Java-Spracherweiterung mit JetBrains‘ MPS

Version 1.1, Jan 5, 2009

Markus Völter, Independent/itemis
(voelter@acm.org)

Abstract

Jetbrains, der Hersteller von IntelliJ IDEA, Resharper und anderen Entwicklungwerkzeugen haben am 10. Dezember die Beta-Version von MPS, dem Meta Programming System, veröffentlicht. In einem früheren Artikel [1] hatte ich bereits kurz darüber geschrieben.

MPS ist ein System zur sprachorientierten Programmierung (Language Oriented Programming). Mit MPS lassen sich effizient domänenspezifische Sprachen erstellen, kombinieren und verwenden. In diesem Artikel zeige ich, wie man mit MPS Java um eigene Sprachkonzepte erweitert: ein neues Schlüsselwort, um einfacher mit Java 5 Locks zu arbeiten.

MPS ist kein klassischer Texteditor, weswegen einige Editieroperationen nicht in gewohnter Weise funktionieren. Dies ist in einem Artikel wie diesem sehr schwer zu beschreiben, weswegen ich es erst gar nicht versuche, sondern begleitende Screencasts erstellt habe [2].

Hintergrund

Domänenspezifische Sprachen (DSLs) sind Sprachen, die auf eine bestimmte Anwendungsdomäne zugeschnitten sind. Bei MPS handelt es sich um ein System zur Implementierung textueller, externer DSLs die durch Transformationen auf ausführbaren Code abgebildet werden.

Besonders spannend ist, dass man mittels MPS sehr leicht Sprachen kombinieren und erweitern kann. MPS kommt mit einer Reihe von vorimplementierten Sprachen, darunter XML und *BaseLanguage*, einer Implementierung von Java. Diese *BaseLanguage* kann man nun beispielsweise um eigene Sprachkonzepte erweitern. Damit erlaubt MPS eine völlig neuartige Herangehensweise an DSLs. Man startet mit der Basisprogrammiersprache (*BaseLanguage* bzw. Java, zukünftig auch andere) und ergänzt diese inkrementell um domänenspezifische Konzepte (oder beliebige andere Idiome) die sich im Laufe eines Projektes herauskristallisieren. Im Gegensatz zum entsprechenden Ansatz in Ruby oder anderen dynamischen Sprachen kann man weiterhin mit einem statischen Typsystem und den entsprechende IDE-Annehmlichkeiten rechnen. Spracherweiterung gehen automatisch mit einer IDE-Erweiterung

einher. Die neue Syntax verhält sich bezüglich des Toolings exakt so, wie die Syntax der Basissprache.

Funktionsprinzip und Überblick

Das Kernprinzip von MPS ist Structured Editing. Im Gegensatz zu klassischen Programmiersprachen bearbeitet man in MPS keinen Quelltext, sondern arbeitet direkt auf dem Syntaxbaum, der allerdings genauso aussieht wie Quelltext und sich auch *fast* so verhält. Wenn man beispielsweise *int i;* eingibt, so erstellt man tatsächlich eine Instanz des Konzepts *VariableDeclaration* dessen Namensattribut den Wert „i" hat. Ein klassischer Parsevorgang findet nie statt. Die Vermeidung des Lexens und Parsens hat den Vorteil, dass man bei der Kombination von Sprachen keine Grammatiken verknüpfen muss.

Um beispielsweise ein neues Schlüsselwort in Java einzuführen, geht man folgendermaßen vor: Man definiert eine neue Sprache die von *BaseLanguage* erbt. In dieser neuen Sprache definiert man ein neues Konzept welches seinerseits von dem Java- Konzept *Statement* erbt. Dann definiert man einen Editor, der die syntaktische Struktur des neuen Statements definiert. Wenn man dann eine Instanz (Programm, Modell) dieser neuen Sprache anlegt, kann man das neue Statement überall dort verwenden, wo Java ein *Statement* erwartet. Um es dann auch ausführen zu können, muss man noch Reduktionsregeln definieren (das sind mehr oder weniger Codegenerierungstemplates) die das neue Sprachkonzept auf original Java Code zurückführen. Wir werden dies nun anhand eines Beispiels zeigen.

Unser Beispiel

Motivation

Im Rahmen unseres Beispiels wollen wir ein neues Schlüsselwort einführen für den Umgang mit *java.util.concurrent.locks.Lock*. Dieses Interface liegt in verschiedenen Implementierungen vor die ihrerseits verschiedene Locking-Strategien umsetzen. Üblicherweise arbeitet man mit *Locks* wie mit allen Acquire/Release Geschichten in Java:

```
Lock l = …
try {
  l.lock();
  // do something with the locked resource
} finally {
   l.unlock();
}
```

Dieses Vorgehen hat zwei Probleme: zum einen ist es relativ viel Text den man schreiben (und lesen!) muss. Zum anderen kann man das *unlock()* natürlich auch vergessen, was dann sehr wahrscheinlich in einem Deadlock resultieren wird.

Eine kompaktere und sicherere Syntax wäre folgende:

```
Lock l = …
lock( l ) {
  // do something with the locked resource
}
```

Diese Syntax soll natürlich semantisch identisch sein mit der obigen; wir bilden also die neue Syntax auf obigen *try-finally*-Block ab.

Mit MPS lässt sich eine solche Java Erweiterung sehr leicht implementieren. Im Rest dieses Artikels zeige ich, wie.

Sprachen und Lösungen

Die grobgranularste Organisationsstruktur in MPS sind Projekte. Darunter finden sich Lösungen (Solutions) und Sprachen (Languages). Sprachen enthalten Metaprogramme, eben alles was zur Definition neuer Sprachen und Spracherweiterungen notwendig ist. Dazu gehören

- Konzepte, also die Definition von abstrakter Syntax (oder das Metamodell, je nach Terminologie).
- Editoren, also die Abbildung der Konzepte auf die „textuelle“ Syntax
- Constraints, also Regeln, die die Verwendung und Ausprägung der Konzepte einschränken (bspw. darf eine Variablenreferenz nur auf Variable im aktuellen Scope referenzieren)
- Typsystemregeln, die die Typisierung der Konzepte berechnen (bspw. die Tatsache, dass die *Expression* im *WhileStatement* vom Typ *boolean* sein muss)
- Generatoren, die Konzepte auf andere Konzepte (typischerweise solche aus der Basissprache) abbilden

Solutions sind dagegen Anwendungsprojekte (Modelle) die die eine oder mehrere Sprachen verwenden.

Wir wollen hier eine Spracherweiterung erstellen. Dementsprechend legen wir in einem beliebigen Projekt eine neue Language an. Wir geben ihr den Namen *concurrent*.

Da unsere neue Sprache auf Java (also *BaseLanguage)* beruhen soll, tragen wir in den Language Properties von *concurrent* ein, dass diese neue Sprache *BaseLanguage* erweitert.

Konzeptdeklaration

Innerhalb dieser neuen Sprache legen wir ein neues Konzept an, definieren also die abstrakte Syntax. Aus dem Beispielcode oben geht hervor, dass wir unser neues Konzept überall da verwenden wollen, wo *BaseLanguage* ein *Statement* erlaubt. Demnach erbt unser *LockStatement* von dem aus

BaseLanguage importierten abstrakten Konzept *Statement*. Konzepte verhalten sich ähnlich wie Klassen und können demnach auch von anderen Konzepten erben. Wie in der OO Programmierung, können Instanzen von Untertypen eines Konzeptes überall dort verwendet werden, wo eine Variable auf den Obertyp getypt ist. In unserem Fall gibt es *InstanceMethodDeclarations*, die eine *StatementList* enthalten, die ihrerseits *Statements* enthalten.

Abbildung 1 zeigt die Situation in UML Notation.

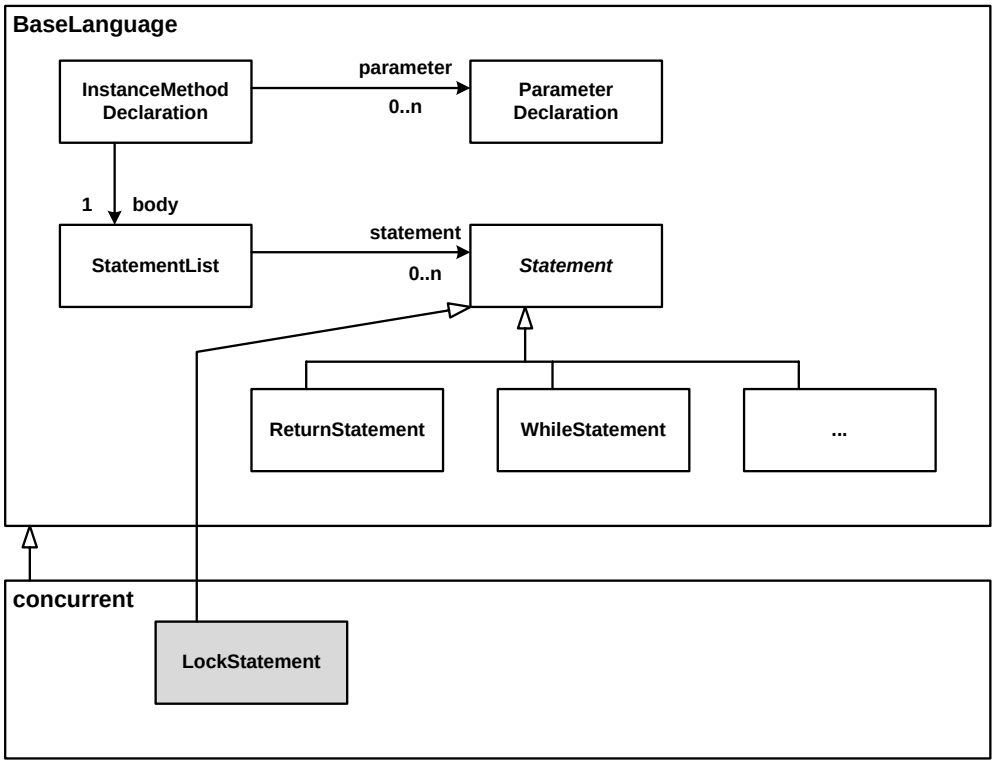

Abbildung 1: Abstrakte Syntax mit dem neuen LockStatement

Unser *LockStatement* hat zwei Eigenschaften. Zum einen enthält es eine *Expression* die das *Lock*-Objekt zurückliefert, das wir zur Synchronisation verwenden wollen. Zum anderen hat *LockStatement* einen *body* vom Typ *StatementList*. *Expression* und *StatementList* sind Konzepte die in *BaseLanguage* definiert werden; wir verwenden diese hier lediglich. Abbildung 2 zeigt die Konzeptdeklaration für das *LockStatement*.

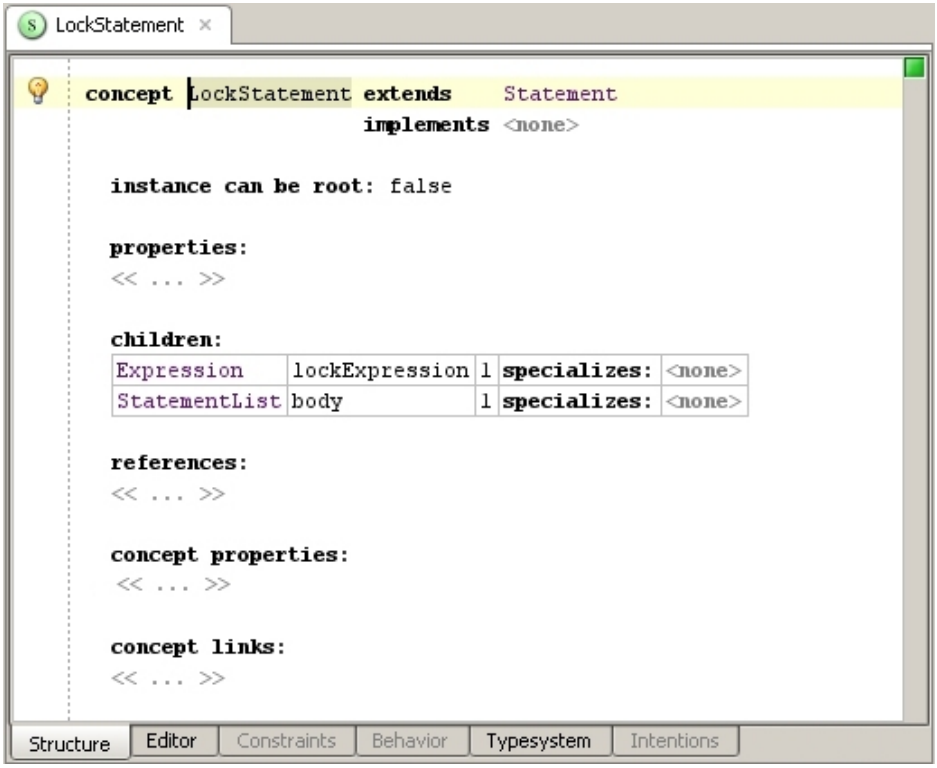

Abbildung 2: Konzeptdeklaration für das LockStatement

Konkrete Syntax

Bevor wir unser neues Sprachkonstrukt ausprobieren können müssen wir noch die konkrete Syntax definieren. Vielleicht sollten wir das aber auch eher Notation nennen, oder, wie es MPS vorschlägt, Editor. MPS parst nicht. Es gibt also keine Grammatik im eigentlichen Sinne. Wir definieren lediglich, wie und wo die einzelnen Eigenschaften eines Konzeptes dargestellt werden. Abbildung 3 zeigt den Editor.

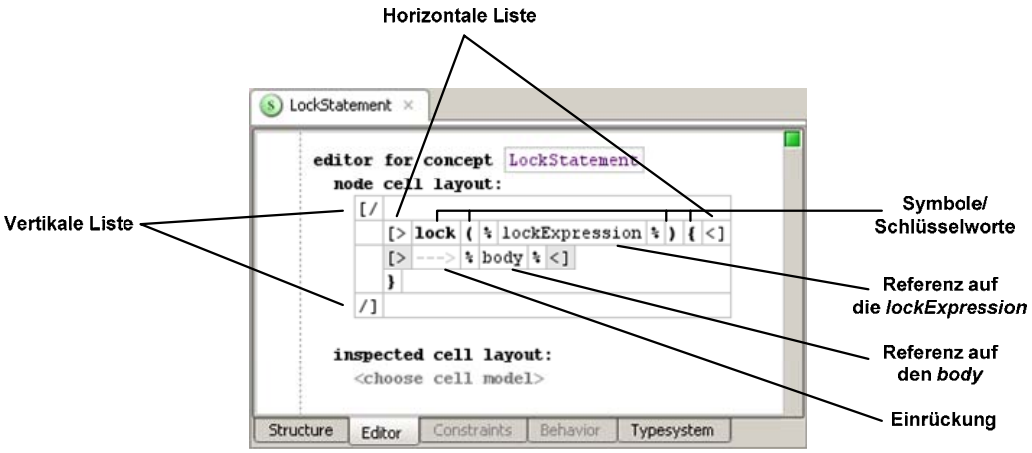

Abbildung 3: Editordefinition für LockStatement

Die Beschreibung des Editors geschieht (konsequenterweise!) mit einer DSL zur Editorbeschreibung. Der Editor besteht aus geschachtelten Zellen, die mittels horizontalen oder vertikalen Listen gelayoutet werden. Die Editordefinition bezieht sich dabei natürlich auf das Konzept, für das er den Editor darstellt.

Ausprobieren, Teil 1

Da wir nun die neue Abstraktion inklusive der Notation definiert haben, können wir das ganze nun ausprobieren. Wir legen dazu eine neue *Solution* an. In den Properties dieser Solution fügen wir unsere neue Sprache *concurrent* in die Liste der *Used Languages* ein. Damit steht uns unser „erweitertes Java“ zur Verfügung.

Innerhalb der Solution erstellen wir ein neues *Model* … und darin eine neue *Class*. Um unser neues *Statement* ausprobieren zu können, brauchen wir natürlich zunächst eine Klasse mit einer Methode. Auch dem Model müssen wir erklären, dass wir die Sprache *concurrent* verwenden möchten, daher müssen wir in den *Model Properties concurrent* zu den *Used Languages* hinzufügen.

Wir können nun innerhalb der Methode unser *LockStatement* benutzen. Abbildung 4 zeigt dies.

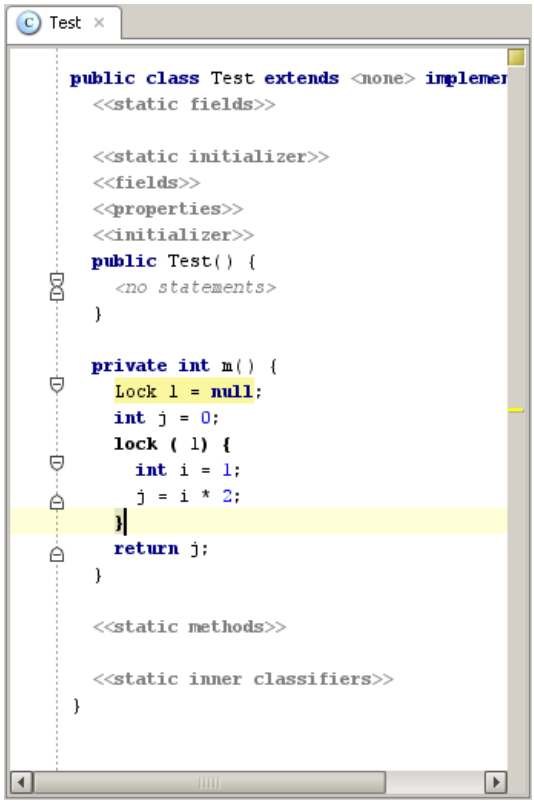

Abbildung 4: Verwendung des LockStatements

Bei der Verwendung unseres Statements fallen aber zwei Dinge auf. Zum einen kann eine beliebige Expression verwendet werden – unabhängig von

ihrem Typ. Wir könnten schreiben *lock(2+4) { … }* was natürlich keinen Sinn macht. Wir müssen den Typ der Expression einschränken.

Zum anderen müssen wir unser neues Statement explizit per Ctrl-Leertaste aus der Dropdown-Liste auswählen. Im Gegensatz dazu kann man im Falle des *while (…) {…}* einfach nur „while" tippen, und es erscheint das Skelett eines *WhileStatement*.

Alias

Um letzeres Problem zu lösen, definieren wir für unser Konzept einen Alias. Ein Alias ist ein String, anhand dessen MPS erkennt, welche Auswahl aus einer Dropdown-Liste getroffen werden soll. In den *Concept Properties* von *LockStatement* fügen wir *Alias* = *lock* hinzu. Wir können nun dort wo ein Statement erwartet wird einfach „lock" tippen und – voilà – wir sehen direkt das Skelett eines *LockStatements*.

```
public void m2() {
  lock ( <lockExpression>) {
    <no statements>
  }
}
```

Typsystem

Damit das *LockStatement* Sinn macht, muss der Typ der *lockExpression* natürlich *Lock* sein. Dazu müssen wir eine Typsystemregel definieren. Abbildung 5 zeigt diese Regel. Sie besagt, dass der Typ der *lockExpression* eines *LockStatements* vom Typ *Lock* sein muss. Damit dies funktioniert, müssen wir in den *Model Properties* des Typsystem-Modells das Paket *java.util.concurrent.locks@java_stub* zu den *Imported Models* hinzufügen.

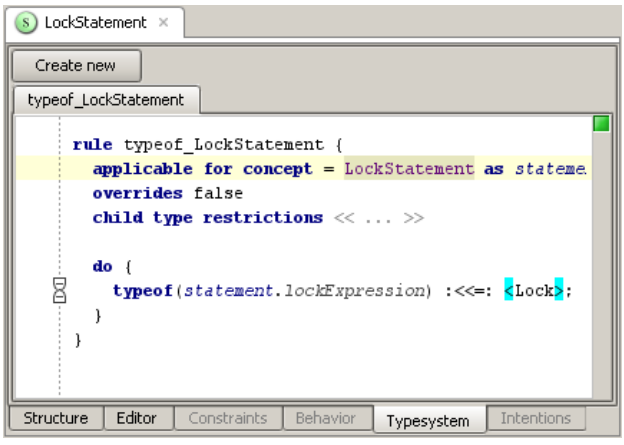

Abbildung 5: Typsystemregel

Wenn wir nun eine Expression mit falschem Typ innerhalb des *lock(…) {}* eintragen, werden wir einen Fehler im Editor bekommen.

Und um das nochmal zu verdeutlichen: man kann hier wirklich vollständige Java Expression zur „Berechnung“ des *Locks* verwenden; siehe Abbildung 6.

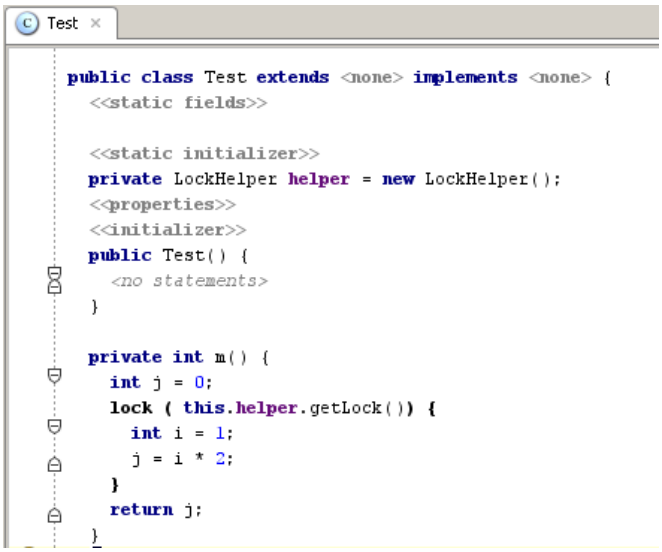

Abbildung 6: Verwendung einer Expression zur "Berechnung" des Locks

Generator

Um den mit unserem *LockStatement* erweiterten Code kompilieren zu können, muss das *LockStatement* natürlich auf „echtes“ Java zurückgeführt werden.

MPS kommt mit einem Framework zur Codegenerierung – wobei man eigentlich sagen müsste, dass es sich um ein Framework zur Modelltransformation handelt, da man ja keinen Text erzeugt, sondern Instanzen der Zielsprache. Das Framework enthält verschiedene Transformationsregelarten. Wir definieren eine Reduktionsregel. Reduktionsregeln sind Übersetzungsregeln, die automatisch ausgeführt werden, wenn eine Instanz des Ausgangselements im Modell gefunden wird.

Wir legen innerhalb unserer neuen Sprache *concurrent* einen neuen *Generator* an. Darin befindet sich zunächst nur eine *Mapping Configuration* namens *main*. In dieser erstellen wir eine neue *Reduction Rule*.

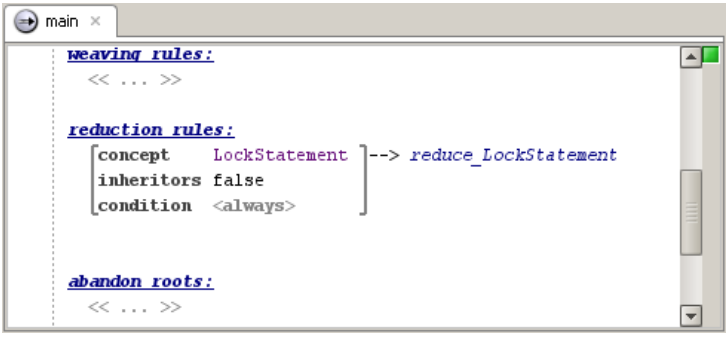

Abbildung 7: Die Konfiguration der Reduction Rule

Die Deklaration in Abbildung 7 besagt, dass Instanzen von *LockStatement* immer mit Hilfe der Reduktionsregel *reduce_LockStatement* verarbeitet werden sollen.

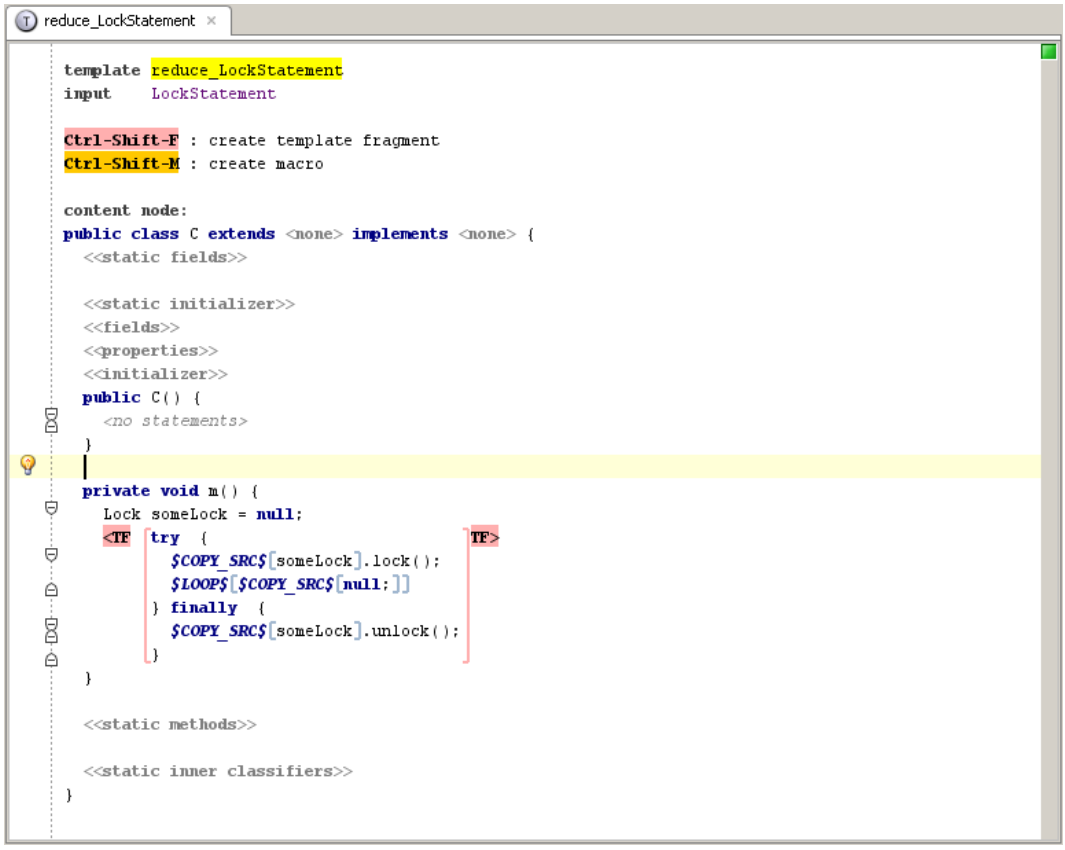

Abbildung 8: Die Reduktionsregel für LockStatements

reduce_LockStatement an sich ist in Abbildung 8 gezeigt – und bedarf sicher einiges an Erläuterungen. Übersetzen wollen wir ja nun ein *LockStatement*. Aufgrund der Struktur von *BaseLanguage* können diese allerdings nur im Kontext einer Methode auftauchen. Damit wir dies erreichen, definieren wir eine Dummy-Klasse (*C*) und eine Dummy-Methode (*m*), in der wir das Template-Fragment zur Reduktion einbetten können. Wie gesagt: zur Reduktion unseres *LockStatements* ist nur das Template-Fragment relevant, die Struktur der Klasse und der Methode ist nicht, sie dienen nur als strukturelle Kontext für das *LockStatement*.

Dies scheint auf den ersten Blick vielleicht etwas seltsam. Ich gebe aber zu bedenken, dass derartige Transformationsregeln tatsächlich Codecompletion etc. für die generierte Zielsprache ermöglichen – etwas, was man in traditionellen Codegeneratoren nicht findet. Im Endeffekt führen wir hier eine Modell-zu-Modelltransformation aus, wobei wir die konkrete Syntax der Zielsprache verwenden!

Zurück zur eigentlichen Transformation. Im Template Fragment legen wir einen *try-finally*-Block an, da ein *LockStatement* in einen *try-finally*-Block übersetzt werden soll. Innerhalb dessen wird in der ersten Zeile auf dem Code der *lockExpression* die Methode *lock()* aufgerufen. Wir wissen ja, dass die Expression den Typ *Lock* hat, somit können wir die betreffende Methode typsicher aufrufen. Wir verwenden das *COPY_SRC* Makro, um den Inhalt der *lockExpression* zu klonen und im *try-finally*-Block einzubetten; wir rufen dann *lock()* darauf auf. In der zweiten Zeile im *try*-Block iterieren wir über die Statements im *body* des *LockStatements* und kopieren diese hierher. Der Code für das *unlock()* funktioniert genauso wie der für das Locking.

Abbildung 9 zeigt den Code im Inspektor für das *LOOP* Makro. Selbst schreiben muss man nur die Expression *node.body.statement*. *node* ist der Kontextknoten, also unser *LockStatement*. Darauf rufen wir das Property *body* auf, das die *StatementList* enthält, die im den Body des *lock()*-Statements zu finden ist (siehe Konzeptdefinition für *LockStatement*). *statement* ist dessen Property das die eigentlichen Statements enthält. Per *LOOP* iterieren wir über diese, um die dann im Body des *try-finally*-Blocks einzubetten.

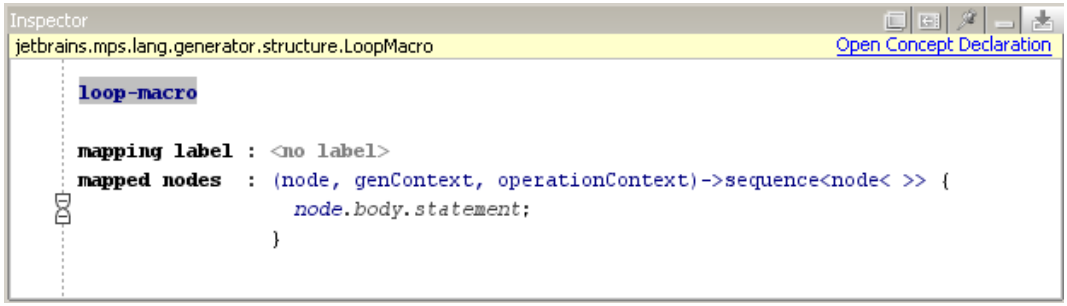

Abbildung 9: Der Inspektor für das LOOP Makro

Damit sind wir am Ende der Implementierung, und wir können das ganze nochmal ausprobieren.

Ausprobieren, Teil 2

Wir können nun unser Modell wie wir es oben erstellt haben übersetzen. Damit dies funktioniert, müssen wir in den Model Properties des Beispielmodells noch *concurrent* bei den zur Generierung verwendeten Sprachen eintragen. Dann einfach *Generate Files* im Kontextmenü des Modells in der Solution verwenden.

Um das Generat zu inspizieren, einfach im Explorer in den *File System View* wechseln und in der Solution den Ordner *source_gen* öffnen. Dort finden sich *LockHelper.java* (ohne Veränderungen) und *Test.java* – letzeres aber mit dem korrekt reduzierten *LockStatement*.

```
private int m() {
    int j = 0;
    try {
      this.helper.getLock().lock();
      int i = 1;
      j = i * 2;
    } finally {
      this.helper.getLock().unlock();
    }
    return j;
  }
```

Dies ist regulärer Java Code, der nun kompiliert werden kann. MPS kommt mit einer Integration in IntelliJ IDEA, um den resultierenden Quellcode zu übersetzen.

Fazit

Wie obiges Beispiel zeigt, ist die domänenspezifische Erweiterung von Java mit MPS wirklich relativ einfach. Natürlich hat das Werkzeug noch einige Macken, es befindet sich ja auch noch in der Beta-Phase. Der Ansatz an sich ist meiner Meinung nach aber sehr erfolgversprechend. Die inkrementelle Erweiterung einer allgemeinen Programmiersprache stellt eine erheblich niedrigere Hürde dar bzgl. der Einführung domänenspezifischer Sprachen.

Wie ich im vorigen Artikel [1] schon erwähnt habe, ist der Haken an der Sache eben der, dass man mit MPS arbeiten muss (und nicht auf beliebige Texteditoren umsteigen kann) und dass das Editieren von Bäumen ein klein wenig Umgewöhnung fordert. Je mehr ich mich mit MPS beschäftige, desto mehr komme ich zu der Überzeugung, dass man diese Umstellung schaffen kann, und es sich auch lohnt.

Zum Abschluss verweise ich nochmal auf die Screencasts [2], die verdeutlichen, wie man mit MPS tatsächlich arbeitet.

Referenzen

[1] http://it-republik.de/jaxenter/news/Neues-DSL-Werkzeug-Jetbrains-Meta-Programming-System-046601.html

[2] http://drop.io/mpsdemo_Lockstatement

Über den Autor

Markus Völter arbeitet als freiberuflicher Berater und Coach für die itemis AG in Stuttgart. Seine Schwerpunkte liegen dabei auf Architektur,

Modellgetriebener Softwareentwicklung und domänenspezifischen Sprachen sowie Produktlinenengineering. Er hält regelmäßig Vorträge auf den entsprechenden Konferenzen und ist (Mit-) Autor verschiedener Bücher, Patterns und Artikel. Markus ist zu erreichen unter www.voelter.de sowie via voelter@acm.org