ORM-Frameworks – Des einen Freud, des anderen Leid

Autor: Peter Bekiesch, Herrmann & Lenz Services GmbH

ORM-Frameworks (Object Relational Mapper) erfreuen sich in den letzten Jahren immer größerer Beliebtheit und bilden den Mörtel zwischen der objektorientierten Programmierwelt eines Softwareentwicklers und der relationalen Modellierungswelt des ER-Designers. Nicht selten ist das Produkt fertig und getestet in den Live-Betrieb übergegangen, und prompt kommen die Beschwerden der Endanwender über inflationäre Antwortzeiten. Die Ursachen sind meistens sehr vielschichtig und der ersehnte „Go-Faster-Schalter“ wurde noch nicht entwickelt. In diesem Artikel kommen wir einigen Ursachen auf die Spur.

Professionelle Anwendungsentwicklung für Geschäftsprozesse ohne einen zentralen transaktionsfähigen Datenspeicher zu betreiben, ist in der heutigen Zeit kaum noch denkbar, wenn nicht sogar unmöglich. Moderne Enterprise-Applikationen setzen hier gewöhnlich auf relationale Datenbank-Management-Systeme (RDBMS). Die Anwendungsentwickler folgen ihrer Rolle entsprechend und speichern ihre Objekte im RDBMS und bei Bedarf lesen sie die Daten wieder aus. Soweit so gut.

Am Anfang steht das WIE!

Entwickler von Softwarebausteinen stehen am Anfang eines Software-Entwicklungszyklus vor der allseits (un-)beliebten und nicht zu unterschätzenden Architektur-Frage. Vereinfacht betrachtet kommt in den meisten Fällen ein mehr oder weniger ähnliches Schichtenmodell zum tragen, wie in Abbildung 1 dargestellt.

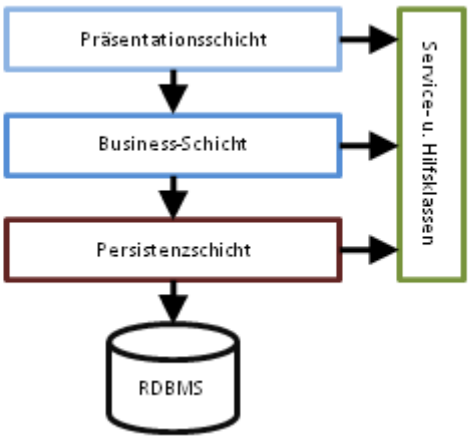

Abbildung 1: Vereinfachtes Schichtenmodell

In diesem Artikel klammern wir die Präsentationsschicht bewusst aus und konzentrieren uns nur auf die darunter liegenden Schichten. Von der Projektleitung oder einer höheren Instanz werden i.d.R. grundlegende Faktoren definiert, die das Endprodukt erfüllen muss:

- Produktivität
- Wartungsfreundlichkeit
- Investitionssicherheit
- Performance
- Skalierbarkeit
- Fehlerfreiheit

Diese Liste kann noch beliebig erweitert werden. Darüber hinaus sieht sich der Projektleiter bzw. jeder einzelne Entwickler mit einigen weiteren *BuzzWords*, wie „InTime“ und „InBudget“ konfrontiert. Die Definition der Begriffe ist selbsterklärend.

Anfangs wird ein Großteil der Aufmerksamkeit der Persistenz-Schicht und dessen interner Struktur und Arbeitsweise gewidmet. Prinzipiell ist die Persistenzschicht dafür verantwortlich, dass alle an sie übergebenen Daten korrekt in der Datenbank gespeichert und bei Bedarf, möglichst performant, ausgelesen werden. Der Persistenzschicht stehen, je nach eingesetzter Programmiersprache und Datenbank, unterschiedlichste Schnittstellen zur Erfüllung ihrer Aufgabe zur Verfügung.

Die heutige Softwareentwicklung bestreitet ihr Aufgabenspektrum überwiegend mit objektorientierten Programmiersprachen wie z.B. C++, Java oder C#. Für die meist vertretenen Programmiersprachen existieren geeignete API´s und Treiber zu den verschiedensten Datenbank-Systemen. In C/C++ zum Beispiel kann der Zugriff über die OCI-API (Oracle Call Interface) auf eine Oracle-Datenbank erfolgen. OCI ist eine C-Schnittstelle, die entsprechende Methoden zur Kommunikation mit der Datenbank bereitstellt. Eine Abstraktionsstufe höher wird, ebenfalls von Oracle, die OCCI-Klassenbibliothek (Oracle C++ Call Interface) zur Verfügung gestellt, die die Komplexität und Fehleranfälligkeit von OCI verbirgt und dem Entwickler eine objektorientierte Klassenstruktur anbietet. In Java erfolgt der Zugriff gewöhnlich über die JDBC-API und die passenden JDBC-Treiber. Nicht zu vergessen ist die unter

Windows bekannte ODBC-API. Auch diese Liste ließe sich hier noch weiter fortführen.

Kapselung – Ich bin eine Zwiebel!

Wie die Architektur einer Software, die in logische Schichten unterteilt wird, lässt sich ebenfalls die Persistenzschicht in einzelne logische Schichten aufteilen. Die unterste Ebene bilden die datenbankspezifischen Schnittstellen und Treiber, auch Datenzugriffsschicht genannt, die die eigentliche Kommunikation mit dem Datenbank-System bewerkstelligt. Jede weitere Schicht darüber ist eine Abstraktion und Kapselung der unterliegenden Schicht. Warum der Aufwand?

Die Realisierung des Absetzens eines einfachen SELECT-Befehls mit anschließender Verarbeitung der Ergebnismenge kann beim Einsatz einer in der untersten Ebene liegenden API mitunter kompliziert, fehleranfällig, auf jeden Fall jedoch zeitaufwändig sein. Wer schon mal in C/C++ mittels OCI eine ähnliche Aufgabe gelöst hat, der kann sicherlich ein Lied hiervon singen. Auch in Java über die JDBC-API zum Beispiel ist der Sourcecode, der für die Lösung geschrieben werden muss, nicht mit 5 Code-Zeilen erledigt. Der Wunsch, die immer wiederkehrenden und essenziellen Aufgaben zu automatisieren und somit zu abstrahieren und zu kapseln, ist groß. Was manchmal beobachtet werden kann ist ein reines Schichtenfeuerwerk. Mancher Entwickler erliegt dem Bedürfnis nach immer mehr Kapselung, noch mehr Automatisierung, noch weniger Code. Bewusst überspitzt formuliert steuern wir mehr oder weniger erfolgreich dem Endziel entgegen: Der „One-Button-API“.

Datenzugriff – Aber wie und wo?

Die Zugriffe auf die Datenbank direkt an der Stelle im Sourcecode, wo sie benötigt werden, also in der Business-Schicht, manuell mittels SQL und der passenden API abzusetzen, ist natürlich machbar, sollte aber schon aufgrund der Wartungsunfreundlichkeit in jedem Fall vermieden werden. Ein weiterer Grund ist die Verletzung der Schichtenarchitektur. Der Zugriff auf die Datenbank mittels SQL sollte unbedingt in der Persistenzschicht erfolgen. Die Business-Schicht erhält nur eine abstrahierte Sicht auf die unter ihr liegende Schicht und sollte bewusst keinerlei Kenntnis über die eingesetzte API, die technischen Details, geschweige denn über

unterschiedliche SQL-Dialekte besitzen, was bei Unterstützung von mehreren RDBMS-Anbietern unvermeidlich ist.

Ein möglicher Lösungsweg ist das DAO-Muster (Data Access Object). Die Zugriffe auf ein Datenbank-Objekt werden zentral in einer Klasse gebündelt. Die implementierten Methoden sind für das Lesen, Schreiben oder Löschen verantwortlich. Die SQL-Befehle sind weiterhin innerhalb der Methoden hart kodiert. Passend hierzu wird i.d.R. das VO- (Value Object) bzw. DTO-Muster (Data Transfer Object) einbezogen, indem ein Datensatz einer Tabelle auf eine Instanz einer VO-/DTO-Klasse abgebildet (mapped) wird. Der Transport der Daten zwischen den Schichten erfolgt dann mit diesen Objekten. Leider ist die Lösung nicht den immer größer werdenden Anforderungen gewachsen. Immer kürzere Produktzyklen verbunden mit immer höheren Qualitätsanforderungen rufen förmlich nach anderen, erprobten und etablierten Lösungen und Frameworks.

Heute stehen dem Entwickler etliche Frameworks zur Verfügung, mit denen diese komplexe Aufgabe bewältigt werden kann. In fast allen gängigen Programmiersprachen existiert heute eine Vielzahl von Bibliotheken, die um die Gunst der Entwickler buhlen. Die ORM-Frameworks, auch Persistence-Framework genannt, haben sich inzwischen einen festen Platz im Architektur-Diagramm erarbeitet. An dieser Stelle sparen wir uns die Definition, was ein ORM-Framework ist und wie es im Detail arbeitet, dies würde den Rahmen dieses Artikels sicher sprengen. Hervorzuheben ist die Motivation für ORM-Frameworks, die Überbrückung der Kluft zwischen der objektorientieren und der relationalen Welt. Es sollte jedem klar sein, dass hier ein Paradigmenwechsel stattfindet.

Eigenentwicklung oder nicht?

Es gehörte damals und auch heute noch zum guten Ton eines Entwicklerteams, sich sein *eigenes* ORM-Framework zu schreiben. Die Motivation mag sehr vielfältig sein. Sie reicht von der anfänglich hohen Hürde, sich in ein bestehendes Framework einzuarbeiten, es zu testen und letztendlich einzusetzen bis hin zum menschlichen Faktor und der überaus sportlichen Meinung: „*Ich kriege das besser, schneller, schöner hin*“, gemäß dem Motto „W*as der Bauer nicht kennt…*“. Eine Recherche auf dem Portal *sourceforge.net*, einer Plattform für freie Software, kann einen ersten

Eindruck vermitteln, wie beliebt das Thema in den Entwicklerbüros ist. Die Suche nach den Begriffen „Object Relational Mapper“ gibt eine Liste von mehreren tausend Projekt-Einträgen (nicht nur Java-basiert) wieder. Beim Blättern durch die Ergebnismenge und einer genaueren Betrachtung der Spalten „Activity“ und „Downloads“ lässt schnell erahnen, dass sehr viele Projekte über deren Alpha- oder gar Beta-Status nicht hinausgekommen sind. Von einer allgemeinen Bekanntheit, was durch entsprechend hohe Downloads belegbar wäre, ganz zu schweigen. Die Vermutung liegt nahe, dass die Dunkelziffer derartiger Projekte noch weitaus höher ist.

Eine gründliche Abwägung aller Argumente für oder gegen eine Eigenentwicklung sollte also in jedem Fall erfolgen, um kostbare Zeit und vor allem das sowieso knapp bemessene Budget nicht über zu strapazieren. Ganz zu schweigen von der Komplexität dieses Themas, die fast immer unterschätzt wird und in vielen Fällen das Projekt scheitern lässt.

Die Java-basierten ORM-Frameworks, die wir im weiteren Verlauf näher betrachten, sind seit der zunehmenden Popularität dieser Programmiersprache und deren gegebenen Möglichkeiten regelrecht wie Pilze aus dem Boden geschossen. Inzwischen haben sich einige Produkte auf dem Markt behauptet und erfreuen sich zunehmender Beliebtheit. Sie sind stabil und haben in unzähligen Projekten ihre Eignung für den Einsatz in Produktivsystemen längst bewiesen. Zu den populärsten Vertretern gehören z.B. TopLink aus dem Hause Oracle, oder Hibernate, ein Open-Source-Projekt (LGPL) und wohl auch das bekannteste ORM-Framework.

Hibernate – Ein zeitgemäßes ORM-Framework

Hibernate verfolgt den leichtgewichtigen Ansatz mittels POJO´s (Plain Old Java Object) Daten zu persistieren. Hibernate kann sowohl in Standalone-Applikationen als auch in komplexen JEE-Umgebungen eingesetzt werden. Es werden neben ORACLE auch andere Datenbanken wie z.B. DB2, MySQL oder SQLServer unterstützt. Über wenige Konfigurationsdateien können alle notwendigen Einstellungen vorgenommen werden. Die Deklaration von Meta-Informationen, die Hibernate für das Mapping der Objekte auf Tabellen und Tabellenspalten benötigt, kann in XML-Form oder auch über Annotations (ab JDK 1.5) erfolgen. Aus Sicht des

Entwicklers ist es relativ einfach, via Hibernate einen Zugriff auf die Daten einer Datenbank zu realisieren, bemerkenswerterweise, ohne eine einzige Zeile SQL zu schreiben. Die Möglichkeiten wie z.B. one-to-one oder one-to-many Assoziationen, uni- und bidirektionale Objektnavigation oder auch Vererbung abzubilden, lassen das Herz eines OOP-Entwicklers höher schlagen, immer unter der Maßgabe die Daten korrekt in der Datenbank abspeichern und auch wieder abrufen zu können. Selbstverständlich gehört die Unterstützung von Transaktionen ebenfalls zum Funktionsumfang wie eine eigene Abfragesprache, die HQL (Hibernate Query Language), die sehr stark an SQL bzw. EJB-QL erinnert. Die Anbindung von weiteren Frameworks, wie Connection-Pools und Caches, kann sowohl den Funktionsumfang erweitern als auch die Performance und Skalierbarkeit der Software massiv beeinflussen.

Dies ist wohl gemerkt nur ein Teil der Stärken von Hibernate und wir möchten nicht verschweigen, dass auch andere etablierte ORM-Frameworks ähnliches leisten. Es sollte klar geworden sein, dass eine Eigenentwicklung keine echte Option ist, es sei denn, sehr triftige Gründe sprechen dafür.

Die Lösung aller Probleme?

Die Frage aller Fragen, die sich stellt ist, ob der Einsatz eines ORM-Frameworks, sei es nun Hibernate, TopLink oder ein vergleichbares Framework, nun DIE Lösung ist, mit der man alle denkbaren Konstellationen und Anforderungen im Persistenz-Bereich abdecken kann? Diese Frage mit einem klaren „Ja“ oder „Nein“ zu beantworten wäre in unseren Augen nicht seriös und zudem nicht zielführend. Es kommt nach wie vor auf die spezifischen Anforderungen im Projekt an. Es bleibt jedoch festzuhalten, dass der Einsatz eines professionellen ORM-Frameworks immer eine genauere Untersuchung wert ist.

Wo ist der Haken?

Der aufmerksame Leser wird sicherlich festgestellt haben, dass ORM-Frameworks grundsätzlich eine „Gute Sache“ sind. Wo ist denn nun „des anderen Leid“?

Mancher Entwickler neigt dazu, und dies ist kein Vorwurf, sondern eine auf langjähriger Erfahrung beruhende Feststellung, es sich sehr einfach zu machen,

gewisse gegebene Möglichkeiten vollkommen auszureizen und den Bogen, ohne weiteres bewusstes Zutun, zu überspannen. Bei diesem Vorgehen geht verständlicherweise die Sensibilität für einige Begleitaspekte verloren. Die Berücksichtigung von Randbedingungen und Konsequenzen des eigenen Handelns findet sehr begrenzt bis gar nicht statt. So sehr sich das Schichten-Muster seinen Platz in der Hall-of-Fame der Architektur-Muster verdient hat, so nachteilig wirkt es sich auf die Motivation des Entwicklers aus, seine schichtübergreifenden Methodenaufrufe selbstkritisch zu hinterfragen.

Wir möchten betonen, dass dies keine allgemeine Anklage der Entwicklerzunft ist. Vielmehr ist es ein provokant formulierter Aufruf, sich in Test-Szenarien intensiv mit dem eingesetzten ORM-Framework auseinander zu setzen und dessen Potential wie Fallstricke näher zu beleuchten. Die Erkenntnisse sind in Form von Workshops und/oder Schulungen an die Entwicklungsabteilung weiter zu geben.

Es muss kritisch hinterfragt werden, welche der unzähligen Features, die die ORM-Frameworks bieten, auch tatsächlich eingesetzt werden. Hierbei trennt sich oft die Spreu vom Weizen. Die Besinnung auf die Basis-Funktionalität, also das reine Mapping der Objekte auf Tabellen und Datensätze, kann manchmal zielführender sein, als der Anspruch, alle erdenklichen Features, z.B. Vererbung, bidirektionale Objektnavigation oder kaskadierendes Object-Fetching/Saving, bis ins Detail auszureizen. Ein weiterer Aspekt, der noch nicht behandelt wurde, ist die Konfiguration des ORM-Frameworks, Quelle vieler Probleme und Fallstricke.

Um an dieser Stelle den Kreis zum Entwickler zu schließen fällt auf, dass gerade bei Performance- oder Skalierungs-Problemen die Verantwortung gern *schichtweise delegiert* wird. Der oft gehörte Satz: „Ich rufe nur die Methode X in der Klasse Y vom Kollegen Z auf“, entlarvt den Übeltäter nicht sofort, da der vermeintlich Schuldige „Kollege Z“ einen ähnlichen Satz mit anderen Akteuren parat hat. Dieses Spiel geht dann so weiter, bis alle Beteiligten der überzeugten Meinung sind, die viel zu klein dimensionierte Hardware sei schuld. Wenn alles so einfach wäre.

Klar ist, zwei- und mehrschichtige Applikationen mit Anbindung an eine oder mehrere Datenbanken sind komplex und die Anzahl der innen liegenden logischen Schichten

ist hoch. Verständlicherweise gehen durch die Transformation von Daten und Methodenaufrufen zwischen den einzelnen Schichten gewisse Details und damit die Sensibilität für die Auswirkungen verloren.

An dieser Stelle sind effektive Maßnahmen gefragt, die angesprochenen negativen Auswirkungen einzudämmen, im optimalen Fall gar nicht erst aufkommen zu lassen:

1. Geeignetes Logging-, Profiling- und Test-Framework integrieren
2. Realistisches Datenvolumen für Tests vorbereiten
3. Realistische Test-Szenarien entwerfen und implementieren
4. Isolierte Tests der Persistenz- und Datenzugriffsschicht durchführen
5. Unit-Tests der Geschäftslogik durchführen
6. Auswertung der Logging- und Profiling-Ergebnisse nach jedem Testlauf
7. Aufbau eines Best-Practice-Katalogs basierend auf den neuen Erkenntnissen
8. Re-Design von suboptimalen Applikationsteilen
9. Schulung der Entwickler und Weitergabe der Erkenntnisse
10. Aktuellen Ist-Zustand prüfen und die Iteration ab Punkt 2 wiederholen

Selbstverständlich kann dieser Maßnahmenkatalog noch um weitere sinnvolle Schritte und Iterationen erweitert werden. Er sollte viel mehr als ein roter Leitfaden betrachtet werden, sich diesem komplexen Thema zu nähern. Eine einzelfall- und entwicklungsprozessabhängige Anpassung ist hierbei dem verantwortlichen Team überlassen. Der Kreis der Akteure sollte in jedem Fall den Datenbank-Administrator bzw. einen Datenbank-Spezialisten einbeziehen, der das Datenbank-Tracing bzw. Logging beherrscht.

Fazit

ORM-Frameworks sind grundsätzlich zu empfehlen und in den Framework-Katalog aufzunehmen und zu integrieren. Von einer Eigenentwicklung sollte weitestgehend Abstand genommen werden. Eine Kombination und Integration mit weiteren sinnvollen Frameworks, wie z.B. dem Spring-Framework, sollte ebenfalls in die Architekturüberlegungen einfließen. Der Einsatz eines ORM-Frameworks sollte wohl durchdacht, geplant und vorbereitet sein. Die Durchführung von Test-Szenarien mit einem realistischen Datenvolumen darf auf keinen Fall vernachlässigt werden. Die u.a. in Logging- und Profiling-Auswertungen gemachten Erkenntnisse sollten zu Best-Practices transformiert in einen allgemein zugänglichen Katalog aufgenommen werden. In Schulungen und Workshops sind dann die auf Basis der Best-Practices erstellten Prototyp-Anwendungsfälle allen Entwicklern nahe zu bringen.

Sollten wir Sie mit diesem Artikel angesprochen haben, so würden wir uns über ein Feedback, weitere Anregungen oder einen Erfahrungsaustausch sehr freuen. Wir werden diesen Themenkreis in weiteren Beiträgen vertiefen, die Maßnahmen im o.a. Katalog weiter ausführen, einige Best-Practices und Anti-Pattern aufzeigen und diese durch konkrete Beispiele und Fakten untermauern.

Weitere Informationen

- Sourceforge - http://sourceforge.net/
- Hibernate - http://www.hibernate.org/
- TopLink - http://www.oracle.com/technology/products/ias/toplink/index.html
- Spring-Framework - http://www.springsource.org/

Kontakt:

Peter Bekiesch

peter.bekiesch@hl-services.de