MODELLGETRIEBENE SOFTWARE-ENTWICKLUNG IN EINEM GROSSPROJEKT: EIN PRAXISBERICHT ÜBER DEN ERFOLGREICHEN EINSATZ VON MDD

der autor

Benedikt von Treskow
(benedikt.vonTreskow@credit-suisse.com)
ist Solution Architekt bei der Credit Suisse AG und verantwortlich für Architektur und Design von serviceorientierten JEE-Applikationen. Seine Schwerpunkte sind die Konzeption und Umsetzung von modellgetriebenen Entwicklungsmethoden.

Die Vorteile von MDD sind bekannt: erhöhte Produktivität durch Automatisierung sowie verbesserte Qualität und Wartbarkeit, um nur die wichtigsten zu nennen. Wie Modelltransformationen funktionieren und welche Werkzeuge es gibt, beschreiben thematisch verwandte Artikel. Aber wie funktioniert der MDD-Ansatz in einem realen Großprojekt? Dieser Artikel befasst sich mit der MDD-Vorgehensweise in der Praxis und stellt aus den gemachten Erfahrungen eine Reihe von Bausteinen vor, die zu einem erfolgreichen Einsatz von MDD führen. Dabei steht nicht nur Technologie im Vordergrund, sondern es werden auch Prozessfaktoren und organisatorische Faktoren mit einbezogen.

Im Folgenden stelle ich kurz das Projekt vor, um das es in diesem Artikel geht, und erläutere die Beweggründe, weshalb in der Initiierungsphase des Projekts der modellgetriebene Weg eingeschlagen wurde. Die Motivation für eine modellgetriebene Softwareentwicklung (*Model Driven Development, MDD*) lässt sich auf andere ähnliche Projekte übertragen. Anschließend beschreibe ich die wichtigsten Bausteine näher und gehe auf die Lektionen ein, die wir bei der Einführung und Verwendung modellgetriebener Methoden gelernt haben. Eine vollständige Auflistung aller Faktoren ist aus Platzgründen nicht möglich. Abschließend gehe ich der Frage nach, wie MDD mit agilem Vorgehen kombiniert werden kann und welche Herausforderungen dabei bestehen.

Das Projekt

Die in diesem Artikel geschilderten Erfahrungen stammen aus einem mehrjährigen Modernisierungsprojekt, das ein in die Jahre gekommenes Kernsystem auf einer modernen SOA-Plattform neu implementiert. Dabei geht es nicht um eine Code-zu-Code-Migration, sondern um den Entwurf einer von Grund auf neuen Architektur für ein dezentrales System, das sich mit unterschiedlichen Schnittstellen-Technologien in die Applikationslandschaft einfügt.

In einem Vorprojekt wurden bereits modellgetriebene Ansätze gewählt, um das Altsystem nachzudokumentieren sowie Systemstrukturen und Abhängigkeiten zu analysieren. Ziel war dabei die möglichst vollständige Erfassung der bestehenden Funktionalität sowie die Planung des mehrjährigen Migrationsvorgehens.

Zu den bestehenden Anforderungen des Ist-Systems werden durch die Optimierung der Geschäftsprozesse zusätzliche Anforderungen erhoben. Doch nicht nur funktionale, sondern auch hohe nicht-funktionale Anforderungen, wie Performance und hohes Transaktionsvolumen, die für zukünftige Ansprüche auch nach oben skalierbar sein müssen, sind Herausforderungen für das Projekt. Ein solches Modernisierungsvorhaben ist nur mit einem großen Team zu bewerkstelligen. Circa 80 Projektmitglieder arbeiten gemeinsam an der Lösung vielfältiger Aufgaben.

Die MDD-Methode wurde in drei Phasen schrittweise eingeführt (siehe Abbildung 1). Zu Beginn entwickelte das Architekturteam eine Referenzapplikation. In dieser wurden diverse technische Prototypen vereint, um die Architektur zu verifizieren und generierbare Teile zu identifizieren. Die Referenzapplikation diente später als Testumgebung und Hilfsmittel für das Einlernen neuer Teammitglieder.

In der zweiten Phase wurde die MDD-Werkzeugkette finalisiert und darauf basierend das erste Applikations-Release 0.5 mit einem kleineren Entwicklerteam verwirklicht. Ziel war es dabei, die *End-to-End*-Machbarkeit der modellgetriebenen Methode zu prüfen und die Werkzeugkette für das Ausrollen auf ein größeres Team vorzubereiten.

Meilensteine der dritten Phase waren die Verwendung der MDD-Werkzeugkette im gesamten Team inklusive Offshore-Mitarbeitern und die erfolgreiche Lieferung des folgenden Applikationsrelease 1.0. Aktuell ist die MDD-Werkzeugkette im Wartungsmodus und bildet die Basis für zukünftige Releases.

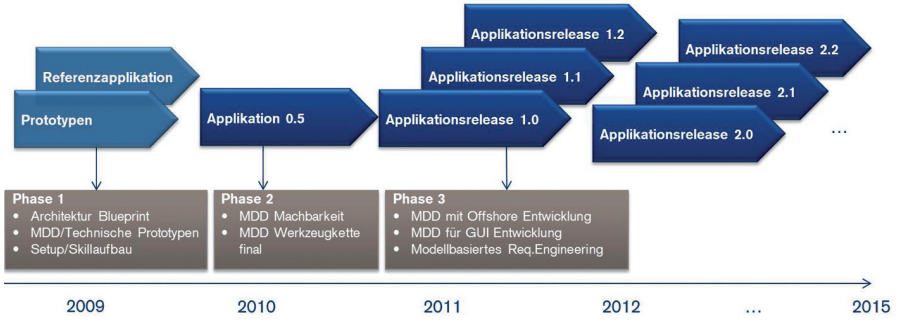

Abb. 1: Einführung von MDD im Projekt.

Treiber für den MDD-Einsatz

Im Folgenden werden die Treiber des konkreten Projekts erläutert, weshalb eine modellgetriebene Vorgehensweise gewählt wurde. Die hier genannten Gründe sind nicht projektspezifisch, sondern bekannte Gegebenheiten in vergleichbaren Projekten.

Wartbarkeit und Langlebigkeit

Für das zu entwickelnde System wird eine lange Lebenszeit erwartet, sodass hohe Anforderungen an die Wartbarkeit gestellt werden. Dies gelingt nur durch eine stets aktuelle Dokumentation des Systems, die durch Modelle (=Dokumentation) und den Einsatz von Codegeneratoren immer synchron mit der Implementierung gehalten wird.

Die Erfahrung mit dem Altsystem hat gezeigt, dass eine vernachlässigte Dokumentation zu schwerer Wartbarkeit führen kann. Sehr aufwändig sind Re-Dokumentationen und Analysen, die immer durch Abstraktion (modellbasiert, grafische Abhängigkeiten) bewerkstelligt werden. Durch MDD wird diese Abstraktion von Anfang an gelebt und dadurch eine hohe Qualität der Dokumentation erreicht (*Document-first*). Modelle stellen nicht das einzige Dokumentationsmedium dar. Sie bieten aber immer einen Einstiegspunkt in das System und referenzieren weiterführende technische Dokumentation.

Kommunikation in verteilten Teams

Auch wenn es der Idealzustand wäre, dass alle Projektmitglieder in einem Raum arbeiten, sind die Teams in derartigen Projekten zum Teil global verteilt. Häufig werden Anforderungen und Geschäftsprozesse von einem Team spezifiziert, während ein anderes Team Design und Implementierung übernimmt.

Die Größe des Projektteams erfordert präzise Spezifikationen. Vor allem in Offshore-Szenarien können durch den Einsatz von MDD Anforderungen eindeutig beschrieben werden. Durch Modelle wird eine gemeinsame Sprache definiert, die durchgängig nachvollziehbar ist – von den Anforderungen, über das Design bis hin zum Code. Die durch MDD herbeigeführte Abstraktion ermöglicht es, Fachexperten einzubeziehen, die nicht unbedingt die eingesetzte Zieltechnologie beherrschen müssen.

Automatisierung

Die mehrjährige Laufzeit des Projekts erfordert ein einheitliches Vorgehen auf der vorgegebener Architektur. Zwar müssen Korrekturen (*Refactorings*) auch an der Architektur möglich sein, der grundlegende Architekturstil (z. B. Schichtenmodell) sowie eingesetzte Muster und Frameworks werden jedoch zu Beginn definiert und nicht jedes Jahr neu festgelegt.

Im Sinne der „Software-Industrialisierung“ (vgl. [Zie10]) errichtet man mit MDD eine Fertigungsstraße, auf der in geteilten Arbeitsschritten Anforderungen spezifiziert sowie Design und Implementierung erstellt werden. Die Architektur- und Design-Richtlinien werden von Anfang an durch Codegenerierung durchgesetzt, sodass sich die Entwickler auf das Wesentliche, nämlich die Implementierung der Fachlogik, konzentrieren können.

Technologische Bausteine

„Out of the box“ eignen sich MDD-Werkzeuge nur bedingt für den produktiven Einsatz. Üblich ist die Anpassung von MDD-Generatoren an die Gegebenheiten der Zielplattform sowie an die Architektur- und Unternehmensstandards, wie beispielsweise die Verwendung existierender Frameworks oder Code-Richtlinien.

In diesem Abschnitt beschreibe ich die wesentlichen technologischen Bausteine für die Erstellung bzw. Anpassung eines Generator-Frameworks.

Domänenspezifische Sprache

Die Definition eines Metamodells ist die Basis der domänenspezifischen Sprache (*Domain Specific Language, DSL*) und bildet ein eindeutiges Verständnis und gemeinsames Vokabular für die Zielarchitektur. Die Erstellung des Metamodells ist typischerweise ein iterativer Prozess. Zu Beginn steht ein konzeptioneller Entwurf (*Blueprint*), der dann in weiteren Schritten stetig verfeinert wird.

In Abbildung 1 zeigen die Schritte 1 und 2 exemplarisch, wie das konzeptionelle, schichtenbasierte SOA-Modell „Architektur Blueprint“ in das „Formale Metamodell“ überführt wird. Im Projektbeispiel wurden im Blueprint zuerst die Schichten konzipiert, beispielsweise: Wo finden Input/Output-Mapping und Validierung statt? Wo werden Business-Regeln ausgeführt? Wo werden Datenzugriffe implementiert oder externe Systeme aufgerufen? Während der Verfeinerung zum „Formalen Metamodell“ werden dann konkrete Elemente auf den Schichten spezifiziert sowie deren Attribute und Abhängigkeiten untereinander. Hier werden beispielsweise spezifische Servicetypen definiert (Datenbank, Regelausführung, externer Aufruf usw.) und deren Eigenschaften beschrieben. Einerseits ist es wichtig, das Metamodell in einen stabilen Zustand zu bringen (häufige Änderungen könnten sich später negativ auf den Entwicklungsprozess auswirken), andererseits sollte das Metamodell so entworfen werden, dass erforderliche Erweiterungen flexibel vorgenommen werden können.

Abbildung 2 zeigt das Metamodell als Bestandteil der 3. Transformationskette. Es ist die Grundlage für den Modellierungsstil (konkrete Syntax, z. B. durch Ableitung eines UML-Profils) und wird durch die Modell-zu-Modell-Transformation instanziiert. Die Generator-Templates beziehen sich auf das Metamodell und sind UML-frei.

Verhältnis generierter/manuell geschriebener Code

Es hängt stark von der Applikationsart und der eingesetzten Programmiersprache ab, wie viel Code generiert werden kann. Sicher gibt es Plattformen, auf denen 100 % Codegenerierung möglich und erwünscht ist (z. B. eingebettete Systeme).

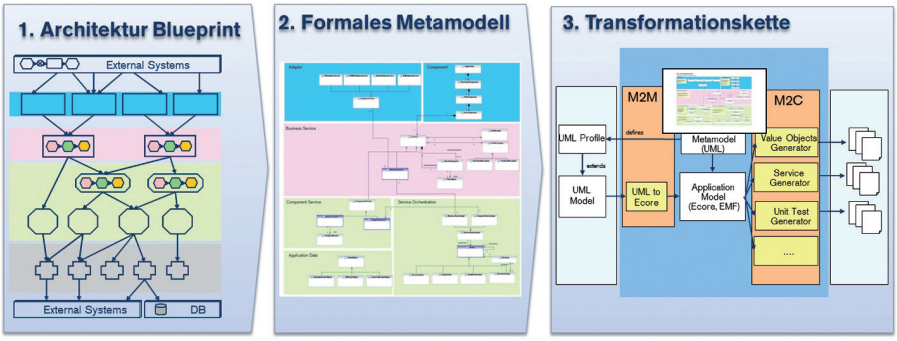

Abb. 2: Vom Architektur-Blueprint, über das Metamodell zur Transformationskette.

Schicht	Generierte Elemente	Manuell erstellte Elemente	Generierter Anteil
Adapter	JEE-Beans (MDB, SessionBeans)	Mapping-Klassen	50 %
Business-Services / Compound-Services Basis-Services	Java-Interfaces Abstrakte Klassen	Konkrete Klassen	40 %
Service-Orchestrierung	Java-Klassen	–	100 %
Datenmodell	JPA-Entitäten	–	100 %
Unit-Tests	JUnit-Tests (Infrastruktur-Code)	Testdaten-Instanzen	70 %

Tabelle 1: Auszug generierte Anteile.

Im Rahmen von Geschäftsapplikationen ist eine vollständige Generierung nicht das Ziel. Das würde die Komplexität der Business-Logik lediglich auf die Modellebene heben, aber nicht davon abstrahieren. Tabelle 1 listet generierte und manuell erstellte Anteile auf den jeweiligen Schichten auf.

Die hier gezeigten strukturellen Codebestandteile zählen eher zum MDD-Standard-Repertoire. Im Folgenden hebe ich die Service-Orchestrierung hervor, mit der wir im Projekt sehr gute Erfahrungen machen. Die Service-Orchestrierung stellt die Implementierung eines *Compound Service* dar (Definition aus [Jos08], häufig wird der Begriff *Micro-Flow* äquivalent verwendet). Dieser fasst durch eine Aufrufkette interne Basis-Services in einer Transaktion zusammen.

Abbildung 3 zeigt ein Beispiel für eine Service-Orchestrierung. Jede Aktion stellt einen Verarbeitungsschritt dar, in dem Basis- oder weitere Compound-Services aufgerufen werden (z. B. Daten validieren, Daten anreichern, Daten weitersenden). Der hierfür angepasste Modellierungsstil ist auf wenige Elemente beschränkt. Dadurch werden die Ablauf-Diagramme nicht für Verhaltensprogrammierung missbraucht, sondern bleiben auf einer fachlichen Ebene, während algorithmische Details im Code implementiert werden. Diagramme wie das in Abbildung 3 werden im Projekt auch von technologiefernen Rollen verwendet, um die fachliche Korrektheit zu verifizieren.

Die vollständige Generierung der Service-Orchestrierung ist ein gutes Beispiel dafür, wie die Dokumentation mit der Implementierung vollkommen synchron gehalten wird.

Frühe Modellvalidierung

Häufig bieten Generator-Frameworks eine integrierte Validierungs-API an, sodass während der Codegenerierung die Eingabe-Modelle auf Korrektheit geprüft werden können. Die Erfahrung hat gezeigt, dass eine frühere Validierung – nicht nur aus generierbaren Modellen, sondern auch aus Anforderungsspezifikationen und High-Level-Designs – sehr wichtig ist, um früh auf Unvollständigkeiten reagieren zu können oder Projektkonventionen (z. B. Namensgebung) rechtzeitig einzuhalten.

Kategorie	Beispielregel
Vollständigkeit	Ist Modellelement dokumentiert? Sind Typen an Datenattributen definiert? Sind Typen an Serviceparametern definiert?
Korrektheit	Referenzieren Datenattribute korrekte Typen (z. B. referenziert eine Entität kein Datentransferobjekt)? Verwenden Serviceparameter die korrekten Typen (Datentransferobjekte)? Ist eine Abhängigkeit zwischen zwei fachlichen Komponenten erlaubt?
Konvention	Namenskonventionen

Tabelle 2: Drei Kategorien der Validierungsregeln.

Sinnvoll sind Modellierungswerkzeuge, die „on-the fly“ validieren und dem Benutzer unmittelbar Feedback geben, wie man es von Programmier-Editoren kennt. Neben der automatischen syntaktischen Validierung sind semantische Reviews, z. B. über die Granularität der Services oder Datenstrukturen, für die Qualitätssicherung nicht wegzudenken und können durch Werkzeuge nicht ersetzt werden. Alle Validierungsregeln lassen sich in drei Kategorien einteilen (siehe Tabelle 2).

Eingeschränkter Modellierungsstil

Abhängig vom gewählten Modellierungswerkzeug werden bei einem gegebenen Metamodell Werkzeugerweiterungen automatisch erzeugt (z. B. angepasste Diagrammtypen, Modellierungspaletten und Kontextmenus).

Bewährt haben sich im Projekt speziell entwickelte Erweiterungen, um den Modellierer besser zu führen und das Design effizient nach Richtlinien zu erstellen. Dazu gehören – neben Modellvorlagen und -mustern – die unterstützte Navigation durch das Modell, die schnelle Suche über referenzierte Elemente oder die einfache Navigation vom Modellelement zur entsprechenden Quellcode-Stelle.

Die Diskussion über grafische und textuelle Modelle kann ich aus Platzgründen hier nicht weiter vertiefen. Wir machen mit grafischer Notation sehr gute Erfahrungen, gerade weil Abläufe, komplexe Datenmodelle und deren Beziehungen grafisch am besten erfassbar sind. Ganz typisch sind Meetings mit an die Wand gestrahlten Ablauf-Diagrammen einer Service-Orchestrierung, die dann mit Fachexperten Architekten und Entwicklern gemeinsam besprochen werden. Es gibt auch Verwendung für textuelle Modellierung, z. B. im GUI-Design oder als formale Geschäftsregel-Definition.

Continuous Integration

Aus der Projekterfahrung ist ein weiterer wichtiger Baustein die Einbindung von MDD in den kontinuierlichen Integrationsprozess (*Continuous Integration*). Diese ermöglicht nicht nur die ständige Überwachung der Modellqualität (*Validation*), sondern auch die Prüfung, ob Modell und Code synchron zueinander sind. Regelmäßig werden sämtliche Generatoren auf allen Modellen ausgeführt und anschlie-

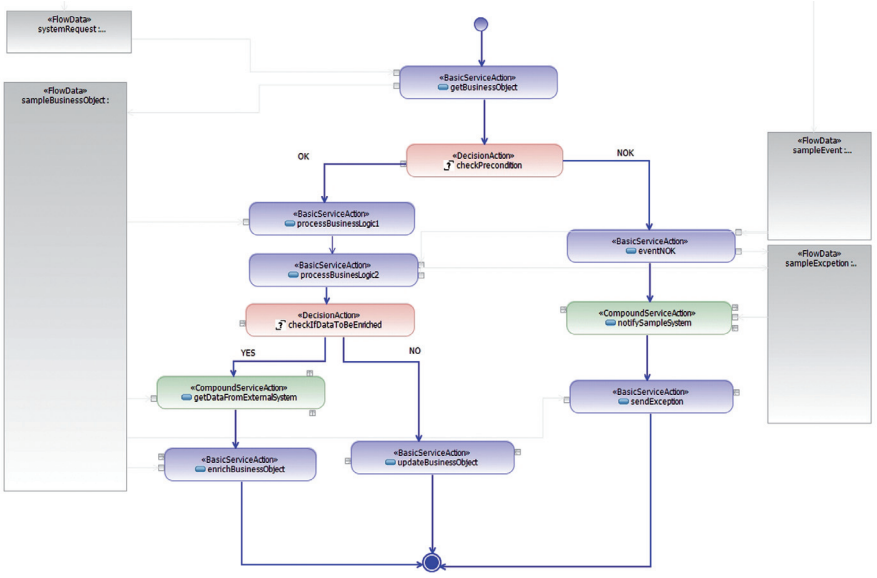

Abb. 3: Beispiel „Service-Orchestrierung“

ßend prüft der Compiler, ob die Implementierung zu den generierten Schnittstellen und abstrakten Klassen kompatibel ist.

Darüber hinaus berichten speziell entwickelte Reports über Codebestandteile, die ihren Ursprung nicht im Modell haben (auch solche können natürlich ihre Berechtigung haben). Zu empfehlen ist der Einsatz gängiger *Best Practices* über MDD und den Entwurf domänenspezifischer Sprachen (vgl. hierzu z. B. [Sta06], [Völ11]).

Organisatorische Bausteine

Das Bereitstellen einer funktionierenden Werkzeugkette führt noch nicht zur erfolgreichen Umsetzung der MDD-Methodik. Im Folgenden beschreibe ich die nicht weniger wichtigen soften Faktoren, die ein Projekt berücksichtigen sollte.

Rollenverteilung

Die Konzeption und Implementierung der MDD-Werkzeugkette wird in der Initiierungsphase des Projekts zeitgleich mit der Definition der Zielarchitektur durchgeführt. Ein dediziertes Team (siehe „Architekturumsetzungsteam“ in Abbildung 4), das aus Spezialisten der Ziel-Plattform und MDD-Experten besteht, ist hierfür verantwortlich. In enger Zusammenarbeit werden anhand von Prototypen die Design-Richtlinien und generierbare Teile definiert. Dieses Team ist außerdem für die Laufzeitaspekte und die Einhaltung der nichtfunktionalen Anforderungen verantwortlich.

Wichtig ist, dass das Architekturumsetzungsteam schnell auf Änderungsanforderungen und Erweiterungen reagieren kann und einen eigenen Release-Zyklus der MDD-Werkzeugkette und des Laufzeit-Frameworks pflegt, der mit dem Implementierungsteam und dessen Entwicklungsprozess abgestimmt ist.

Modellbasiertes Requirements-Engineering

Es bewährte sich im Projekt, das Requirements-Engineering nahezu vollständig in die MDD-Methode einzubeziehen. So können die Anforderungen an Schnittstellen mit Hilfe von konzeptionellen Datenmodellen (Business-Objekt-Modell) beschrieben werden. Auch die formale Beschreibung von Anwendungsfällen im Modell (z. B. Aktivitäten- oder Sequenzdiagramme) ermöglicht eine einheitliche und validierbare Spezifikation. Aus den Anforderungen abgeleitete Designmodelle referenzieren ihre Herkunft und stellen somit eine nahtlose Nachvollziehbarkeit (*Traceability*) her. Daher lassen sich in der Wartung Änderungsanforderungen schnell umsetzen, da mit Hilfe des Modells die zu ändernden Systembestandteile effizient identifiziert werden. Abbildung 5 zeigt dies an einem einfachen Beispiel: Die Änderungsanforderung „E-Mail senden“ wird in einem alternativen Ablauf des Anforderungsfalls beschrieben (1). Über die verlinkten Designmodelle ist die Stelle für die Anpassung effizient lokalisiert und wird in der Service-Orchestrierung ergänzt (2). Die Code-Generierung sorgt dafür, dass für den Entwickler die Schablone vorgeneriert wird, in der der eigentliche Basis-Service für den E-Mail-Versand aufgerufen wird (3).

Das Testteam profitiert von den im Modell definierten Anforderungen: Mit Hilfe von speziellen Reports lassen sich die zu testenden Bestandteile aus dem Modell automatisch beziehen, z. B. durch das Auslesen aller fachlichen Pfade eines Anforderungsfalls.

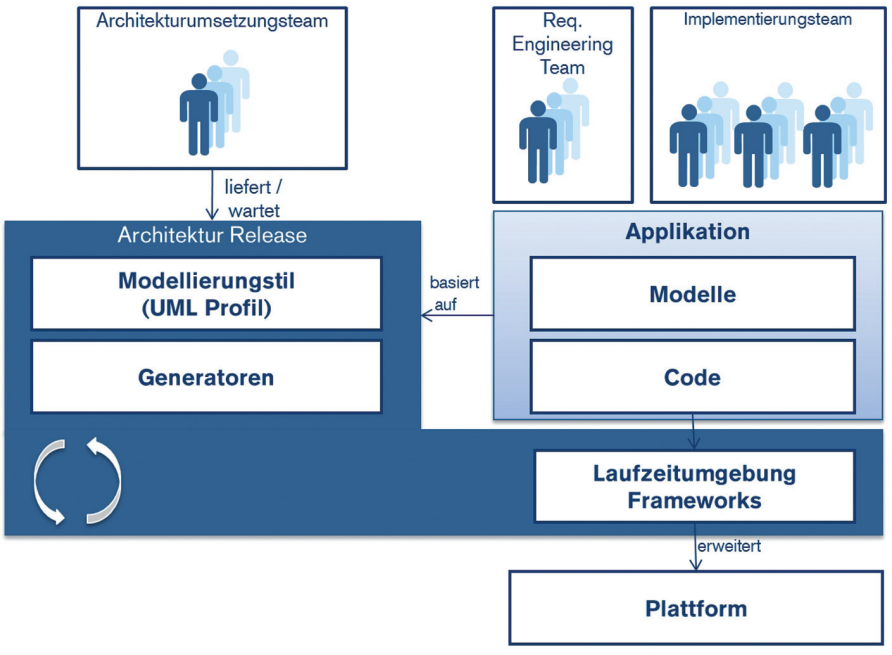

Abb. 4: Architekturumsetzungsteam.

Skalierbarkeit des MDD-Ansatzes
Um den modellzentrierten Ansatz in einem größeren Team auszurollen, muss zu Beginn (d. h. bevor das Team die Zielgröße erreicht) feststehen, wie mit dem Modell gearbeitet wird. Auch wenn Modellierungswerkzeuge generell Teamwork unterstützen, sollten Fragen zu Teamwork, Locking-Granularität, Modell-Merging etc. geklärt werden.

Im konkreten Projekt wurde das Modell nach fachlichen Komponenten partitioniert und die Verantwortlichkeiten wurden auf kleine Teams verteilt. Dadurch wird die Wahrscheinlichkeit von Modellkonflikten durch konkurrierende Modifikationen minimiert. Darüber hinaus können Zugriffsberechtigungen erzwingen, welche Teams auf welchen Modellpartitionen arbeiten. Letzterer Mechanismus wird im Projekt lediglich auf sehr zentralen Modellpartitionen (z. B. Datenmodell) eingesetzt. Das Mergen und gelegentliche Auflösen von Konflikten wird mit Werkzeugunterstützung vorgenommen. Durch ein gemeinsam genutztes (Modell-) Repository hat das gesamte Projektteam ständig eine gemeinsame Sicht auf Anforderungen, Design und Code.

MDD und agile Prozesse

Kritiker des MDD-Ansatzes wenden ein, dass man mit MDD an Flexibilität verliert. Diesen Eindruck mag das von der OMG beschriebene MDA-Vorgehen vermitteln, das falsch als Wasserfall-Modell interpretiert werden kann (*Computation Independent Model -> Platform Independent Model -> Platform Specific Model*, vgl. [OMG12]).

Im Folgenden beschreibe ich, wie sich agile und modellgetriebene Ansätze gewinnbringend ergänzen, aber auch, wo wesentliche Unterschiede bestehen.

Iteratives Vorgehen mit kleinen Teams
Modelle stellen kein Hindernis dar, um ein agiles Vorgehen einzusetzen. Die Voraussetzung wird zuerst durch die Projektorganisation geschaffen. Ein großes Team muss in mehrere kleine unterteilt werden, denen die Lieferverantwortung für eine Teilkomponente übertragen wird.

Die kleinen Teams bestehen aus diversen Rollen (Requirements-Engineers und Entwickler), sodass Kommunikationswege optimiert werden. Innerhalb dieser Teams werden agile Methoden angewandt, z. B. eigenständige Sprint-Planung und regelmäßige Scrum-Meetings. Das funktioniert bei gegebener Infrastruktur auch über Zeitzonen hinweg.

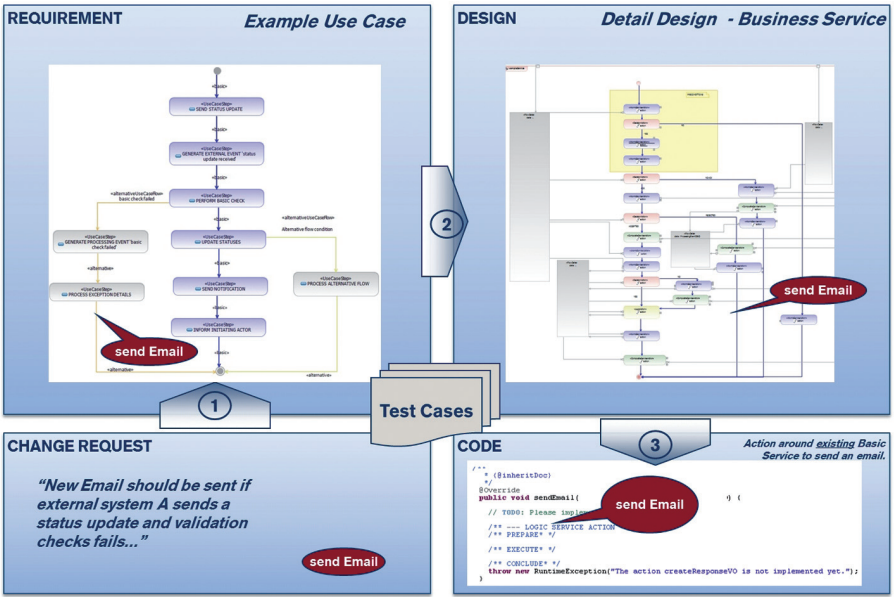

Abb. 5: Von der Änderungsanforderung, über das Modell zum Code.

Geliefert werden fachliche abgeschlossene Anwendungsfälle, die jeweils einer Komponente zugewiesen werden. Es ist nicht auszuschließen, dass mehrere Teams gemeinsame Komponenten modifizieren, z. B. Datenmodelle oder Basis-Services. Diese zentralen Komponenten werden durch dedizierte Teams gewartet, die Änderungsanforderungen entgegennehmen und diese kontrolliert umsetzen.

Restrukturierung (Refactoring)
Refactorings auf Codeebene werden optimal durch Entwicklungswerkzeuge unterstützt. Bei Verwendung des MDD-Ansatzes kann Mehraufwand entstehen, weil die Synchronisation von Modellen und Code ganzheitlich betrachtet werden muss. Allerdings gibt es technische Lösungen, um den Mehraufwand zu minimieren: Für mögliche Änderungen helfen Refactoring-Skripte, wie sie z. B. in „Eclipse JDT“ (vgl. [Ecl]) unterstützt werden. Falls Massenänderungen (Verschieben, Umbenennen, etc.) erforderlich sind, leiten wir diese Skripte automatisch aus dem Modell ab und führen sie auf der Codeebene aus.

Wert der Dokumentation
Beim Thema Dokumentation besteht der wesentliche Unterschied der Ansätze. Langlebige aktuelle Dokumentation widerspricht dem agilen Prinzip „der Code ist die Dokumentation“ (vgl. [Fow05]). Dieses Prinzip ist nicht anwendbar, wenn es sich um eine nicht mehr gelehrte Sprache handelt und nur wenig Experten-Know-how vorhanden ist. Spätestens in ein paar Jahren, wenn vielleicht auch Java nicht mehr State-of-the-Art ist, muss diese Aussage hinterfragt werden.

Fazit

Modellgetriebene Ansätze (auch in anderen Varianten als hier dargestellt) sind aus größeren Projekten nicht wegzudenken. Bei der Berücksichtigung von bewährten Praktiken ist die Einführung von MDD aus technischer Sicht unproblematisch. Das alleinige Bereitstellen einer MDD-Werkzeugkette reicht allerdings häufig nicht aus. Wichtig sind die ganzheitliche Betrachtung von MDD und die Adressierung softer Faktoren, wie Organisation und Prozesse.

Ein wesentlicher Beitrag zum Erfolg von MDD im konkreten Projekt war die holistische Umsetzung, d. h. die Einbeziehung aller Mitarbeiter des Projekts als Nutzer der Modelle. So werden die Vorteile von MDD nicht nur für eine „elitäre“ Gruppe sichtbar, sondern für alle Projektbeteiligten. Vergleichsschätzungen über die fachliche Komplexität haben gezeigt, dass im Projekt für die Implementierung neuer Anforderung die Effizienz um ca. 30 bis 40 % gesteigert wurde.

Beim Skalieren über die Teamgröße wurden die meisten Lektionen gelernt. In einem wachsenden Team bedarf es spezieller Rollen, die das Wissen als Coachs ebenso weitergeben wie gute Dokumentation über

die Installation und Anwendung der MDD-Werkzeugkette (z. B. durch eine Referenzapplikation). Die Einarbeitung neuer Mitarbeiter ist nicht zu vernachlässigen. Haben aber Entwickler die Zusammenhänge von Modell, Generatoren und Code verstanden, können sie sich schnell produktiv einbringen.

Nicht nur Teams ändern sich, sondern auch die Modellgröße und die Art, wie auf das Modell zugegriffen wird. In dem hier vorgestellten Projekt ging die erste Festlegung der Modellpartitionierung nicht weit genug und wurde später korrigiert, nachdem die Teilkomponenten und Teamverantwortlichkeiten feststanden.

Auch bei der Generatorentwicklung gab es gelernte Lektionen. Beispielsweise hat die Modellierung von Testdaten-Instanzen zu Beginn sehr gut funktioniert und ermöglichte die vollständige Generierung von Unit-Tests. Mit wachsendem Datenmodell erwies sich aber dieser Ansatz als zeitaufwändig. Nach einer Korrektur nimmt der Unit-Testgenerator nun dem Entwickler immer noch die Arbeit für das Erstellen von Unit-Tests ab, die Dateninstanzen werden aber im Code oder in einer Datenbank hinterlegt. Generell sollte es das Ziel der Codegenerierung sein, möglichst sinnvoll, statt möglichst viel zu generieren.

Nicht jedes Projekt eignet sich für MDD, z. B. wenn die Projektgröße den initialen Aufwand für die Erstellung der MDD-Werkzeugkette nicht rechtfertigt. Hat aber eine Organisation den MDD-Prozess etabliert, so ist die Wahrscheinlichkeit groß, dass sich verwandte Projekte mit einem ähnlichem Architekturstil finden. Dann kann ein Unternehmen durch Software-Industrialisierung über Projektgrenzen hinaus profitieren, indem bereits implementierte MDD-Werkzeugketten in Projekten wiederverwendet werden. Eine Weiterentwicklung des klassischen MDD-Vorgehens ist die Kombination von MDD mit agilen Vorgehensmodellen. In diesem Artikel habe ich gezeigt, dass diese Methoden sich nicht ausschließen und ein Projekt von beiden Ansätzen profitieren kann. ■

Literatur & Links

[Ecl] Eclipse, Eclipse Java development tools (JDT), siehe: eclipse.org/jdt

[Fow05] M. Fowler, CodeAsDocumentation, 2005, siehe: martinfowler.com/bliki/CodeAsDocumentation.html

[Jos08] N. Josuttis, SOA in der Praxis, dpunkt.verlag 2008, siehe: soa-in-der-praxis.de/soa-glossar.html

[OMG12] Object Management Group (OMG), MDA Specifications, 2012, siehe : omg.org/mda/specs.htm

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

[Völ11] M. Völter, DSL Best Practices, 2011, siehe: voelter.de/data/pub/DSLBestPractices-2011Update.pdf

[Zie10] S. Ziegler et. al., Industrielle Software Entwicklung, BITKOM 2010, siehe: bitkom.org/files/documents/Industrielle_Softwareentwicklung_web.pdf