Aus Alt mach Neu:
Do's and Don'ts bei der Forms2Java-Migration

Björn Christoph Fischer & Achim Radtke
Triestram & Partner GmbH (t&p) Bochum

Schlüsselworte

Migration einer Oracle Forms Anwendung nach Java mittels Framework auf Basis von Open Source Technologien und Standards

Einleitung

Oracle Forms ist bekannt und geschätzt für seine außerordentliche Produktivität in der Entwicklung und seiner Stabilität im Betrieb. Dies sind zwei sehr gute und wichtige Gründe, die für Forms sprechen. Allerdings müssen sich Forms-Anwendungen im Business-Umfeld immer mehr an dem messen lassen, was Business-User aus ihrer Freizeit und dem Internet kennen: Rich Client Applikationen, die sehr ergonomisch zu bedienen sind, die der Benutzer an seine Bedürfnisse anpassen kann, die die Daten interaktiv visualisieren, und die sogar via Internet von überall aus aufgerufen werden können. Solche Rich Client Anwendungen prägen die Erwartungshaltung von Business-Benutzern und setzen neue Standards im Bereich der *Usability*. Daraus erwächst ein spürbarer Druck auf Software-Produkte auf Forms Basis, diese Erwartungshaltung ebenfalls zu erfüllen.

Die Triestram & Partner GmbH (t&p) entwickelt eine solches Standard-Software-Produkt namens lisa.lims für den Einsatz im Labor. Bei lisa.lims handelt es sich um eine typische Online-Transaction-Processing-Anwendung (OLTP), bei der es darum geht, im Mehrbenutzer-Betrieb Daten in Dialogen konsistent zu erfassen und automatisch von externen Quellen wie zum Beispiel Labor-Geräten über Schnittstellen zu übernehmen, die Daten zu bearbeiten, zu validieren bzw. auszuwerten, und in Berichten zu veröffentlichen. Bis zur Version 9 basierte die Software auf der Oracle Datenbank mit ca. 400 PL/SQL-Packages, ca. 150 Oracle Forms und ca. 250 Oracle Reports. Für die Version 10 wurden - unter Beibehaltung der Oracle Datenbank - die Dialoge und die Berichte auf Java Open Source Technologien umgestellt, nämlich

- die Eclipse Rich Client Platform (RCP) für die Desktop-Version der Dialoge
- die Eclipse Rich Ajax Platform (RAP) für die Web-Version der Dialoge und
- die Business Intelligence and Reporting Tools (BIRT) für die Berichte.

Technische Schlüssel-Anforderungen an die migrierte Anwendung

Der eigentlichen Migration ging eine Technologie-Studie voran, bei der die strategischen und technischen Anforderungen an die migrierte Software erhoben und bewertet wurden. Die migrierte Anwendung soll

1. die vorhandene Business-Logik in PL/SQL in der Datenbank wiederverwenden, um die Investition zu schützen, die bislang dort hinein geflossen ist.
2. eine individualisierbare Benutzeroberfläche haben.
3. auf Java als Basis-Plattform setzen.
4. sowohl als Desktop- als auch als Web-Anwendung verfügbar sein.
5. in ihrer Web-Variante mindestens mit dem Microsoft Internet Explorer und Mozilla Firefox kompatibel sein.
6. mit möglichst vielen Applikations-Servern kompatibel sein.

Zunächst wurde versucht, diese Anforderung auf Basis der Java-Enterprise-Architektur (JEE) mit einem JEE-konformen Application Server umzusetzen. Dabei ist das JEE-Paradigma mittelschicht-zentriert: Daten werden aus der Datenbank in den Application Server auf der Mittelschicht geladen und die Anwendung verändert die Daten ausschließlich über den Application Server. Die Konsistenz der Daten wird vom Application Server sichergestellt, Transaktionen werden üblicherweise ebenfalls vom Application Server verwaltet. Die Pilot-Migration der vorhandenen Forms-Anwendung in eine JEE-Architektur scheiterte u.a. daran, dass es im Umfeld von lisa.lims außer dem Anwendungs-UI noch weitere Systeme – wie zum Beispiel Labor-Geräte – gibt. Diese Geräte schreiben ihre Messdaten mittels PL/SQL-Aufrufen direkt in die Datenbank, also ohne den Weg über den Application Server zu gehen. Die Kombination aus JEE-konformen Application Server plus „Satelliten“ mit direktem Schreibzugriff auf die Datenbank stellte sich als inkompatibel heraus, da beides zusammen die Konsistenz der Daten in der Datenbank gefährdet[1].

Welche Relevanz hat das für die Migration des *User Interface* (UI) von Forms nach Java? Forms ist ein UI-Framework, das dem Entwickler bestimmte UI-Standard-Funktionalitäten „frei Haus“ zur Verfügung stellt, zum Beispiel:

- die Bedienbarkeit der Anwendung per Funktionstasten
- die Synchronisation mit der Datenbank, wenn der Benutzer Datensätze in die Anzeige einfügt, ändert oder löscht
- die Validierung eingegebener Werte (z.B. per WHEN-VALIDATE-Trigger)
- die Synchronisation von Master- und zugehörigen Detail-Datensätzen in der Anzeige
- die Möglichkeit, an beliebigen Stellen der Anwendung SQL-Abfragen abzusetzen
- die Möglichkeit, Datensätze in die Anzeige „lazy“ nachzuladen, wenn der Benutzer anfängt, in der Ergebnis-Liste zu scrollen

Wenngleich der mittelschicht-zentrierte Ansatz von JEE für Anwendungen, die „auf der grünen Wiese“ entstehen sicherlich vorteilhaft ist, ergab die Technologie-Studie jedoch, dass die OLTP-Anforderungen von lisa.lims nicht mit der JEE-Architektur umsetzbar waren. Insbesondere das pessimistische Locking und die Wiederverwendung der vorhandenen PL/SQL-Packages verhinderten den Einsatz von JEE für das Migrationsprojekt.

[1] Für die Details hierzu siehe das Paper zum Vortrag: Björn Christoph Fischer & Oliver Zandner: "Der Tiger im Tank: PL/SQL-Logik in Java-Anwendungen optimal nutzen", DOAG 2011.

Also mussten die beschriebenen Funktionalitäten mit anderen Technologien als JEE umgesetzt werden. Zum Zeitpunkt der Evaluation (2008) gab es außerhalb von JEE keine Java-Frameworks, die die beschriebenen datenbank-orientierten UI-Anforderungen „von der Stange“ erfüllt hätten. Dieses Paper beschreibt in den folgenden Abschnitten, wie die Anforderungen mittels eines eigenen Frameworks (rapid.Java) in Java *außerhalb* von JEE umgesetzt wurden. Der Fokus liegt hierbei auf der Implementierung der Client-Funktionalitäten, die das neu zu entwickelnde Framework im Rahmen seiner Model-View-Controller-Architektur bieten musste, um die gegebenen Anforderungen zu erfüllen.

Datensatz-Management im UI

Datensatz-Management bedeutet, dass der *Controller* den Zustand der Datensätze im UI „kennen“ und auf diese verschiedenen Zustände „reagieren“ muss – zum Beispiel: Wurde ein Datensatz vom Benutzer neu in die Anzeige eingefügt, so muss der *Controller* diesen auch dem *Model* hinzufügen. Wird ein aus der Datenbank gelesener Datensatz in der Anzeige verändert, muss der *Controller* zunächst prüfen, ob der Datensatz nicht eventuell bereits durch einen anderen Benutzer verändert worden ist, falls ja ist eine entsprechende Meldung ausgeben usw. Die folgende Grafik gibt einen Überblick über die verschiedenen Status im Lebens-Zyklus eines Datensatzes:

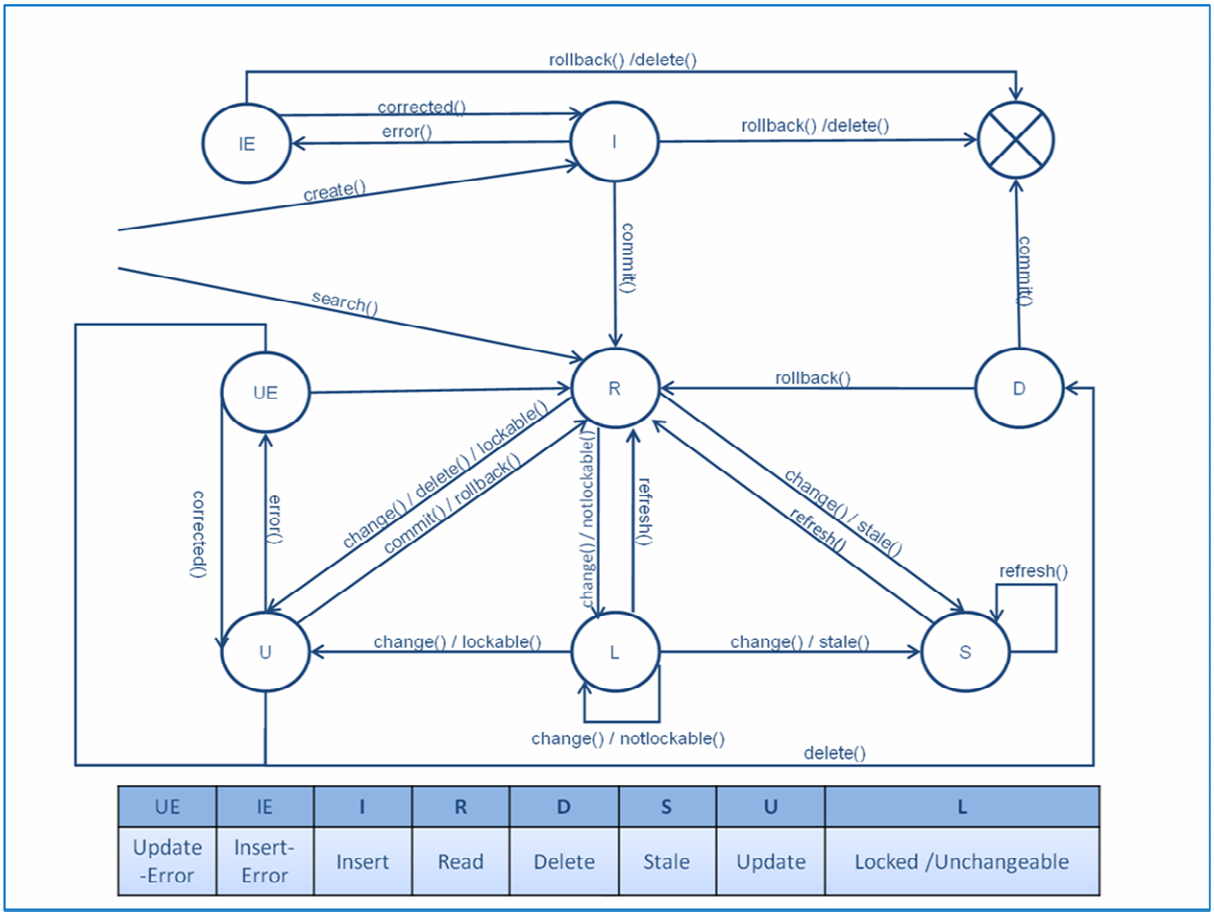

Abbildung 1: Die möglichen Zustände im Lebenszyklus eines Datensatzes

Das Management dieser verschiedenen Zustände im Lebenszyklus eines Datensatzes ist einer der Automatismen, die Forms dem Entwickler bietet und der bei der Migration von Forms nach Java zu berücksichtigen ist. In unserem Migrations-Projekt wurde sie als Teil der MVC-Architektur auf der Client-Seite implementiert und ist eine der Basis-Funktionalitäten des rapid.Java-Frameworks.

Master-Detail-Synchronisation

Die Master-Detail-Synchronisation betrifft die Frage, wie das UI Master-Detail-Relationen aus der Datenbank darstellt, wie etwa Aufträge und Auftragspositionen. Dabei sind zwei Use-Cases zu bedenken:

1. Wenn der Benutzer in der Ansicht von einem zum nächsten Master-Datensatz wechselt, dann müssen in der Detail-Ansicht automatisch die passenden Detail-Datensätze nachgeladen werden.
2. Was ist zu tun, wenn der Benutzer einen Detail-Datensatz ändert, und anschließend in der Master-Ansicht weiter scrollt? Hier ist eine Warnung notwendig, auf ungespeicherte Änderungen aufmerksam macht.

Diese Anforderungen wurden in rapid.Java wie folgt gelöst: Für jedes Anzeige-Element im UI (Tabelle oder Formular) existiert im *Model* eine Instanz einer speziellen Container-Klasse, die in rapid.Java als *EntityCollection* bezeichnet wird. Die Relation zwischen den Datensätzen aus der Datenbank wird durch eine in XML abgebildete Baumstruktur definiert, welche die existierenden *EntityCollections* hierarchisch Gliedert. Dadurch „kennt" der *Controller* die Abhängigkeiten zwischen den Objekten im *Model* und „weiß", welche Details jeweils nachzuladen sind, wenn in der *View* von einem zum nächsten Master-Datensatz gescrollt wird. Der Controller implementiert ebenfalls das Tracking geänderter Detail-Datensätze und löst entsprechende Prüfungen und Warnhinweise aus, wenn im Master-Datensatz gescrollt wird.

Validierungen

Forms sorgt durch seine Validierungs-Trigger dafür, dass der Cursor im aktuellen Feld „festgehalten" wird, wenn der Benutzer dort eine inkonsistente Eingabe vorgenommen hat. Wie soll in solchen Situationen in der migrierten Java-Anwendung verfahren werden?

Unsere Erfahrung zeigt, dass sich das Validierungsverhalten von Forms mit der gewählten UI-Bibliothek nur schwer umsetzen lässt. Das liegt unter anderem daran, dass editierbare Tabellen-Ansichten, sog. *Data-Grids*, an sich schon eine komplizierte Logik zum aktivieren und deaktivieren von Zell-Editoren abbilden müssen. In diese Logik einzugreifen und das Verlassen einer Zelle zu verhindern, hat sich als extrem schwierig herausgestellt. Außerdem wird in Eclipse RCP das Rendering der Steuerelemente vom darunter liegenden Betriebssystem vorgenommen. Damit ist die Steuerung des Fokus im Falle einer fehlgeschlagenen Validierung ebenfalls vom Betriebssystem abhängig, was bei der Realisierung des „festhaltens" immer wieder zu Problemen führte. Deshalb lautet unser Fazit in diesem Punkt, dass das Forms-Verhalten bei Validierungen auf keinen Fall in der migrierten Java-Anwendung „imitiert" werden sollte, sofern man nicht die Absolute Kontrolle über das Verhalten der Steuerelemente der UI-Bibliothek hat. Mit dem Verzicht auf dieses Verhalten stellen sich aber auch die folgenden Fragen, die es zu beantworten galt:

1. Wie bemerkt der Benutzer überhaupt, dass er einen Eingabe-Fehler gemacht hat?
2. Wie soll mit mehreren Fehlern innerhalb eines Datensatzes umgegangen werden?
3. Wie soll mit mehreren Fehlern an verschiedenen Datensätzen umgegangen werden?
4. Wo werden die fehlerhaften Werte zwischengespeichert, bis der Benutzer sie korrigiert?

In unserem Migrations-Projekt wurden diese Probleme wie folgt in unserem rapid.Java-Framework gelöst:

1. Der Benutzer kann mehrere Fehler bei der Eingabe machen, aber nur in *einem* Datensatz. Existieren Fehler, ist ausschließlich der fehlerhafte Datensatz noch editierbar.
2. Wenn der Benutzer den fehlerhaften Datensatz mit dem Cursor verlässt, wird ein modaler Dialog mit einer Fehlerliste angezeigt, sodass kein Fehler übersehen wird.
3. Solange die Fehler im aktuellen Datensatz existieren, werden neue Eingaben verhindert wie Speichern, Einfügen, Löschen und Wechseln zu einem anderen Master-Datensatz.

Paging oder kein Paging bei einer Forms-Ablösung?

In Forms lässt sich per Parameter bestimmen, in welchen Paket-Größen Datensätze beim Scrollen nachgeladen werden, wenn die Größe der Treffermenge die Anzahl der anzeigbaren Datensätze übersteigt (Paging). Da dieses Verhalten Vor- und Nachteile besitzt, muss bei einer Migration nach Java entschieden werden, ob ein äquivalentes Paging im Framework umgesetzt wird, oder alternativ auf Paging – und damit auch auf dessen Vorteile – verzichtet werden kann.

Im rapid.Java-Framework wird auf das Paging verzichtet und stattdessen die gesamte Treffermenge auf den Client geladen. Übersteigt die Treffer-Menge einen einstellbaren Schwellwert X, wird vor dem Ausführen der Datenbank-Abfrage vom Benutzer abgefragt, ob die Treffermenge von der Größe X tatsächlich angezeigt werden soll. Bei Überschreiten eines systemweit einstellbaren Maximalwertes (z.B. 50.000 Datensätze) wird die Ausführung der Abfrage komplett abgewiesen.

Der Verzicht auf das Paging bietet folgende Vorteile:

1. Die Sortierung auf dem Client erfordert keine erneute Datenbank-Abfrage, wenn die gesamte zu sortierende Treffermenge bereits in das *Model* geladen wurde.
2. Ebenso sind Gruppen-Funktionen auf dem Client ohne erneute Abfrage zu berechnen.
3. Das Scrollen in großen Treffermengen ist performanter, weil die Datensätze nicht nachgeladen werden müssen.

Wie können in einem Java-Client SQL-Abfragen abgesetzt werden?

Forms bietet die Möglichkeit, an beliebigen Stellen im Code SQL-Abfragen an die Datenbank abzusetzen.

In Java stehen sind dafür folgende Möglichkeiten denkbar:

1. Die Java Persistence Query Language (JPQL):
   JPQL ermöglicht es, über einen Objekt-Relationalen-Mapper Abfragen an die Datenbank abzusetzen, die mit Objekten (*Entity Beans* bzw. *Persistent Entities*) und deren Attributen arbeiten, statt mit Tabellen und Spalten. Grundlage hierfür ist die Definition eines Mappings von Datenbanktabellen auf Entity-Klassen in Java. Die Syntax von JPQL ist an SQL angelehnt und ermöglicht es, von der eigentlichen Abfrage-Sprache der Datenhaltungs-Schicht zu abstrahieren.
2. Natives SQL:
   Die native Abfrage-Sprache der jeweiligen Datenhaltungs-Schicht, in der Regel also ein SQL-Dialekt. Die entsprechenden SQL-Statements werden als Strings in die Anwendung integriert und via JDBC oder über JPA (Java Persiscence API) als „Native Queries“ an die Datenbank gesendet.
3. Eigene Java-API:
   In einer 3-Tier-Architektur kann auf Server-Seite eine selbst definierte API implementiert werden, welche die Ausführung von Abfragen kapselt. So kann die Komplexität von JPQL bzw. SQL verborgen werden, und eine einfache Möglichkeit geschaffen werden, aus dem Java-Client heraus Abfragen abzusetzen.

Aufgrund der gewählten 3-Schicht-Architektur (Datenbank, Server-Komponente und Client) war die direkte Verwendung des OR-Mappers oder JDBC auf dem Client nicht möglich. Stattdessen bringt rapid.Java eine eigene API mit ("rapid.Java Persistence API"). Diese stellt vereinfachte Abfrage- und

DML-Mechanismen bereit, wobei für besondere Situationen auch die Ausführung von JPQL oder SQL über den Server ermöglicht wird. Die API enthält:

- eine zentrale Server-Komponente für DML-Operationen (INSERT, UPDATE, DELETE).
- eine Methode namens find(), die Abfragen in der Datenbank auf Basis des Primär-Schlüssels ermöglicht.
- eine Familie überladener Methoden namens findByCriteria(), die Abfragen in der Datenbank auf Basis anderer als der Primär-Schlüssel-Felder ermöglichen und dabei weniger komplex, aber auch weniger flexibel sind als JPQL- oder SQL-Abfragen.
- verschiedene Methoden zur Ausführung von JPQL- oder Native Queries, welche für sehr komplizierte oder performance kritische Abfragen verwendet werden können.

Auf diese weise können Standardfälle schnell mit dem vereinfachten API abgebildet werden, ohne dass die Flexibilität verloren geht, die für komplexere Abfragen notwendig ist.

Häufige UI-Anforderungen, die Forms nicht bietet

Der Einsatz einer neuen Technologie ist nicht ausschließlich ein Ersatz für Forms, sondern bietet darüber hinaus Funktionalitäten, die in Forms bislang nicht verfügbar und umsetzbar waren. Im Folgenden wird beschrieben, wie diese zusätzlichen UI-Anforderungen im Framework realisiert wurden.

Individualisierbare und editierbare Tabellen-Darstellung

In Forms bestehen Tabellen aus einzelnen Formular-Steuerelementen, die vom Forms-Developer in Form einer Matrix angeordnet werden. Das bringt die folgenden Nachteile mit sich:

- Die Inhalte der Spalten sind nicht automatisch per Mausklick auf den Spaltenkopf sortierbar.
- Die Breite der Spaltenbreite kann zur Laufzeit nicht durch den Benutzer verändert werden.
- Die Reihenfolge der Spalten kann nicht zur Laufzeit durch den Nutzer geändert werden.

In rapid.Java basiert das UI auf dem *Standard Widget Toolkit* (SWT) von Eclipse und der JFace-Bibliothek, die auf SWT aufsetzt und zusätzliche UI-Komponenten zur Verfügung stellt. SWT enthält ein UI-Element für die Darstellung von Data Grids, das sog. *Table* Widget. Das Anpassen der Reihenfolge und Breiten der Spalten ist bereits ein Feature des SWT.

Die JFace-Bibliothek bietet zusätzlich eine Möglichkeit, Tabellen automatisch per Klick auf den Spalten-Header zu sortieren. Diese Sortier-Lösung war jedoch für das rapid.Java-Framework nicht ausreichend, da sie sich lediglich auf die Ansicht der Daten, nicht aber auf die Reihenfolge der Entitäten im *Model* bezieht. Das ist immer dann problematisch, wenn eine Aktion an einer bestimmten Stelle in der Folge der Datensätze erfolgen muss (z.B. Einfügen eines neuen Satzes zwischen zwei vorhandenen). Eine Differenz der Sortierungen im UI und im Model würde bei solchen Aktionen zu falschen Ergebnissen führen. Deshalb wurde die Sortierbarkeit von Tabellen in rapid.Java so implementiert, dass die Sortierung direkt im *Model* abgebildet wird. Dadurch wird anschließend auch die Anzeige neu sortiert und Anzeige und *Model* bleiben weiterhin synchron.

Das Editieren von Tabellen im UI basiert in rapid.Java auf dem „Editing-Support“ von JFace. Dieser wurde um folgende Funktionalitäten erweitert:

1. Sperren der editierten Datensätze in der Datenbank:
   Sobald ein Benutzer Datensätze in der *Table* editiert, werden diese von Java/RCP nicht standardmäßig in der Datenbank gesperrt. Das *Locking* wurde deshalb in rapid.Java implementiert.
2. Komfortable Editierungs-Möglichkeiten:
   Die Zell-Editoren der JFace-Bibliothek wurden so erweitert, dass dem Benutzer komfortable Möglichkeiten zum Editieren der Daten zur Verfügung gestellt werden können. Hierzu gehören neben dem einfachen Text-Zell-Editor auch Combo- und Checkboxen, sowie ein Dialog-Zell-Editor um Werte komfortabel auszuwählen (*List of values*, *Date Picker*-Dialog oder komplexe selbst definierte Auswahl-Dialoge).

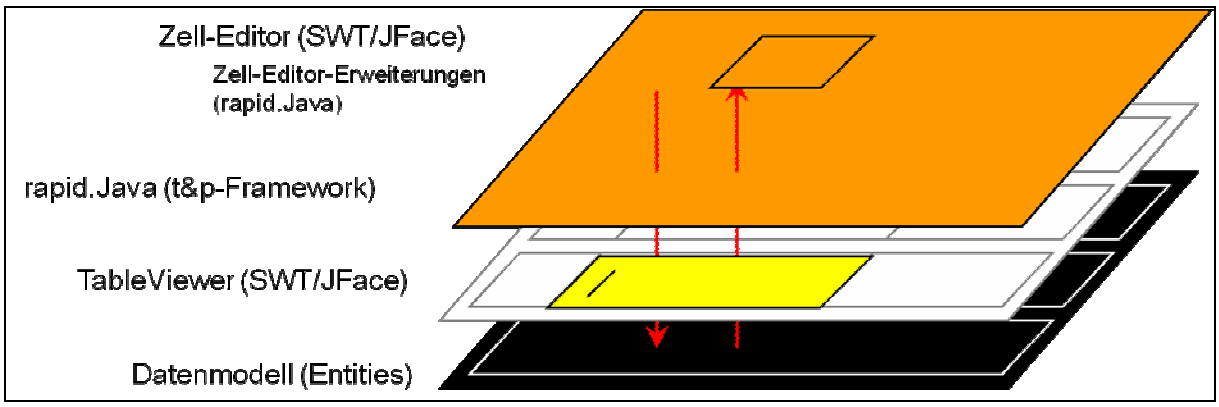

Abbildung 2: Editierbare Tabellen im rapid.Java-Framework durch Erweiterung von SWT

UI-Konfiguration

Die Konfigurierbarkeit des UI ist eine typische Anforderung aus der Entwicklung einer Standard-Software. Sie ermöglicht es, das UI an Kunden-Spezifika anzupassen, ohne dafür den Quellcode der Anwendung verändern zu müssen. Gleichzeitig ist eine sehr weitgehende Konfigurierbarkeit des UI aus einem Datenbank-Repository in vielen Java-Anwendungen eher unüblich.

In rapid.Java ist insbesondere die Darstellung und Funktionalität von Feldern Tabellen- und Formularansichten konfigurierbar. Das Erscheinungsbild und Verhalten beruht dabei auf Konfigurations-Tabellen in der Datenbank.

1. Konfiguration von Tabellen im UI:
   Hierbei wird in der Datenbank angegeben, auf welcher Datenbank-Tabelle die Tabelle im UI beruht und welche ihrer Spalten wie dargestellt werden sollen. Im Code der Anwendung genügt dann der Aufruf einer speziellen Framework-Methode (createTable()), den Rest übernimmt das rapid.Java-Framework.
2. Konfiguration von Formularen im UI:
   Bei Formularen wird deren Layout und die Eigenschaften der Felder in der Datenbank abgelegt. Für jedes anzuzeigende Feld ist im Anwendungscode lediglich ein Aufruf der createField()-Methode nötig, die Konfiguration des Feldes wird von rapid.Java vorgenommen.

Die folgende Eigenschaften von Feldern in Tabellen oder Formularen können per Konfiguration in der Datenbank festgelegt werden:

- Welches Attribut von welcher Entity-Klasse soll angezeigt/bearbeitet werden?
- Von welchem Typ soll das Feld sein (Text, Combo-Box, Check-Box) und welchen Daten-Typ soll es enthalten (String, Date, Number, Boolean)?
- Welche visuellen Attribute soll es haben (Hintergrund- und Vordergrund-Farbe, Schriftart)?
- Welche Beschränkungen oder Funktionalitäten gibt es bei der Eingabe (maximale Eingabe-Länge, erlaubte Zeichen, *List of values*, Pflicht-Feld, Zahlen- oder Datumsformat usw.)
- Welche Validierungsregeln sollen beim Verlassen des Feldes angewandt werden (*Regular Expression* oder selbst definierte Validierung per SQL-Abfrage)?

Internationalisierung

Die Internationalisierung einer Anwendung umfasst neben den Übersetzungen sämtlicher Texte im UI (*Labels*, Meldungstexte etc.) auch die landesspezifische Formatierung von Zahlen und Datumswerten.

In Java-Anwendungen werden für gewöhnlich die im Betriebssystem konfigurierten Landes- und Spracheinstellungen (*Locale*) verwendet, um Texte aus Properties-Dateien zu übersetzen, die mit der Anwendung ausgeliefert werden. Da eine Anforderung an das rapid.Java darin bestand, Sprach- und Länder-Kennzeichen aus den Benutzereinstellung in der Datenbank zu verwenden, sowie auch die Übersetzungen ausschließlich zentral in der Datenbank statt dezentral im Dateisystem vorzunehmen, wurde ein eigener Ansatz gewählt.

Basierend auf dem Sprach- und Länder-Kennzeichen in der Datenbank erstellt das Framework beim Login ein passendes *Locale*, was im Verlauf der Sitzung zur Erstellung von Datums- und Zahlenformaten auf dem Client verwendet wird. Zusätzlich berücksichtigt der PL/SQL-Teil des Konfigurations-Frameworks die Kennzeichen und gibt Texte für Labels, Meldungen etc. bereits in der korrekten Benutzersprache zurück. So muss der Übersetzungsvorgang nicht mehr separat auf dem Client durchgeführt werden, sondern wird automatisch von rapid.Java vorgenommen.

Kernanforderungen, die von gängigen Java-Frameworks (JEE) nicht unterstütz werden

In Forms werden Datensätze für die Dauer ihrer Bearbeitung automatisch in der Datenbank gesperrt (*pessimistic locking*). Auf diese Weise stellt Forms sicher, dass die Daten in der Datenbank im Mehrbenutzer-Betrieb konsistent bleiben. Die Übertragung dieses Verhaltens in eine JEE-konforme Architektur scheiterte in unserem Migrations-Projekt u.a. an zwei Automatismen der Application Server, nämlich dem *Caching* der Daten aus der Datenbank auf der Mittelschicht und dem *Connection-Pooling*[2]. Diese Mechanismen verhinderten in unserem Fall auch die Integration von PL/SQL in der Datenbank in eine JEE-Architektur.

[2] *Connection Pooling* bedeutet, dass beim Start des Application Servers eine gewisse Anzahl von Datenbank-Sitzungen gestartet wird. Führt ein Benutzer in der Anwendung eine datenbank-bezogene Aktion aus, wird ihm dazu eine der Verbindungen aus dem Pool zugewiesen, die nach Beendigung der Aktion wieder in den Pool

Daher wurde im rapid.Java-Framework eine Infrastruktur für das *pessimistic locking* und die Integration von PL/SQL in der Datenbank implementiert[3].

Fazit

Welches Fazit stand am Ende des Migrations-Projektes?

1. Die Migration einer Individual-Software von Oracle Forms nach Java ist anspruchsvoller und komplexer als zunächst angenommen. Das liegt vor allem daran, dass Oracle Forms die Entwicklung von OLTLP-Anwendungen durch zahlreiche Automatismen und eine entsprechende Infrastruktur stark vereinfacht. Für diese Automatismen und diese Infrastruktur müssen in der Java-Welt Äquivalente gefunden werden. Wo es diese in der Java-Welt nicht gibt, müssen sie selbst implementiert werden.
2. Zum Zeitpunkt unserer Technologie-Evaluation gab es keine Open Source Frameworks in Java, welche die oben genannten typischen OLTP-Features von Oracle Forms „von der Stange" geboten hätten. Sicherlich: Es gab zu diesem Zeitpunkt schon kommerzielle Java-Frameworks wie Oracle ADF. Diese basieren jedoch auf einer wesentlich „schwergewichtigeren" Infrastruktur als der nun geschaffene Ansatz von rapid.Java.

Aufgrund dieser Umstände hat sich t&p dazu entschlossen, ein eigenes, leichtgewichtiges Framework für die Forms2Java-Migration zu entwickeln. Dieses Paper hat im Sinne von „Do's and Don'ts" gezeigt, welche Punkte bei der Framework-Entwicklung zu berücksichtigen sind, welche Wege man gehen sollte und welche Pfade besser zu vermeiden sind, um an das gewünschte Ziel zu gelangen.

Kontaktadresse:

Björn Christoph Fischer & Achim Radtke

Triestram & Partner GmbH (**t&p**)

Kohlenstraße 55

D-44795 Bochum

Telefon: +49 (0) 234-9 43 75 - 0

Fax: +49 (0) 234-45 22 06

E-Mail b.fischer@t-p.com bzw. a.radtke@t-p.com

Internet: www.t-p.com

---

zurückgegeben wird. Eine Konsequenz ist, dass alle Benutzer der Anwendung mit demselben Datenbankbenutzer angemeldet werden, was bei vorhandenen PL/SQL-Packages Probleme bereiten kann.

[3] Für die Details hierzu siehe das Paper von Thomas Haskes & Jens Hüttemann: "Der Tiger im Tank: PL/SQL-Logik in Java-Anwendungen optimal nutzen".