Modulare domänenspezifische Sprachen

Für ein neuartiges Überwachungssystems von Lieferketten entwickeln wir mit Groovy und Java eine modulare DSL-Engine, welche es ermöglicht, eine domänenspezifische Sprache zu implementieren, deren Syntax und Semantik jederzeit durch neue Module ergänzt werden kann. Durch dieses modulare Konzept lässt sich die Funktionalität eines Systems, wie auch die DSL, sehr einfach an die sich ständig verändernden Bedürfnisse der Domänenspezialisten anpassen.

Jürg Luthiger, Markus Knecht | juerg.luthiger@fhnw.ch

Für die Überwachung der Zulieferer und deren Produkte wird im Forschungsprojekt APPRIS (Advanced Procurement Performance and Risk Indicator System) ein neuartiges Überwachungssystem entwickelt. Die Überwachung geschieht auf der Basis einer frei definierbaren Regelmenge. Wird eine Regel verletzt, so generiert das System ein Warnsignal und macht die Benutzerin auf eine mögliche Störung in einer der überwachten Lieferketten aufmerksam, damit sie frühzeitig Gegenmassnahmen einleiten kann. Die bei der Überwachung verwendeten Regeln müssen Informationen aus unterschiedlichen Datenquellen nutzen, wie zum Beispiel ERP-Systeme, Web-Services verschiedener Anbieter oder unstrukturierte Inhalte aus dem Internet.

Problemstellung

Die Definition von Regelmengen ist eine domänenspezifische Aufgabe. Beispielsweise ist es das Team aus dem Einkauf, welches die Zusammenhänge in der Lieferkette am besten kennt. Deshalb braucht es für dieses Team eine adäquate Schnittstelle zum Monitoring-Tool, über welche die Teammitglieder einfach und effektiv die Regeln des Überwachungssystems festlegen können. Oft wird für eine solche Aufgabe eine formale Sprache eingesetzt, welche speziell für ein bestimmtes Problemfeld (die Domäne) entworfen und implementiert wird. Eine solche Sprache wird als *Domain-Specific Language* (DSL) bezeichnet. Beim Entwurf einer DSL wird man bemüht sein, einen hohen Grad an Problemspezifität zu erreichen. Dadurch ist die DSL durch Domänenspezialisten ohne besonderes Zusatzwissen einsetzbar.

Mit dem ständigen Einsatz eines Überwachungssystems wachsen auch die Ansprüche der Benutzerinnen und Benutzer daran. Daher muss es leicht möglich sein, die Regelmengen ständig den Bedürfnissen anzupassen und dabei auch neue Datenquellen zu erschliessen.

Die Anforderungen in APPRIS gehen über die üblichen Anforderungen an DSLs heraus. So müssen in unserem Fall die einzelnen Module des Überwachungssystems über die DSL kontrolliert und konfiguriert werden können und jedes Modul kann wieder neue Funktionen einführen, die wiederum über die DSL genutzt werden sollen. Um ein derartiges Verhalten zu ermöglichen, muss die DSL möglichst flexibel erweitert[1] werden können. Jedes Modul, das neue Funktionen einbringt, muss in der Lage sein, die DSL so zu erweitern, dass die neuen Funktionen auch über die DSL nutzbar werden. Die DSL ist somit modular, d.h. sie wird aus unterschiedlichen Modulen zusammengesetzt. Herkömmliche DSL-Ansätze müssen demnach um eine Modularisierungsmöglichkeit erweitert werden.

Beispiel einer DSL

In einer DSL kann zum Beispiel eine Regel zur Überwachung des Silberpreises konfiguriert werden (siehe Listing 1). Es wird eine Warnung ausgegeben, falls der Silberpreis über 1.15 Schweizer Franken steigt.

```
when price > 1.15 CHF then
produce warning
```

Listing 1: Beispiel einer einfachen Regel zur Überwachung eines Preises

Soll nun diese Regel erweitert werden, um den Silberpreis mit dem Lagerbestand eines Lieferanten zu kombinieren, so kann die Regel wie in Listing 2 ergänzt werden. Damit kann ein maximales Limit überwacht werden.

```
when price * stock > 10000 CHF
then produce warning
```

Listing 2: Beispiel einer erweiterten Regel zur Überwachung eines Lagerbestandes

Diese Erweiterung führt dazu, dass eine neue Datenquelle integriert werden muss, welche bei-

1 In APPRIS ist es nicht erforderlich, dass die Erweiterung zur Laufzeit erfolgen muss, im Gegensatz zu einem Vorgängerprojekt [Kne12].

spielsweise das ERP-System eines amerikanischen Silberlieferanten nutzt, um den aktuellen Silberlagerstand abzufragen. In Kombination mit dem Silberpreis kann so der aktuelle Wert des Silberlagers des Lieferanten ermittelt werden.

Dieses Beispiel zeigt, wie durch Hinzunahme eines neuen Moduls die vorhandene DSL erweitert wird. Das neue Modul bietet neue Eigenschaften an (stock), fügt eventuell neue Regeln hinzu und greift allenfalls auf neue Datenquellen zu (ERP-System des Silberlieferanten).

Implementierung einer DSL

Es gibt es zwei grundlegend unterschiedliche DSL-Typen: die interne und die externe DSL. Eine externe DSL ist eine formale Sprache, die analog zu einer generellen Programmiersprache (*General Purpose Language*) einen eigenen Übersetzer (Compiler) oder Interpreter besitzt, welcher den DSL-Ausdruck in eine ausführbare Anweisung überführt oder ihn direkt interpretiert.

Bei einer internen DSL hingegen wird der Compiler oder Interpreter einer bestehenden Programmiersprache verwendet, der sogenannten Host-Sprache. In dieser Host-Sprache wird eine Engine für die DSL entwickelt, welche die Möglichkeiten der Hostsprache nutzt, um eine Umgebung zu schaffen, in der jeder DSL-Ausdruck auch zu einem gültigen Ausdruck in der Host-Sprache wird. Nicht alle Host-Sprachen sind dafür geeignet. Hervorragende Kandidaten sind Sprachen wie Scala, Groovy und Ruby, da diese eine Syntax haben die sehr flexibel ist, wodurch der Charakter der Host-Sprache nicht oder nur eingeschränkt in einem DSL Ausdruck zu erkennen ist.

In APPRIS haben wir uns für die Implementierung einer DSL-Engine in der Host-Sprache Groovy entschieden, weil interne DSLs in einem modularen Umfeld gegenüber externen DSLs einige Vorteile bieten:

- Viele Host-Sprachen haben Sprach-Features, wie zum Beispiel Interfaces und Erweiterbare Klassen, welche ein modulares Konzept effizient unterstützen.
- Ein Modulentwickler muss nur die Host-Sprache sowie die Funktionsweise der DSL-Engine verstehen und nicht eine ganz neue Compiler-Infrastruktur.
- Es existieren diverse Frameworks, wie zum Beispiel OSGi (*Open Services Gateway initiative*), die es erlauben, Anwendungen und ihre Dienste per Komponentenmodell zu modularisieren und zu verwalten.

Groovy ist eine moderne, dynamische Sprache für die Java Virtual Machine (JVM). Groovy bietet viele Techniken, die zur DSL-Entwicklung verwendet werden können und die mit relativ geringem Aufwand in eine modulare Umgebung übertragbar sind. Da Groovy auf der JVM läuft und eine sehr gute Java-Integration hat, kann man das umfangreiche Java Ökosystem problemlos nutzen. Dies erlaubt eine Implementierung der DSL-Engine, welche nicht nur Groovy-, sondern zum Beispiel auch Java-Komponenten enthält.

Modulare DSL-Engine

Die Kernstücke unserer DSL-Engine sind die Groovy Shell (ein Groovy-Interpreter) und das OSGi-Framework. Über das OSGi-Framework werden Module geladen, welche Dienste zur Verfügung stellen, um die DSL zu erweitern. Die Engine baut aus diesen Diensten eine Umgebung auf, in deren Kontext DSL-Statements ausgeführt werden können.

Anhand des Groovy Sprach-Features *Category* zeigen wir, wie entsprechende Groovy Sprach-Features auch in einer modularen Umgebung eingesetzt werden können. Eine Kategorie ist eine Klasse mit statischen Methoden. Diese stellen *Extension-Methods* dar, d.h. sie erweitern eine bestehende Klasse um diese pseudo-statischen Methoden. Eine Kategorie, die zum Beispiel alle Integer-Instanzen um die Methode *printWithUnit()* erweitert, ist in Listing 3 aufgeführt. Der erste Parameter einer solchen Methode wird jeweils als Receiver-Objekt interpretiert (siehe *self* in Listing 3).

```
class IntegerCategory {
  static printWithUnit(Integer self,
          String unit) {
    println("""${self} ${unit}""")
  }
}
```

Listing 3: Beispiel einer Groovy Category

Eine Kategorie kann für einen Codeblock mit Hilfe der *use()*-Methode aktiviert werden. Die Groovy-Laufzeitumgebung sorgt dafür, dass die statischen Methoden der aktiven Kategorien beim dynamischen Binden miteinbezogen werden. In Listing 4 wird gezeigt wie die Kategorie aus Listing 3 verwendet wird.

```
use(IntegerCategory) {
  5.printWithUnit "Meter"
}
```

Listing 4: Verwendung der *IntegerCategory* führt zur Ausgabe «5 Meter»

Bei einer Erweiterung der DSL kann zum Beispiel ein neues Modul einen OSGi-Dienst zur Verfügung stellen. Das Modul wird eine entsprechende Erweiterungsschnittstelle, d.h. ein Interface zur Erweiterung der DSL, beinhalten müssen. Auf diesem Interface gibt es eine Methode *getCategories()*, die eine Liste aller *Category*-Klassen des Moduls zurückgibt. Die DSL-Engine selber aktiviert nun alle Kategorien von jeder DSL-Erweiterung. So hat die DSL-Engine über die aktivierten Kategorien Zugang zu neuer Funktionalität, welche in den *Extension-Methods* implementiert ist. Dies erlaubt

es jederzeit neue Methoden zu definieren, zu implementieren und zu integrieren, welche dann aus der DSL heraus angesprochen werden können.

Die Erweiterungsschnittstelle definiert weitere Methoden, die es erlauben andere Groovy Sprach-Features zu verwenden, die in einem modularen DSL-Umfeld wichtig sind [Dae10], wie zum Beispiel: *Metaclass*, *Builders* und *Expandos*. Die DSL-Erweiterungsschnittstelle ist eine herkömmliche Java-Schnittstelle, weshalb sie nicht nur mit Groovy-Komponenten verwendet werden kann, sondern zum Beispiel auch mit normalen Java-Komponenten.

Praxisbeispiel APPRIS

Kehren wir zu unserem eingangs gezeigten Beispiel mit dem Silberpreis zurück. Um die Regel aus Listing 1 ausführen zu können, müssen folgende OSGi-Module existieren:

1. Ein Basismodul, welches die DSL-Struktur und die Schlüsselwörter „when“, „>“, „then“, „produce“ und „warning“ zur Verfügung stellt.
2. Ein Price-Modul, welches die Methode „price“ zur Verfügung stellt und Abfragen auf unterschiedlichen Datenquellen zur Preisermittlung durchführt.
3. Ein Money-Modul, welches Währungen verwalten und Umrechnungen zwischen den Währungen durchführen kann und welches auch den Ausdruck „CHF“ bereitstellt.

Um die Regel aus Listing 2 aufsetzen zu können, muss die bestehende DSL um ein neues Modul ergänzt werden. Mit dem neuen Modul sollen die Lagerbestände eines Lieferanten miteinbezogen werden können, vorausgesetzt eine entsprechende Datenquelle ist angebunden. Die OSGi-Modullandschaft für diese erweiterte DSL ist in Abbildung 1 dargestellt. Wir sehen, dass der Funktionsumfang der DSL aus unterschiedlichen Modulen kommt. Es gibt dabei Module, welche die DSL direkt erweitern und solche, welchen bestimmten Modulen neuen Datenquellen hinzufügen. Die Module können auch Abhängigkeiten untereinander haben. Das Price-Modul ist zum Beispiel vom Money-Module abhängig. Dadurch können Preise in der korrekten Währung ausgegeben werden.

Implementierung einer DSL-Erweiterung

Anhand des Stock-Modules wollen wir nun zeigen, wie eine DSL-Erweiterung aussehen könnte. Das Money-Module führt den Multiplikationsoperator „*“ ein, um den Preis einer Einheit (z.B. Gramm) mit der Anzahl Einheiten multiplizieren zu können. Damit sich jedoch der Ausdruck „price * stock“ verarbeiten lässt, muss zudem eine Eigenschaft „stock“ vorhanden sein, welche die Lagermenge (in Gramm) für einen Lieferanten und ein Produkt ausfindig macht. Diese Lagermenge wird über einen Dienst eines OSGi-Modul geliefert. Damit kann das Stock-Modul die entsprechenden

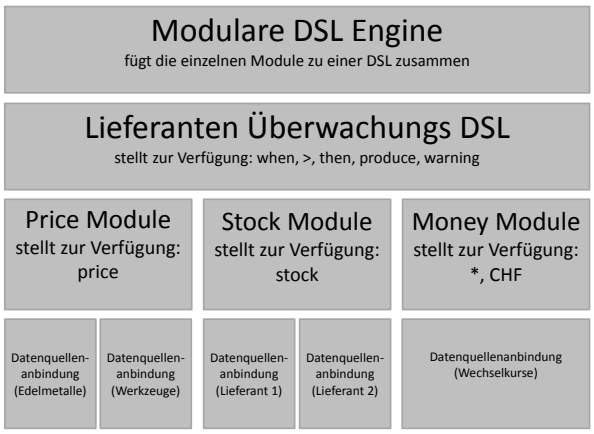

Abbildung 1: Module für den DSL-Ausdruck in Listing 2

Daten durch die „stock“-Eigenschaft der DSL zur Verfügung stellen.

Die DSL-Engine führt alle Ausdrücke innerhalb einer Instanz einer Klasse namens Global aus. Das heisst, dass die *this*-Referenz während der DSL-Ausführung vom Typ Global ist. In Groovy wird der Ausdruck „stock“ als *this.getStock()* interpretiert. Wir müssen also sicherstellen, dass die Methode *getStock()* auf der *this*-Instanz vorhanden ist. Dies kann durch eine Kategorie erreicht werden, welche die Global-Klasse um die Methode *getStock()* erweitert (siehe Listing 5).

```
class StockCategory {
  static getStock(Global self) {
    return …
  }
}
```

Listing 5: Implementierung der Kategorie „Stock“

Die neue Kategorie muss anschliessend in der DSL-Engine registriert werden. Dazu wird das *IDslExtensionInterface* implementiert. Für unser Beispiel ist hier nur die Methode *getCategories()* von Interesse (siehe Listing 6).

```
class StockExtension implements
          IDslExtensionInterface {
  …
  public List<Class> getCategories() {
    return [StockCategory]
  }
  …
}
```

Listing 6: Implementierung der Stock-Extension über die Erweiterungsschnittstelle

Damit die Erweiterung als OSGi-Dienst verwendet werden kann, braucht es noch eine Registrierung, die je nach verwendetem OSGi-Framework und der gewählten Technik unterschiedlich aussehen kann.

Wird nun ein DSL-Ausdruck ausgeführt, holt sich die DSL-Engine alle *IDslExtensionInterface*-Dienste, sammelt alle Kategorien und aktiviert sie. Dann wird der Ausdruck interpretiert und dabei wird die Eigenschaft „stock“ ausgeführt, wo-

bei das dynamische Binden nach einer Methode namens *getStock()* auf der *this*-Instanz sucht und unsere *Category*-Methode findet und ausführt.

Fazit und Ausblick

Die vorgestellte DSL-Engine erlaubt es eine DSL zu entwickeln, die im Einsatz wachsen kann. Groovy hat sich als eine sehr gute Hostsprache für eine modulare DSL herausgestellt. Dank der dynamischen Natur von Groovy können einzelne DSL-Module unabhängig voneinander kompiliert und in die DSL-Engine integriert werden. Die Sprach-Features, die in Groovy für eine DSL verwendet werden können, sind grösstenteils leichtgewichtig und bieten trotzdem viele Einsatzmöglichkeiten.

Unsere DSL-Engine unterstützt die Möglichkeit, eine DSL zu definieren, die je nach Einsatzgebiet aus unterschiedlichen Modulen besteht. Eine sinnvolle Erweiterung unserer DSL-Engine wäre beispielsweise ein Rollenkonzept, welches es erlauben würde, je nach Rolle andere DSL-Funktionalitäten zur Verfügung zu stellen. Jede Rolle nutzt dabei eine andere Menge von Modulen. Darauf basierend könnte auch ein Sicherheitskonzept realisiert werden, welches sensitive Daten und Funktionen nur bestimmten Rollen zur Verfügung stellt.

Referenzen

[Dae10] Daerle, F.: Groovy for Domain-Specific Languages. Packt Publishing, 2010.

[Kne12] Knecht, M.: Modulare DSL zur Lieferantenüberwachung. Projektbericht, 2012. http://www.fhnw.ch/personen/markus-knecht/dateien/Modulare%20DSL%20zur%20Lieferantenüberwachung.pdf