Metaebene – Fabric als Code Generation Framework für Java, C und C++

SASCHA SEIDEL

Fabric ist ein quelloffenes Code Generation Framework mit Unterstützung für Java, C und C++. Code schreiben, um Quelltext zu generieren. Was zunächst seltsam erscheint, macht in der Praxis durchaus Sinn. Vielfach erleichtern Codegeneratoren Entwicklern schon heute die tägliche Arbeit. Dieses Paper gibt Einblicke in die Architektur von Fabric und berichtet anhand einer Fallstudie vom praktischen Einsatz des Systems.

Allgemeine Einordnung: Fabric, Code Generation Framework, autom. Codeerzeugung

Zusätzliche Schlagworte: XML Schema, WSDL, Java, C/C++, GraphViz, DOT, EXI

1. EINFÜHRUNG

1.1 Codegeneratoren in der Praxis

Code schreiben, um Quelltext zu generieren. Was zunächst seltsam erscheint, macht in der Praxis durchaus Sinn. Schon heute erleichtern Codegeneratoren Entwicklern vielfach die Arbeit. Und dies nicht selten ohne Hintergedanke, können Computer doch viele Dinge besser als menschliche Entwickler: Sie arbeiten schneller, machen weniger Fehler und können speziell wiederkehrende Aufgaben effizienter lösen.

Freilich sind Codegeneratoren kein Allheilmittel und ebenso wenig für jeden Anwendungsfall geeignet. Dennoch kommt ihnen im Entwickleralltag eine zunehmend wichtigere Rolle zu. Geht es in einer Anwendung etwa darum, sie an ein Datebank-System anzuschließen, werden heute häufig sog. *objektrelationale Mapper* (ORM) eingesetzt. Diese schlagen eine Brücke zwischen dem zumeist SQL-sprechenden Datenbank-Server und der objekorientierten Welt der jeweiligen Programmiersprache (z.B. Java oder PHP). Existiert eine **formale Beschreibung der Daten**, die das System verarbeiten soll, so ist es für einen Codegenerator ein Leichtes, daraus Klassen für den objektorientierten Zugriff auf die Datenbank zu erzeugen.

Ähnliche Lösungen kommen im Bereich der verteilten Systeme zum Einsatz, etwa wenn Stub- und Skeleton-Klassen für Webdienste zu erstellen sind. Auch hier existiert mit der *Webservice Description Language* (WSDL) eine standardisierte Form der Schnittstellenbeschreibung. Diese ist maschinenlesbar und somit ebenfalls dazu geeignet, mit Codegeneratoren verarbeitet zu werden.

Sascha Seidel ist studierter Informatiker und arbeitet als selbstständiger Entwickler in Norddeutschland. Seine Forschungsinteressen liegen in den Bereichen Software Engineering, verteilte Systeme, Web-Entwicklung und Datenbank-Technologien.

E-Mail: fabric@nptech.de
Website: www.nptech.de/fabric

Begleitend zu Vortrag auf dem LinuxTag 2012, Berlin.

Weitere prominente Vertreter für Codegeneratoren finden sich in modernen Webframeworks wie Ruby on Rails, Django, Symfony, CakePHP oder Grails. Der hier unter dem Begriff *Scaffolding* bekannt gewordene Gerüstbau, also die automatische Erzeugung von CRUD-Formularen[1], ist letztendlich nichts Anderes als die Transformation einer formalen Datentypbeschreibung in Quelltext einer Zielsprache.

1.2 Grundidee des Fabric Frameworks

Das Fabric Framework wurde etwa im Jahr 2006 von Dr. Dennis Pfisterer am Institut für Telematik der Universität zu Lübeck ins Leben gerufen. Ursprünglich als Codegenerator für drahtlose Sensornetzwerke gedacht, lässt sich das System heute als universelles Code Generation Framework betrachten. Das heißt, es kann beispielsweise auch zur Generierung von handelsüblichen Java Dektop-Anwendungen genutzt werden und ist nicht mehr primär auf ressourcenbeschränkte Zielplattformen ausgerichtet.

Während die meisten Codegeneratoren in der Praxis noch immer domänen- oder anwendungsspezifisch sind, verfolgt Fabric einen anderen Weg. Das Projekt stellt zunächst ein allgemeines Framework zur Quellcode-Erzeugung bereit. Dabei ist es weder sprach- noch zweckgebunden. Die eigentliche Ausgabeform wird erst durch Module bestimmt, die in das System eingebunden und dann selektiv aufgerufen werden.

2. SYSTEMARCHITEKTUR

Das Fabric Framework lässt sich grob in zwei Bestandteile gliedern: Kern des gesamten Projekts ist eine Bibliothek zur dynamischen Erstellung von Quelltext. Sie wird nachfolgend als *Code Generation Framework* (CGF) bezeichnet. Das CGF lässt sich eigenständig nutzen, erlangt aber erst durch den Rest des Frameworks seine volle Mächtigkeit. Fabric bietet über die Kernbibliothek hinaus nämlich weitere Werkzeuge und Mechanismen an, die das automatische Generieren von Quellcode erleichtern. Zu ihnen zählt etwa eine ereignisorientierte *Modul-API*.

2.1 Code Generation Framework

Möchte man aus einem Programm heraus Quellcode erzeugen, erscheint dies zunächst als ein triviales Problem. Im weiteren Verlauf soll als Beispiel ein einfaches *Hello World!*-Programm erstellt werden. Listing 1 zeigt, wie dieses in der Programmiersprache Java aussehen würde.

Ein naiver Lösungsansatz könnte zunächst darin bestehen, den gesamten Quelltext in eine Zeichenkette zu schreiben, eine Datei auf der Festplatte anzulegen und den Code dann in diese auszugeben. Eine mögliche Umsetzung zeigt der Code in Listing 2.

[1] Create, Read, Update and Delete

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

Listing 1. *Hello World!*-Programm in Java

Bewertet man diese Herangehensweise, ist zunächst festzuhalten, dass sie zweifelsfrei praktikabel wäre. Tatsächlich erscheint der gewünschte Code in der Zieldatei, sodass sich diese kompilieren und ausführen lässt. Möchte man jedoch komplexere Programme erstellen, offenbart sich schnell der Nachteil dieser Lösung: Der generierte Quellcode verliert zur Erstellungszeit unmittelbar seine innere Struktur und wird zur *Black Box*.

Möchte man in der Klasse `HelloWorld` etwa neben der `main`-Methode noch eine weitere Funktion erzeugen, stößt man rasch an die Grenzen des Machbaren. Der `StringBuilder` gestattet zwar das Anhängen von Code und erlaubt mit geringem Aufwand auch das Voranstellen neuer Codezeilen, strukturelle Veränderungen an bereits eingefügtem Text sind aber schlecht umsetzbar.

```
1  FileWriter file = new FileWriter("HelloWorld.java");
2  StringBuilder code = new StringBuilder();
3
4  code
5    .append("public class HelloWorld {\n")
6    .append("\tpublic static void main(String[] args) {\n")
7    .append("\t\tSystem.out.println(\"Hello World!\");\n")
8    .append("\t}\n");
9    .append("}");
10
11 file.write(code.toString());
12 file.close();
```

Listing 2. Naiver Ansatz zur Code-Erzeugung

Hier verfolgt Fabric einen anderen Ansatz, der als *virtueller Code* bezeichnet wird. Die Grundidee dabei ist es, die Struktur des Codes so lange wie möglich zu erhalten. Das CGF bildet deshalb einen Großteil der Sprachelemente von Java, C und C++ auf eigene Java-Objekte ab. Möchte der Entwickler in seinem Code etwa eine neue Funktion erstellen, so instanziiert er dazu wahlweise eine `JMethod`, eine `CFun` oder eine `CppFun`. Eine geeignete Verzeigerung der Instanzen erhält auch komplexere Programmstrukturen.

Während der Laufzeit von Fabric wird sämtlicher virtueller Code in einem sog. *Workspace* abgelegt. Dabei handelt es sich, anders als etwa bei der Entwicklungsumgebung Eclipse, nicht um einen Ordner im Dateisystem, sondern ebenfalls um ein Objekt im Speicher.

Es dient Fabric zur Verwaltung aller erstellten virtuellen Dateien und ist in erster Linie als Gedankenkonstrukt zu verstehen. Da das Framework zur Laufzeit zunächst keinerlei Ausgaben auf die Festplatte schreibt, kann der virtuelle Code noch jederzeit nachträglich verändert werden. Erst am Schluss des Generatorlaufs werden alle Daten aus dem Workspace serialisiert und in Form von physischen, d.h. tatsächlich vorhandenen Dateien, in ein Ausgabeverzeichnis geschrieben.

Listing 3 zeigt, wie sich ein *Hello World!*-Programm mit Fabric erzeugen lässt.

```
1  // Create properties object
2  Properties prop = new Properties();
3  prop.put("fabric.output_directory", "target");
4
5  // Create Fabric workspace
6  Workspace workspace = new Workspace(prop);
7
8  // Get Java source file
9  JSourceFile file = workspace.getJava()
10   // Packet and class name
11   .getJSourceFile("de.linuxtag.hello", "HelloWorld");
12
13 // Create class
14 JClass helloWorldClass = JClass.factory.create("
       HelloWorld");
15
16 // Create main method
17 JParameter argsParam = JParameter.factory.create("String
       []", "args");
18 JMethodSignature jms = JMethodSignature.factory.create(
       argsParam);
19
20 JMethod mainMethod = JMethod.factory.create(
21   // Modifier, return type, method name and signature
22   JModifier.PUBLIC | JModifier.STATIC, "void", "main",
         jms);
23
24 // Append code to method body
25 mainMethod.getBody().appendSource(
26   "System.out.println(\"Hello World!\");");
27
28 // Add method to class and class to file
29 helloWorldClass.add(mainMethod);
30 file.add(helloWorldClass);
31
32 // Serialize code to disk
33 workspace.generate();
```

Listing 3. Code-Erzeugung mit Fabric

Der Code gliedert sich grob in drei Phasen:

(1) Erstellung und Konfiguration des Workspace
(2) Erzeugung des virtuellen Codes
(3) Verzeigerung und Herausschreiben

Obwohl der Code in Listing 3 dem naiven Ansatz gegenüber aufgebläht wirkt, bietet diese Variante dem Programmierer wesentliche Vorteile: Er ist bei Änderungen flexibler und hat jederzeit die volle Kontrolle über die Struktur seines Quelltextes. Ferner verschafft das Konzept des virtuellen Codes ihm die Möglichkeit, die Codeerzeugung besser in logische Schritte zu unterteilen.

Die Modul-API von Fabric greift auf einen zentralen Workspace zurück, den sich alle Module teilen. Dies gestattet eine bessere Strukturierung komplexer Codegeneratoren und erhält dennoch den Vorteil, dass alle Module in einem Programmdurchlauf auf der selben Eingabe operieren.

2.2 Treewalker und Modul-API

Um die Codeerzeugung weiter zu vereinfachen, bietet Fabric eine Reihe zusätzlicher Werkzeuge und Mechanismen. Abbildung 1 zeigt in groben Zügen die Architektur und das Verarbeitungsmodell des Frameworks.

Die graue Box mit der Beschriftung *Code Generation Framework* repräsentiert die vorgenannte Kernbibliothek zur Codeerzeugung. Die Bedeutung der verbleibenden farbigen Boxen wird im weiteren Verlauf näher erläutert.

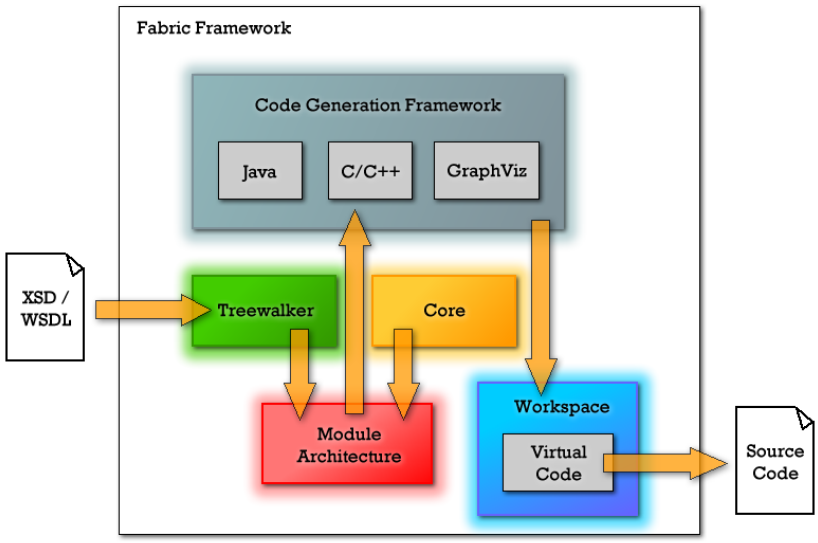

Abbildung 1. Architektur des Fabric Frameworks[2]

Linkerhand liest das System wahlweise ein XML Schema-Dokument oder eine WSDL-Datei ein. Diese enthalten eine formale Beschreibung dessen, was von Fabric generiert werden soll. Das Framework läuft die Eingabedatei anschließend mit einem sog. *Treewalker* ab und erzeugt beim Erreichen gewisser Schlüsselstellen innerhalb der XML-Baumstruktur verschiedene Ereignisse (z.B. Start des Dokuments, Start eines Elements, Ende eines Elements, Ende des Dokuments).

Module können auf diese Ereignisse reagieren und dann entsprechenden Code zur Behandlung aufrufen. Das Verarbeitungsmodell von Fabric ist *ereignisorientiert* und gleicht dem von SAX-Parsern her bekannten Verhalten. Die Module, allesamt Nachfahren einer Klasse `FabricDefaultHandler`, können dann unter Verwendung des eingangs bereits erwähnten Code Generation Framework beliebigen Quelltext generieren.

```
1  public class DemoModule implements FabricModule {
2    private Properties prop;
3
4    public DemoModule(Properties properties) {
5      this.prop = properties;
6    }
7
8    @Override
9    public String getName() {
10     return "demo1";
11   }
12
13   @Override
14   public String getDescription() {
15     return "Fabric demo module.";
16   }
17
18   @Override
19   public FabricSchemaTreeItemHandler getHandler(
20     Workspace workspace) throws Exception {
21       return new DemoHandler(workspace, this.prop);
22   }
23 }
```

Listing 4. Grundgerüst für ein Fabric Modul

Wie die Informationen aus der Eingabedatei dabei behandelt werden, ist dem Modulentwickler überlassen. Das Fabric Framework reicht alle Daten aus dem XML Schema zunächst unverändert an die Module weiter. Die Umsetzung der vorgegebenen XML-Standarddatentypen auf die jeweilige Zielprogrammiersprache (Java, C/C++ oder GraphViz) kann dadurch beispielsweise sehr individuell gestaltet werden.

[2]Bildquelle: `http://www.nptech.de/fabric`

Listing 4 zeigt den Grundaufbau eines jeden Fabric Moduls. Um es später verwenden zu können, muss es im Maven-Modul *Fabric::Core* bekanntgemacht werden. Erst wenn ein Modul dort registriert wurde, ist es auf der Kommandozeile aufrufbar. Über ein `Properties`-Objekt, das einfache Schlüssel-Werte-Paare enthält, lässt sich ein Fabric Modul bei Bedarf zusätzlich konfigurieren. Der wichtigste Bestandteil ist jedoch die Methode `getHandler()`, die eine Instanz der Klasse zurückgibt, mit der das Modul die Ereignisse des Treewalkers behandelt.

Der Code in Listing 5 zeigt einen Ausschnitt eines solchen Ereignisbehandlers. In den exemplarisch aufgeführten *Callbacks* kann das Modul jeweils auf einzelne Ereignisse reagieren. Über die Argumente der Methoden stehen je nach Ereignis zusätzliche Informationen (z.B. Elementname und -typ) zur Verfügung.

```
1  public class DemoHandler extends FabricDefaultHandler {
2    [...]
3
4    @Override
5    public void startSchema(FSchema schema) {
6      // Handle event...
7    }
8
9    @Override
10   public void endSchema(FSchema schema) {
11     // Handle event...
12   }
13
14   @Override
15   public void startTopLevelElement(FElement element) {
16     // Handle event...
17   }
18
19   @Override
20   public void endTopLevelElement(FElement element) {
21     // Handle event...
22   }
23
24   [...]
25 }
```

Listing 5. Ereignisbehandlung in abgeleiteter Handler-Klasse

Alle in Fabric registrierten Module teilen sich während der Laufzeit einen gemeinsamen Workspace und können an den selben Ausgabedateien Änderungen vornehmen. Das Framework lässt sich dazu auf der Kommandozeile mit einer Liste von Modulen (hier: `demo1` und `demo2`) aufrufen, die dann nacheinander ausgeführt werden.

Programmaufruf:

```
java -jar fabric.core-1.0.one-jar.jar
     ↪ -x /path/to/Schema.xsd -m demo1,demo2 -v
```

3. FALLSTUDIE: FABRIC IN DER PRAXIS

Im Jahr 2011 wurde an der Universität zu Lübeck (UzL) eine Fallstudie durchgeführt, die eine effiziente Binärserialisierung von Java- und C++-Objekten zum Ziel hatte. Dazu implementierte ein mehrköpfiges Entwicklerteam zwei verschiedene Fabric Module, die ausgehend von einer Datentypbeschreibung zunächst Container-Klassen und dann Code zur De-/Serialisierung generieren sollten.

Für die Kodierung kam dabei der erst im März 2011 verabschiedete W3C-Standard *Efficient XML Interchange*[3] (EXI) zum Einsatz. Dieses auf Grammatiken basierende Beschreibungsformat macht sich die unterschiedlichen Wahrscheinlichkeiten, mit denen einzelne Elemente in einem XML-Dokument vorkommen können, für eine besonders platzsparende binäre Serialisierung zu Nutze. Die wahrscheinlichste Alternative wird mit weniger Bits kodiert (sog. Entropiekodierung), was zu geringerem Bandbreitenbedarf bei der Netzwerkkommunikation führt.

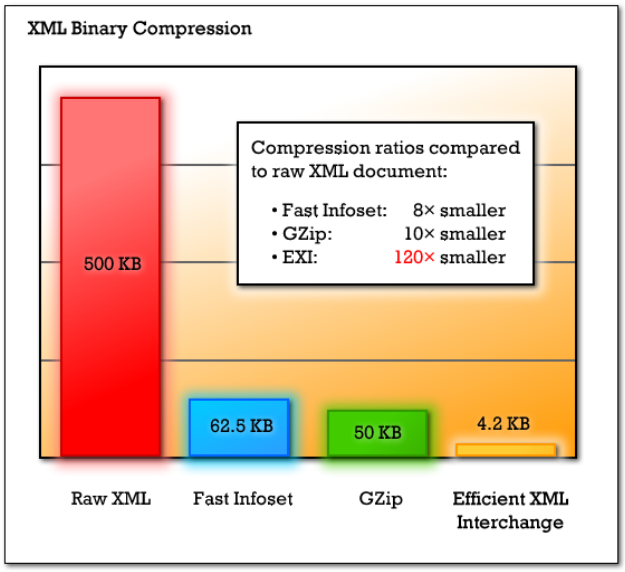

Abbildung 2. Binäre XML-Kompressionsverfahren im Vergleich[4]

Durch eine optionale Lauflängenkodierung können die sog. *EXI Event Codes* sowie die Nutzdaten bei Bedarf noch zusätzlich komprimiert werden. Wie Abbildung 2 zeigt, sind EXI-serialisierte Datenströme dadurch um bis zu 120-mal kleiner als XML-Dokumente im Rohformat.

[3] `http://www.w3.org/TR/exi`
[4] Bildquelle: `http://www.nptech.de/fabric`

Die Studie der UzL setzte Fabric zunächst dazu ein, um aus XML Schemata entsprechende Container-Klassen in Java und C++ zu erzeugen. Diese JavaBean-ähnlichen Konstrukte bestehen im Wesentlichen aus Klassen mit typisierten Attributen und zugehörigen Setter- und Getter-Methoden. Sie sollen von XML-Dokumenten abstrahieren und dem Anwendungsentwickler später den Umgang mit seinen Daten erleichtern. Mit Fokus auf ressourcenbeschränkte Sensornetzwerke galt das Credo, dass XML-Verarbeitung teuer ist und auf jeden Fall zu vermeiden sei.

Anschließend wurde ein weiteres Modul für Fabric entwickelt, das Code generiert, der zur EXI-Kodierung der zuvor erzeugten Bean-Klassen in der Lage ist. Auf der Java-Seite wurde dabei auf zwei bestehende EXI-Implementierungen (EXIficient[5] und OpenEXI[6]) zurückgegriffen. Für C++ wird hingegen die Implementierung einer eigenen EXI-Bibliothek angestrebt, die für jedes XML Schema dynamisch von Fabric generiert wird. Sie enthält nur Code, der aktuell auch tatsächlich benötigt wird. So soll den hohen technischen Ansprüchen Rechnung getragen werden, die Sensorknoten aufgrund ihrer beschränkten Hardware-Ressourcen stellen. Die Entwicklung des Fabric EXI-Moduls für C++ hält aktuell jedoch noch an.

4. ZUSAMMENFASSUNG

Das Fabric Framework wurde mit dem Ziel ins Leben gerufen, wiederkehrende Prozesse bei der Software-Entwicklung zu automatisieren. Aus dem akademischen Kontext erwachsen, eignet sich das Projekt heute auch zur Umsetzung von komplexer Generatorlogik. Dabei kombiniert das System Sprachunabhängigkeit mit einer leistungsfähigen Modul-Architektur. Der virtuelle Code der Kernbibliothek stellt alle Mittel zur Verfügung, um gängige Java-, C- und C++-Anwendungen dynamisch zu erzeugen.

Fabric erscheint unter der *Modified BSD-License* und ist offen für jedwede Unterstützung durch engagierte Open Source-Entwickler. Die Plattform bietet viel Raum für Erweiterungen, insbesondere bei der Sprachunterstützung im Code Generation Framework sowie bei der Implementierung neuer Module.

Interessierte Software-Architekten, Programmierer und Tester, die sich an der Weiterentwicklung von Fabric beteiligen möchten, finden auf der Website des Projekts[7] nähere Informationen und Kontaktdaten. Der Quellcode von Fabric steht über GitHub[8] frei zum Download bereit.

[5] `http://exificient.sourceforge.net`
[6] `http://openexi.sourceforge.net`
[7] `http://www.nptech.de/fabric`
[8] `https://github.com/nepa/fabric`