Implementierung eines Prolog-Adapters für Java

Praktikum von

Johannes Werner

Institut für Software Design und Qualität
Betreuender Mitarbeiter: M.Sc. Stephan Seifermann

1 Einführung

Programmiersprachen haben unterschiedliche Stärken und Schwächen. Das liegt daran, dass Sprachen meist für ein bestimmtes Problem oder eine bestimmte Domäne an Problemen entwickelt werden. Schaut man sich Java und Prolog an, wird dies besonders deutlich. Java ist eine objekt-orientierte Programmiersprache mit dem Ziel, möglichst zugänglich und gleichzeitig möglichst mächtig zu sein. Durch den imperativen Programmierstil ist diese Sprache außerdem für viele Menschen gut verständlich und kann für eine Vielzahl an Problemen eingesetzt werden. Prolog hingegen verfolgt einen ganz anderen Ansatz. Anstatt eine Sequenz an Befehlen zu programmieren, die angeben, *wie* ein Problem gelöst wird, definert man in Prolog lediglich *was* das Problem ist und der Prolog-Interpreter löst es dann. Das ist vorallem dann von Vorteil, wenn man vorrangig logische Probleme lösen will.

Um die Vorteile einer Sprache auch sprachübergreifend nutzen zu können, werden häufig Bibliotheken bereitgestellt, die als Schnittstelle zu anderen Sprachen dienen. Beispielsweise stellen einige Prolog-Interpreter Java-Bibliotheken bereit, die einen einfachen Zugang zu Prolog in Java-Code ermöglichen. Allerdings haben diese Bibliotheken keine einheitliche API, was es erschwert, verschiedene Implementierungen derselben Sprach in einem Programm zu nutzen. Dabei ist eine Auswahl verschiedener Implentierungen genau das, was in manchen Fällen erforderlich sein kann. Gründe dafür können sich per Einsatzszenario ändernde Anforderungen sein, wie beispielsweise Lizenzbedingungen oder Performanceunterschiede. Um dennoch mehrere verschiedene Implemntierungen verwenden zu können, kapselt man die unterschiedlichen APIs in einem Adapter und stellt nach außen hin eine einheitliche API zur Verfügung. Abbildung 1 zeigt, wie ein solcher Adapter shematisch aussieht. Schnittstellen zu einer Zielsprache werden entweder als eine native Bibliothek zur Verfügung gestellt, oder die Zielsprach ist direkt in

der Quellsprache implementiert. Der Adapter sellbst ist in der Quellsprache implementiert und kapselt die unterschiedlichen Schnittstellen-APIs. Dadurch kann das Hauptptogramm einheitlich auf die verschiedenen Implementierungen der Zielsprache zugreifen.

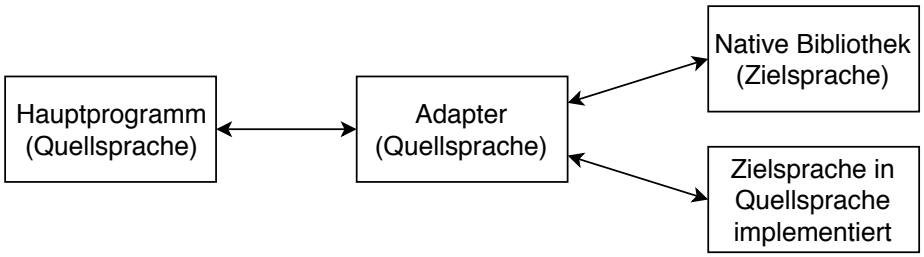

Abbildung 1: Ein Adapter ist in der Quellsprache implementiert und greift entweder auf eine native Bibliothek oder auf eine Quellsprach-Implementierung der Zielsprache zu.

1.1 Projektrahmen

Dieses Praktikum findet im Rahmen des Projektes Trust 4.0[2] statt. Trust 4.0 hat es sich als Ziel gesetzt, Datenschutz in einem Softwaresystem garantieren zu können. Dazu wird mittels eines Plugins in Eclipse ein Architekturmodell des zu prüfenden Softwaresystems erstellt und um Datenflussannotationen ergänzt. Diese Annotation beschreiben, welchen Weg personenbezogne Daten über verschiedene Software-Module hinweg nehmen und aufgrunddessen kann man analysieren, ob es an einer Stelle zu Datenlecks kommen kann. Das Eclipse-Plugin selbst ist in Java implementiert, jedoch hat man sich bei der Datenflussanalyse für Prolog entschieden. Wie bereist weiter oben erläutert, ist Prolog speziell für rein logischen Probleme entwickelt worden, weshalb es sich für die Analyse besser eignet als Java. Um es in Zukunft dem Nutzer zu ermöglichen, von dem Eclipse-Plugin aus die Analyse auf einem frei gewählten Prolog-Interpreter auszuführen, ist es das Ziel dieses Praktikums, einen Prolog-Adapter für Java zu entwickeln.

2 Anforderungen

Wie bereits in Kapitel 1.1 geschildert, soll der Adapater im Projekt Trust 4.0 eingesetzt werden. Daraus egeben sich einige Anforderungen, damit eine Integration in die bestehende Codebasis möglich ist.
Der Adapter soll in dem projekteignen Eclipse-Plugin eingesetzt werden. Dieses Plugin ist unter der Eclipse Plugin Licence (EPL)[1] lizenziert, was zu der Anfordung führt, dass Software, die in dem Plugin zum Einstaz kommt, auch unter EPL oder zu EPL kompatibel lizenziert sein muss. Diese Anforderung gilt ebenso für den Adapter und die durch ihn angebundene Interpreter.
SWI-Prolog[3] ist der quasi Standard-Interpreter in der Prologwelt. Kein anderer Interpreter ist in dem Maße verbreitet und wird auch dementsprchend aktiv weiterentwickelt, weshalb eine Anbindung daran unerlässlich ist. Außerdem muss mindestens ein Weiterer

der angebundenen Interpreter in Java implementiert sein. Der Grund dafür ist, dass das Plugin bei der Auslieferung möglichst wenige Abhängigkeiten auf bereits vorinstallierte Software haben soll. Ein in Java implementierter Interpreter kann in das Plugin-Packet miteingebunden werden, wodurch der Nutzer nicht auf einen vorinstallierten Interpreter auf seinem System angewiesen ist. Im Nachhineinen gibt es jedoch immernoch die Möglichkeit bei Bedarf auf SWI-Prolog o.ä. zu wechseln.
Wenn man Anfragen an einen Interpreter im interaktiven Modus im Terminal schickt, übergibt man die Anfrage selbst als einen String und als Antwort erhält man ebenfalls einen String. Dieses Verhalten ist gewünscht, wenn man in einem Terminal arbeitet, da man die Ergebnisse in der Regel nicht programmatisch weiterverarbeiten will. Sendet man Anfragen jedoch, wie in unserem Fall, aus Java-Code heraus, will man mit dem Ergebnis einer Anfrage weiter arbeiten. Da wäre es unvorteilhaft, würde man als Antwort vom Interpreter ausschließlich Strings erhalten, da man diese erst selbst in ein gewünschte Format parsen müsste. Solche Parser sind fehleranfällig und aufwändig zu implementieren, bringen aber gleichzeitig dem Projekt wenig Mehrwert. Deshalb sollten man bei der Verwendung des Adpaters bereits verwertebare Objekte als Antwort erhalten, wie beispielsweise *Integer*, *Float* oder selbst definierte Klassen.

3 Interpreterwahl

In Kapitel 2 wurde bereits SWI-Prolog als ein anzubindenden Interpreter fest vorgebeben, sowie die beiden Anforderungen, dass die Interpreter EPL-kompatibel lizenziert sein müssen und außerdem mindestens ein Interpreter in Java implementiert sein muss.
Insgesamt habe ich die Wahl, welche Interpreter angebunden werden sollen, nach folgenden Kriterien getroffen:

EPL-kompatibel lizenziert* Der Adapter wird Teil eines Eclipse-Plugins sein, welches unter EPL lizenziert ist. Daraus ergibt sich, dass ein verwendeter Interpreter mit EPL kompatibel lizensiert sein muss.
Java-Schnittstelle* Um einen Interpreter von Java-Code aus zu verwenden muss dieser bereits eine eigene Java-Schnittstelle mitbringen. Es wäre zwar möglich auch selber eine solche Schnittstelle zu entwickeln, allerdings liegt das außerhalb des Umfangs dieses Praktikums.
Paket im Maven-Repository Ein Paket, welches im Maven-Repository liegt, kann ohne großen Aufwand in den Build-Prozess eines Projektes eingebunden werden. Dies verringert den Wartungsaufwand und stabilisiert den Build-Prozess über Platformen und Geräte hinweg, da ein Paket direkt aus dem Repository geladen werden kann, anstatt dass nach lokalen Bibliotheken gesucht werden muss. Ein Interpreter, der seine Java-Schnittstelle als Packet im Maven-Repository bereitstellt, verursacht daher allgemein weniger Aufwand bei der Einbindung.
Java-Implementierung Mindestens ein Interpreter sollte in Java implementiert sein, sodass dieser mit dem Plugin zusammen ausgeliefert werden kann. Dadurch muss der Nutzer keinen eigenen Interpreter auf seinem System installieren.
Zeitpunkt des letzten Releases Ein Projekt, welches nicht gewartet wird, bringt ein

gewisses Risiko bei der Verwendung mit sich. Zum Einen werden logische Programmfeheler, die zu falschen Ergebnissen bei Berechnungen führen können, nicht mehr gefixt und zum Anderen könen auch ungepatchte Sicherheitslücken zu einem Problem für den Nutzer werden. Außerdem kann mit der Veröffentlichung neuer Versionen von Drittsoftware die Kompatibilität mit veralteten Projekten gebrochen werden. Daher ist es wichtig, dass ein Interpreter, zumindest zum jetzigen Zeitpunkt, noch aktiv weiter entwickelt wird.

3.1 SWI-Prolog

Dieser Interpreter ist Quasi-Standard in der Prolog-Welt

3.2 TuProlog

3.3 Projog

4 Prolog4J

5 Fazit

6 Aussicht

Literatur

[1] "Eclipse Plugin Licence". In: (). URL: https://www.eclipse.org/legal/epl-v10.html.

[2] Robert Heinrich Stephan Seifermann und Ralf Reussner. "Data-Driven Software Architecturefor Analyzing Confidentiality". In: (2019). URL: https://sdqweb.ipd.kit.edu/publications/pdfs/seifermann2019b.pdf.

[3] "SWI-Prolog". In: (). URL: https://www.swi-prolog.org/.