sowohl alle Methodenaufrufe aus Henshin, als auch alle gegenseitigen Ecore-Aufrufe sind nun in β berücksichtigt.

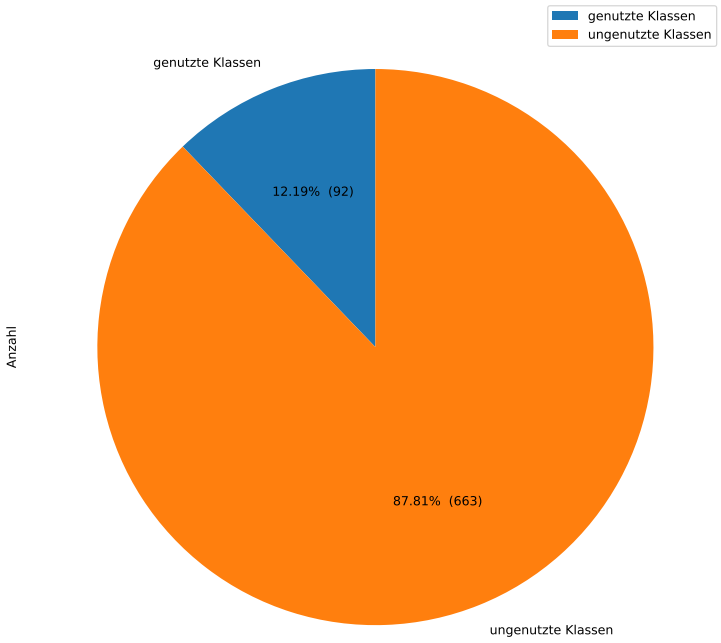


Abbildung 11: Menge der Klassen von β im Vergleich zu α

4 Evaluation

Durch das in Abschnitt 3 beschriebene Vorgehen erhalten wir zwei gefüllte Listen: Die α-Liste mit allen Ecore-Klassen und Methoden und die β-Liste, gefüllt mit den von Henshin genutzten Ecore-Klassen und Methoden. Ehe wir diese Listen miteinander vergleichen, untersuchen wir die Größe der α-Liste, um ein Gefühl für die Datenmenge zu bekommen.

4.1 Klassenvergleich

Ecore besteht, wie in Kapitel 2 beschrieben, aus 3 Paketen, die sich in 20 weitere Subpakete aufteilen. In diesen werden 755 unterschiedliche Klassen implementiert, 159 davon sind Interfaces. All diese Klassen implementieren in Summe 8179 Methoden von denen 1825 die Sichtbarkeit „private“ oder „protected“ haben.

Im Vergleich dazu benötigt Henshin nur einen kleinen Bruchteil. Die β-Liste besteht aus lediglich 92 Klassen (12,19 %). Damit fallen 663 (87,81 %) aller Klassen weg (siehe Abb. 11).

Bevor wir uns die Methoden ansehen, sind noch zwei weitere Punkte interessant. Zum einen können wir die Vererbung nachvollziehen. Java erlaubt es, Methoden anderen Klassen über 'extend' zu erben. Der Aufbau von Ecore nutzt Vererbung exzessive. Das heißt auch, dass ein beträchtlicher Teil der 92 β-Klassen nur für ihre Vererbung existieren. 54 (58,70 %) Klassen in β implementieren keine einzige Methode, die Körper dieser Klassen sind leer. Sie existieren nur zum Zugriff auf geerbte Funktionen (siehe Abb. 12).

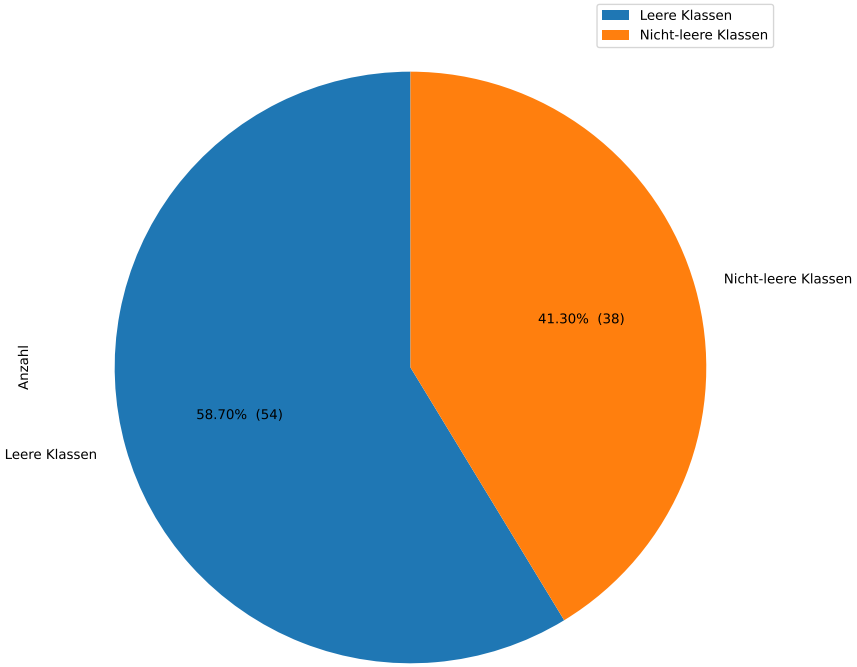


Abbildung 12: Verhältnis der Klassen in β die nur für ihre Vererbung existieren

In Summe sind es damit nur 38 Ecore-Klassen die Henshin tatsächlich benötigt. Alle anderen 717 Klassen sind nicht relevant. Zusätzlich aufgeschlüsselt nach Interfaces ergibt sich die Aufteilung in Tabelle 1. Dies ist ein beeindruckendes Ergebnis, die 717 nicht relevanten Klassen sind für Henshin scheinbar reiner Overhead. In einer leichtgewichtigeren Implementation von Ecore, könnten diese Klassen (und alle ihre Inhalte) weggelassen werden, ohne Henshins Funktionalität zu beeinflussen.

Klassenaufteilung in β			
Anzahl Klassen (inklusive leere Klassen)	davon Interfaces	Anzahl Klassen (exklusive leere Klassen)	davon Interfaces
92	50 (54,35 %)	38	24 (63,16 %)

Tabelle 1: Klassen und Interfaces in β

4.2 Methodenvergleich

Nachdem wir jetzt die Grundlagen für den Vergleich mit den Klassen gelegt haben, können wir uns die Methoden ansehen. Wie in Tabelle 2 zu sehen, implementiert α 8.179 Methoden. Bevor wir diese mit β vergleichen sehen wir uns erst die Ergebnisse des Parsings aus Kapitel 3.4 an.

Ecore (α-Liste)			
Klassen	davon Interfaces	Methoden	davon private oder protected
755	159	8179	1825

Tabelle 2: Anzahl Klassen und Methoden in Ecore

Insgesamt gibt es in α 30.909 Methodenaufrufe (MethodCallExpression/mcxpr). Davon konnten 23.691 (76,65 %) erfolgreich aufgelöst werden, 3.912 nicht.

Parsing der mcxprs		
Parsing Schritt(nach Kapitel 3.4)	Vorgehensweise	aufgelöste Methoden
2	JavaParser	7.713
3	Einzigartigkeit in α	3.898
4	Imports	7.963
5	extends	2.028
6	Argumente	207
7	Eigene Methode	1.882
	Summe	23.691
Parsing Schritt	Vorgehensweise	Nicht aufgelöst
1	Existenz in α	3.306
8	Konnte nicht aufgelöst werden	3.912
	Summe	30909

Tabelle 3: Auflösen der mcxpr

Das Ergebnis ist, dass der Großteil der aufgelösten Methoden nicht relevant für Henshin ist. Von den 23.691 Methoden sind nur zehn (0,042 %) relevant. Des Weiteren ist eine Klasse in β hinzugekommen. Eine Feststellung, wie viele nicht aufgelöste Methoden für β relevant sind, ist leider nicht möglich. Für eine weiterführende Einschätzung dieser Zahlen siehe Kapitel 5.1.

Nach dem Behandeln der mcxpr können wir nun die Anzahl der Methoden in α und β vergleichen. Von den 8.047 Methoden in α werden 132 in β implementiert (siehe Abb. 13).

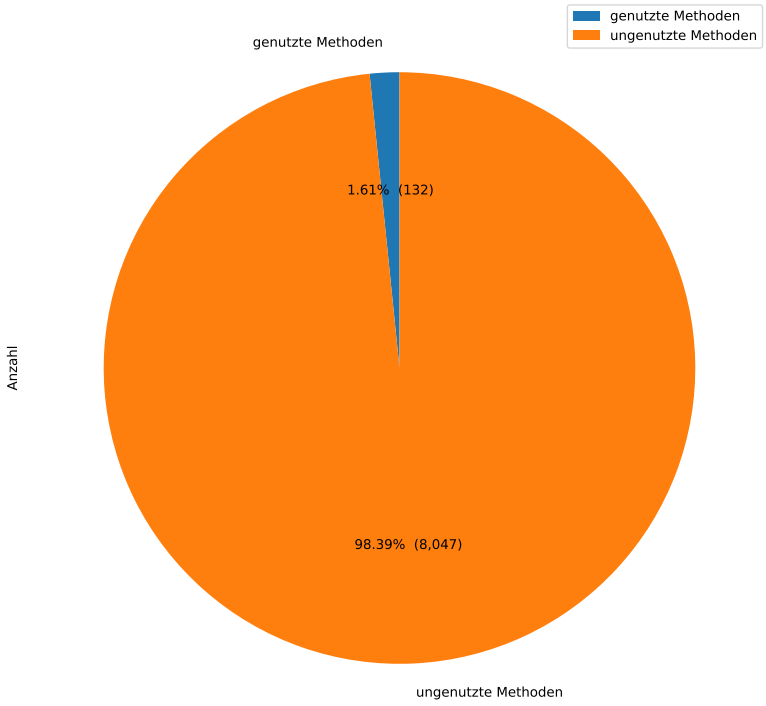


Abbildung 13: Menge der Methoden von β im Vergleich zu α

Wir können eindeutig sehen, dass eine gewaltige Anzahl der Methoden von Ecore nicht benötigt wird. Dies ist ein sehr vielversprechendes Ergebnis, es deutet darauf hin, dass große Teile von Ecore nicht relevant sind für Henshin. Eine leichtgewichtigere Implementation von Ecore (für Henshin) könnte diese Teile ignorieren. Eine weitere interessante Betrachtung ist, wie sich die Methoden aufteilen. Zunächst betrachten wir, wie viele Methoden jeweils in Interface-Klassen wegfallen (Tabelle 4 und 5).

Klasse	Methoden in β	Methoden in α	Prozentualer Wert von β
common.notify.Adapter	4	4	100,00 %
common.notify.AdapterFactory	0	5	0,00 %
common.notify.Notification	5	29	17,24 %
common.notify.NotificationChain	2	2	100,00 %
common.notify.Notifier	2	4	50,00 %
common.notify.NotifyingList	0	3	0,00 %
common.util.Diagnostic	0	7	0,00 %
common.util.DiagnosticChain	1	3	33,33 %
common.util.EList	1	2	50,00 %
common.util.EMap	0	12	0,00 %
common.util.Enumerator	0	3	0,00 %
common.util.Logger	0	1	0,00 %
common.util.ResourceLocator	0	6	0,00 %
common.util.TreeIterator	0	1	0,00 %
ecore.resource.Resource	4	22	18,18 %
ecore.resource.ResourceSet	1	14	7,14 %
ecore.util.InternalEList	0	18	0,00 %
ecore.xmi.XMIResource	0	4	0,00 %
ecore.xmi.XMLResource	0	21	0,00 %
ecore.EAttribute	1	3	33,33 %

Tabelle 4: Prozent der Methoden in Interfaces

Klasse	Methoden in β	Methoden in α	Prozentualer Wert von β
ecore.EClass	10	28	35,71 %
ecore.EClassifier	4	11	36,36 %
ecore.EcoreFactory	1	15	6,67 %
ecore.EcorePackage	18	180	10,00 %
ecore.EDataType	0	2	0,00 %
ecore.EEnum	0	4	0,00 %
ecore.EEnumLiteral	0	7	0,00 %
ecore.EFactory	2	5	40,00 %
ecore.EGenericType	0	11	0,00 %
ecore.EModelElement	0	2	0,00 %
ecore.ENamedElement	1	2	50,00 %
ecore.EObject	10	15	66,67 %
ecore.EOperation	0	7	0,00 %
ecore.EPackage	3	10	30,00 %
ecore.EReference	3	9	33,33 %
ecore.EStructuralFeature	2	17	11,76 %
ecore.ETypedElement	2	14	14,29 %
ecore.EValidator	0	3	0,00 %
ecore.InternalEObject	4	28	14,29 %
common.notify.Adapter$Internal	1	1	100,00 %
common.util.BasicEMap$Entry	0	3	0,00 %
ecore.impl.ESuperAdapter$Holder	0	2	0,00 %
ecore.resource.Resource$Factory	0	1	0,00 %
ecore.resource.Resource$Internal	0	4	0,00 %
ecore.util.InternalEList$Unsettable	0	2	0,00 %
ecore.EPackage$Registry	1	2	50,00 %
ecore.EStructuralFeature$Internal	0	10	0,00 %
ecore.EStructuralFeature$Setting	2	6	33,33 %
ecore.EValidator$Registry	0	1	0,00 %
ecore.EStructuralFeature$Internal $DynamicValueHolder	0	3	0,00 %

Tabelle 5: Prozent der Methoden in Interfaces (Fort.)

Wir sehen in den beiden Tabellen (4 & 5) recht eindeutig, dass der Großteil der Interfaces nicht benötigt wird. Die wenigen Interfaces, die zu 100% genutzt werden, sind meistens recht kleine Klassen. Am auffälligsten ist hier „ecore.EcorePackage“, das von 180 Methoden gerade mal 10 % benötigt.

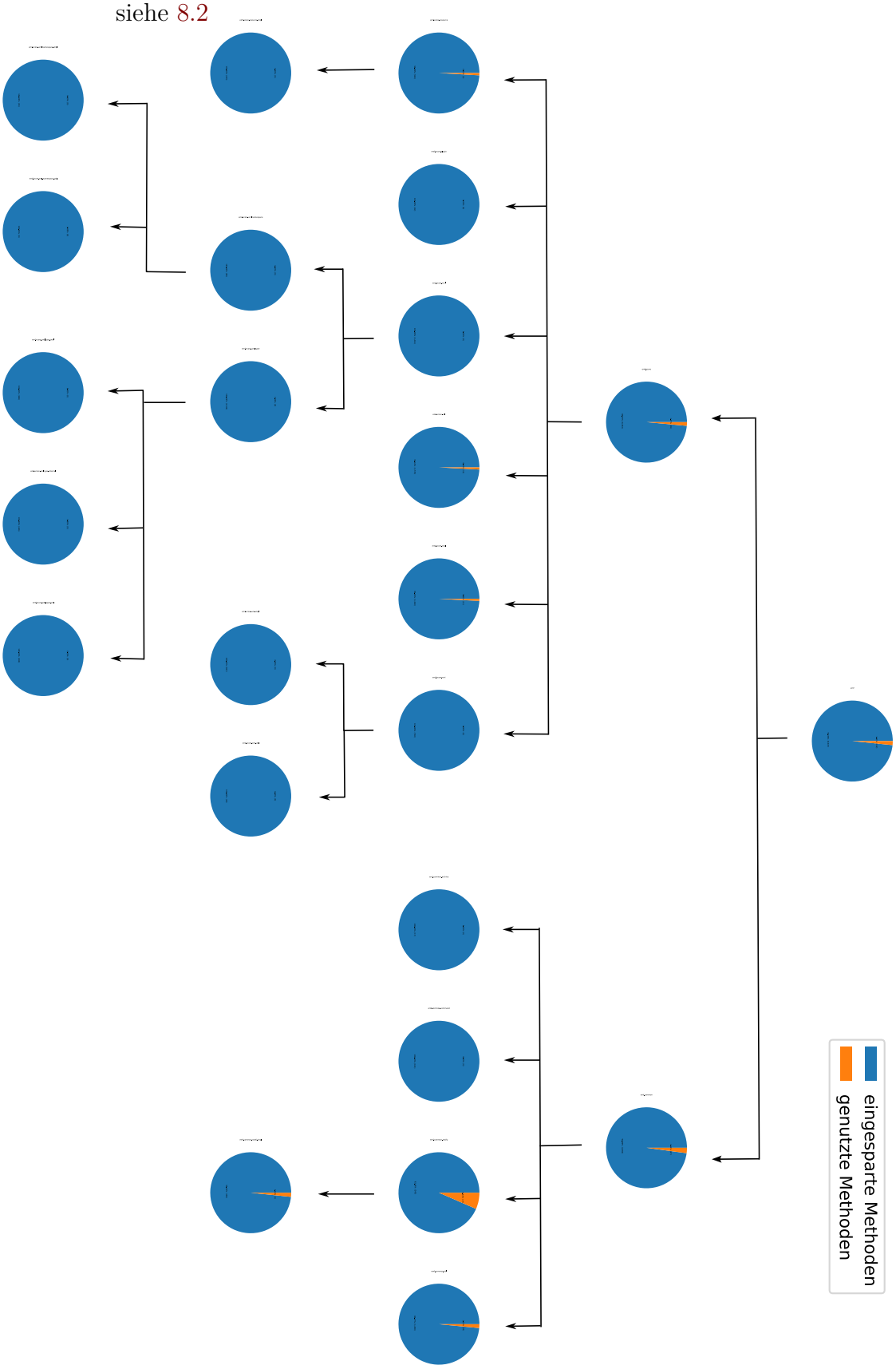
Diese Nutzung der Interfaces können wir auf Basis des SOLID-Prinzips [7] evaluieren. Insbesondere interessant ist hier das „Interface-Segregation-Prinzip (ISP)“. Dieses besagt, Klassen sollten nur von Interfaces abhängen, die sie auch verwenden. Daraus folgt insbesondere, ein Interface sollte nur genau jene Funktionen bereitstellen, die zusammen genutzt werden und nicht mehr.

Für die leeren Interfaces in β ist dieses Prinzip offensichtlich verletzt. Henshin hängt hier von ungenutzten Interfaces ab. Aber auch viele der genutzten Interfaces verletzten das ISP, da sie nur ein Bruchteil der Methoden nutzen. Im bereits erwähnten EcorePackage nutzt Henshin ganze 162 Methoden nicht. Ein so großer ungenutzter Teil bedeutet, dass das Interface zu viele Methoden enthält, die funktional weit auseinander liegen, eine Verletzung des ISP und damit von SOLID. Wichtig ist dabei zu beachten, dass diese Einschätzung nur für β und damit für die Integration in Henshin gilt, nicht zwingend für die Klasse im Allgemeinen, wie sie in Ecore oder anderen Teilen von EMF genutzt wird.

Des Weiteren ist es interessant sich anzusehen, wie β sich auf die einzelnen Packages (Abb. 1) aufteilt. Dazu überprüfen wir den Teil der implementierten Methoden für jedes der drei „Hauptpakete“ und dann sukzessive ihrer Subpakete. Diese bauen wir dann zur einfacheren Übersicht in eine Baumstruktur (Abb 14).

Diese Grafik zeigt recht eindeutig, dass ein großer Teil Ecores von Henshin nicht benötigt wird. Vor allem sehen wir, dass die benötigten Methoden sich auf einige Packages konzentrieren und nicht übermäßig verteilt sind. So fallen „ecore.xmi“ und „ecore.xml“ sowie ihre Subpakete vollständig weg.

Abbildung 14: Vergleich Methoden β / α in den Packages von emf.ecore. Für Rohdaten, siehe 8.2



In Summe sehen wir damit, dass ein großer Teil von Ecore nicht für Henshin relevant ist. Fast 90 % aller Klassen werden nicht benötigt. In den Methoden ist dieses Ergebnis noch viel eindeutiger, über 98 % aller Ecore-Methoden werden von Henshin nicht benötigt. Auch bei Interfaces wird nur ein verschwindend geringer Teil benötigt. Dabei können große Teile von Ecore direkt „rausgeschnitten“ werden, da die relevanten Methoden sich auf einzelne Pakete fokussieren.

Folglich sollte es möglich sein, wie zu Beginn angenommen, die genutzten Teile von Ecore durch andere, leichtgewichtigere Objekte zu ersetzten. Durch den Wegfall großer Teile ungenutzten Codes sollte die Wartbarkeit und Erweiterbarkeit wesentlich leichter werden. Die Abläufe und Prozesse von Henshin sollten einfacher nachvollziehbar werden, wenn weniger ungenutzter Code und Referenzen vorliegen.

Eine Einsparung an Speicherkapazität oder Rechenzeit lässt sich allerdings nicht direkt ableiten. Diese Einsparung wäre möglich, müsste aber erst überprüft werden - durch die tatsächliche Implementation eines solchen alternativen Objektes für Ecore.

5 Kritische Punkte

5.1 Bedrohung der Validität

In Kapitel 3.4 haben wir den JavaParser und einen rudimentären eigenen Symbolsolver benutzt, um Methodenaufrufe (MethodCallExpressions / mcxrp) ebenfalls nachzuvollziehen. Dadurch finden wir in unserer Analyse auch solche Ecore-Methoden die nicht direkt von Henshin benötigt werden, sondern auch von anderen Ecore-Methoden. Bei unserer Herangehensweise ergeben sich zwei Schwachpunkte, die die Analyse möglicherweise negativ beeinflussen.

Der erste Punkt ist die Anzahl der aufgelösten Methoden. Wie in Tabelle 3 zu sehen ist, können 3.912 mcxpr nicht erfolgreich aufgelöst werden. Diese Methodencalls (und damit die implementierten Methoden) kommen dementsprechend nicht in dieser Analyse vor. Von den aufgelösten sind nur zehn mcxpr relevant, daher haben wir die Annahme getroffen, dass auch die Anderen keinen übermäßigen Einfluss auf das Ergebnis hätten.

Jedoch ist es möglich, dass sich ausgerechnet in diesen mcxprs die „Verbindung“ zum restlichen Ecore befindet. Dies könnte vor allem auftreten, wenn diese mcxprs auf einen integralen Bestandteil von Ecore verweisen. Da davon ausgegangen werden kann, dass die nicht auflösbaren mcxprs eine ähnliche Struktur haben, wäre dies durchaus möglich.

Ein weiteres Risiko besteht in unserem Symbolsolver (unter der Annahme, dass der JavaParser fehlerfrei funktioniert). Unser Solver funktioniert in mehreren Schritten. Dabei ist es möglich, dass Methoden zur falschen Methode zugewiesen wurden. Dieses Risiko besteht insbesondere bei Funktionen mit sehr 'allgemeinen' Namen, wie etwa „toString()“. Ferner besteht das Risiko in zwei Richtungen:

Zunächst, dass die erfolgreich aufgelösten mcxpr fehlerhafterweise in eine neue Methode aufgelöst wurden. Also ein Methode die eigentlich bereits in β ist, wurde stattdessen in eine neue Methode aufgelöst. Aufgrund der geringen Anzahl der neuen Methoden (gerade mal Zehn) ist dieser Fehler vernachlässigbar.

Die andere Richtung ist aber auch möglich. Mcxpr, die eigentlich in 'neue' Methoden aufgelöst werden müssten, wurden stattdessen in β gefunden. Dieser Fehler kann leider nicht quantifiziert werden. Wir können lediglich davon ausgehen, dass das Parsing größtenteils erfolgreich war, dann wäre auch dieser Fehler in seiner Anzahl vernachlässigbar.

Beide Risiken könnten durch einen effektiveren SymbolSolver mitigiert werden.

5.2 Risiken in Komplexität

Diese Analyse ist eine reine Quantitative und auch das nur beschränkt auf die Anzahl der Klassen/Methoden und deren Eigenschaften. Dabei gibt es einige Punkte, die nicht abgedeckt wurden.

Allem voran sagt die Analyse nichts darüber aus, 'wie' die relevanten Funktionen

aussehen. Es ist möglich, dass sich die großen und komplexen Vorgänge und Strukturen von Ecore in diesen relevanten Methoden befinden. Das würde bedeuten, dass zwar viele Methoden und Klassen wegfallen, diese aber größtenteils „Füllmaterial“ sind. Dann würde weit weniger gespart werden, als die Zahlen vermuten lassen. Aufgrund der Komplexität von Ecore, ist das jedoch eher unwahrscheinlich.

Eine nicht berücksichtigte quantitative Analyse ist auch die Größe der Methoden. Dieser Punkt ist eng verwandt mit dem Vorhergegangen. Es ist möglich, dass ausgerechnet die relevanten Methoden wesentlich größer sind als die nicht relevanten. Das würde bedeuten, dass die Einsparung an bspw. Lines of Code (LoC) wesentlich geringer ist als die Analyse vermuten lässt.

Auch das ist eher unwahrscheinlich. Zum einen wäre dies sehr untypisch für große Softwareprojekte, dass sich die LoC so ungleich verteilen. Dies wiegt noch schwerer, da viele β-Klassen Interfaces sind[1] (siehe Kapitel 4), die in Ecore zumeist keinerlei Funktion selbst implementieren.

[1] Gemeint sind die original Klassen in α, die für Henshin relevant sind. In β sind wie in Kapitel 3 beschrieben nur Interfaces

6 Konklusion

Wir haben in dieser Arbeit gezeigt, dass ein Ersetzten von Ecore als Grundlage von Henshin ein sinnvolles Unterfangen seien kann. Durch eine Analyse der benötigten Ecore-Methoden wurde gezeigt, dass Henshin nur einen Bruchteil der Klassen und deren implementierten Methoden tatsächlich benötigt. Dies gilt zumindest für den Interpreter-Teil von Henshin, nicht jedoch den Model-Teil. Wir konnten zeigen, dass 87,81 % der Klassen nicht benötigt werden, von den Benötigten sogar 59,78 % nur für ihre Vererbung existieren und gerade einmal 1,49 % der Ecore-Methoden benötigt werden. Dies schließt sowohl Methodenaufrufe von Henshin ein, als auch von anderen Ecore-Methoden.

Zwar ist diese Analyse kein Garant dafür, dass dieses Austauschen von Ecore tatsächlich ein positiven Einfluss auf die Laufzeit oder Speicherkomplexität von Henshin hat, aber sie zeigt, dass der Versuch durchaus lohnenswert ist. Insbesondere eine erleichterte Wartbarkeit, durch kleinere und übersichtlichere Software, scheint erreichbar.

7 Weiterführende Arbeiten

Wie bereits in der Motivation(2) dargelegt, ist diese Arbeit nur der erste Schritt in einer tiefgreifenden Überarbeitung von Henshin. Wir haben gezeigt, dass diese Überarbeitung prinzipiell möglich und sinnvoll seien kann. Nun müsste diese durchgeführt und qualitativ überprüft werden.

Dabei gibt es mehrere Stufen, in denen vorgegangen werden kann. Zunächst sollte überlegt werden, wie genau dieses neue Ecore (im folgenden der Einfachheit β-Ecore genannt, das Original α-Ecore) aussehen sollte. Dabei sollte darauf geachtet werden die Komplexität von α-Ecore wann immer möglich zu vermeiden. Gerade diese Komplexität (und damit einhergehende, für Henshin unnötige Funktionalität) ist es gerade, die die Laufzeit Henshin möglicherweise beeinträchtigt und die Wartbarkeit/Erweiterung des Codes unnötig erschwert. Um die in dieser Arbeit implementierten Interfaces als Basis zu nehmen, bieten sich normale Java-Objekte an.

Als nächstes sollte sich eine Anbindung an Ecore und Henshin überlegt werden. Wie zuvor beschrieben ist es unwahrscheinlich, dass Ecore komplett ausgekoppelt werden kann, da Teile von Henshin und EMF dies benötigen. Gleichzeitig wäre es ein gewaltiges Unterfangen, Henshin so umzuschreiben um auf eine eigene API zuzugreifen. Einfacher wäre es, eine Art Adapter zwischen zu schalten, der die Henshin Anfragen bearbeitet und wann immer möglich an β-Ecore weiterleitet und nur wenn nötig an α-Ecore. Dabei muss beachtet werden, dass ein solcher Adapter Rechen- und Speicherkapazität benötigt, was das ganze Projekt möglicherweise abschwächt oder unmöglich macht. Die Neuimplementation von β müsste schneller sein als α und der Adapter zusammen.

Im letzten Schritt muss β-Ecore implementiert werden. Anschließen müsste das ganze auf Lauf- und Speicherkomplexität mit dem Original verglichen werden.

8 Appendix

8.1 Liste aller type-casts in Henshin

```
File | Line of Code | line
org.eclipse.emf.henshin.interpreter...
impl.MatchImpl 214 EClass nodeType = node.getType();
impl.MatchImpl 225 EReference edgeType = edge.getType();
info.VariableInfo 93 EClass type = node.getType();
info.VariableInfo 149 constraint = new ReferenceConstraint(target, edge.getType(), index, false);
info.VariableInfo 154 constraint = new ReferenceConstraint(target, edge.getType(), constant, true);
info.VariableInfo 160 constraint = new ReferenceConstraint(target, edge.getType(), null, true);
info.VariableInfo 178 ReferenceConstraint constraint = new ReferenceConstraint(target, edge.getType().getEOpposite(), null,true);
info.VariableInfo 192 for (Entry<List<EReference>, Integer>  entry : PathFinder.findReferencePaths(node, target, true, true)
info.VariableInfo 208 constraint = new AttributeConstraint(attribute.getType(), value, false);
info.VariableInfo 212 constraint = new AttributeConstraint(attribute.getType(), constant, true);
info.VariableInfo 241 EReference type = edge.getType();
info.VariableInfo 255 EReference oppType = edge.getType().getEOpposite();
info.VariableInfo 260 if (remoteNode.getOutgoing(oppType, node) == null) {
info.VariableInfo 272 if (node.getOutgoing(oppType, remoteNode) == null) {
impl.EngineImpl 527 constraint_.increaseOutgoing(mulEdge.getType(), count);
impl.EngineImpl 533 constraint_.increaseIncoming(mulEdge.getType(), count);
impl.EngineImpl 562 varEntry.getValue().variableId=EcoreUtil.getURI(varEntry.getKey()).toString();
impl.EngineImpl 861 rule.eAdapters().add(ruleListener);
impl.EngineImpl 964 EClass type = node.getType();
impl.EngineImpl 989 changes.add(new AttributeChangeImpl(graph, object, attribute.getType()));
impl.EngineImpl 1001 completeMatch.getNodeTarget(edge.getTarget()), edge.getType(), false));
impl.EngineImpl 1007 resultMatch.getNodeTarget(edge.getTarget()), edge.getType(), true));
impl.EngineImpl 1028 resultMatch.getNodeTarget(edge.getTarget()), edge.getType(), newIndex));
impl.EngineImpl 1038 attribute.getType().getEAttributeType(), attribute.getType().isMany());
impl.EngineImpl 1044 changes.add(new AttributeChangeImpl(graph, object, attribute.getType(), value));
impl.EngineImpl 1090 return castValueToDataType(evalResult, attribute.getType().getEAttributeType(), attribute.getType().isMany());
impl.Interpreter 129 res.saveEObject(graph.getRoots().get(0), filename);
impl.Interpreter 135 protected EGraph loadGraphFromFile(String inputModelPath) {return new EGraphImpl(res.getResource(inputModelPath));}
impl.LoggingApplicationMonitor 183 resourceSet = new HenshinResourceSet();
matching.jit.TreeJITMatcherCompiler 165 EGraph graph = new EGraphImpl(resourceSet.getEObject("init-grid4x4.xmi"));
util.HenshinEGraph 89 EClass nodeType = node.getType();
util.HenshinEGraph 99 EAttribute attrType = attr.getType();
util.HenshinEGraph 116 EReference edgeType = edge.getType();
util.HenshinEGraph 173 node.setType(eObject.eClass());
util.HenshinEGraph 339 attribute.setType((EAttribute) feature);
util.HenshinEGraph 360 edge.setType((EReference) feature);
matching.constraints.PathConstraint 72 for (EObject trg : (List<EObject>) src.eGet(reference)) {
matching.constraints.PathConstraint 76 EObject trg = (EObject) src.eGet(reference);
matching.conditions.DebugApplicationCondition 1557 List<EReference> references = pathConstraint.getReferences();
matching.conditions.DebugApplicationCondition 1559 for (EReference ref : references) {
```

8.2 Rohdaten des Baumes aus Abb. 14

Package	ungenutzte Methoden	genutzte Methoden
emf	8047	132
emf.ecore	6353	95
emf.ecore.impl	1340	11
emf.ecore.plugin	62	0
emf.ecore.resource	540	5
emf.ecore.resource.impl	433	0
emf.ecore.util	1778	15
emf.ecore.xmi	792	0
emf.ecore.xmi.impl	607	0
emf.ecore.xmi.util	18	0
emf.ecore.xml	1441	0
emf.ecore.xml.namespace	99	0
emf.ecore.xml.namespace.impl	53	0
emf.ecore.xml.namespace.util	9	0
emf.ecore.xml.type	1342	0
emf.ecore.xml.type.impl	388	0
emf.ecore.xml.type.internal	446	0
emf.ecore.xml.type.util	280	0
emf.common	1694	37
emf.common.archive	17	0
emf.common.command	101	0
emf.common.notify	243	17
emf.common.notify.impl	180	3
emf.common.util	1290	20

Tabelle 6: Genutzte und ungenutzte Methoden

Literatur

[1] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, pages 121–135, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] E. Biermann, C. Ermel, and G. Taentzer. Lifting parallel graph transformation concepts to model transformation based on the eclipse modeling framework. *ECEASST*, 26, 01 2010.

[3] C. Ermel, E. Biermann, J. Schmidt, and A. Warning. Visual modeling of controlled emf model transformation using henshin. *ECEASST*, 32, 01 2010.

[4] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE, 2007.

[5] D. Kolovos, R. Paige, F. Polack, and L. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6:53–69, 10 2007.

[6] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. Polack, and G. Botterweck. Taming emf and gmf using model transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 211–225. Springer, 2010.

[7] R. C. Martin. Design principles and design patterns. http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf, 2000. Accessed: 2022.11.01.

[8] G. McCluskey. Using java reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html, January 1998. Accessed: 2022.11.01.

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[10] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2 edition, 2009.

[12] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy. Henshin: A usability-focused framework for emf model transformation development. In *International Conference on Graph Transformation*, pages 196–208. Springer, 2017.

[13] D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling. Scalability of model transformations: Position paper and benchmark set. In *BigMDE@STAF*, 2016.

[14] M. Tichy, C. Krause, and G. Liebel. Detecting performance bad smells for henshin model transformations. *Amt@ models*, 1077, 2013.

[15] D. van Bruggen. Java parsing tool. https://javaparser.org/. Accessed: 2022.11.01.

[16] various. Eclipse modeling framework. https://www.eclipse.org/modeling/emf/. Accessed: 2022.11.01.

[17] various. Ecore components diagram. https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html. Accessed: 2022.11.01.

[18] various. Kermeta 3. http://diverse-project.github.io/k3/index.html. Accessed: 2022.11.01.

[19] various. Henshin project. https://www.eclipse.org/henshin/, July 2014. Accessed: 2022.11.01.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 1. November 2022 Hohmann