November/Dezember 2014 • Nr. 6
€ 9,90 Österreich € 10,90 Schweiz sfr 18,20

OBJEKTspektrum

IT-Management und Software-Engineering

Modelle & Generatoren www.OBJEKTspektrum.de

Modelle & Generatoren

Thomas Stahl & Christoph Gutmann

„Das richtige Setup für Ihr MDD-Projekt"

G 6540F

Meta-Architektur: Das richtige Setup für Ihr MDD-Projekt

Die Motivation zum Einsatz modellgetriebener Techniken (Model-Driven Development) liegt typischerweise in der Automatisierung, Abstraktion oder Softwarequalität. Für den langfristigen Erfolg einer modellgetriebenen Lösung ist auch die Architektur der MDD-Infrastruktur entscheidend. Wartung und Weiterentwicklung müssen genauso wie bei der Zielarchitektur sichergestellt werden können – die aufgebaute MDD-Infrastruktur ist Teil der Lösung. In diesem Artikel werden Projektkategorien identifiziert und mit Lösungsmustern assoziiert. So entsteht ein auf konkreter Projekterfahrung basierender Leitfaden, der dabei hilft, das richtige Setup eines MDD-Projekts zu finden und somit Overhead-Kosten durch Fehlanpassungen zu vermeiden.

In diesem Artikel geht es um Softwarearchitektur – allerdings nicht, wie meist üblich, auf der Ebene von Softwaresystemen für Endanwender, sondern eher auf der Werkzeugebene. Konkret wollen wir uns mit domänenspezifischen Modellierungs- und Generierungsinfrastrukturen befassen. Dabei handelt es sich um individuell erstellte Werkzeugketten, die auf eine bestimmte Problem-Domäne zugeschnitten sind und der Erstellung, der Integration bzw. dem Management von Softwaresystemen für Endanwender dienen. Wir reden also von Infrastruktur-Architektur im Kontext von *Model-Driven (Software) Development (MDSD* – vgl. [Sta06]) bzw. *DSL*-Engineering (*Domain Specific Language*).
Eine MDD-Infrastruktur besteht im Wesentlichen aus folgenden Teilen

- DSLs und Metamodelle
- Transformationen bzw. Generatoren
- Plattform bzw. Laufzeitumgebung

Abbildung 1 zeigt die Zusammenhänge und die Verwendung der MDD-Infrastruktur.

Die Rolle der Domäne

Allen MDD- und DSL-Engineering-Ansätzen ist gemeinsam, dass die Modellierungssprache auf ein klar abgrenzbares Wissensgebiet (= Domäne) eingeschränkt und eben nicht universell (wie z.B. Java, C# oder Scala) ist. Domänen können technisch, architektonisch oder fachlich gewählt sein. Beispiele sind:

- Architektur für datenzentrierte E-Business-Applikationen
- Robotics
- Regelungstechnik
- SmartHome
- Tarifierungsmodelle für Versicherungen
- Technische SOA

Nach unserer Erfahrung hat jedoch die konkrete Ausprägung der Domäne nur einen geringen Einfluss auf die Architektur der MDD-Infrastruktur.

Meta-Architektur

Da eine MDD-Infrastruktur offensichtlich selbst ein Softwaresystem ist, stellen sich im Großen und Ganzen auch ganz ähnliche Fragen bzw. Anforderungen bezüglich der Softwarearchitektur. Wir wollen in diesem Artikel aber versuchen, aus Projekterfahrungen eine Kategorisierung solcher MDD-Infrastrukturen abzuleiten, die es ermöglicht, konkretere Architekturempfehlungen für das Setup zu geben. Statt des länglichen Begriffs *Architektur der MDD-Infrastruktur* benutzen wir im Folgenden den Begriff *Meta-Architektur* als Synonym.

Roundtrip-Engineering

Wie der Titel dieses Artikels ja bereits nahelegt, wollen wir hier nur Ansätze betrachten, die eine echte – und damit domänenspezifische – Abstraktion gegenüber dem Quellcode einer Programmiersprache darstellen. Roundtrip-Werkzeuge (z.B. UML-basierte) tun genau das nicht: Hierbei handelt es sich um Quellcode-Visualisierungen bzw. grafische Editoren auf der Abstraktionsebene von Quellcode.

Kategorisierung

Wir können auf circa 15 Jahre Erfahrungen aus unterschiedlichsten MDD- bzw. DSL-basierten Projekten zurückblicken. Die Idee ist nun, diese Erfahrungen greifbar zu machen, indem wir die unterschiedlichen Ausgangsszenarien – also den Problemraum – möglichst trennscharf kategorisieren und dann für die entstandenen Kategorien jeweils Lösungsmuster angeben. Dabei treten nach unserer Erkenntnis zwei Aspekte besonders hervor, die sich zur Kategorisierung eignen und damit die Dimensionen für unser Raster darstellen. Beide haben etwas mit dem Scope der Modelle bzw. der Modellverwendung zu tun:

1) Die Nachhaltigkeit von Modellen
2) Die Komplexität des Modell Fan-In

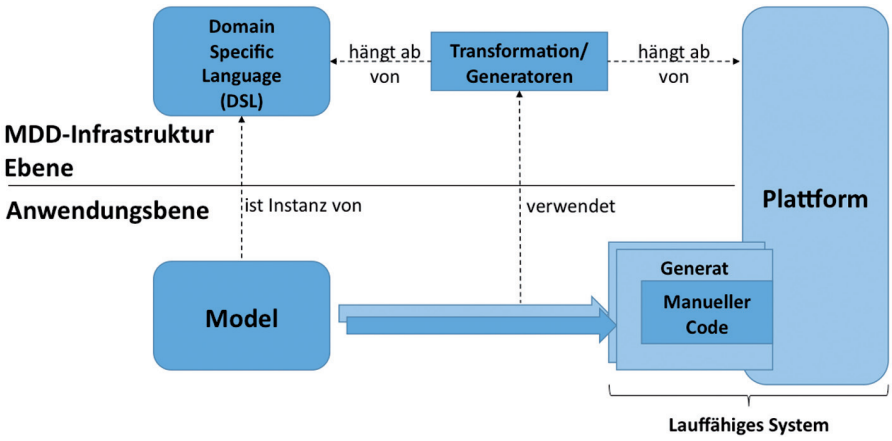


Abb. 1: Bestandteile und Wirkungsweise einer MDD-Infrastruktur.

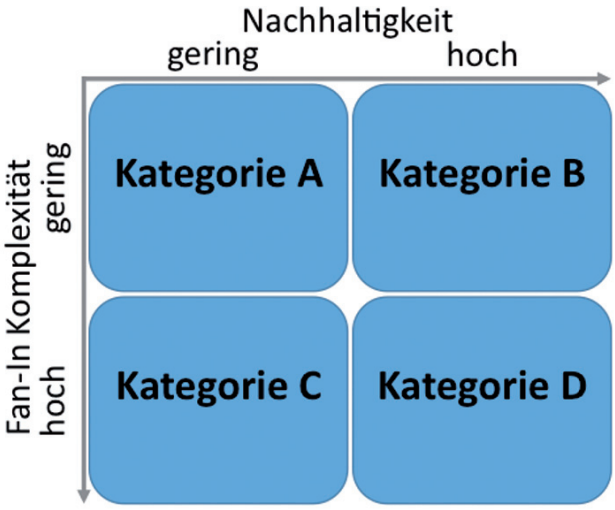


Abb. 2: Kategorisierung durch Nachhaltigkeit und Fan-In-Komplexität.

Das ist sicher erklärungsbedürftig: Nachhaltigkeit kann man in gewisser Weise auch als zeitlichen Aspekt verstehen: Werden die Modelle und die Modellierungsinfrastruktur nur in einem (zeitlich begrenztem) Projekt oder sogar nur innerhalb einer Projektphase benötigt, so bezeichnen wir das als *geringe Nachhaltigkeit*. Umfasst die Nutzung der Modelle mehrere, auch zukünftige Projekte oder Produktentwicklungen (z.B. im Sinne von Produktlinien) oder findet die Nutzung nicht nur auf Projektebene, sondern auf Unternehmensebene statt, bezeichnen wir das als *hohe Nachhaltigkeit* bzw. einfach als *nachhaltig*. Diese Unterscheidung hat also unter Anderem etwas mit Evolutionsfähigkeit zu tun, was entscheidenden Einfluss auf die Wahl einer adäquaten Meta-Architektur hat, wie wir noch sehen werden.

Die Begriffe *Fan-In* und *Fan-Out* stammen aus der Multiplexer-Metapher: Dort gibt es mehrere potenzielle Ein- und Ausgänge, die aufeinander verschaltet werden. Genau dies kann man sich eben auch für ein- und ausgehende Modell-Information in einer Verarbeitungskette vorstellen. Die *Fan-In-Komplexität* umfasst dann mehrere Teilaspekte:

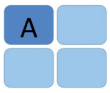
- Die Heterogenität der Eingangskanäle, aus denen Modelle entstehen: Beispielsweise editieren Entwickler Modelle vielleicht mit Texteditoren, ein Manager verwendet eine grafische Benutzungsoberfläche und ein externes Tool liefert Informationen in einem proprietären Format.
- Der Grad an Multi-User-Fähigkeit auf diesen Kanälen.
- Der Grad der Verknüpfung von Informationen aus verschiedenen Kanälen.

Wir unterscheiden im Folgenden der Einfachheit halber lediglich zwischen *geringer* und *hoher Fan-In-Komplexität*. Es folgen noch Beispiele dazu.

Mit diesen beiden Dimensionen mit je zwei Ausprägungen haben wir nun ein Raster, das vier Kategorien bildet: A, B, C und D. Diese Kategorisierung ist für die wesentlichen Aussagen des Artikels bereits hinreichend (**siehe Abbildung 2**).

Um die Kategorien A bis D besser verstehen und analysieren zu können, geben wir für jede Kategorie mindestens ein charakteristisches Beispiel an, das wir als typisch und bis zu einem gewissen Grad als stellvertretend für die jeweilige Kategorie ansehen. Davon ausgehend geben wir Empfehlungen für die Meta-Architektur des Setups pro Kategorie. Zwei dieser Empfehlungsbausteine sind besonders wichtig und werden daher etwas detaillierter in Form von Pattern beschrieben.

Kategorie A: MDD-Infrastrukturen mit geringer Nachhaltigkeit und geringer Fan-In-Komplexität



Charakteristische Beispiele für diese Kategorie sind:

- Bootstrapping
- Applikations-Prototyping mit textueller DSL
- Einzelprojekt mit fachlich schmaler und semantisch flacher DSL

Beim Bootstrapping einer Software mittels einer MDD-Infrastruktur ist erstere anschließend unabhängig lebensfähig bzw. weiter entwickelbar. Ein konkretes Beispiel wäre die Modellierung von Datentypen mittels einer DSL und der initialen Generierung von Data Transfer Objects und JPA Entities. Eine Wiederverwendung der MDD-Infrastruktur ist nur eingeschränkt möglich und auch nicht gefordert – steht das rasche Ergebnis im Vordergrund. Die Lebensdauer der MDD-Infrastruktur und demzufolge auch diejenige der Modelle ist kurz und es gibt nur einen echten Eingangskanal für die Entstehung von Modellen, nämlich die DSL. Bei Kategorie A stehen Pragmatismus und das schnelle Erstellen einer lauffähigen MDD-Infrastruktur im Vordergrund. Da der Aufwand, grafische Editoren zu erstellen, im Gegensatz zur Erzeugung textueller DSLs aus Grammatiken in den meisten Fällen deutlich größer ist, sollte für Projekte in dieser Kategorie in der Regel folgendes Setup passgenau sein.

Typisches Setup für Kategorie A
(**siehe auch Abbildung 3**)

- Rein textuelle DSL.
- Editor(en) aus DSL-Grammatik(en) erzeugt.

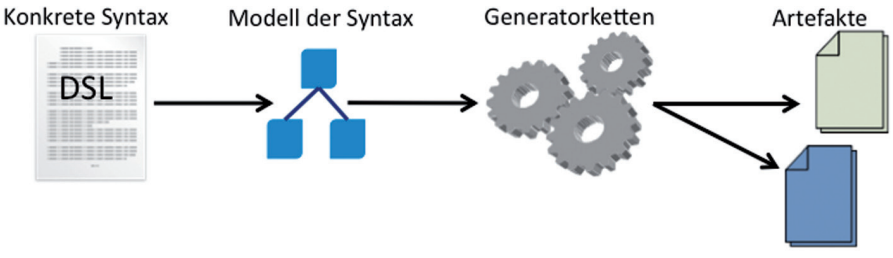
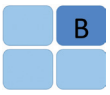


Abb. 3: Typisches Setup für MDD-Infrastrukturen der Kategorie A.

- Metamodell(e) aus Grammatik(en) erzeugt.
- Generatoren setzen direkt auf Metamodell der DSL-Grammatik auf.
- Textuelle DSL-Dateien in einem (Standard-)Quellcode-Repository zusammen mit gegebenenfalls vorhandenem manuell implementierten Code versionieren/verwalten.

Mit diesem Setup lassen sich extrem schnell erste sichtbare Erfolge produzieren.

Kategorie B: Nachhaltige MDD-Infrastrukturen mit geringer Fan-In-Komplexität



Ein charakteristisches Beispiel für diese Kategorie ist:

- Softwarefabrik mit architekturzentrischer DSL

Softwarefabriken automatisieren den Entwicklungsprozess für die maßgeschneiderte Umsetzung einer Lösung innerhalb des vorgegebenen Softwarearchitektur-Stils. Der Scope des Metamodelles ist an den Architekturstil gebunden und in der Implementierung der Generatorketten eingegossen. Für die Zielarchitektur stellt die Softwarefabrik aufgrund der Standardisierung einen Governance-Enabler dar. Der Scope der Modelle entspricht dem jeweils zu generierenden Projekt oder Produkt, wobei die Modelle unterschiedlicher Projekte bzw. Produkte voneinander isoliert sind. Charakteristisch für die MDD-Infrastruktur einer Softwarefabrik ist somit deren Wiederverwendung über Projekt- bzw. Produktgrenzen hinaus. Die Fan-In-Komplexität einer Softwarefabrik ist gering, da nur Entwickler modellieren und zwar mit genau einer, gegebenenfalls partitionierten DSL.

Als konkretes Beispiel für eine Meta-Architektur dieser Kategorie werfen wir an dieser Stelle einen Blick auf die interne Produktentwicklungsplattform der b+m AG.

Die Laufzeit-Architektur ist eine moderne Layer-Architektur mit einem typischen JEE-Technologie-Stack (Hibernate/JPA/EJB3, WebServices, JBPM, Vaadin). Doch hier geht es nicht weiter um die Zielarchitektur, sondern um die MDD-Infrastruktur. Diese zeigt aus statischer Sicht grob **Abbildung 4**.

Analog zu den Schichten der Zielarchitektur – Workflow, Frontend, Service, Entity – gibt es in der MDD-Infrastruktur Partitionen, d.h. in sich abgeschlossene, lose gekoppelte architekturzentrierte DSLs und Generatorketten. Für den Workflow wird BPMN2.0 verwendet, alle anderen Partitionen besitzen textuelle DSLs.

Die Softwarefabrik selbst muss ebenso wie die damit erstellen Branchenprodukte zukunftsfähig, d.h. evolutionsfähig, sein, denn sie ist integraler Bestandteil der Produkte. Genauer: Jedes Release eines Produkts basiert auf einer konkreten Version der Fabrik (**siehe auch Abbildung 1**). Evolution kann bei einer Softwarefabrik auf verschiedenen Ebenen relevant sein:

- *Technologiewechsel:* Beispielsweise der Wechsel der Frontend-Technologie in der Plattform von JSF auf Vaadin. Der Impact eines Technologiewechsels sollte bei einer Softwarefabrik im Wesentlichen auf den Generator beschränkt sein.

- *DSL-Evolution:* Es entsteht eine neue Partition in der DSL-Landschaft, die es bislang nicht gab, oder die DSL ist auf Grund von Feedback aus den nutzenden Projekten Änderungen unterworfen.

Aus diesen Problemstellungen heraus wird bereits klar: Eine Softwarefabrik benötigt eine komplexere Architektur als MDD-Infrastrukturen der Kategorie A und wir kommen zu unserem ersten Lösungsmuster „Explizites Domänenmodell“ (**siehe Kasten 1**).

In **Abbildung 4** sieht man das Domänenmodell in der Mitte der Transformationskette von der DSL- zur Plattform-Ebene. Damit kommen wir zu folgendem repräsentativen Setup.

Typisches Setup für Kategorie B

- Partitionierte DSL, falls die Domäne einen größeren Scope besitzt.

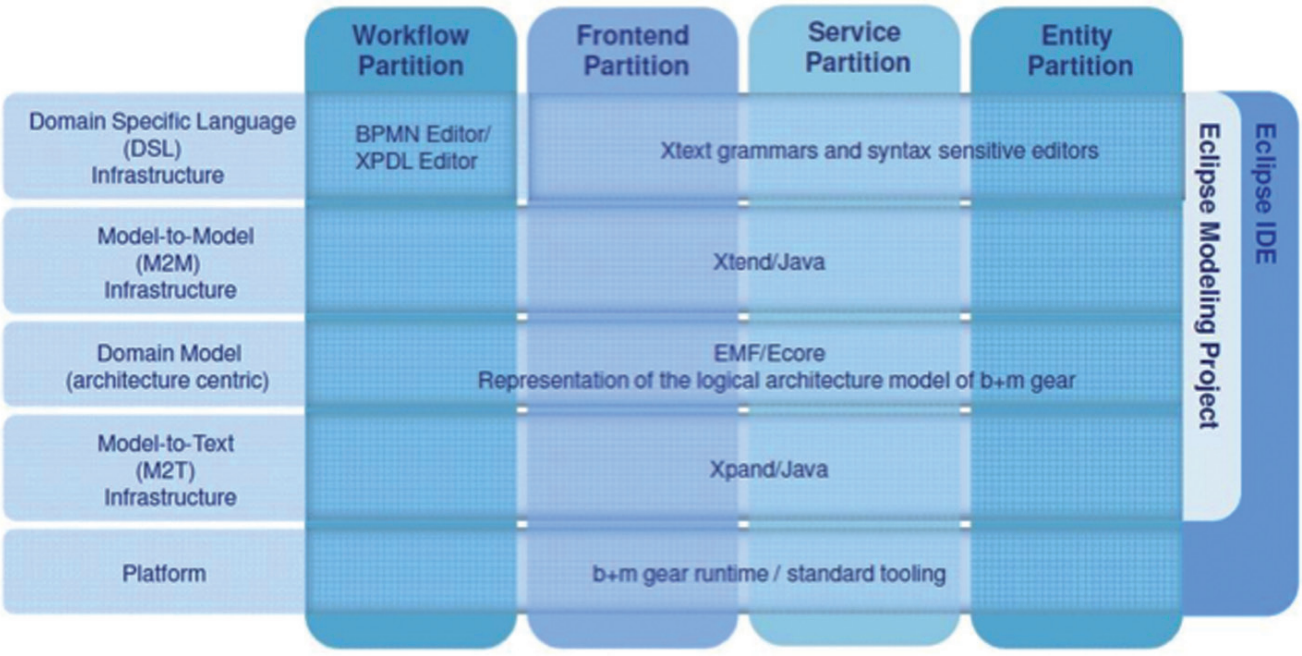
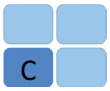


Abb. 4: Beispiel einer MDD-Infrastruktur der Kategorie B (Quelle: b+m AG).

- Textuelle und gegebenenfalls grafische Editoren je DSL-Partition.
- Grafische Read-Only-Repräsentationen (Reports) für bestimmte Partitionen, auf Basis des Domänenmodells generiert (Fan-Out).
- Textuelle Editoren aus DSL Grammatiken erzeugt.
- Gegebenenfalls Einbindung standardisierter, grafischer Editoren (im Beispiel: BPMN2.0).
- Explizites Domänenmodell, transient.
- Modell-zu-Modell-Transformationen zwischen DSL(-Partitionen) und Domänenmodell.
- Generatorketten für Fan-Out setzen auf dem Domänenmodell auf.
- Textuelle DSL-Dateien und serialisierte grafische Modellteile in einem (Standard-)Quellcode-Repository zusammen mit gegebenenfalls vorhandenem manuell implementierten Code versionieren/verwalten.

Kategorie C: MDD-Infrastrukturen mit geringer Nachhaltigkeit und hoher Fan-In-Komplexität



Ein charakteristisches Beispiel für diese Kategorie ist:

- Model-Driven Software Modernisation

Die Modernisierung eines (Legacy-)Softwaresystems mit Hilfe modellgetriebener Techniken benötigt im Sinne unserer Kategorisierung „nur" eine MDD-Infrastruktur mit geringer Nachhaltigkeit, denn nach Abschluss der Modernisierung ist ihr Zweck ja zunächst einmal erfüllt. Dass man die modernisierte Anwendung mit Hilfe von DSLs und Generatoren vielleicht auch weiter entwickeln bzw. warten möchte, steht auf einem anderen Blatt. Mit Modernisierung meinen wir übrigens nicht die kurzfristige Rettung von Legacy-Systemen durch regelbasierte Cross-Compiler, die z.B. aus COBOL-Code Java-Code produzieren, sondern eine nachhaltige Modernisierung, die ein zukunftsfähiges System hinterlässt. Typische technische Ziele solcher Vorhaben sind Architektur-Restrukturierung, Service-Enabling für eine SOA (BPM-Fähigkeit) und Modularisierung. Im Extremfall kann oder soll die bestehende Software selbst vielleicht gar nicht einer Transformation unterzogen, sondern teilweise oder ganz durch ein Standardprodukt ersetzt werden. Doch selbst dann ist Wissen über die Altanwendung nötig.

Eine entsprechende Modernisierung muss zunächst sehr heterogene Informationskanäle, wie statische Analysen durch Parser, dynamische Analysen durch Monitoring und Ergänzungen durch menschliches Wissen zusammenführen und sie vor allem *miteinander verknüpfen*. Daraus entsteht dann ein holistisches Gesamtmodell des Altsystems, das nicht nur den Code repräsentiert, sondern auch die Architektur und die Nutzungsszenarien. Es ist dann die Grundlage für die eigentliche Modernisierung oder die Abschätzung bzw. Planung eines Ersatzes. Dieses Modell ist wesentlich abstrakter und reichhaltiger als der

Pattern-Name
Explizites Domänenmodell.

Context
Aufbau von nachhaltigen Modeling Infrastrukturen.

Problem
Die Anforderung an das Modell auf der Seite des Editors unterscheiden sich wesentlich von denen auf der Seite der Generatoren. Trotzdem soll die Implementierung des Editors und der Generatoren wartbar und somit nachhaltig sein.

Forces
- Die konkrete Syntax der DSL gibt die Rahmenbedingungen für das unterliegende Modell vor.
- Nachhaltige Implementierungen von Generatoren sind auf einfach navigierbare Modelle angewiesen.
- Die Lebenszyklen von Editoren und Generatoren sind unabhängig.

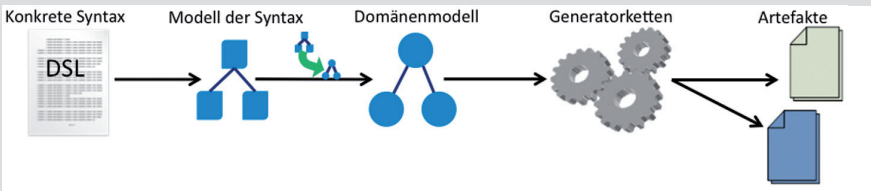


Abb. 5: Pattern: Explizites Domänenmodell.

Solution
Entkoppelung zwischen Fan-In und Fan-Out, um beide Seiten möglichst unabhängig voneinander variieren zu können.

Da wir hier von domänenspezifischen Modellierungs-Infrastrukturen sprechen, ist das Konzept der Domäne immanent und man kann es zur Entkopplung einsetzen, indem man es explizit codiert. D.h. es wird ein rein fachlich motiviertes, domänenspezifisches Metamodell erstellt (z.B. mit EMF), das genau die Konzepte der Domäne und ihrer Beziehungen untereinander enthält. Dieses Metamodell heißt „Domänenmodell" und ist insbesondere unabhängig von der konkreten Syntax der DSL. Das Domänenmodell bleibt stabil, selbst wenn z.B. eine textuelle DSL durch eine grafische ersetzt wird. Es evolviert nur, wenn sich neue Erkenntnisse über die Domäne selbst ergeben, die eine konzeptionelle Änderung nach sich ziehen, oder wenn neue, bisher nicht implementierte Aspekte hinzukommen. Alle Modelle werden nun technisch als Instanzen des Domänenmodells gehandhabt bzw. in solche transformiert. Die Generatorketten basieren dann nur noch auf dem Domänenmodell und haben keine Abhängigkeit zur DSL-Seite mehr – oder allgemeiner dem Fan-In. Dadurch können DSL-Syntax und Technologien in der Zielarchitektur unabhängig voneinander variieren – insbesondere wird die DSL änderbar. Das Pattern trifft zunächst keine Aussage darüber, ob Domänenmodell-Instanzen persistent oder transient sind (**siehe Abbildung 5**).

Kasten 1: Das Pattern „Explizites Domänenmodell".

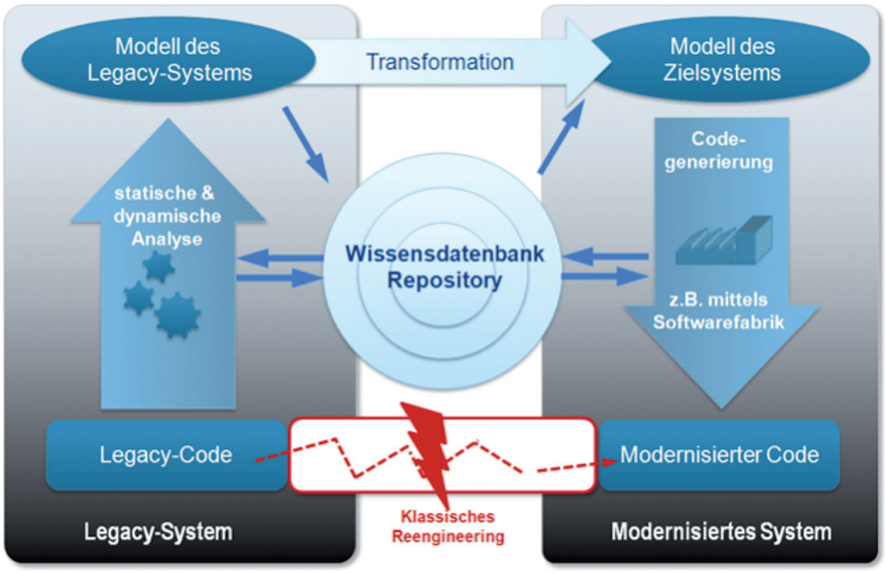


Abb. 6: Model-Driven Modernization (Quelle: b+m AG).

Quellcode des Altsystems und im besten Falle kann man es (teil-)automatisch mittels Modell-zu-Modell-Transformation in eine neue Zielarchitektur überführen. Das Software-Modernisierungsverfahren der b+m AG folgt dem Hufeisen-Prinzip: *Abstraktion->Transformation->Konkretisierung* (vgl. [Dyn]). **Abbildung 6** zeigt das Konzept.

Zur Neutralisierung der heterogenen Eingangskanäle in die Modellwelt einerseits und zur Verknüpfung der Informationen andererseits (hohe Fan-In Komplexität) liegt es nahe, hier ein Repository ins Zentrum der Meta-Architektur zu legen, womit wir bei unserem zweiten Pattern, dem „Modell-Repository“ wären (**siehe Kasten 2**).

Das *Modell-Repository* in unserem charakteristischen Beispiel ist in **Abbildung 6** prominent zu sehen, das *Explizite Domänenmodell* ist jedoch nur in Form der Ringe angedeutet: Der innerste Ring enthält Informationen auf Quellcode-Ebene, der mittlere Ring beschreibt die Architektur des Altsystems und der äußere Ring die Nutzungs- und Anforderungsebene. Die Eingangskanäle, die notwendig sind, um dieses holistische Informationsmodell zu befüllen, erzeugen die hohe Fan-In-Komplexität. Der Kollaborationsaspekt ist in Kategorie C jedoch typischerweise eher schwach ausge-

Pattern-Name
Modell-Repository

Context
Bereitstellung einer Informationsdrehscheibe bei hoher Fan-In-Komplexität.

Problem
Die Konsistenz des Modells ist über Mehrbenutzer-Szenarien und/oder unterschiedliche Eingangskanäle sicherzustellen.

Forces
- Das Modell muss zentral persistiert werden.
- Das Modell wird kollaborativ bearbeitet.
- Verschiedene Modell-Bearbeitungswerkzeuge für unterschiedliche Arten von Stakeholdern werden benötigt.
- Die Konsistenz des Modells ist zentraler Erfolgsfaktor.
- Suche- und Recherche-Möglichkeiten auf der Abstraktionsebene der Domäne, nicht der Persistenz-Technologie.

Solution
Bereitstellung einer Informationsdrehscheibe auf Modellebene, dem Modell-Repository. Neben der Implementierung der nackten Persistenz unterstützt das Modell-Repository die Realisierung hoher Fan-In-Komplexität sowohl in Bezug auf Multi-User-Szenarien als auch auf die Anbindung unterschiedlicher Eingangskanäle. Umgesetzt wird dies durch ein Transaktionskonzept auf fachlicher Ebene. Eine textuelle DSL beispielsweise ist also nur ein möglicher Eingangskanal. Ein zielgruppengerechtes User-Interface oder ein Adapter zu einer externen Informationsquelle könnten weitere Eingangskanäle sein.

Related Patterns
Impliziert *Explizites Domänenmodell*.
Ein Modell-Repository ist eine Art Laufzeit-Plattform für Modelle. Die Heterogenität der Eingangskanaltypen, die Notwendigkeit der semantischen Verknüpfung der Informationen aus diesen Kanälen sowie der Bedarf für eine Entkopplung zwischen Fan-In und Fan-Out erfordern es, das oben definierte Pattern *Explizites Domänenmodell* zu verwenden. Konkret liegen damit Instanzen des Domänenmodells persistent im Repository und nicht (serialisierte) Instanzen der konkreten DSL-Syntax. Das Domänenmodell selbst stellt somit eine Konfiguration des Repositories dar. Das Pattern *Modell-Repository* impliziert das Pattern *Explizites Domänenmodell*, aber nicht umgekehrt. Ein Beispiel für eine domänenunabhängige Modell-Repository-Basistechnologie ist CDO (vgl. [CDO]).

Kasten 2: Das Pattern Modell-Repository.

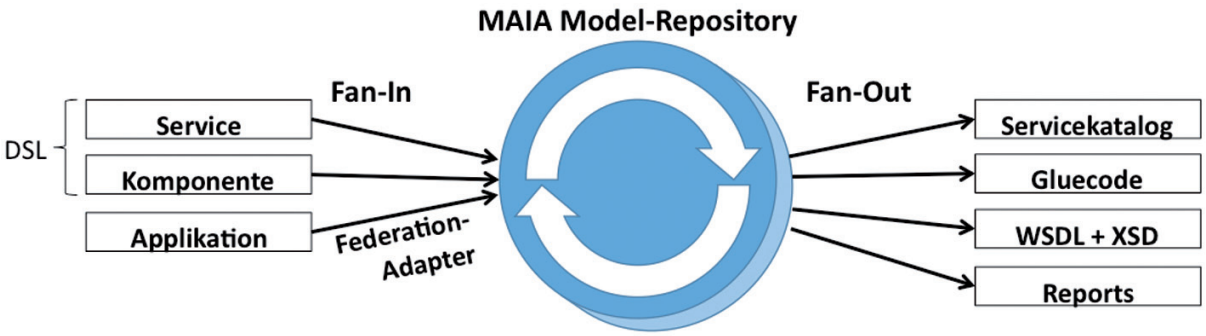


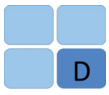
Abb. 7: Das MAIA Modell-Repository.

prägt. Insgesamt führt dies zu dem folgenden Setup.

Typisches Setup für Kategorie C

- Partitionierte DSL, falls die Domäne einen größerem Scope besitzt.
- Fan-In: DSL-Editoren, Adapter für externe Werkzeuge, GUIs.
- Modell-Repository mit *gegebenenfalls nur schwacher Kollaborationsunterstützung*.
- Explizites Domänenmodell.
- Modellpersistenz und Versionierung auf Abstraktionsebene des Domänenmodells sind Dienste des Modell-Repositories.
- Navigation und Suchfunktionalität auf Abstraktionsebene des Domänenmodells sind durch das Modell-Repository unterstützt.
- Modell-zu-Modell-Transformationen zwischen Fan-In-Artefakten und Domänenmodell.
- Generatorketten für Fan-Out setzen auf dem Domänenmodell auf und sind als Dienste (Plug-Ins) in das Modell-Repository integriert.

Kategorie D: Nachhaltige MDD-Infrastrukturen mit hoher Fan-In-Komplexität



Ein charakteristisches Beispiel für diese Kategorie ist:

- Infrastruktur für Model-Driven SOA

Motivation für eine Model-Driven SOA ist eine möglichst hohe Interoperabilität sowie eine gute Managbarkeit einer heterogenen Systemlandschaft und der beteiligten Prozesse auf verschiedenen Ebenen (z. B. Enterprise Architektur Management (EAM), Entwicklung, Betrieb). Als Grundlage einer Model-Driven SOA dient in der Regel eine Zielarchitektur mit Grundprinzipien, wie die einzusetzenden Message-Exchange-Patterns oder fachliche Versionierungskonzepte. Der Scope des Metamodells ist an die gesamte zu bebauende Landschaft gebunden. Auf Instanzebene existiert als Konsequenz ein einziges holistisches Modell der gesamten SOA-Landschaft: Business-Services, technische Service-Interfaces, Typ-Systeme, Komponenten, Ports, Wirings, Deployments usw. Die Eingangskanäle für ein solches Modell sind grundsätzlich vielfältig: Elemente der SOA-Landschaft haben ihren Ursprung sowohl in unterschiedlichen Organisationsebenen wie der Informationssystemarchitektur oder der Entwicklung, als auch in unterschiedlicher Partitionierung auf Instanzebene auf Grund fachlicher Segmentierung. Als Konsequenz tauchen in der Anforderungsliste in der Regel Multi-User-Szenarien auf. Am Ende des Tages liegt ein horizontal und vertikal segmentiertes Modell vor und stellt eine Informationsdrehscheibe dar.

Auf der Nutzungsseite wird das Modell zur Generierung von unterschiedlichsten Artefakten verwendet. Ein wichtiges Grundprinzip und Treiber ist auch hier die Konsistenz zwischen allen Zielsystemen auf allen Ebenen – von der Entwicklung bis in den Betrieb.

Kategorie D besitzt offensichtlich den höchsten Anforderungsgrad bezüglich unserer Kategorisierung. Hier stellen das Erreichen der Konsistenz einer langlebigen und mehr oder weniger holistischen Modellinstanz sowie der Umgang mit sehr heterogenen Informationsquellen die zentralen Herausforderungen für die Meta-Architektur dar. Daher ist der Einsatz eines *Modell-Repositories* auch hier essenziell.

Eine konkrete Model-Driven SOA ist im Projekt *MAIA (Mobiliar Application and Infrastructure Automation)* entstanden. In MAIA werden die Service-Schnittstellen und -Komponenten der Mobiliar SOA-

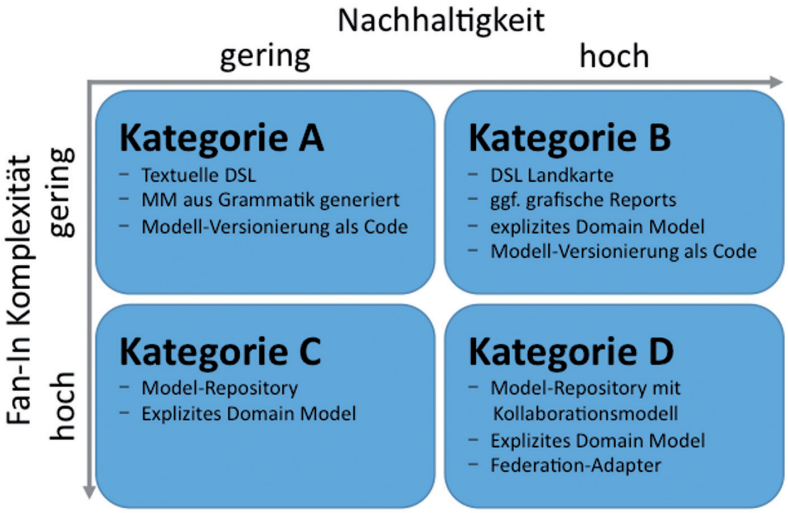


Abb. 8: Empfehlungen zu den vier Kategorien.

Landschaft verwaltet. Aus dem Modell werden einerseits automatisiert konsistenter Code (z. B. für Service-Stubs- und Skeletons), Gluecode zur Abstraktion der unterliegenden Transportschicht sowie Entwickler-Dokumentation erzeugt, und andererseits ein dazu konsistenter Service-Katalog. Damit wird bei hoher Qualität der Aufwand für die Service-Entwicklung reduziert, die Wiederverwendbarkeit der Services sichergestellt und die Abhängigkeiten der Komponenten werden sichtbar gemacht. Die Konsistenz innerhalb des Modells und zur Realität ist dabei ein wesentlicher Erfolgsfaktor. Innerhalb des Modells wird die Konsistenz direkt beim Editieren überprüft. Alles, was im Modell definiert ist, steht grundsätzlich für die Generierung, das Reporting sowie aktuelle und zukünftige Analysen zur Verfügung. Denkbar ist die Auswertung von Key Performance Indicators (KPIs), die Erstellung von Heatmaps, Abhängigkeitsanalysen und vieles mehr (**siehe Abbildung 7**).

Als Fan-Out für die Entwicklung sind insbesondere Service-Endpunkte und -Clients zu nennen. Weiter werden aus dem Modell der Service-Katalog mit fachlicher Dokumentation und Abhängigkeitsinformationen für den Betrieb sowie Reports in Form von Graphen und Tabellen generiert.

Auf der Fan-In-Seite kommt neben mehreren textuellen DSL-Partitionen auch ein Federation-Adapter zur Anbindung der in einem EAM-Tool verwalteten Applikations-Entitäten zum Einsatz: Um Modelle, die in unterschiedlichen Welten leben, verknüpfen zu können, müssen das so genannte Information-Ownership und das Vorgehen bei der Bearbeitung von „Grenzentitäten“ scharf definiert sein. Diese Grenzdefinition legt fest, welche Entitäten und Attribute mindestens in das jeweils fremde System provisioniert werden müssen. Minimal sind dies die Identitäten der referenzierten Entitäten.

Die Autoren

Thomas Stahl
(t.stahl@bmiag.de)
arbeitet als Chief Software Architect / CTO bei der b+m Informatik AG. Er ist Buchautor und Sprecher auf Konferenzen. Die Themen Model-Driven Development und DSL-Engineering gehören zu seinen Arbeitsschwerpunkten.

Christoph Gutmann
(christoph.gutmann@mobi.ch)
arbeitet als technischer Architekt bei der Schweizerischen Mobiliar Versicherungsgesellschaft. Seine Schwerpunkte sind die Konzipierung und Umsetzung der Model-Driven-Entwicklung im Kontext der technischen SOA sowie die Unterstützung der internen Softwareentwicklung.

Die DSL-Editoren und die Generatoren teilen sich dieselbe, auf dem Domänenmodell realisierte Validierung.

Durch den Einsatz des Musters „Modell-Repository“ besitzt Kategorie D auf der Lösungsseite offensichtliche Überschneidungen mit Kategorie C, allerdings kommen hier erhöhte Anforderungen an die Kollaborationsfähigkeit hinzu.

Typisches Setup für Kategorie D

Setup wie bei der Kategorie C, zusätzlich:

- Partitionierte DSL.
- Modell-Repository mit Kollaborationsaspekt.
- Federation-Adapter für die Anbindung anderer Informationsquellen (z. B. auf EAM-Ebene).

Zusammenfassung und Fazit

Abbildung 8 fasst noch einmal die Setup-Empfehlungen aller vier Kategorien zusammen. Als Take-Away möchten wir Ihnen Folgendes mitgeben: Over-Engineering produziert unmittelbar Zusatzkosten, Under-Engineering produziert mittelfristig Zusatzkosten. Um beides zu vermeiden, sollten sie ein MDD-Projekt in Kategorie X auch (zumindest ungefähr) mit dem Setup von Kategorie X ausstatten. Je weiter Projekt- und Setup-Kategorie auseinanderliegen, desto größer ist die Wahrscheinlichkeit, dass eine Fehlanpassung vorliegt. Das ist eine qualitative Aussage und wir können den resultierenden wirtschaftlichen Schaden natürlich nicht allgemein quantifizieren – aber beispielsweise kann das irgendwann notwendige „Heben“ eines Kategorie-D-Projekts, das mit einem Kategorie-A-Setup versehen ist, leicht mehrere hundert Personentage Overhead erzeugen. Solche Fehlanpassungen (meist leichtere) sind uns in der Praxis leider schon häufiger begegnet. Weiterhin kann es durchaus vorkommen, dass sich die Anforderungen an ein Projekt so ändern, dass es in eine andere Kategorie wechselt. In diesem Falle sollten Sie darüber nachdenken, auch die Meta-Architektur entsprechend anzupassen. Also: „Software Architecture matters” – und zwar auch auf der Ebene der MDD-Infrastruktur. ■

Literatur & Links

[Dyn] DynaMod – Dynamische Analyse für modellgetriebene Software-Modernisierung, siehe: `http://kosse-sh.de/dynamod/`

[CDO] The Eclipse Foundation, The CDO Model Repository, siehe: `http://www.eclipse.org/cdo/`

[Sta06] T. Stahl, M. Völter, Model-Driven Software Development, Wiley 2006