Codetransformatie

Met Spoon

Bewust of onbewust maakt elke Java ontwikkelaar gebruik van compile time analyse en transformatie van broncode en bytecode. We laten bijvoorbeeld onze IDE code voor ons genereren en gebruiken tools als Findbugs voor het opsporen van fouten. Het zelf bouwen van code-analyse en generatietools kan best complex zijn. De Java Reflection API biedt ons de mogelijkheid tot bijvoorbeeld code-analyse. Deze staat echter geen transformatie toe. Gelukkig is er Spoon!

Spoon (**http://spoon.gforge.inria.fr**) is een framework, waarmee Java code gemakkelijk is aan te passen en te creëren. Het is gebaseerd op een simplificatie van de Java Abstract Syntax Tree (AST), die iedere standaard Java compiler genereert. Dit metamodel vormt de basis voor Java source code-analyse en transformatie, volledig geschreven in Java. Hierdoor is Spoon een krachtig framework met een lage instapdrempel voor Java ontwikkelaars.

Waarom code-analyse en transformatie?

Zowel voor code-analyse als codetransformatie zijn verschillende praktische toepassingen te bedenken. Voor code-analyse zou je bijvoorbeeld specifieke validaties kunnen maken, bijvoorbeeld voor het opsporen van beveiligingsfouten op codeniveau. Wij hebben gebruik gemaakt van de analyse functionaliteit van Spoon om te kijken of het mogelijk is om een oplossing voor een programmeeropdracht automatisch te beoordelen op basis van de gebruikte codeconstructies. Codeconstructies zijn bijvoorbeeld if statements, declaraties van variabelen, combinaties van gebruikte operators, etc.

Huidige geautomatiseerde beoordelingsmethoden schieten onder andere te kort in het beoordelen van goede oplossingen met kleine fouten. Waar mensen hier nog steeds een ruime voldoende toekennen, scoren de oplossingen vaak erg laag in een automatische beoordeling. Het doel was om te onderzoeken of we een lerend beoordelingsmodel konden genereren, dat in staat is oplossingen te beoordelen naar menselijke maatstaven, dankzij de met Spoon gedefinieerde codeconstructies. Spoon stelde ons in staat om relatief eenvoudig deze codeconstructies te classificeren en te tellen, binnen hun context. Zo kun je bijvoorbeeld alle variabele toewijzingen binnen een methode opvragen of van een if statement alle geneste statements op te vragen. Ook kun je Spoon gebruiken voor het generen van documentatie, zoals RAML beschrijvingen voor REST API's.

Met betrekking tot codetransformatie kun je denken aan het automatisch loggen van methode aanroepen. Door middel van een annotatie geef je aan welke methodes je gelogd wilt hebben. Met Spoon kun je code aan deze methodes toevoegen, die deze logging uitvoeren. Ook kun je denken aan het generen van proxy klassen met extra functionaliteit, zoals transacties of security op basis van bestaande klassen.

Erwin de Gier is software architect en trainer bij de business line Open Source van Sogeti.

Spoon

Om gebruik te maken van Spoon voegen we de volgende Maven dependency toe (zie **Listing 1**).

```
<dependency>
   <groupId>fr.inria.gforge.spoon</groupId>
   <artifactId>spoon-core</artifactId>
   <version>5.4.0</version>
</dependency>
```

Listing 1

De Spoon API kan worden aangesproken via Java code of via een Maven plug-in. De Maven plug-in maakt het mogelijk om de code-analyses en transformaties onderdeel te maken van je build proces (zie **Listing 2**).

Binnen het configuratiedeel van de plug-in kun je Processors toevoegen. Een processor kun je zien als een taak, die je zelf kunt definiëren. Het is het startpunt voor Spoon voor het uitvoeren van een analyse of een transformatie. Als je een nieuwe transformatie of analyse wilt gaan schrijven, dan begin je dus met het maken van een Processor. Het direct aanroepen van Spoon vanuit Java code gaat als volgt (zie **Listing 3**).

```
<plugin>
  <groupId>fr.inria.gforge.spoon</groupId>
  <artifactId>spoon-maven-plugin</artifactId>
  <version>2.2</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <processors>
      <processor>nl.edegier.spoon.GetSetProcessor</processor>
    </processors>
  </configuration>
</plugin>
```

Listing 2

```
final SpoonAPI spoon = new Launcher();
spoon.addInputResource("src/main/java");
spoon.setSourceOutputDirectory("target");
spoon.addProcessor(new GetSetProcessor());
spoon.run();
```

Listing 3

Als input gebruiken we de Maven source folder: *src/main/java*. Als output folder gebruiken we 'target'. Vervolgens voegen we de GetSetProcessor toe. Deze code doet nu hetzelfde als de Maven plug-in configuratie. Zowel met de Maven plug-in als via de Java API is het mogelijk om meerdere processors tegelijkertijd uit te voeren. Nu we weten hoe we de Spoon Processors aansturen, kunnen we kijken naar een praktijkvoorbeeld voor de realisatie van een Processor.

Metamodel

Om de verschillende code elementen te beschrijven, maakt Spoon gebruik van een metamodel van de Java code. Het volledige metamodel staat beschreven op de website en zul je vaak als referentie gebruiken tijdens het werken met Spoon (**http://spoon.gforge.inria.fr/structural_elements.html**). Het metamodel van Spoon is specifiek gemaakt, zodat ontwikkelaars er eenvoudig mee kunnen werken. Dit in tegenstelling tot bijvoorbeeld het metamodel van de Java Compiler zelf, die gemaakt is om Java code om te zetten naar bytecode. **Diagram 1** bevat een aantal van de concepten, die we in het voorbeeld gebruiken, met als doel om meer context te geven bij het lezen van de codevoorbeelden. Het beschrijven van code met behulp van code is een abstracte bezigheid, die een extra denkstap vereist.

Een onderdeel van een stuk Java code heet een CtElement (compile time element). Een element kan bijvoorbeeld een instantieveld zijn, een methode, een codeblok of een hele klasse. CtType (**http://spoon.gforge.inria.fr/mvnsites/spoon-core/apidocs/index.html?spoon/reflect/reference/CtTypeReference.html**) dient als superklasse voor Java klassen, interfaces, annotaties en type parameters (generics). Onder CtClass is er ook nog een CtEnum, die wordt gebruikt om een Java Enum te beschrijven. In de voorbeelden komen we ook nog CtReturn tegen, dit is de beschrijving van een return statement. CtExpression wordt gebruikt voor het beschrijven van expressies, zoals bijvoorbeeld operators, literals en het gebruik van this.

Codetransformaties

Om te laten zien hoe je code toevoegt aan bestaande klassen, gaan we een generator maken voor get en set methoden. We maken hiervoor eerst een nieuwe annotatie met de naam GetSet. Omdat we deze annotatie alleen tijdens het build proces gebruiken, krijgt deze de retentie policy "source". Verder geven we aan dat deze annotatie alleen gebruikt kan

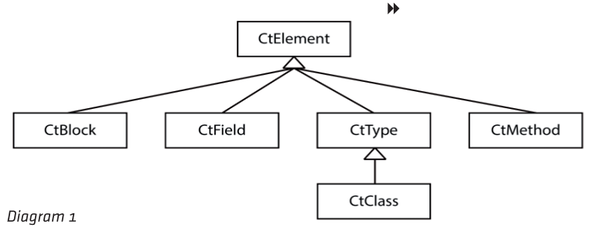

Diagram 1

worden op velden.

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.FIELD)
public @interface GetSet {
}
```

Listing 4

Als input voor de generator gebruiken we een eenvoudige klasse met als enige veld de naam van een persoon. Dit veld is geannoteerd met onze GetSet annotatie.

```
public class Person {
   @GetSet
   private String name;
}
```

Listing 5

De bedoeling is dat we met behulp van Spoon zowel een get als een set methode generen voor de instance variabele "name'. We maken hiervoor een Spoon Processor met de naam "GetSetProcessor". De processor heeft als superclass de Spoon klasse "AbstractProcessor<?>". Het subtype is CtClass, omdat de input voor deze processor een Java klasse is. We moeten zelf de process methode implementeren, deze heeft een parameter van het type CtClass (zie **Listing 6**).

De eerste stap is het opvragen van alle velden, die geannoteerd zijn met onze notatie "GetSet". Voor het zoeken naar bepaalde code-elementen gebruikt Spoon het concept van een Filter. Bij Spoon zitten een aantal Filters meegeleverd. Ook is het mogelijk om zelf een Filter te definiëren door het implementeren van de Filter interface. Deze interface heeft één methode ('matches') met als input een element en als output een booleaanse waarde, die aangeeft of het element voldoet aan het filter. In dit voorbeeld gebruiken we het bestaande "AnnotationFilter". Zoals de naam al aangeeft, kan dit filter gebruikt worden om te zoeken naar elementen, die voorzien zijn van een bepaalde annotatie. Door het type van onze GetSet annotatie als constructor argument mee te geven, geven we aan dat we willen filteren op deze annotatie. Aangezien de annotatie beperkt is tot het gebruik op velden, weten we zeker dat we alleen velden terugkrijgen. Per veld dat geannoteerd is met onze GetSet annotatie kunnen we nu een getter en een setter genereren (zie **Listing 7**).

Om een methode toe te voegen aan een bestaande klasse, moeten we eerst de methode aanmaken. Het creëren van code-elementen binnen Spoon gebeurt met behulp van een Factory. De base factory is beschikbaar in een processor door het aanroepen van de methode getFactory(). Met behulp van getFactory(). Method() krijg je toegang tot de MethodFactory. Om onze getName() methode aan te maken, geven we de volgende informatie mee aan de create() methode van de MethodFactory. (zie **Tabel 1**).

Het aanmaken van code blokken (CtBlock) doen we met de CodeFactory (getFactory().Code()). Return statements maak je aan met de CoreFactory (getFactory().Core().createReturn()). We voegen aan dit return statement een expressie toe, in ons geval is dat alleen de naam van het veld (return name;). We gebruiken hiervoor getFactory().Code().createCode SnippetExpression(). Onze methode is nu af. Wat nog rest is het toevoegen van de methode aan de klasse. Dit doen we door middel van ctClass.addMethod(getter).

```
public class GetSetProcessor extends AbstractProcessor<CtClass<?>> {

   @Override
   public void process(CtClass<?> ctClass) {
       for (CtElement ctElement : ctClass.getElements(new AnnotationFilter<>(GetSet.class))) {
           if (ctElement instanceof CtField) {
               CtField ctField = (CtField) ctElement;
               generateGetter(ctClass, ctField);
               generateSetter(ctClass, ctField);
           }
       }
   }
```

Listing 6

```
private void generateGetter(CtClass<?> ctClass, CtField<?> ctField) {
   CtReturn returnStatement = getFactory().Core().createReturn();
   CtExpression returnExpression =
  getFactory().Code().createCodeSnippetExpression(ctField.getSimpleName());
   returnStatement.setReturnedExpression(returnExpression);
   CtBlock body = getFactory().Code().createCtBlock(returnStatement);
   CtMethod getter = getFactory().Method().create(ctClass, new
  HashSet<ModifierKind>(Arrays.asList(ModifierKind.PUBLIC)),
   ctField.getType(), "get"+capitalize(ctField.getSimpleName()), null, null, body);
   ctClass.addMethod(getter);
}
```

Listing 7

Het resultaat van deze processor met als input onze Person klasse is als volgt:

```
public class Person {
   @nl.edegier.annotation.GetSet
   private java.lang.String name;

   public java.lang.String getName() {
       return name;
   }
}
```

Listing 8

Op een vergelijkbare manier kunnen we set methodes generen (zie **Listing 9**).

De set methode heeft als return type void. Dit geef je aan door een referentie aan te maken met als type void.class. De set methode heeft een lijst van parameters. In dit geval is er één parameter met het type en de naam van het veld waarvoor we de setter genereren.

In plaats van een return statement, maken we een code snippet aan op basis van een String (this.name = name). We gebruiken hier wederom de veldnaam voor. Het code snippet wordt weer toegevoegd aan een CtBlock, die als argument voor de create methode dient.
Het resultaat van deze processor is een set methode op onze Person klasse (zie **Listing 10**).

CtClass target	De klasse waarop de methode wordt aangemaakt: ctClass
Set<ModifierKind> modifiers	Ons methode moet public worden: ModifierKind.PUBLIC
CtTypeReference<?> returnType	Het return type van de methode is hetzelfde als het Type van het veld waarvoor we de getter maken: ctField.getType().
String name	De naam van de methode: get + ctField.getSimpleName()
List<CtParameter<?>> parameters	De lijst met parameters is in ons geval leeg: null
Set<CtTypeReference<? extends Throwable>> thrownTypes	De lijst met excepties is in ons geval leeg: null
CtBlock<B> body	De body van de methode bestaat voor ons uit een return statement en het veld waarvoor we de getter aanmaken

Tabel 1

```
public void setName(java.lang.String name) {
   this.name = name;
}
```

Listing 10

Code templates en testen

Het volledig opbouwen van Java code met behulp van Java code is veel werk. Daarom biedt Spoon de mogelijkheid om gebruik te maken van code templates. Een code template is een valide Java klasse en kan daarom gevalideerd worden door de Java compiler. Dit maakt het schrijven en onderhouden van templates minder foutgevoelig.

Om het schrijven van unit tests voor Spoon code te vereenvoudigen, biedt het Spoon framework een aantal assertions in de vorm van een Assert klasse. Zo zijn er assertions voor het vergelijken van Java files en voor het controleren van CtElements en CtPackage transformaties.

Conclusie

Spoon stelt je in staat om zelf analysetools te maken voor je Java code en om aan de slag te gaan met het genereren van Java code. Mocht je specifieke functionaliteit nodig hebben, waarvoor geen bestaande tools of Maven plug-ins beschikbaar zijn, dan kan Spoon een goede optie zijn. Hierbij is het voornamelijk zaak om het gebruikte metamodel goed door te krijgen. Spoon is een krachtig framework, waarmee veel mogelijk is op het gebied van analyse, transformatie en creatie van code. Er zijn echter niet heel veel voorbeelden beschikbaar. Ook het zoeken naar specifieke fouten of situatie met Spoon levert in Google (nog) weinig resultaten op. ■

```
   private void generateSetter(CtClass<?> ctClass, CtField ctField) {
      CtTypeReference voidReference = getFactory().Code().createCtTypeReference(void.class);

      CtParameter<?> parameter = getFactory().Core().createParameter();
      parameter.setType(ctField.getType());
      parameter.setSimpleName(ctField.getSimpleName());
      List<CtParameter<?>> parameterList = Arrays.asList(parameter);

      CtCodeSnippetStatement assignment =
    getFactory().Code().createCodeSnippetStatement("this."+ctField.getSimpleName()+" = "+ctField.getSimpleName());
      CtBlock body = getFactory().Code().createCtBlock(assignment);

      CtMethod setter = getFactory().Method().create(ctClass, new
   HashSet<ModifierKind> (Arrays.asList(ModifierKind.PUBLIC)),voidReference,
   "set"+capitalize(ctField.getSimpleName()), parameterList, null, body);
      ctClass.addMethod(setter);
   }
```

Listing 9