Julius-Maximilians-Universität Würzburg

Integration der Connector-Architecture CAPJA in eine Entwicklungsumgebung

– Masterarbeit im Fach Informatik –

vorgelegt von **Mirco Lukas**

Eingereicht bei:	Prof. Dr. Dietmar Seipel
Betreuung durch:	Prof. Dr. Dietmar Seipel, Ludwig Ostermayer
Abgabedatum:	31. Januar 2017

Inhaltsverzeichnis

Abstract

Abstract – English Version

This master thesis is related to multi-paradigm programming with Prolog and Java supported by CAPJA. The latter is an integration framework which provides a semiautomatic interface between these languages and allows Java coders to use the advantages of Prolog like backtracking in an object-oriented environment. CAPJA uses *lambda expressions* newly introduced in Java 8. The software is supported by a fully automated mapping mechanism which converts Java objects to Prolog predicates. To use this program it is not necessary to know the details of Prolog programming since capja actuates the Prolog engine and you work with java classes and objects exclusively. You are not bound to a certain Prolog implementation, too.
Aims of this thesis are providing a user-friendly interface for CAPJA, converting Prolog predicates to Java code with the help of the parser generator ANTLR and interpreting a *domain-specific language* for classes defined in Prolog syntax. To achieve this, an implementation in *Swing* was chosen which is – due to the usage of the *Model-View Controller Design Pattern* – provides easy extensibility. Thereby, one aim is to assist the programmer with creation of *mapper classes* by creating mapper classes automaticly. These classes act as an adapter between Prolog and Java. Optionally, you can provide annotations to assist this process. Finally, evaluated Prolog predicates may be converted to Java.

Kurzbeschreibung – Deutsche Version

Die vorliegende Masterarbeit beschäftigt sich mit der Multiparadigmen-Programmierung mit Prolog und Java, unterstützt durch CAPJA. Dieses Integrationsframework bietet eine halbautomatische Schnittstelle zwischen den beiden Programmiersprachen und gibt Java-Entwicklern somit die Möglichkeit, die Vorteile von Prolog als logischer Programmiersprache (wie bspw. Backtracking) in objektorientierter Umgebung zu verwenden. Hierbei bedient sich die Software den in Java 8 neu eingeführten *Lambda-Ausdrücken*, um Anfragen zu formulieren. Unterstützt wird diese Schnittstelle zudem durch einen vollautomatisierten Mapping-Mechanismus von Java-Objekten zu Prolog-Termen. Hierzu ist es nicht erforderlich, dass der Programmierer selbst mit Prolog im Detail vertraut ist, da CAPJA die Prolog-Engine ansteuert und auf der Java-Seite nur mit gewöhnlichen Klassen und Objekten gearbeitet wird. Ebensowenig ist man an eine bestimmte Prolog-Implementierung gebunden.

Ziele dieser Arbeit sind, eine anwenderfreundliche Schnittstelle für CAPJA zu erstellen, Prolog-Prädikate mittels des Parsergenerator ANTLR in Java-Code zu konvertieren und eine *Domain-Specific Language* für Klassen, die in Prolog-Syntax definiert sind, zu interpretieren. Zur Realisierung der graphischen Oberfläche wurde eine Implementierung in *Swing* gewählt, die allerdings dank des *Model-View-Controller-Entwurfsmusters* leicht erweiterbar ist. Daneben soll die Software den Programmierer bei der Erzeugung von Mapperklassen, die als *Adapter* zwischen Prolog und Java dienen, unterstützen, indem der Vorgang weitgehend automatisiert wird. Dieser Vorgang kann zusätzlich optional durch Annotationen unterstützt werden.

1 Einführung

Es gehört zur modernen Software-Entwicklung, die Stärken verschiedener Programmiersprachen zu verbinden, um die jeweiligen Vorteile nutzen zu können. Dies zeigt sich z. B. bei der Webentwicklung in der Verwendung von PHP auf der Server- und JavaScript auf der Clientseite. Ein großer Vorteil liegt allerdings in der Möglichkeit, nicht nur verschiedene Programmier*sprachen*, sondern Programmier*paradigmen* miteinander interagieren zu lassen.

An der Universität Würzburg wird die Software *CAPJA* im Rahmen der Dissertation von Ludwig Ostermayer entwickelt, ein Framework zur objektorientierten, vereinfachten Multiparadigmen-Programmierung mit Java und Prolog [Ost17, S. 73], die im Abschnitt 2 auf Seite 5 genauer beschrieben wird. Mit CAPJA wird eine einfache Unterstützung von Java-Typen in Prolog und umgekehrt geschaffen, indem Objekte mittels Hilfsklassen (sog. *Mapperklassen*) automatisiert in Prolog-Prädikate umgewandelt werden und umgekehrt. Antworten von Prolog-Goals können in Objekte gekapselt werden. Dies dient der Verbindung zweier wichtiger Programmier-Paradigmen, dem *objektorientierten* und dem *Logikparadigma*.

CAPJA besteht zum aktuellen Zeitpunkt aus einem Set von Tools. Eine übersichtliche Schnittstelle ist Voraussetzung, um das Programm effizient und einfach nutzbar zu machen. Dazu kann eine GUI dienen. Im Laufe dieser Masterarbeit wird eine solche mit Swing entwickelt, eine Bibliothek, die in Java selbst bereits vorhanden und daher auf jedem System lauffähig ist.

Da sich diese Arbeit mit zwei verschiedenen Programmiersprachen beschäftigt, die verbunden werden sollen, ist es erforderlich, die Struktur von Programmen – etwa Klassenobjekte, Annotationen oder Prädikate – untersuchen zu können. Wichtige Werkzeuge hierfür sind der Parsergenerator ANTLR und die *Java Reflection API*. Dabei sollen die Programmierparadigmen – insbesondere Erweiterbarkeit – und die Anwendung von Design Pattern beachtet werden, um eine zukunftsfähige und wartungsfreundliche Software zu erstellen. Beispielsweise wird wert darauf gelegt, dass erzeugte Klassen übersichtlich formatiert sind und die Generierung durch klare Aufteilung in Einzelschritte nachvollziehbar wird. Hierfür wird der Einsatz des Entwurfsmusters *Builder (Erbauer)* in Erwägung gezogen. Das Programm sollte Daten zudem über eine einheitliche Schnittstelle vorhalten.

Schließlich soll CAPJA als Plugin für Eclipse, bei der es sich um eine sehr häufig verwendete Programmieroberfläche für Java handelt, verfügbar gemacht werden.

Strukturierung der Arbeit.

- Das Kapitel 2 auf der nächsten Seite beschäftigt sich mit CAPJA und erläutert die Vorteile der Multiparadigmen-Programmierung.
- Das Kapitel 3 auf Seite 19 erklärt wichtige Grundbegriffe, setzt sich mit dem Parsergenerator ANTLR auseinander und beschreibt die verwendeten Entwurfsmuster (*Design Patterns*).
- Kapitel 4 auf Seite 27 stellt die Module der entwickelten GUI vor.
- In Kapitel 5 auf Seite 37 wird die Implementierung besprochen.
- Das Kapitel 6 auf Seite 55 beschreibt die Entwicklungsumgebung Eclipse und das hierfür entwickelte CAPJA-Plugin.
- Schließlich beschreibt Kapitel 7 auf Seite 61, welche Tests mit der Software durchgeführt wurden, um die Funktionsfähigkeit zu überprüfen.

Einschränkungen. Das Modul *JPLambda*, welches Anfragen an CAPJA zulässt, wird im Rahmen dieser Masterarbeit nicht integriert. Die Menüpunkte werden im entwickelten Eclipse-Plugin noch kein Ereignis auslösen.

2 Mulitparadigmen-Programmierung mit Prolog und Java

Zunächst stellt dieses Kapitel die Programmierparadigmen (Logik- und Objektorientierte Programmierung) einander gegenüber und zeigt, wie diese durch Prolog bzw. Java erfüllt werden. Dann werden zwei bereits existierende Implementierungen, die der Interaktion zwischen Prolog und Java dienen, besprochen. Eine Beschreibung von CAPJA und dessen Zusammenarbeit mit Java schließt das Kapitel ab.

2.1 Übersicht über Programmierparadigmen

Bei Programmierparadigmen handelt es sich um Ansätze, wie man Computer durch Programmierung steuern kann. Es existieren verschiedene Paradigmen (Denkmuster). Prolog wurde dabei nach dem logikbasierten Paradigma entwickelt, Java nach dem objektorientierten. Letzteres ist ein Untertyp des Imperativen Paradigmas. [Mü03b, F. 3ff.] Im Folgenden wird eine kurze Einführung über die Paradigmen i. Allg. gegeben, der Fokus liegt auf den genannten Sprachen.

2.1.1 Logikprogrammierung

Beschreibung von Fakten und Regeln. Logikprogrammierung basiert auf den Regeln des logischen Schließens. Es existieren Fakten und Regeln:

- Fakten sind gesichertes Wissen.
- Regeln dienen dazu, auf neue Fakten zu schließen.

Beide werden in einer Wissensdatenbank gespeichert [wik16b]. Man wendet Regeln (ggf. auch mehrfach) auf Fakten an und erweitert somit immer wieder das Wissen des Systems [Mü03b, F.15].

Prolog als Programmiersprache. Eine häufig verwendete Logikprogrammiersprache ist Prolog, welche Anfang der 1970er Jahre von Alain Colmerauer entwickelt wurde. Fakten werden wie in der BCNF in Listing 2.1 notiert.

```
1 fakt_name '(' argument [',', argument]* ')' '.'
```

Listing 2.1: Aufbau eines Fakts in Prolog

Ein Komma ist dabei als logischen *UND* zu verstehen. Regeln sind wie in Listing 2.2 aufgebaut. Eine Bedingung ist dabei z. B. `Y < 1000`.

```
1 regel_name :-
2     fakt | bedingung
3     [',' fakt | bedingung]* '.'
```

Listing 2.2: Aufbaus einer Regel in Prolog

Um Fakten zu erfragen, kann man sich – wie in jeder Programmiersprache – Variablen bedienen. Bezeichner, die mit einem Großbuchstaben beginnen, werden als Variablen interpretiert; solche mit Kleinbuchstaben oder in einfachen Hochkommata als konstante Ausdrücke, d. h. Prädikate. Es ist die lower_snake_case-Schreibweise üblich.

Inzwischen existiert ein ISO-Standard für diese Sprache, um einen gemeinsamen Nenner für verschiedene Implementierungen zu bestimmen [wik16b]. Eine bekannte, freie Implementierung ist SWI-Prolog (`http://www.swi-prolog.org/`).

Problemlösung mittels Backtracking. Ein Stammbaum könnte in Prolog beispielsweise wie folgt modelliert werden (Variablen sind groß zu schreiben):

```
1 parent('Elizabeth', 'Queen Mum').
2 parent('Elizabeth', 'George').
3 parent('Charles', 'Elizabeth').
4 parent('William', 'Charles').
5
6 grandparent(X, Z) :-
7     parent(X, Y),
8     parent(Y, Z).
```

Listing 2.3: Beispielcode `grandparent` in Prolog [Sei15, S. 77 (verändert)]

Die Anfrage in den Zeilen 6ff. prüft für zwei Personen `X`, `Y` Folgendes:

- *Hat `X` einen Elternteil `Y`?* Hierbei ist `Y` eine neu eingeführte, lokale Variable.
- *Gibt es für das gefundene `Y` auch einen Elternteil, der `Z` heißt?*

Für die Anfragen `grandparent('Charles', 'Queen Mum').` erhält man das Ergebnis `true.`, für `grandparent('Charles', 'Elsiabeth').` hingegen `false.`. Mit dem Aufruf `grandparent(X, Y).` kann man sich alle Ergebnisse ausgeben lassen:

```
?- grandparent(X, Y).
X = 'Charles',
Y = 'Queen Mum' ;
X = 'Charles',
Y = 'George' ;
X = 'William',
Y = 'Elizabeth'.
```

Listing 2.4: Ergebnis der `grandparent(X,Y)`-Anfrage

Bei der Suche bei der Antwort auf die Anfrage findet Backtracking statt. Prolog baut nach dem Lösen eines Goals eine Baumstruktur seiner Wissensdatenbank auf, die mittels SLD-Resolution aufgelöst wird. Gelangt das Prolog-System bei der Auswertung einer Anfrage an einen Punkt, an dem keine Lösung mehr gefunden werden kann, so geht es im Baum soweit wie nötig zurück und sucht nach einem anderen Pfad [Fis]. Diese Verzweigungen bezeichnet man als Choice Points [Ost17, S. 24]. In Abb. 2.1 wird der SLD-Baum für die genannte Anfrage gezeigt.

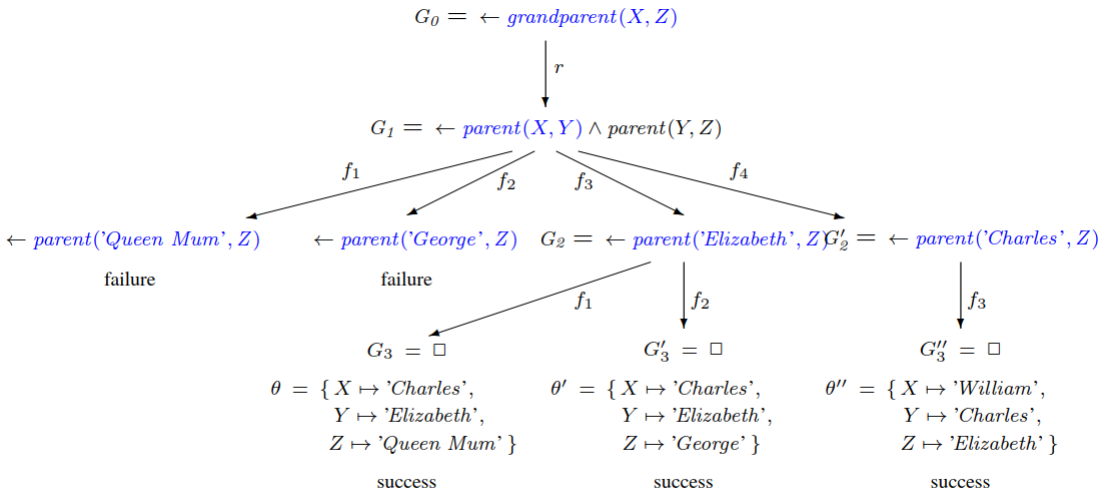

Abbildung 2.1: SLD-Resolutionsbaum für `grandparent(X,Y)` [Sei15, S. 78]

Da für die Anfrage `parent('Queen Mum', Z)` (f_1) keine Lösung existiert, wird der Zweig f_2 betreten. Die Programmiersprache Prolog nimmt nur die Informationen als gegeben an, die in ihrer Wissensdatenbank hinterlegt sind. Mit anderen

Worten: nur Fakten und aus Regeln ableitbares Wissen ist bekannt. Dies bezeichnet man als Closed-World Assumption [Ost17, S. 25].

Vor- und Nachteile von Prolog. Das Logikprogrammierparadigma ermöglicht das schnelle Entwerfen von Programmen. Durch die Turing-Vollständigkeit Prologs können alle Programme entworfen werden, die auch in anderen Programmiersprachen entwickelt werden könnten. Je nach Zielsetzung ist es jedoch ratsam, sich über das zu verwendende Paradigma Gedanken zu machen. Folgende Aspekte sprechen für Prolog [Ost17, S. 36]:

Prolog ist deklarativ: Programme können mit Prädikatenlogik erster Stufe beschrieben werden.

Prolog ist relational: Prädikate können sowohl als Ein- als auch als Ausgabe fungieren. Dies ermöglicht sehr kompakte Programme.

Prolog basiert auf Regeln: Daher ist diese Sprache für Anwendungsfälle geeignet, die stark auf Regeln beruhen.

Prolog ist kompakt: Komplexe Strukturen wie die Großelternbeziehung in Listing 2.3 auf Seite 6 wären in Java nur mit höherem Aufwand zu realisieren, insbesondere die Rekursivität.

Ein Nachteil von Prolog ist neben der fehlenden Typisierung auch die Plattformabhängigkeit, da die meisten Prolog-Varianten in C implementiert sind.

2.1.2 Objektorientierte Programmierung

Imperative Programmierung als Oberbegriff. Imperative Programmierung bezeichnet eine Folge von Anweisungen, die nacheinander ausgeführt werden. Diese strikte Reihenfolge kann z. B. durch Kontrollanweisungen, Schleifen und Unterprogrammaufrufe variiert werden. Informationen werden in Variablen gespeichert, die verschiedene Datentypen besitzen können [Jan05, S. 6].

Die lt. des TIOBE-Index' am häufigsten verwendeten imperativen Programmiersprachen waren in den Jahren 2001 bis 2004 unangefochten Java und C (vgl. Abb. 2.2 auf Seite 10). Nicht zuletzt das Smartphone-Betriebssystem Android, dessen Apps in Java entwickelt werden, dürfte zu diesem Erfolg beigetragen haben [Ost17, S. 37].

Das objektorientierte Paradigma. Ein wichtiges Element dieses Paradigmas ist die Klasse. Sie definiert die Struktur und das Verhalten der von ihr abgeleiteten Objekte, welche auch als Instanzen bezeichnet werden. Jedes Objekt gehört zu

genau einer Klasse [Bal99, S. 23], wobei das Objekt seine Klasse kennt, aber nicht umgekehrt [Bal99, S. 25]. Es können mehrere Instanzen existieren, die jeweils voneinander unabhängig sind. Objekte können mittels Methoden und Funktionen miteinander kommunizieren [BEG16, S. 16]. Sie besitzen zudem einen Zustand, der durch die Werte der Attribute, d. h. Eigenschaften, eines Objekts charakterisiert ist [Bal99, S. 20]. Es kann darüber hinaus eine Hierarchie von Objekten geben, die die Gundprinzipien der Objektorientierten Programmierung ermöglicht [KK15, F. 31ff.]:

Abstraktion: Man fasst mehrere Klassen mit gemeinsamen Merkmalen zu einer allgemeinen Oberklasse zusammen und baut somit eine Hierarchie auf, die man mit *ist-ein* beschreiben kann: Eisbären, Katzen und Hunde *sind-ein* Säugetier und daher sind jene als Unterklassen von `Säugetier` zu implementieren.

Kapselung: Eigenschaften werden an Objekte gebunden und sind nur nach innen sichtbar.

Vererbung: Eigenschaften werden in der Vererbungshierarchie weitergegeben.

Polymorphismus: Unterklassen können Methoden überschreiben und somit unterschiedliches Verhalten modellieren.

Java als Programmiersprache. Java ist – im Gegensatz zu C und genau wie C++ – objektorientiert. Es existieren zwar einige primitive Datentypen, allerdings gibt es auch für sie ein Objekt-Pendant (eine sog. Wrapper-Klasse). Eine Java-Klasse kann Variablen (Member) enthalten, die mittels Methoden verändert werden können. Beispielsweise kann eine Klasse `Person` die Member `vorname` und `nachname` haben und davon können die Instanzen `Person->P1` und `Person->P2` erzeugt werden. Die im Abschn. 2.1.2 auf der vorherigen Seite genannten Paradigmen erfüllt Java wie folgt:

Abstraktion: Jede Klasse hat `java::lang::Objekt` implizit als Oberklasse (außer `Object` selbst). Diese enthält allgemeine Funktionen *aller* Java-Objekte, z. B. eine Methode zum Berechnen des Hashcodes [Ora16].

Kapselung: Java besitzt sog. Modifikatoren, die die Sichtbarkeit einer Methode oder einer Membervariable einschränken: `private`, `public`, `protected` oder paketsichtbar (ohne Modifikator) [Bal99, S. 261].

Vererbung: Java vererbt alle Methoden und Variablen an Unterklassen, sofern dies nicht explizit (etwa mit dem Modifikator `private`) ausgeschlossen wurde.

Polymorphismus: Das Überschreiben in Unterklassen ist möglich, wenn es nicht explizit ausgeschlossen wurde.

Abbildung 2.2: Populärste Programmiersprachen 2001–2014 [TI15]

Vor- und Nachteile von Java. Ebenso wie Prolog gibt es auch für Java Anwendungsfälle, für die es gut geeignet ist und solche, bei der andere Programmiersprachen verwendet werden sollten [Ost17, S. 37f.]:

Java legt Wert auf Sicherheit: Es existiert eine Typprüfung, aber keine Pointer-Arithmetik.

Java ist portabel: Wenn eine Java Virtual Machine für die Zielplattform existiert, so können andere Systeme den gegebenen Code ebenfalls ausführen. Änderungen – wie bei C – entfallen in den meisten Fällen.

Java ist strukturiert: Neben der Einhaltung der erwähnten objektorientierten Paradigmen bietet es ein System von Paketen zur Strukturierung von Code.

Java ist kostenfrei: Der Download ist ohne Gebühren möglich.

Der erstgenannte Punkt ist jedoch zugleich ein Nachteil. Durch den fehlenden direkten Zugriff auf Speicheradressen ist das Programmieren von hardwarenahen Programmen – z. B. Treibern – mit Java nicht möglich. Zudem enthalten Javaprogramme sehr viel Code für Standardprogramme (engl. Boilerplate Code). Dies lässt sich exemplarisch am *Hallo Welt*-Programm zeigen, welches in Prolog (Listing 2.5) bzw. Java (Listing 2.6) gezeigt wird.

```
1 writeln('Hallo Welt!').
```

Listing 2.5: Minimales *Hallo-Welt!*-Programm in Prolog

```
1 public class HalloWelt {
2     public static void main(String[] args) {
3         System.out.println("Hallo Welt!");
4     }
5 }
```

Listing 2.6: Minimales *Hallo-Welt!*-Programm in Java

2.2 Ausgewählte Schnittstellen zwischen Prolog und Java

CAPJA ist nicht das erste Projekt, welches sich mit der Multiparadigmen-Programmierung mit Prolog und Java beschäftigt. In diesem Abschnitt werden exemplarisch zwei bereits bestehende Schnittstellen diskutiert.

2.2.1 Interprolog

Beschreibung. Interprolog verwendet den Java-Serialisierungsmechanismus, um Objekte in Bytestreams umzuwandeln. Auf Seiten Prologs existiert eine Grammatik, die diese Nachrichten analysiert. Prolog wiederum sendet serialisierte Nachrichten, die Java deserialisieren kann. Zur Serialisierung ist es notwendig, dass eine Klasse das Interface `java::io::Serializable` implementiert [Ost17, S. 55]. Die Sprache bietet somit eine bidirektionale Schnittstelle, die auf Seiten Prologs beispielsweise die Funktion `java::lang::String::substring(int, int)` anbietet, was in Listing 2.7 auf der nächsten Seite zu sehen ist. Die einzelnen Argumente definieren Folgendes:

1. das Ziel der Nachricht,
2. das Resultat und
3. die Nachricht selbst [int16].

```
java( string('InterProlog Java bridge'), string(Middle),
   ↪  substring(int(12),int(16)) ).
// Ergebnis: Middle = Java.
```

Listing 2.7: `substring()` in Interprolog

Die Methode `XSBSubprocessEngine::deterministicGoal()` kann etwa von Java aus aufgerufen werden, um Prolog-Code auszuführen (vgl. Listing 2.8).

```
import com.declarativa.interprolog.*;
// ...
ProcessEngine engine = new XSBSubprocessEngine("
   ↪ MyXSBpath");
    if (engine.deterministicGoal(
       "javaMessage(
           'java.lang.System'-out,println(
              string('Hello from Prolog,Java world!')))"
        ))
    System.out.println("This goal succeeded");
    engine.shutdown();
```

Listing 2.8: Beispiel für einen Goal in Interprolog [Ost17, S. 56]

Diskussion. Interprolog verwendet String-Repräsentationen von Prädikaten in Java. Dies hat den Vorteil, dass die Anwendung für erfahrene Prolog-Anwender recht intuitiv ist, für Java-Entwickler ist sie das jedoch i. d. R. nicht. Zudem sind Strings sehr fehleranfällig, insbesondere, wenn sie manuell konkateniert werden. Weiterhin kann die Programmieroberfläche (IDE) keine Autovervollständigung anbieten. Daher wird zum Vergleich im Folgenden JPL als eine objektbasierte Schnittstelle betrachtet.

2.2.2 JPL

Beschreibung. Auch bei JPL handelt es sich um ein bidirektionales Interface. Es basiert auf dem Java Native Interface. Dieses ermöglicht Java, auf in anderen Sprachen wie bspw. C implementierte Funktionen zuzugreifen. Dies kann für

betriebssystemspezifische Funktionen oder effiziente Implementierungen nützlich sein. Diese Funktionen werden mit dem Schlüsselwort `native` gekennzeichnet, wie etwa in der Klasse `java::io::FileInputStream`, die auszugsweise in Listing 2.9 zu sehen ist.

```
1 public native int read() throws IOException;
```

Listing 2.9: Eine `native`-Methode in Java [Ull12]

Goals in Prolog können als String oder mittels spezieller Typen definiert werden, was im Listing 2.10 zu sehen ist. Anfragen von Prolog aus nach Java sind ebenfalls möglich.

```
1 Variable x = new Variable("X");
2 Query q = new Query("father", new Term[] {
3         new Atom("dietmar"), x });
4 while (q.hasMoreSolutions()) {
5     Hashtable solution = q.nextSolution();
6     System.out.println(solution.get("X"));
7 }
```

Listing 2.10: Beispielcode für eine Anfrage in JPL [Ost17, S. 58]

Diskussion. JPL bietet wie andere Frameworks (z. B. PBR4J [Ost17, S. 60f.]) eine objektbasierte Schnittstelle. Diese Implementierung vermeidet die meisten oben genannten Nachteile von Strings, z. B. die fehlende Autovervollständigung. Allerdings entsteht auch hier *Boilerplate*-Code, der das Programm schlecht lesbar macht.

2.3 CAPJA als Integrationsframework für Prolog und Java

2.3.1 Überblick

Trends in Software-Entwicklung. Moderne Software-Entwicklung muss immer produktiver werden. Dies zeigen die relativ jungen Entwicklungsverfahren wie *Extreme Programming* und *Rapid Prototyping*. Unter *Extreme Programming* versteht man eine Variante des Programm-Entwurfs für kleine und mittelgroße Teams, deren Anforderungen sich schnell ändern [Zel01, F. 7]. *Rapid Prototyping* wiederum bezeichnet die Bereitstellung kurzfristig verfügbarer und kostengünstiger Produktprototypen [Hof, S. 1].

Ein Programm, das die schnelle Entwicklung moderner Programme ebenfalls unterstützen kann, ist CAPJA, die Connector Architecture for Prolog and Java. Hierbei handelt es sich um ein Integrations-Famework, welches auf einem halbautomatischen Mechanismus beruht. Dieser wiederum stellt einen vollautomatischen Mechanismus zur Verfügung, um ein sogenanntes Objekt-zu-Term-Mapping zu realisieren.

CAPJA ist vollständig in Java implementiert und vereint die Vorteile beider oben beschriebenen Programmierparadigmen, indem es ermöglicht, mittels eines objektorientierten Interfaces auf Prolog zuzugreifen. Hierbei ist man weder an ein bestimmtes Betriebssystem noch an eine bestimmte Prolog-Implementierung gebunden [Ost17, S. 3] und hat trotz dieser Abstraktionsschicht kaum negativen Einfluss auf die Laufzeit-Geschwindigkeit.

Domänenspezifische Sprachen. Anfragen an die Prolog-Engine werden in einer Domänenspezifische Sprache (Embedded Domain Specific Language , eDSL) notiert. Unter einer DSL i. Allg. versteht man eine Programmiersprache, die für ein bestimmtes Problemfeld (Domäne) geschaffen wurde. Ist die DSL eine Teilmenge einer anderen Sprache, so spricht man von einer eDSL [wik16a]. Capja verwendet die in Java 8 neu eingefügten [Ora16] Lambda-Ausdrücke als eDSL. Diese eDSL wird als *Java-Prolog Query Language* (JPQL) bezeichnet. Somit ist es möglich, Anfragen an Prolog in Java-Syntax zu formulieren.

2.3.2 Angewendete Technologien

Durch das oben erwähnte *Objekt-zu-Term-Mapping* ist ein konfigurierbares Bindeglied zwischen Prolog und Java gegeben. Diese Anbindung beschränkt sich dabei nicht auf ein einzelnes Prolog-System, sondern lässt dem Nutzer die Wahl. Wichtige Eigenschaften von CAPJA sind:

- Es verzichtet auf Bytecodemapping. Einige Prolog-Implementierungen sind in der Programmiersprache C geschrieben und können somit Standardprädikate nicht ohne weiteres mappen. Stattdessen wird eine nachrichtenbasierte Kommunikation verwendet. Somit entfallen auch Anpassungen der *Java Virtual Machine*, die den Code schwer portierbar machen würden.

- Die Repräsentation erfolgt mittels Mappern, die automatisch generiert werden können. Diese werden in Kap. 3.1 auf Seite 19 erklärt. Eine Repräsentation in Form von Strings ist zu fehleranfällig und führt dazu, dass die Vorteile der Objektorientierung nicht mehr genutzt werden können. Stattdessen wird eine Abstraktionsschicht mit Objekten erzeugt. Diese Schicht erforderte vor

CAPJA das Programmieren zusätzlichen Codes (vgl. z. B. Listing 2.8 auf Seite 12).

- Anfragen können in der für Java-Entwickler gewohnten Syntax gestellt werden. Eine mögliche Anfrage ist in Listing 2.11 zu sehen und das Ziel der Anfrage ist auch ohne Kenntnisse über CAPJA oder Prolog verständlich. Dies führt zu leichter wart- und testbarem sowie selbsterklärendem Quellcode.

- CAPJA erlaubt Backtracking in Java.

```
1 JPQuery<Employee> query = new JPQuery<>(
2    employee -> employee.getSalary()      > 5000
3        && (    employee.getFirstName() == "Baker"
4             || employee.getLastName()  == "Baker" ));
```

Listing 2.11: Beispielanfrage mittels `JPLamda`-Ausdruck zur Filterung einer Menge von `Employees` [Ost17, S. 110 *(gekürzt)*]

2.3.3 Aufbau von CAPJA

CAPJA lässt sich auf der Java-Seite in drei Komponenten teilen: JPMapping, JPLambda und JPGateway [Ost17, S. 74]:

JPMapping: CAPJA beruht auf dem bereits erwähnten *Objekt-zu-Term-Mapping*. Diese Komponente reguliert diesen Mechanismus. In Richtung Prolog wandelt sie Objekte in Terme um, in Richtung Java die Ergebnisse von Prolog-Anfragen in Objekte. Zur Steuerung des Mapping-Mechanismus' gibt es die `@JPMapping`-Annotationen, die festlegen, wie eine Klasse gemappt werden soll. Die Annotationen werden in Abschnitt 5.2.1 auf Seite 38 beschrieben. Genauso kann ein Mapping auch durch die *PSN* beschrieben werden. Diese wurde bereits im Kap. 1 auf Seite 3 kurz beschrieben. Näher wird darauf in Abschnitt 4.2.4 auf Seite 31 eingegangen. Ist kein Mapping vorgegeben, wird ein Default-Mapping verwendet. [Ost17, S. 75]

In Grafik 2.3 auf der nächsten Seite wird der Mechanismus noch einmal in Gänze beschrieben.

JPLambda: Anfragen können auch in der Java-Prolog Query Language (JPQL) modelliert werden. Die Anfragen (Querys) werden von der generischen Klasse `JPQuery<T>` gekapselt. Zu beachten ist, dass der gegebene Ausdruck zur Laufzeit *nicht* ausgewertet wird. Stattdessen wird er analysiert und als

Anfrage an Prolog gerichtet [Ost17, S. 77]. Diese Anfrage könnte bspw. so aussehen, wie es Listing 2.12 zeigt. Die Anfrage wird von der Komponente JPCompiler in einen Abstract Syntax Tree (AST) umgewandelt. Dieser wiederum wird anschließend analysiert. Das Ergebnis ist eine Unterklasse von `JPQueryTranslator<T>`, die erneut durch Java kompiliert wird. [Ost17, S. 120]

```
1 JPQuery<Employee> myQuery = new JPQuery<Employee> (
2   employee ->   employee.getSalary() > 3000
3         && employee.getFirstName().contains("ew")
4 );
```

Listing 2.12: Beispiel eines *JPLambda*-Ausdrucks [Ost17, S. 121]

JPGateway: Das Java-Interface *JPGateway* ermöglicht es, CAPJA an mehr als eine Prolog-Implementierung anzubinden. Zu beachten ist, dass sich die anderen genannten Komponenten nur auf diese Schnittstelle beziehen und somit ebenfalls unabhängig sind [Ost17, S. 75].

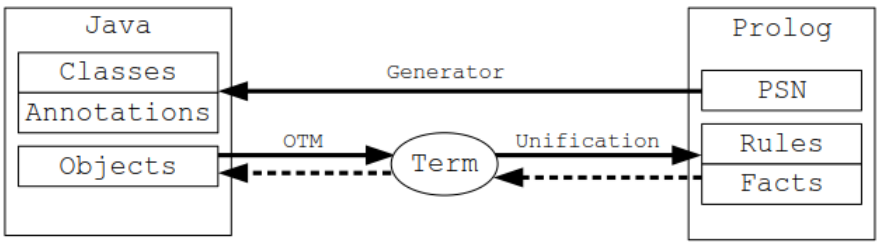

Abbildung 2.3: Die *JPMapping*-Komponente von CAPJA [Ost17, S. 76]

Der erweitere Buildprozess von CAPJA. Anfragen werden in einem erweiterten Java-Buildprozess, der in Abbildung 2.4 auf der nächsten Seite beschrieben wird, vorbereitet, um von Java nach Prolog geschickt werden zu können. Dabei findet *keine* Kompilierung von Java nach Prolog statt; CAPJA bietet lediglich ein *objektorientiertes Interface* zu Prolog an.

Es werden die folgenden Schritte durchlaufen [Ost17, S. 76]:

- Zunächst generiert der Java-Compiler `javac` Java-Bytecode, d. h. `*.class`-Dateien.
- Anschließend werden die Dateien vom `JPCompiler` analysiert, neue Klassen generiert und Modifikationen an bestehenden Klassen vorgenommen.

- Ein weiterer Compiler-Durchlauf generiert die `*.class`-Dateien der neuen und der geänderten Quelldateien.

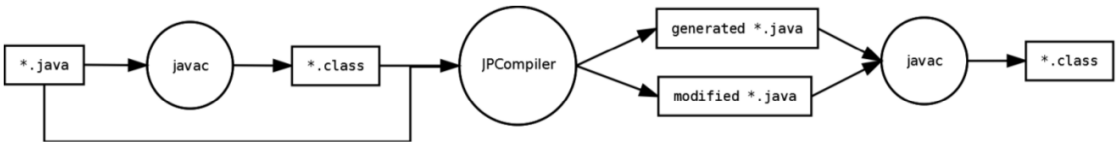

Abbildung 2.4: Der erweiterte Buildprozess in CAPJA mit *JPCompiler* [Ost17, S. 77]

2.3.4 Verwendete CAPJA-Module

CAPJA bietet verschiedene Möglichkeiten der Term-Integration, von der in dieser Masterarbeit jedoch nur die ersten beiden betrachtet werden:

PSN: Klassen können durch die *Predicate-Signature Notation* (PSN) charakterisiert werden. Hierbei handelt es sich um eine Beschreibung von Klassen in Java in Prolog-Syntax, um jene automatisiert generieren zu lassen. Diese wird im Abschnitt 4.2.4 auf Seite 31 beschrieben.

@JPMapping: Mittels Annotationen ist es möglich, Einfluss auf generierte Klassen zu nehmen. Dies beschränkt sich jedoch auf Klassen, auf deren Quellcode man Zugriff hat.

JPML: Im Unterschied zu den Annotationen müssen die Lambda-Ausdrücke, die in der *Java-Prolog Mapping Language* programmiert werden, nicht an die entsprechende Klasse gebunden werden. Dies ermöglicht auch das Mappen von Klassen, die nur kompiliert vorliegen, z. B. Bibliotheksfunktionen.

Darüber hinaus wird ein weiteres Modul entwickelt, dass die Generierung von Klassen (und daraus wiederum Mapperklassen) aus einem gewöhnlichen Prädikat, z. B. `person(first_name, last_name, address(street, city)).`, gestattet.

3 Erläuterung wichtiger Begriffe und Konzepte

Dieses Kapitel soll dem Leser einige Begriffe erklären und eine Einführung in verwendete Design Pattern bieten. Zudem beschäftigt es sich mit dem Parsergenerator ANTLR.

3.1 Begriffsdefinitionen

Einfacher Datentyp. Unter einem einfachen Datentyp verstehen wir einen der folgenden:

- alle primitiven Java-Datentypen wie `float` oder `int`,
- alle dazugehörigen Wrapperklassen,
- Objekte vom Typ `java::lang::String`,
- Objekte von Typ `Enum`.

Komplexer Datentyp. Ein komplexer Datentyp ist in unserem Verständnis ein Referenzdatentyp. Collections, d. h. Klassen, die das Interface `java::util::Collection` implementieren, werden gesondert betrachtet.

Prädikat. Ein Prädikat ist eine Menge von Fakten und Regeln. Ein Prädikat kann n Parameter haben ($n \in \mathbb{N}_0$); n bezeichnet man als die Stelligkeit.

Atom. Ein Atom ist ein nullstelliges Prädikat. [sew16]

Signatur. Unter einer Signatur verstehen wir die Beschreibung eines syntaktisch-semantischen Aufbaus von Prädikaten in Prolog.

Case. Je nach Programmiersprache werden zusammengesetzte Variablen unterschiedlich repräsentiert. Prolog verwendet die lower_snake_case-Schreibweise, während in Java die lowerCamelCase-Schreibweise gebräuchlich ist.

JavaBean. Die offizielle Definition einer JavaBean von Sun Microsystems lautet wie folgt:

> A Java Bean is a reusable software component that can be manipulated visually in a builder tool.
>
> — *[Sun97, S. 9]*
>
> Eine JavaBean ist eine wiederverwendbare Software-Komponente, die mit einer unterstützenden Werkzeugoberfläche visuell verändert werden kann.

Die Methoden, die den Zugriff auf JavaBean ermöglichen, haben die Form `public <Type> getX();` bzw. `public void setX(<Type> x);`, wobei der Name `X` in lowerCamelCase anzugeben ist. Für Boolean-Variablen ist neben der `get`-Variante auch `public <Type> isX();` in Verwendung [Sun97, S. 55].

Reflection-API (Reflexives Programmieren). Eines der bekanntesten deutschsprachigen Bücher für Java-Programmierer, *Java ist auch eine Insel*, definiert Reflections wie folgt:

> Das Reflection-Modell erlaubt es uns, Klassen und Objekte, die zur Laufzeit von der JVM im Speicher gehalten werden, zu untersuchen [...]. Das Konzept der Reflection [...] ist besonders bei JavaBeans oder Hilfsprogrammen zum Debuggen oder bei GUI-Buildern interessant. Diese Programme heißen auch Metaprogramme, da sie auf den Klassen und Objekten anderer Programme operieren. Reflection fällt daher auch in die Schlagwortkategorie »Meta-Programming«.
>
> — *[Ull10b]*

Somit bietet Java von Haus aus eine einfache Möglichkeit, Objekte und ihre Eigenschaften zur Programmlaufzeit zu analysieren.

3.2 Der Parsergenerator ANTLR

3.2.1 Einführung

Bei ANTLR, das momentan in der Version 4 vorliegt, handelt es sich um einen Parsergenerator. Ein solcher Generator erzeugt Programme zur grammatikalischen Analyse und Transformation von Programmiersprachen [wik]. Dabei wird die Syntax mittels einer Grammatik analysiert. Diese wiederum besteht aus einer Liste von Regeln, die die Struktur einer Sprache formen – analog zur natürlichen Sprache. Die Verarbeitung eines Quellcodes durch ANTLR läuft wie folgt ab [Par13, S. 10f.]:

1. Zunächst werden einzelne Zeichen zu Symbolen (Token) zusammengefasst. Dies geschieht durch den Lexer .

2. Der anschließende Vorgang wird als Parsen bezeichnet. Der Parser liest die Tokens ein und erschließt die Struktur der Eingabe. Dies kann z. B. durch einen Parse Tree geschehen, wie es bei ANTLR der Fall ist.

ANTLR wird sowohl im akademischen Umfeld als auch von verschiedenen Firmen, darunter auch Twitter, verwendet [Par13, S. xi].

Ein Beispiel für eine durch ANTLR parsbare Grammik zeigt Listing 3.1. Jede Grammatik muss mit der Zeile `grammar NameDerGrammatik;` beginnen, wobei die Datei den Namen `<NameDerGrammatik>.g4` erhalten muss [Par13, S. 33, 58, 256]. Anschließend folgen Regeln für den Parser und den Lexer. Ersterer akzeptiert nach dem Schlüsselwort `hello` die Eingabe eines Wortes $\omega \in \{a, \ldots, z\}^+$.

```
1 grammar Hello;        // Define a grammar called Hello
2 r : 'hello' ID ;      // match keyword hello followed by
3                       // an identifier
4 ID : [a-z]+ ;         // match lower-case identifiers
5 WS : [ \t\r\n]+ -> skip ;  // skip symbols
```

Listing 3.1: Beispiel für eine ANTLR-Grammatik ([Par13, S. 6])

Für den Quellcode `hello world` entsteht der in Abb. 3.2.1 gezeigte Baum.

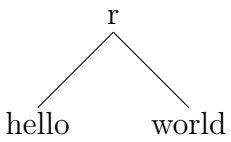

Abbildung 3.1: Parse-Baum zum Programm `hello world` (nach [Par13, S. 7])

ANTLR erzeugt aus der Grammatik neben Lexer und Parser auch Visitor- und Listenerklassen in Java, für unser Beispiel diejenigen in Abb. 3.2 auf der nächsten Seite. Hierdurch kann auf Ereignisse, d. h. das Auftreten bestimmter Strukturen oder Schlüsselwörter, ein Ereignis ausgelöst werden. In dieser Masterarbeit wird lediglich das Visitor-Konzept verwendet. Es wird im folgenden Abschnitt erläutert.

3.2.2 Das Visitor-Konzept

Um ANTLR verwenden zu können, muss der `XBaseVisitor` erweitert werden, wobei `X` dem Namen der Grammatik entspricht.

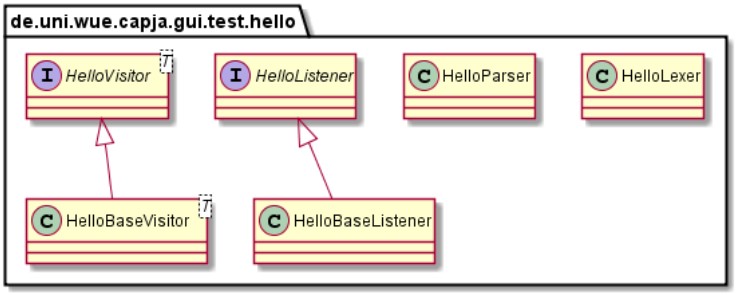

Abbildung 3.2: Klassendiagramm der von ANTLR erzeugten Klassen für das `Hello.g4`-Beispiel

Eine mögliche Implementierung des Visitors zeigt Listing 3.2.

```
1  public class HelloVisitorImplementation extends
   ↪ HelloBaseVisitor<String> {
2
3    public HelloVisitorImplementation() {
4      ANTLRInputStream inputStream =
5              new ANTLRInputStream ("hello world");
6      HelloLexer lexer =
7              new HelloLexer(inputStream);
8      CommonTokenStream tokens =
9              new CommonTokenStream(lexer);
10     HelloParser parser = new HelloParser(tokens);
11     ParseTree tree = parser.r();
12     visit(tree);
13   }
14
15   @Override public String visitR(RContext ctx) {
16     System.out.println("Hello "+ctx.ID().getText()+"!");
17     return super.visitR(ctx);
18   }
19
20   public static void main(String[] args) {
21     new HelloVisitorImplementation();
22   }
23 }
```

Listing 3.2: `HelloVisitorImplementation.java`

In dieser Klasse geschieht Folgendes:

Zeilen 4f.: Liest den Programmcode aus dem übergebenen String aus. Möglich wäre auch, eine Datei als Eingabe zu verwenden.

Zeilen 6–10: Erzeugt diverse Hilfsklassen.

Zeile 11: Definiert den Einstiegspunkt in das Programm, hier die Regel `r`.

Zeile 12: Besucht den Parse-Tree, d. h., das Programm durchläuft den Baum und sucht nach Regeln, um die dafür implementierten Funktionen auszuführen.

Zeilen 15–18: Diese Methode wird aufgerufen, wenn die Grammatikregel `r` gefunden wird. In diesem Beispiel wird `Hello world` ausgegeben.

3.3 Design Pattern

In diesem Abschnitt werden die verwendeten Design Pattern erläutert. Hierbei handelt es sich um wiederverwertbare Software-Entwürfe. Sie helfen beim Erstellen flexibler Programme. Allerdings sollten sie mit Bedacht eingesetzt werden, da sie i. d. R. den Entwurf um zusätzliche Klassen erweitern und somit möglicherweise komplizierter machen [ES10, S. 1f.].

3.3.1 Das Beobachter-Entwurfsmuster

Dieses Pattern, das auch unter dem Namen Observer bekannt ist, kann angewendet werden, wenn ein Objekt über Zustandsänderungen informiert werden soll. Es wird den sog. Verhaltensmustern zugeordnet. Dies ist beispielsweise sinnvoll, wenn ein Datensatz eine visuelle Repräsentation besitzt. Werden die Daten aktualisiert, so soll auch die Repräsentation aktualisiert werden. Dies funktioniert, indem Klassen, die informiert werden möchten, sich beim observierbaren Objekt (ConcreteObservable oder ConcreteSubject) registieren und somit bei Änderungen über die Methode `notify()` informiert werden. Das Pattern existiert in Java im Interface `java::util::Observer` bzw. der Klasse `java::util::Observable`. In Abbildung 3.3 auf der nächsten Seite ist das Klassendiagramm des Patterns zu sehen.

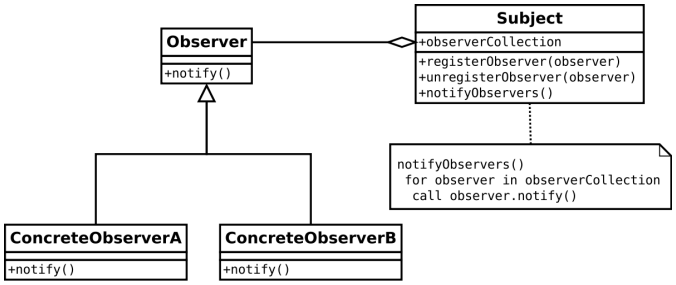

Abbildung 3.3: Klassendiagramm des *Observer*-Patterns [Wik10]

Abbildung 3.4 verdeutlicht mittels eines Sequenzdiagramms, wie die Komponenten miteinander interagieren.

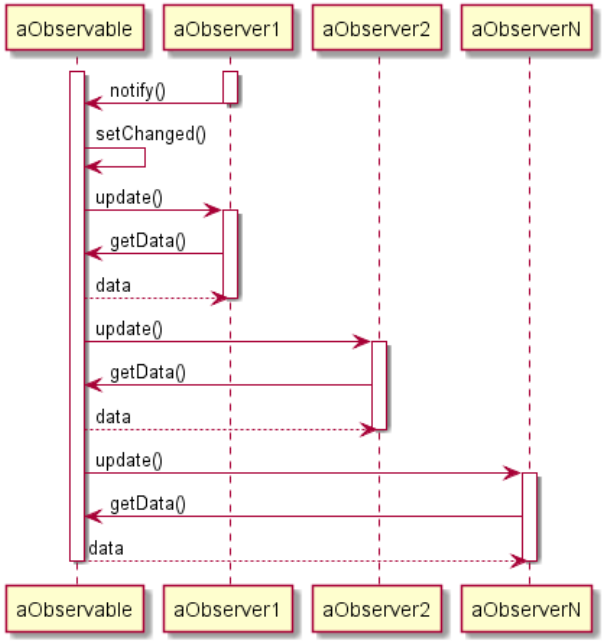

Abbildung 3.4: Sequenzdiagramm des *Observer*-Pattern

3.3.2 Das MVC-Entwurfsmuster (Model-View-Controller)

Dieses Pattern dient dazu, Schnittstellen von der Anwendungsfunktionalität zu entkoppeln. Die einzelnen Bestandteile haben folgende Aufgaben [ES10, S. 77f.]:

Model: Das Model hält die Daten vor. Es existiert genau ein Model je Programm.

View: Die View beinhaltet die visuellen Elemente wie Buttons oder Fenster.

Controller: Der Controller verarbeitet die Benutzereingaben, indem er die Modelldaten und den View ändert. Zu jeder View existiert ein Controller.

In dieser Arbeit wird eine Variante des Patterns benutzt, die Martin Fowler als Passive View bezeichnet [Fow06]; dieses Pattern ist auch unter dem Namen Model View Presenter bekannt [Ezr07]. Hierbei kommunizieren View und Model nicht direkt miteinander, sondern nur über den Controller, wie in Abb. 3.5 ersichtlich ist.

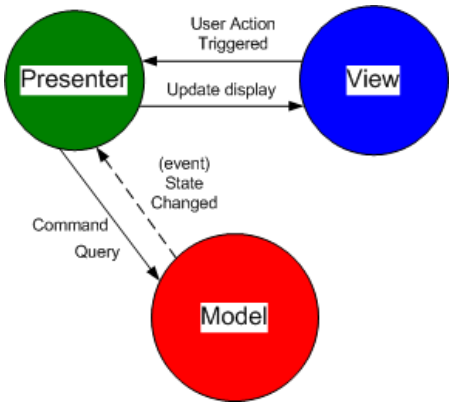

Abbildung 3.5: Veranschaulichung des Model-View-Presenter-Pattern [Ezr07]

3.3.3 Das Erbauer-Entwurfsmuster

Das Erbauer- oder Builder-Pattern wird den Erzeugungsmustern zugeordnet. Die Konstruktion eines Objekts wird dabei in eine eigene Klasse ausgelagert. Dabei soll die Erzeugung Schritt für Schritt geschehen, indem nacheinander verschiedene Eigenschaften des Builder-Objekts gesetzt werden. Zuletzt wird eine spezielle Methode (z. B. `getResult()` oder `build()`) aufgerufen, die ein Objekt aus seinen Teilen zusammensetzt und zurückliefert. Abbildung 3.6 auf der nächsten Seite zeigt das Klassendiagramm. Vor- und Nachteile dieses Patterns sind [ES10, S: 28ff.]:

+ Die Modularisierung erleichtert die Erweiterung,
+ Fehler können behoben werden, ohne dass die vorherigen Schritte des Erbauens wiederholt werden müssen,
+ Neue Builder können leicht integriert werden.
– Es entsteht eine enge Kopplung zwischen den beteiligten Klassen.

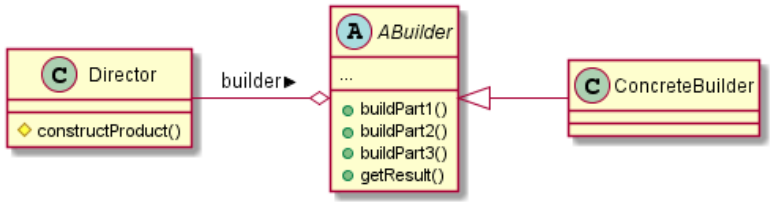

Abbildung 3.6: Klassendiagramm zum Builder-Pattern (nach [ES10, S: 29])

Abbildung 3.7 zeigt ein Sequenzdiagramm, um die Interaktion der beteiligten Klassen zu veranschaulichen.

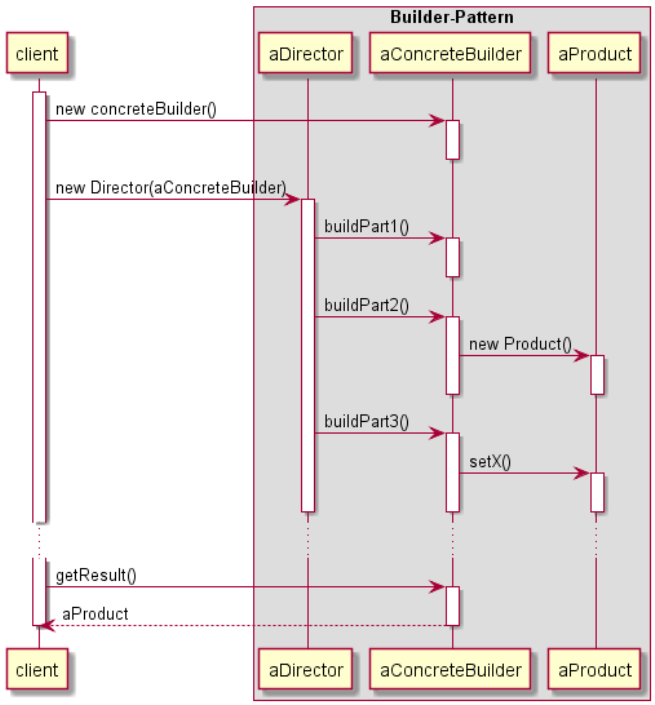

Abbildung 3.7: Klassendiagramm zum Builder-Pattern (nach [ES10, S. 29], angepasst)

4 Die *JPMapperGUI 2.0*

Im Folgenden wird zunächst beschrieben, welche Erweiterungen CAPJA im Rahmen dieser Masterarbeit erhalten hat. Es folgen dann Beschreibungen der einzelnen Module in Kurzform; die technische Seite wird im Kapitel 5 auf Seite 37 beschrieben.

4.1 Erweiterungen CAPJAs durch diese Arbeit

Der Beitrag, den die vorliegende Arbeit leistet, um CAPJA besser zugänglich zu machen, lässt sich in zwei Gruppen unterteilen, nämlich Adaptierung von Prolog nach Java und umgekehrt:

- Java-Klassen sollen durch *strukturelle Transformation* in Prolog-Prädikate überführt werden.
    1. Der Nutzer muss eine komfortable Möglichkeit erhalten, eine Klasse aus dem aktuellen Projekt zur Analyse auszuwählen. Hierzu sollten vertraute, betriebssystemeigene Oberflächen zur Anwendung kommen. Ist eine Klasse mit einer `@JPMapper`-Annotation versehen, so sollen die darin gespeicherten Informationen ausgelesen und verarbeitet werden. Diese Arbeit leisten die Klassen im Paket `fromClassOrAnnotation`.
    2. Nun muss ein Weg gefunden werden, die Struktur der zu analysierenden Klassen zu erlernen. Hierbei ist es wichtig, alle wesentlichen Informationen über die relevanten Variablen in einem einheitlichen Format zu speichern. Zur Analyse werden der Parsergenerator ANTLR und Javas Reflection-Mechanismus verwendet. ANTLR bietet die Möglichkeit, die Quelltextanalyse enorm zu vereinfachen und kann damit die Lücken der Reflection-API schließen. Reflections stellen einen mächtigen Mechanismus zur Quellcodeanalyse dar, können jedoch nicht alle Informationen liefern.

       Die betreffenden Klassen befinden sich ebenfalls im oben genannten Paket.
    3. Die Analyseergebnisse sollen nun in einer interaktiven Oberfläche bereitstehen, in der der Nutzer die Repräsentation der Java-Klasse als

Prolog-Prädikat sieht. Er soll Änderungen am automatisch generierten Code vornehmen und somit das Ergebnis nach seinen Wünschen anpassen können. Allerdings ist dies ein optionaler Schritt, da der vorgegebene Code bereits lauffähig sein muss.

4. Nach dem Generieren liegen gewöhnliche Java-Dateien vor.

Dieser Prozess wird im Rahmen des Plugins in den Eclipse-eigenen Build-Prozess eingebaut.

- Es ist auch möglich, Java-Klassen (mittels der Predicate Signature Notation (PSN) oder aus gewöhnlichen Prädikaten oder Atomen) aus Prolog-Code zu erzeugen.
  1. Aus einer PSN kann sofort eine Java-Klassen erzeugt werden, da jene alle nötigen Informationen beinhaltet. Anschließend kann der Nutzer wie zuvor beschrieben Mapper erzeugen. Der Nutzer wird hierbei durch Dialogfelder geführt. Alle Klassen zur Generierung von Javacode befinden sich im Paket `fromPSN`.
  2. Es wird ein Prolog-Prädikat analysiert, um daraus eine oder mehrere Klassen zu erzeugen. Ein Dialogfeld fragt das Prädikat ab und stellt die zu generierenden Klassen in einer Tabellenstruktur dar. Es besteht die Möglichkeit, Modifikationen an den Klassenvariablen und Konstruktoren vorzunehmen. Für jede dieser Klassen kann auch hier ein Mapper erzeugt werden. Der verwendete Code liegt im Paket `fromPredicate`.

Zusätzlich existieren einige Hilfsklassen wie z. B. ein erweiterter `StringBuilder` und diverse String-Konverter. Diese befinden sich im Paket `basic`.

Der Name *JPMapperGUI 2.0*. Das Modul, das für die Klassengenerierung aus einem Prolog-Prädikat zuständig ist, wurde bereits im Rahmen des Masterpraktikums entwickelt (Version 1.0). Da die in dieser Arbeit überarbeitete Version umfangreiche Erweiterungen enthält, wurde die Versionsnummer 2.0 vergeben.

4.2 Funktionsübersicht

In diesem Abschnitt wird der Funktionsumfang der im Rahmen dieser Masterarbeit entwickelten Software besprochen. Die Implementierung wird in Kap. 5 auf Seite 37 beschrieben.

4.2.1 Das Willkommensfenster

Um einen leichten Einstieg in die Software zu ermöglichen, wurde ein Begrüßungsfenster erstellt, das alle Module zur Auswahl anbietet und das in Abbildung 4.1 zu sehen ist. Diese Module werden in den folgenden Abschnitten beschrieben.

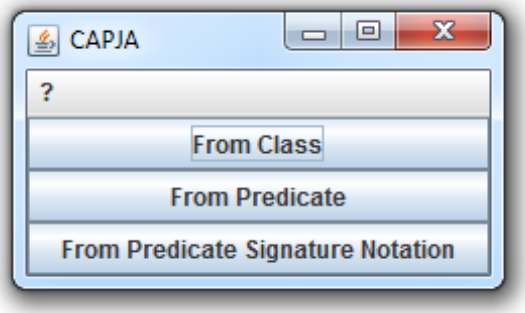

Abbildung 4.1: Willkommensfenster der *JPMapperGUI 2.0*

4.2.2 Konfiguration

Die Datei `config.xml` dient zur Konfiguration der Software. Listing 4.1 zeigt die Standard-Einstellungen.

```
1 <?xml version="1.0" ?>
2 <capja>
3     <gui>
4         <mapperDestination>
5             generated.mapper
6         </mapperDestination>
7     </gui>
8 </capja>
```

Listing 4.1: Beispiel für `config.xml`

Dies Zeilen 4–6 definieren das Zielpaket für generierte Mapper.

4.2.3 Klassengenerierung aus einem Prolog-Prädikat

Dieses Modul basiert auf der im studienbegleitenden Masterpraktikum entstandenen Prototyp, welcher im Wintersemester 2016/17 an der Universitat Würzburg entwickelt wurde. Es wurde für diese Masterarbeit erweitert, sodass es nun eine mächtigere Sprache akzeptiert. Bisher war weder eine Angabe der Paketstruktur

noch eine Umbenennung der resultierenden Klasse möglich und es konnten auch keine Mapper erzeugt werden. Eine bereits existierende Funktion der Software ist es, Java-Klassen aus einem gegebenen Prolog-Prädikat zu generieren. Im Folgenden verwenden wir das Beispiel

```
person(-FirstName, +LastName, ?Age, -address(+Street, Zip, -City)).
```

Die Generierung der Klassen läuft in folgenden Schritten ab:

1. Nach der Auswahl des Menüpunktes *From Predicate* wird der Benutzer nach dem Prädikat gefragt (vgl. Abb. 4.2).

Abbildung 4.2: Eingabemaske für den Klassengenerator

2. Nun wird nacheinander eine Klasse für jedes mindestens einstellige Prädikat generiert. Der Nutzer hat dabei die Möglichkeit, das Ergebnis mit der in Abb. 4.3 auf der nächsten Seite gezeigten GUI zu beeinflussen. Allen Atomen kann man nun einen einfachen Datentyp (diese werden in Abschnitt 3.1 auf Seite 19) zuweisen. Zudem kann man festlegen, ob die Variable schon im Konstruktor gesetzt wird. Hat das Argument den Modifikator `+`, so ist letzteres obligatorisch. Dieser Vorgang wiederholt sich für jede zu erzeugende Klasse.

3. Die Klasse wird erzeugt und der Kompiliervorgang angestoßen, da die kompilierten `*.class`-Dateien für die Generierung eines Mappers bereits existieren müssen, andernfalls wird ein `NoClassDefFoundError` geworfen.

4. Anschließend öffnet sich der Mapper-Konfigurator. Dessen Funktionsweise wird in Kap. 4.2.5 auf Seite 33 beschrieben.

Der Nutzer erhält mit dieser Komponente eine einfache Möglichkeit, Java-Code aus Prolog heraus zu exportieren, ohne Veränderungen am Code vornehmen zu müssen. Dies eignet sich jedoch nur für einfache Terme. Komplexere Termstrukturen können mit dem im folgenden Kapitel beschriebenen PSN-Konverter importiert werden.

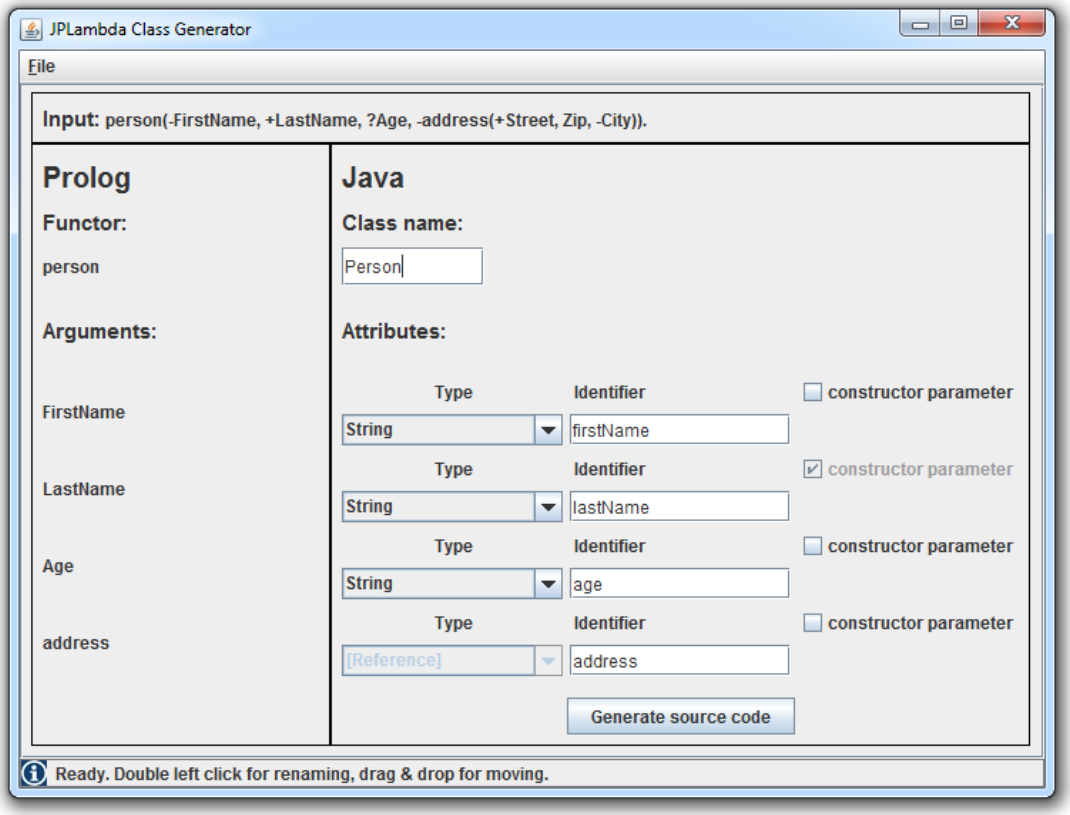

Abbildung 4.3: Konfiguratinsoberfläche für den Klassengenerator

4.2.4 Klassengenerierung aus einer *Predicate-Signature Notation*

Beschreibung der PSN. Bei der Predicate-Signature Notation handelt es sich um eine Notation, die es Prolog-Entwicklern ermöglicht, auf die Generierung von Objekten in Java aus Prädikaten in Prolog Einfluss zu nehmen. Grundsätzlich hat eine PSN den folgenden Aufbau:

```
predicate(+Name, +Type, +Arguments).
```

Die Parameter bedeuten im Einzelnen [Ost17, S. 94]:

+Name: Der Name wird

- in der Form <`Funktor`>/<`Stelligkeit`> notiert, wenn `compound` als Typ gewählt wurde oder
- in der Form <`Funktor`> notiert, wenn es sich um eine `list` handelt.

In beiden Fällen wird der Funktor als Klassenname der generierten Java-Klasse verwendet (sofern erforderlich, wird er von lower_snake_case zu lowerCamelCase konvertiert). Möchte man einen anderen Klassennamen verwenden, so kann man ihn jeweils mit dem Suffix `:Klassenname` definieren; z. B. erzeugt `library/2:'library.library_main'` eine Java-Klasse mit dem Bezeichner `LibraryMain` im Paket `library`.

+**Type:** Der Typ ist genau eines der Elemente aus {`list`, `compound`}. Dies hängt davon ab, ob das Prädikat eine Liste oder ein zusammengesetzter Typ ist. In Java wird der Typ als `List<T>` bzw. als Klasse ausgebildet.

+**Arguments:** Argumente charakterisieren das aktuelle Prädikat. Sie werden in Java als Variablen repräsentiert und können folgende Strukturen haben:

- <`ArgumentName`>,
- <`ArgumentName`>, <`Typ`> oder
- <`ArgumentName`>, `list`, <`Members`>.

+**ArgumentName:** Die Syntax und Semantik entspricht derjenigen von `Name` in `predicate/3`. Gibt es keine weiteren Argumente, so wird ein Java-Objekt des Typs `java::lang::Object` gebildet.

+**Typ:** Der Typ legt den Datentyp fest, der in Java verwendet werden soll. Er kann genau einen der folgenden Werte erhalten (angegeben sind der Prolog- und der Java-Datentyp): {`float` → `Float`, `atom` → `String`, `integer` → `Integer`, `Compound` → `Reference`} [Ost17, S. 86/94]. Dabei wird `compound` durch den Referenzdatentyp ersetzt, den der ArgumentName vorgibt. Im Beispiel in Listing 4.2 verweisen die Argumente auf eine Klasse `Station` bzw. `Line`.

+**Member:** Dieses Argument kann genau dann, wenn das zweite Argument `list` ist, angegeben werden. Es definiert, welchen Typs die Einträge der Liste sind, gibt also den generischen Typ der `List<T>` in Java vor. Beispielsweise kann eine `List<Author>` mit `argument(author_list, list, author)` definiert werden.

```
1 predicate(connected/3:'Connection', compound, [
2     argument(station/1:station1, compound),
3     argument(station/1:station2, compound),
4     argument(line/1, compound)
5 ]).
```

Listing 4.2: Beispiel einer PSN [Ost17, S. 144]

Das Modul *JPPSN2Class Generator*. In Abb. 4.4 ist das Startfenster des Generators zu sehen.

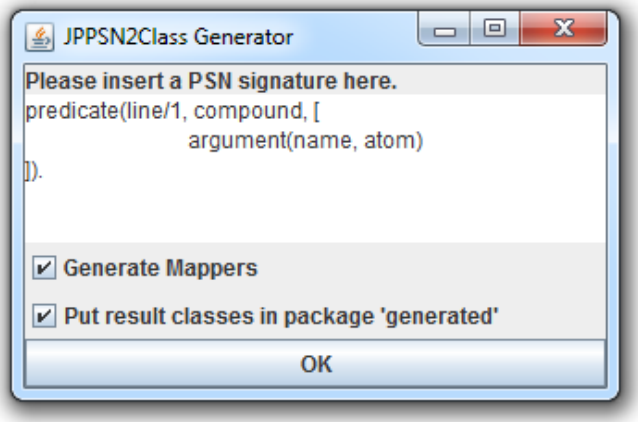

Abbildung 4.4: Der Startbildschirm des *JPPSN2Class*-Generators

Im Textfeld ist bereits Beispielcode vorgegeben (im Screenshot gekürzt). Sollte das Fenster zu klein sein, werden bei Bedarf automatisch Scroll-Balken angezeigt. Weiterhin exisitieren folgende Checkboxen zur Konfiguration des Programms:

Generate Mappers: Ist diese Box aktiviert, so werden nicht nur die Klassen generiert, sondern auch das Fenster geöffnet, in dem die Mapper konfiguriert werden.

Put result classes in package 'generated': Ist diese Box aktiviert, so wird jede erzeugte Klasse in das Oberpaket `generated` gespeichert. Darin wird die Paketstruktur der PSN nachgebildet, d. h. `book/2:'bookpkg.buch'` wird als Klasse `generated::bookpkg::Buch.java` erzeugt.

Nach dem Klick auf *OK* wird der Code mit ANTLR analysiert (Details zur Implementierung folgen im Kapitel 5 auf Seite 37). Wurden Syntaxfehler gefunden, so wird eine Fehlermeldung mit Zeilennummern wie in Abb. 4.5 auf der nächsten Seite angezeigt, die eine einfache Fehlersuche ermöglicht. Anschließend wird der Mappergenerator – sofern ausgewählt – gestartet. Dieser wird im folgenden Abschnitt 4.2.5 kurz beschrieben.

4.2.5 Der Klassengenerator für Mapper

Startet man den Mapper-Generator direkt, um aus einer (annotierten) Klasse einen Mapper zu generieren, so wird man mit der in Abbildung 4.6 gezeigten Meldung

```
line 2:1 mismatched input 'xargument' expecting 'argument'
for input:
---
001  predicate(book/6:'bookpkg.buch', compound, [
002     xargument(title:'Titel', atom),
003     argument(isbn, atom),
004     argument(author_list, list, author),
005     argument(edition, atom),
006     argument(publisher, atom),
007     argument(year, integer)
008  ]).
---
```

Abbildung 4.5: Meldung bei fehlerhafter Eingabe

aufgefordert, einen Ordner auszuwählen, in dem sich die zu mappenden Klassen befinden. Es erscheint anschließend ein Ordner-Auswahldialog.

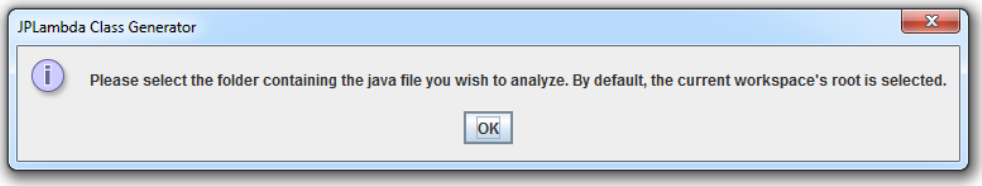

Abbildung 4.6: Die Willkommensmeldung des Generators

Unabhängig davon, ob man eine PSN angegeben oder ein Prädikat oder eine Klasse analysiert hat, kommt man als letzten Schritt zur Generierung der Mapper. Das Hauptfenster ist in Abbildung 4.7 auf der nächsten Seite zu sehen. Es teilt sich in vier Bereiche:

- die *Menüleiste* oben;
- die *Statusleiste* unten;
- der *Dateiexplorer* in der Mitte links, der die Struktur des aktuellen Ordners anzeigt und
- der *Mapperbereich* in der Mitte rechts, in denen eine Mapper-Vorschau angezeigt wird und wo ggf. Änderungen vorgenommen werden können.

Es können nur Änderungen vorgenommen werden, wenn die Mapper nicht aus einer Annotation generiert werden.

Nach einem Klick auf *Generate now... »* werden die Mapper in das Paket `generated::mapper::X` gespeichert, wobei `X` die Paketstruktur der zu mappenden

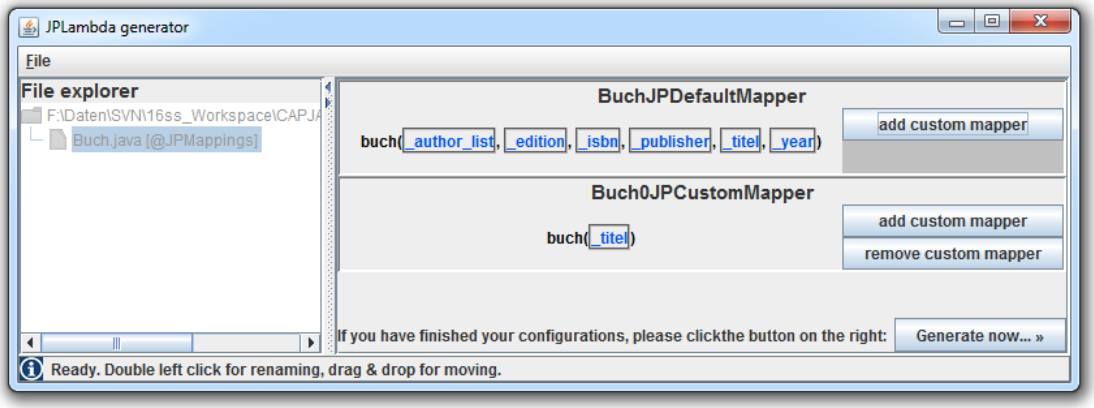

Abbildung 4.7: Der Mapper-Generator

Klasse entspricht. Ein Mapper der Klasse `de::uni::wue::test::Person` wird also in das Paket `generated::mapper::de::uni::wue::test` gespeichert. Das Zielpaket kann durch eine Änderung des Tags `gui/mapperDestination` in der Datei `config.xml` beeinflusst werden (vgl. Abschnitt 4.2.3 auf Seite 29).

5 Implementierung

Dieses Kapitel beschreibt, wie die Software implementiert wurde. Es geht sowohl auf die interne Strukturierung als auch auf die verwendeten Patterns ein. Die Beschreibung wird durch UML-Diagramme unterstützt. Hierbei werden nur die wichtigsten Klassen und Pakete beschrieben. Das Projekt befindet sich im Paket `de:uni::wue::capja::gui`. Dieses Paket-Präfix wird aus Gründen der Übersichtlichkeit bei der Nennung von Unterpaketen weggelassen.

5.1 Das Willkommensfenster

Das Willkommensfenster, das in Abbildung 4.1 auf Seite 29 zu sehen ist, ist eine Implementierung eines `JFrame`. Es dient als Einstiegspunkt in die *JPMapperGUI 2.0* in einheitlicher Gestaltung. Somit sind die Einstellungen für alle Klassen an einem Ort gesammelt.

Wichtige Voreinstellung. Damit Eclipse in Echtzeit ohne Nutzer-Interaktion erkennt, dass neue Klasse generiert worden sind – wie beispielsweise beim Verarbeiten einer *Predicate-Signature Notation* –, muss Eclipse konfiguriert werden: Es muss sichergestellt sein, dass die Checkbox neben *Refresh using native hooks or polling* in den Einstellungen unter [Gerneral]/[Workspace] aktiviert ist (vgl. Abbildung 5.1). Über diese Einstellung wird der Nutzer auch beim Start der *JPMapperGUI 2.0* und ggf. bei der Generierung von Klassen informiert.

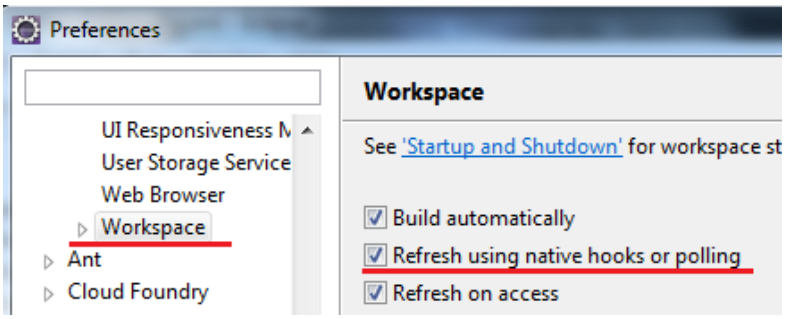

Abbildung 5.1: Einschalten der automatischen Aktualisierung des Workspace

5.2 Das Basispaket

5.2.1 Kleine Werkzeugklassen und -pakete

Die Klasse `basic::MyStringBuilder`. Diese Klasse fügt dem `StringBuilder` einige nützliche Funktionen hinzu. Sie delegiert die meisten Funktionen an einen internen `StringBuilder`, da diese Klasse `final` ist und somit nicht ohne weiteres erweitert werden kann. Der `MyStringBuilder` ist unabhängig von anderen Klassen und kann somit auch flexibel für andere Projekte verwendet werden. Abbildung 5.2 zeigt das zugehörige Klassendiagramm.

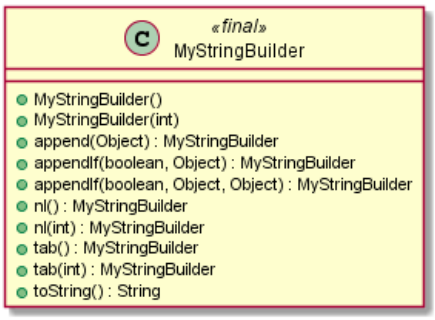

Abbildung 5.2: Die Klasse `basic::MyStringBuilder`

`public class MyStringBuilder`:

- `public MyStringBuilder appendIf(condition, thenObject)`:
  Hängt den zweiten Parameter nur an, wenn die Bedingung `condition` zutrifft.
- `public MyStringBuilder appendIf(condition, thenObject, elseObject)`:
  Hängt den zweiten Parameter nur an, wenn die Bedingung `condition` zutrifft, sonst den dritten.

Das Paket `basic::annotations`. Dieses Paket enthält die CAPJA-Annotationen, welche eine Möglichkeit bieten, Mappings vorzudefinieren. Die Hauptannotation ist `@JPMapping`. Diese Annotationen können dank der Java-Reflection-API zur Laufzeit abgefragt werden, da sie mit `@Retention(RUNTIME)` annotiert sind [Ull10a]. Sie geben die Struktur der Mapper vor und enthalten die im Folgenden beschriebenen Attribute [Ost17, S. 88f.].

`public annotation @JPMapping`:

- `public type id()`:
  Dieses Feld muss angegeben werden. Es dient zur Identifizierung des Mappings und muss daher eindeutig sein.

`public boolean isDefaultMapping()`:
Ist diese bool'sche Variable auf `true`, so verwendet CAPJA dieses Mapping als Standard-Mapping bei der Generierung eines Prolog-Prädikats.

`public String functor()`:
Dieses Attribut überschreibt ggf. den Standard-Funktor.

`public String[] argumentOrder()`:
Wird dieses Feld angegeben, so wird die Reihenfolge der Argumente im entstehenden Prädikat festgelegt. Argumente, die hier nicht auftauchen, werden im Mapping überhaupt nicht berücksichtigt. Fehlt dieses Feld, so wird lexikographisch sortiert.

Das Paket `basic::converters`. Dieses Paket enthält verschiedene String-Konverter, die beispielsweise einen in lower_snake_case gegebenen String in lowerCamelCase umwandeln (Klasse `ConvertToLowerCamelCase`). Da die Koverter an das Interface `IConverter<T>` gebunden sind, ist eine flexible Austauschbarkeit gegeben. Um Ressourcen zu sparen, gibt es eine Sammlung von Konverterintstanzen in der Klasse `ConverterCollection`. Die Klassen sind zudem serialisierbar.

Das Paket `basic::dictionary`. Um das Programm schnell in verschiedenen Sprachen übersetzten zu können, wurde ein XML-basiertes Wörterbuch implementiert. Diese Wörterbuch-Datei muss dem Aufbau in Listing 5.1 entsprechen. Die `locale`-Tags werden für jede Sprache wiederholt; das Attribut `lang` muss genau einem Namen der `Enums` aus der Klasse `dictionary::LanguagesAvailable` entsprechen.

```
1 <dictionary>
2     <locale lang="en_us"> <!-- English -->
3         <item id="entry">entry</item>
4     </locale>
5 </dictionary>
```

Listing 5.1: Beispiel einer minimalen Wörterbuchdatei `dictionary.xml`

Das Paket `basic::genericObserver`. Das in Kapitel 3.3.1 auf Seite 23 erläuterte Observer-Pattern existiert in Java bereits seit Version 1. Es wurde in dieser Arbeit durch eine generische Variante erweitert, die als Nachrichten ausschließlich `Enum<T>`-Variablen erlaubt. Dies hilft bei der Fehlervermeidung, indem es die Menge möglicher Nachrichten an die Beobachter genau definiert und somit einschränkt. Auch dieses Paket kann unabhängig von Capja verwendet werden.

Das Paket `basic::priorityRunnable`. Die Klassen in diesem Paket bieten eine Warteschlange mit eigenem Prioritätstyp an. Die Implementierung nutzt dabei die in Java schon vorhandene `PriorityBlockingQueue`. Eine Klasse, die in diese Warteschlange eingereiht werden soll, muss `IPrioritizedRunnable<T>` implementieren. In dieser Arbeit wird das Paket verwendet, um eine Fenster-Warteschlange zu realisieren. Somit wird verhindert, dass zu viele gleichzeitig geöffnete Fenster existieren.

5.2.2 Variablenabstraktion von und nach Java

Die beiden Pakete `fromJava` und `fromProlog` im Paket `basic::variableWrappers` enthalten Klassen, die die Konvertierungen einer Java-Variable in ein Prolog-Prädikat und vice versa. Ihr gemeinsames Oberpaket `basic::variableWrappers` enthält einige Hilfsklassen.

Das Paket `variableWrappers`. Die `JavaPrologMappingRepresentation` ist die wichtigste Klasse. Sie speichert die Mapper zusammen mit der gemappten Klasse:

`public class JavaPrologMappingRepresentation`:

`public static final String DEFAULT_MAPPER_NAME_PREFIX`:
Enthält das Präfix eines jeden Default-Mappers. Standardwert ist der leere String.

`public static final String DEFAULT_MAPPER_NAME_SUFFIX`:
Enthält das Suffix eines jeden Default-Mappers, Standard ist hier der Wert `JPDefaultMapper`.

`public static final String CUSTOM_MAPPER_NAME_PREFIX`:
Enthält das Präfix eines jeden Custom-Mappers, den leeren String.

`public static final String CUSTOM_MAPPER_NAME_SUFFIX`:
Enthält das Suffix eines jeden Custom-Mappers, per Default `JPCustomMapper`.

`private void fillDefaultMapper(String, Map<>)`:

`private void fillCustomMapper(String, Map<>)`:
Diese Methoden befüllen die Mapper. Dazu werden die in der `Map` gegebenen Variablen analysiert:

1. Für jedes Element aus der `Map<>` wird geprüft, ob eine Variable öffentlich oder eine *JavaBean* ist. Ist sie keines von beiden, so wird eine `NonAccessibleMemberException` geworfen.

2. Je nach Typ der analysierten Variable wird nun unterschiedlich verfahren:

   a) Ist eine Klasse ein einfacher Datentyp (vgl. Abschn. 3.1 auf Seite 19), so wird eine Instanz der entsprechenden Unterklasse von

`Java2PrologVariableAbstract` erzeugt. Diese wird im folgenden Absatz erläutert.

b) Falls eine `Collection<T>` vorliegt, so wird der generische Typ mit `ClassTypes::getGenericType(Class<?>, String)` ausgelesen (siehe unten). Der Algorithmus zur Bestimmung des generischen Typs stammt von [Pog13].

c) Sonst wird eine `JavaPrologMappingRepresentation` erzeugt. Diese speichert alle Variablen, die die ursprüngliche Klasse besitzt, und arbeitet diesen Algorithmus rekursiv ab.

`public class ClassTypes`:
Diese Klasse bietet statische Analysemethoden für `Class<?>`-Objekte.

`public static Class<?> getGenericType(Class<?> c, String s)`:
Diese Methode liest den generischen Typ der gegebenen Variable namens `s` aus der Klasse `c` mittels der Reflection-API aus. Hierbei muss die Variable genau einen generischen Parameter haben und darf nicht geschachtelt sein. Erlaubt sind also `List<String>` und `IPrioritizedRunnable<Boolean>`, verboten `String`, `List<List<String>>` und `Map<String, String>`.

Das Paket `fromJava`. Die Klassen in diesem Paket dienen der Abstraktion von Variablen, die nach Prolog übertragen werden sollen. Je nach Typ der Variablen gibt es verschiedene Subklassen: `Java2PrologVariableArray`, `-Collection`, `-Enum`, `-Reference`, `-SimpleType` und `-String`. Die Klassen stehen wie in Abb. 5.4 gezeigt miteinander in einer Vererbungshierarchie. Ausgewählte Methoden der Klasse `Java2PrologVariableAbstract<>` zeigt Abbildung 5.3, Abbildung 5.2.2 auf der nächsten Seite enthält das Klassendiagramm des gesamten Pakets.

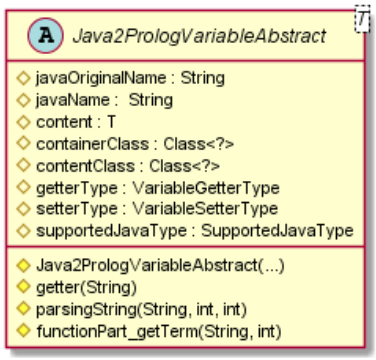

Abbildung 5.3: Die Abstraktionsklasse `Java2PrologVariableAbstract<T>` (in Auszügen)

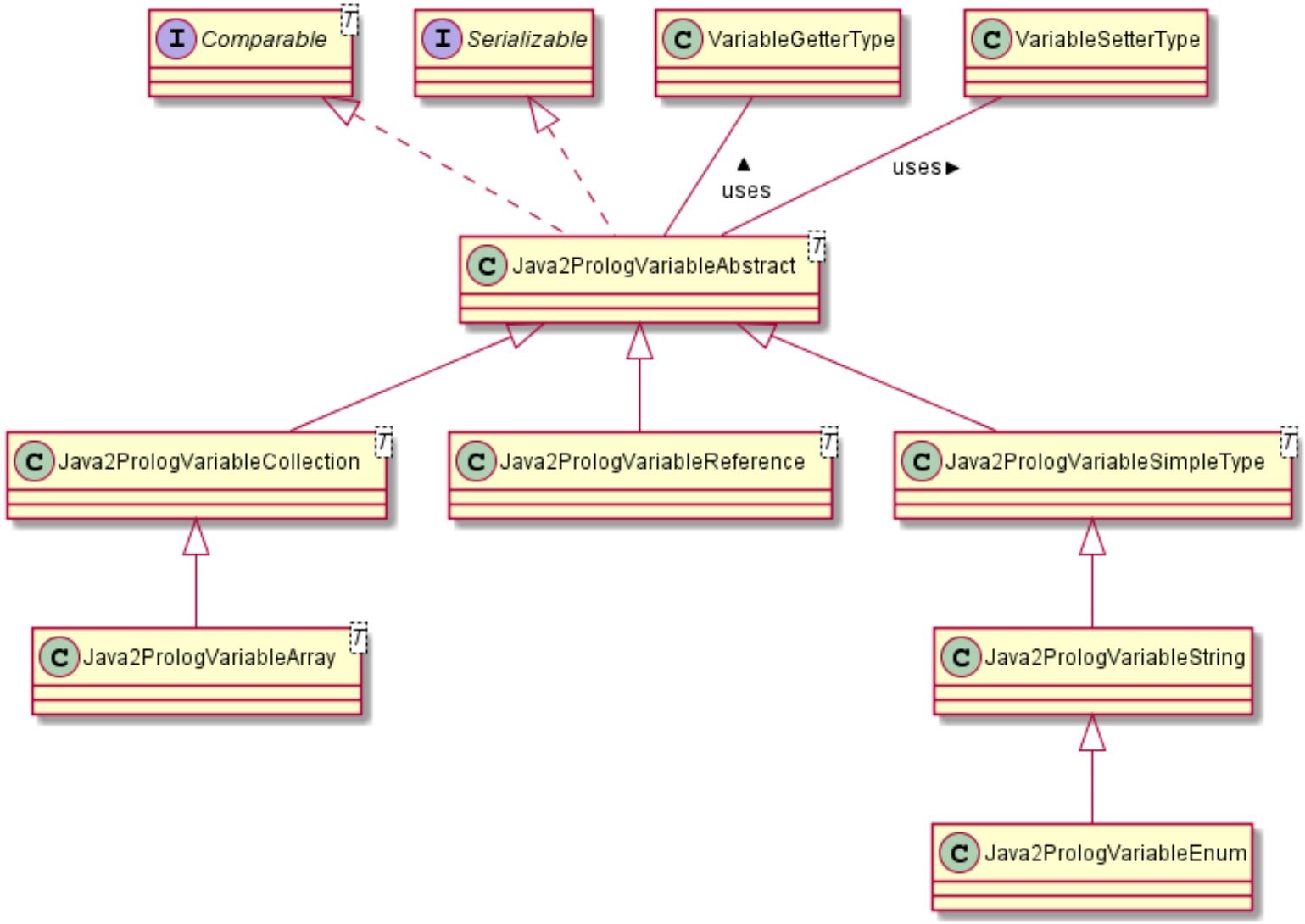

Abbildung 5.4: Die Paketstruktur von `de::uni::wue::capja::gui::basic::variableWrappers::fromJava`

`public class Java2PrologVariableAbstract<T>`:
Bietet eine abstrahierte Darstellung einer Java-Variable. Diese Klasse außerdem bietet Zusatzinformationen wie dem *Getter*-Typ in der ursprünglichen Klasse.

`protected String getter(String instanceName)`:
Generiert einen Getter-String in Abhängigkeit davon, ob eine Variable öffentlich ist oder nicht. Der Parameter `instanceName` gibt den Namen des Objekts an, auf den sich der Aufruf bezieht. Für eine Variable `v` der Instanz `i` sind somit folgende Methodenrückgaben möglich:

- `v.getI()` für eine private Variable mit dem Standard-Getter,
- `v.isI()` für eine private Variable mit dem Standard-Getter für Boolean-Variablen,
- `v.i` für eine öffentliche (`public`) Variable.

Dies ermöglicht den Zugriff auf die Variable über ein definiertes Interface. Analog existiert eine Setter-Methode.

`public String getParsingStringForSimpleTypes(String, int, int)`:
Da die Informationen über eine Klasse in einem Mapper (z. B. die Methode `JPMapper::getInstanceFromBindings()`) als `List<String>` vorliegen, müssen sie ggf. in einen anderen Datentyp geparst werden. Diese Methode liefert für einen gegebenen simplen Typ den Parse-String zurück. Der erste Parameter ist der zu parsende Text. Beispielsweise liefert die Funktion für eine Float-Variable und dem Aufruf `get...("argList.get(0)", 1, 1))` den String `Float.parseFloat(argList.get(0))` zurück.

`public String functionPart_...()`:
Diese Methoden liefern jeweils den entsprechenden String für den Mappergenerator, z. B. `functionPart_getTermFromInstance()`. Eine Implementierung wird exemplarisch in Listing 5.2 gezeigt.

```
1 @Override
2 public String functionPart_getTermFromInstance(String
    ↪ mappedClassLowerInitial, int tabulator) {
3     return new MyStringBuilder(50)
4         .tab(tabulator)          // Einrückungstiefe
5         .append("getSimpleArgumentFromString(")
6         .append(getter(mappedClassLowerInitial))
7         .append(")")
8         .toString();
9 }
```

Listing 5.2: Die Methode `functionPart_getTermFromInstance()` der `Java2PrologVariableString` (zusätzlich kommentiert)

Durch die Kapselung der Variablen und unter Ausnutzung der Polymorphie existiert somit eine flexible und leicht anpassbare Unterstützung bei der Mapper-Generierung.

Das Paket `fromProlog`. In diesem Paket befinden sich – entsprechend dem Paket `fromJava` – Klassen, die zur Abstraktion von Prolog-Variablen dienen.

5.2.3 Der `CapjaMapperBuilder`

Der CapjaMapperBuilder gehört zu den wichtigsten Klassen dieser Software. Aus den automatisch oder manuell erstellten Mappern in der Mapper-Vorschau (Abschnitt 5.5 auf Seite 49) erzeugt der Builder die Mapper-Klassen. Dabei arbeitet er eng mit den Klassen im Paket `variableWrappers::fromJava` zusammen. Auch hier werden nur ausgewählte Konstanten und Methoden beschrieben.

`public class CapjaMapperBuilder`:

Generiert Mapper für CAPJA aus `Java2PrologVariableAbstract`-Instanzen.

`public static final String COLLECTION_MAPPER_NAME`:

Der vollqualifizierte Name des Collection-Mappers: `de.uni.wue.capja.mapping.JPCollectionMapper`

`private static final String CONFIG_XML`:

Der Pfad zur Datei `config.xml`, in der u. a. das Zielpaket für generierte Mapper hinterlegt ist: `config.xml`

`private static final String GENERATE_DEST_FALLBACK`:

Der relative Pfad, in den die Mapper generiert werden sollen, falls die Datei `config.xml` nicht existiert oder nicht ausgelesen werden kann. Standardwert ist `generated.mapper`.

`private String getPkgDeclaration`:

Eine Hilfsmethode, die z. B. `package generated.mapper;` zurückliefert.

`private String destinationPkgFromXml`:

Diese Methode liest das Zielpaket aus der `config.xml` aus. Falls dort ein Fehler (d. h. eine Exception) auftritt, wird `GENERATE_DEST_FALLBACK` zurückgegeben.

`private String function_X()`:

Jede dieser Methoden ($X \in \{$`getTermFromInstance`, `getTerm`, ...$\}$) besitzt eine korrespondierende Methode `functionPart_X` in der bereits besprochenen Klasse `Java2PrologVariableAbstract<T>`. Diese Methoden liefern jeweils den Mapper-Code in AbhängiTgkeit des Datentyps, den die Klasse kapselt.

5.3 Klassengenerierung aus einem Prolog-Prädikat

Die Analyse der eingegebenen Prädikate erfolgt mittels ANTLR, welches in Abschnitt 3.2 auf Seite 20 eingeführt wurde. Im Folgenden werden die erstellte Grammatik beschrieben und zugehörige Klassen diskutiert.

Die ANTLR-Grammatik zur Analyse von Prolog-Prädikaten. Die Grammatik in Listing 5.3 hat den Einstiegspunkt `predicate`. Sie akzeptiert ein Prädikat mit einem Namen und optional einem oder mehreren Argumenten. Auf die mit der Syntax `Bezeichner=RegelName` benannten Parameter kann in der Java-Implementierung mittels eines `XArgumentContext` zugegriffen werden, wobei `X` für den Namen der Regel steht. Dies wird im folgenden Abschnitt verdeutlicht.

```
1  grammar PrologPredicate;
2
3  predicate
4      : predicateName '(' ( predicateArgument (','
         ↪ predicateArgument)* )? ')' '.'?
5      | predicateName '.'? ;
6
7  predicateName
8      : LOWER_CASE_CHAR ( LOWER_CASE_CHAR |
         ↪ UPPER_CASE_CHAR | INT | '_' )*
9      | '\'' stringContent '\'' ;
10
11 predicateArgument
12     : (theModifier=('+' | '-' | '?'))? theName=
         ↪ modifierName
13     | (theModifier=('+' | '-' | '?'))? newPredicate=
         ↪ predicate ;
14
15 modifierName
16     : UPPER_CASE_CHAR ( LOWER_CASE_CHAR |
         ↪ UPPER_CASE_CHAR | INT | '_' )*
17     | '\'' stringContent '\'' ;
```

Listing 5.3: Die Grammatik `PrologPredicate.g4` (in Auszügen)

Zu beachten ist, dass die Reihenfolge der Regeln wesentlich ist. Vertauscht man sie, so wird u. U. eine andere Sprache akzeptiert. Ein Beispiel für dieses Problem ist unter [Kie14] zu finden. Grund ist, dass gelegentlich mehrere Regeln auf eine

Eingabe passen. ANTLR wählt dann die erste Alternative aus [Par13, S. 15]. Dieses Problem wird mit dem Beispiel in Listing 5.4 verdeutlicht.

```
1 BEGIN : 'begin' ; // match b-e-g-i-n sequence;
2                   // ambiguity resolves to BEGIN
3 ID : [a-z]+ ;     // match one or more of any lowercase
```

Listing 5.4: Beispiel einer mehrdeutigen ANTLR-Gramamtik [Par13, S. 15]

`package fromPredicate.analyzer`:

Enthält die Dateien, die die Analyse eines Prädikats ermöglichen.

`public class PrologPredicateVisitorImplementation`:

Diese Klasse implementiert den ANTLR-Visitor.

`public Void visitPredicateArgument(PredicateArgumentContext)`:

Diese Methode erhält ein `PredicateArgumentContext`-Objekt mit den folgenden öffentlichen Variablen:

theModifier: Dies ist genau eines der Elemente {+, –, ?}. Es wird mittels `PrologIOModifier.valueOf(theModifier.charAt(0))` in ein Java-Objekt konvertiert.

theName: Entspricht dem Namen des Prolog-Arguments.

nestedPredicate: Entspricht einem verschachtelten Prädikat, sofern vorhanden.

Aus diesen Informationen werden Objekte der Klasse `Java2PrologVariable` erzeugt, die in Abschnitt 5.2.2 auf Seite 40 erläutert wurden.

5.3.1 Die graphische Oberfläche

Diese Oberfläche ist nach dem in Abschnitt 3.3.2 auf Seite 24 beschriebenen MVC-Pattern erstellt.

Das Model und der Controller. Für diese graphische Oberfläche existiert eine Klasse, die die Daten vorhält (*Model*) und eine abstrakte Oberklasse, die die Controller-Funktionen zur Verfügung stellt. Die Implementierung wird am Beispiel des Mapper-Generators in Abschnitt 5.5.2 auf Seite 51 erläutert. Auch dieses Modell ist ein `GenericObservable<S, T>`, der Controller ein `GenericObserver<S, T>`.

Die View. Diese View besitzt drei Bereiche, die in Abb. 4.3 auf Seite 31 zu sehen sind. In der oberen Zeile wird das eingegebene Prädikat angezeigt. Ist es geschachtelt, so wird nur die Schachtelungsebene angezeigt, die momentan bearbeitet wird, und die untergeordneten Ebenen. Im angegebenen Beispiel würde also `address` nochmals in einem eigenen Fenster angezeigt werden.

In der rechten Spalte werden die ursprünglichen Argument-Namen in Prolog angegeben, rechts davon die Zielvariable in Java. Die Auswahl des Typs beschränkt sich auf einfache Datentypen (vgl. Abschnitt 3.1 auf Seite 19), es sei denn, ein Argument besitzt selbst Argumente und ist somit komplex.

Nach dem Klick auf *Generate Source Code* werden die Klassen generiert und das Model durch den Controller angewiesen, den Mapper-Builder zu starten. Dieser wird in Abschnitt 5.5 auf Seite 49 beschrieben.

5.3.2 Die *Builder*-Klasse

Die Klasse `fromPredicate::model::FromPredicateClassBuilder` bietet u. a. folgende Methoden an:

`public class FromPredicateClassBuilder`:

Implementiert das Builder-Pattern und abstahiert somit die Klassengenerierung.

`public void addVariable(Prolog2JavaVariable)`:

Diese Methode liest die Getter und Setter aus der übergebenen Variable aus und sichert diese Informationen in String-Listen. Zudem speichert sie ggf. die Information, dass die Variable im Konstruktor gesetzt werden muss und erstellt den entsprechenden Code für den Konstruktorrumpf (`this.x = x;`).

`public void buildClass()`:

Diese Methode nutzt einen `StringBuilder`, um nacheinander die Imports, die Klassendeklaration usw. zu einem String zusammenzufügen und in eine `Map<File, String>` zu speichern, die jeweils die Zieldatei derem Inhalt zuordnet. Die Dateien werden mit `writeFiles()` erzeugt.

5.4 Klassengenerierung aus einer *Predicate-Signature Notation*

5.4.1 Analyse mittels der ANTLR-Grammatik

Auch die *Predicate-Signature Notation* wird durch einen mit ANTLR generierten Parser analysiert. Daher wird analog zum Abschnitt 5.3 auf Seite 45 die Grammatik in Auszügen beschrieben. Die Regel `predicateSignatureNotationList` bildet

den Einstiegspunkt der Grammatik und besagt, dass mindestens eine *Predicate-Signature Notation* definiert werden muss. Eine `predicateSignatureNotation` ist jeweils so aufgebaut wie bereits im Kapitel 4 auf Seite 27 beschrieben:

- Sie beginnt mit dem Schlüsselwort `predicate`.
- Anschließend werden der Funktor, die Arität und ggf. ein abweichender Name in Java definiert.
- Schließlich folgen die Argumente.

Im Listing 5.5 wurde aus Platzgründen

```
LOWER_CASE_CHAR | UNDERSCORE | UPPER_CASE_CHAR | SINGLE_INTEGER
```

durch `X` ersetzt.

```
1  plViewType : 'compound' | 'list' ;
2
3  VIEW_OR_ARG_NAME :  LOWER_CASE_CHAR ( X )*
4      | '\'' ( ( X ) ( '.'? ( X ) )? )+ '\'' ;
5
6  predicateSignatureNotationList :
       ↪ predicateSignatureNotation+;
7
8  predicateSignatureNotation
9      : 'predicate' '(' theFunctorName=VIEW_OR_ARG_NAME
          ↪ '/' theArity=integer (':' theJavaClassName=
          ↪ VIEW_OR_ARG_NAME)?
10                        ',' theType=plViewType
11                      (',' theArgs=listOfArgs)?
12                   ')' '.'? ;
13
14 argument : 'argument' '('
15              theArgPrologName=VIEW_OR_ARG_NAME
16              ('/' theArgArity=integer)?
17              (':' theArgJavaName=VIEW_OR_ARG_NAME)? ','
18              theType=type
19              (',' theViewName=VIEW_OR_ARG_NAME)?
20          ')' ;
```

Listing 5.5: Die Grammatik `PredicateSignatureNotation.g4` (gekürzt)

5.4.2 Die graphische Oberfläche

Auch diese Oberfläche ist nach dem in Abschnitt 3.3.2 auf Seite 24 beschriebenen MVC-Pattern erstellt. Es exisiteren – analog zu der Komponente zur Generierung aus einem Prolog-Prädikat – ein Model, welches mit den Controllern interagiert.

5.4.3 Die *Builder*-Klasse

Die Klasse `fromPSN::model::FromPsnClassBuilder` hat neben den üblichen Aufgaben eines Builders die Funktion, sicherzustellen, dass höchstens eine Annotation `@JPMapping` existiert. Andernfalls müssen die Annotationen `@JPMapping` noch durch `@JPMappings` umschlossen werden, da es momentan keine Unterstützung für multiple Annotationen der `@JPMapping`-Annotationen gibt.

5.5 Der Klassengenerator für Mapper

Im Paket `fromClass` befindet sich das Modul, das sowohl Analysewerkzeuge für Annotationen und nicht-annotierte Klassen als auch eine graphische Oberfläche zur Generierung der Mapper bietet. Die Oberfläche wurde bereits in Abb. 4.7 auf Seite 35 gezeigt. Nachfolgend werden die einzelnen Pakete mit ihrer Funktion sowie ausgewählte Klassen beschrieben. Zu beachten ist, dass der Generator vollkommen unabhängig von CAPJA verwendet werden kann (Standalone).

5.5.1 Das Paket `fromClass::analyzer`

In diesem Paket befinden sich Klassen, die Klassen mittels Reflections (vgl. Abschnitt 3.1 auf Seite 19) analysieren. Hierbei werden die Klassen in zwei Gruppen unterteilt: solche mit Annotationen und solche ohne. Dies wird im Klassendiagramm in Abbildung 5.5 beschrieben.

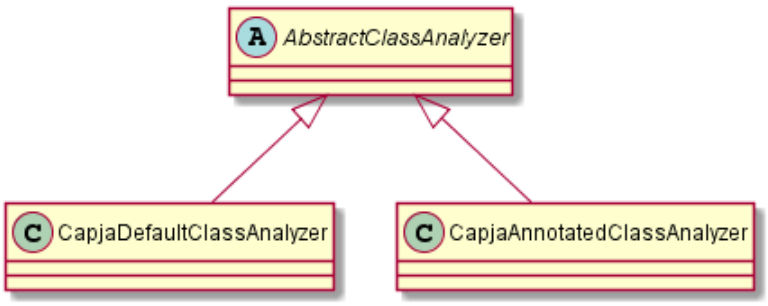

Abbildung 5.5: Klassendiagramm des Pakets `fromClass::analyzer`

`public abstract class CapjaAbstractClassAnalyzer`:
Bieter grundliegende Funktionen zur Klassenanalyse für CAPJA.

`protected Comparator<String> mapperOrder`:
Definiert die Sortierung der gemappten Variablen. Der Standardwert dieser Variablen ist `(s1, s2)-> s1.compareTo(s2)`, also die Standard-Stringsortierung.

`protected Predicate<String> filterVaribleNames`:
Nur Variablen, die dieses Prädikat akzeptiert, werden in die Liste der Variablen aufgenommen. Der Standardwert ist `any -> true`, d. h., dass alle Variablen zugelassen sind.

`protected Map<String, Class<?>> getPublicMembers(Class<?>)`
`protected Map<String, Class<?>> getPrivateMembers(Class<?>)`:
Diese beiden Methoden analysieren die Klasse mittels Reflection auf öffentliche bzw. private Member-Variablen. Hierzu wird die Funktion `getClass().getDeclaredFields()` verwendet. Es werden nur Variablen, die den Test `filterVaribleNames` bestehen, übernommen.

`public class CapjaDefaultClassAnalyzer`:
Diese Klasse analysiert Klassen ohne `@JPMapping`-Annotation.

`public JavaPrologMappingRepresentation createRepresentation()`:
Diese Methode liest alle Membervariablen aus der Klasse und generiert eine Repräsentation in Form der in Abschnitt 5.2.2 auf Seite 40 erläuterten Variable. Mittels Reflection wird für jede Variable ausgelesen, welchen Typ sie hat und ob sie eine JavaBean ist. Falls sie den Test des Prädikats `filterVaribleNames` besteht und eine Bean- oder öffentliche Variable ist, so wird sie in einer Liste gespeichert, um später weiterverarbeitet zu werden.

`public class CapjaAnnotatedClassAnalyzer`:
Diese Klasse analysiert Klassen mit `@JPMapping`- oder `@JPMappings`-Annotation.

`public JavaPrologMappingRepresentation createRepresentation()`:
Diese Methode iteriert über alle `@JPMapping`-Annotationen mittels folgendem Algorithmus:

1. Wenn eine Annotation `argumentOrder` existiert, so werden die dort angegebenen Variablen
   a) als zu mappend übernommen, d. h. `filterVaribleNames` wird überschrieben, sodass dieser `Comparator<>` die Reihenfolge der `argumentOrder` einhält und
   b) in `filterVaribleNames` übernommen, sodass ausschließlich sie erlaubt sind.

   Andernfalls werden die gleichen Variablen wie im `CapjaDefaultClassAnalyzer` verwendet.

2. Die Klasse wird nun wie bei `CapjaDefaultClassAnalyzer` analyisiert.

Anschließend werden der Default-Mapper und ggf. die Custom-Mapper generiert.

Die in Abb. 4.7 auf Seite 35 gezeigten Mapper sind beispielsweise durch die Annotationen in Listing 5.6 definiert. Zu beachten ist, dass sich die Bezeichner der `argumentOrder` auf die Bezeichner der Membervariablen der annotierten Klasse bezieht. Sofern eine genannte Variable nicht existiert, wird eine Exception ausgelöst.

```
1  @JPMappings ({
2      @JPMapping (
3          id = "book/6",
4          functor = "book",
5          argumentOrder = {"titel", "isbn", "authorList",
               ↪ "edition", "publisher", "year"},
6          isDefaultMapping = true      ),
7      @JPMapping (
8          id = "book/1",
9          functor = "book",
10         argumentOrder = {"titel"}    )})
```

Listing 5.6: `@JPMappings` zur Generierung von Mappern

5.5.2 Die graphische Oberfläche

Alle graphischen Oberflächen, die im Rahmen dieser Masterarbeit entstanden sind, folgen dem *Model-View-Presenter*-Pattern, das bereits in Abschnitt 3.3.2 auf Seite 24 beschrieben wurde.

Das Model. Im Paket `fromClass::model` findet sich die Model-Klasse. Sie ist ein `GenericObservable<>` und kann somit von anderen Klassen beobachtet werden. Diese werden dann über Änderungen informiert (vgl. Beschreibung des Observer-Patterns in Abschnitt 3.3.1 auf Seite 23).

Der Controller. Die Klasse `FromGuiAbstractSupervisingController` bildet die Oberklasse der Controller. Alle Implementierungen müssen diese Klasse erweitern. Die wichtigste Methode ist `update()`.

`public class FromGuiAbstractSupervisingController`:

Diese Klasse bildet den Controller im MVC-Pattern, das in Abschnitt 3.3.2 auf Seite 24 beschrieben wurde.

`public final void update(FromGuiModel, FromGuiModelConstants)`:
Diese Methode wird vom Model aufgerufen, das als `Observable` fungiert. Es übergibt sich selbst als ersten Parameter. Der zweite Parameter ist die ausgeführte Aktion, z. B. `MAPPERS_GENERATED`.

Je nach Aktion wird eine der im Klassendiagramm in Abb. 5.6 abgebildeten Methoden aufgerufen. Der Controller liest die geänderten Daten aus dem Model und gibt sie an die aufgerufene Methode weiter.

Es existiert eine Klasse `FromGuiControllerDefaultImpl`, die alle Methoden implementiert, aber lediglich Meldungen auf der Konsole ausgibt. Diese kann als Oberklasse verwendet und polymorph erweitert werden, wenn nicht alle Methoden benötigt werden. Eine Implementierung für die Swing-Oberfläche liegt im Paket `fromClass::implementations::swing::controller`.

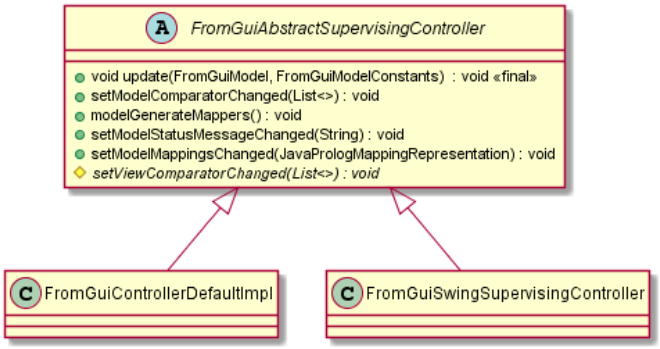

Abbildung 5.6: Die abstrakte Klasse `FromGuiAbstractSupervisingController` (in Auszügen) und implementierende Subklassen

Die View. Die in Swing implementierte View ist eng an den Controller gebunden. In Abbildung 4.7 auf Seite 35 ist eine Vorschau zu sehen. Die einzelnen Bereiche der Oberfläche haben folgendes Aufgaben:

Datei-Explorer: Der Datei-Explorer zeigt den Arbeitsordner an. Es werden alle `.java`-Dateien und rekursiv die Unterordner aufgelistet. Außerdem wird ggf. angezeigt, ob eine Klasse annotiert ist (vgl. Screenshot). Die Implementierung verwendet die Swing-Klasse `JTree`.

Mapperbereich: Der Mapperbereich zeigt eine Vorschau der Mapper. Jeder Mapper wird in einem eigenen Panel (einer Instanz von `AbstractMapperPanel`) gespeichert und ermöglicht Änderungen, d. h. das Vertauschen von Attributen mittels *Drag and Drop* sowie deren Umbenennung. Änderungen werden über den Controller an das Model weitergegeben. Ebenso wird hier das Generieren der Mapper-Klassen angestoßen.

5.6 Gesamtübersicht der Komponenten

Das Aktivitätsdiagramm in Abbildung 5.7 zeigt, wie die Komponenten miteinander interagieren.

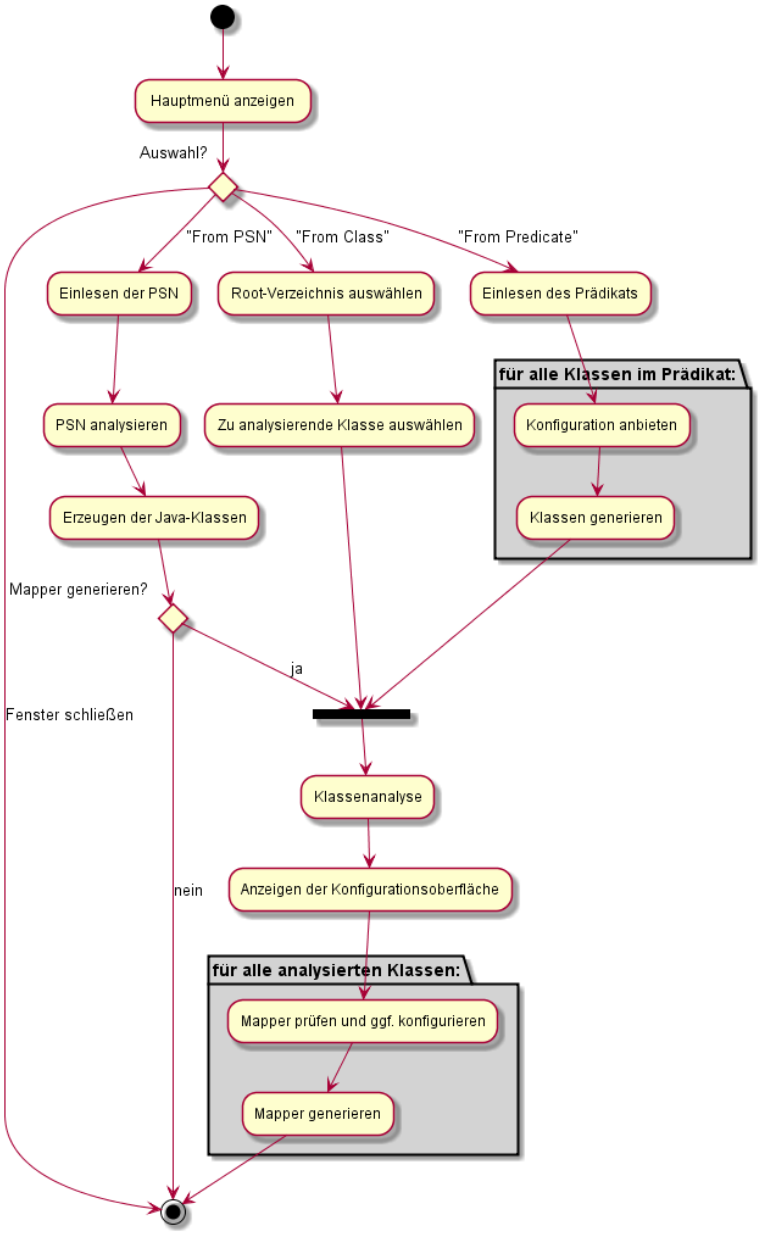

Abbildung 5.7: Aktivitätsdiagramm der *JPMapperGUI 2.0*

6 Integration CAPJAs in die Eclipse-Entwicklungsumgebung

Dieses Kapitel beschreibt die unter Java-Programmierern häufig verwendete Entwicklungsumgebung *Eclipse*. Anschließend wird beschrieben, wie Eclipse durch Plugins erweitert werden kann und schließlich wird CAPJA als Plugin vorgestellt.

6.1 Eclipse als vielseitiges Programmierwerkzeug

Eclipse ist eine quelloffene Entwicklungsumgebung, die 2001 von IBM erstellt wurde. Im Jahr 2004 wurde die ECLIPSE FOUNDATION gegründet, um die Open-Source-Eigenschaft zu etablieren [The]. Eclipse unterstützt dank einer flexiblen Plug-In-Struktur zahlreiche weitere Programmiersprachen neben Java.

Auch für diese Masterarbeit wurde Eclipse zur Entwicklung verwendet, das momentan in der Version *Neon* (Versionsnummer 4) vorliegt. Die Tests wurden daher auch mit dieser Eclipse-Version auf Windows 7 (64 Bit) durchgeführt.

Das Plugin soll hauptsächlich zwei Funktionen erfüllen:

- Die Komponenten sollen über einen Eintrag in einem eigenen Menü erreichbar sein.
- Der erweiterte Build-Prozess, der bereits in Abschnitt 2.3.3 auf Seite 15 beschrieben wurde, soll durch einen Menüeintrag verfügbar gemacht werden.

6.2 Entwicklung eines Eclipse-Plugins

6.2.1 Ein neues Plugin-Projekt erstellen

Im Folgenden wird schrittweise beschrieben, wie ein Plugin mit Eclipse gebaut werden kann, wobei Abweichungen unter verschiedenen Systemen und Versionen möglich sind. Diese Anleitung ist unter Windows 7, 64 Bit, mit *Eclipse Java EE IDE for Web Developers, Version: Neon.2 Release (4.6.2)* entstanden.

Zunächst muss sichergestellt sein, dass Eclipse PDE installiert ist. Dazu klickt man im Eclipse-Fenster auf *Help* → *Eclipse Marketplace*. Man sucht nun nach

dem Begriff `rcp` (!) und prüft, ob die *Plug-in Development Environment (PDE)* installiert ist (links müsste *Installed* angezeigt werden).

Man legt nun ein neues *Plug-In Project* über *File* → *New* → *Other* an. Die Voreinstellungen im Assistenten können übernommen werden. In einem Zwischenschritt wird die Auswahl eines *Templates* angeboten. Diese Möglichkeit sollte man nutzen, um die Funktionen einer Plugin-Applikation kennenzulernen. Im Beispiel wird der Eintrag `RCP 3.x application with a view` ausgewählt (vgl. Abbildung 6.1).

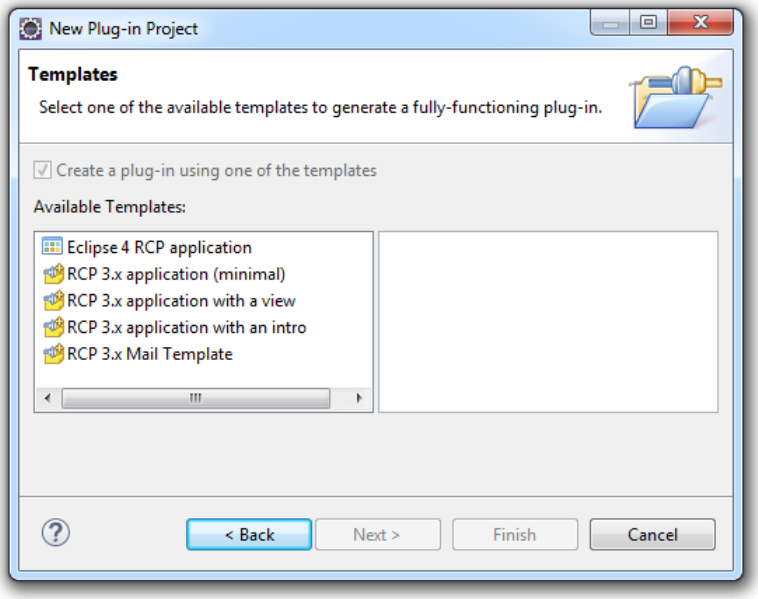

Abbildung 6.1: Eclipse-Assistent für Plug-In-Projekte

Im Anschluss öffnet sich ein Fenster, welches in Abbildung 6.2 auf der nächsten Seite zu sehen ist. Dieses dient der bequemen Konfiguration des Plugins, welche sonst über verschiedene Dateien in unterschiedlichen Formaten (vgl. Namen der Register in der Abbildung) erfolgen müsste.

6.2.2 Analyse und Modifikation der Eclipse-Oberfläche

Der ModelSpy. Ein Werkzeug, das für die Entwicklung von Plugins sehr wichtig ist, ist der ModelSpy. Bei korrekt installiertem *Eclipse PDE* kann dieser über die Tastenkombination *[Alt]+[Shift]+[F9]* aufgerufen werden. Sollte dies nicht funktionieren, kann es zusätzlich erforderlich sein, den *ModelSpy* für die Zielplattform zu installieren (siehe [Vog16]). Mittels des ModelSpy kann man die Oberfläche von Eclipse analysieren und auch modifizieren [Vog16]. Hierzu wählt man am

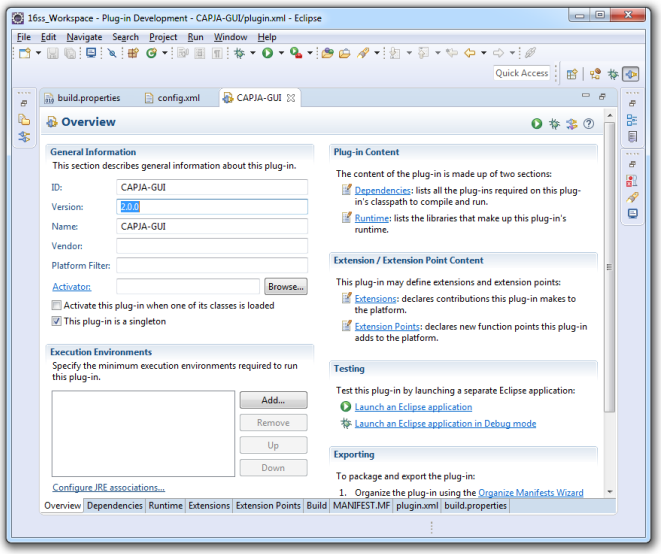

Abbildung 6.2: Konfigurationsfenster eines Eclipse-Plugins (am Beispiel des Capja-Plugins)

besten die Ansicht *List*. Die wichtigen Spalten in dort angezeigten Tabelle sind `elementId` und `label`. Man sucht in der letzten Spalte nach dem Namen desjenigen Menüs, das man verändern möchte. Im Beispiel wird das Menü `&Run` mit `elementId = org.eclipse.ui.run` ausgewählt. Änderungen im ModelSpy werden sofort in Eclipse sichtbar, daher ist es erforderlich, bei der Arbeit mit dem *ModelSpy* Vorsicht walten zu lassen. Um die Funktion besser kennenzulernen und um sicherzustellen, dass der richtige Eintrag ausgewählt wurde, kann man das Menü in `RunMenu` umbenennen. Abbildung 6.3 zeigt das Ergebnis.

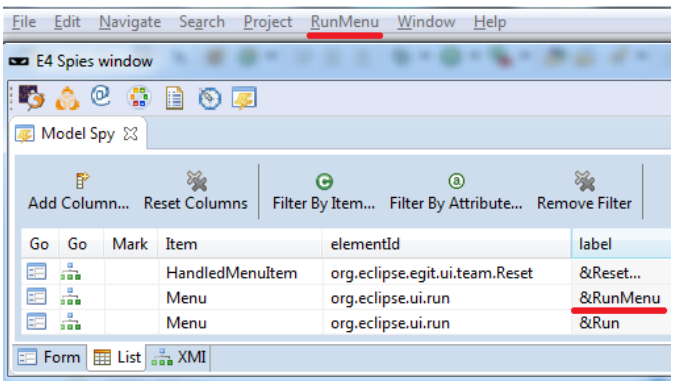

Abbildung 6.3: Bearbeitung des *Run*-Menüs mittels des *ModelSpy*

Oberflächenmodifikation mittels der `plugin.xml`. In der Datei `plugin.xml` wird festgelegt, welche Erweiterungen das Plugin bietet und wie diese implementiert sind [Fou10]. Mittels Definition einer Extension kann festgelegt werden, welche Komponente von Eclipse bearbeitet werden soll. Es sind z. B. folgende Werte möglich [Foua, einschl. Unterseiten]:

- `org.eclipse.ui.commands` definiert Befehle, die Eclipse zur Verfügung gestellt werden.
- `org.eclipse.ui.menus` definieret neue Menüeinträge.
- `org.eclipse.ui.handlers` beschreibt das Verhalten eines Kommandos.
- `org.eclipse.ui.bindings` legt Tastenkombinationen an.

Die Funktionsweise wird am ehesten anhand eines konkreten Beispiels deutlich; daher wird dieser Mechanismus im folgenden Abschnitt 6.3 besprochen.

Problembehebung. Treten Fehler auf, so könnten die folgenden Schritte Abhilfe schaffen:

- Möglicherweise sind nicht alle abhängigen Klassen in der `plugin.xml` aufgelistet (Reiter *Runtime*, Abschnitt *Classpath*).
- Wird ein Java-Projekt nachträglich in ein Plug-In-Projekt konvertiert (im Kontextmenü *Configure* → *Convert to Plug-in Projects* wählen), so kann es bei Fehlermeldungen helfen, den in Listing 6.1 gezeigten Code manuell in die Datei `plugin.xml` einzufügen [Lov09].
- Die Reorganisation des Plugins (Rechtsklick auf Projekt → *Plug-in Tools* → *Organize Manifests...*) kann ebenfalls helfen.

```
1 <requires>
2     <import plugin="org.eclipse.osgi"/>
3 </requires>
```

Listing 6.1: Fehlerbehebung bei Konvertierung in ein Plug-In-Projekt [Lov09]

6.3 CAPJA als Plugin

Die Datei `plugin.xml` enthält – wie im vorherigen Abschnitt bereits beschrieben – die Informationen darüber, wie das Plugin in Eclipse integriert wird. Ein Auszug dieser Datei für CAPJA ist in Listing 6.2 auf der nächsten Seite zu sehen und wird nachfolgend beschrieben.

```
1  <plugin>
2     <extension point="org.eclipse.ui.commands">
3        <command
4              name="CAPJA Main Menu"
5              categoryId="de.uni.wue.capja.gui.eclipse.
                 ↪ category"
6              id="de.uni.wue.capja.gui.eclipse.handlers.
                 ↪ MainMenu">
7        </command></extension>
8     <extension point="org.eclipse.ui.handlers">
9        <handler
10             commandId="de.uni.wue.capja.gui.eclipse.
                 ↪ handlers.MainMenu"
11             class="de.uni.wue.capja.gui.eclipse.handlers
                 ↪ .MainMenuHandler">
12       </handler></extension>
13    <extension
14          point="org.eclipse.ui.menus">
15       <menuContribution
16             locationURI="menu:org.eclipse.ui.run">
17             <menu
18                label="CAPJA"
19                mnemonic="C"
20                id="de.uni.wue.capja.gui.eclipse.menus">
21                <command
22                   commandId="de.uni.wue.capja.gui.
                        ↪ eclipse.handlers.MainMenu"
23                   label="Run CAPJA Main Menu"
24                   mnemonic="M"
25                   id="de.uni.wue.capja.gui.eclipse.
                        ↪ menuEntryGui">
26 </command>(...)</plugin>
```

Listing 6.2: Die Datei `plugin.xml` für CAPJA (in Auszügen)

Z. 3–7: Legt ein neues Kommando mit einer eindeutigen ID an, über die es referenziert werden kann. In diesem Fall soll der Handler für das `MainMenu` aufgerufen werden.

Z. 9–12: Definiert die zum Handler gehörende Java-Klasse.

Z. 15–26: Definiert ein Menü namens *Capja* und den untergeordneten Eintrag `Run CAPJA Main Menu`.

Das Ergebnis dieser Konfiguration ist das in Abbildung 6.4 gezeigte Menü. Um das Plugin zu starten, klickt man mit der rechten Maustaste auf das CAPJA-Projekt und wählt *Run as/Eclipse Application*.

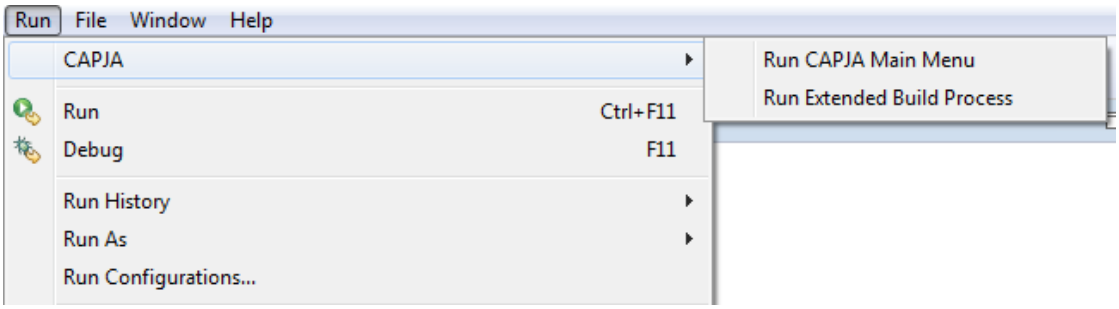

Abbildung 6.4: In Eclipse integriertes CAPJA-Menü

7 Technische Validierung

Dieses Kapitel beschreibt, wie die entwickelte Software getestet wurde. Hierbei wurde die Funktionalität geprüft. Die *Usability* wurde nicht explizit getestet; jedoch ist eine Vereinfachung bereits durch den Einsatz einer GUI gegenüber einer reinen Kommandozeilen-Applikation gegeben. Für diese Tests wurde die in Abschnitt 5.1 auf Seite 37 beschriebene Einstellung aktiviert. Zwischen den einzelnen Tests werden die jeweils generierten Klassen gelöscht. Sowohl die Tests als auch ihre Ergebnisse sind als ZIP-Dateien auf der beiliegenden CD im Ordner `Tests` zu finden.

7.1 Tests der Generierung aus einer *PSN*

1. Der Code in Listing 7.1 testet das Anlegen einer Bibliotheksklasse mit zwei `@JPMapping`-Annotationen. Die Klasse `Library` soll im Paket `generated.de.mircol.library` gespeichert werden. Zusätzlich wird die Klasse `People` erzeugt.

```
1  predicate(library/3:'de.mircol.library.library',
       ↪ compound, [
2      argument(name:'VollerName', atom),
3      argument(year_of_opening, atom),
4      argument(boss_list, list, people)   ]).
5  predicate(library/2:'de.mircol.library.library',
       ↪ compound, [
6      argument(name:'VollerName', atom),
7      argument(year_of_opening, atom)     ]).
8  predicate(people/2, compound, [
9      argument(first_name, atom),
10     argument('Last_name', atom)         ]).
```

Listing 7.1: Test der PSN (i)

Erwartetes Ergebnis: Es sollten zwei Klassen `Library` und `People` und ihre Mapper erzeugt werden.

Ergebnis: Es erscheint eine Erfolgsmeldung und im Eclipse-Projektbaum werden die korrekten Pakete und Klassen angezeigt. Die Mapper stimmen mit den Annotationen überein (vgl. Abb. 7.1) und sind syntaktisch korrekt. Test bestanden

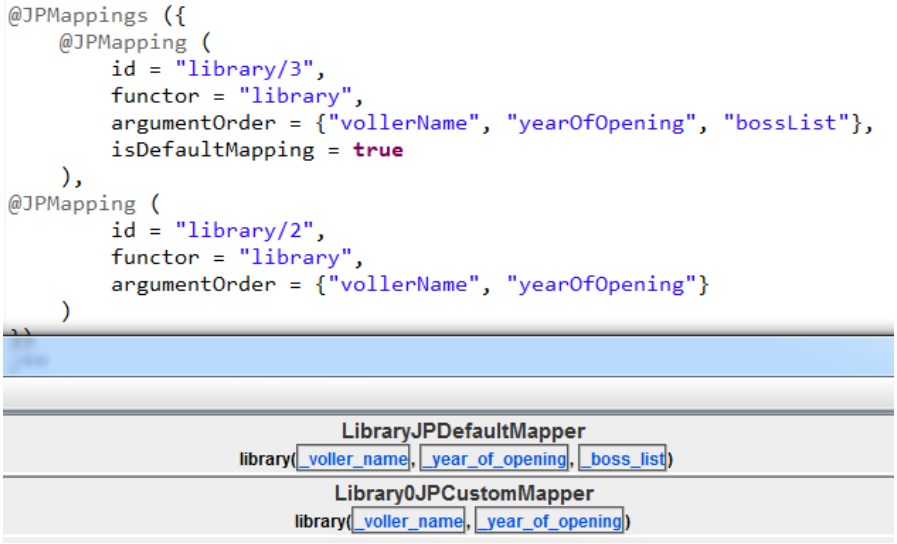

Abbildung 7.1: Ergebnis zu Test 7.1.1 (Auszug)

2. Der Code in Listing 7.2 soll erneut zwei Klassen anlegen, die `Dictionary` und `Language` heißen. In der `Dictionary`-Klasse gibt es einen Referenzdatentyp auf `Language`, letztere besitzt eine Integer- und eine String-Variable.

```
1  predicate(dictionary/1, compound, [
2      argument(language/2:language_iso, compound)  ]).
3  predicate(language/2, compound, [
4      argument(id, integer),
5      argument(iso_name:iso, compound)             ]).
6  predicate(language/3, compound, [
7      argument(id, integer),
8      argument(iso_name:iso, compound),
9      argument(english_name, atom)                  ]).
10 predicate(iso_class/1:iso, compound, [
11     argument(id, atom)                            ]).
```

Listing 7.2: Test der PSN (ii)

Erwartetes Ergebnis: Die Variablen in den generierten Klassen müssen korrekt referenziert werden und den richtigen Datentyp erhalten. Die Klasse `Language` muss zwei `@JPMapping`-Annotationen erhalten. Auch hier müssen Mapper angelegt werden.

Ergebnis: Die Datentypen sind korrekt, ebenso die erzeugten Annotationen. Die Mapper stimmen mit den Annotationen überein und sind syntaktisch korrekt. Die Namen wurden in lowerCamelCase umgewandelt. Test bestanden

3. Der Code in Listing 7.3 sollte nicht funktionieren, da das eine Prädikat keine Projektion des anderen ist.

```
predicate(language/2, compound, [
    argument(id, integer),
    argument(german_name, atom)                    ]).
predicate(language/3, compound, [
    argument(id, integer),
    argument(iso_name, atom),
    argument(english_name, atom)                   ]).
```

Listing 7.3: Test der PSN (iii)

Erwartetes Ergebnis: Es erscheint eine Fehlermeldung.

Ergebnis: Die Eingabe wird regelkonform mit einer Fehlermeldung zurückgewiesen. Test bestanden

7.2 Tests der Generierung aus einem Prädikat

1. Aus dem Prädikat in Listing 7.4 soll eine Klasse `Dictionary` mit drei Variablen erzeugt werden. Die Variablen sollen als Integer (`id`) bzw. Strings übernommen werden. Die `id` muss im Konstruktor gesetzt werden.

```
dictionary(Id, Iso_name, English_name).
```

Listing 7.4: Test der Generierung aus einem Prädikat (i)

Erwartetes Ergebnis: Es soll ein Fenster erscheinen, das die Konfiguration der Variablen erlaubt. Nach der Konfiguration soll der Mapper-Konfigurator erscheinen.

Ergebnis: Die in Abbildung 7.2 auf der nächsten Seite gezeigte Konfiguration erzeugt die korrekte Klasse. Test bestanden

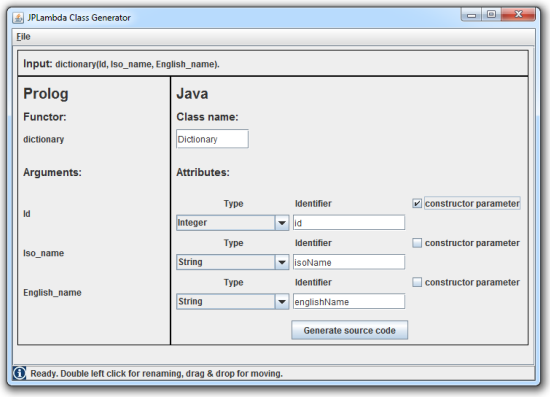

Abbildung 7.2: Test der Generierung aus einem Prädikat (i)

2. Das Prädikat in Listing 7.5 beinhaltet zwei verschachtelte Klassen. Die `id` muss zwingend im Konstruktor gesetzt werden, da sie mit dem Modifikator `+` versehen ist. In der graphischen Oberfläche wird der Datentyp der `id` auf `int` eingestellt und die Klasse `Dictionary` in `Wörterbuch` umbenannt (vgl. Abbildung 7.3).

```
1 dictionary(+Id, Iso_name, pair(English_name,
    ↪ German_name)).
```

Listing 7.5: Test der Generierung aus einem Prädikat (ii)

Erwartetes Ergebnis: Die Klasse sollte korrekt umbenannt werden, `id` sollte als Integer erzeugt werden und es darf nicht möglich sein, `id` als Konstruktor-Parameter zu entfernen.

Ergebnis: In der Abbildung ist zu erkennen, dass die Checkbox für den Konstruktor-Parameter ausgegraut und eingeschaltet ist. Die Klassen werden mit korrektem Namen und korrekten Variablen ausgegeben.

Test bestanden

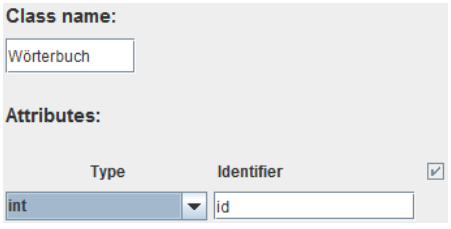

Abbildung 7.3: Test der Generierung aus einem Prädikat (ii)

7.3 Tests der Mapper-Generierung aus einer Klasse

1. Für die in Listing 7.6 gezeigte Klasse `Address` sollen Mapper erzeugt werden. Es ist darauf zu achten, dass die Reihenfolge der Attribute übereinstimmt und `tel` als Referenzdatentyp erkannt wird. Für die referenzierten Datentypen müssen ebenfalls Mapper erzeugt werden, sofern kein `Enum<?>` vorliegt; in diesem Fall muss folglich `TelephoneNumber` (Listing 7.7) zusätzlich gemappt werden. In Abb. 7.4 ist die eingestellte Konfiguration zu sehen.

```
1 public class Address {
2     private String city;
3     private Country country;
4     private String street;
5     private int zipCode;
6     private TelephoneNumber tel;
7     //getter und setter
8 }
```

Listing 7.6: Die Testklasse `Address`

```
1 public class TelephoneNumber {
2     public int prefix, mainNumber, directAccess;
3 }
```

Listing 7.7: Die Testklasse `TelephoneNumber`

Abbildung 7.4: Testkonfiguration der Mapper für die Klasse `Address`

Erwartetes Ergebnis: Es werden zwei Mapper für `Address` mit der vorgegebenen Reihenfolge erzeugt.

Ergebnis: Nach dem Anlegen der Mapper für `Address` liegen noch Syntaxfehler vor. Sobald der Mapper für `TelephoneNumber` erzeugt wurde, verschwinden diese. Der Mapper für `Country` ist nicht erforderlich.

Test bestanden

2. Für eine von Hand annotierte Testklasse `PersonAnnotated` sollen die Mapper erzeugt werden. Die Annotationen werden in Listing 7.8 gezeigt.

   Erwartetes Ergebnis: Die Mapper sollen erkannt und richtig dargestellt werden. Da diese aus einer Annotation generiert wurden, dürfen sie nicht editierbar sein.

   Ergebnis: Die Vorschau entspricht der Annotation (vgl. Abbildung 7.5). Die Mapper besitzen die gleichen Attribute in der gleichen Reihenfolge.
   Test bestanden

```
@JPMappings ({
    @JPMapping(
        id = "person/5",
        functor = "person_4_class",
        argumentOrder = { "age", "givenName", "
            ↪ middleName", "familyName", "income" },
        isDefaultMapping = true
    ),
    @JPMapping(
        id = "person/4",
        functor = "person_3_class",
        argumentOrder = { "income", "givenName", "
            ↪ familyName", "middleName" }
    )
})
```

Listing 7.8: Die Annotationen der Testklasse `PersonAnnotated`

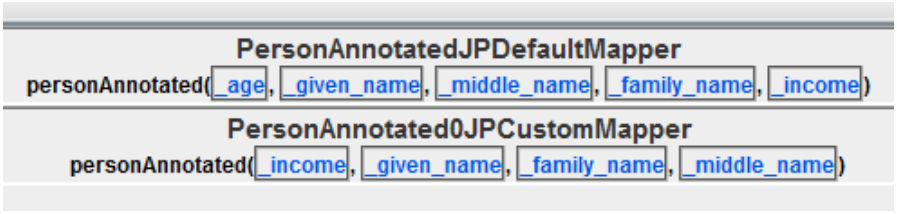

Abbildung 7.5: Ergebnis der Analyse der `PersonAnnotated`-Annotationen

8 Zusammenfassung und Ausblick

Überblick. In dieser Masterarbeit wurde eine benutzerfreundliche und alltagsnahe Möglichkeit geschaffen, CAPJA zu verwenden. Das entwickelte Programm unterstützt Programmierer in realitätsnahen Anwendungen und ist nun als Werkzeug für den direkten Einsatz während der Entwicklung einer Anwendung verfügbar.

Kapitel 2 auf Seite 5 beschäftigte sich mit der Multiparadigmen-Programmierung mit Prolog und Java. Hierbei wurde auf die Stärken und Schwächen beider Sprachen eingegangen und gefolgert, weshalb Multiparadigmen-Programmierung vorteilhaft sein kann. In Kapitel 3 auf Seite 19 wurden wichtige Begriffe wie ANTLR und Entwurfsmuster im Detail erläutert, welche zum Verständnis der nachfolgenden, detailierten Erläuterungen notwendig waren. Das Kapitel 4 auf Seite 27 stellte die Module der Software, d. h. Klassen- und Mappergeneratoren, im Einzelnen vor, um einen kompletten Überblick über Struktur und Aufbau der vollständigen Anwendung zu bieten, während sich Kapitel 5 auf Seite 37 mit Details der Implementierung beschäftigte und auf Design-Entscheidungen einging. Weiterhin wurde die Integration der *JPMapperGUI 2.0* in Eclipse in Kapitel 6 auf Seite 55 übersichtlich behandelt, um so einen schnellen und leichten Einstieg zu ermöglichen, was für den reibungslosen und alltagsnahen Einsatz der entwickelten Anwendung wichtig ist. Schließlich ging es in Kapitel 7 auf Seite 61 um die Validierung. Hierbei wurde die technische Korrektheit der Anwendung wie zuvor beschrieben gezeigt. Komplikationen zeigten sich lediglich an wenigen, unkritischen Stellen.

Die Oberfläche wurde in Swing entwickelt, ist jedoch dank des Aufbaus als *Model-View-Controller* flexibel erweiterbar. Dank des Parsergenerators ANTLR konnte Prologcode ausschließlich mit Java selbst in Klassen konvertiert werden. Auf der Zielseite unterstützen Annotationen und viele Werkzeugklassen den Prozess. Die Nutzerfreundlichkeit wurde dadurch gesteigert, dass eine Plugin-Implementierung für Eclipse vorgestellt wurde.

Ansätze zur Verbesserung der Software. Folgende Maßnahmen, die im Rahmen der vorliegenen Masterarbeit nicht mehr vollzogen werden konnten, können die Qualität dieser Software verbessern:

- Die Erkennung von Prädikaten sollte erweitert werden. Dazu ist die GUI mit weiteren Anfragen zu testen, um Fehler in der ANTLR-Grammtik zu tilgen und ggf. die von ihr erkannte Sprache zu erweitern.
- Der Integrationsmechanismus für Prädikate könnte so eingestellt werden, dass er Prologdateien automatisiert auf *Predicate-Signature Notation*s prüft.
- Es sollte eine exakte Laufzeitanalyse, die Geschwindigkeitsprobleme der Software aufzeigen könnte, durchgeführt werden.
- Die automatische Kompilierung generierter Klassen in Eclipse funktioniert nur mittels Zusatzeinstellungen; es ist zu überlegen, inwiefern ein Eclipse-Plugin hierbei unterstützen kann oder ob eine externe Softwarelösung unterstützen kann.
- Die Annotation `@JPMapping` könnte als multiple Annotation implementiert werden, um den Code kompakter zu machen und zu vereinheitlichen.
- Die Integration der JPML steht noch aus.
- Die Usability könnte durch Tests optimiert werden.
- Eine ähnliche Software könnte auch erstellt werden, um Java und Python miteinander zu verbinden.

Literaturverzeichnis

[ACZ05] Amandi, A. ; Campo, M. ; Zunino, A.: JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming. In: *Computer Languages, Systems & Structures 31.1* (2005), S. 17–33

[Bal99] Balzert, Heide: *Lehrbuch der Objektmodellierung.* 1. Auflage. Spektrum Akademischer Verlag, 1999

[BEG16] Braun, Robert ; Esswein, Werner ; Greiffenberg, Steffen: *Einführung in die Programmierung.* Springer-Verlag, 2016

[BTI06] Banbara, M. ; Tamura, N. ; Inoue, K.: Prolog Cafe: A Prolog to Java Translator. In: *Proc. Intl. Conference on Applications of Knowledge Management, INAP 2005, Lecture Notes in Artificial Intelligence, Vol. 4369* (2006), S. 1–11

[CMM13] Castro, S. ; Mens, K. ; Moura, P.: JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog. In: *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)* (2013), S. 26–42

[ES10] Eilebrecht, Karl ; Starke, Gernot: *Patterns kompakt.* 3. Auflage. Spektrum Akademischer Verlag, 2010

[Ezr07] Ezra, Aviad: *Twisting the MVC Triad - Model View Presenter (MVP) Design Pattern.* Online. Abgerufen am 09.01.2017. `http://aviadezra.blogspot.de/2007/07/twisting-mvp-triad-say-hello-to-mvpc.html`. Version: 7 2007

[Fis] Fischer, Charles: *lecture 34.4 – How Prolog Solves Queries.* Online. Abgerufen am 20.12.2016. `http://pages.cs.wisc.edu/~fischer/cs538.s08/lectures/Lecture34.4up.pdf`

[Foua] Foundation, The E.: *Eclipse documentation – Current Release: Basic workbench extension points using commands.* Online. Abgerufen am 18.01.2017. `http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_cmd.htm&cp=2_0_4_1`

[Foub] FOUNDATION, The E.: *Eclipse documentation - Previous Release: Plug-in Build*. Online. Abgerufen am 18.01.2017. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Ftools%2Feditors%2Fmanifest_editor%2Fbuild.htm

[Fou10] FOUNDATION, The E.: *FAQ What is the plug-in manifest file (plugin.xml)?* Online. Abgerufen am 18.01.2017. https://wiki.eclipse.org/FAQ_What_is_the_plug-in_manifest_file_(plugin.xml)%3F. Version: 2010

[Fow06] FOWLER, Martin: *Passive View*. Online. Abgerufen am 09.01.2017. https://martinfowler.com/eaaDev/PassiveScreen.html. Version: 7 2006

[Hof] HOFFMANN, J.: *Verfahren des Rapid Prototyping – Möglichkeiten und Grenzen*. Online. Abgerufen am 31.12.2016. https://tu-dresden.de/ing/maschinenwesen/if/ff/ressourcen/dateien/pazat/forschung/for_ber_pdf_html/lit_98_html/hoff-983.pdf

[int16] *Java-Prolog bridge*. Online. Abgerufen am 20.12.2016. http://interprolog.com/wiki/index.php?title=Java-Prolog_bridge. Version: 2016

[Jan05] JANA, Debasish: *Java and Object-Oriented Programming Paradigm*. 2005 https://books.google.de/books?id=wxTajapmFyMC&pg=PA6&dq=procedural+programming+paradigm&hl=de&ei=esXOTpCGBIK3hAeMqPHBDQ&sa=X&oi=book_result&ct=result&redir_esc=y#v=onepage&q=procedural%20programming%20paradigm&f=false

[Kie14] KIERS, Bart: *Stack Overflow: Antlr Extraneous Input*. Online. Abgerufen am 12.01.2017. http://stackoverflow.com/a/23639737. Version: 2014

[KK15] KOUNEV, Samuel ; KISTOWSKI, Jóakim: *Grundlagen der Programmierung – VL10: Objektorientierung*. Online (Vorlesungsskript). Abgerufen am 29.10.2015, 2015

[Lov09] LOVELL, Douglas: *Eclipse Community Forums: org.osgi.framework.BundleActivator*. Online. Abgerufen am 15.01.2017. https://www.eclipse.org/forums/index.php/t/66297/. Version: 2009

[Mü03a] Mühlhäuser, Max: *Grundzüge der Informatik – Teil 2: Objektorientierte Programmierung – 2.1 Objekte und ihre Beziehungen.* Online. Abgerufen am 09.12.2016. `http://atlas.tk.informatik.tu-darmstadt.de/LectureNotes/ws0304/GdI1/GdI-T1.5.pdf`. Version: 2003

[Mü03b] Mühlhäuser, Max: *Grundzüge der Informatik 1 – Teil 1: Einführung – 1.5 Programmierparadigmen.* Online. Abgerufen am 09.12.2016. `http://atlas.tk.informatik.tu-darmstadt.de/LectureNotes/ws0304/GdI1/GdI-T1.5.pdf`. Version: 2003

[OFS14] Ostermayer, Ludwig ; Flederer, Frank ; Seipel, Dietmar: PPI – A Portable Prolog Interface for Java. In: *Proc. 10th Workshop on Knowledge Engineering and Software Engineering (KESE)* (2014)

[Ora] Oracle Corporation: *What's New in JDK 8.* Online. Abgerufen am 09.12.2016. `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`

[Ora16] Oracle Corporation: Java 8 API. (2016). `http://docs.oracle.com/javase/8/docs/api/`

[Ost15] Ostermayer, Ludwig: Seamless Cooperation of Java and Prolog for Rule-Based Software Development. In: *Proc. Doctorial Consortium@ 9th International Web Rule Symposium (RuleML)* (2015)

[Ost17] Ostermayer, Ludwig: *Integration of Prolog and Java with the Connector Architecture CAPJA*, Universität Würzburg (erscheint 2017 – Vorabversion vom 28.01.2017 zitiert), Dissertation, 2017

[Par13] Parr, Terence: *The Definitive ANTLR 4 Reference.* O'Reilly UK Ltd., 2013

[Pog13] Pogonets, Anton: *Stack Overflow: Get type of a generic parameter in Java with reflection.* Online. Abgerufen am 16.01.2017. `http://stackoverflow.com/a/20774106`. Version: 12 2013

[Sei15] Seipel, Dietmar: *Datalog und Prolog. Skript zur Vorlesung* Advanced Databases*, Universität Würzburg, SS 2016.* Online. Abgerufen am 23.06.2016, 2015

[sew16] *Prolog – Begriffe.* Online. Abgerufen am 19.12.2016. `https://sewiki.iai.uni-bonn.de/service/knowhow/prolog/start`. Version: 4 2016

[Sun97] Sun Microsystems: *JavaBeans*. Online. Abgerufen am 08.01.2017. http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html. Version: 8 1997

[The] The Eclipse Foundation: *About the Eclipse Foundation*. http://www.eclipse.org/org/#about

[TI15] TIOBE-Index: *Die populärsten Sprachen*. Online. Abgerufen am 09.12.2016. https://www.heise.de/ct/ausgabe/2015-18-Die-passende-Programmiersprache-finden-2767703.html. Version: 2015

[Ull10a] Ullenboom, Christian: *Java ist auch eine Insel*. Kap. *Annotieren von Annotationstypen*. Rheinwerk Computing, 2010. – Online-Fassung. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_25_006.htm#mj6dc62903d2743fc2d8a27fd497950dcd. Abgerufen am 07.01.2017

[Ull10b] Ullenboom, Christian: *Java ist auch eine Insel*. Kap. *Reflection und Annotationen*. Rheinwerk Computing, 2010. – Online-Fassung. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_25_001.htm#mja2e217bf5724cc65eba184a2af8b79cf. Abgerufen am 08.01.2017

[Ull12] Ullenboom, Christian: *Java – Mehr als eine Insel*. Rheinwerk Computing, 2012. – Online-Fassung. http://openbook.rheinwerk-verlag.de/java7/1507_21_001.html#dodtp283ed9de-cbc2-463f-9fad-4f7eef376a8d. Abgerufen am 07.01.2017

[Vog16] Vogel, Lars: *Eclipse model spy- Tutorials*. Online. Abgerufen am 15.01.2017. http://www.vogella.com/tutorials/Eclipse4LiveModelEditor/article.html. Version: 7s 2016

[wik] *Wikipedia: Parsergenerator*. Online. Abgerufen am 08.01.2017. https://de.wikipedia.org/wiki/Parsergenerator

[Wik10] WikiSolved: *File:Observer.svg*. Online. Abgerufen am 07.01.2017. https://commons.wikimedia.org/wiki/File:Observer.svg. Version: 4 2010

[wik16a] *Wikipedia: Domänenspezifische Sprache*. Online. Abgerufen am 9.12.2016. https://de.wikipedia.org/wiki/Dom%C3%A4nenspezifische_Sprache. Version: 11 2016

[wik16b] *Wikipedia: Prolog*. Online. Abgerufen am 20.12.2016. `https://de.wikipedia.org/wiki/Prolog_(Programmiersprache)`. Version: 2016

[Zel01] Zeller, Andreas: *Extreme Programming*. Online. Abgerufen am 31.12.2016. `https://www.st.cs.uni-saarland.de/edu/lehrer/xp.pdf`. Version: 11 2001

Index

F

I

J

K

L

Abbildungsverzeichnis

Listings

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die diesen Quellen und Hilfsmitteln wörtlich oder sinngemäß entnommenen Ausführungen als solche kenntlich gemacht habe.
Die Arbeit habe ich bisher oder gleichzeitig keiner anderen Prüfungsbehörde vorgelegt.

Würzburg, 31. Januar 2017

.....
Mirco Lukas