GraalVM: Neue Serie S. 10

Funqy: Neues Serverless API in Quarkus S. 32

Warum spricht jeder über DDD? S. 74

JAVA Mag

Java™ magazin

Java | Architektur | Software-Innovation

JavaFX & GraalVM

In dubio pro Dukeo!

Brückenbauer in der Open-Source-Welt – Teil 3

Codegenerierung mit CobiGen

Etablierte Referenzarchitekturen wie devonfw machen es möglich, ein hohes Maß an Standardisierung über verschiedene Softwareprojekte hinweg zu etablieren. Starke Vorgaben, wie beispielweise Namenskonventionen für Quellcodeartefakte, bieten darüber hinaus eine klare Sichtbarkeit der Architektur auf der Quellcodeebene. Sind solche Vorgaben für (agile) Entwickler durchzuhalten und führt Schichtenbildung nicht teilweise zu nervigem Boilerplate-Code?

von Malte Brunnlieb

Genau hier hilft CobiGen und nimmt dem Entwickler die langweilige Arbeit ab, sorgt aber gleichzeitig für Struktur, Homogenität und Ordnung. So verbindet man eine wartbare und erweiterbare Architektur mit Effizienz in der Entwicklung.

Starke Codekonventionen gereichen nicht nur uns Entwicklern und Architekten zum Vorteil, sondern erlauben es auch, querschnittlich Werkzeuge zu etablieren, um diese Strukturinformationen zu verstehen, um Informationen aufzubereiten oder neue Artefakte automatisiert zu erstellen. Letzteres sollte über einen generativen Ansatz implementiert werden, der jedoch schnell auf kritisches Feedback trifft, wie zum Beispiel:

- Lohnt sich der Aufwand der Entwicklung?
- Wer wartet den Generator?
- Die Ergebnisse sind meist viel zu spezifisch oder auch generisch, als dass man sie einfach im Projekt benutzen und einen Mehrwert generieren kann.

Eine Welt voller Programmiersprachen und Technologiestacks

Nicht zuletzt im Kontext von devonfw merkt man schnell, dass die Open-Source-Welt in den letzten Jah-

ren eine Vielzahl an Programmiersprachen und Technologien auf den Markt gespült hat, derer kaum jemand Herr wird. Dabei bedient ein Großteil der Programmiersprachen oder Technologien bestimmte Nischen in der Softwareentwicklung und nur eine Handvoll von ihnen die breite Masse. Nichtsdestotrotz kommen gerade im Hinblick auf Technologien auch immer wieder Nischen- oder Alternativtechnologien in Softwareprojekten zum Einsatz, da jeder Kundenkontext mit seinen speziellen Anforderungen kommt. Schnell ist man dann nicht nur bei den oben aufgelisteten kritischen Fragen zu einem generativen Ansatz, sondern auch bei der essenziellen Frage angelangt, für welche Programmiersprache und welche Technologien wir den generativen Ansatz überhaupt implementieren wollen. Gerade im Hinblick auf devonfw mit seinem Fokus, mehrere Programmiersprachen und Technologiestacks auf Basis von Java, Node.js oder .NET zu standardisieren und dabei jedem Softwareprojekt als Anwender gerecht zu werden, wird ein Generator zur Mammutaufgabe.

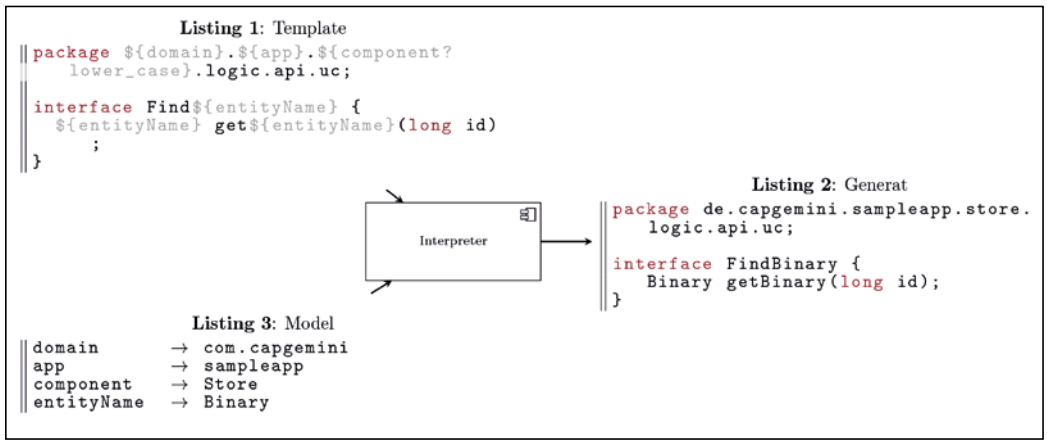

Abb. 1: Basiskonzept eines Text-Template-basierten Generators

Es gibt im Kontext von Generatoren zwei Kategorien: den sprach- und technologiespezifischen Generator, der bis hin zur syntaxsicheren Generierung häufig sehr zuverlässig und sicher das generiert, wofür er geschaffen wurde. Hierzu zählen beispielsweise Generatoren, die mittels des konkreten oder eines abstrakten Syntaxbaums einer spezifischen Programmiersprache Quellcode modellieren und serialisieren können. Zu der zweiten Kategorie gehören Generatoren, die grundsätzlich kein Verständnis über die zu generierende Zielsprache besitzen. Hierzu zählen u. a. Text-Template-basierte Generatoren wie Apache FreeMarker [1], Apache Velocity [2] und viele weitere. Die Idee bei ihnen ist die Spezifikation eines sogenannten Templates (**Abb. 1**), das mittels einer vordefinierten Sprache durch einen Interpreter interpretiert wird und Text als Generat erzeugt. Das Generat entbehrt dabei jeglicher Form und ist damit auch nicht auf eine Programmiersprache als garantiertes Ergebnis abbildbar.

Ich möchte jedoch der Vollständigkeit halber auch auf akademische Ansätze wie Repleo [3], SafeGen [4] sowie einen weiterführenderen Ansatz des hier präsentierten Generators CobiGen [5] verweisen, die syntaxsichere Generierung mit Bezug auf eine vordefinierte Zielsprache gewährleisten können. Wir werden uns im Rahmen dieses Artikels auf die einfachste von FreeMarker und Velocity bereitgestellte Form der Template-basierten Generierung konzentrieren, die auch in der Praxis ein höheres Maß an Flexibilität bieten als syntaxsichere Ansätze.

Ein Quellcodegenerator muss her

Nach ersten Anforderungsanalysen in verschiedenen Softwareentwicklungsprojekten wurde deutlich, dass ein Generator implementiert werden musste,

1. der verschiedene Zielsprachen unterstützt,
2. dessen Ergebnisse für Projekte einfach veränderbar sind,
3. dessen Generat ohne ein bestimmtes Framework oder den Generator selbst lauffähig ist (Stichwort Vendor Lock-in),
4. dessen Generat für den Entwickler editier- und veränderbar ist,
5. der gemäß der im Projekt vorherrschenden Architekturvorgaben und Namenskonventionen den Quellcode direkt in die nach den Vorgaben vorgesehenen Quellcodedateien generiert
6. der ggf. bereits vorhandenen (potenziell manipulierten) Quellcode mit neu generiertem Quellcode vereinen kann
7. der aus Quellcode Informationen für die Generierung weiteren Quellcodes ziehen kann, ohne dabei eine dedizierte Spezifikation wie ein Modell als Input zu benötigen

Durch die ersten beiden Anforderungen ist die Entscheidung einfach und ist auch in unserem Fall auf Text-Template-basierte Generierung gefallen, die es mit in der Industrie etablierten Interpretern wie FreeMarker und Velocity umzusetzen galt. Anforderung drei ist aus der Erfahrung verschiedener Softwareentwicklungsprojekte heraus ein wichtiger Punkt, um beim Kunden keinen Vendor Lock-in zu verursachen, der der Nutzung des Generators in 90 Prozent der Fälle im Weg stehen würde. Die Anforderungen fünf bis sechs basieren auf Erfahrungen von Entwicklern bei der vorherigen Nutzung anderer Generatoren. Hierbei sollten u. a. generierte und vor Editierung geschützte Quellcodebereiche vermieden werden, die die Flexibilität der Entwickler bei der Entwicklung spezieller Kundenwünsche zu stark beeinträchtigen kann. Außerdem soll der generierte Quellcode sich der Architektur und Namenskonventio-

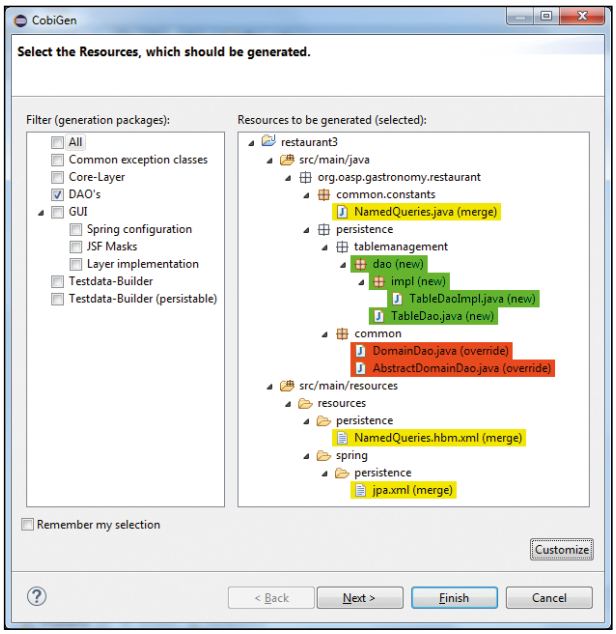

Abb. 2: Beispieldialog von CobiGens Eclipse-Integration zur Auswahl der zu generierenden Inkremente (links) mit Datenvorschau (rechts)

nen unterordnen und nicht als Nebenprodukt außerhalb des manuell editierten Quellcodes existieren, um keinen mentalen Bruch durch technische Unzulänglichkeiten in der Organisation von fachlichem Quellcode zu verursachen. Schlussendlich kommen gerade mit dieser letzten Anforderung jedoch automatisch die Fragen nach der erneuten Generierung von Quellcode und die Frage der Integration von manuell geschriebenem und generiertem Quellcode auf. Anforderung sieben basiert auf der Erfahrung, dass die Entwicklung von Modellen je nach Mächtigkeit der Modellierungssprache letztlich meist nicht mehr weit vom Quellcode entfernt ist und dennoch eine weitere zu erlernende Sprache darstellt. Daher haben wir uns gerade für den Einsatz in kleinen Softwareentwicklungsprojekten gegen einen modellgetriebenen Generierungsansatz entschieden, der den Rahmen vieler Projekte sprengen würde.

Auf Basis eines Text-Template-basierten Interpreters wurde zur Adressierung aller Anforderungen der codebasierte, inkrementelle Generator (CobiGen) [6] im Rahmen der devonfw Open Source Platform [7] implementiert. CobiGen versteht sich als Framework zur Erstellung eigener Generatoren, wobei die direkt ausführbare Version von CobiGen bereits ein voll funktionstüchtiger Quellcodegenerator für die Standardanwendungsfälle von devonfw ist. Im Folgenden soll ein Anwendungsfall exemplarisch dargestellt werden.

Entwicklerfokus

CobiGen basiert auf der Annahme, dass der Entwickler seine Software immer in kleinen Inkrementen implementiert. Die wenigsten Entwickler werden vereinzelt Quellcode editieren, ohne dabei zwischenzeitlich einen kompilierbaren oder testbaren Stand zu erhalten. CobiGen spricht daher in der Benutzerführung von Inkrementen. Ein Inkrement ist technisch eine Menge von Templates. Am Beispiel von **Abbildung 2** sieht man, dass sich die Inkremente auf strukturelle Quellcodeartefakte beziehen, die durch eine Referenzarchitektur wie devonfw geprägt sein können. Hier am Beispielinkrement „DAOs“, das in diesem Beispiel Datenzugriffsobjekte repräsentiert. Die durch das Inkrement zu generierenden Java-Klassen und XML-Dateien in diesem Beispiel sieht man auf der rechten Seite des Dialogs. Außerdem sind die Dateien in der Vorschau mit Farben und Kommentaren (in Klammern hinter dem Dateinamen) markiert. Diese Informationen betreffen bereits die Indikation des Verhaltens zur finalen Integration des Quellcodes in die potenziell bereits vorhandene Quellcodedatei. Es werden in der Vorschau lediglich drei unterschiedliche Indikationen gegeben:

- New – eine noch nicht vorhandene Datei wird generiert/erzeugt
- Override – eine bereits existierende Datei wird mit den neu generierten Inhalten überschrieben
- Merge – die Inhalte einer bereits existierenden Datei werden strukturell mit den neu generierten Inhalten zusammengeführt

Zusammen mit dem Zielpfad wird auch das Verhalten der Zusammenführung pro Template innerhalb der Konfiguration zu jedem Template festgelegt. Alle Templates und ihre Konfigurationen können dabei jederzeit vom Entwickler geändert und auf die eigenen oder Projektbedürfnisse angepasst werden. Die Zusammenführungsstrategien zur meist strukturellen Zusammenführung von Quellcode sind hierbei zwar aus einer Menge bereits existierender Implementierungen auswählbar, gegeben durch Erweiterungen des CobiGen Frameworks. Eine einfache

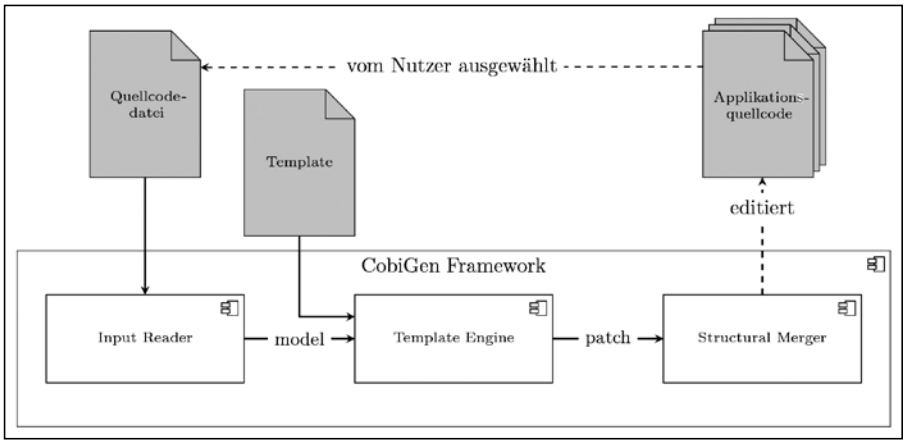

Abb. 3: Skizzierung der Komponenten sowie der Ein- und Ausgabedaten des CobiGen Frameworks

Modifizierung der Algorithmen ist jedoch zum Zeitpunkt der Templateentwicklung und Konfiguration nicht möglich.

Die zur Generierung und Zusammenführung von Quellcode benötigten Komponenten des CobiGen Frameworks sowie deren Ein- und Ausgabedaten sind in **Abbildung 3** dargestellt. Das CobiGen Framework lässt sich grob in drei Komponenten unterteilen. Einen sogenannten Input Reader, der am Beispiel der Eclipse-Integration eine vom Nutzer ausgewählte Datei mit Quellcode als Eingabe erhält und sie in ein Modell transformiert, im Fall des Java-Beispiels in einen abstrakten Syntaxbaum. Mittels dieses Modells und des ausgewählten Inkrements wird die Template-Engine aufgerufen, um mit dem Modell die gegebene Menge an Templates zu interpretieren und den daraus resultierenden Text (Patch) an die Structural-Merger-Komponente weiterzugeben. Diese Komponente übernimmt die Zusammenführung der generierten Patches mit Hilfe der Zielpfade aus der Konfiguration der Templates im Quellcode der Zielanwendung.

Dir gehört der Quellcode

Die Generierung von Quellcode direkt in die Quellcodestruktur der vorgegebenen Architektur hat sich in der Praxis als Killerfeature erwiesen. Durch die gerechte Anpassung der Templates an das Zielbild und die Fähigkeit von CobiGen, strukturell Quellcode zusammenzuführen, werden Entwickler durch kleine, während der Entwicklungszeit generierbare Inkremente in ihrer Arbeit unterstützt. Lästiges Anpassen von generiertem Code zur Einpassung in die Zielarchitektur oder Konventionen wird verhindert.

Das strukturelle Zusammenführen von Java-Quellcode sei als ein Beispiel für eine bessere Vorstellung des Zusammenführens genannt. In der Implementierung des Java Mergers wird die Zusammenführung von Quellcode auf Basis ganzer Methoden und Feldern von Klassen definiert. Hierbei werden die Signaturen von Feldern und Methoden verglichen und im Fall einer gleichen Signatur aufgrund einer Konfiguration entschieden, ob das Feld oder die Methode mit dem Patch überschrieben wird oder unverändert bleibt. Es werden dann je nach Konfiguration bei einem Konflikt zwischen Patch und existierendem Code innerhalb einer Klasse entweder der existierende Quellcode bspw. einer Methode bevorzugt oder der Quellcode des Patches. Auch auf Annotationen oder die Vererbung von Interfaces wirkt sich die Zusammenführung gleichermaßen aus, indem neue Interfaces zusätzlich vererbt werden oder nicht vorhandene Annotationen erstellt oder um weitere Eigenschaften angereichert werden können.

Durch CobiGen generierbare Quellcodeartefakte haben außerdem gezeigt, dass gerade in einem Juniorenteam weniger Fehler gemacht werden, da Basisartefakte wie CRUD (Create, Read, Update, Delete) Services, Datenbankanbindungen oder Logikimplementierungen bereits standardisiert generierbar sind. Auch Namenskonventionen können einfach etabliert werden, wenn ein Generator wie CobiGen bereits vormacht, wie der Quellcode strukturiert wird. Hier kommt wiederum dem Entwickler zugute, dass der generierte Quellcode direkt zur Hand des Entwicklers generiert wird und nicht isoliert als Fremdkörper existiert.

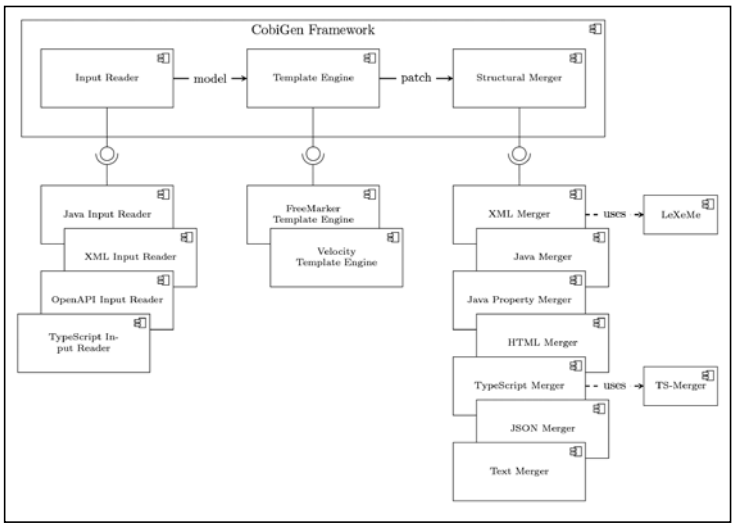

Abb. 4: CobiGen-Erweiterungen

Flexibilität und Erweiterbarkeit

Wie bereits angerissen, wird jedoch eine etwas breitere Kompatibilität bzgl. des Input Readers sowie eines strukturellen Mergers benötigt, um verschiedene Quellcodeartefakte einlesen oder verschiedene Zielsprachen sinnvoll zusammenführen zu können. Aus diesem Grund sind die in **Abbildung 3** dargestellten Komponenten lediglich Fassaden, die einen Erweiterungsmechanismus implementieren. In Wirklichkeit werden verschiedene lose gekoppelte Plug-ins nach ihrer Fähigkeit, eine angefragte Eingabe einzulesen oder eine resultierende Zieldatei zusammenzuführen, angefragt. CobiGen realisiert mittlerweile einen bunten Blumenstrauß an Plug-ins, die **Abbildung 4** zu entnehmen sind.

Nicht zuletzt wird es durch diese Erweiterbarkeit Projekten leicht gemacht, eigene spezielle Input Reader oder strukturelle Merger-Erweiterungen zu implementieren, die ganz spezielle Wünsche eines Projekts realisieren, und dabei weiterhin auf existierende strukturelle Merger-Implementierungen zurückgreifen zu können. So könnten beispielsweise neue Eingaben (wie projektspezifische Modelle) oder einfache Formate (wie Excel-Dateien) eingelesen werden. Die vorhandene Flexibilität der verschiedenen Plug-ins für die sprachspezifische Zusammenführung von Quellcode oder die Anbindung verschiedener Template-Engines bleibt dabei durch CobiGen weiterhin nutzbar.

Fazit und Ausblick

CobiGen erfreut sich neuer Beliebtheit als Standardwerkzeug, um ganze CRUD-Server und Clientimplementierungen aus dem Datenmodell oder einer OpenAPI-Service-Spezifikation zu generieren. Oder aber

auch als Framework für die Entwicklung projektspezifischer Anwendungsfälle. Insbesondere für Letzteres hat CobiGen bereits einen großen Mehrwert gebracht, da sich die Generatorentwicklung selbst für kleine Projekte wieder lohnen kann. Meist müssen nur eigene Templates geschrieben werden, um neue projektspezifische Generate zu erhalten.

Durch den generischen templategetriebenen Generierungsansatz haben heute bereits verschiedene Projekte CobiGen in devonfw-fremden Kontexten wie im Kontext der Referenzarchitektur RegisterFactory oder kundenspezifischen Architekturen und Frameworks nutzen können. Es hat sich gezeigt, dass der Ansatz mit überschaubarem Aufwand solide auf spezifische Architekturanforderungen skalierbar ist. Ein weiteres Beispiel ist ein Projekt, das datengetriebene Dialoge komplett aus einer Excel-Spezifikation über einen eigens dafür geschriebenen Input Reader in JSF und JEE generiert und dabei Nutznießer bereits existierender Merger-Implementierungen geworden ist.

Nebst der hier dargestellten Integration in die Entwicklungsumgebung Eclipse bietet CobiGen auch ein Kommandozeileninterface (CLI) sowie eine Maven-Integration zur automatischen Generierung von Artefakten während des Build-Prozesses an. Gerade durch das CLI ist CobiGen jetzt auch für Entwickler zugänglich geworden, die andere IDEs als Eclipse, etwa IntelliJ oder VS Code, nutzen.

Malte Brunnlieb ist seit 2012 für Capgemini als Architekt und Berater tätig. Seine Passion liegt in der Standardisierung von Architekturen und der Automatisierung der Softwareentwicklung von Custom Solutions.

Links & Literatur

[1] https://freemarker.apache.org

[2] http://velocity.apache.org

[3] https://www.researchgate.net/publication/221108687_Repleo_A_syntax-safe_template_engine

[4] https://link.springer.com/chapter/10.1007/11561347_21

[5] Brunnlieb, Malte: „Source Code Transformation Based on Architecture Implementation Patterns“; Verlag Dr. Hut, 2019

[6] https://devonfw.com/website/pages/docs/master-cobigen.asciidoc_document-description.html#Guide-to-the-Reader.asciidoc

[7] https://devonfw.com