Jürgen Krey
(E-Mail: juergen.krey@msg-systems.com)
ist Chefarchitekt im Geschäftsbereich Travel & Logistics bei msg systems. Er verfolgt seit vielen Jahren die Ansätze, mit denen die Entwicklung in Java beschleunigt werden kann, darunter MDD, MDA und AOP.

Dr. Vladimir Rubin
(E-Mail: vladimir.rubin@msg-systems.com)
ist Senior IT-Consultant im Geschäftsbereich Travel & Logistics bei msg systems. Seine Schwerpunkte liegen im Bereich modellgetriebene Softwareentwicklung, Geschäftsprozessmodellierung und Java-Architektur.

Sebastian Rose
(E-Mail: sebastian.rose@msg-systems.com)
ist IT-Consultant im Geschäftsbereich Travel & Logistics bei msg systems. Seine Schwerpunkte liegen im Bereich der modellgetriebenen Softwareentwicklung.

Modellgetriebene Entwicklung einer Rich Client-Anwendung – ein Erfahrungsbericht

Wir haben den clientseitigen Teil eines Großprojekts als Wartungsaufgabe übernommen und konnten Stärken, Schwächen und Grenzen eines umfangreichen Generierungsansatzes kennenlernen. Der ursprüngliche Ansatz wurde vereinfacht und mit sorgfältig ausgewählten modellgetriebenen Entwicklungs- und Framework-Konzepten so angepasst, dass die Software besser wartbar wurde und weiterentwickelt werden kann. Neben den konkreten Lösungen geben wir mit den Erfahrungen aus unserem Projekt Hinweise zum modellgetriebenen Vorgehen.

Viel zu generieren, hilft auch viel?

Mit zunehmender Projektgröße entdeckt man in Softwareprojekten Anforderungen und Muster, die an unterschiedlichen Stellen auftreten und gleichartig behandelt werden können. Solche Gemeinsamkeiten lassen sich schon während der Spezifikations- oder der Entwurfsphase herausarbeiten und modellieren – unabhängig von der verwendeten Programmiersprache. Um sie in ausführbare Programme zu überführen, bieten sich unterschiedliche Techniken an: von selbstentwickelten Code-Generatoren über OpenSource Frameworks bis hin zu Modellsprachen und -interpreten sowie ambitionierten Ansätzen, wie Model Driven Architecture (MDA).

Als wir im April 2010 die Client-Architektur einer großen Java-basierten Anwendung zur Wartung und Weiterentwicklung übernahmen, fanden wir einen hochgradig modellgetriebenen Ansatz zur Softwareentwicklung (Model Driven Development, MDD) vor. Im Vordergrund stand der Anspruch, das System so weit wie möglich in UML zu modellieren und daraus möglichst viel Code generieren zu lassen.

Zum Einsatz kamen MagicDraw als Modellierungs- und openArchitectureWare [oAW] als Generierwerkzeug. Dazu hatte man für die Client-Entwicklung das RCP Framework von Eclipse [RCP] ausgewählt und es um einige Features und Oberflächenelemente ergänzt. Damit hatte das Architekturteam den Anteil der manuellen Programmierung auf ein Viertel des Codeumfangs eingedämmt: mehr als 12.000 Klassen und Interfaces mit etwa 1,4 Millionen Zeilen Java-Code wurden generiert.

Auf den ersten Blick schien dies ein erfolgreicher Ansatz, um Gemeinsamkeiten einheitlich und durchgängig umzusetzen. So wurden anscheinend der Programmieraufwand reduziert und Fehlerquoten gering gehalten.

Im Lauf der Weiterentwicklung hatten sich Anforderungen für weitere fachliche Komponenten ergeben, die die Nachteile des Projektansatzes aufdeckten: die Erweiterungen an den Generatoren führten zu unerwarteten Nebeneffekten und zu schlecht strukturiertem Code. Die Anbindung von Geschäftsobjekten an Dialoge erwies sich als unangemessen und wenig pragmatisch. Die Generierung von Steuerungs- und Ablauflogik aus UML-Aktivitätsdiagrammen erzeugte Java-Klassen, die bei fehlerhaftem Verhalten kaum zu verstehen und noch schwerer zu debuggen waren. Und nicht zuletzt wurde sehr viel stereotyper Java-Code erzeugt, der beim Generieren und Kompilieren enorm viel Rechenzeit verschlang und den Speicherbedarf auf den Client-Rechnern in die Höhe trieb.

Die Schere ansetzen

Abbildung 1 zeigt eine einfache Skizze für die Architektur einer grafischen Benutzeroberfläche, wie sie beispielsweise in [HuO07] zu finden ist. Anhand dieser Skizze erläutern wir, an welchen Stellen welche Maßnahmen umgesetzt wurden, um die Client-Anwendung zu vereinfachen und für erste Erweiterungen vorzubereiten.

Eine derartig aufgebaute Rich Client-Anwendung besteht aus Dialogen, in Form eines Eingabefensters (1) mit Anzeige- und Eingabefeldern sowie Steuerelemen-

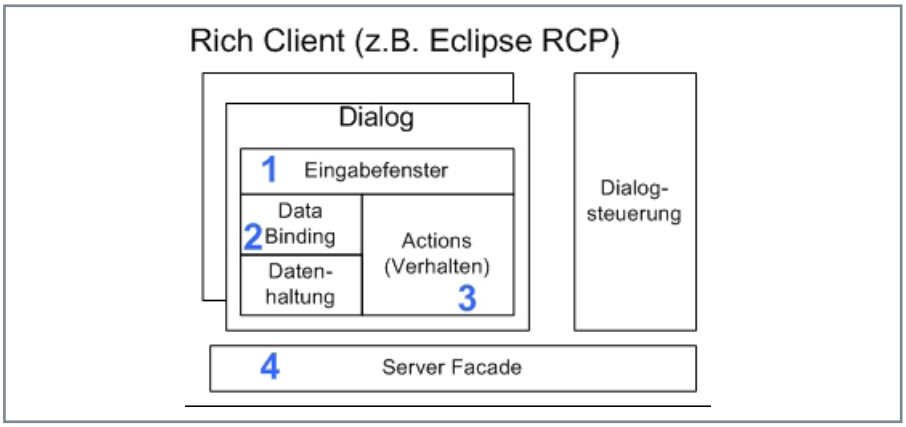

Abb. 1: Schematische Skizze einer Client-Architektur

ten usw., und aus einem Dialograhmen, der als Ablaufumgebung dient. Ein Dialog bietet Daten zum Lesen und Bearbeiten an, sie werden per Data Binding (2) an Geschäftsobjekte angebunden. Des Weiteren lassen sich über Steuerelemente Aktionen (3) auslösen, die in Summe das Verhalten des Dialogs ausmachen. Für den Aufruf serverseitiger Dienste haben wir eine Server-Fassade (4) vorgefunden.

Um die Komplexität des Generierungsansatzes für den clientseitigen Teil der Anwendung zu verringern, wurden in den genannten vier Bereichen verschiedene Änderungen vorgenommen.

Eingabefenster

Die Eingabefenster wurden vorher im UML-Modellierungswerkzeug definiert, indem Klassen aus dem Klassenmodell mit entsprechenden Stereotypen annotiert wurden. Dieses Verfahren war unübersichtlich, weil es der Fachseite und den Entwicklern keinen Eindruck von dem fertigen Fenster vermittelte, insbesondere fehlten die Hinweise zur Anordnung der Elemente, wie Layout, Abstände usw. (**Abbildung 2**).

Dieses Verfahren wurde abgelöst und durch den SWT-Designer, ein gängiges Werkzeug zur Erstellung von Masken, ersetzt. Nun lassen sich neue Fenster intuitiv entwerfen und können schon während des Entwurfs mit dem Fachbereich diskutiert werden. Dies führte zu einem Medienbruch zwischen den Werkzeugen, denn die Daten-/Anzeigeelemente im SWT-Designer stammten nicht mehr unmittelbar aus den UML-Klassenmodellen. Dieses Vorgehen produzierte übersichtliche Ergebnisse, und der Fachbereich übernahm die Aufgabe, die inhaltliche Konsistenz zu prüfen.

Datenhaltung und Data Binding

Für die Erstellung von Datenobjekten und ihre Anbindung an Eingabe- und Anzeigefelder wurde ebenfalls das Klassenmodell annotiert. Daraus erzeugten die Generatoren Interfaces, Factories und mehrere Klassen mit sogenannten „protected regions“, in denen die Entwickler die Anbindungs- und die Konvertierungslogik programmieren mussten.

Wir haben das alte Verfahren abgelöst, indem wir die Generierung von Java-Klassen vereinfacht haben (Interfaces und Factories entfielen) und für die Anbindung einige neue Hilfsmethoden im unserem Client-Framework eingeführt haben. In Anbindungsklassen können die Entwickler Datenobjekte und Felder in deklarativem Programmierstil einander zuordnen. Dabei kam das Fluent Interface-Muster zum Einsatz (vgl. [mar] und siehe **Abbildung 3**).

Dieser Programmierstil hat sich für die technische Umsetzung des Data Binding als übersichtlich und leicht verständlich erwiesen und führt zu deutlich weniger Java-Code. Außerdem wird er von der Eclipse-Entwicklungsumgebung angenehm unterstützt.

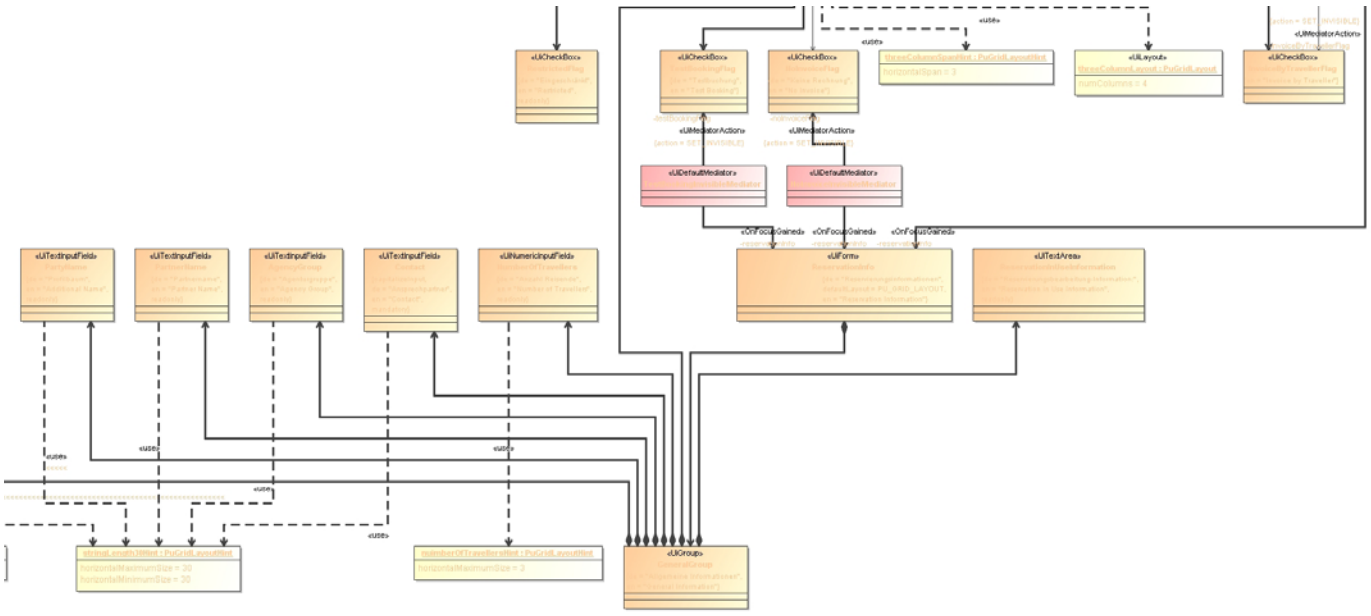

Abb. 2: Eingabefenster wurden in Klassendiagrammen annotiert

```
protected void createBinding(MainForm form, EnhancedComplexDataType m) {

        ...
        bind(form.getTextName()).model(m).property(name)
                .converter(SimpleTextConverter.getInstance());
        bind(form.getComboCountry()).model(m).property(country);

        ...
}
```

Abb. 3: Fluent API für Data Binding

```
public final void executeClientAction() throws ClientActionException {
     try {
         while (clientActionStep != QUIT) {
             switch (clientActionStep) {
             case CHECK_DIRTY: {
                 clientActionStep = internalCheckDirty();
                 break;
             }
             case MAINTAIN_DECISION: {
                 clientActionStep = internalMaintainDecision();
                 break;
             }

     ...
}
```

Abb. 4: Vorgefundene Lösung für das generierte Verhalten

Actions/Dialogsteuerung
Das Verhalten eines Dialogs und seine Zustandsübergänge wurden mithilfe von UML-Aktivitätsdiagrammen spezifiziert. Daraus erzeugte der Generator mehrere Klassen, in denen sämtliche Übergänge durch einen großen switch-case-Codeblock umgesetzt wurden.

Die Fachlogik einer ausgelösten Aktion (action) wurde auf „protected regions" mehrerer Klassen verteilt, was bei den Entwicklern immer wieder zu Missverständnissen führte. Die Suche nach Fehlern in den switch-case-Codeblöcken, unter anderem verursacht durch mangelhafte Modellierung und fehlende Konsistenzprüfungen im UML-Werkzeug, hatte in der Vergangenheit erheblichen Aufwand verursacht.

Diese switch-case-Codeblöcke und die „protected regions" wurden entfernt und neue Methoden im Client-Framework eingeführt. Zusammen mit den Entwicklern haben wir typische Codemuster erstellt und dokumentiert, um die Zustände und Übergänge aus den Aktivitätsdiagrammen auszuprogrammieren. Dies führte zu besser lesbarem, handgeschriebenem Code, der nach einfachen Prinzipien erweiterbar und dabei gut zu debuggen ist.

Service-Aufrufe
Für Service-Aufrufe wurden aus den UML-Aktivitätsdiagrammen clientseitige Proxy-Klassen generiert. Bei näherer Betrachtung stellte sich heraus, dass der erzeugte Code hochgradig schematisch und so auf ein Muster verallgemeinert werden konnte. Wir haben alle generierten Proxy-Klassen entfernt und durch einen parametrisierten Aufruf einer neuen generischen DynamicProxy-Klasse ersetzt, die wir dem Client-Framework zufügten.

Abb. 5: Problem Server-Fassade: die generierten Klassen lassen sich durch Aufruf einer generischen Klassen ersetzen

Diese Methode führte zu einer Einsparung von mehr als 200 Proxy-Klassen und zu einer deutlich geringeren Größe des Client-Programms.

In der folgenden Tabelle haben wir die Problemfelder, die ergriffenen Maßnahmen und den daraus resultierenden Nutzen zusammengestellt:

Architekturthema	Alter Ansatz	Problem	Neue Lösung	Verbesserung
Eingabefenster	■ UML-Klassendiagramme	■ Kein Kommunikationsmittel mit Fachbereich ■ Fehlende Layoutinformation	■ SWT-Designer für komplexe Dialoge ■ Sichtbares Layout der Fenster	■ Gute Kommunikation mit dem Fachbereich ■ Rapid Prototyping und schnelles Feedback vom Fachbereich
Datenhaltung und Data Binding	■ UML-Klassendiagramme für die Datenmodelle ■ Generierung von Klassen mit Factories und Interfaces	■ Mischung von generiertem und handgeschriebe-nem Code in Protected Regions ■ Generierung von unnötigen Interfaces und Factories	■ Vereinfachte Generierung von Klassen ■ Eine generische Factory für alle Datentypen ■ Fluent API für Data Binding	■ Einfachere generierte Artefakte ■ Trennung von generiertem und handgeschriebenem Code

Actions/ Dialogsteuerung	■ UML-Aktivitätsdiagramme ■ Switch/case-Code-Blöcke ■ 4 bis 5 Klassen pro Action	■ Semantisch inkorrekte Konstrukte wurden nicht erkannt ■ Code war ohne das Modell kaum verständlich ■ Mischung von generiertem und handgeschrie-benem Code	■ Manuelle Entwicklung von Actions anhand von Programmiervorgaben	■ Trennung von generiertem und handgeschriebenem Code ■ Fachlogik nur in einer Klasse pro Action
Service-Aufrufe	■ UML-Aktivitätsdiagramme ■ Generierung von Java Proxies für die Kommunikation mit dem Server	■ Hohe Anzahl von Proxy-Klassen mit ähnlicher Struktur ■ Schlechte Übersicht-lich-keit durch die Verteilung auf Modelle und Code	■ Dynamisches Proxy für die Kommunikation aller Serviceaufrufe	■ Einfachheit und Übersichtlichkeit ■ Generische Framework-lösung ■ Reduzierter Pflegeaufwand

Tabelle: Lösungsansätze

MDD mit Augenmaß einsetzen

Die Tiefe des Generierungsansatzes auf Basis extensiver UML-Modellierung war auf den ersten Blick sehr beeindruckend – schließlich handelte es sich um eine lauffähige Anwendung, die sich in Produktion befand. Die großen Probleme bei der fachlichen Erweiterung haben jedoch deutlich gemacht, dass die Evolution eines MDD-basierten Verfahrens nicht-trivial ist: konzeptionelle Schwächen und Architekturmängel traten zutage, die teilweise anfangs nicht absehbar waren. Sie führten zu Qualitätsproblemen und hohem Aufwand bei der Weiterentwicklung und Wartung.

Als Erfahrungen haben sich herausgestellt:

- Für die Spezifikation von grafischen Benutzeroberflächen halten wir UML-Profile für wenig geeignet. Die Erstellung von UI-Elementen kann durch einen spezialisierten GUI-Designer erledigt werden. Dies ist effektiver als die Spezifikation und Generierung aus UML.
- Die Generierung aus Verhaltensbeschreibungen der UML führte – mit einem eher simplen Generierungsansatz – zu fehlerbehaftetem und schwer debugbarem Code. Die Modelle müssen überprüfbaren Einschränkungen unterworfen sein und benötigen eine eindeutige Semantik.
- Die Generierung kann teilweise durch generischen Code abgelöst werden. In einigen Bereichen ist es effektiver, schlanken Quellcode zu schreiben als den Generator zu erweitern und den Code generieren zu lassen.

Wir halten darüber hinaus folgende Erkenntnisse aus unserem Projekt für erwähnenswert:

- Erstellung und Pflege von Generatoren verursachen permanent Aufwand. Während der Entwicklung sind Kosten für die Modellierung, den Modellexport und die anschließende Generierung zu berücksichtigen.
- Beim Bau von Generatoren sollte der Quellcode eines ausreichend großen Prototyps, der die wesentlichen Prinzipien der Zielarchitektur umsetzt, als Vorlage dienen. Erst an einem großen Beispiel lässt sich erkennen, welche Codeteile Kandidaten für Generierverfahren sind – oder auch nicht – und welche Zusatzinformationen dazu in den Generator einfließen müssen.
- „Protected regions“ können besonders kurz vor Lieferterminen dazu führen, dass manuelle Änderungen in generierten Codeabschnitten „passieren“, die nach der nächsten Generierung verloren sind. Deshalb sollte generierter von programmiertem Code strikt getrennt werden, entweder auf Klassen- oder auf Paketebene, oder besser noch nur als unveränderbares Archiv, also als JAR-Datei, bereitgestellt werden.
- Auch generierter Code sollte gut strukturiert sowie gut lesbar und verständlich sein, um das Auffinden von Fehlern zu erleichtern.
- Die Versionierung von Modellen, Generatoren und Quellcode muss nach einem umfassenden Konzept erfolgen, damit nachvollziehbar ist, welcher Code aus welchen Modell- und Generator-Versionen generiert wurde. Außerdem wurde auch in unserem Projekt deutlich, dass die Client- und Server-Entwicklungsteams neu generierte Artefakte zu unterschiedlichen Zeitpunkten benötigen. Demgemäß ist darauf zu achten, dass beide Entwicklungsteams entkoppelt voneinander gegen stabile Schnittstellen des jeweils anderen entwickeln können.

Einen Überblick über weitere Risiken und Lösungsmuster beim Einsatz von MDD geben Völter und Bettin in [VoBe04].

Fazit und Ausblick

Wir haben ein (Wartungs-) Projekt mit einem starken MDD-Ansatz übernommen: fast alles sollte in UML-Modellen spezifiziert und Code daraus generiert werden. Einige dieser Ansätze haben sich für die weitere Entwicklung als nicht pragmatisch erwiesen, sodass ausgewählte technische Aspekte aus dem Generierverfahren entfernt und durch DSL- und Framework-Lösungen ersetzt wurden. Dabei standen vor allem die pragmatischen Prinzipien im Vordergrund [HuT99], die existierenden Ansätze an den Stellen einfacher und schlanker zu machen, die im Lauf der Zeit unnötig komplex und damit schwer beherrschbar geworden waren.

Die Client-Entwicklung stand in diesem Erfahrungsbericht im Vordergrund,

allerdings dürften die gleichen Prinzipien und Methodiken auch für den Server relevant sein. Für die Querschnittsthemen wie Internationalisierung, Rollen und Berechtigungen, Validierung und Fehlerbehandlung können ähnliche Optimierungsprinzipien angewendet werden. Unsere Überarbeitungen sind von allen Beteiligten angenommen worden und haben zum Erfolg des Projekts beigetragen. ■

Referenzen

[HuO07] M. Haft, B. Olleck, Komponentenbasierte Client-Architektur, in: INFORMATIK-SPEKTRUM, Volume 30, Number 3, 143-158, DOI: 10.1007/s00287-007-0153-9, 2007

[HuT99] A. Hunt, D. Thomas, The pragmatic programmer: from journeyman to master, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999

[RCP] http://www.eclipse.org/home/categories/rcp.php

[VuB04] M. Völter, J. Betting, Patterns for Model-Driven Development, EuroPLoP, 2004; (als PDF auch unter http://www.voelter.de/data/pub/MDDPatterns.pdf)

[oaW] http://www.openarchitectureware.org/

[mar] http://www.martinfowler.com/bliki/FluentInterface.html