Paarlauf

Von der Entwicklung zur Beschreibung – MDSD und JEE bei der Commerzbank

Frank Jarsch

Die Commerzbank sieht in einem ausgewogenen Verhältnis von Generierungsansätzen wie modellgetriebener Softwareentwicklung (MDSD, englisch Model-Driven Software Development) und individuell erstelltem Code deutliche Effizienzsteigerungen. Neben der gesteigerten Effizienz werden eine ganze Reihe weiterer positiver Ziele erreicht: Da das zugrunde liegende Commerzbank-Java-Standard-Framework allen internen Architekturvorgaben und Code-Konventionen entspricht, gilt dies auch implizit für Anwendungen, die auf dieser Basis generiert werden. Unternehmen, die bei der Code-Generierung klug vorgehen und mit den Risiken offen umgehen, können die Qualität und die Wartbarkeit der Anwendung durch eine standardisierte Umsetzung der Anforderungen signifikant erhöhen.

Motivation

Die Commerzbank beschreitet bei der Entwicklung von JEE-Anwendungen neue Wege. Bei der Etablierung eines konzernübergreifenden Personen-Stammdatensystems werden große Teile der webbasierten Frontends nicht mehr individuell von der IT programmiert, sondern von den Fach-Kollegen im Projekt in einer eigens für sie entwickelten Sprache beschrieben. Zum Einsatz kommt eine sogenannte DSL (Domain-Specific Language), eine fachspezifische Beschreibungssprache, in der das Web-Frontend, die Eingabemasken und die zu bearbeitenden Dateninhalte festgelegt werden. Anschließend wird diese Beschreibung „übersetzt" und „echter" Java-Quellcode durch Code-Generatoren generiert. Die erste Version der Anwendung ging Mitte Juni 2013 in Produktion.

Mit MDSD etablierte Technologiestandards generieren

Bei der Entwicklung webbasierter Intranet-Anwendungen in Unternehmen wird oftmals eine Vielzahl von gleichartigen Dialogen und Masken erstellt, mit denen Standard-Verarbeitungsabläufe wie beispielsweise Erfassung, Anzeigen, Änderung, Löschung von Daten unterstützt werden. Ungeachtet dieser gleichartigen Anforderungen an Struktur, Layout und Workflow der jeweiligen Dialoge werden Anwendungen – oft auf Basis interner Standardarchitekturen und Frameworks – nach wie vor individuell entwickelt. Effizienter ist es, den erreichten Standardisierungsgrad verfügbarer Technologieframeworks direkt in die Entwicklung zu integrieren, um sich im Idealfall nur auf die Entwicklung der Geschäftslogik konzentrieren zu können.

Möglich wird dies, wenn Konzepte und Technologien des Model-Driven Software Development (MDSD, deutsch: modellgetriebene Softwareentwicklung) auf bestehende Technologieframeworks angewendet werden. Dadurch lassen sich Vorteile der internen Standardisierung und Model-Driven Development miteinander verknüpfen, um weitere Synergien zu heben. Dafür müssen die individuellen Anforderungen mittels einer Beschreibungssprache beschrieben werden können, um die dazu passende, konkrete Projekt-Anwendungsarchitektur mitsamt individueller Ausprägungsformen zu generieren.

Verschiedene Beschreibungssprachen für verschiedene Aufgaben

Wichtigste Start-Voraussetzung ist daher die Erkenntnis, dass Code-Generierung nur für Domänen infrage kommt, für die sich ein sinnvolles Modell abstrahieren lässt. Und selbstverständlich muss jeder generative Ansatz auch individuelle Erweiterungen durch klassische Codierung ermöglichen, denn Modelle werden die Realität selten hinreichend beschreiben können.

Die gesamte Topologie einer Anwendungsarchitektur mittels Metamodell zu beschreiben, ist eine ebenso abstrakte wie komplexe Aufgabenstellung. Deutlich weniger abstrakt werden Modelle, wenn sie so konkret wie möglich (aber so abstrakt wie nötig) an der jeweiligen Domäne bleiben, die dadurch beschrieben werden soll. Und was spricht dagegen, verschiedene Anwendungsbereiche – beispielsweise Frontend und Backend – als verschiedene Domänen zu verstehen und somit mehrere Modelle abzuleiten?

Wir betrachten somit zwei verschiedene Ansätze für die tragenden Bereiche einer JEE-Anwendungsarchitektur, die über hohes Effizienzpotenzial verfügen:

- Die Präsentationsschicht umfasst sowohl den gesamten Mix der eingesetzten Frontend- und Layout-Technologien für die Maskenerstellung als auch das View-Modell, indem die Daten gehalten werden, die für die Anzeige und Bearbeitung auf diesen Masken benötigt werden.
- Die Schichten für Geschäftslogik und Datenzugriff (DAOs): Java-Frameworks, die über saubere Schichtentrennung so-

wie Transaktionsmanagement verfügen und JPA als Basis-Standard mit entsprechenden Referenzimplementierungen integrieren, verfügen über eine hohe Standardisierung, die lediglich erweitert werden muss, indem die Definition von Entitäten mit einer Beschreibungssprache ermöglicht wird.

Webbasierte Frontends, die bisher individuell von der IT programmiert werden, könnten zukünftig in einer eigens entwickelten *Dialog-Beschreibungssprache* als Domain-Specific Language (DSL) beschrieben werden. Eine Beschreibung würde das benötigte Know-how eines Java-Entwicklers – alle eingesetzten Frontend-Technologien – deutlich reduzieren. Es reichte für Standardanforderungen, die Beschreibungssprache anwenden zu können.

Diese Idee ist nicht neu. Neuartig für die Commerzbank war es, den MDSD-Ansatz mit dem Potenzial der eigenen Technologiestandards zusammenzuführen. Die DSL kann individuell und domänenspezifisch entwickelt und erweitert werden und sollte somit unternehmenswichtige Anforderungen in Bezug auf Funktionalität und Standardisierung repräsentieren. Der auf dieser Basis generierte Quellcode entspricht dadurch den Unternehmensstandards für Anwendungs-Frontends.

Für die Beschreibung von Masken und Geschäftslogik bieten sich separate Beschreibungssprachen an. Eine Dialog-DSL soll es ermöglichen, Masken zu beschreiben; eine Entitäten-DSL soll es ermöglichen, fachliche Entitäten zu spezifizieren und somit das Generieren aller Klassen der Geschäftslogik auf Entitätenbasis ermöglichen.

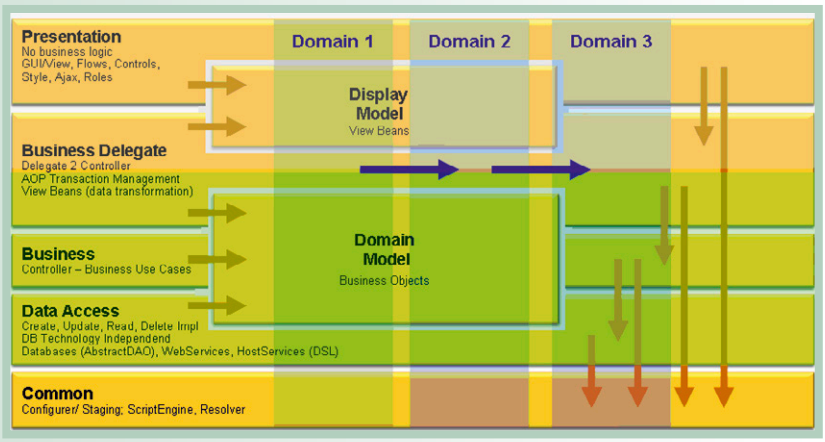

Abb. 1: Verschiedene Spezifika der Anwendungsarchitektur werden durch verschiedene DSLs abgedeckt

Auf Basis der Dialog-DSL wird später die Präsentationsschicht generiert; auf Basis einer Entity-DSL soll später das Backend generiert werden. Beide Generate (Quellcode) müssen in mehrschichtigen Anwendungsarchitekturen nahtlos ineinandergreifen.

Das Modell und die Sprache

Eine große Herausforderung ist die Entwicklung einer deterministisch sauberen und zugleich intuitiven Sprache, mit der Benutzeroberflächen umfassend beschrieben werden können. Die Grammatikstandards sollten intensiv und zielgruppenbezogen abgestimmt werden, denn die DSL, die dabei entsteht, wird ein langfristiges Medium für die Softwareentwicklung im Unternehmen werden und entsprechende Akzeptanz benötigen.

Basis der Sprache ist das Domänenmodell als Metamodell. Die Domäne *FrontEnds* ist eine technische Domäne, zudem handelt es sich um eine gut bekannte und für jedermann nachvollziehbare Domäne. Hierzu kann man sich an umfassenden Spezifikationen, wie der Interaction Flow Modeling Language (IFML) der OMG, oder konkreteren (und gerne auch einfacheren) existierenden Programmiersprachen orientieren. Es gilt, einen vernünftigen Kompromiss zwischen Abstraktion und Konkretisierung zu finden. Je konkreter die Sprache definiert ist, desto effektiver wird der Generator, der am Ende die Anwendung erzeugt. Je mehr Vorgaben durch Framework-Standards bereits gesetzt sind, desto weniger Parameter sind nötig, um Masken zu erzeugen.

Wer eine DSL für die Maskengenerierung verschiedener Projekte ermöglichen will, muss übergreifend denken und im Anforderungsfokus Aller bleiben. Konkreter, spezifischer Quellcode, der nur für ein einzelnes konkretes Projekt hilfreich ist, wird weiterhin individuell programmiert. Ein minimalistischer, übergreifender Anwendungsfall ist CRUD – Create, Read, Update, Delete. Für den Anwendungsfall Präsentationsschicht müssen zuvor im Metamodell diese vier CRUD-Operationen definiert werden, damit dem Anwender der DSL für die Maskenbeschreibung die dafür erforderliche Sprachgrammatik und Schlüsselwörter verfügbar sind.

Ähnliches gilt für den Anwendungsfall Entitäten-DSL. Hier müssen auf Metaebene Attribute, wie Name, Felder und Primärschlüssel, spezifiziert werden, damit diese später in der DSL als Syntax zur Verfügung stehen.

Eine textuelle DSL programmieren

Erstellt werden können solche textuellen DSLs mit dem Open-Source-Framework Xtext. Xtext basiert auf dem Eclipse Modeling Framework (EMF) und setzt damit auf dem Ecore-Objektmodell auf. Dies ermöglicht die Definition von Sprachgrammatiken als Ableitung eines zuvor spezifizierten Metamodells.

Xtext stellt nicht nur einen Parser für die selbst realisierte Syntax zur Verfügung, sondern stellt als Ergebnis des Parsings einen abstrakten Syntaxbaum mit Java-Klassenmodell bereit. Dies erlaubt den flexiblen programmierten Zugriff für spätere Generierungszwecke. Darüber hinaus wird ein Eclipse-Editor generiert und stellt dem DSL-Anwender – wie bei anderen Programmiersprachen – ein Werkzeug mit allen

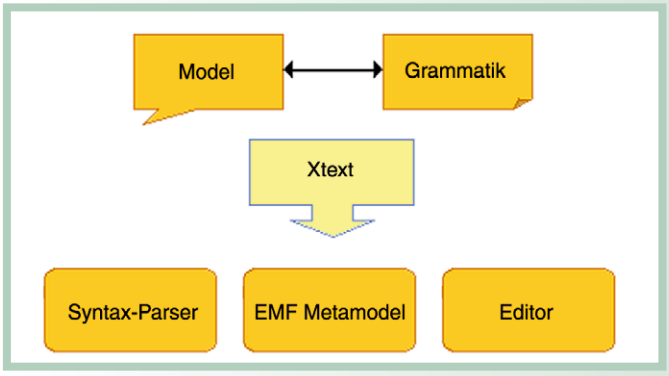

Abb. 2: Xtext stellt Parser, Objektmodell und Editor für MDSD bereit

üblichen Eclipse-Features zur Verfügung. In unserem Anwendungsfall bedeutet dies, der Anwender editiert („programmiert“) in einem eigenen Dateityp (z. B. *.maske) mittels DSL die Maske und nutzt Codevervollständigung, Attribut- und Methodenauswahl-Listen.

Steht das Metamodell – eine abstrakte Masken- oder Entitätenbeschreibung –, wird im nächsten Schritt die Grammatik definiert. Ein Modell für Dialoge wird, neben dem primären CRUD-Funktionsumfang, sicher immer die klassischen Eingabe-Controls (auch hier wahlweise „Inputfield“ oder „Eingabefeld“) und Labels umfassen, aber auch deren Gruppierungen, Tab-Register, Frames oder Sektionen, Navigationen und ein Berechtigungsattribut für diese Elemente.

Auf Basis dieser Grammatikdefinition generiert Xtext den dazu passenden Syntax-Parser, der aus der beschriebenen DSL-Syntax das Metamodell *inferiert*. Hierunter versteht man die Ableitung des kompletten, konkreten Objektbaums aus dem EMF-Metamodell, faktisch alle referenzierten Abhängigkeiten der selbst definierten DSL-Typen. Hieraus generiert Xtext einen eigenen Eclipse-Editor. Individuelle Erweiterungen dieses Objektmodells außerhalb der Typendefinition der DSL sind durch Xtext-Interfaces möglich und ermöglichen funktionale Hooks, mit denen die inferierte Objektstruktur (Xtext-intern injiziert mittels Google Guice) angereichert werden kann.

```
PersonAnlegen.maske

Verwende Modul mdsdDemo.*

Maske PersonAnlegen "Person anlegen" erzeugt Person ps;
  Bereich anlage "Personen-Anlage":
    Eingabefeld name: Referenziert: ps.name (Pflichtfeld: Ja)
    Eingabefeld vorname: Referenziert: ps.vorname
    Eingabefeld geburtsdatum: Referenziert: ps.geburtsdatum
    Auswahlfeld anrede: Referenziert: ps.anrede Wertebereich: "Herr" "Frau"

  Layout:
    Zeile: Feld anrede "Anrede"
    Zeile: Feld name "Name", Feld vorname "Vorname"
    Zeile: Feld geburtsdatum "Geburtsdatum"
```

Abb. 3: Beschreibungssprache für Bildschirmmasken und die daraus generierte Bildschirmmaske

Einen Generator programmieren

Alle bisherigen Schritte fokussieren auf das Bereitstellen einer Beschreibungssprache und des Toolings. Es wird natürlich auch ein Code-Generator benötigt, der die Klammer zwischen der inferierten Objektstruktur und dem eigentlichen Java-Anwendungs-Quellcode bildet. Dazu werden vorab anforderungsbezogene Implementierungen (als Templates) entwickelt und qualitätsgesichert. Anschließend werden für diesen zu erzeugenden Code entsprechende Generatoranweisungen definiert. Der generierte Quellcode entspricht dadurch exakt der vorher erstellten Referenzimplementierung, auf Basis der Technologien und Architekturkonzepte des zugrundeliegenden Unternehmens-Frameworks.

Die Generatoranweisungen werden in Xtend geschrieben, einer ebenfalls durch Xtext bereitgestellten Sprache. Xtext stellt hierzu Bibliotheken einschließlich Generator-Interface bereit, die es ermöglichen, alle inferierten Domainobjekte – jetzt „Ressourcen“ (Domainobjekte für Masken, Eingabefelder, Navigation, Entitäten, Attribute) – zu iterieren und für jedes referenzierte Objekt, ähnlich einem Compiler, weiteren Code auszuführen. Als Xtend-Compileranweisung werden an dieser Stelle Generatoranweisungen definiert, die Code-Snippets beziehungsweise Textbausteine – letztlich Strings – enthalten, die in einem kompletten konsistenten Ablauf ganze Java-Klassen, XML-Dateien, HTML oder sonstige Dateien repräsentieren und als solche in ein definiertes Verzeichnis gespeichert werden.

Individuelle Programmierung mit „Generation Gap“ ermöglichen

Modelle bergen das grundsätzliche Risiko, dass früher oder später Anforderungen aufkommen, die sich möglicherweise nicht in die ursprünglich festgelegte Modellstruktur einordnen lassen oder sehr spezifisch und individuell sind. Kein Modell kann die Realität abschließend beschreiben. Mit möglichen Defiziten der Modelle gehen Defizite bei der Code-Generierung einher. Es ist daher unabdingbar, von vornherein die Möglichkeit vorzusehen, auch „echten“ programmierten Code einzubringen, mit dem generierter Code nachträglich ergänzt werden kann.

Hier kann durch ein „Generation Gap“-Pattern ganz leicht Flexibilität geschaffen werden, indem jede zu generierende Klasse grundsätzlich als Basisklasse und eine davon zu erbende Klasse generiert werden kann. Die Basisklasse wird standardmäßig generiert und folglich regelmäßig überschrie-

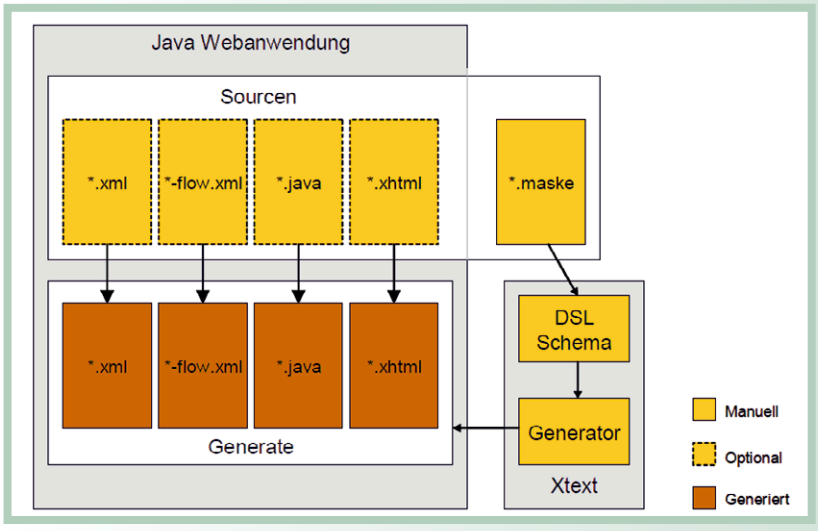

Abb. 4: Alle Artefakte der Präsentationsschicht werden generiert und können manuell ergänzt werden

ben. Die geerbte Klasse hingegen bleibt erhalten und erlaubt es, ergänzenden Code zu definieren oder generierten Code zu überschreiben, wenn individuelle Abweichungen vom generierten Standard erforderlich werden. Dadurch bleiben die hohen Flexibilitätsgrade individueller Quellcode-Entwicklung erhalten.

An dieser Stelle offenbaren sich weitere Risiken. Nämlich, dass durch individuellen Code generative Standards nachträglich ausgehebelt werden können oder schlicht kontraproduktive Implementierungen entstehen können. Dies macht deutlich, dass die domänenbezogenen Grundannahmen des Metamodells immer tragen müssen – oder gegebenenfalls zu erweitern oder partiell zu re-designen sind, um zu vermeiden, dass ein nicht mehr zu den Anforderungen passendes Modell durch individuelle Projekt-Entwicklung korrigiert werden muss.

Um die DSLs in der Praxis der realen Projektanforderungen zu härten, sollten sie auch in ebenfalls agilen Projekten eingesetzt werden. In der Abhängigkeit eines Generator-Teams und der Entwicklungsprojekte, die diese DSLs und Generatoren einsetzen, gibt es eine Reihe von Wechselwirkungen, die nachfolgend betrachtet werden.

Metaagile Entwicklung – MDSD mit Kundenprojekt koppeln

Abhängig von Domäne und Zielgruppe sind Komplexität des Metamodells und die Grammatik und Semantik der Sprache (abstrakt oder Prosa) zu spezifizieren.

Das erklärte Ziel von MDSD ist es, Komplexitäten der herkömmlichen Programmierung – Sprachen, Architekturen usw. – für den Anwender der DSL zu reduzieren. Eine Anwendungstopologie – via MDSD konsequent reduziert – befähigt Fachabteilungen, standardkonforme Anwendungen mittels Dialog-DSL oder Entitäten-DSL weitgehend eigenständig zu erstellen oder zumindest dabei aktiv mitzuwirken und dadurch eine ganz neue Mitwirkung zu entwickeln.

Metamodelle richten sich jedoch an übergreifenden Abstraktionen aus. Eine Dialog-DSL muss allgemein auf die Domain *Dialog* ausgerichtet bleiben und jegliches projektbezogenes Verständnis davon fernhalten. Unabdingbar ist es daher, in der Rolle des *Product Owners* eine vom Kunden unabhängige Instanz einzubringen, die den technischen und domain-bezogenen Gesamtblick bewahrt.

Bedeutsam ist es, den Entwicklungsprozess beider unabhängigen Projekte sprint-versetzt zu synchronisieren:

- Das *Kundenprojekt* erstellt das Stammdatensystem mit zahlreichen Masken. Die technische Herausforderung ist dabei, die Verbindung zu zahlreichen Systemen im Unternehmen zu schlagen. Die Frontend-Entwicklung kann weitgehend davon entkoppelt realisiert werden.
- Das *MDSD-Projekt*, welches iterativ das CRUD-basierte Dialog-Modell und somit die Grammatik und dazu gehörende Generatoren entwickelt.

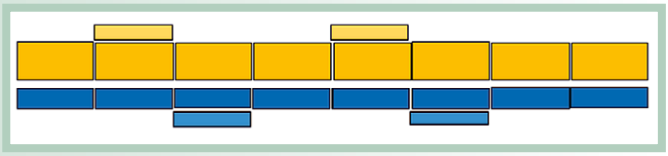

Abb. 5: Sprint-versetzte Sprint-Koordination. Das MDSD-Team (gelb) stellt DSL- & Generatorerweiterungen iterativ als DSL-Release bereit, das Fachprojekt (blau) beschreibt Masken mittels iterativ erweitertem DSL-Sprachumfang

In Scrum etablieren beide Projekte eine direkte Abstimmung als Planning Game, wobei die aus Sicht des Kundenprojekts aktuell wichtigsten (Fach-)Anforderungen bestimmt werden. Diese Feature-Anforderungen werden auf ihre Metamodell-Konformität hin konsolidiert und entweder in das Modell aufgenommen – und damit zukünftig generiert und via DSL-Sprachkonstrukt bereitgestellt – oder, weil zu individuell, an das Kundenprojekt zurückgewiesen und müssen dort klassisch programmiert oder mittels Generation Gap-Muster individuell vervollständigt werden.

Das MDSD-Projekt erweitert somit iterativ das Modell, Grammatik, Editor und Generatoren in einem Sprint, damit dieses als Plug-in released werden kann und im nächsten Sprint des Fachprojekts als DSL-Erweiterung verfügbar gemacht wird.

Fazit: Code-Generierung und Individualentwicklung ergänzen einander

Eine hohe Zahl wiederkehrender und vergleichbarer fachlicher Funktionen und die damit einhergehenden ähnlichen Bearbeitungsabläufe und Maskengestaltung bietet ausreichend Potenzial für die Investition in die Modellierungs- und Generierungswerkzeuge für Unternehmen, in denen die Softwareentwicklung durch den Einsatz zentraler Frameworks und Infrastruktur einen hohen Standardisierungsgrad erreicht hat.

Unternehmen, die Standard-Frameworks in der Entwicklung einsetzen, können mittels MDSD und textuellen Beschreibungssprachen mit Xtext nun effiziente Werkzeuge und Code-Generatoren etablieren, um Fachanwendungen zu beschreiben und somit die bestehenden hohen Standards effizient in die Projekte zu multiplizieren. Für die Kollegen von der Fachseite bietet dieser Ansatz, insbesondere in einem agilen Projektumfeld, ein direktes Feedback für ihre in der DSL formulierten Fachanforderungen.

Die Zusammenarbeit mit der Fachseite, die durch die Anwendung der Beschreibungssprachen selbst stärker IT-bezogen arbeitet, schafft eine neue Ebene ergebnisorientierter Zusammenarbeit zwischen IT und Business und schärft das Bewusstsein, dass die wachsenden Komplexitäten von Business und Technologie beide Seiten vor Herausforderungen stellen, die nur gemeinsam bewältigt werden können.

Frank Jarsch ist Senior Software Engineer bei der Commerzbank AG in Frankfurt am Main. Er begleitet die Commerzbank-interne JEE-Framework-Entwicklung seit 13 Jahren in verschiedenen Rollen. Derzeit leitet er die Entwicklung des zentralen Commerzbank-Java-Standard-Frameworks, welches eine integrierte JEE-Entwicklungsplattform für Intranetprojekte bereitstellt. Neben der Bereitstellung von JEE-/Open-Source-Technologien, Tooling und Infrastruktur-Schnittstellen umfasst dies auch integrierte Lösungsansätze modellgetriebener Softwareentwicklung, mit denen die Effizienz des Frameworks deutlich gesteigert wird.
E-Mail: frank.jarsch@commerzbank.com