Metaebene

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

Sascha Seidel

www.nptech.de/fabric

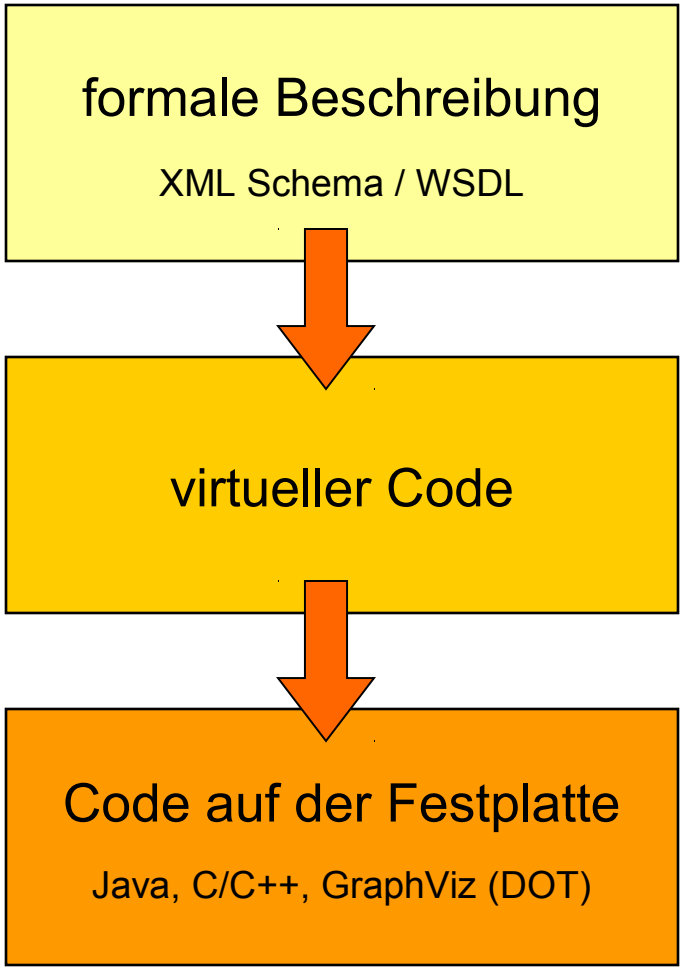
Agenda

- Grundidee des Projekts

- Software-Architektur
- Bibliothek zur Code-Erzeugung
- Treewalker und Modul-API

- Diskussion und Fragen

Grundidee des Projekts



Was soll generiert werden?

Wie wird es intern dargestellt?

Welche Ausgabe erhält man?

Fabric in Schlagworten

- **Framework** zur Code-Erzeugung:
  - Universell (Sprache & Zweck)
  - Datentyp-zentriert
- **Java**-Projekt
- **Maven**-Module
- **Open Source**
- Nischenprodukt

Ursprung von Fabric

UNIVERSITÄT ZU LÜBECK

ca. 2006: **Entwicklungsbeginn**

- Initiator: Dr.-Ing. Pfisterer, Institut für Telematik
- Universität zu Lübeck

Idee: Code-Generator für Sensornetzwerke

Freigabe unter *Modified BSD-License*

heute: universelles Code Generation Framework

Gründe für Code-Generatoren

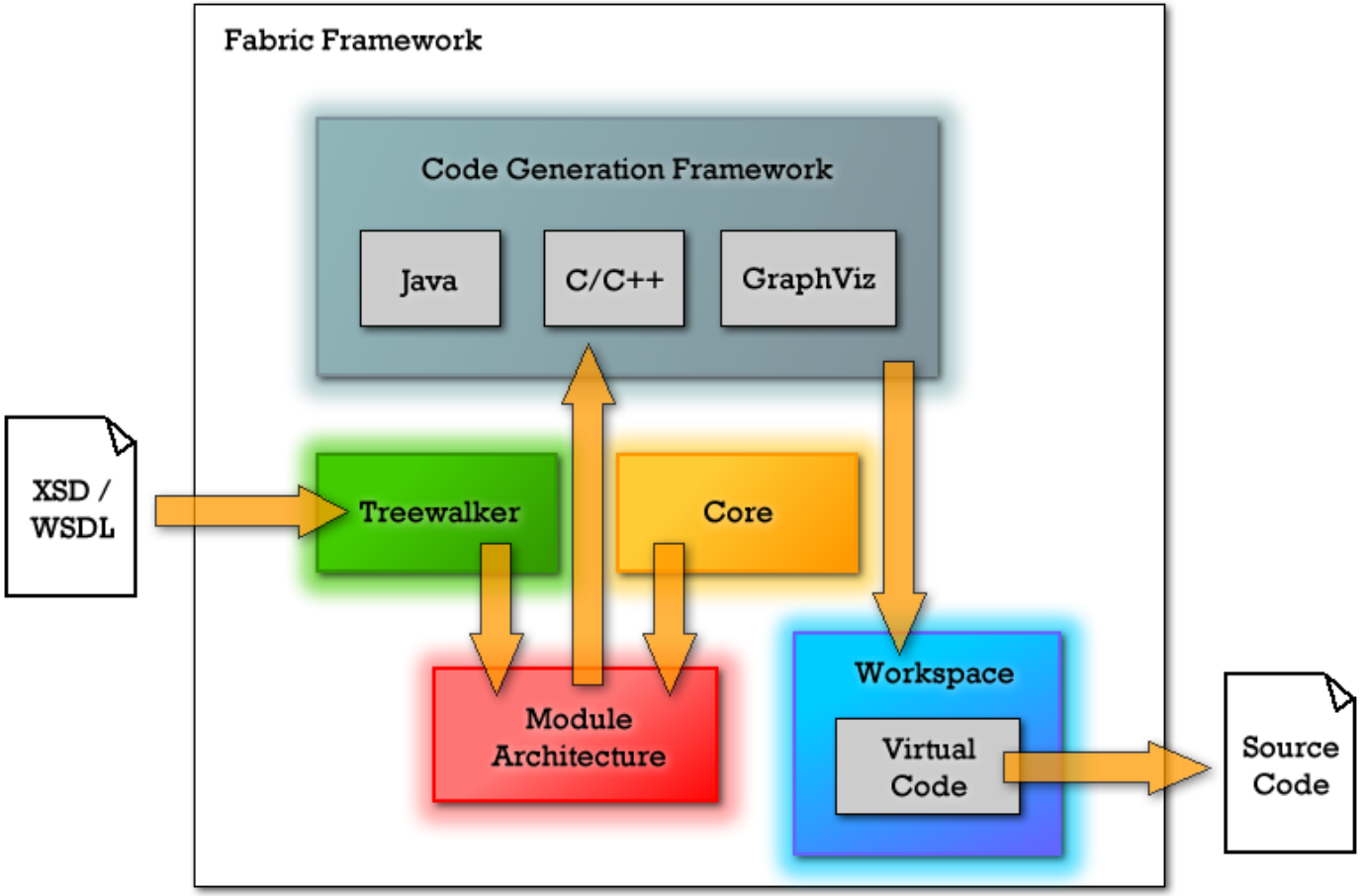
- **Automatisierung** von (Entwicklungs-)Prozessen
- Insbesondere Generierung von Code für datenorientierte Arbeitsabläufe:
  - Container-Klassen (*Beans*)
  - Netzwerk-Kommunikation
  - Verschlüsselung & Kompression
  - Datenbank-Anbindung
- Computer können diese Aufgaben **schneller** und besser lösen

Typische Vertreter aus der Praxis

- Objektrelationale Mapper für Datenbanken:
  - Java Persistence API
  - Propel (PHP)

- Schnittstellen für Middleware-Systeme:
  - JAX-WS für Webdienste
  - Google Protocol Buffers
- *Scaffolding* in Ruby on Rails

Software-Architektur



Beispiel: Hello World!

- **Ziel:** Generiere ein Hello World!-Programm
- In Java:

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

Naiver Ansatz

```
FileWriter file = new FileWriter("HelloWorld.java");
StringBuilder code = new StringBuilder();

code
  .append("public class HelloWorld {\n")
  .append("\tpublic static void main(String[] args) {\n")
  .append("\t\tSystem.out.println(\"Hello World!\");\n")
  .append("\t}\n");
  .append("}");

file.write(code.toString());
file.close();
```

Bewertung: naiver Ansatz

- Prinzipiell funktionsfähig
- **`StringBuilder`** kann intern „weitergereicht“ werden
- Anfügen von Code am Ende möglich

- **Aber:** Gezielter Zugriff auf String-Inhalte schwierig

➔ Struktur des Codes geht verloren
➔ Quelltext wird zur **Black Box**

Fabric verfolgt einen anderen Ansatz

- **Virtueller Code** im sog. **Workspace**
- Java-Objekte für alle Elemente der Zielsprache:
  - z.B. **`JMethod`**, **`CFun`** und **`CppFun`**
- Struktur bleibt bis zum *Write-Out* erhalten

➔ Änderungen sind jederzeit möglich
➔ Entwickler gewinnt **Kontrolle** zurück

Hello World! mit Fabric

```
// Create properties object
Properties prop = new Properties();
prop.put("fabric.output_directory", "target");

// Create Fabric workspace
Workspace workspace = new Workspace(prop);

// Get Java source file
JSourceFile file = workspace.getJava()
  // Package and file name
  .getJSourceFile("de.linuxtag.hello", "HelloWorld");

[...]
```

Hello World! mit Fabric (Forts.)

```
[...]

JClass helloWorldClass = JClass.factory.create("HelloWorld");

JParameter argsParam = JParameter.factory.create(
                                                "String[]", "args");
JMethodSignature jms =
  JMethodSignature.factory.create(argsParam);

JMethod mainMethod = JMethod.factory.create(
  // Modifier, return type, method name and signature
  JModifier.PUBLIC | JModifier.STATIC, "void", "main", jms);

[...]
```

Hello World! mit Fabric (Forts.)

```
[...]

// Append code to method body
mainMethod.getBody().appendSource(
  "System.out.println(\"Hello World!\");");

// Add method to class and class to file
helloWorldClass.add(mainMethod);
file.add(helloWorldClass);

// Serialize code to disk
workspace.generate();
```

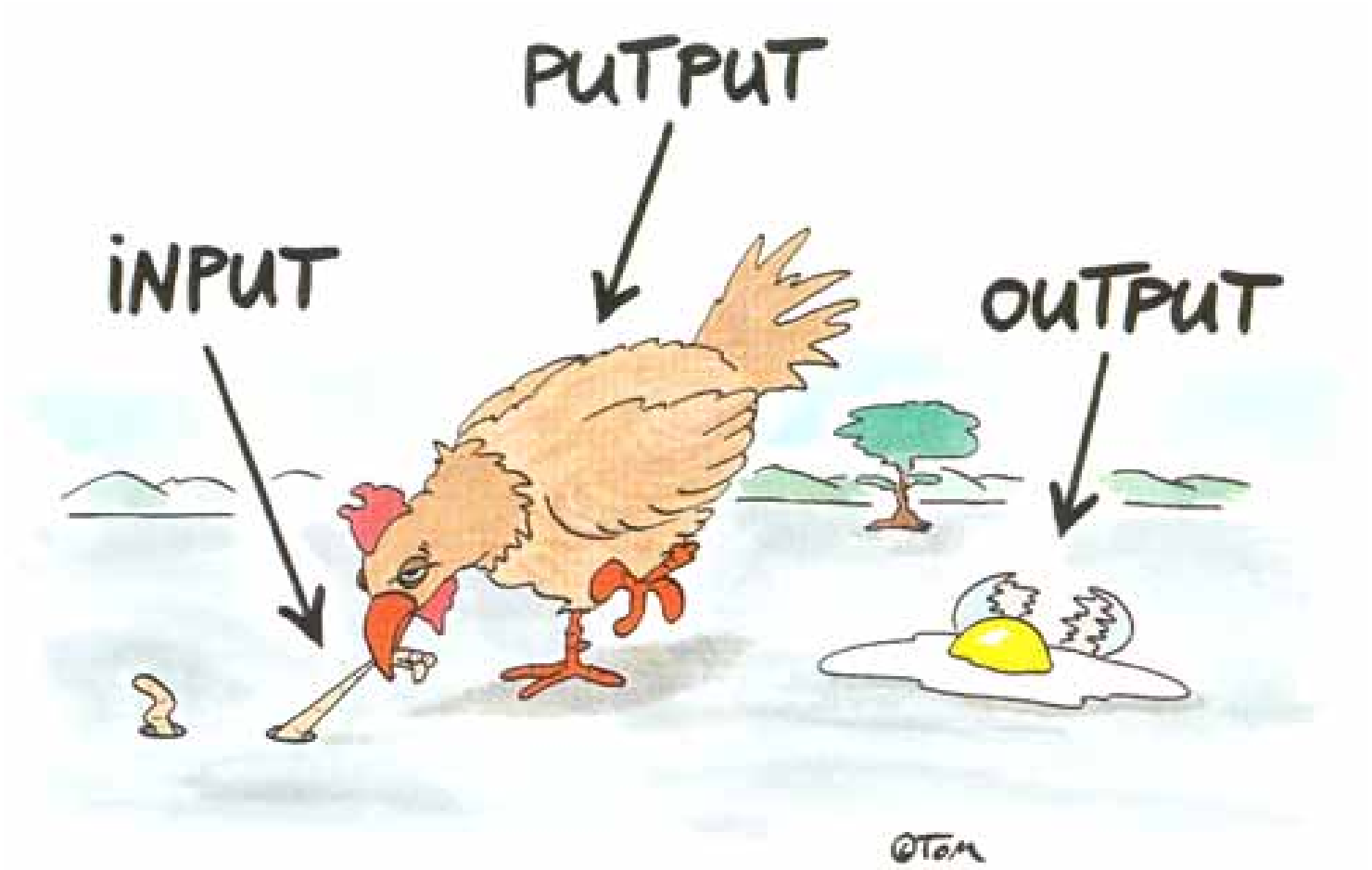
Bewertung: virtueller Code

- Lösung wirkt zunächst aufgebläht
- Bietet in der Praxis aber **mehr Flexibilität**

- **Objektorientierter Zugriff** auf Quelltext

  ➔ Änderungen und Erweiterungen einfacher

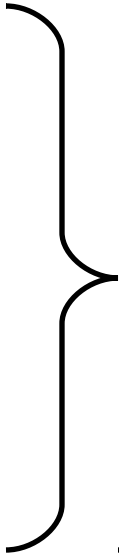
- Funktioniert für C/C++ und GraphViz analog


PUTPUT
INPUT
OUTPUT
©Tom

Eingabeformate

- **Voraussetzung:**
  Formale Beschreibung dessen, was generiert werden soll.

- **XML Schema**-Dokumente
  → beschreiben Daten
- **WSDL**-Dateien
  → beschreiben Dienste
  u. Schnittstellen



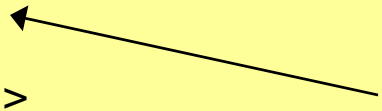
„universelle“
Interface Definition Language

Beispiel: XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ...>

  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Age" type="xs:int" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```



Namespace-Definitionen wurden weggelassen.

Treewalker

- Läuft Baumstruktur aus Eingabedatei ab
- **Ereignisorientierte Verarbeitung** (vgl. SAX Parser)

- Erreichen gewisser Schlüsselstellen löst entsprechende Ereignisse aus:
  - `startSchema`, `endSchema`
  - `startTopLevelElement`, `endTopLevelElement`
  - `startLocalElement`, `endLocalElement`
  - […]

Modul-Architektur

- Module können **auf Ereignisse reagieren**
- Fabric gibt Schnittstellen vor:
  - Moduldatei (**`FabricModule`**)
  - Ereignisbehandler (**`FabricDefaultHandler`**)
- **Sequentieller Aufruf** von Modulen möglich
- Module teilen sich einen **gem. Workspace**

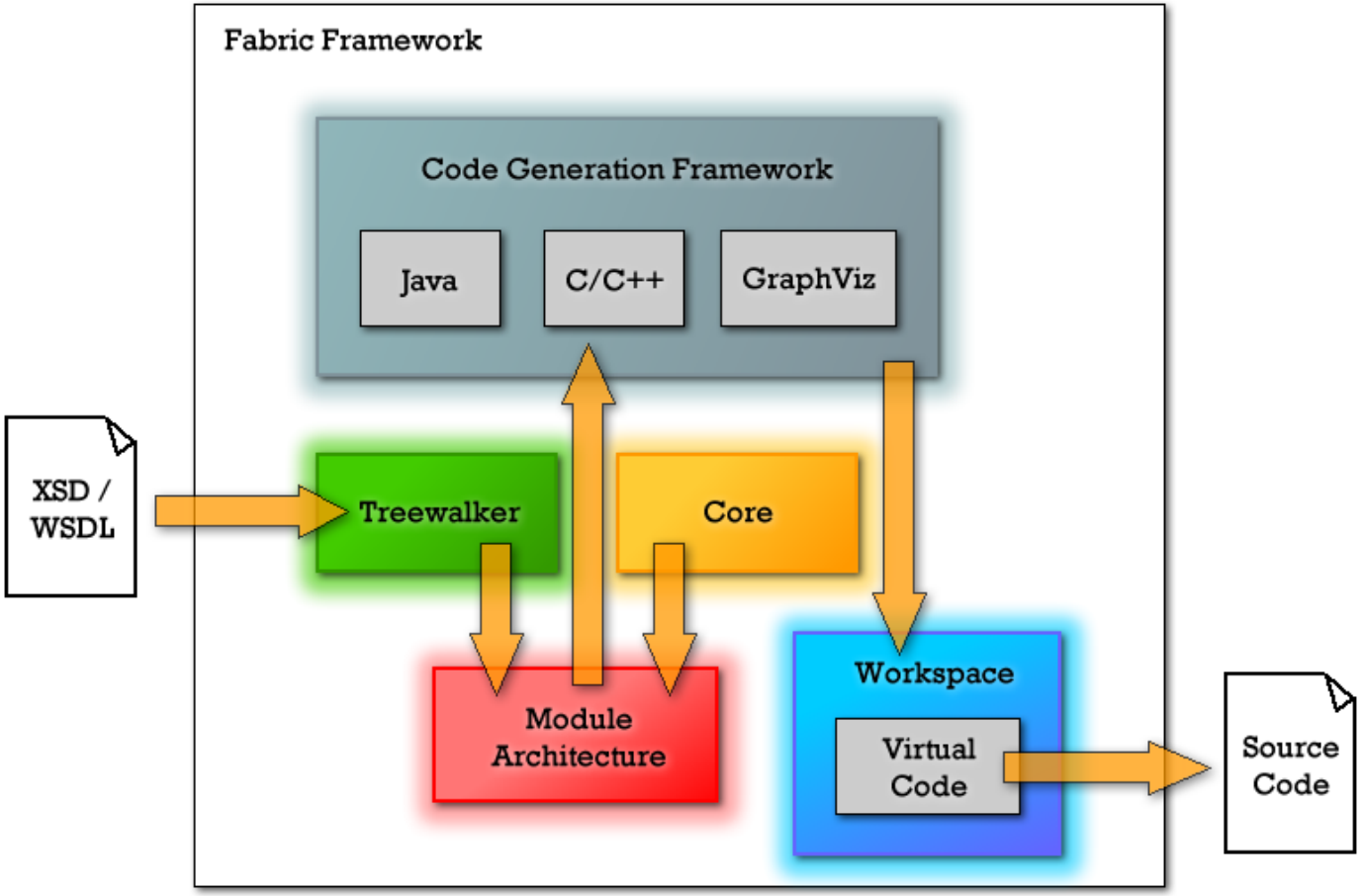
```
public class DemoHandler extends FabricDefaultHandler {
  @Override
  public void startSchema(FSchema schema) {
    // Handle event...
  }

  @Override
  public void endSchema(FSchema schema) {
    // Handle event...
  }

  @Override
  public void startTopLevelElement(FElement element) {
    // Handle event...
  }

  […]
}
```

Basisklasse gibt div. Methoden zur Ereignisbehandlung vor.


Fabric Framework
Code Generation Framework
Java
C/C++
GraphViz
XSD / WSDL
Treewalker
Core
Module Architecture
Workspace
Virtual Code
Source Code

Vielen Dank für Ihre Aufmerksamkeit.

Fragen?

Fabric Website:

`http://www.nptech.de/fabric`

`https://github.com/nepa/fabric`

Kontakt:

`Sascha Seidel`
`fabric@nptech.de`