Forms2ADF mal anders: Wie aus einer Oracle-Vision Praxis wird

Markus Klenke, TEAM GmbH; Marvin Grieger, s-lab Universität Paderborn; Wolf G. Beckmann, TEAM GmbH

Der von Oracle aufgezeigte Weg, Forms-Applikationen zu ADF zu migrieren, wurde schon in diversen White-Papers und Präsentationen erläutert. Dennoch wird in der Praxis mit einfachen „1:1"-Migrationstools und viel Handarbeit migriert. Wie ein modellgetriebener Migrationsansatz in der Praxis aussieht und welche Vorteile er gegenüber den konventionellen Migrationsstrategien bietet, wurde von TEAM in einem ZIM-geförderten Projekt in Zusammenarbeit mit der Universität Paderborn ermittelt. Das Ergebnis ist ein Werkzeug zur semi-automatisierten Migration, mit dem native ADF-Applikationen aus Forms-Anwendungen erstellt werden.

Beginnen wir mit einem kleinen Beispiel. Es besteht eine Suchmaske mit einigen Eingabefeldern und einem Anzeige-Button. Bei dessen Anklicken wird eine unter der Suchmaske befindliche Tabelle gefüllt. Die einfachste Art, diese Funktionalität zu realisieren, besteht darin, jedes Eingabefeld als Einschränkung („Bind Variable") für die Datenbank-Abfrage am Ausgabeblock zu verwenden.

Dieser Dialog soll nun migriert werden. Eine „1:1"-Migration würde vorsehen, für die Datenbank-Abfrage die benötigten Business-Service-Elemente zu erstellen, für die Eingabefelder äquivalente Input-Elemente zu erzeugen und über eine Aktion die Abfrage ausführen zu lassen. Dies ist ein valider Ansatz, doch muss man sich die Frage stellen: Ist das eine optimale Überführungsstrategie? Vielleicht, wenn ausschließlich der Technologiewechsel im Vordergrund steht. Wenn jedoch eine echte Modernisierung erfolgen soll, hätten wir unser Ziel verfehlt.

ADF bietet durch seine moderne Technologie insbesondere eine dynamisch generierte Suchkomponente, die sich für diese Funktionalität viel besser eignen würde, da sich zusätzlich zur Such-Funktionalität unter anderem vom Benutzer eingegebene Suchen speichern lassen. Somit haben wir einen viel benutzerorientierteren Dialog geschaffen, die Funktionalität im Kern aber beibehalten. Wir haben den Dialog modernisiert, nicht nur migriert.

Auch diesen Schritt könnte man automatisieren. Jedoch ist an dieser Stelle das explizite Wissen vonnöten, dass es sich um einen Suchdialog handelt, denn wir wollen schließlich nicht alle Dialoge mit Eingabefeldern auf Suchmasken abbilden. Ist damit die Idee einer Migration mit Modernisierung von Forms zu ADF gestorben? Die semiautomatisierte Migration beziehungsweise Modernisierung befasst sich im Kern mit genau dieser Fragestellung und liefert aufgrund der manuellen Interaktionsmöglichkeiten eine Antwort auf diese Problemstellung: Nein, es ergibt durchaus Sinn, eine Modernisierung von Forms mit ADF durchzuführen, sofern die Migrationsgründe es zulassen *(siehe Abbildung 1)*.

Warum werden in der Praxis trotz dieser Problemstellung „1:1"-Migrationen

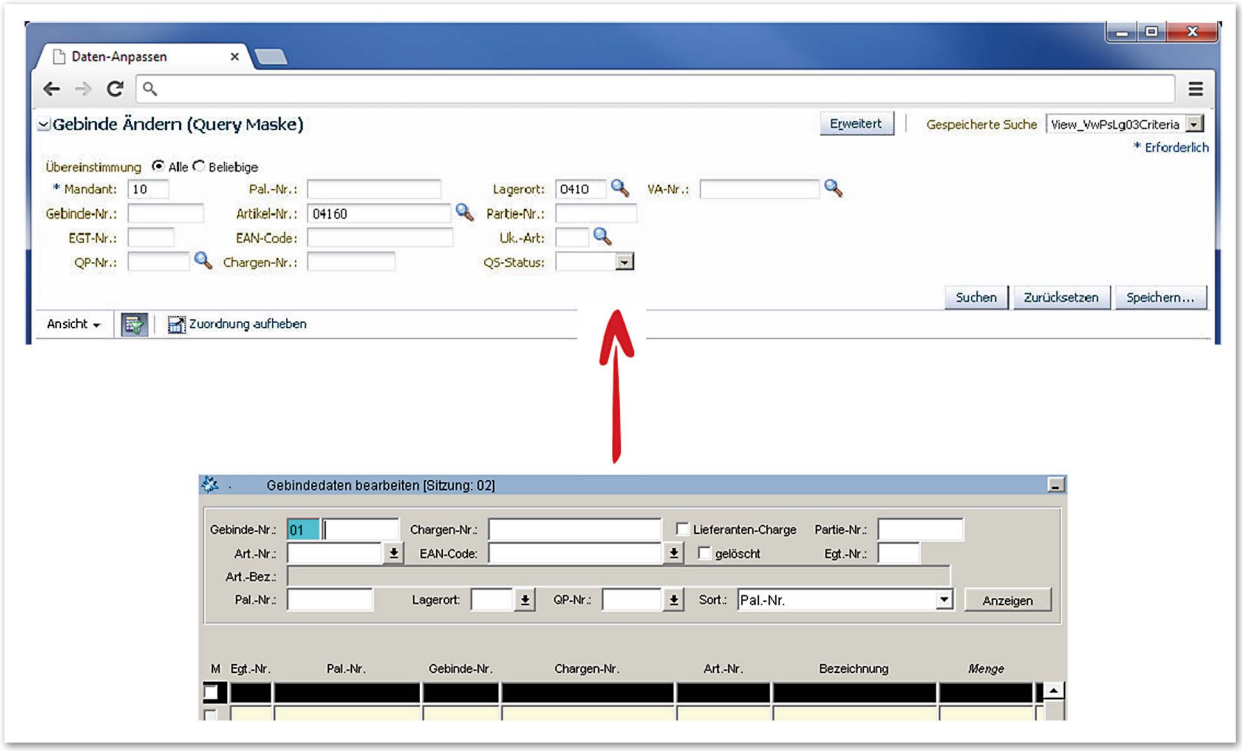

Abbildung 1: Einzelne Forms-Items zur Suche (unten) werden zur individualisierbaren ADF-Suchkomponente (oben)

durchgeführt? Weil Programme zur automatisierten „1:1"-Migration mit deutlich weniger Aufwand entwickelt werden können als Tools für die modellgetriebene Migration von Forms nach ADF. Bei einer „1:1"-Migration wird für jedes Element aus dem Quellsystem eine Abbildung auf ein Element in der neuen Umgebung definiert. Es scheint also, dass eine automatisierte Migration von Forms nach ADF relativ einfach realisierbar ist.

Warum tun sich so viele Projekte schwer mit einer Migration? Einen Effekt erkennt man schon an dem einfachen Beispiel aus der Einleitung: Syntaktisch gleiche Elemente sind auf funktional unterschiedliche Komponenten abzubilden. Für direkte Transformationen stellt dieses Unterfangen schon eine sehr komplexe Herausforderung dar. Sollen hingegen Konzepte überführt werden, die in der Quell-Plattform gänzlich anders realisiert sind (Navigation, Templates) oder gar nicht zur Verfügung gestanden haben (hierarchische Applikationsstruktur), so können diese Anforderungen schlicht und einfach nicht mit einer „1:1"-Migration durchgeführt werden, da eine Abbildungsvorschrift fehlt.

Diese fehlenden Vorschriften gehen aus dem starken architektonischen Unterschied zwischen Forms und ADF hervor und stellen daher einen – gerade bei großen Applikationen – nicht unerheblichen Teil der Migration dar. Die Qualität der Überführung dieser Konzepte steht in direkter Korrelation mit der Qualität der resultierenden ADF-Applikation. Aus diesem Grund werden in der Praxis entweder wenige Quelldateien automatisiert überführt oder es ist eine erhebliche manuelle Nachbearbeitung erforderlich.

Ein Ei gleicht nicht dem anderen

Wie erwähnt, sind Forms und ADF architektonisch grundverschieden. Während Forms monolithisch geprägt ist, also eine sehr enge Verzahnung zwischen Oberfläche und Datenbank anstrebt, ist die Reali-

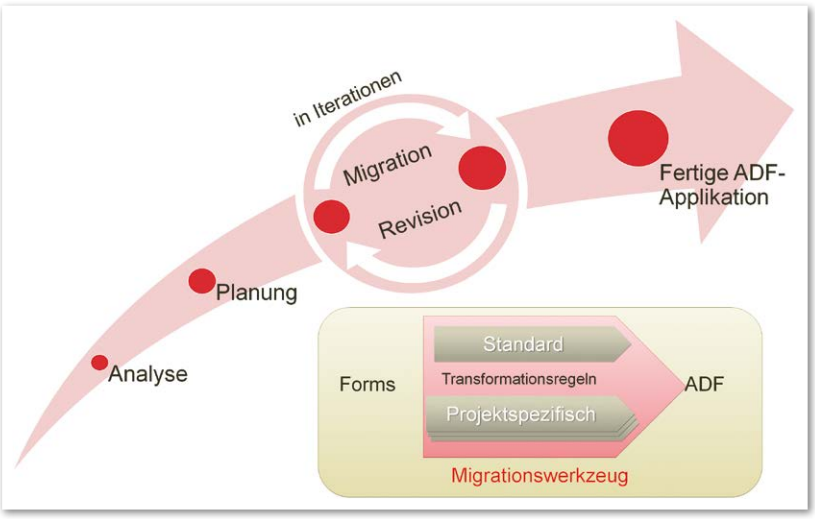

Abbildung 2: Migrationsprozess eines semi-automatisierten Ansatzes

sierung in ADF eine gegenläufige: Die Oberfläche einer Applikation soll nur zur Anzeige und Interaktionsmöglichkeit des Benutzers dienen, ist also in der Theorie strikt von jeglicher Logik zu trennen. Auch die Business-Service-Schichten der beiden Plattformen unterscheiden sich vehement voneinander.

Noch schwerwiegender fällt allerdings folgender Punkt ins Gewicht: Die Denkweise, in der auf beiden Systemen entwickelt wird, ist unterschiedlich. Während Forms-Applikationen dialoglastig und kompakt entwickelt werden, ist ADF prozessorientiert und jede Applikation kann als wiederverwendbare Komponente in einem höher liegenden Prozess gesehen werden.

Zudem bietet ADF zahlreiche Schnittstellen und Möglichkeiten zur Individualisierung und Erweiterbarkeit. Alle generierten Komponenten sollten das Potenzial der Wiederverwendung, die sie durch die Plattform mitbringen, auch ausnutzen. Schließlich ist das der enorme Vorteil von ADF gegenüber Forms; Daten-Anbindungen werden zentral als Basiseinheit gebildet, Labels von Texten werden in Ressourcen ausgelagert und Oberflächen sind benutzerspezifisch anpassbar.

Zusätzlich zur architektonischen Differenz der Plattformen kommen die unterschiedlichen Entwicklungsmuster, die sich in den vielen Jahren der Forms-Entwicklung in den Quelldateien niedergeschrieben haben. Kaum eine Forms-Applikation verhält sich wie eine äquivalente Applikation in einem anderen Projektumfeld. Daher ist es für die Migration immens wichtig, ein Verständnis für die Funktionalitäten und Fokussierungen der Altanwendung aufzubauen. Insbesondere ist es wichtig, die Applikation nicht einfach nach „Schema F" auf die neue Plattform zu überführen, sondern angepasste Transformationsregeln darüber aufzustellen, wie die Migration einzelner Komponenten durchgeführt werden soll. Dazu bietet es sich oft an, nicht nur ein Objekt einzeln zu behandeln, sondern mehrere Objekte zu einer konzeptionellen Komponente zusammenzufassen und gemeinsam zu untersuchen oder Objekte vom gleichen Typ als unterschiedliche Stereotype zu betrachten (etwa unterschiedliche Typen von Items). Kehren wir noch einmal zum Beispiel zurück, so erkennen wir dieses Schema selbst an den einfachsten Anwendungsfällen: Mehrere Eingabefelder und ein Aktionselement werden zu einer Suchkomponente zusammengeführt, da sie ihre Funktionalität nur im Zusammenspiel ausführen können.

Wie lautet also der Plan?

Der Schlüssel zum Erfolg bei einer erfolgreichen Software-Migration (automatisiert oder manuell) ist ein gut geplanter und durchgeführter Migrationsprozess. Eine Automatisierung ohne Verständnis ist genauso wenig erstrebenswert wie eine vollständige manuelle Re-Implementierung. Wie können nun also diese beiden Extrema miteinander verschmolzen und somit das Beste der beiden Wege extrahiert werden? Ein sehr effektiver Ansatz ist der Migrationsprozess, der in *Abbildung 2* beschrieben ist.

Das Migrationsprojekt beginnt mit einer Analyse-Phase. Dabei bewerten Migrationsexperten die Architektur und die Migrationskomplexität der Alt-Anwendung sowohl mithilfe von Analyse-Werkzeugen, mit den Entwicklern der Applikation, als auch manuell. Unter Beachtung der zusätzlich erfassten Projektziele und des Projektkontexts wird als Ergebnis eine vorläufige Migrationsstrategie für die Applikation erstellt.

In der anschließenden Planungs-Phase erfolgen tiefergehende Analysen der Alt-Anwendung nicht nur bezüglich der Architektur, sondern auch bezüglich der vorhandenen Entwicklungsmuster. Basierend auf diesen Erkenntnissen wird durch ADF-Experten eine detaillierte Ziel-Architektur entworfen und ein Migrationspfad, also Abbildungen zwischen Alt- und Ziel-Applikation, festgelegt.

Aufgrund des semi-automatisierten, modellgetriebenen Vorgehens kann die Quell-Applikation sehr stark umstrukturiert und trotzdem zu einem hohen Grad automatisiert migriert werden. Anschließend werden Migrationspakete gebildet, um eine iterative Überführung zu ermöglichen, und eine Umstellungsstrategie (wie Parallelbetrieb oder schrittweise Ablösung) festgelegt, um eine Unterbrechung des laufenden Betriebs der Applikation zu vermeiden. Das Ergebnis dieser Phase ist folglich eine angepasste und verfeinerte Migrationsstrategie. Parallel dazu werden die eigentliche Migration vorbereitet, also die Infrastruktur auf-

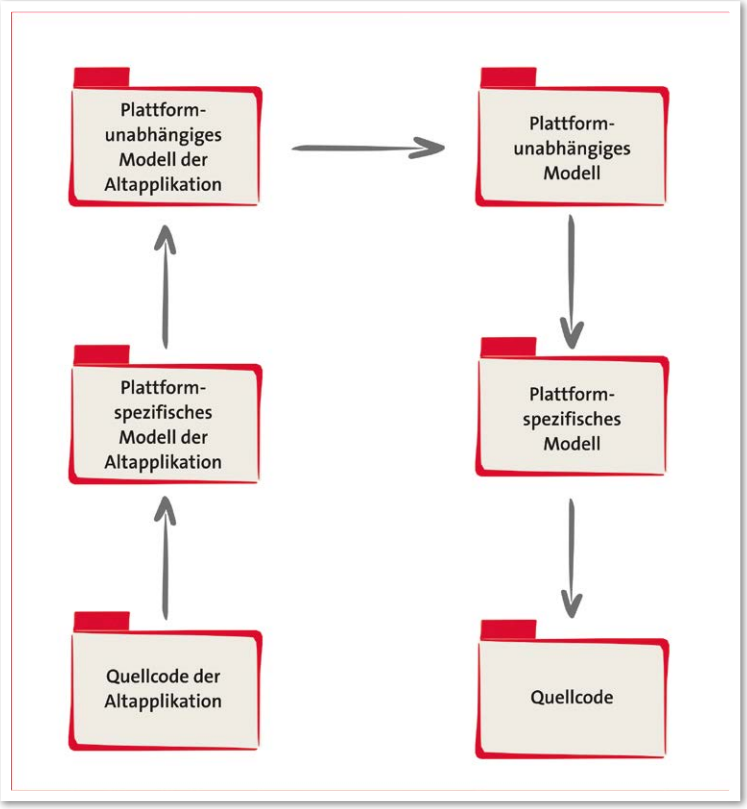

Abbildung 3:Technische Darstellung des Migrationsansatzes über Abstraktion

gebaut, und gegebenenfalls die ehemaligen Forms-Entwickler in ADF geschult.

Jetzt beginnt die eigentliche Migration in Iterationen. Diese läuft immer in vier Schritten ab:

1. Implementierung von projektspezifischen Regeln für die Erkennung und das Umsetzen von Mustern aus der Altanwendung nach ADF
2. Automatisches Migrieren eines der geplanten Module von Forms nach ADF
3. Manuelles Fertigstellen des Moduls
4. Revision der Fertigstellungsphase, um zu ermitteln, welche weiteren Muster zusätzlich automatisiert umgesetzt werden können. Diese werden dann im ersten Schritt der nächsten Iteration in das Migrationswerkzeug eingearbeitet.

Gerade bei der ersten Migrationsiteration werden noch viele weitere Muster gefunden, wodurch von Iteration zu Iteration immer weniger manuell fertiggestellt werden muss. Durch das iterative Vorgehen werden dabei sehr früh Teilergebnisse sichtbar. Diese Vorgehensweise hat noch einen Seiteneffekt: Da typischerweise die ehemaligen Forms-Entwickler die einzelnen Module fertigstellen, werden sie sanft an die neue Entwicklungsumgebung herangeführt.

Modellgetriebene, semi-automatisierte Migration in Softwareform

Aufgrund der oben genannten Unterschiedlichkeit von Applikationen ist es für eine Migration einer Software unerlässlich, Anpassungsmöglichkeiten und Erweiterungsmechanismen für das Migrationswerkzeug bereitzustellen. Die von TEAM entwickelte Implementierung eines modellgetriebenen, semi-automatisierten Migrationsmodells charakterisiert sich besonders durch den hohen Abstraktionslevel und die damit verbundenen Restrukturierungsmöglichkeiten *(siehe Abbildung 3)*.

Entgegen der bekannten „1:1"-Migrationsstrategien liegt der Fokus nicht auf der Transformation der einzelnen Elemente wie Eingabefelder oder Buttons, sondern auf dem Verständnis der Funktionalität der Alt-Applikation. Sind Strukturen der Applikation verstanden worden, die eine komplexere Funktionalität realisieren (beispielsweise das Suchfeld), werden diese extrahiert und durch ein Plattformunabhängiges Modell repräsentiert. Es dient zunächst als Zwischenrepräsentation der Alt-Applikation und wird im Laufe des Migrationszyklus durch Restrukturierungen und Funktionsanreicherungen zur gewünschten Ziel-Applikation transformiert. Diese wird nun mithilfe projektspezifischer Transformationsregeln zu einer ADF-Applikation generiert, die zum einen die Best-Practices von Oracle implementiert, zum anderen die während des Migrationsprojekts entstandenen Richtlinien einhält und Komponenten beinhaltet. Somit entsteht am Ende eines durchgeführten Migrationszyklus eine Applikation, die den projektspezifischen Migrationsanforderungen genügt.

Ein Blick hinter die Fassade

Im Folgenden sind die einzelnen Schritte aus *Abbildung 3* aus technischer Sicht beleuchtet. Als technologische Grundlage für die genutzten Modelle bietet sich mit EMF ein flexibles und erweiterbares Framework an, das den Anforderungen und dem notwendigen Individualismus eines Migrationsprojekts gewachsen ist. Um die Modelle zu persistieren und eine performante Schnittstelle zwischen Code und Modell herzustellen, werden die Modelle innerhalb eines CDO-Repository gespeichert, dessen Grundlage wiederum eine relationale Datenbank ist. Somit ist es möglich, mit bekannten SQL-Queries gezielt auf Modell-Elemente aus dem Repository zuzugreifen.

Zunächst werden alle notwendigen Quelldateien (FMB, PLL, MMB, OLB) sowie externe Komponenten (Datenbank, Forms-Plattform) in ein eigens erstelltes Modell geladen. Dieses wurde auf Basis

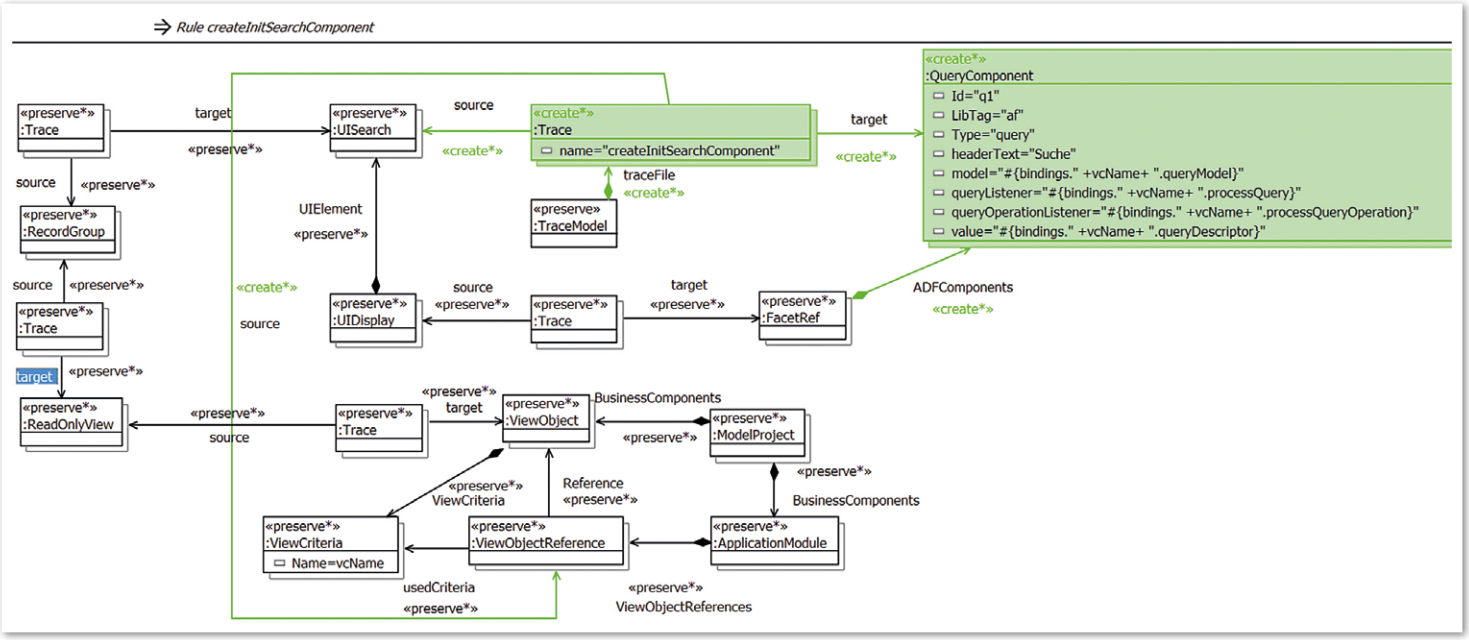

Abbildung 4: Transformationsregeln zur Erstellung einer Suchmaske

der „jdapi“-Schnittstelle des Forms Builder sowie aus den Definitionen der Sprache SQL sowie PL/SQL und des Oracle Data Dictionary erstellt. Basis des Plattform-unabhängigen Modells ist eine Erweiterung des OMG-Standards „Knowledge Discovery Metamodel“. Es erlaubt, eine Applikation in einem standardisierten Format Plattform-unabhängig zu repräsentieren. Da das Meta-Modell eine Beschreibungssprache für allgemeine Software-Applikationen ist, ist das Werkzeug flexibel genug, um auf Wünsche und Zielplattform-Einschränkungen beziehungsweise -Erweiterungen zu reagieren und diese in die Migration mit einfließen zu lassen.

Die Beschreibung der ADF-Modelle wird durch ein vollständig selbstentwickeltes ADF-Meta-Modell realisiert, das wiederum auf Best-Practices beruht. Der Fokus dieses Meta-Modells liegt auf Templates und wiederverwendbaren Komponenten, entspricht dabei allerdings immer den ADF-Standards. Somit können sich ADF-Neulinge sowie erfahrene ADF-Entwickler sofort in die neue Applikation einfinden.

Um aus dem ADF-Modell Quellcode für die ADF-Applikation zu generieren, kommt das Acceleo-Framework zum Einsatz. Es bietet eine einfache Möglichkeit, mithilfe von textuellen Templates (ähnlich wie bei Velocity und Texen) Quellcode zu generieren. Der Vorteil dieses Frameworks liegt allerdings in der direkten Implementierung einer EMF-Schnittstelle, durch die beim Erstellen der Templates eine direkte Einbindung des Modells möglich ist und daher bei fehlerhaften beziehungsweise nicht validen Eingaben ein zugehöriger Editor vor unerlaubten Aktionen warnt.

Um Modelle von einem Zustand in einen anderen zu transformieren, wird die Modell-Transformationssprache „Henshin“ eingesetzt. Diese ist im Kontext eines Eclipse-Incubation-Programms entstanden und bietet dem begleitenden Migrationsexperten die Möglichkeit, Transformationsregeln deklarativ und visuell zu erstellen. Die direkte Anbindung an die genutzten Meta-Modelle schränkt ihn sinnvoll ein, sodass valide Modelle immer auf valide Modelle abgebildet und nur durch das Migrationsprojekt erlaubte Elemente erstellt werden. Da die Transformationsregeln durch Henshin aus technischer Sicht ebenfalls Modelle sind, können diese zusätzlich in das Modell-Repository gespeichert werden. Somit wird kein Migrationsprojekt bei null gestartet, sondern kann bei Bedarf und Wunsch auf zuvor erstellten Transformationsregeln basieren *(siehe Abbildung 4)*.

Fazit

Was ist letztendlich der Vorteil, der durch eine semi-automatisierte Migration entsteht? Zunächst einmal scheint es durch die komplexen und zeitaufwändigen Vorbereitungs- und Verständnis-Schritte einen enormen Mehraufwand im Vergleich zu einer „1:1“-Migration zu geben. Dieser initiale Mehraufwand wandelt sich allerdings im Laufe des Migrationsprojekts in eine Beschleunigung der Arbeitspaket-Bearbeitung. Je präziser und detaillierter die Lösung für das erste Migrationspaket ist, desto größer wird der Gewinn für die folgenden Migrationspakete ausfallen. Darüber hinaus werden durch die zyklische Arbeitsweise die späteren Migrationspakte ebenfalls durch neue Features oder weitere Automatismen geschmälert, was letztendlich zu einer drastischen Kürzung des Migrationsaufwands führt.

Nicht nur das: Im Gegensatz zu einer vollautomatisierten „1:1“-Migration, die den Fokus auf die unmittelbare Ausführung des generierten Quellcodes richtet, wobei allerdings die Individualisierung auf der Strecke bleibt, erhält man zum Abschluss eines semi-automatisierten Migrationsprojekts genau die Ausprägungen an ADF-Software, die für die Ausführung der Funktionalitäten der Alt-Applikation innerhalb der neuen Umgebung sinnvoll sind; Templates, Basis-Komponenten und Applikationsrahmen inklusive.

Durch die vielen manuellen Interaktionsmöglichkeiten innerhalb des Migrationszyklus lässt sich der Weg zu einer Applikation, die man als ADF-Entwickler erwartet, sehr gut ebnen. Jeder Entwickler, der sich in der Zielumgebung auskennt, wird für die Entscheidung eines semi-automatisierten Prozesses dankbar sein. Auch der Einstieg für Entwickler des Quellsystems ist durch die Konzeption nahe an einer Referenz-Architektur einfacher.

Durch die diversen manuellen Interaktionsmöglichkeiten kann bereits in den ersten Phasen der Migration an wiederverwendbaren Komponenten, möglichen Benutzerprozessen oder Code-Libraries gearbeitet werden, die später in der Generation nutzbar sind. Genau diese Features sorgen dafür, dass am Ende einer semi-automatisierten Migration wirklich eine Applikation erzeugt wird, die von einer manuell erstellten ADF-Applikation kaum mehr zu unterscheiden ist. Das sollte schließlich das Ziel eines jeden Automatismus sein.

Wolf G. Beckmann
wb@team-pb.de

Markus Klenke
mke@team-pb.de

Marvin Grieger
mgrieger@s-lab.upb.de