Fachhochschule Aachen

Fachbereich 9
Medizintechnik und Technomathematik

Studiengang Scientific Programming

Methoden zur Teilmigration von RPG nach JAVA

Seminararbeit

Lennart Küppers
Matr.-Nr. 3160603

Betreuer 1: Prof. Dr. Alexander Voß
Betreuer 2: Daniela Ophey

20. Dezember 2019

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Methoden zur Teilmigration von RPG nach JAVA

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Lennart Küppers

Aachen, den 20. 12. 2019

Lennart Küppers

Unterschrift der Studentin / des Studenten

Inhaltsverzeichnis

1. Einleitung

1.1. Abstract

Im Rahmen dieser Seminararbeit werden Grundlagen über die RPG(Report Program Generator) Programmierung vermittelt und Methoden angewendet, um bereits von RPG nach Java migrierten Code strukturierter, lesbarer und wartbarer, in einem objektorientierten Stil darzustellen. Hierbei wird im Besonderen auf verschiedene Probleme in RPG hingewiesen und wie man sie mit der Anwendung von Programmierparadigmen* verbessern kann.
Im weiteren Verlauf der Arbeit werden die Hintergründe zu diesem Thema, sowie die Anwendungsstruktur und Produktlinie bei VEDA kurz erläutert. Voraussetzungen und benötigte Kenntnisse für diese Arbeit sind ein Grundverständnis der Programmiersprache Java und objektorientierter Programmierung.

1.2. Hintergründe

Unsere Anwendung *VEDA HR Entgelt* läuft auf der „System i“, einer Computerbaureihe von IBM†. Besonders die Optimierungen für kaufmännische Anwendungen wie eine eigene integrierte Datenbank, machten das System attraktiv für unsere Software.

Bis heute machen die zahlreichen Vorteile des Systems, wie beispielsweise kaum vorhandene Sicherheitslücken oder einheitlicher/einfacher Betrieb(Fischer 2016), VEDA GmbH zu einem Marktführer in Sachen Personal- und Entgeltverwaltung. Mit RPG als eine der Hauptsprachen der OS/400‡, entwickelte VEDA GmbH in den vergangenen 40 Jahren eine performante und moderne Anwendung. Aufgrund der ständig wachsenden Anforderungen an die Software und einer stärkeren Dominanz anderer Programmiersprachen entschied sich das Unternehmen als zukünftige Programmiersprache Java zu nutzen. Die neuen Möglichkeiten und Wünsche nach modernen und benutzerfreundlichen Anwendungen ließen sich teilweise nicht mehr in RPG realisieren. Für eine automatisierte Übertragung des RPG Codes nach Java entwickelte VEDA GmbH das Produkt *VEDA JUMP*. Dieses Produkt migriert jede in RPG III geschriebene Software nach Java.

---

* Ein **Programmierparadigma** ist ein bestimmter Stil in dem ein Programm entworfen wird. Sie dienen nicht nur als Richtlinien für „schönen Code“, sondern können auch in speziellen Situationen eine Herangehensweise für Softwareprobleme bereitstellen(*Programmierparadigma* 2019).

† Die „International Business Machines Corporation“(**IBM**) ist ein weltweit führendes Unternehmen für Hardware, Software und Dienstleistungen in der IT-Branche(Rouse 2016).

‡ Das Operating System(**OS**), also Betriebssystem der System i.

1.3. Problemstellung

Der Nachwuchs an RPG-Programmierern ist sehr gering, Kosten und Wartung von RPG Programmen dementsprechend sehr hoch. Nicht nur die Popularität der Sprache ist laut Tiobe-Popularitätsindex* für Programmiersprachen sehr gering, auch die Online Präsenz bezüglich Dokumentation und Support ist beschränkt. Mit Platz 92 von 100 liegt RPG in 2019 sehr weit unten(*TIOBE-Index: Die aktuellen Top-Programmiersprachen im Ranking* 2019). Auch Universitäten unterrichten eher die heutzutage modernen und beliebten Programmiersprachen, wie Java, C++, C oder Pascal(Schlosser 2019). Einschränkungen und fehlende Funktionalitäten im Gegensatz zu moderneren Programmiersprachen, wie beispielsweise Java, brachten die VEDA GmbH dazu, all ihre RPG Programme nach Java zu migrieren.
Mit *VEDA JUMP* gelingt es VEDA GmbH eine 100-prozentige Migration nach Java zu schaffen. Der resultierende Code ist vollständig kompilierbar und lauffähig, ohne die Funktionalitäten der Programme negativ zu beeinträchtigen. Somit wurde die Software nicht nur plattformunabhängig, auch die Bindung an die integrierte Datenbank(**DB2**†) konnte aufgelöst werden.
Durch die Eigenarten und Besonderheiten von RPG ist der Code von *VEDA JUMP* jedoch lediglich auf die pure Migration fokussiert, anstatt auf eine java-nahe und strukturierte Codestruktur. Einige Stellen des migrierten Codes benötigen für das Verständnis weiterhin RPG Fachwissen, was für reine Java Entwickler ein Problem darstellen kann. Wird zum Beispiel ein neues Feld in der Anwendung benötigt, dann muss dieses in den RPG Programmen eingetragen und hinzugefügt werden. Daher müssen viele Codesequenzen bis heute in RPG zuerst angepasst und anschließend neu migriert werden.
In solch einem Fall bietet die Migration keinen Vorteil und stellt eher Zusatzarbeit dar. Genau hier setzt das Thema dieser Arbeit an. Der Code soll anhand von Analysen, Machbarkeitsstudien, Konventionen und Programmierparadigmen optimiert werden.

1.4. Lösungsansatz

Damit auch zukünftig Java Entwickler unseren Code warten können und um eine Abgrenzung zu RPG zu schaffen, muss der derzeitige Code optimiert werden. Ein Hauptbestandteil von RPG sind Funktionen, welche ausschließlich mit den übergebenen Parametern arbeiten. Während man in Java Methoden auf Objekten und Feldern aufruft, werden in RPG diese Funktionen statisch aufgerufen und erhalten alle beteiligten Objekte und Felder als Parameter. Dies widerspricht dem objektorientierten Ansatz von Java. Im Rahmen dieser Arbeit soll dieses Kernproblem analysiert und mit Hilfe verschiedener Methoden, Programmierparadigmen und Software Mustern‡ behoben werden.

---

* Der **Tiobe-Popularitätsindex** ist eine Ranking Liste für 100 Programmiersprachen

† Database 2(**DB2**) ist ein kommerzielles relationales Datenbankmanagementsystem, dass von IBM entwickelt wurde. Es ist in der System i integriert(*Db2* 2019).

‡ **Software Muster**(-Pattern) sind bewährte Methoden zur Entwicklung effizienter Software.

2. Allgemeines

Dieses Kapitel behandelt die Programmiersprache RPG (Report Program Generator). Außerdem werden Probleme der damaligen Zeit und resultierende Gründe für einen Umstieg nach Java aufgezeigt.

2.1. RPG

In den Anfängen der Datenverarbeitung, als noch mit Lochkarten* gearbeitet wurde, entwickelte das Unternehmen IBM die Hochsprache† RPG. Diese wurde besonders im Bereich der Großrechner und Minirechnern der mittleren Datentechnik‡, wie beispielsweise der AS/400 (System i) verwendet. Die Syntax der Sprache war sehr stark an die Arbeitsweise von Tabelliermaschinen§ angelehnt, weshalb auch der Sprachaufbau spaltenorientiert ist(*RPG (Programmiersprache)* 2019).

Im Laufe der Zeit entwickelte sich RPG immer mehr zu einer problemorientierten¶ Programmiersprache. Inzwischen gibt es neben den Weiterentwicklungen RPG II, RPG III und RPG/400, RPG IV (ILE RPG) als neuesten Stand. RPG ist eine der wenigen für Lochkarten-Maschinen entwickelten Sprachen, welche bis heute aufgrund ihrer ständigen Weiterentwicklung genutzt und gewartet wird(Rother 2015).

Das Besondere an einem RPG Programm ist, dass es in einem eigenen Programmzyklus läuft, welcher sich ständig wiederholt. Zuerst werden allgemeine Informationen der Kopfzeile verarbeitet. Danach beginnt das Programm den ersten Record‖ einzulesen,

---

* **Lochkarten** bestehen aus hochwertigem Karton und systematisch angeordneten Löchern. Spezielle Maschinen konnten diesen Lochcode erstellen und interpretieren(*Hollerith und der Lochkartencomputer* 2018).

† Eine **Hochsprache** ist eine, weit von Maschinensprache abstrahierte, Programmiersprache. Befehle können nicht direkt vom Mikroprozessor interpretiert werden und müssen durch Interpreter oder Compiler übersetzt werden(Schiffler 2005).

‡ **Großrechner** bezeichnen komplexe und große Computersysteme die meist über Serversysteme hinaus gehen. Sie werden besonders in der Massendatenverarbeitung verwendet. **Minirechner der mittleren Datentechnik** bezeichnen eher kleine Systeme meist bestehend aus Rechner, Software und Support(*Großrechner und Minirechner der mittleren Datentechnik* 2019).

§ **Tabelliermaschinen** dienen zur Auswertungen von Lochkarten.

¶ **Problemorientierte** Sprachen kommen der menschlichen Sprache sehr nah und sind einfacher zu verstehen im Gegensatz zur Maschinensprache(Lagotzki 2001).

‖ Ein **Record** entspricht einem Datensatz.

verarbeitet diesen und gibt letztendlich das Resultat aus. Dies wird dann mit nachfolgenden Datensätzen wiederholt.
Da RPG sehr intensiv mit Schaltern arbeitet, gibt es auch einen Schalter zur Beendigung des Programms. Wird der „LR“-Schalter(Last-Record) auf ON gesetzt, so führt dies zum Abbruch. Da RPG sich hier automatisch um die Datensätze kümmert, muss der Programmierer nicht auf den Zugriff der Datensätze achten(*Einführung in die RPG/400 Programmierung* 2019). Zur Veranschaulichung soll die folgende Abbildung als Grundlage dienen.

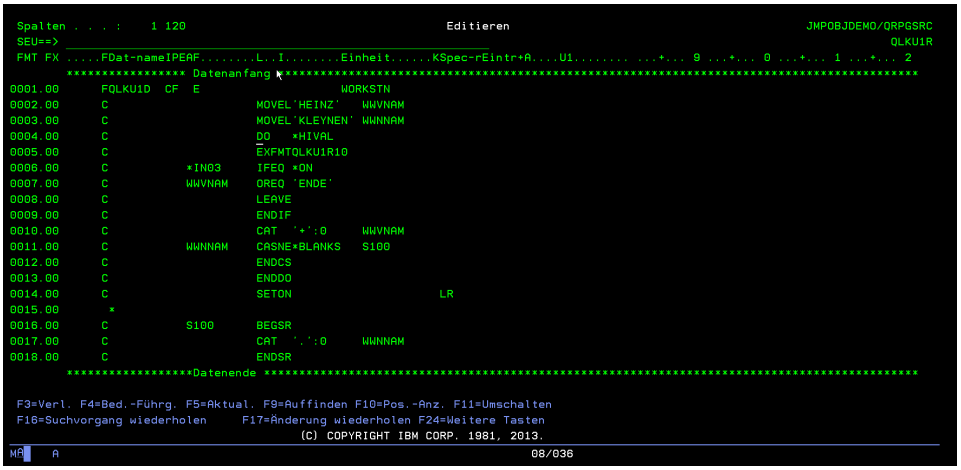

Abbildung 2.1.: RPG Programm in OS/400

Das obige RPG Programm besteht aus einem Vor- und einem Nachnamen Feld. Es belegt diese jeweils mit einem Beispielnamen vor. In der Ausführung wartet es auf eine Bestätigung des Benutzers, was hier dem Drücken von Enter entspricht. Ist dies passiert, wird ein „+“ an den Vornamen und ein „.“ an den Nachnamen angehangen, insofern dieser nicht leer ist. Drückt der Nutzer F3 oder schreibt „ENDE“ in das Feld Vorname, wird das Programm beendet.

Alle folgenden Beschreibungen der verschiedenen Befehle und Funktionen sind auf der offiziellen Seite von IBM nachzulesen(*Operation Codes List* 2019).
Wie in Abbildung 2.1 zu sehen beginnen die verschiedenen Zeilen mit Buchstaben. Hier wird die Rechenbestimmung, beziehungsweise die Funktion einer Zeile festgelegt. Das **C** kennzeichnet den nachfolgenden Code als ausführbare Statements, während beispielsweise ein **D** das Definieren von Variablen oder ein **F** den Verweis auf eine Datei impliziert.

Der Befehl **MOVEL** ist einer der „operation codes“(„OP Codes“), welche in genau dieser Spalte verwendet werden müssen. Sie gleichen Funktionen wie sie in Java bekannt sind. In diesem Fall bewirkt die Funktion, dass eine Zeichenfolge „HEINZ“ ganz nach links an ein Feld gesetzt wird. Das Besondere hierbei ist, dass dieser Text nicht einfach an das Feld angehangen wird. Der Befehl überschreibt den Inhalt des Ziels von ganz links ausgehend. Das bedeutet, ist die Zeichenfolge länger als das Zielfeld, werden alle zulässigen

Stellen überschrieben und der Rest verworfen. Ist die Zeichenfolge jedoch kürzer, werden die möglichen Zeichen überschrieben und die restlichen Stellen des Feldes beibehalten.

Weitere Beispiele für OP Codes sind in der folgenden Tabelle, zusammen mit ihren jeweiligen Bedeutungen und Java Äquivalenten aufgelistet.

Op Code	Bedeutung
MOVEL	Transferiert eine Zeichenfolge in ein Zielfeld, ausgehend vom links äußersten Character. **Java**: Kombination aus replace() und substring() da unterschiedliche Längen unterschiedlich verarbeitet werden müssten
MOVE	Transferiert eine Zeichenfolge in ein Zielfeld, ausgehend vom rechts äußersten Character. **Java**: replace() und substring() Konstrukt
ADD	Es wird eine Addition mit kompatiblen Objekten angewendet (Bsp. Typ Integer oder Array Elemente). **Java**: z = 5+a+b[0];
CALL	Hier wird eine parametrisierbare Funktion aufgerufen. **Java**: klasse.callmethod(x,y,...);
CAT	Zeichenfolgen werden hintereinander addiert (sofern die Feldlänge ausreicht). **Java**: name = „HEINZ“ + nachname;

Tabelle 2.1.: Operation Codes

Nun begibt sich das Programm in eine do-Schleife, beginnend mit dem Befehl **EXFMT**(-QLKU1R10). Bei diesem Befehl stoppt das Programm und blockiert die weiteren Prozesse, bis der Benutzer eine Eingabe gemacht und bestätigt hat. Hat der Benutzer nun eine Eingabe getätigt, so wird der weitere Inhalt der Schleife ausgeführt.

In Zeile 6 und 7 gibt es noch zwei if-Abfragen, mit zwei verschiedenen Input-Feldern. Zuerst wird überprüft, ob **IN03** auf ON gesetzt ist. In RPG ist diesem Schalter die F3 Taste zugeordnet. Es wird also nur geprüft ob der Benutzer diese betätigt hat, und der Schalter auf ON gesetzt wurde. Generell gilt in der OS400 Umgebung, dass man mit F3 den aktuellen Menüpunkt oder das derzeitige Programm in dem man sich befindet, verlassen kann. Die zweite Bezeichnung „WWVNAM“ entspricht einem Feld aus der Oberfläche. Gleicht der Inhalt „ENDE“ so wird ebenfalls der Code der if-Prozedur ausgeführt. Dass das Programm auf dieses Feld zugreifen kann liegt daran, dass in Zeile 1 durch die Kennzeichnung **F** auf eine Datei namens „QLKU1D“ referenziert wird. In diesem Beispiel ist das ein Displayfile, welches dazu da ist die UI* der Anwendung zu erstellen und zu gestalten. Dort befindet sich das Feld mit dem Namen „WWVNAM“ .

Sobald das Programm nun in diese Abfrage kommt, wird **LEAVE** ausgeführt, was einem „break“ in Java entsprechen würde. Die Schleife wird verlassen und in Zeile 14 wird der **LR** Schalter auf ON gesetzt, was die Beendigung des Programms zur Folge hat.

* Das User Interface(**UI**) ist die Benutzerschnittstelle einer Anwendung. Über sie kann ein Benutzer mit einer Maschine interagieren und kommunizieren.

Bestätigt der Benutzer jedoch seine Eingabe ohne Drücken der F3 Taste oder Schreiben des Wortes Ende, so wird die if-Abfrage übersprungen, und ein Befehl **CAT** wird ausgeführt. Dieser Befehl hängt einen Text (in diesem Fall ein „+“) an das gewünschte Feld („WWVNAM“). Die 0 als weiteren Parameter definiert nur, wieviele freie Stellen zwischen dem anzuhängenden Text und dem Feldtext bestehen sollen. Hier wird also das „+“ direkt hinter den Feldtext geschrieben. Das lässt sich mit RPGs festen Feldlängen erklären. Da jedes Feld eine fest definierte Feldlänge haben muss, muss genau angegeben werden an welches Byte etwas angehängt werden soll.
Zusätzlich dazu wird in Zeile 11 mit dem Befehl **CAS** eine bedingte Subroutine(Methode) namens **S100** aufgerufen. **NE** steht für „Not Equals“, wenn also das Feld „WWNNAM“ ungleich ***BLANKS**(nicht leer) ist, dann wird die Subroutine aufgerufen. In der Subroutine gibt es einen weiteren **CAT** Befehl welcher äquivalent zum vorherigen einen Punkt anstatt einem Plus an das Feld anhängt.
Besonders für nicht RPG-Entwickler führt die fehlende Einrückung sowie die Verteilung von Parametern und Funktionen zu einer schlechten Lesbarkeit. Das nachfolgende Ablaufdiagramm (Abbildung 2.2) soll den beschriebenen Programmfluss verdeutlichen.

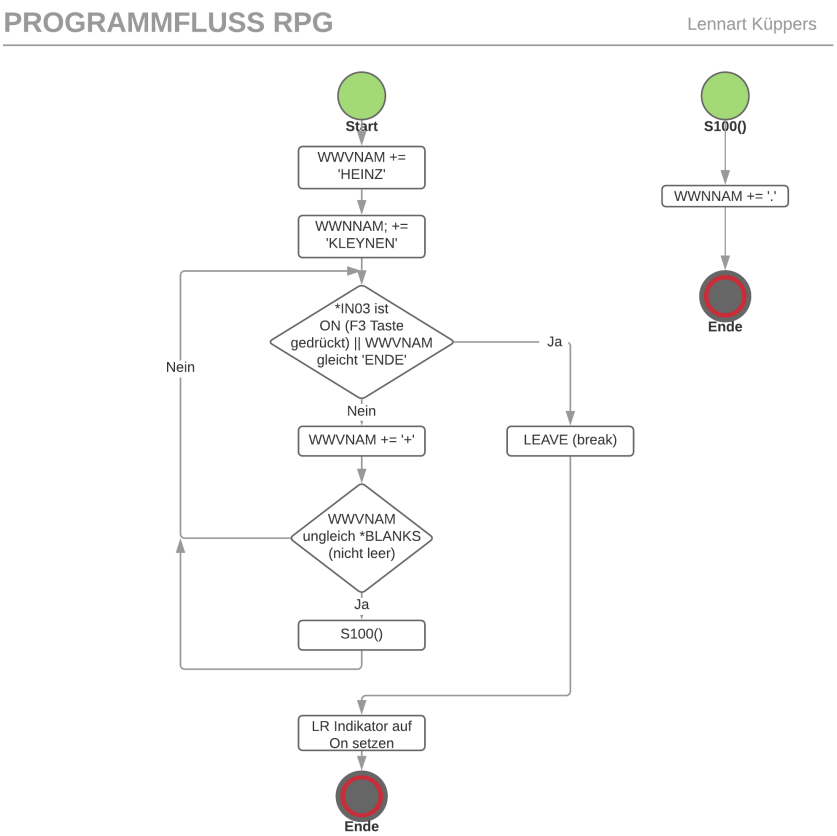

Abbildung 2.2.: Ablaufiagramm zu Abbildung 2.1

2.2. Gründe für einen Umstieg

Besonders im Vergleich zu modernen Programmiersprachen, ist RPG in vielerlei Hinsicht unkomfortabel und nicht sehr fortschrittlich. In diesem Kapitel werden ein paar der Probleme und Gründe für den Umstieg auf Java erläutert.
Einige dieser Probleme wurden im weiteren Entwicklungsverlauf von RPG behoben, da es zur damaligen Zeit jedoch keine Alternativen gab oder der jeweilige Aufwand größer als der Nutzen war, werden hier ein paar speziellen Beweggründe aufgezeigt.

2.2.1. Datenbanken

Deutliche Nachteile der Programmiersprache lagen in den Datenbankprogrammen. Da Speicherplatz begrenzt und rar war, gab es in RPG feste Feldlängen. Bei Änderungen wie beispielsweise der Erweiterung von Postleitzahlen oder aber simplen Werterweiterungen aufgrund neuer Anforderungen, wurde die vordefinierte Länge eines Feldes zum Problem. Abseits von Datenbanken hat auch die Programmierumgebung mit diesem Problem zu kämpfen. Ab bestimmten Spaltennummern hat der jeweilige Code eine neue Bedeutung und kennzeichnet beispielsweise Feld-/Variablen-Namen oder OPCode(siehe Tabelle 2.1 auf Seite 9), welche ebenfalls in ihrer Länge beschränkt sind.

Außerdem, als damals die Möglichkeiten der Datenbanken eher begrenzt waren, wurde aus Performance-Gründen auf Funktionen wie Referentielle Integrität (RI)* verzichtet. Die Datenbanken auf der System i waren nicht immer in dritter oder höherer Normalform vorzufinden, was zur Folge einem schlechten Datenbankdesign entsprach(Kleutgens 2010).

2.2.2. UI

Heutzutage sind Modell View Controller Strukturen ein fester Bestandteil von vielen Softwarelösungen. Die strikte Trennung zwischen Modell (Daten- und Businesslogik), View (reine Oberfläche) und Controller (Vermittler/Brücke zwischen Modell und View) sorgt für eine gute, logische und unabhängige Modularisierung des Codes. In den RPG Anwendungen existiert jedoch kein Eventmodell und die Schichten besonders View und Controller sind vermischt. Mehr oder weniger hart codierte Aktionsvorgaben und Abläufe des Programms machten den Code unflexibel und schlecht austauschbar.

Ein Beispiel hierfür ist beispielsweise der Job-Stack einer IBM i Anwendung. Die RPG Programme laufen in einem Job und haben einen eigenen Stack. Ruft also ein Programm A ein Programm B auf, so kann B nur über A erreicht werden und wenn man B verlässt landet man sofort wieder in A. Man sieht, dass der Controller nicht frei ist und die Maske aus Programm B immer eine Verarbeitung von Programm A erfordert(Kleutgens 2010).

* Referentielle Integrität(**RI**) ist eine Bedingung der Datenkonsistenz von Datenbanken. Fremdverweise in Datenbanken müssen auf existierende Datensätze verweisen(*Referentielle Datenintegrität* 2019).

2.2.3. Businesslogik

Aufgrund der generell fehlenden Objektorientierung, wurde keine strikte Trennung zwischen Funktionen eingehalten. Zur Folge hatte man teils monolithische Programme, welche zwar aus Performance-Sicht effizient waren, jedoch keinen zukunftstauglichen Code darstellten. Da zwischen den Programmen viele Job-Abhängigkeiten bestanden, war es besonders schwierig zusammenhängende Funktionen in isolierte Services aufzuteilen. Die dazu benötigte dynamische Bindung von Programmen erforderte eine hohe Rechenleistung. Die Konsequenz waren zu große und unübersichtliche Services und eine schlechte Modularisierung(Kleutgens 2010).
Auch Sprunganweisungen („GOTOs“)sind ein Teil von RPG und der Software von VEDA GmbH. Aus Zeitgründen boten sich diese meist gut an, jedoch wurde der Code dadurch weiterhin unstrukturierter. Besonders seit Edsger W. Dijkstras Aufsatz „Go To Statement Considered Harmful“ wurden diese Sprunganweisungen als kritisch angesehen(Dijkstra 1968). In Java wurden diese bewusst weggelassen(*Sprunganweisungen* 2019).

3. Methoden

Im Programmierverlauf dieser Arbeit wurden bestimmte Methoden und APIs verwendet. Anknüpfend folgt eine Erläuterung der genutzten Hilfsmittel und angewendeten Techniken, die zur Erfüllung des Ziels der Seminararbeit dienten.

3.1. Java Abstract Syntax Tree

Das Eclipse Java Development Tools (JDT)* Projekt beinhaltet Schnittstellen, mit denen man auf Java Code zugreifen und diesen manipulieren kann. Es besteht aus den APIs Java Model und Abstract Syntax Tree(AST). Das Java Model gleicht der Strukturierung der Java Projekte wie man sie in Eclipse kennt. Es beinhaltet die oberflächliche Aufteilung der verschiedenen Elemente im Projekt Explorer. Darunter fallen beispielsweise Projekt Ordner (IJavaProject), Pakete (IPackageFragment) oder source-Files (ICompilationUnit). Hier spiegelt sich die bekannte Baumstruktur wieder(Vogel, Scholz und Pfaff 2018).

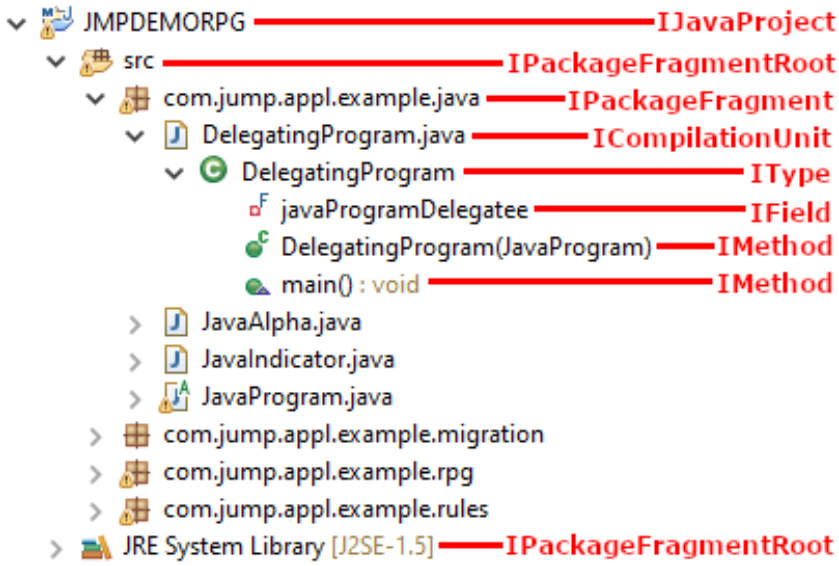

Abbildung 3.1.: Java Model und AST in Eclipse

Der Abstract Syntax Tree jedoch ist eine detaillierte Baumdarstellung des jeweiligen Java Codes. Mit dieser API ist es möglich Code nicht nur zu erstellen, sondern auch zu lesen und zu modifizieren.

* **JDT** ist eine Sammlung von Plugins für Eclipse

3.1.1. Struktur

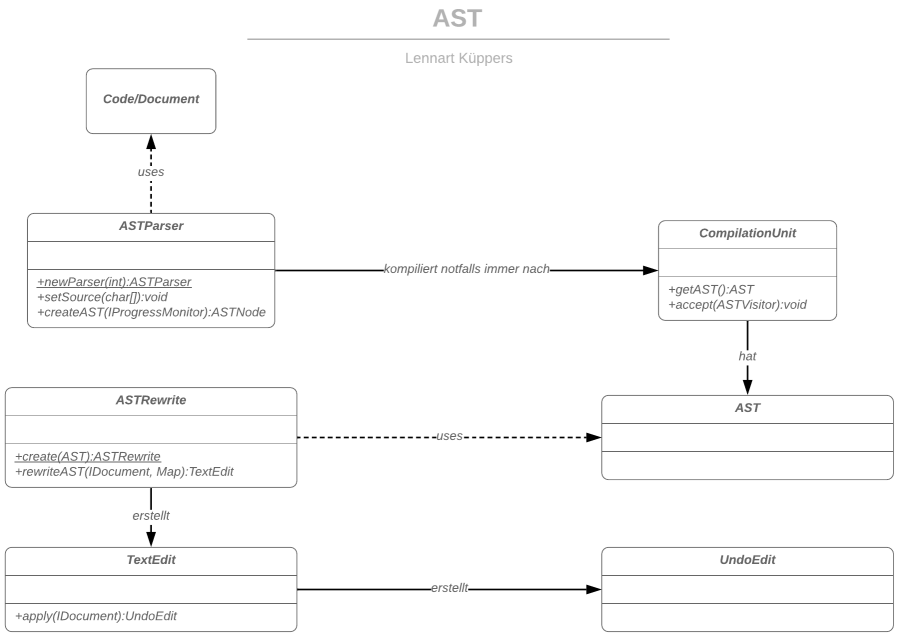

Abbildung 3.2.: UML-Diagramm AST Modifizierung

Das obige UML* Diagramm visualisiert die Klassen und Methoden, welche dazu dienen ein vorhandenes Dokument zu modifizieren. Zu Beginn wird ein ASTParser erzeugt, welcher mit einem Dokument, bzw. dem Source Code versehen wird. Nun kann eine ASTNode erzeugt werden. Im Grunde lässt sich je nachdem wie die Quelle aussieht, dieser Node zu verschiedenen Unterklassen parsen, die Klasse CompilationUnit jedoch spiegelt das gesamte Dokument wieder und kann daher immer verwendet werden.
Hat man nun einen Knoten gefunden, besitzt auch dieser einen abstrakten Syntaxbaum, welchen man mit einer einfachen Get-Anfrage erhalten kann. Dieser AST kann jedoch nicht direkt modifiziert werden. Er stellt das gesamte Dokument als Objekt in Java dar, um ihn jedoch ändern zu können, muss eine weitere Klasse ASTRewrite verwendet werden. Diese Klasse erzeugt einen Klon von dem übergebenen AST, auf dem gearbeitet werden darf.
Über sogenannte ASTVisitoren ist es möglich bestimmte Codezeilen je nach Bedeutung zu besuchen bzw. zu durchlaufen. So kommt man genau an die Programmbereiche wo man gegebenenfalls Änderungen durchführen möchte. Sind nun alle Änderungen auf diese Kopie angewendet worden, kann man den AST über die Methode rewriteAST() umschreiben. Es wird ein TextEdit Objekt zurückgegeben, welches man letztendlich noch auf das Dokument anwenden muss um auch dieses final zu ändern(Boehr 2015).

* Die „Unified Modeling Language“(**UML**) ist eine vereinheitlichte Modellierungssprache, mit der Software-Teile und Systeme konstruiert und modelliert werden können(*Was ist ein UML Diagramm* 2019).

3.1.2. ASTVisitor

Ein ASTNode lässt sich mit beliebigen ASTVisitoren verknüpfen. Jede Codezeile lässt sich einem anderen Knoten zuordnen.
So gibt es beispielsweise einen Visitor für If-Abfragen, Import Deklarierungen oder aber Kommentaren. Akzeptiert also ein ASTNode einen If-Visitor, so werden alle Stellen im Code besucht, wo eine If-Abfrage verwendet wird. Der besuchte Knoten, in dem Fall vom Typ IfStatement, kann dann komponentenweise ausgelesen und abgeändert werden. So besteht die Bedingung beispielsweise aus einer Expression und die auszuführenden Blöcke innerhalb der Abfrage jeweils aus Statements. Diese Aufteilung zwischen Visitoren und Elementen lässt sich mit dem Visitor Pattern beschreiben.

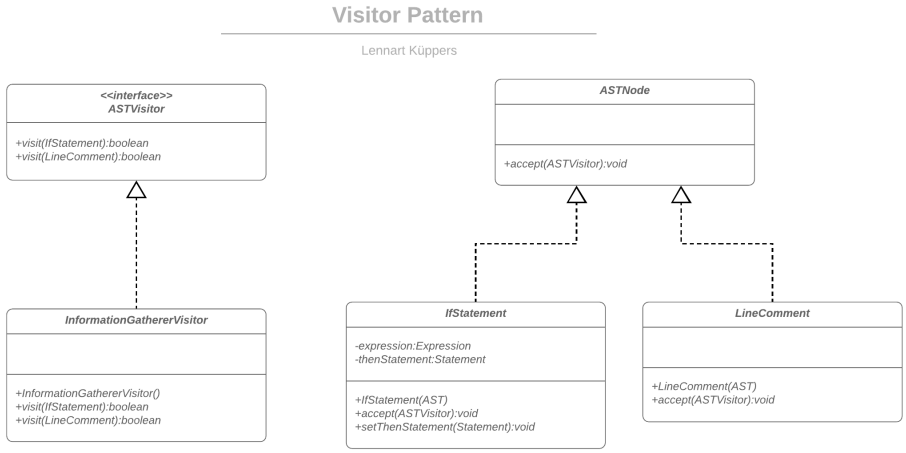

Abbildung 3.3.: Visitor Pattern am Beispiel ASTVisitor

Abbildung 3.3 veranschaulicht das Visitor Pattern am Beispiel der AST Struktur. Das Interface ASTVisitor enthält alle visit Methoden mit ihren jeweilig dazugehörigen Elementen als Parameter. Davon abgeleitet kann man individuelle Besucherklassen erstellen, in denen dann die eigentliche Logik implementiert wird. Die einzelnen Elemente besitzen die accept Methode, um bestimmte Besucher an die jeweiligen Objekte anzuhängen. Besonders im Compilerbau wird dieses Muster häufig angewendet, da Änderungen größtenteils in den Besucherklassen anstatt in allen konkreten Elementen getätigt werden müssen(*Visitor Pattern* 2019).

3.1.3. Vorteile

- Die Besucher lassen sich sehr einfach und beliebig erweitern. Gibt es Verhaltensänderungen wird ein neuer Besucher abgeleitet vom Visitor Interface als Klasse erstellt(*Visitor Pattern* 2019).
- Es gibt eine zentrale Stelle zur Änderung und Erweiterung von Funktionen. Die Implementierung befindet sich in der Visitor Klasse und muss nicht erneut in den

zu besuchenden Klassen implementiert werden(*Visitor Pattern* 2019).

- Besucherklassen können Modul- und Klassenübergreifend verwendet werden und sind nicht an eine bestimmte Klassenhierarchie gebunden(*Visitor Pattern* 2019).
- Je nach Anwendungsfall können so wichtige Daten und Zwischeninformationen aus besuchten Elementen einer Struktur gesammelt und weiterverarbeitet werden(*Visitor Pattern* 2009).

3.1.4. Nachteile

- Gibt es konkrete neue zu besuchende Klassen, so muss jeder Visitor dementsprechend geändert werden(*Visitor Pattern* 2019).
- Je mehr konkrete Klassen es zu unterstützen gilt, desto mehr visit Methoden muss es in den Visitor Klassen geben. Dies kann sehr schnell sehr unübersichtlich werden(*Visitor Pattern* 2019).
- Das Prinzip der Codekapselung* wird verletzt, indem der Besucher auf viele verschiedene und gegebenenfalls verstreute Elemente und Objekte außerhalb der eigenen Klasse zugreift. Zusätzlich muss der Besucher die jeweiligen Elementklassen kennen um mit ihnen zu interagieren während, die Elemente selbst die Struktur des Besuchers nicht kennen(*Visitor Pattern* 2009).

* **Datenkapselung** bezeichnet den kontrollierten Zugriff auf bzw. das Verbergen von internen Datenstrukturen.

4. Ergebnisse

Im Folgenden werden die Programmierergebnisse vorgestellt. Anhand einer händisch verfassten Vision, wurde die Methode aus Kapitel 3 angewendet. Vor Allem stehen der Ausgangscode, die Vision und das letztendliche Resultat im Vergleich zueinander.

4.1. Problemstellen und Vision

Der nachfolgende Code, ist die migrierte Version des RPG Codes aus Kapitel 2. Die roten Bereiche kennzeichnen nur ein paar der Problemstellen, die mit der Java Migration einhergehen. Doch um eines der Kernprobleme der Objektorientierung zu beschreiben reicht dieses Programm aus.

```
19 public final class QLKU1RNEU extends RpgProgram {1.
20
21     public DSPF qlkuld = createDSPF("QLKU1D", false); // qlkuld
22     public DSPFRecord qlkulrl0 = (DSPFRecord)qlkuld.getRecord("qlkulrl0"); // qlkuld || qlkulrl0
23     public RpgAlpha wwvnam = new RpgAlpha(30); // qlkuld || qlkulrl0 || wwvnam
24     public RpgAlpha wwnnam = new RpgAlpha(30); // qlkuld || qlkulrl0 || wwnnam
25                                                                          2.
26     public QLKU1RNEU() {
27         creationInfos = "13.11.2019 15:09:53"; // Transformed by lku
28         optimizedDO = true;
29         optimizedIF = true;
30         optimizedSELEC = true;
31         optimizedSearching = true;
32         // file declarations
33         setDSPF(qlkuld);
34
35
36
37     }
38
39     public void main() {
40
41
42
43 // qlkuld (WORKSTN
44     movel("HEINZ", wwvnam);
45     movel("KLEYNEN", wwnnam);  3.
46     LOOP_4: while (true) {
47         exfmt(qlkulrl0);
48         if (eq(IN03, ON)
49           || eq(wwvnam, "ENDE")
50         ){                                                          4.
51             break LOOP_4;
52         }
53         cat("+", 0, wwvnam);  5.
54         if (ne(wwnnam, BLANKS)) {s100();}
55     }                                                               6.
56     seton(INLR, null, null);
57                                                                    7.
58
59     }
60
61     public void s100() {
62
63         cat(".", 0, wwnnam);  8.
64     }
65 }
66
```

Abbildung 4.1.: Der jetzige unmodifizierte Code

- (1) Der neue Code sollte nicht direkt von einer RPG Klasse erben und auch grundsätzlich keine sofort ersichtlichen Abhängigkeiten zu diesen besitzen, da eine Un-

abhängigkeit angestrebt wird.

- (2) Der Typ RpgAlpha bietet sich einerseits nicht nur vom Namen schlecht an, die Klasse verhindert auch einen objektorientierten Zugriff auf die Variable, da sie nicht dafür vorhergesehen ist.
- (3)(5)(8). Hier werden typische RPG Funktionen aufgerufen. Im Endeffekt handelt es sich hierbei um simple Wertveränderungen von Feldern. Anstatt statische Funktionen aufzurufen, sollten Methoden hinzugefügt werden mit denen man über das jeweilige Objekt die Wertänderung erzielt (getText(), setText(), etc.).
- (4)(7) Auch hier werden RPG Funktionen statisch aufgerufen. Zusätzlich ist die Abfrage ob der Schalter IN03 lediglich auf ON gesetzt ist, zu umständlich. Eine intuitivere Abfrage wie isON() auf dem Schalter-Objekt wäre eine typische Herangehensweise in Java.
- (6) Die RPG-Funktion „ne“ steht für „not Equals“. In Java wird für alle verneinten Abfragen ein „!“ verwendet. So müsste nicht für jede Funktion eine verneinte Implementierung existieren.

```
13 public final class QLKU1R_formatted extends JavaProgram {  1.
14
15     public DSPF qlkuld = createDSPF("QLKU1D", false); // qlkuld
16     public DSPFRecord qlkulr10 = (DSPFRecord)qlkuld.getRecord("qlkulr10"); // qlkuld || qlkulr10
17     public JavaAlpha wwvnam = new JavaAlpha(30); // qlkuld || qlkulr10 || wwvnam
18     public JavaAlpha wwnnam = new JavaAlpha(30); // qlkuld || qlkulr10 || wwnnam   2.
19
20     public QLKU1R_formatted() {
21         creationInfos = "13.11.2019 15:11:13"; // Transformed by lku
22
23         setOptimizedDO(true);
24         setOptimizedIF(true);
25         setOptimizedSELEC(true);
26         setOptimizedSearching(true);   3.
27
28         // file declarations
29         setDSPF(qlkuld);
30     }
31
32     public void main() {
33         // qlkuld (WORKSTN)
34         wwvnam.setText("HEINZ");
35         wwnnam.setText("KLEYNEN");   4.
36
37         LOOP_4:
38         while (true) {
39             exfmt(qlkulr10);
40             if (IN03().isOn() || wwvnam.isEmpty()){   5.
41                 break LOOP_4;
42             }
43             wwvnam.concat("+", 0);   6.
44             if (!wwnnam.equals(BLANKS)){   7.
45                 s100();
46             }
47         }
48         INLR().setOn(null, null);   8.
49     }
50
51     public void s100() {
52         wwnnam.concat(".", 0);   9.
53     }
54 }
```

Abbildung 4.2.: Handgefertigte Vision

Diese Vision gibt eine erste Vorstellung, wie ein optimierter Code in dieser Form aussehen könnte. Die grünen Bereiche stellen hier die veränderten Codesequenzen dar.

- (1) Als Basisklasse verwendet das Programm nun eine für Java optimierte Klasse.
- (2) Die Objekttypen der Felder wurden angepasst. Die neuen Klassen sollten vor allem auf einen objektorientierten Zugriff ausgelegt sein und genau die jeweiligen Methoden enthalten, welche bisher statisch über die Basisklasse aufgerufen wurden.
- (3) Wichtige Schalter welche gegebenenfalls über einen großen Teil des Programmflusses entscheiden können, sollten nicht direkt modifizierbar sein. Es sollten Getter- und Setter- Methoden zum Lesen und Setzen dieser existieren.
- (4)(6)(8)(9) Wertemanipulationen wie das Setzen eines Feldtextes sollten auch genauso über entsprechende Methoden erfolgen.
- (5)(7) Equals-Abfragen sollten grundsätzlich auf genau dem Objekt aufgerufen werden, welches auch verglichen werden soll. Ob es ein String Objekt oder ein Feld selbst ist. Außerdem wie in (7) ersichtlich, sollten verneinte Abfragen mit einem „!“ gekennzeichnet sein. Das verbessert nicht nur die Lesbarkeit, sondern spart auch zusätzlichen Code.

4.2. Strukturierung

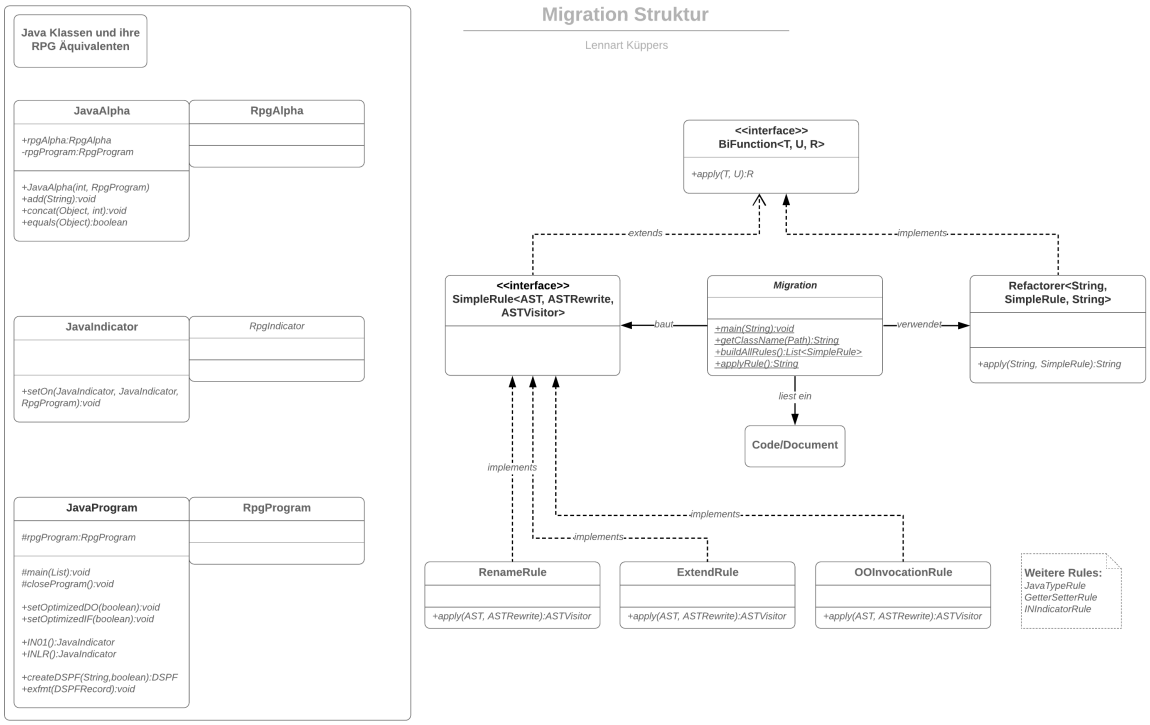

Abbildung 4.3.: Struktur des Migrator Codes

Die Hauptlogik des Migrators befindet sich in der Klasse „Migration“. Hier wird zuallererst der Programmcode aus den Startparametern eingelesen und als String abgespeichert.

Danach wird der eigentliche Refactorer erzeugt. Diese Klasse implementiert das Interface „BiFunction<T,U,R>“. Dabei sind die Typen **T** und **U** Eingabeparameter und **R** der Rückgabewert einer Funktion „apply“. Auf den Code sollen viele verschiedene Funktionen und Regeln angewendet werden. Als Konsequenz muss auch die Implementierung und generelle Klassenstruktur flexibel und erweiterbar sein. Als Parameter erhält der Refactorer nur den Code als String und eine anzuwendende Regel/Funktion. Mit einem String als Rückgabeparameter soll so nach Anwendung jeder Regel der Code Stück für Stück optimiert werden.
In der apply-Methode selbst, wird der ASTRewrite Prozess (beschrieben in Abbildung 3.2 auf Seite 14) ausgeführt. Der Refactorer erzeugt einen ASTParser und parst den Quellcode in eine CompilationUnit. Nun werden ein dazugehöriger AST und ASTRewrite erstellt. Nachdem die Regel dann auf den Rewrite angewendet wurde, wird auch der ursprüngliche Ast verändert und dementsprechend aktualisiert.

Im Anschluss an die Erzeugung des Refactorers, werden in der Migration Klasse die einzelnen Regeln über eine Methode „buildAllRules()“ gebaut. In dieser Methode lassen sich neue Regeln und Funktionen flexibel hinzufügen. Manchmal müssen bestimmte Funktionen jedoch hintereinander angewendet werden, da sie bedingt auf der Ausführung und den Änderungen der vorherigen Regeln agieren. Beruht eine Regel beispielsweise darauf, dass eine Methode objektorientiert auf einem Objekt aufgerufen wird, obwohl die dafür verantwortliche Regel noch nicht aktiv gewesen ist, so werden die neuen Modifikationen gegebenenfalls nicht beachtet. Es kann sogar zu Kompilierfehlern kommen.

Die Regeln implementieren ein Interface „SimpleRule“, welches wie der Refactorer von der Klasse BiFunction abhängig ist. Anstatt einem String und einer SimpleRule als Parameter, werden hier ein AST und ein ASTRewrite übergeben. Dadurch kann in den einzelnen Regeln der Code effizient modifiziert werden. Der Rückgabewert ist vom Typ ASTVisitor, damit der Refactorer die Regeln später an den Code anhängen kann. Die Besucher erhalten somit in den Regel-Klassen ihre jeweiligen Funktionen und die zu besuchende Elemente. Die folgende Abbildung zeigt wie solch eine Funktion zu implementieren wäre.

```
public ASTVisitor apply(final AST ast, final ASTRewrite rewrite) {
    return new ASTVisitor() {

        public boolean visit(Assignment node) {
            String methodname = node.getLeftHandSide().toString();
            if(indicators.contains(methodname)){
                MethodInvocation setter = ast.newMethodInvocation();
                setter.setName(ast.newSimpleName("set" + methodname));
                if(node.getRightHandSide() instanceof BooleanLiteral){
                    BooleanLiteral parameter = (BooleanLiteral) node.getRightHandSide();
                    setter.arguments().add(ast.newBooleanLiteral(parameter.booleanValue()));
                }
                rewrite.replace(node, setter, null);
            }
            return true;
        }
    };
}
```

Abbildung 4.4.: Beispielmethode

Wurden alle Regeln gebaut, werden diese in einer Liste abgespeichert. Mit einer For-Schleife wird über diese Liste iteriert und jede Regel wird auf den Refactorer über die apply-Methode angewendet.
Ist auch dieser Schritt abgeschlossen, wird der entstandene Code als String, in eine neue Datei geschrieben. Abseits der eigentlichen Migration, mussten auch neue Java Klassen erstellt werden(siehe linker Block in Abbildung 4.3 auf Seite 19). Diese Klassen machen im Grunde nichts anderes als ihre RPG Äquivalenten, legen jedoch eine Art Maske über den Code, dass es so aussieht als würde man in einem ganz normalen Java-Programm arbeiten. So ist zum Beispiel die Klasse „JavaAlpha“ das Gegenstück zu der RPG-Klasse „RpgAlpha“. Sie enthält Methoden wie „equals“ oder „concat“, welche zwar das Gleiche tun wie die statischen Funktionsaufrufe von RPG, jedoch auf dem konkrete Objekt aufgerufen werden. So entsteht der Schein von Objektorientierung.

4.3. Angewendete Regeln/Besucher

Hier werden die unterschiedlichen Regeln/Besucher erläutert. Dabei wird eine kurze Erklärung über die zugehörige Funktion und die Umsetzung gegeben.

4.3.1. RenameRule

Nach der ersten Migration ohne irgendeine Regel auf den Code anzuwenden, gab es einen Kompilierfehler in dem entstandenen Programm. Da der neue Code mit dem Namenszusatz „_refactored.java“ abgespeichert wurde, widersprach dies dem noch unberührten Klassen- und Konstruktornamen im Quellcode selbst. Für den Klassennamen besucht der Besucher eine ASTNode „TypeDeclaration“. Hier wird lediglich der Namenszusatz mit an den Klassennamen angehangen, damit dieser mit dem Dateinamen übereinstimmt. Für die Konstruktoren wird nach „MethodDeclarations“ gefiltert. Wenn dann zusätzlich der Rückgabewert null entspricht, dann bedeutet das, dass ein Konstruktor vorliegt.

4.3.2. ExtendRule

Auch für den Extend wird „TypeDeclaration“ verwendet. Anstatt jedoch den Namen der Klasse zu ändern, benennt man die Basisklasse nach „JavaProgram“ um. Diese Klasse kann nun wie eine Java Klasse behandelt und weiterentwickelt werden.

4.3.3. JavaTypeRule

Damit die Objektorientierung funktioniert, musste wie oben beschrieben ein neuer Java-Typ für die Felder hinzugefügt werden. Hierzu muss man beim Migrieren die Deklaration, also die linke Seite, sowie die eigentliche Instanziierung, also die rechte Seite anpassen. Dabei mussten Name und Erzeuger-/Konstruktoraufruf aktualisiert werden.
Gespeichert in einer HashMap*, können beliebig viele neuen Datentypen hinzugefügt werden. Diese werden mit dem RPG-Typen als Key und dem Java Äquivalent als Value abgespeichert, damit der Migrator die Datentypen einfach und flexibel austauschen kann.

4.3.4. GetterSetterRule

Zuweisungen, zum Beispiel von boolean Variablen, werden mit der ASTNode „Assignment“ beschrieben. Weiterhin werden hier alle unterstützten Variablen mit ihrem Namen in einer Liste abgespeichert. Wenn also der Besucher an einer dieser Variablen vorbeikommt, wird eine neue „MethodInvocation“ erzeugt. Dieser Methodenaufruf besteht aus einem „set“+ den jeweiligen Variablennamen. Auf der rechten Seite wird der gleiche Zuweisungswert verwendet.

4.3.5. OOInvocationRule

Mit dieser größeren Regel wird das eigentliche Problem der fehlenden Objektorientierung korrigiert. Der Besucher iteriert dabei über alle „MethodInvocations“, also Methodenaufrufe. Sobald eine Funktion gefunden wird, welche auch wie bei der „JavaTypeRule“ in einer HashMap gespeichert wurde, setzt der Änderungsprozess ein.
Das jeweilige Feld oder die jeweilige Variable wird als „Expression“ an den Anfang gesetzt. Dahinter wird der ebenfalls in der HashMap gespeicherte neue Methodenname gehangen, also über das Feld aufgerufen. Die Parameter werden kopiert und der Methode mitgegeben.
Ein Problem in der fachspezifischen Logik von RPG ist hier jedoch, dass die Reihenfolge der Funktionsparameter variieren kann. So kann es sein dass das Ursprungsfeld, also das Feld auf das die Funktion letztendlich angewendet wird, der erste oder der letzte Parameter ist. Daher musste eine neue innere Klasse „JavaMethod“ erstellt werden, welche anstatt eines Namens in der HashMap abgespeichert wird. Ein Objekt dieser Klasse besitzt nun zusätzlich ein Attribut „sourceVarIsFirstParameter“. Damit wird wie der Name bereits verrät angegeben ob das Ursprungsfeld der erste Parameter ist oder nicht.

* Eine **HashMap** ist eine Struktur welche über einen Key einen schnellen Zugriff auf den zugehörigen Datensatz(value) erlaubt.

4.3.6. INIndicatorsRule

Diese Regel sorgt dafür, dass man die verschiedenen Indikatoren über eine Methode aufrufen kann. Dabei wurde der neue Typ „JavaIndicator“ hinzugefügt. Er erlaubt die Abfrage auf einem Indikator-Objekt anstatt über eine Funktion.
An diesem Beispiel kann man gut die jeweiligen Beeinflussungen der Regeln aufeinander erklären. Ohne diese Regel würde es Kompilierfehler geben. Das liegt daran, dass die Indikatoren weiterhin RPG-Variablen bleiben würden und diese nicht die erforderten Methoden der OOInvocationRule enthalten. Daher müssen beide Regeln in Kombination angewendet werden, um ein kompilierbares Programm zu bilden.

4.4. Finaler Code

```
19 public final class QLKU1RNEU_refactored extends JavaProgram {
20
21     public DSPF qlkuld = createDSPF("QLKU1D", false); // qlkuld
22     public DSPFRecord qlkulrl0 = (DSPFRecord)qlkuld.getRecord("qlkulrl0"); // qlkuld || qlkulrl0
23     public JavaAlpha wwvnam = new JavaAlpha(30, rpgProgram); // qlkuld || qlkulrl0 || wwvnam
24     public JavaAlpha wwnnam = new JavaAlpha(30, rpgProgram); // qlkuld || qlkulrl0 || wwnnam
25
26     public QLKU1RNEU_refactored() {
27         creationInfos = "13.11.2019 15:09:53"; // Transformed by lku
28         setOptimizedDO(true);
29         setOptimizedIF(true);
30         setOptimizedSELEC(true);
31         setOptimizedSearching(true);
32         // file declarations
33         setDSPF(qlkuld);
34
35
36
37     }
38
39     public void main() {
40
41
42
43 // qlkuld (WORKSTN
44     wwvnam.add("HEINZ");
45     wwnnam.add("KLEYNEN");
46     LOOP_4: while (true) {
47         exfmt(qlkulrl0);
48         if (IN03().equals(ON)
49           || wwvnam.equals("ENDE")
50         ){
51             break LOOP_4;
52         }
53         wwvnam.concat("+", 0);
54         if (wwnnam.notEquals(BLANKS)) {sl00();}
55     }
56     INLR().setOn(null, null, rpgProgram);
57
58
59     }
60
61     public void sl00() {
62
63         wwnnam.concat(".", 0);
64     }
65 }
```

Abbildung 4.5.: Das Resultat des Migrators

Wie man in Abbildung 4.5 sehen kann, konnten die meisten Verbesserungen aus der Vision angewendet werden. Das grundsätzliche Ziel der Objektorientierung wurde durch das Anwenden der verschiedenen Regeln erfüllt. Jedoch gab es Änderungen bei ein paar Methodenaufrufen. Da bisher die Funktionen immer auf Seiten des RPG Programms aufgerufen wurden, konnten die Funktionen ohne Weiteres ausgeführt werden. Da aber mit der Abkopplung vom diesem und dem Einführen von Objekten neue Methoden dazukamen, mussten bei einigen als Parameter das RPG Programm mitgegeben werden. Dies ist eine weitere Problemstelle die es zu lösen gilt.

5. Fazit und Ausblick

In dieser Arbeit wurden Grundlagen über die Programmstruktur der VEDA GmbH und die Programmiersprache RPG vermittelt. Es wurden Hintergründe erläutert warum eine Migration zu der damaligen Zeit unumgänglich war. Anhand eines Beispielprogramms wurde ein Prototyp eines Migrators entwickelt, welcher sich an ein Kernproblem der Objektorientierung richtet.
Bei der Programmierung war es besonders entscheidend eine flexible und erweiterbare Lösung zu erstellen, da die behandelten Probleme nur einen Bruchteil der noch vorhandenen Arbeit symbolisieren. Besonders schwierig war es, die einzelnen Probleme in eigenen Rule-Klassen zu sammeln. Denn immer wieder kam es vor, dass eine Funktion ein wenig anders als die anderen funktionierte. So musste man abwägen, ob eine neue Klasse eingefügt, oder die bisherige um die Neuerung erweitert werden musste.
Gerade unverzichtbare Features wie beispielsweise Nullpointer-Safety haben es, dem zeitlichen Rahmen der Seminararbeit geschuldet, nicht in den Prototypen geschafft. Trotzdem sollten diese aber standardmäßig in Java Programmen zu finden sein.

Zukünftig soll das vermittelte Wissen weiterhin ausgebaut und auf größere und signifikantere Probleme angewendet werden. Dabei soll ein bereits vorhandener Prototyp weiterentwickelt werden.

A. Literatur

Boehr, Dirk (2015). *AUTOMATISCHE REFACTORINGS MIT ECLIPSE JDT*. URL: https://www.openknowledge.de/automatische-refactorings-mit-eclipse-jdt/ (besucht am 18. 12. 2019).

Db2 (2019). URL: https://de.wikipedia.org/wiki/Db2 (besucht am 20. 12. 2019).

Dijkstra, Edsger W. (1968). *Go To Statement Considered Harmful*. URL: http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html (besucht am 18. 12. 2019).

Einführung in die RPG/400 Programmierung (2019). URL: http://harl3kin.tripod.com/einf_rpg.htm (besucht am 17. 12. 2019).

Fischer, René (Dez 2016). *DIE ZUKUNFT DER PERSONALABRECHNUNG - AS/400 und die Alternativen*. URL: https://midrange.de/as400-und-die-alternativen/ (besucht am 16. 12. 2019).

Großrechner und Minirechner der mittleren Datentechnik (Nov. 2019). URL: https://de.wikipedia.org/wiki/Minirechner (besucht am 17. 12. 2019).

Hempel, Tino (2006). *Programmiersprachen*. URL: https://www.tinohempel.de/info/info/sprachen/paradigmen.htm (besucht am 17. 12. 2019).

Hollerith und der Lochkartencomputer (Jan. 2018). URL: https://www.planet-wissen.de/technik/computer_und_roboter/geschichte_des_computers/%5Cnewline%20pwiehollerithundderlochkartencomputer100.html (besucht am 17. 12. 2019).

Kleutgens, Harald (Nov. 2010). *Software Development Strategie VEDA GmbH 2010 - Klassische IBM i Anwendungen*. (Besucht am 17. 12. 2019).

Lagotzki, Stefan (2001). *Die dritte Generation: Problemorientierte Sprachen*. URL: http://www.lagotzki.de/scripte/vba/sprachen.html (besucht am 17. 12. 2019).

Operation Codes List (2019). URL: https://www.ibm.com/support/knowledgecenter/en/SSAE4W_9.6.0/com.ibm.etools.iseries.langref.doc/evferlsh259.htm#HDROPERXCD (besucht am 17. 12. 2019).

Programmierparadigma (Aug. 2019). URL: https://de.wikipedia.org/wiki/Programmierparadigma (besucht am 19. 12. 2019).

Referentielle Datenintegrität (2019). URL: https://www.datenbanken-verstehen.de/datenmodellierung/referentielle-integritaet/ (besucht am 19. 12. 2019).

Rother, Dr. Wolfgang (Mai 2015). *NACHWUCHSMANGEL - Sind RPG-Programmierer vom Aussterben bedroht?* URL: https://www.it-zoom.de/dv-dialog/e/sind-rpg-programmierer-vom-aussterben-bedroht-10672/ (besucht am 18. 12. 2019).

Rouse, Margaret (Apr. 2016). *IBM (International Business Machines)*. URL: https://whatis.techtarget.com/de/definition/IBM-International-Business-Machines (besucht am 19. 12. 2019).

RPG (Programmiersprache) (Jan. 2019). URL: https://de.wikipedia.org/wiki/RPG_(Programmiersprache) (besucht am 18. 12. 2019).

Schiffler, Ansgar (2005). *Grundbegriffe zu Programmiersprachen*. URL: http://fbmathe.bbs-bingen.de/Informatik/C_plusplus/grundbegriffe.htm (besucht am 16. 12. 2019).

Schlosser, Hartmut (2019). *Programmiersprachen an US-Unis: Python vor Java*. URL: https://jaxenter.de/programmiersprachen-an-us-unis-python-vor-java-845 (besucht am 20. 12. 2019).

Sprunganweisungen (Okt 2019). URL: https://de.wikipedia.org/wiki/Sprunganweisung (besucht am 20. 12. 2019).

TIOBE-Index: Die aktuellen Top-Programmiersprachen im Ranking (Jan. 2019). URL: https://www.informatik-aktuell.de/aktuelle-meldungen/2019/januar/tiobe-index-die-aktuellen-top-programmiersprachen-im-ranking.html (besucht am 16. 12. 2019).

Visitor Pattern (Okt 2009). URL: https://wiki.thm.de/Visitor_Pattern (besucht am 18. 12. 2019).

Visitor Pattern (2019). URL: https://deacademic.com/dic.nsf/dewiki/1468938 (besucht am 18. 12. 2019).

Vogel, Lars, Simon Scholz und Fabian Pfaff (2018). *Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model*. URL: https://www.vogella.com/tutorials/EclipseJDT/article.html (besucht am 18. 12. 2019).

Was ist ein UML Diagramm (2019). URL: https://www.lucidchart.com/pages/de/was-ist-ein-uml-diagramm (besucht am 19. 12. 2019).

B. Abbildungsverzeichnis

C. Tabellenverzeichnis