Generierung von UML Klassendiagrammen aus Java-Code in Eclipse

Enno Martin Schwanke

Bachelorarbeit
eingereicht im Jahr 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. Christian Schneider

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Die Erstellung von UML-Klassendiagrammen zur grafischen Visualisierung von objektorientierten Softwaresystemen ist eine äußerst hilfreiche Methode, um diese Systeme nachzuvollziehen oder für andere nachvollziehbar zu machen. Es ist jedoch, insbesondere bei komplexeren Systemen, meistens sehr aufwändig Klassendiagramme zu erstellen. Somit wurde in der Vergangenheit versucht, Werkzeuge bereitzustellen, die einem diese Arbeit erleichtern oder sogar den Aufwand auf ein Minimum reduzieren.

Ziel dieser Arbeit ist es ein Eclipse-Plugin zu entwickeln, welches aus einer Selektion von Java-Klassen, -Methoden und -Attributen ein Klassendiagramm erzeugt und dieses dann mittels KIELER Lightweight Diagrams darstellt. Es soll erreicht werden, dass die Erstellung einer Selektion im Package oder Project Explorer möglichst schnell und komfortabel möglich ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDT	Java Development Tools
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KLighD	KIELER Lightweight Diagrams
MDSE	Model-Driven Software Engineering
MOF	Meta Object Facility
OMG	Object Management Group
RCA	Rich Client Application
RCP	Rich Client Platform
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Kapitel 1

Einleitung

Es gibt viele Methoden, das Design von objektorientierten Softwaresystemen zu erleichtern. Einige Methoden, wie zum Beispiel die Spezifikation, die Visualisierung oder die Dokumentation von Softwaresystemen, können mit der Unified Modeling Language (UML) realisiert werden. Sie wird benutzt um Informationen über ein System zu verstehen oder zu überblicken, aber auch um das Design zu unterstützen, indem zum Beispiel Strukturschwächen erkannt werden. Die UML umfasst verschiedene grafische Modellierungssprachen, mit denen verschiedene Aspekte eines Systems dargestellt werden können. Es gibt zum einen Modellierungssprachen, die die Struktur eines Systems offenlegen, aber auch verschiedene Modellierungssprachen zur Beschreibung der Dynamik eines Systems, welche das Verhalten eines Systems über einen Zeitraum darstellen (zum Beispiel Sequenzdiagramme und Zustandsdiagramme). Eine Darstellung der Struktur ist mittels grafischer Repräsentation durch Klassendiagramme gegeben. Klassendiagramme bieten eine statische Sicht auf Softwaresysteme, indem sie Klassen mit deren Inhalt und Beziehungen untereinander modellieren. [RJB99]

1.1. Problembeschreibung

Es ist oft mühsam Klassendiagramme anzufertigen, da vor allem größere Softwaresysteme viele Klassen und dementsprechend auch viele Abhängigkeiten haben. Es ist also hilfreich Werkzeuge zu benutzen, die das Erstellen solcher Klassendiagramme erleichtern, oder dem Nutzer sogar vollständig die Arbeit abnehmen. So existieren bereits zahlreiche Werkzeuge, die diese Anforderungen erfüllen. Es gibt Werkzeuge, die das Zeichnen des Diagramms mithilfe einer textuellen Syntax erleichtern (z.B. yUML[1]) und es existieren schwergewichtigere Lösungen, welche eine Vielzahl von Anpassungsmöglichkeiten und Optionen bieten und die Entwicklung eines Klassendiagramms immens erleichtern (z.B. Omondo UML[2]). Ferner wurden Ansätze entwickelt, welche als Erweiterung in die Integrated Development Environment (IDE) *Eclipse* integriert werden (z.B. ObjectAid[3] und Green [Wan+07]).

[1] http://www.yuml.me

[2] http://www.omondo.com/

Ziel dieser Arbeit ist es, eine Eclipse-Erweiterung zu entwerfen, die es erlaubt, aus Softwaresystemen Klassendiagramme zu synthetisieren. Die Aufgabe besteht darin, ein Werkzeug zu entwickeln, welches optisch ansprechende Diagramme erzeugt, die man beispielsweise zur Dokumentation verwenden kann, um Implementierungsstruktur und -aufbau zu beschreiben. Ferner könnten die entstandenen Diagramme bei der Verbesserung des Strukturdesigns bei größeren Systemen helfen. Hierzu soll ein Werkzeug entwickelt werden, welches zum einen die Darstellung von einzelnen Diagrammelementen ermöglicht und zum anderen diese Diagrammelemente angemessen anordnet. Weiterhin ist das Ziel eine möglichst leichtgewichtige Lösung zu entwickeln, die dennoch so viele Anpassungsmöglichkeiten wie möglich bietet, um qualitativ hochwertige Diagramme zu erzeugen. Dem Anwender soll ermöglicht werden eine Auswahl von zu visualisierenden Elementen zu erstellen und daraus ein Diagramm zu generieren, wobei die Anordnung der Diagrammelemente (Layouting) automatisch erfolgen soll. Für das automatische Layouting wird das KIELER Lightweight Diagrams (KLighD) Framework verwendet [SSH13].

1.2. Visualisierung mittels KLighD

KLighD ermöglicht die Komposition von Diagrammstrukturen und das automatische Anordnen der Diagrammelemente auf der Zeichenfläche. Durch diese Automatisierung wird die Zeit, die beim Erstellen von Klassendiagrammen stets ein störender Faktor ist, minimiert. Der Anwender muss lediglich entscheiden, welche Daten dargestellt werden sollen. Weiterhin wird die Integration von Model-Driven Software Engineering (MDSE) Konzepten und Werkzeugen gefördert. Entsprechend wurde in dieser Arbeit zum Beispiel die Modelltransformations- und Codegenerierungssprache Xtend[4] verwendet. Eine weitere Stärke von KLighD ist die Effizienz. Die Betrachtung von größeren Diagrammen ist genauso komfortabel wie die Betrachtung von kleinen Diagrammen. Es entstehen hierbei kaum bemerkbare Verzögerungen.

1.3. Aufbau dieser Arbeit

Zunächst werden in der vorliegenden Arbeit einige verwandte Arbeiten behandelt und andere schon bestehende Werkzeuge vorgestellt. Daraufhin werden im Kapitel Entwurf die Anforderungen an die zu entwickelnde Lösung herausgearbeitet und ein Metamodell für die zu modellierenden Daten entwickelt. Weiterhin wird die Übersetzung von Klassenstrukturen in Klassendiagramme ausgearbeitet. Zuletzt werden im Kapitel Entwurf die Designentscheidungen beschrieben und

[4] http://www.xtend-lang.org

begründet. Im Kapitel darauf wird beschrieben, welche Technologien verwendet werden, um die auf den Entwurf folgende Implementierung zu unterstützen. Im Kapitel Implementierung wird auf Implementierungsdetails eingegangen. Zuletzt wird die Implementierung durch Anwendung an einem Beispiel und durch Performancetests evaluiert und es werden im Fazit Erweiterungsmöglichkeiten diskutiert.

Kapitel 2

Verwandte Arbeiten

In diesem Kapitel werden zunächst einige Werkzeuge vorgestellt, die bei der Erstellung von Klassendiagrammen helfen.

Des Weiteren werden Zusammenhänge zwischen dieser Arbeit und der von Wißmann [Wiß13] herausgearbeitet und vergleichen.

Zuletzt wird das Thema Layouting von Klassendiagrammen behandelt. Hierzu wird eine Publikation von Eichelberger behandelt, welche sich mit diesem Thema auseinandersetzt [Eic02]. Daraufhin wird auf evolutionäres Layout eingegangen, welches bei der Ermittlung von einem optimalen Layout für ein Klassendiagramm sehr von Nutzen sein kann. Eine Publikation hierzu wurde von Gudenberg et al. [Gud+06] veröffentlicht. Der technische Bericht von Spönemann et al. [SDH14] führt dazu, dass eine mögliche spätere Implementierung des evolutionären Verfahrens in das hier entwickelte Werkzeug diskutiert wird.

2.1. Tools zur Entwicklung von UML Diagrammen

Es existieren bereits zahlreiche Werkzeuge, um eine erleichterte Erstellung von UML-Diagrammen zu ermöglichen. Einige Werkzeuge helfen nur bei der Erstellung eines bestimmten Diagrammtyps (wie zum Beispiel Klassendiagramm) und andere Werkzeuge decken ein großes Spektrum der UML-Modellierungssprachen ab. Im Folgenden werden einige dieser Werkzeuge vorgestellt.

2.1.1. yUML

Das webbasierte Werkzeug yUML[1] ermöglicht die Erstellung von Klassen-, Use-Case- und Aktivitätsdiagrammen. Die Diagramme müssen vom Benutzer dabei mit einer textuellen Syntax beschrieben werden, wie in Listing 2.1 zu sehen ist.

[1] http://www.yuml.me

Listing 2.1. yUML-Code für ein Beispiel-Klassendiagramm

```
1  [Human|name;address;email|breath()],
2  [Student|studentID;averageGrade|study()],
3  [Professor|salary|teach()],
4  [Lecture|location],
5  [Grade|grade],
6  [Human]^-[Professor],[Human]^-[Student],
7  [Student]<->[Lecture],
8  [Professor]++->[Lecture],
9  [Lecture]1-0..*[Grade],
10 [Student]++1-0..*>[Grade]
```

Klassen werden hierbei in eckigen Klammern beschrieben, und die ihnen unterliegenden Umgebungen für Attribute und Methoden innerhalb der eckigen Klammern mit einem vertikalen Strich abgegrenzt. Attribute und Methoden werden durch Semikola getrennt (Listing 2.1 Zeile 1). Kanten zwischen Klassen werden zwischen zwei durch eckige Klammern definierte Klassen beschrieben. Die Klassen müssen hierbei nicht neu definiert werden, sondern es muss nur auf ihren Namen referenziert werden. Generell wird eine Kante durch einen Bindestrich symbolisiert. Die verschiedenen Pfeilspitzen entsprechen bestimmten Symbolen. Vererbung wird durch „^" (Zeile 6), eine normale Pfeilspitze durch „>" oder „<" (Zeile 7) und Assoziationspfeilspitzen wie Aggregation und Komposition durch „<>" und „++" (Zeile 8) dargestellt. Labels liegen entsprechend im Quellcode zwischen Pfeilspitze und Bindestrich (Zeilen 9 und 10).

Nachdem das Klassendiagramm textuell beschrieben wurde, kann das Diagramm erstellt werden. Das Layouting wird dabei automatisch vollzogen. Das generierte Ergebnis aus dem Quellcode aus Listing 2.1 ist in Abbildung 2.1 zu sehen.

Da das Werkzeug auf einer sehr kompakten textuellen Syntax basiert, kann es sehr mühselig und unübersichtlich werden, ein solches Diagramm zu erstellen. Bei vielen Klassendiagrammelementen kann nicht durch Intuition erschlossen werden, welche die entsprechenden textuellen Entsprechungen sind. Schon der in Listing 2.1 gezeigte Quellcode ist nicht besonders übersichtlich, obwohl dieser nur ein kleines Klassendiagramm erzeugt.

yUML verwendet ein automatisches Layout, welches viel Arbeit beim Anordnen der einzelnen Diagrammelemente erspart. Als *Look & Feel* wird von yUML das „Scruffy"-Design benutzt. Statt diesem eher unkonventionellen Design kann ein schlichteres Design ausgewählt werden. Es wurden eine Vielzahl von Elementen der UML-Klassendiagramm Semantik umgesetzt. Als weiteren Vorteil beschreiben die Entwickler von yUML die Möglichkeit Diagramme einzufärben. Die Einfärbung von Knoten ist auch Thema dieser Arbeit. Genauso ist die auto-

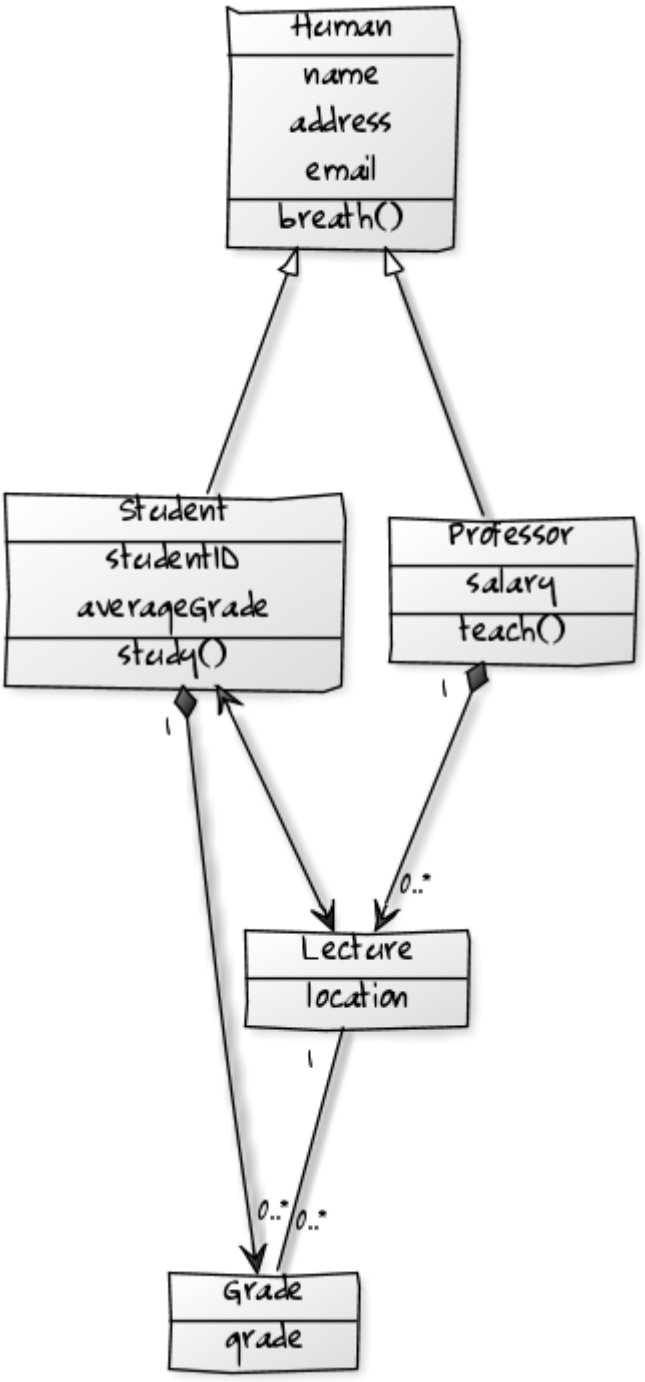


Abbildung 2.1. Ein, von yUML erzeugtes, Klassendiagramm

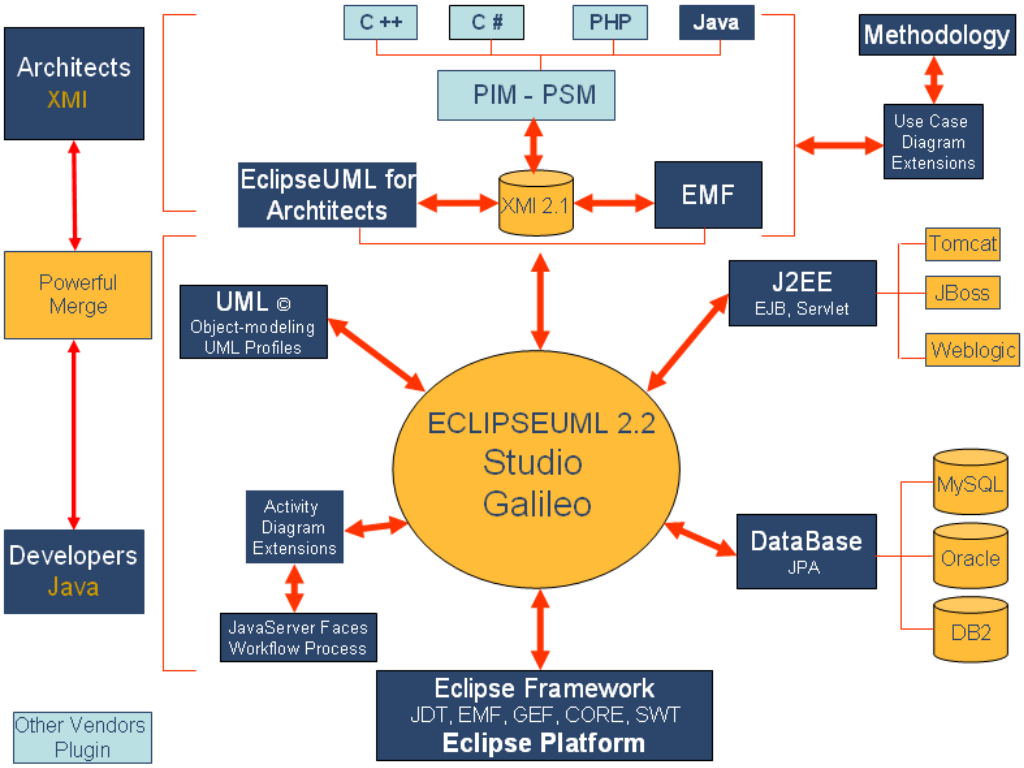


Abbildung 2.2. Die Architektur[3] von Omondo UML (EclipseUML 2.2)

matische Synthese von Klassendiagrammen aus Java-Quellcode, was yUML nicht unterstützt, auch in dieser Arbeit umgesetzt.

2.1.2. Omondo UML

Omondo UML[2] ist eine auf Eclipse basierende Client Applikation. Wie in Abbildung 2.2 zu sehen ist, ermöglicht Omondo UML nicht nur die Erzeugung von UML-Diagrammen, sondern es werden auch andere Features unterstützt. Wenn eine Software sehr umfangreich ist, können die zusätzlichen Features hinderlich für die Übersichtlichkeit sein, sodass eine Einarbeitungszeit erforderlich sein kann.

Diese Arbeit beschäftigt sich mit der Generierung von Klassendiagrammen aus Java-Code. Omondo UML ermöglicht dies und bietet zusätzlich auch die Möglichkeit einer Generierung von Code aus den UML-Modellen. Es kann mithilfe eines Editors ein Klassendiagramm erzeugt werden, aus dem dann Java-Code erzeugt wird. Mit seinen vielen Anpassungsmöglichkeiten gewährleistet Omondo

[2] http://www.omondo.com/
[3] http://www.omondo.com/features.html

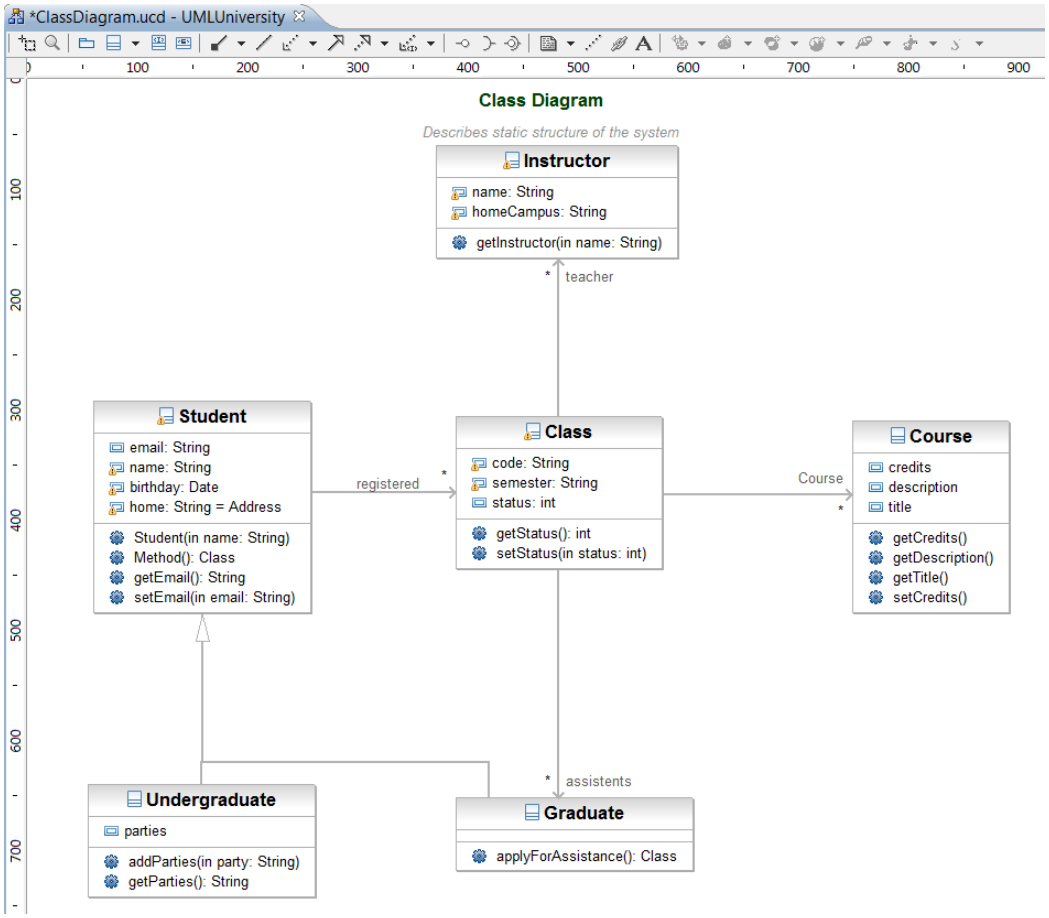


Abbildung 2.3. Ein Klassendiagramm,[4] erstellt mit Omondo UML (EclipseUML 2.2)

UML zwar, dass hochwertige Diagramme automatisch erzeugt oder manuell erstellt werden können, es sind hierzu aber oft kompliziertere Auswahlprozesse in den Einstellungen nötig, um ein gewünschtes Ergebnis zu erreichen. Weiterhin ist Omondo UML ein kommerzielles Werkzeug, welches nur kostenpflichtig erstanden werden kann.

Das in Abbildung 2.3 dargestellte Klassendiagramm wurde mit Omondo UML erzeugt. Es ist optisch ansprechend und beinhaltet zahlreiche Details, wie Attribute, Methoden, Typen und Klassenbeziehungen.

2.1.3. ObjectAid

Für Omondo UML musste eine neue Instanz von Eclipse installiert werden. Es ist hingegen möglich Erweiterungen in Eclipse zu integrieren, anstatt eine externe Software oder eine andere Instanz von Eclipse zu benutzen.

[4] `http://www.omondo.com/features.html`

Eine dieser Erweiterungen ist ObjectAid.[5] Es können Klassen per *Drag'n'Drop* direkt aus dem Package Explorer in eine Diagramm-Datei kopiert werden. Die ausgewählten Klassen werden dann automatisch, mit ihren Beziehungen untereinander, dargestellt. Ein automatisches Anordnen der Diagrammelemente ist nicht möglich und die Diagramme sind optisch einfach und einfarbig gehalten. Ein Vorteil der Erweiterung ist, dass das Diagramm mit dem Java-Code synchronisiert wird.

Die Vorgehensweise dieser Eclipse-Erweiterung ähnelt sehr des in dieser Arbeit vorgeschlagenen Ansatzes in Bezug auf die Verwendung des Package oder Project Explorer zur Auswahl von Klassen. Es ist sehr komfortabel, da man direkt bei der Entwicklung des Softwaresystems die Klassen und deren Methoden, Attribute und Beziehungen untereinander einsehen kann. Man kann jedoch nicht Methoden und Attribute einzeln zur Visualisierung abwählen, um beispielsweise das Diagramm übersichtlicher zu gestalten.

In Abbildung 2.4 wird beispielhaft die Drag'n'Drop-Funktion und die daraus entstandene Visualisierung der ausgewählten Klasse dargestellt.

2.1.4. Green

Eine weitere Eclipse-Erweiterung wurde an der University at Buffalo entwickelt [Wan+07]. Green wurde für Studenten entwickelt, die sich in den Anfängen ihres Informatikstudiums befinden. Diese sollen das objektorientierte Programmieren aus der Design-Perspektive erlernen. UML-Diagramme wurden als die ideale Lösung für dieses Ziel gesehen.

Das generierte Klassendiagramm und der Java-Code werden automatisch synchronisiert. Es können somit Änderungen sowohl am Quellcode als auch am Diagramm vorgenommen werden, die dann bei dem jeweils anderen synchronisiert werden. Zu beachten ist, dass Green ein Universitätsprojekt ist und daher nicht die Qualität einer kommerziellen Software aufweist. Die Entwickler sprechen jedoch von einem kontinuierlichen Prozess. Green solle stets weiterentwickelt werden, damit es dem stetigen Weiterentwicklungsprozess von Java und der UML angepasst werden kann. Die Anwendung ähnelt sehr dem Prinzip von ObjectAid, wie in Abbildung 2.4 dargestellt. Es wird ebenfalls das Drag'n'Drop-Prinzip verwendet. Ein automatisches Layout wird, wie auch bei ObjectAid, nicht unterstützt.

[5] http://www.objectaid.com

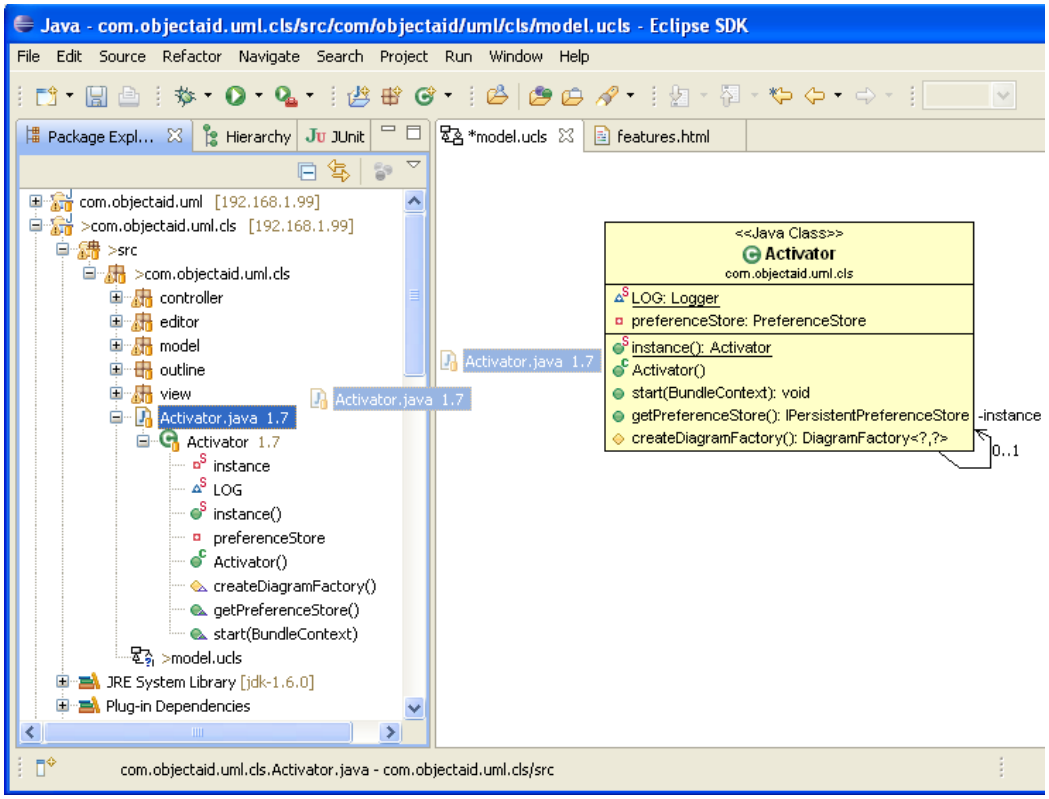


Abbildung 2.4. Prinzip zur Erstellung eines Klassendiagramms mithilfe von ObjectAid[6]

2.2. Grafisches Debugging mittels KLighD

Die Arbeit von Wißmann [Wiß13] beschäftigt sich mit der Darstellung von Java-Variablen zur Laufzeit. Die Darstellung kann zum Debugging benutzt werden. Beim Debugging werden Instanzen von Klassen beziehungsweise Modellen betrachtet, welche nach der Meta Object Facility (MOF)-Architektur (Abbildung 2.5) der Ebene M0 zugeordnet werden. Diese Arbeit beschäftigt sich mit der Darstellung von Klassen in UML-Klassendiagrammen. Es handelt sich dabei um Modelle. Modelle werden der Ebene M1 zugeordnet. Modellierungssprachen um solche Modelle zu erstellen, wie zum Beispiel die UML, werden durch Metamodelle beschrieben, welche der Ebene M2 angehören. Es wird eine weitere Ebene benötigt, welche die Meta-Metamodelle beinhaltet. Meta-Metamodelle bieten die Möglichkeit Metamodelle zu beschreiben, aber sie müssen auch durch sich selbst beschreibbar sein, um die MOF-Architektur zu begrenzen. Die MOF selbst gehört zum Beispiel zur Ebene M3.

[6] http://www.objectaid.com/class-diagram

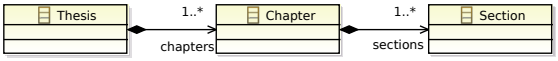
M3	Meta-Metamodell	MOF, Kermeta, KM3, Ecore
M2	Metamodell	UML, Petri nets, Xtext, DSLs
M1	Modell	
M0	Modellinstanzen	

Abbildung 2.5. Die Model Object Facility Architektur (Vorlage aus [Sch11])

Es lässt sich also ein enger Zusammenhang zwischen dieser Arbeit und der von Wißmann erkennen. In beiden Arbeiten wird die Sicht auf ein System auf einer anderen Ebene visualisiert.

Ein weiterer Zusammenhang zwischen den beiden Arbeiten lässt sich in der Technologie zur Visualisierung erkennen. Beide Arbeiten verwenden KLighD, um entsprechend das gewünschte Modell beziehungsweise die gewünschten Instanzen eines Modells darzustellen.

2.3. Layout von Klassendiagrammen

Da die UML-Spezifikation[7] keine Anforderungen an das Layout eines Klassendiagramms stellt, muss dieses Thema gesondert betrachtet werden. Speziell zu dem Thema Layout von Klassendiagrammen gibt es eine Publikation von Eichelberger [Eic02]. In dieser Publikation werden die Kriterien für das Layout beschrieben, die zu beachten sind, wenn ein verständliches und übersichtliches Diagramm gewünscht ist. Die wichtigsten Kriterien betreffen die Kanten in den Diagrammen. Bei Kanten muss zum einen auf eine Minimierung der sich überschneidenden Kanten geachtet werden, zum anderen sollte eine einheitliche Richtung gerichteter Kanten und Orthogonalität erreicht werden. Außerdem ist eine angemessene Platzierung von Labels, die Kanten zugewiesen sind, zu beachten. Zuletzt sollte auf eine übersichtliche Platzierung von Knoten geachtet werden, insbesondere von zueinander in Beziehung stehenden Knoten. Knoten sollten sich nicht gegenseitig überlappen.

In Abbildung 2.6 ist ein Beispiel-Klassendiagramm zu sehen, welches verschiedene Segmente der Modellierungssprache für Klassendiagramme enthält. Dieses

[7] http://www.omg.org

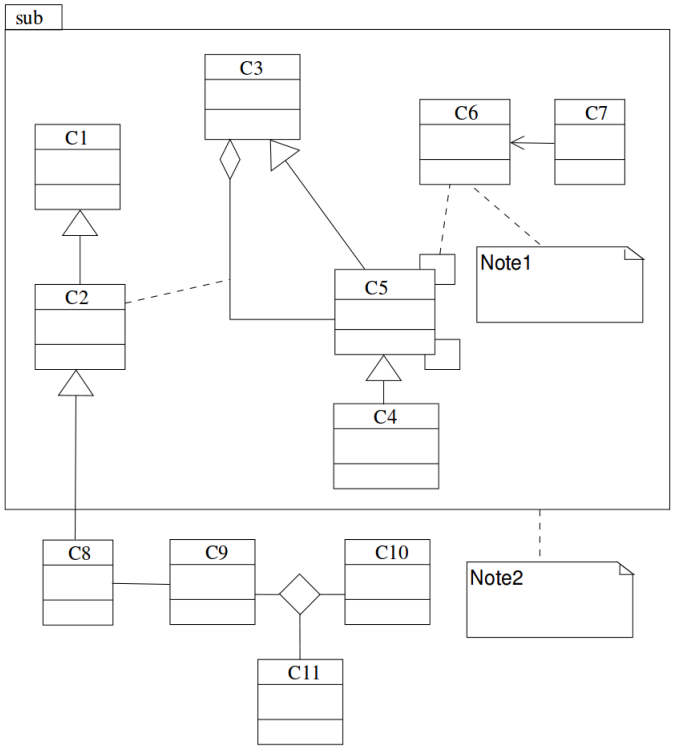


Abbildung 2.6. Ein Beispiel-Klassendiagramm für Layouting mit Constraints [Eic02]

Diagramm ist nach den zuvor beschriebenen Kriterien angeordnet. Abbildung 2.7 zeigt wiederum das gleiche Klassendiagramm, auf welches ein Layout angewendet wurde, das nicht die oben genannten *Constraints* berücksichtigt.

Des Weiteren wird in der Publikation von Eichelberger [Eic02] eine Implementierung eines Layoutingalgorithmus vorgeschlagen.

2.3.1. Evolutionäres Layout

Es ist meistens sehr schwierig, einen Layoutalgorithmus mit entsprechenden Einstellungen zu finden, der ein gewünschtes Layout erreicht. Deshalb können evolutionäre Algorithmen verwendet werden, um ein besseres Layout zu finden. Hierzu wird eine *Fitness Function* definiert. Die Fitness Function hilft dabei, die Diagramme zu bewerten und dann zu vergleichen. Die erzeugten Diagramme können dann durch Mutationen der einzelnen Parameter, wie Knotenposition oder die Position des Ports eines Knotens, bessere Werte von der Fitness Function erhalten. [Gud+06]

Besonders interessant für eine mögliche Erweiterung ist die Arbeit von Spönemann et al. [SDH14]. Eine evolutionäre Methode für das Layouting wurde in KIELER implementiert. Es werden dabei schon bestehende Algorithmen verwen-

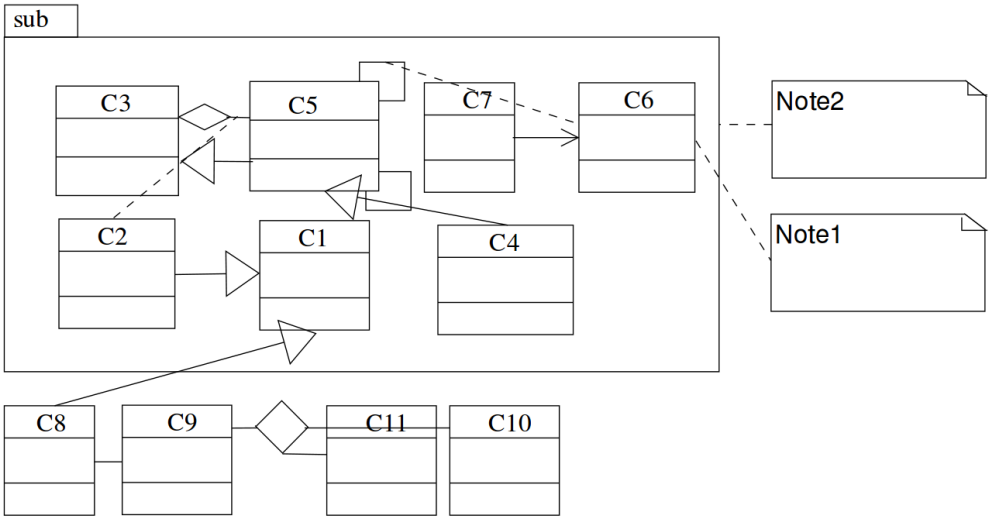


Abbildung 2.7. Beispiel für schlechtes Layout des Diagramms in Abbildung 2.6 [Eic02]

det und nach dem evolutionären Prinzip angepasst. Vom Benutzer wird mittels Schiebereglern für die verschiedenen Kriterien eine Fitness Function erzeugt.

Da KLighD als Layouting-Werkzeug für diese Arbeit verwendet wurde, könnte der evolutionäre Algorithmus verwendet werden, um das Layout der erzeugten Klassendiagramme zu verbessern.

In meiner Arbeit werde ich das Layout-Problem nicht ausführlicher behandeln, sondern es wird stattdessen ein Planarisierungs-Algorithmus oder ein hierarchischer Layoutalgorithmus für das Layouting verwendet. Eine Verknüpfung mit den oben erwähnten Implementierungen wäre jedoch eine interessante Erweiterungsmöglichkeit, um noch hochwertigere Diagramme zu erhalten.

Kapitel 3

Entwurf

In diesem Kapitel werden zunächst verschiedene Anforderungen, an die in dieser Arbeit entwickelte Software, zur Generierung von UML-Klassendiagrammen aufgezeigt. Danach werden die verschiedenen Aspekte des umgesetzten Softwaredesigns erläutert.

3.1. Anforderungen

Es soll eine Software entwickelt werden, die in eine vom Entwickler benutzte Entwicklungsumgebung als Erweiterung eingebunden werden kann. Die Anforderungen an diese Software lassen sich in drei Gruppen einteilen. Es gibt Anforderungen an das aus dem Syntheseprozess resultierende Klassendiagramm. Die Anforderungen an das Resultat folgen zu großen Teilen aus der Spezifikation der UML von der OMG.[1] Des Weiteren gibt es Anforderungen an die GUI durch deren Benutzung ein automatischer Syntheseprozess initiiert wird. Zuletzt müssen bestimmte Anforderungen an die direkten Schnittstellen der Erweiterung gestellt werden.

3.1.1. Anforderungen an die resultierenden Klassendiagramme

Im Folgenden werden Anforderungen an die Darstellung von Klassen, Attributen, Methoden und verschiedenen Kanten erläutert.

Knotentypen

Für die Visualisierung der verschiedenen Diagrammelemente ist es wichtig zu erkennen, welchen Typs eine Knoten ist. Es kann sich um eine konkrete Klasse, eine abstrakte Klasse, ein Interface oder eine Enumeration handeln. Dies soll durch verschiedene Identifikatoren, wie zum Beispiel „«Interface»" über dem Namen oder durch eine entsprechende Farbwahl für den Knoten, ausgezeichnet werden. Außerdem können Klassen bei Java auch innerhalb anderer Klassen liegen, was entsprechend gekennzeichnet werden soll.

[1] http://www.omg.org

Attribute und Methoden

Zu Klassen können beliebig viele Attribute und Methoden gehören. Diese sollen in zwei separaten Feldern unterhalb des Klassennamens linksbündig dargestellt werden.

Die Attribute haben jeweils einen Typ, einen Namen und eine Sichtbarkeit, wobei die Sichtbarkeit sich auf die Eigenschaften, `private` („–"), `public` („+") und `protected` („#"), beziehen.

Die Methoden zeichnen sich durch ihren Namen, ihre Parametertypen und die dazugehörigen Parameternamen, den Rückgabetyp und ebenfalls ihre Sichtbarkeiten aus.

Vererbung

Zwischen verschiedenen Knoten können Vererbungsbeziehungen bestehen, welche ebenfalls erkannt und visualisiert werden müssen.

Assoziationen

Außerdem gibt es Assoziationen, die beschreiben, ob eine Klasse eine Instanz einer anderen Klasse, die auch visualisiert wird, als Attribut enthält. Sollte dies der Fall sein, wird zwischen beiden Klassen eine Assoziation dargestellt.

Durch Multiplizitäten an den Kanten wird signalisiert, wie viele Instanzen einer anderen Klasse enthalten sind.

Sollte der Typ eines Attributs Liste, Map oder ein anderer generischer Typ sein, muss überprüft werden, ob einer der Typparameter durch eine der visualisierten Klassen parametrisiert wird. Ist dies der Fall, muss es entsprechend ausgezeichnet werden.

Ebenfalls können Assoziationen auch voll qualifiziert dargestellt werden, sodass eindeutig visualisiert wird, welchem Attribut die Assoziation dem Namen nach entspricht.

Pakethierarchie

Zuletzt könnte eine Darstellung der Pakethierarchie von Nutzen sein, um dem Diagramm weitere Struktur zu verleihen oder auch Strukturschwächen im Quellcode zu erkennen. Hierbei sollen auch verschachtelte Hierarchien beachtet werden, wenn ein Paket ein Unterpaket eines anderen Pakets ist.

3.1.2. Anforderungen an die grafische Benutzeroberfläche

Als Nächstes werden alle gewünschten Interaktionsmöglichkeiten aufgezeigt, die der Benutzer mit der Software haben soll.

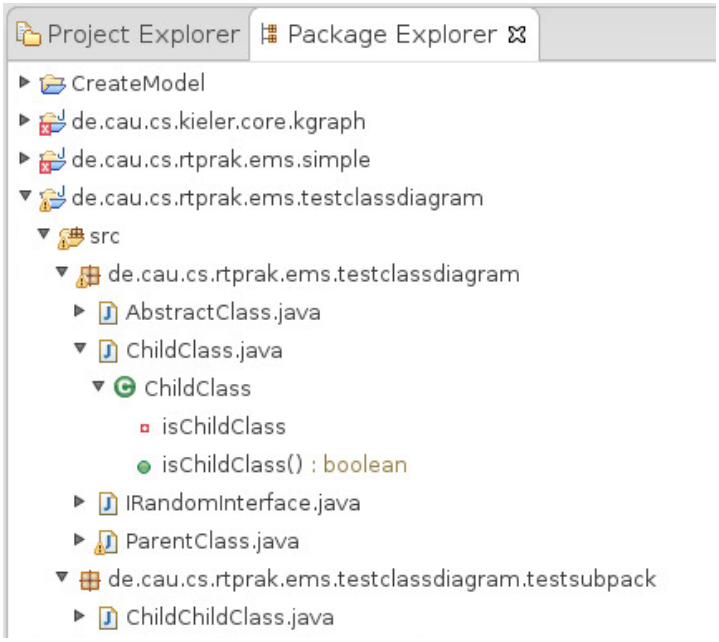


Abbildung 3.1. Sicht auf ein Java-Projekt mithilfe des Package Explorers

Die Auswahl, welche Elemente in dem Klassendiagramm visualisiert werden sollen, wird in einem Package oder Project Explorer vorgenommen. Dieser zeichnet sich meist dadurch aus, dass betrachtete Projekte mit ihren verschiedenen Klassen in einer Baumstruktur dargestellt werden. Ein Beispiel für solch einen Explorer ist in Abbildung 3.1 zu erkennen. Es muss möglich sein, die einzelnen Klassen zu inspizieren oder zur Bearbeitung auszuwählen. Wichtig ist für die hier zu erstellende Software, dass auch die Attribute und Methoden, welche zu den einzelnen Klassen gehören, ausgewählt werden können, damit diese für die Visualisierung jeweils an- oder abgewählt werden können.

Wenn nun eine Auswahl erstellt wurde, soll über das Kontextmenü ein Syntheseprozess gestartet werden. In Abbildung 3.2 ist dies zu sehen. Die Synthese des Klassendiagramms basiert auf der zuvor erstellten Auswahl von Klassen, Methoden und Attributen.

Weiterhin sollte es möglich sein zuvor erstellte Selektionen wiederherzustellen. Somit muss es eine Möglichkeit geben, Selektionen abzuspeichern. In Abbildung 3.2 ist die Funktion „Get Previous Selection Of Project" zu sehen, welche die Selektion des Projekts, die bei der letzten Synthese getroffen wurde, wiederherstellt.

Während der Synthese müssen alle gewünschten Diagrammelemente erzeugt und angeordnet werden. Ist ein Klassendiagramm synthetisiert worden, sollte das Diagramm durch Synthese- und Layoutoptionen anpassbar sein. Eine Einführung zu den Synthese- und Layoutoptionen ist in Kapitel 4.2 zu finden.

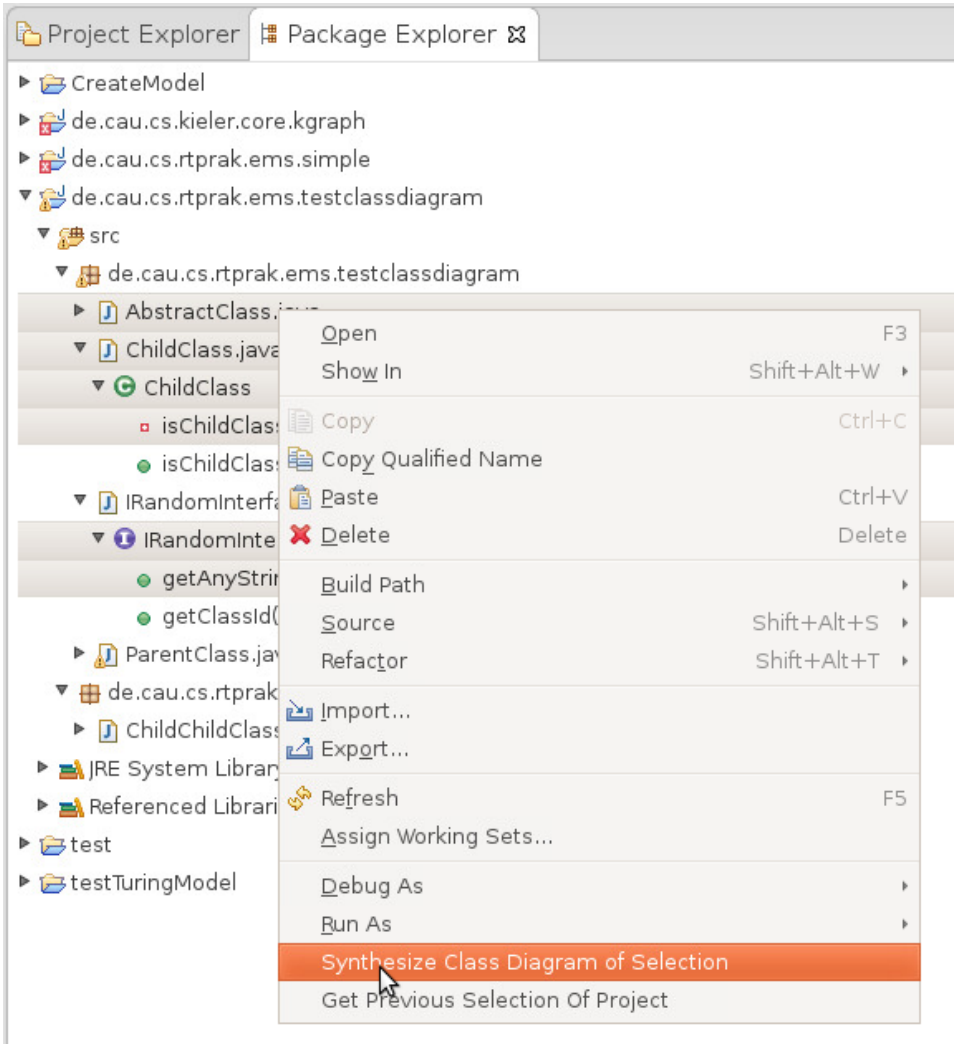


Abbildung 3.2. Start der Diagrammsynthese über das Kontextmenü

3.1.3. Anforderungen an die Schnittstellen

Um korrekte Klassendiagramme zu erhalten, wird schon bestehende Software benutzt und es wird auf Basis von schon bestehenden Strukturen gearbeitet. So wird vorausgesetzt, dass die selektierten Elemente im Package oder Project Explorer einer gewissen Struktur unterliegen. Die Struktur könnte ähnlich der in Abbildung 3.3 abgebildeten Struktur aussehen. `IMember` steht hierbei für Java-Elemente, die Angehörige einer Klasse sein können. Dies sind einerseits Attribute und Methoden, aber auch innere Klassen können Angehörige anderer Klassen sein. `IType` modelliert Klassen jeglicher Art, wie an den letzten 4 Methoden zu sehen ist. Diese Methoden identifizieren, um was für eine Klasse es sich handelt. Außerdem muss die Klasse `IType` zum Beispiel auch Methoden bereitstellen, mit

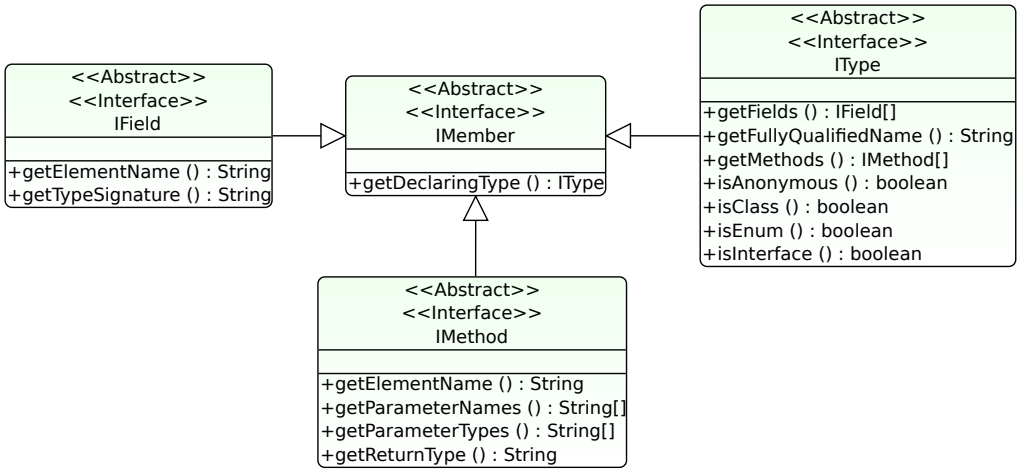


Abbildung 3.3. Struktur der Elemente im Package oder Project Explorer

denen Attribute und Methoden von Klassen ermittelt werden können (1. und 3. Methode von `IType`).

`IField` und `IMethod`, welche entsprechend Attribute und Methoden repräsentieren, müssen wiederum Funktionen bereitstellen, damit eindeutig bestimmt werden kann, um welches Attribut (bzw. Methode) es sich handelt. Es wird keine direkte Referenz von Methoden und Attributen zu ihrer Klasse benötigt, da sie ihr durch die Baumstruktur zugeordnet sind. In der Abbildung 3.3 sind die wichtigsten Funktionen visualisiert, welche die Grundfunktionen für eine gewünschte Synthese bieten. Es werden jedoch noch weitere Funktionen benötigt, um zum Beispiel Vererbung oder Assoziationen zu ermitteln. Diese werden in diesem Kapitel jedoch nicht näher betrachtet. Im Kapitel Verwendete Technologien werden die Java Development Tools (JDT) vorgestellt, welche die in Abbildung 3.3 dargestellten Funktionen bereitstellen und noch weit umfassendere Strukturen und Funktionen beinhalten.

Weiterhin wird eine Software beziehungsweise ein Eclipse-Plug-in benötigt, welches ein automatisches Layout auf Knoten-Kanten-Diagramme anwenden kann.

3.2. Designentscheidung

In diesem Teilkapitel werden die Konzepte, die zur Umsetzung der Anforderungen benutzt wurden, erläutert.

Dazu wird zunächst dargelegt, wie die Anforderungen an die resultierenden Klassendiagramme umgesetzt werden. Daraufhin wird ein Metamodell entwickelt, in welches die Selektionen, die im Package oder Project Explorer getätigt wurden,

überführt werden sollen. Die abstrakten Anforderungen an die Schnittstellen der Erweiterung werden im Kapitel 4 konkret gelöst.

3.2.1. Darstellung von Diagrammelementen

Um die Anforderungen an das resultierende Klassendiagramm zu erfüllen, sollen die Diagrammelemente folgendermaßen dargestellt werden.

Die verschiedenen Knotentypen sollen unterscheidbar sein. So wird bei allen Knoten ein linearer Farbverlauf von einer dezenten Farbe zu Weiß verwendet. Diese Option ist deaktivierbar, falls kein Farbverlauf gewünscht ist. Knoten werden in Rechtecken mit abgerundeten Ecken dargestellt, um ein optisch ansprechenderes Diagramm zu generieren. Bei konkreten Klassen wird die Quellfarbe *Light Sky Blue 2* verwendet. Sie ist dezent und ähnelt der Farbwahl von Visual Paradigm.[2] Bei vielen Werkzeugen, wie auch bei Visual Paradigm, wird eine kräftige Farbe ohne Farbverlauf verwendet, sodass es optisch sehr kontrastreich ist. Für Interfaces wurde als Quellfarbe *Honeydew*, eine Farbe, welche hauptsächlich grün enthält, benutzt. Für Enumerations wurde ein leichtes Lila als Quellfarbe gewählt. Weiterhin werden die verschiedenen Knotentypen durch die Identifikatoren, „«Interface»" und „«Enumeration»", direkt über dem Klassennamen angeordnet, gekennzeichnet. Abstrakte Klassen oder explizit abstrakt ausgezeichnete Interfaces werden nur durch ein Textfeld „«Abstract»" über dem Klassennamen von nicht abstrakten abgegrenzt.

In Abbildung 3.4 ist eine Möglichkeit zu sehen, die verschiedenen Typen mittels Farbe und Text auszuzeichnen.

Attribute und Methoden werden mit all ihren Eigenschaften unterhalb des Klassennamens visualisiert. In Abbildung 3.5 ist zu sehen, wie die Informationen in einer Klasse angeordnet werden.

Vererbungsbeziehungen erfordern die Verwendung von verschiedenen Kantentypen. So wird die Vererbung von einem Interface zu einer Klasse durch eine gestrichelte Linie signalisiert. Bei allen anderen Vererbungsbeziehungen wird eine durchgezogene Linie verwendet. Die Pfeilspitze ist bei jeder Vererbung ein nicht ausgefülltes Dreieck. In Abbildung 3.6 sind die verschiedenen Vererbungsbeziehungen visualisiert.

Assoziationsbeziehungen mit ihren Multiplizitäten sind in Abbildung 3.7 zu sehen. In dieser Abbildung ist eine Person dargestellt, der zwei Arme, zwei Beine und kein bis viele Haare zugewiesen sind. Assoziationen werden mit normalen Pfeilspitzen dargestellt. Die Attribute werden, bei Visualisierung mit Assozia-

[2] `www.visual-paradigm.com`

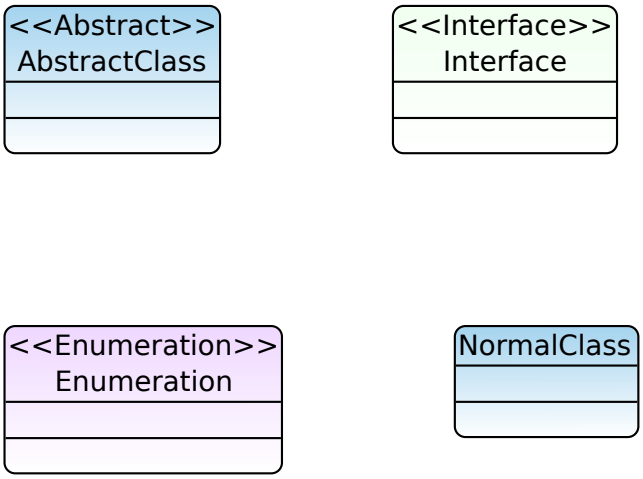


Abbildung 3.4. Darstellung von abstrakter Klasse, Interface, Enumeration und konkreter Klasse

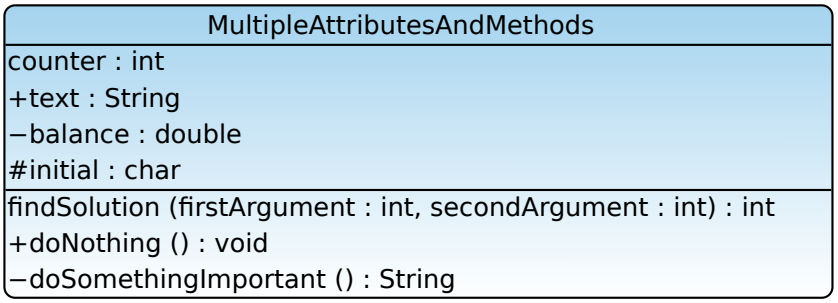


Abbildung 3.5. Darstellung einer Klasse, die Attribute und Methoden beinhaltet

tionen, nicht in der Klasse dargestellt. Sollten keine Assoziationsbeziehungen gewünscht sein, so sähe das Diagramm wie in Abbildung 3.8 aus. Voll qualifizierte Assoziationen werden hier zunächst nicht umgesetzt und können in zukünftigen Arbeiten umgesetzt werden.

Pakete werden zunächst nur durch eingegraute Flächen mit einem Farbverlauf zu weiß visualisiert. Auf den grauen Flächen befinden sich dann die Klassen, die zu einem dem entsprechenden Paket gehören. Pakete, die anderen Paketen unterzuordnen sind, werden hier zunächst nur neben ihrer höheren Hierarchieebene dargestellt.

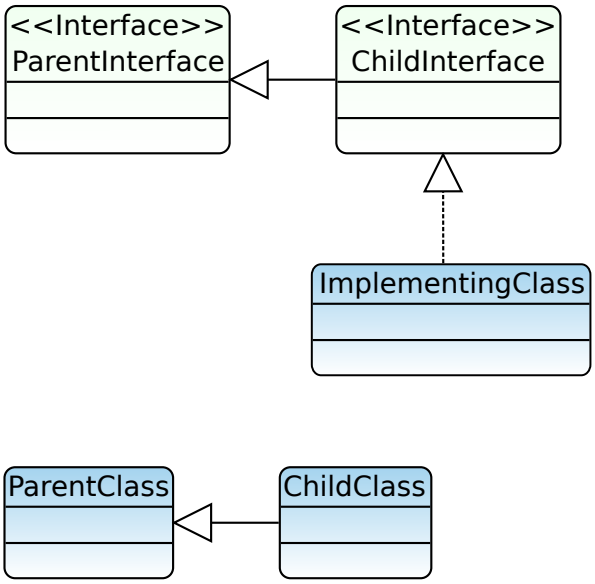


Abbildung 3.6. Darstellung der verschiedenen Vererbungsbeziehung

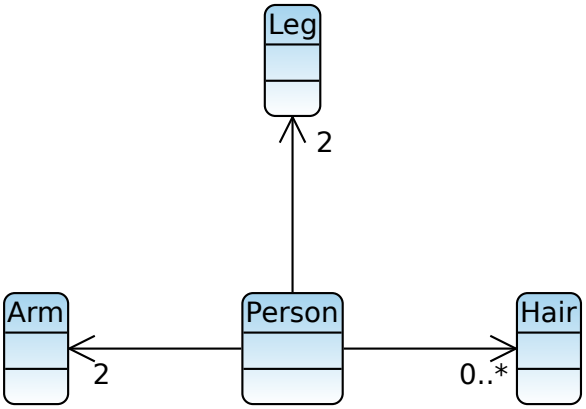


Abbildung 3.7. Darstellung von Klassen mit Assoziationen

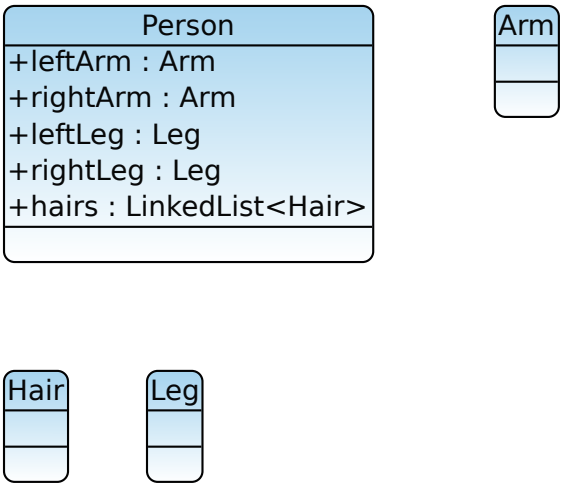


Abbildung 3.8. Darstellung der Klassen aus Abbildung 3.7 ohne Assoziationen

3.2.2. Metamodell für Selektionen

Um eine Selektion einfach synthetisieren zu können und sie auch, wie in den Anforderungen gefordert, abspeichern zu können, ist es hilfreich, ein Metamodell zu entwickeln. Aus diesem Metamodell lässt sich automatisch Implementierungscode generieren. Es kann dann mit den erzeugten Modellen einfach weitergearbeitet und die Abspeicherung ist durch die einheitliche Struktur vereinfacht.

Es werden nun verschiedene Entwürfe diskutiert, aus denen dann ein Endergebnis folgt, welches in der Implementierung verwendet wird.

Entwürfe

Der Ausgangspunkt ist eine Selektion von Klassen. Klassen beinhalten Attribute und Methoden. Es müssen jeweils die Daten, die zu einer Klasse, einem Attribut oder einer Methode gehören, im Modell enthalten sein. In Abbildung 3.9 wird dies entsprechend durch `typeData`, `fieldData` und `methodData` dargestellt. Diese Daten, die dann im Modell enthalten sind, umfassen sehr viele Informationen, die zum Abspeichern nicht praktisch sind, da sie nicht einfach serialisierbar sind.

Es würde ausreichen, für jede Klasse den voll qualifizierten Namen und für die entsprechenden Methoden und Attribute nur einen Namen abzuspeichern, denn beim Wiederherstellen einer Selektion können die entsprechenden Elemente anhand der abgespeicherten Namen identifiziert werden. Des Weiteren sollen auch nicht-selektierte Methoden und Attribute abgespeichert werden, um eine spätere Modifizierung der Selektion über einen Editor möglich zu machen. Dies wird durch einen Wahrheitswert ermöglicht, der bei der Auswahl des entsprechenden

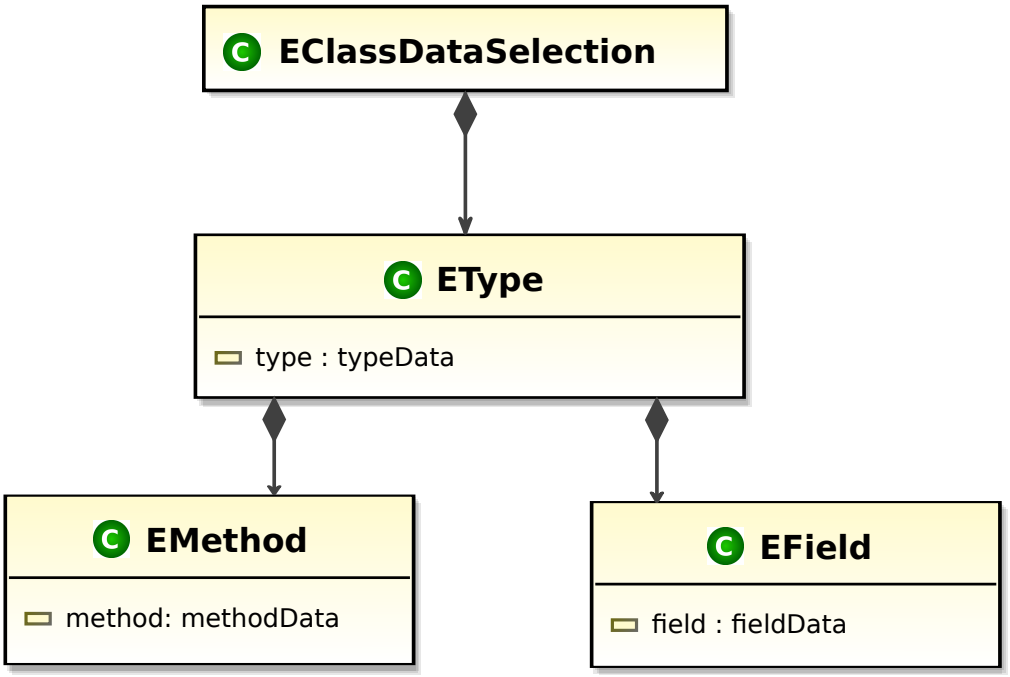


Abbildung 3.9. Erster Entwurf eines Metamodells für Selektionen

Elements auf `true` gesetzt wird. In Abbildung 3.10 ist das modifizierte Metamodell dargestellt.

Endergebnis

In Java können Methoden unter demselben Namen definiert werden, solange sich ihre Signatur unterscheidet. Dies nennt sich „Überladen von Methoden". Wenn nun eine Selektion serialisiert wird und danach wieder deserialisiert wird, kann nur durch den Namen nicht eindeutig festgestellt werden, um welche Methode es sich handelt. Deshalb müssen ebenfalls die Parameter abgespeichert werden. Aus dieser Schlussfolgerung ergibt sich als Resultat das Metamodell in Abbildung 3.11.

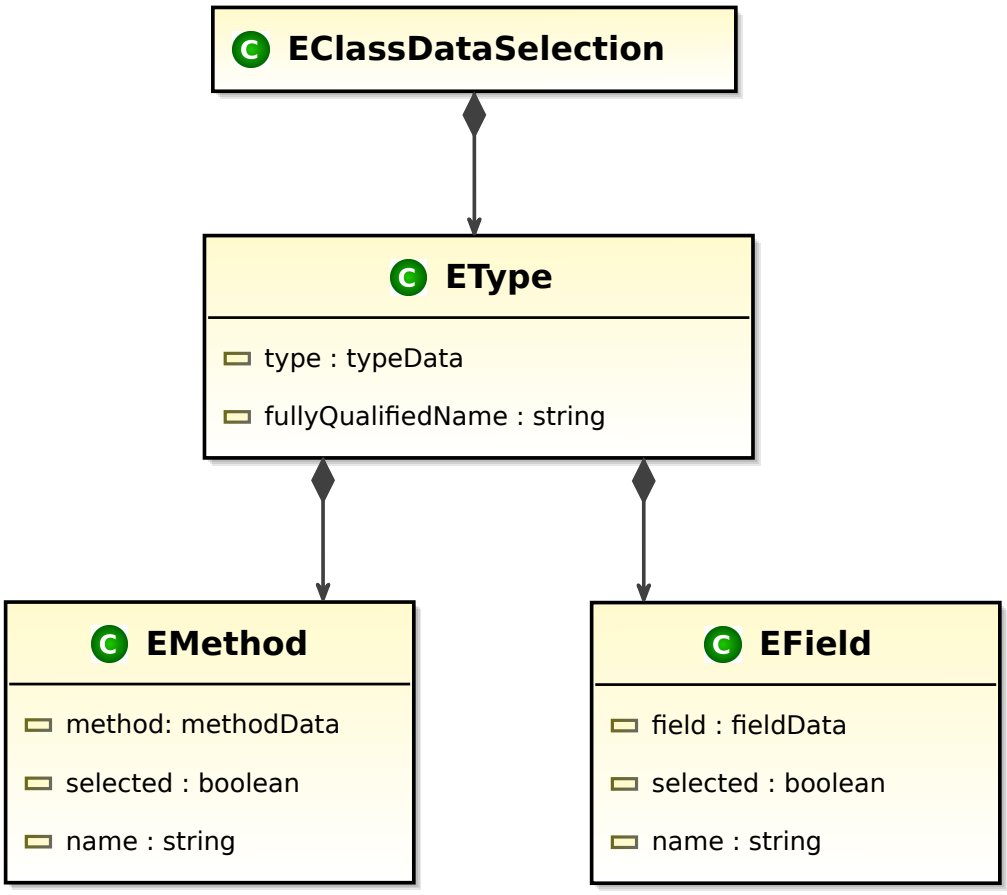


Abbildung 3.10. Modifizierter Entwurf eines Metamodells für Selektionen

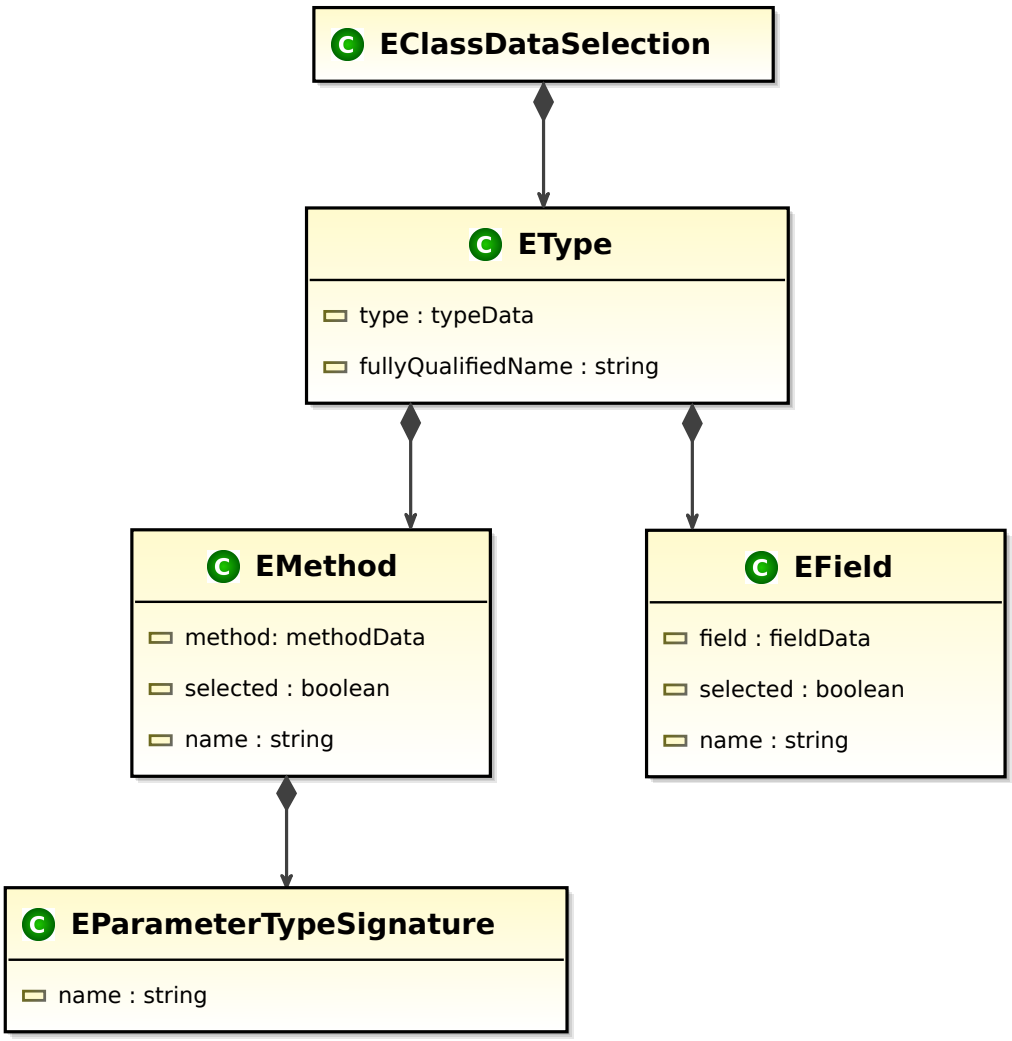


Abbildung 3.11. Entgültiger Entwurf des Metamodells für Selektionen

Kapitel 4

Verwendete Technologien

Im folgenden Kapitel werden die in dieser Arbeit verwendeten Technologien vorgestellt. Hierzu gehören zum einen das Eclipse Projekt und außerdem ein Teil des KIELER-Projekts, KIELER Lightweight Diagrams, welches am Lehrstuhl Echtzeitsysteme und Eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel entwickelt wurde.

4.1. Das Eclipse Projekt

Das Eclipse-Projekt ist ein quelloffenes Projekt, welches von IBM im Jahr 2001 ins Leben gerufen wurde. Eclipse wird meistens im Zusammenhang mit der Java-Entwicklung genannt, bietet jedoch viele weitere Funktionen. Eclipse ist nach der OSGi-Architektur[1] aufgebaut. Diese Architektur wurde entwickelt, damit neue Applikationen aus verschiedenen wiederverwendbaren Komponenten (*Bundles*) zusammengesetzt werden können. So ist die Eclipse-Plattform aus einer minimalen Menge von Bundles zusammengesetzt, die in Eclipse als Plug-ins bezeichnet werden. Durch den auf Komponenten basierenden Aufbau wird Eclipse als Rich Client Platform (RCP) bezeichnet. Es können auf Eclipse basierende Rich Client Applications (RCAs) entwickelt werden. Eine dieser auf Eclipse basierenden Anwendungen ist der Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). Dieser Client wird zum grafischen Modellentwurf von komplexen Systemen benutzt.

Die Plug-in-Architektur von Eclipse erlaubt es, die IDE um Funktionen zu erweitern. Damit eine Kommunikation zwischen Plug-ins möglich ist, gibt es *Extension Points* und *Extensions*. Mit den Extension Points wird festgelegt, an welchen Stellen ein Plug-in andere Plug-ins um Funktionen erweitert. Durch das Hinzufügen einer Extension wird festgelegt, an welchen Stellen ein Plug-in von anderen Plug-ins erweitert werden kann.

In Abbildung 4.1 ist eine Übersicht der Eclipse-Plattform zu sehen. Es wird hierbei zwischen der *Eclipse Platform*, der *Programming IDE* und der *Modeling Environment* unterschieden. Da man in Eclipse beliebig Plug-ins einbinden kann, ist nicht festgelegt, dass jede Eclipse-Instanz genau der in der Abbildung dar-

[1] http://www.osgi.org/

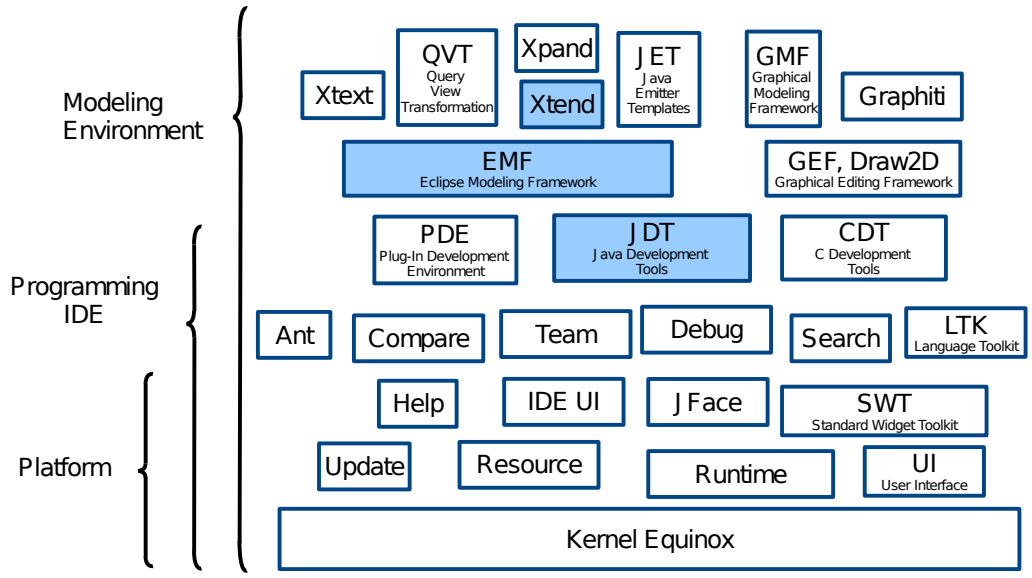


Abbildung 4.1. Übersicht der Eclipse Plattform (Vorlage aus [Fuh11])

gestellten Übersicht entspricht. Im Folgenden werden die blau hervorgehoben Plug-ins näher erläutert.

4.1.1. Java Development Tools

Die Plug-ins der Java Development Tools werden in die *Programming IDE* eingebunden. Sie stellen Funktionen zur Verfügung, welche Eclipse zu einer Entwicklungsumgebung für Java Applikationen macht. Durch die von den JDT bereitgestellten Plug-ins wird eine bestimmte Sicht auf Java-Projekte zur Verfügung gestellt. Des Weiteren wird die Java-Entwicklung durch verschiedene *Views*, *Editors* und weitere Werkzeuge erleichtert. So stellt das Plug-in `jdt.ui` beispielsweise den in dieser Arbeit benutzten Package Explorer bereit. Die JDT erfüllen alle Anforderungen die im Kapitel 3 an die *Low Level* Schnittstelle gestellt wurden. Die Strukturen der JDT stellen sogar noch weit mehr Möglichkeiten bereit, welche zum Beispiel zur Ermittlung von Vererbungs- und Assoziationsbeziehungen zwischen Knoten benötigt werden.

4.1.2. Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) wird zur *Modeling Environment* gezählt. Das Framework wird in drei fundamentale Teile unterteilt. Der Kern des Frameworks wird zur Beschreibung von Meta-Modellen benutzt. Auf diese Weise wird auch das Meta-Metamodell von der Sprache Ecore beschrieben, die zur

Beschreibung von Metamodellen verwendet wird. Ein weiterer Teil von EMF ist die Codegenerierung. Aus den entwickelten Meta-Modellen lässt sich Java-Code generieren. Mithilfe der *Factory Implementation Class* lassen sich dann Modelle erzeugen, die in XML Metadata Interchange (XMI) serialisierbar sind. Weiterhin bietet EMF die Möglichkeit, Code für einen Editor zu generieren, der die Modellerstellung mithilfe einer GUI erleichtert. Die Editoren basieren auf dem dritten Teil von EMF.

In dieser Arbeit wurde mithilfe von EMF ein Metamodell für Selektionen erstellt, aus dem dann Java-Code generiert wurde. Der Java-Code wurde zur Überführung von Selektionen des Package Explorers in ein Modell verwendet, welches in XMI abspeicherbar ist.

4.1.3. Xtend

Die Transformationssprache Xtend[2] ist eine Programmiersprache, welche in verständlichen Java-Code übersetzt wird. Xtend ähnelt Java zwar sehr, sie bietet jedoch zusätzliche Sprachkonstrukte, die oft als „syntaktischer Zucker" bezeichnet werden. Die Transformation von Selektionen in Diagrammmodelle wurde in dieser Arbeit mit Xtend durchgeführt.

4.2. KIELER Lightweight Diagrams

KIELER Lightweight Diagrams (KLighD) ist ein Teil des KIELER Pragmatik-Bereichs. Es wird zur Erstellung und Anzeige von leichtgewichtigen Diagrammen benutzt. KLighD wurde für KIELER entwickelt, kann aber aufgrund der OSGi Spezifikation, die Eclipse anwendet, auch in eine normale Eclipse-Instanz eingebunden werden. Die in KLighD erstellten Diagrammelemente werden mithilfe automatischer Layoutalgorithmen angeordnet. So kann die Entwicklungszeit von Diagrammen signifikant verringert werden, da ein manuelles Anordnen nicht nötig ist. Das Betrachten des Diagramms ist ohne nennenswerte Verzögerung möglich. Des Weiteren erlaubt KLighD die Integration von MDSE-Konzepten und -Werkzeugen. KLighD ermöglicht zudem das Hinzufügen von Synthese- und Layoutoptionen, welche die Möglichkeit bieten, generelle Anpassungen wie die Ausrichtung der Kanten vorzunehmen. Hierbei wird jedoch stets Augenmerk auf automatisches Layout gelegt. Einzelne Diagrammelemente können mit den Optionen nicht explizit verschoben werden.

In der vorliegenden Arbeit wird KLighD als Visualisierungswerkzeug für die Klassendiagramme benutzt, um die Anforderungen an die *High Level*-Schnittstelle zu erfüllen.

[2] http://www.eclipse.org/xtend/

Kapitel 5

Implementierung

In diesem Kapitel wird aufbauend auf den Entwürfen aus Kapitel 3 die Implementierung eines Eclipse-Plug-ins vorgestellt. Hierzu werden zunächst Aufbau und Struktur beschrieben. Daraufhin werden die einzelnen Schritte, die zu einem gewünschten Klassendiagramm führen, anhand von Quellcode-Beispielen erläutert. Zuletzt werden die zur Verfügung gestellten Synthese- und Layoutoptionen vorgestellt.

5.1. Funktionalität und Struktur

Wie in den Anforderungen beschrieben, soll eine Funktion zur Verfügung gestellt werden, um eine manuell erstellte Selektion abzuspeichern und daraus ein Klassendiagramm zu erstellen. Weiterhin soll eine Funktion zur Verfügung gestellt werden, die eine abgespeicherte Selektion wiederherstellt und ebenfalls die Synthese des dazugehörigen Klassendiagramms durchführt. In Abbildung 5.1 ist die Struktur der Implementierung zu sehen. Der Pfad von „Neue Auswahl" entspricht dem Ablauf wenn eine Selektion manuell im Package oder Project Explorer erstellt wird und dann eine automatische Synthese gestartet wird. Die Selektion von Java-Elementen wird in ein Modell überführt, welches auf dem Metamodell basiert, das im Kapitel 3 entworfen wurde (Selektionsmodell). Das Modell wird abgespeichert, damit die Selektion wiederhergestellt werden kann. Daraufhin wird auf Basis des Modells die Synthese in Xtend durchgeführt (Synthese), deren Ergebnis dann mit KLighD visualisiert wird. Zuletzt wird dem Nutzer durch Synthese- und Layoutoptionen die Möglichkeit geboten, das Diagramm anzupassen (KLighD-View). In den Abbildungen 5.2 und 5.3 ist der Unterschied zu sehen, der durch die Veränderungen der Syntheseoptionen entsteht.

Der Pfad von „Auswahl wiederherstellen" beschreibt den Ablauf, der bei einer Wiederherstellung der Synthese durchgeführt wird. Auf Basis des aktuell selektierten Elements wird die letzte Selektion, die von dem Projekt synthetisiert wurde, wiederhergestellt (Selektionsmodell) und das dazugehörige Klassendiagramm erzeugt (Synthese).

Das Plug-in wurde in drei verschiedene Pakete aufgeteilt. Ein Paket beinhaltet die beiden *Handler* der neu hinzugefügten *Menu Contributions*, über welche die Kontextmenüeinträge zur Verfügung gestellt werden. In einem weiteren Paket

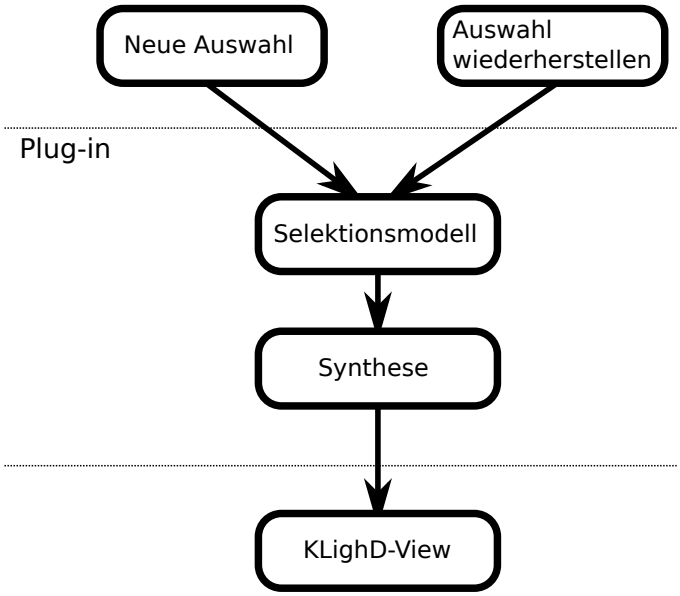


Abbildung 5.1. Struktur der Implementierung

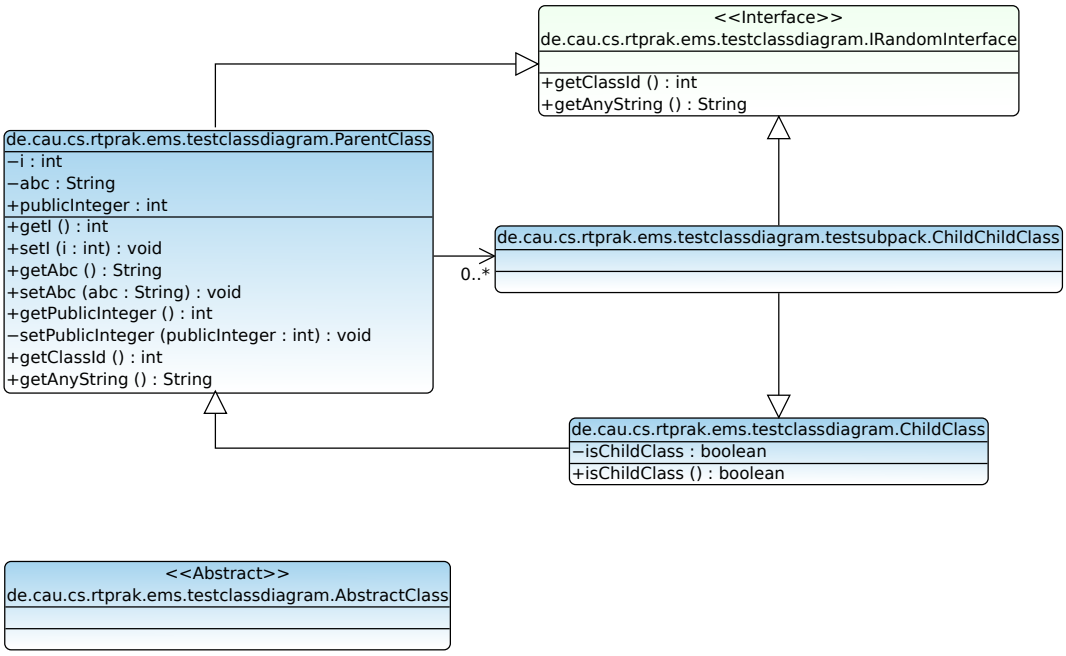


Abbildung 5.2. Ungefiltertes Klassendiagramm

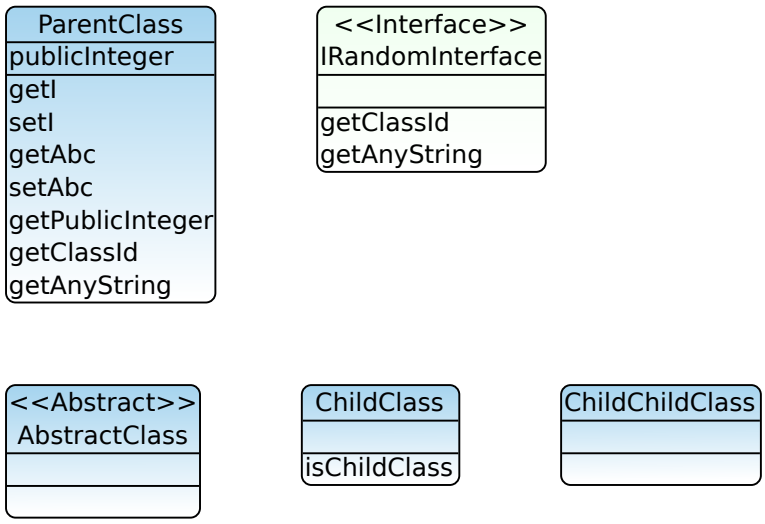


Abbildung 5.3. Gefiltertes Klassendiagramm

befindet sich das Metamodell für Selektionen, welches im Kapitel 3 entworfen wurde und dessen Implementierungscode. Im dritten Paket befindet sich die Klassendiagrammsynthese, die mithilfe einer Xtend-Klasse durchgeführt wird.

5.2. Überführung von Selektionen in Modelle

Nachdem eine Selektion vom Benutzer erstellt und die Synthese initiiert wurde, muss diese in ein Modell basierend auf dem Metamodell aus Kapitel 3 überführt werden. Alle selektierten Klassen werden mit ihrem voll qualifizierten Namen direkt in das Modell übernommen, da dieser zum späteren Abspeichern geeignet ist. Außerdem werden alle Attribute und Methoden der Klassen in das Modell übernommen. Mit ihnen wird ebenfalls ein Name, der zum Abspeichern benötigt wird, übernommen. Es wird jedoch auch für jedes Attribut und jede Methode ein *boolean* im Modell gespeichert, welcher initial *false* gesetzt wird, um zu signalisieren, dass das Attribut oder die Methode zurzeit als nicht selektiert festgelegt ist. So kann auch eine Visualisierung von nicht selektierten Attributen und Methoden durchgeführt werden. In Listing 5.1 ist zu sehen, wie eine `IType`-Instanz von den JDT in eine `EType`-Instanz des Metamodells überführt wird. Zunächst werden in den Zeilen 3 bis 5 die Daten der Klasse in eine `EType`-Instanz übernommen. In den Zeilen 8 bis 14 und den Zeilen 17 bis 30 werden jeweils alle Attribute und Methoden der `EType`-Instanz als `EField`-Instanz und `EMethod`-Instanz zugewiesen. Den Methoden (`EMethod`) werden entsprechend auch deren Parameter zur eindeutigen Identifizierung übergeben.

Sollte es sich bei einem Element um ein Attribut oder eine Methode handeln,

```
1  private EType createETypeOfType(IType type) throws JavaModelException {
2      // Extract type-data needed for the metamodel.
3      EType eType = factory.createEType();
4      eType.setType(type);
5      eType.setFullyQualifiedName(type.getFullyQualifiedName());
6      // Extract field-data needed for the metamodel, but initially set the
7      // selected-boolean 'false'.
8      for (int j = 0; j < type.getFields().length; j++) {
9          EField field = factory.createEField();
10         field.setField(type.getFields()[j]);
11         field.setName(type.getFields()[j].getElementName());
12         field.setSelected(false);
13         eType.getFields().add(field);
14     }
15     // Extract method-data needed for the metamodel, but initially set the
16     // selected-boolean 'false'.
17     for (int j = 0; j < type.getMethods().length; j++) {
18         EMethod method = factory.createEMethod();
19         method.setMethod(type.getMethods()[j]);
20         method.setName(type.getMethods()[j].getElementName());
21         for (int k = 0; k < type.getMethods()[j].getParameterTypes()
22                                                 .length; k++) {
23             EParameterTypeSignature signature = factory
24                     .createEParameterTypeSignature();
25             signature.setName(type.getMethods()[j].getParameterTypes()[k]);
26             method.getParameterTypeSignatures().add(signature);
27         }
28         method.setSelected(false);
29         eType.getMethods().add(method);
30     }
31     return eType;
32 }
```

Listing 5.1. Ermittung der Daten einer selektierten Klasse

wird nun, vorausgesetzt dessen Klasse ist selektiert worden, dessen `boolean` auf `true` gesetzt. In Listing 5.2 ist der Quellcode zu sehen, mit dem dies für jedes selektierte Attribut durchgeführt wird. Die `if`-Bedingung in den Zeilen 4 bis 8 überprüft, ob das Attribut, das dem `EType` hinzugefügt werden soll, überhaupt ein Kind der Klasse ist. Sollte dies nicht der Fall sein, wird das Attribut nicht hinzugefügt. Ist es Kind der Klasse, wird in den Zeilen 11 bis 17 das Attribut im `EType` gesucht und dessen `boolean-flag` auf `true` gesetzt.

Analog wird dies auch für jede selektierte Methode realisiert.

5.3. Abspeichern und Wiederherstellen der Selektion

Nachdem eine Selektion in ein Modell auf Basis des Metamodells überführt wurde, wird das Modell im XMI-Format abgespeichert. Beim Abspeichern gehen alle Informationen bis auf die voll qualifizierten Namen und die Hierarchien, die durch die Struktur des Metamodells gebildet werden, verloren. Später kann

```
for (int j = 0; j < classDataSelection.getTypes()
        .size(); j++) {
    // Find the class that contains this field.
    if (((IType) ((IField) je).getParent()))
            .getFullyQualifiedName().equals(
                    classDataSelection.getTypes()
                            .get(j)
                            .getFullyQualifiedName())) {
        // Get the index of the field
        // where it is in the model.
        int indexOfFieldInType = Arrays.asList(
                ((IType) je.getParent()).getFields())
                .indexOf(je);
        // Set it's boolean 'true'.
        classDataSelection.getTypes().get(j)
                .getFields().get(indexOfFieldInType)
                .setSelected(true);
    }
}
```

Listing 5.2. Booleans der selektierten Attribute *true* setzen

die Selektion dennoch, auf Basis des voll qualifizierten Namen der Klassen, wiederhergestellt werden.

Das Modell wird in den Metadaten des Plug-ins gespeichert. Dort wird ein Ordner für jedes Projekt erstellt, dessen Elemente selektiert sind. In jedem dieser Ordner wird das Modell unter dem Namen `selection.xmi` abgespeichert.

Zurzeit kann für jedes Projekt nur eine Selektion im entsprechenden Ordner existieren, wobei dies immer der zuletzt abgespeicherten entspricht. Um die Selektion wiederherzustellen, muss ein Element des gewünschten Projekts ausgewählt werden und „Get Previously Visualized Selection Of Project" im Kontextmenü ausgewählt werden. Die Daten, die beim Abspeichern verloren gegangen sind, werden dann anhand der voll qualifizierten Namen wiederhergestellt. Es wird hierbei nicht abgefangen, ob ein Element im Java-Code nicht mehr vorhanden ist. Wie beim manuellen Erstellen der Selektion wird eine Synthese durchgeführt.

5.4. Synthese mittels Xtend

Nachdem die Selektion abgespeichert oder wiederhergestellt wurde, wird die Diagrammsynthese durchgeführt. Diese wird mithilfe der Programmiersprache Xtend realisiert. Für jede Klasse, jedes Interface und jede Enumeration des Modells wird ein Knoten erstellt. Es werden dann der Typ und der Name als Textfelder hinzugefügt (Abbildung 3.4). Attribute werden nur hinzugefügt, wenn sie keine Assoziationsabhängigkeiten zu anderen visualisierten Klassen haben. Dies wurde bereits im Kapitel 3 in den Abbildungen 3.7 und 3.8 dargestellt. Außerdem ist zu beachten, ob die Attribute zuvor selektiert wurden. Listing 5.3 zeigt, wie die

```
1  def String getAttributeData(IField field) {
2      var String visibility, attributeType = ""
3      if (Flags.isPrivate(field.flags)) {
4          visibility = "\u2212"
5      } else if (Flags.isPublic(field.flags)) {
6          visibility = "+"
7      } else if (Flags.isProtected(field.flags)) {
8          visibility = "#"
9      }
10     attributeType = Signature.getSignatureSimpleName(field.typeSignature)
11     return visibility + field.elementName + " : " +  attributeType
12 }
```

Listing 5.3. Ermittlung von Attributdaten

Daten aus dem Attribut extrahiert werden. So basiert die Sichtbarkeit auf den *flags* des Attributs (Zeilen 3 bis 9). Der Typ wird durch dessen Typsignatur festgelegt (Zeile 10).

Methoden werden immer hinzugefügt, wenn sie selektiert wurden, da bei diesen keine Assoziationsabhängigkeiten bestehen können. Sie besitzen jedoch zusätzlich noch Parameter, welche entsprechend dem Modell entnommen werden. Der Aufbau des Klasseninhalts wurde im Kapitel 3 in Abbildung 3.5 dargestellt.

Bei der Ermittlung der Vererbungsbeziehungen werden der *SuperType* und die *SuperInterfaces* betrachtet. So werden für jeden Knoten, wie in Listing 5.4 zu sehen, dessen *superClass* und dessen direkte *superInterfaces* ermittelt (Zeilen 1 bis 3 und Zeilen 14 bis 16). Sollte einer der visualisierten Knoten eine *superClass* oder ein *superInterface* sein, werden entsprechende Vererbungskanten hinzugefügt (Zeilen 4 bis 12 und Zeilen 17 bis 32).

Um die Assoziationsbeziehungen zu ermitteln, muss für jedes Attribut jeder Klasse überprüft werden, ob dessen Typ einer anderen visualisierten Klasse entspricht. Außerdem muss überprüft werden, ob der Typ des Attributs eine Liste oder eine Map ist. Sollte dies der Fall sein, müssen die generischen Typparameter mit den anderen visualisierten Klassen verglichen werden. Auf diese Weise lassen sich die Multiplizitäten der Assoziationskanten berechnen. Sollte eine Klasse ein Attribut enthalten, das vom Typ einer anderen visualisierten Klasse ist, wird die maximale Multiplizität der zugehörigen Assoziation zu der anderen visualisierten Klasse, inkrementiert. Wenn eine Klasse eine Liste oder eine Map enthält, deren generischer Typparameter durch eine visualisierte Klasse parametrisiert wird, wird dies in den Multiplizitäten als „0..*" (keine bis viele) ausgezeichnet. Da ein Attribut einer Klasse stets den Wert *null* haben kann, bleibt die untere Grenze der Multiplizität 0. In Listing 5.5 wird zunächst überprüft, ob der Typ des Attributs *Collection* implementiert (Zeile 1). Sollte dies der Fall sein, wird der generische Typparameter geparst (Zeilen 2 bis 4) und dann mit der gerade betrachteten

```
1  val IType superClass = classData.type
2    .newSupertypeHierarchy(new NullProgressMonitor)
3    .getSuperclass(classData.type);
4  if (classDataToBeCompared.type == superClass) {
5      createEdge.putToLookUpWith(classData) => [
6          it.source = classData.node
7          it.target = classDataToBeCompared.node
8          it.addPolyline().putToLookUpWith(classData) => [
9              it.addInheritanceTriangleArrowDecorator()
10         ]
11     ]
12 }
13
14 val IType[] directSuperInterfaces = classData.type
15   .newSupertypeHierarchy(new NullProgressMonitor)
16   .getSuperInterfaces(classData.type);
17 if (directSuperInterfaces.contains(classDataToBeCompared.type)) {
18     createEdge.putToLookUpWith(classData) => [
19         it.source = classData.node
20         it.target = classDataToBeCompared.node
21         it.addPolyline().putToLookUpWith(classData) => [
22             if (classData.type.isInterface) {
23                 // if interface has super interface the line is solid.
24                 it.lineStyle = LineStyle::SOLID
25             } else {
26                 // if class has super interface the line is dashed.
27                 it.lineStyle = LineStyle::DASH
28             }
29             it.addInheritanceTriangleArrowDecorator()
30         ]
31     ]
32 }
```

Listing 5.4. Synthese von Vererbungsbeziehungen

```
1  if (Collection.isAssignableFrom(Class.forName(fieldTypeFQN))) {
2      var String genericTypeOfCollection = generic.substring(
3        generic.indexOf("<") + 1,
4          generic.lastIndexOf(">"))
5      if (genericTypeOfCollection.equals(
6          classDataToBeCompared.type.elementName)) {
7          // If the field's generic parameter type is the checked class,
8          // set the upper bound of the multiplicity to infinity (-1).
9          classHasAssociationToThisClass.set(1, -1)
10     }
11 }
```

Listing 5.5. Ermittlung von den Multiplizitäten bei Listen

Klasse verglichen. Bei Übereinstimmung wird die obere Grenze auf -1 gesetzt, was entsprechend „bis viele" bedeutet (Zeilen 5 bis 10).

5.5. Synthese- und Layoutoptionen

Mit Synthese- und Layoutoptionen lassen sich synthetisierte Diagramme nachträglich anpassen. In dem erstellten Plug-in wurden folgende, in Abbildung 5.4 dargestellte, Synthese- und Layoutoptionen verwendet. Syntheseoptionen werden dort als Diagrammoptionen bezeichnet.

Syntheseoptionen

Generell kann entschieden werden, ob ein Farbverlauf verwendet werden soll, die Pakethierarchie dargestellt werden soll und entweder die Selektion von Attributen und Methoden oder der gesamte Inhalt von den Klassen visualisiert werden soll. Weiterhin kann für Klassen entschieden werden, ob deren voll qualifizierter oder normaler Name angezeigt werden soll. Bei den Attributen können private Attribute trotz Selektion ausgeblendet werden. Des Weiteren können Typ und Sichtbarkeit gefiltert werden. Analog können diese Optionen auch bei Methoden ausgewählt werden und zusätzlich kann noch die Visualisierung von Parametern deaktiviert werden. Die Kanten für Vererbungs- und Assoziationsbeziehungen können ebenfalls deaktiviert werden. In Listing 5.6 werden einige Anpassungen im Quellcode dargestellt.

Layoutoptionen

Bei den Layoutoptionen können die Ausrichtung des Diagramms und der minimale Abstand zwischen verbundenen Knoten durch den Schieberegler *Spacing* bestimmt werden.

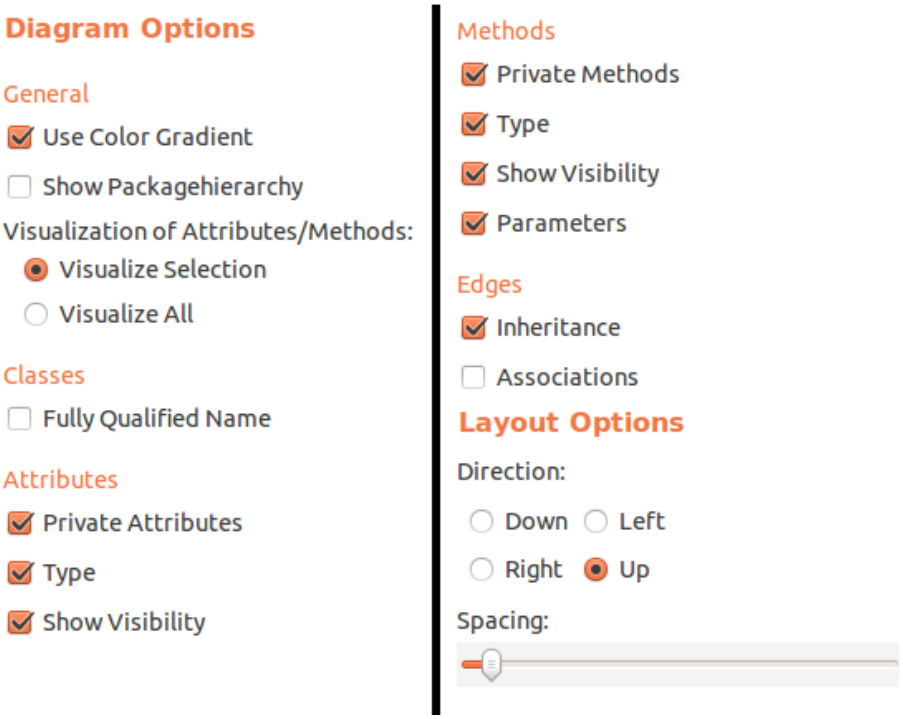


Abbildung 5.4. Synthese- und Layoutoptionen in KLighD

```
// If inheritance shall be visualized, create all inheritance edges.
if (EDGES_INHERITANCE.booleanValue) {
    model.createInheritanceEdges
}
...
// If Layout shall be with gradient, use gradient, else not.
if (COLOR_GRADIENT.booleanValue) {
    rect.setBackgroundGradient(getNodeColor(classData), "white".color, 90)
} else {
    rect.setBackground(getNodeColor(classData))
}
...
// if only selected fields shall be visualized only add selected fields.
// Else also add not selected fields.
if (VISUALIZE_ALL_OR_SELECTION.objectValue == VISUALIZE_SELECTION) {
    if (it.isSelected) {
        rect.addText(it.method.getMethodData) => [
            it.setHorizontalAlignment(H_LEFT)
        ]
        addedMethod.set(0, true)
    }
} else {
    rect.addText(it.method.getMethodData) => [
        it.setHorizontalAlignment(H_LEFT)
    ]
    addedMethod.set(0, true)
}
```

Listing 5.6. Einbindung von Syntheseoptionen

Kapitel 6

Anwendung und Evaluation

In diesem Kapitel wird die Klassendiagrammsynthese auf ein Java-Projekt angewendet, welches im Laufe eines Programmierpraktikums an der Christian-Albrechts-Universität zu Kiel entwickelt wurde. Daraufhin wird die Performance des entwickelten Plug-ins überprüft, indem die Klassendiagrammsynthese auf eine kleine, mittlere und große Anzahl von ausgewählten Elementen durchgeführt wird.

6.1. Synthese auf ein Projekt anwenden

Die Ergebnisse der Synthese auf kleinere Beispiele wurden bereits im Kapitel 3 präsentiert. Nun wird die Klassendiagrammsynthese auf ein größeres Softwareprojekt angewendet. Dabei handelt es sich um ein Projekt, welches während des Programmierpraktikums an der Christian-Albrechts-Universität zu Kiel entwickelt wurde.

6.1.1. Projekt: OceanLife

Eine erste Visualisierung aller Klassen basierend auf der Selektion, die in Abbildung 6.1 zu sehen ist, bildet das Klassendiagramm in Abbildung 6.2.

Die Syntheseoption „Visualize all" bewirkt, dass zusätzlich alle beinhalteten Attribute und Methoden visualisiert werden, sodass ein relativ großes Klassendiagramm erzeugt wird. KLighD ermöglicht jedoch ein komfortables Browsen durch das Diagramm, sodass in der Abbildung 6.3 ein Ausschnitt des Klassendiagramms fokussiert ist, welcher die wichtigsten Klassen des Projektes visualisiert.

Die Klasse `Ocean` hat eine Liste von Ozeanobjekten. Wie man sieht, ist `OceanObject` eine abstrakte Klasse, von der die konkreten Klassen `Fish`, `Bubble`, `Stone` und `Plant` erben. Ozeanobjekte zeichnen sich durch eine Position, ein Ziel und den Wahrheitswert, ob es entfernt werden muss, aus. Aus dem Diagramm lässt sich also schließen, dass ein Ozean kein bis viele Fische, Blasen, Steine und Pflanzen haben kann.

Es lassen sich also Klassendiagramme erzeugen, die nicht zu viel, aber dennoch hinreichend Informationen enthalten, sodass sie von einem Benutzer verarbeitbar sind.

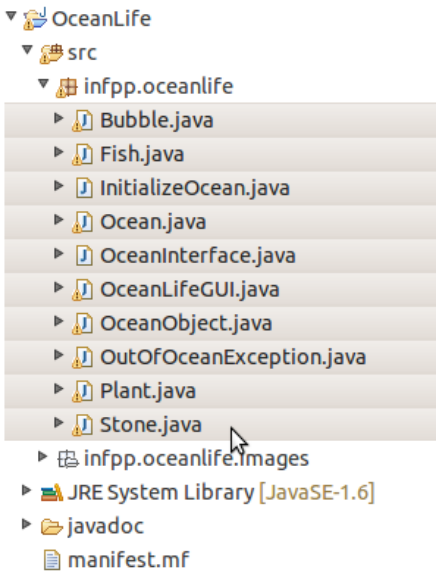


Abbildung 6.1. Selektion von Klassen des Projekts OceanLife

Tabelle 6.1. Testergebnisse der Performancetests

Messergebnis	1 Element	100 Elemente	1000 Elemente
1.	0,2 Sekunden	1,4 Sekunden	5,8 Sekunden
2.	0,2 Sekunden	1,5 Sekunden	5,1 Sekunden
3.	0,2 Sekunden	1,5 Sekunden	5,3 Sekunden

6.2. Performancetest

Es wurden außerdem Tests durchgeführt, um zu überprüfen wie viel Zeit die Synthese und die Visualisierung benötigen. Hierbei wurden als Eingaben ein Diagrammelement, 100 Diagrammelemente und 1.000 Diagrammelemente, also Klassen, Attribute und Methoden, verwendet. Jeder Test wurde drei Mal durchgeführt. In Tabelle 6.1 sind die Testergebnisse zu sehen.

Für ein Element wurde in allen Tests eine Dauer von 0,2 Sekunden gemessen. Bei 100 Elementen wurde eine durchschnittliche Dauer von 1,45 Sekunden gemessen. Die Tests für 1000 Elemente ergaben eine durschnittliche Dauer von 5,4 Sekunden.

Ein Diagramm, das 1.000 Elemente beinhaltet ist sehr unübersichtlich. 5,4 Sekunden sind daher eine vertretbare Zeit, da so große Diagramme wahrscheinlich nur selten erzeugt werden.

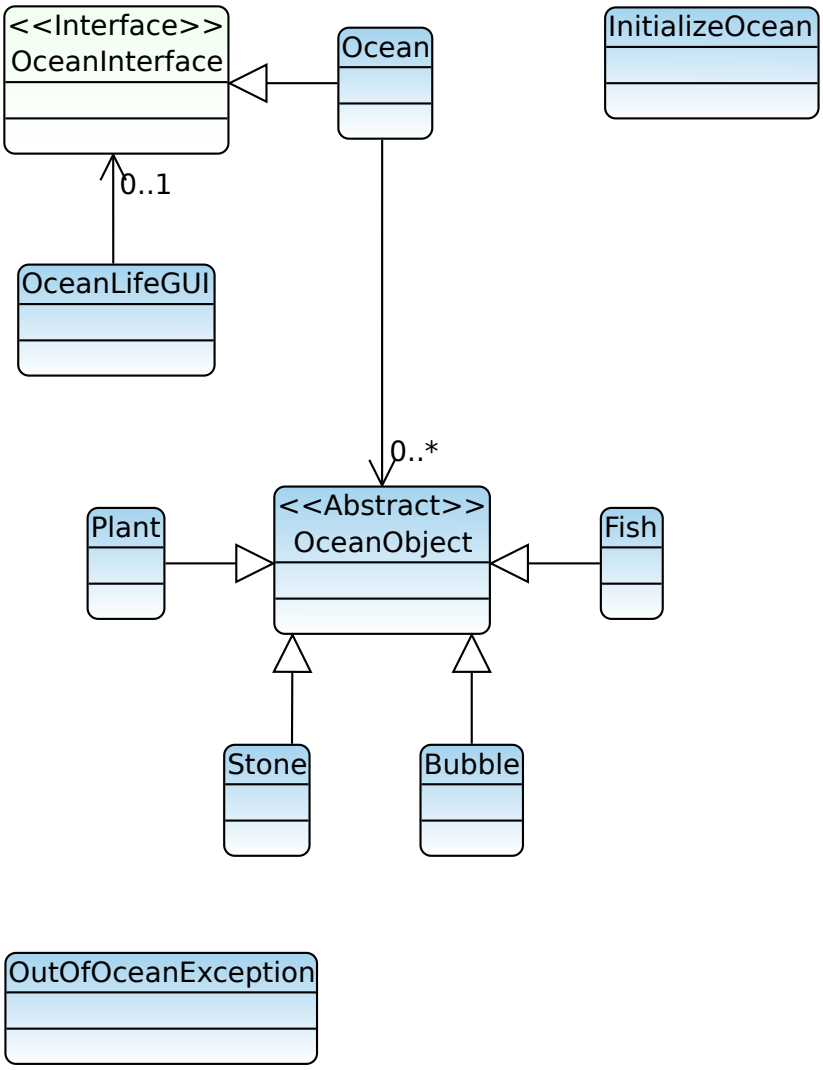


Abbildung 6.2. Kompaktes Klassendiagramm basierend auf der Selektion in Abbildung 6.1

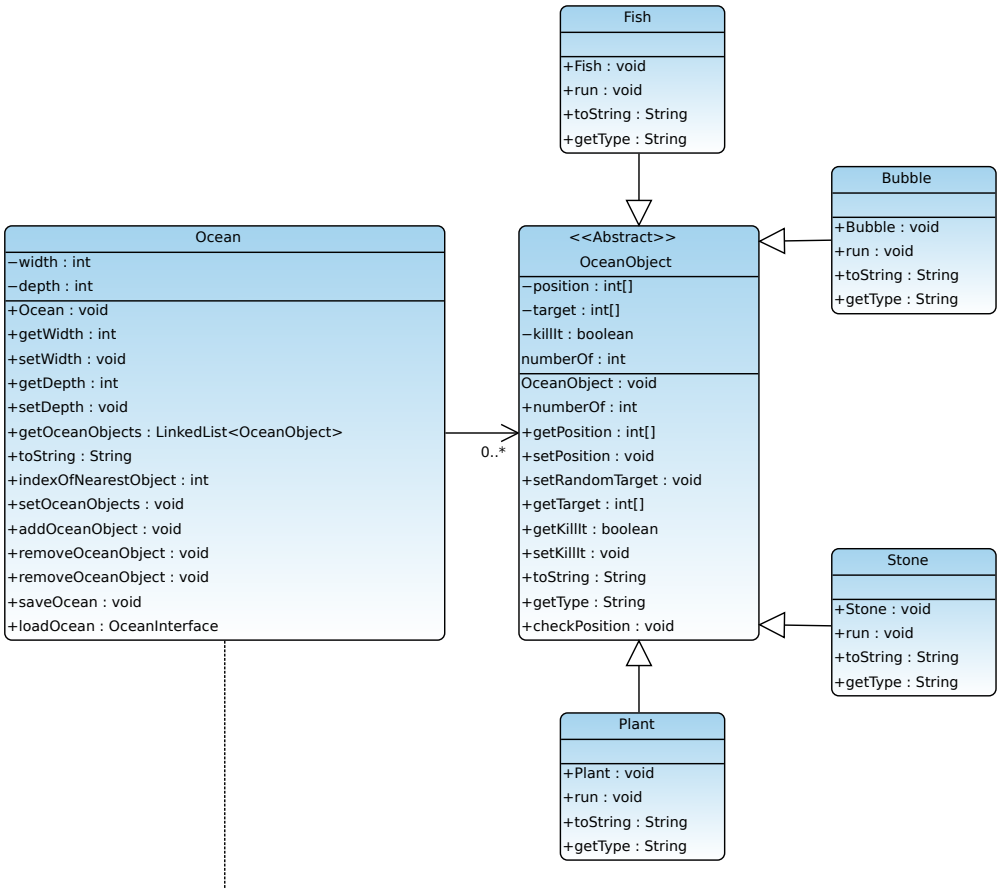


Abbildung 6.3. Ausschnitt des Klassendiagramms basierend auf der Selektion in Abbildung 6.1

Kapitel 7

Fazit

Im abschließenden Kapitel wird zunächst eine Zusammenfassung der umgesetzten Arbeit gegeben. Daraufhin wird erläutert, wie die Implementierung ursprünglich geplant war. Zuletzt werden die Erweiterungsmöglichkeiten für das erstellte Plug-in aufgezeigt.

7.1. Zusammenfassung

Es wurde erfolgreich ein Eclipse-Plug-in entwickelt, mit dem eine automatische Synthese von Klassendiagrammen aus Java-Quellcode durchgeführt werden kann. So werden die gewünschten Diagrammelemente angezeigt und verschiedene Beziehungen, wie Vererbung und Assoziationen, zwischen ihnen erkannt. Es wird weiterhin eine Unterscheidung zwischen konkreten Klassen, abstrakten Klassen, Enumerations und Interfaces getroffen. Das resultierende Klassendiagramm lässt sich durch Synthese- und Layoutoptionen weiter anpassen. Außerdem wurden erste Funktionen implementiert, welche die letzte Selektion, die synthetisiert wurde, abspeichern und wiederherstellen.

7.1.1. Vergleich zum Proposal

Ursprünglich wurde vorgeschlagen, für das Erstellen von Selektionen eine eigene GUI zu entwerfen (siehe Abbildung A.1). Dieser Vorschlag wurde jedoch durch die leichtgewichtigere Lösung ersetzt. Es wurde stattdessen auf schon bestehenden Elementen, wie dem Package- oder Project Explorer, gearbeitet. Somit muss sich ein Nutzer nicht an neue Oberflächen gewöhnen, sondern kann seine Selektion von Klassen, Methoden und Attributen mithilfe bekannter GUI-Elemente erstellen.

7.2. Erweiterungsmöglichkeiten

Während der Entwicklung sind einige Probleme aufgetreten. So werden Assoziationsbeziehungen zu Enumerations nicht erkannt. Des Weiteren werden bei mehreren Klassen mit gleichem Namen Vererbungs- und Assoziationsbeziehungen fälschlicherweise doppelt erkannt und visualisiert. Selbstassoziationen

werden nicht visualisiert, da der verwendete Layoutalgorithmus, ein Planarisierungsalgorithmus, diese nicht korrekt layouten kann. Bei einer verschachtelten Pakethierarchie, bei der Pakete andere Pakete enthalten, werden einzelne Pakete derzeit nebeneinander angezeigt, auch wenn ein Paket eigentlich in dem anderen enthalten ist. Innere Klassen können zurzeit ebenfalls noch nicht visualisiert werden.

In Zukunft könnte das Plug-in um eine Domain Specific Language (DSL) erweitert werden, sodass sich die Selektionen alternativ bearbeiten lassen, da bei einem falschen Mausklick die gesamte Markierung aufgehoben wird. Weiterhin wäre eine Synchronisation von Quellcode und abgespeicherter Selektion sehr hilfreich, da die synthetisierten Klassendiagramme bei Änderungen am Quellcode sehr schnell veraltet sind, sodass neue Diagramme erzeugt werden müssen.

Ein alternativer Layoutalgorithmus könnte die erzeugten Klassendiagramme optisch noch ansprechender machen und die Aussagekraft erhöhen. So wurde im Kapitel 2 evolutionäres Layout vorgestellt. In der Arbeit von Spönemann et al. [SDH14] wurde eine evolutionäre Methode zur Ermittlung eines gewünschten Layoutalgorithmus implementiert. Von Vorteil ist hierbei besonders, dass die Implementierung von Spönemann et al. KLighD als Layouting-Werkzeug verwendet, sodass eine Anbindung der Implementierung an meine Arbeit nicht sonderlich schwierig sein sollte.

Literatur

[Eic02] Holger Eichelberger. „Aesthetics of Class Diagrams“. In: *In Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, IEEE, 2002, S. 23–31.

[Fuh11] Hauke Fuhrmann. „‚On the Pragmatics of Graphical Modeling‘“. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[Gud+06] J. Wolff v. Gudenberg, A. Niederle, M. Ebner und H. Eichelberger. „Evolutionary Layout of UML Class Diagrams“. In: *Proceedings of the 2006 ACM Symposium on Software Visualization*. SoftVis ’06. Brighton, United Kingdom: ACM, 2006, S. 163–164. ISBN: 1-59593-464-2. DOI:]http://dx.doi.org/10.1145/1148493.1148525`10.1145/1148493.1148525`. URL: `http://doi.acm.org/10.1145/1148493.1148525`.

[RJB99] James Rumbaugh, Ivar Jacobson und Grady Booch, Hrsg. The Unified Modeling Language Reference Manual. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999. ISBN: 0-201-30998-X.

[Sch11] Christian Schneider. „‚On Integrating Graphical and Textual Modeling‘“. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf`. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2011.

[SDH14] Miro Spönemann, Björn Duderstadt und Reinhard von Hanxleden. Evolutionary Meta Layout of Graphs. Technical Report 1401. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Jan. 2014.

[SSH13] Christian Schneider, Miro Spönemann und Reinhard von Hanxleden. „Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice“. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’13)*. With accompanying]http://rtsys.informatik.uni-kiel.de/ biblio/downloads/papers/vlhcc13-poster.pdf`poster`. San Jose, CA, USA, 15–19 September 2013.

[Wan+07] Gene Wang, Brian McSkimming, Zachary Marzec, Josh Gardner, Adrienne Decker und Carl Alphonce. „Green: A Flexible UML Class Diagramming Tool for Eclipse“. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, S. 834–

835. ISBN: 978-1-59593-865-7. DOI:]http://dx.doi.org/10.1145/1297846.1297913[10.1145/1297846.1297913. URL: http://doi.acm.org/10.1145/1297846.1297913.

[Wiß13] Heiko Wißmann. „‚Graphische Visualisierung von Java-Variablen zur Laufzeit‘“. Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, März 2013.

Anhang A

Proposal

A.1. Einleitung

Zur Visualisierung von Software-Projekten werden oft Klassendiagramme verwendet. Diese Klassendiagramme helfen dann dabei, den Sourcecode zu verstehen oder zu präsentieren. Es ist oft sehr mühselig, Klassendiagramme manuell zu erstellen. Dementsprechend wäre es sehr hilfreich diese Diagramme automatisch aus Java-Sources zu synthetisieren. Des Weiteren ist es sehr hilfreich direkt ein Eclipse-Plug-in zu benutzen, welches diese Funktionalitäten bereitstellt.

A.2. Zielsetzung

Mit der Bachelorarbeit soll ein Eclipse-Plugin entwickelt werden, mit dem es möglich ist Klassendiagramme aus Java-Projekten in KLighD zu synthetisieren, sodass die Anordnung der verschiedenen Komponenten automatisiert wird. Denn die Anordnung von den einzelnen Komponenten kann, vor allem bei großen Projekten, sehr mühselig werden. Es soll möglich sein, festzulegen, welche Klassen, Methoden und Attribute dargestellt werden sollen. Des Weiteren soll festgelegt werden können, welche Kanten (Vererbung, Assoziationen, Abhängigkeiten, Realisationen), aller Klassen modelliert werden sollen.

Die grafische Oberfläche könnte beispielsweise wie in Abbildung A.1 aussehen.

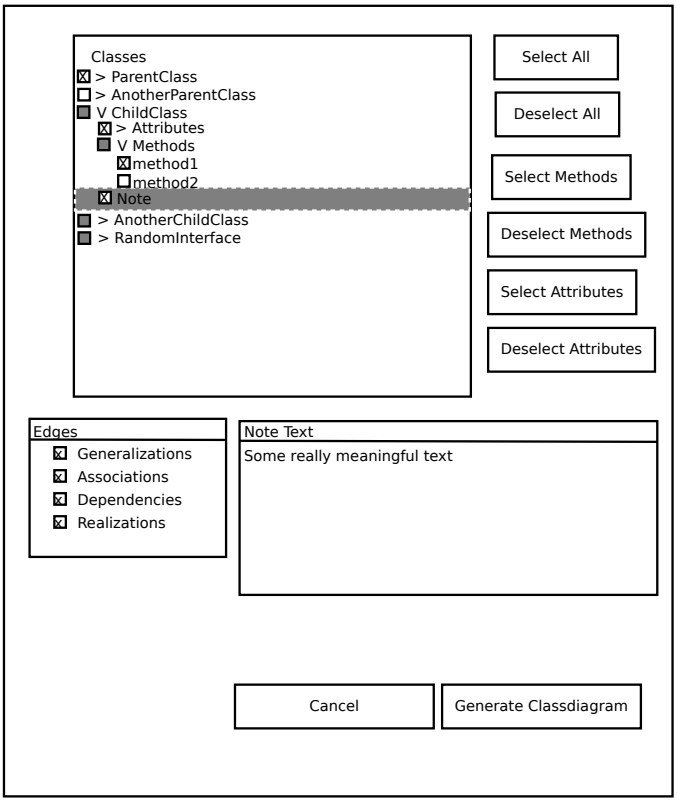


Abbildung A.1. Ein GUI-Beispiel für eine automatische Klassendiagrammsynthese

Eine mögliche Klassendiagramm Synthese könnte mittels KLighD der Darstellung in Abbildung A.2 entsprechen.

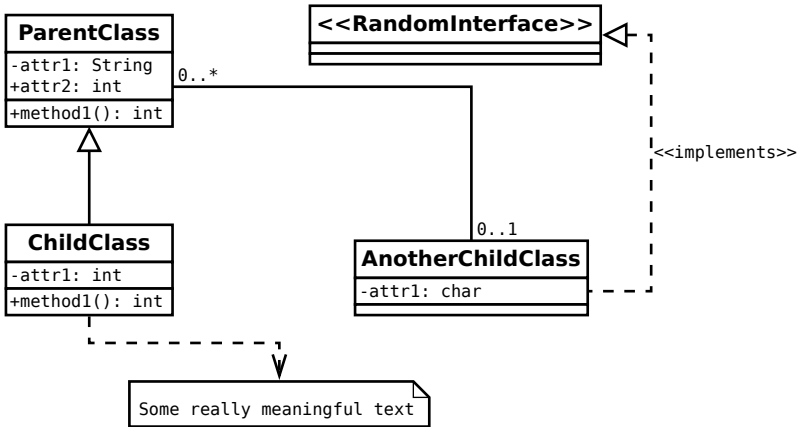


Abbildung A.2. Eine mögliche Klassendiagrammsynthese

A.3. Arbeitsschritte

Zunächst möchte ich ermitteln welche Informationen über ein Projekt von Eclipse bereitgestellt werden. Aus diesen Informationen möchte ich eine entsprechende Baumstruktur, wie in der beispielhaften GUI gezeigt, entwickeln. Diese Baumstruktur soll mittels Aktivierung und Deaktivierung der einzelnen Elemente für die letztendliche Synthese bearbeitet werden können.
Es müssen dann die einzelnen Klassen und Kanten mithilfe von Xtend synthetisiert werden und dann mittels KLighD visualisiert werden.