Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 5

Migrationsstrategie für die sanfte Überführung von Bestandanwendungen zu modell-basierten Multi-Plattformanwendungen mittels MOHITO

Autor: A. Schwichtenberg (CAS)

Co-Autoren: G. Hübsch (CAS)

E. Gailus (B2M)

H. Groenda (FZI)

Fertiggestellt am: 31.05.2014

Schlagworte: Migration, Datenhaltung

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
01	AS	07.05.2014	Initiale Fassung
02	EG	09.05.2014	Migration der Datenhaltung
03	EG	14.05.2014	Migration des Datenaustauschs
04	GH	16.05.2014	Migration der Serveranbindung
05	GH	19.05.2014	Review CAS
06	HG	02.06.2014	Review FZI

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

Inhaltsverzeichnis

Abbildungen

1 Einleitung

Im Rahmen des MOHITO Projektes sollen Methoden und Mechanismen zur Generierung von Datenhaltungs-Stacks für verschiedene Plattformen und Endgeräte entwickelt werden. In dem vorliegenden Dokument wird beschrieben, wie eine bestehende Anwendung zu einer zur MOHITO Referenzarchitektur konformen Anwendung mit generierter Datenhaltung migriert werden kann.

Grundlage für die hier beschriebene Migrationsstrategie ist die MOHITO Referenzarchitektur (B. Klatt K. , 2013), die finale DSL (A. Schwichtenberg G. H., 2013), die finale Version der Generatortemplates und des Modellinterpreters (A. Schwichtenberg G. H., 2014), sowie die in (B. Klatt E. G., 2013) beschriebene initiale Migrationsstrategie.

Die Migration von bestehenden Anwendungen hin zur MOHITO Referenzarchitektur ist maßgeblich durch die Lokalität des Datenzugriffs geprägt. Sowohl für Datenaustausch, Synchronisation, Caching als auch den einfachen Lese- und Schreibzugriff gilt, dass eine Migration umso einfacher ist, je besser die Zugriffe bereits gekapselt sind.

Idealerweise sind bereits alle Formen des Datenzugriffs in einer einzigen, zentralen Datenzugriffsschicht gekapselt. Ist dies der Fall, so gestaltet sich eine Migration recht gradlinig und man muss nur diese Zugriffsschicht selbst anpassen. Ist dies nicht der Fall, so muss zunächst ein entsprechender Zustand hergestellt werden.

Gleiches gilt auch für die Ausgestaltung des Datenmodells bzw. der Datenmodelle im Fall einer verteilten Anwendung. Nutzen alle Teilsysteme das gleiche logische Datenmodell so kann dies direkt in eine von MOHITO unterstütze Modellierung überführt werden. Gegebenenfalls können sogar bestehende Transformationen / Imports genutzt werden um automatisiert das benötigte Ecore Datenmodell zu erhalten. So wird von der Eclipse Plattform zum Beispiel direkt ein Import von UML Datenmodellen in Ecore Datenmodelle angeboten.

Wird kein einheitliches Datenmodell genutzt, so sind die unterschiedlichen Datenmodelle der verschiedenen Teilsysteme zunächst untereinander abzugleichen. Idealerweise zu einem einheitlichen Modell, gegebenenfalls aber auch in ein Gesamtmodell, von dem jedes Teilsystem nur einen Teilbereich nutzt.

Zusammenfassend ergibt sich ein maximaler Migrationsprozess, von dem je nach Zustand der Bestandssoftware einzelne Prozessschritte ausgelassen werden können:

1. Je Teilsystem: Kapselung der Schreib- und Lesezugriffe in zentraler Datenzugriffsschicht.
2. Je Teilsystem: Kapselung von Datenaustausch und -kommunikation in zentraler Datenzugriffsschicht.

3. Je Teilsystem: Kapselung von Konfliktlösung in zentraler Datenzugriffsschicht.
4. Abgleich der Datenmodell zwischen den Teilsystemen
5. Überführung des Datenmodells in ein Ecore Datenmodell
6. Datenmodell anreichern / annotieren
7. Generierung und Austausch der zentralen Datenzugriffsschicht

Der letzte Prozessschritt (7) stellt bereits die technische Migration der Anwendung hin zu einer MOHITO-basierten Implementierung dar und ist nicht mehr Teil der Migration zur Kompatibilität zur MOHITO-Referenzarchitektur.

Die genannten Migrationsschritte wurden in (B. Klatt E. G., 2013) beschrieben und im Rahmen der Demonstrator-Entwicklung angewandt. Das vorliegende Dokument beschreibt die Erkenntnisse, die aus der Migration zweier Bestandsanwendungen gewonnen wurden.

2 Migration der Datenhaltung nach MOHITO

Das MOHITO Framework stellt, wie eingangs beschrieben, einen kompletten Datenhaltungs-Stack mit Caching- und Synchronisationsmechanismen zur Verfügung. Dementsprechend muss in einem ersten Migrationsschritt sichergestellt werden, dass die Datenhaltungsschicht einer entsprechenden Bestandsanwendung durch das MOHITO Framework ersetzt werden kann.

2.1 Hintergrund: Datenhaltung und -zugriff im MOHITO-Framework

Grundlegendes Konzept in MOHITO sind die sogenannten Mohito Entitäten, welche die zu persistierenden Daten darstellen und entsprechende Datenzugriffsobjekte (DAO), welche die Funktionalitäten zum Zugriff auf Entitäten zur Verfügung stellen. Verwaltet werden die DAO durch entsprechende DAO Manager, die wiederum über Storage Manager erzeugt und verwaltet werden (siehe Abbildung 1).

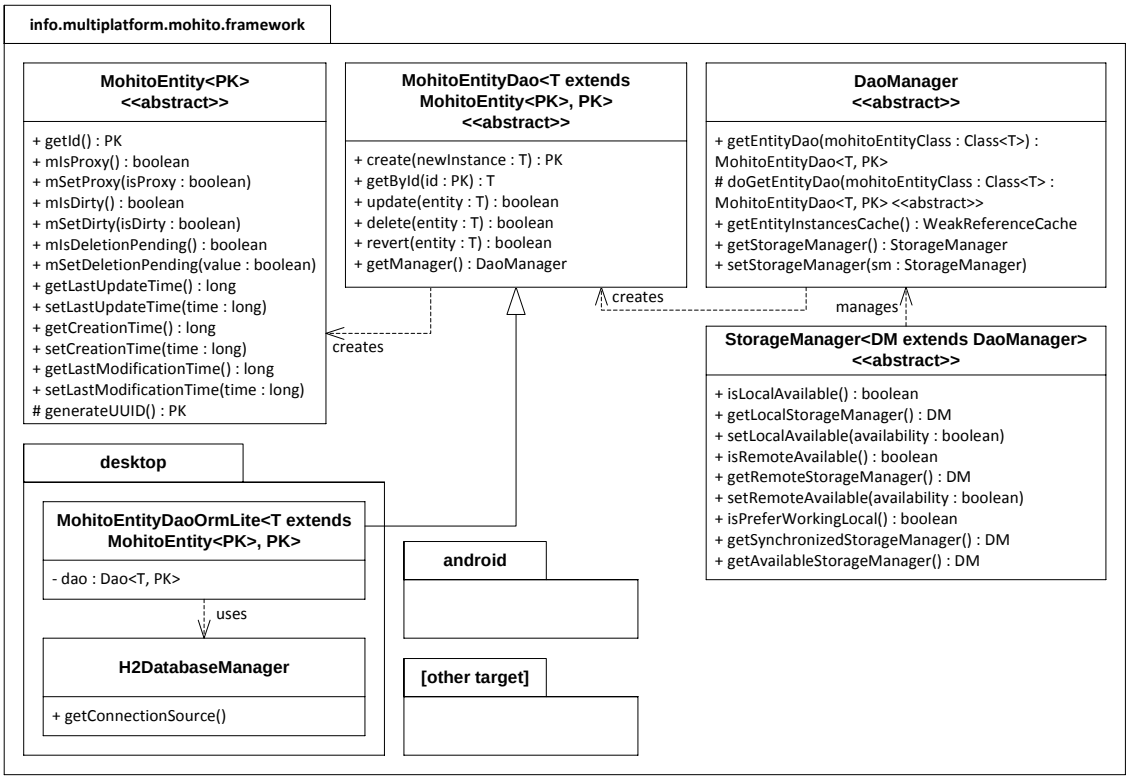

Abbildung 1 Überblick MOHITO Framework

Eine detaillierte Beschreibung des MOHITO Frameworks und seiner Schnittstellen stellt (E. Gaillus, 2013) dar.
Erster Schritt der Migration ist daher eine geeignete Abbildung von in der Bestandsanwendung existierenden Entitäten auf MOHITO Entitäten.

2.2 Kapselung Schreib- und Lesezugriffe

Um die Datenhaltungsschicht durch eine MOHITO Datenhaltungsschicht austauschen zu können, müssen alle Schreib- und Lesezugriffe auf lokal persistierte Daten in einer einheitlichen Schicht gekapselt sein (vgl. Abbildung 2 Verstreuter vs. gekapselter Datenbankzugriff). Unter Umständen muss hierbei auch das Datenzugriffskonzept angepasst werden.

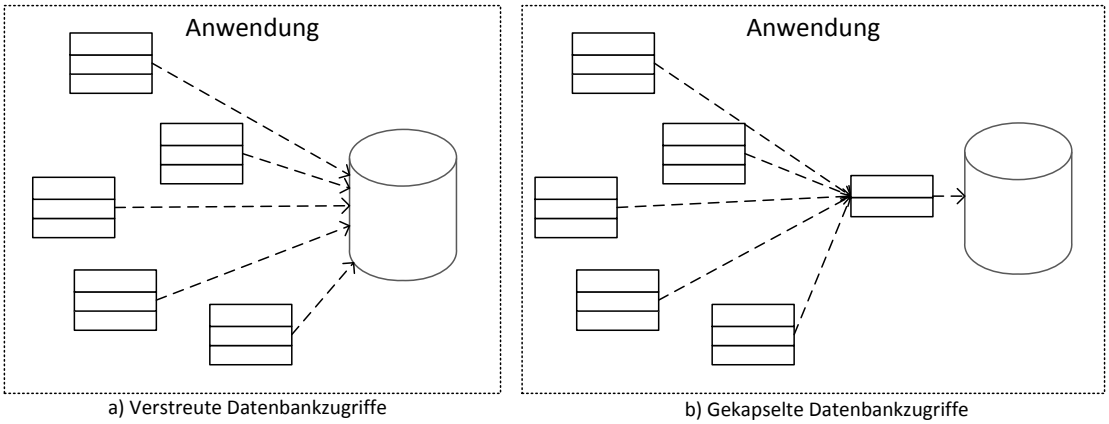

Abbildung 2 Verstreuter vs. gekapselter Datenbankzugriff

Das MOHITO Framework bietet nur synchrone Datenanfragen an. Grund dieser Designentscheidung ist das Framework einfach zu halten und nicht mit zusätzlicher Thread-Behandlung und Logik zu überfrachten. Selbige sollte insbesondere durch die Komplexität bei asynchronem Zugriff immer die in der restlichen Anwendung vorherrschenden Muster berücksichtigen. Da keine generische Abdeckung aller Muster möglich ist muss der synchrone Aufruf in einem Client entsprechend des verwendeten Musters asynchron gekapselt werden. Ist diese Erweiterung einmalig erfolgt, kann der asynchrone Zugriff in der gesamten Anwendung genutzt werden.

Es hat sich gezeigt, dass in den meisten Fällen, der Datenzugriff innerhalb einer Anwendung bereits sauber in einer Schicht gekapselt ist und in diesem Zusammenhang keine unerwarteten Probleme zu erwarten sind.

2.3 Kapselung Datenaustausch

Der Teil der zu migrierenden Anwendung, welcher sich mit dem Austausch von Daten beschäftigt, muss ebenfalls austauschbar gestaltet sein, um durch das MOHITO Framework ersetzt werden zu können. Das bedeutet, dass die komplette Datenaustauschlogik in einer Schicht gekapselt werden muss. Damit kann sie in einem weiteren Migrationsschritt ausgetauscht werden. Wichtig ist hierbei darauf zu achten, dass ein Datenaustauch über entfernte und daher langsame Verbindungen asynchron durchzuführen und gemäß der existierenden Anwendungskonventionen beispielsweise in einen eigenen Thread auszulagern ist. Der zu erwartende Datenumfang ist zu berücksichtigen und kann gegebenenfalls über Paging Mechanismen aufgeteilt werden.

2.4 Konkrete Migration am Beispiel eines Android Clients

Im Folgenden soll die Migration einer bestehenden Clientanwendung hin zur Nutzung des MOHITO Frameworks zur Datenhaltung und zum Datenaustausch genauer betrachtet werden. Hierzu soll zu Beginn ein generelles und den meisten Clients gemeinsames Entwurfsmuster analysiert und darauf basierend eine Migrationsstrategie entwickelt werden. Das nachstehend erläuterte Muster entstammt der Android-Welt und benutzt dementsprechend Android-spezifische Komponenten. Es kann aber in leicht modifizierter Form verallgemeinert und auf eine Vielzahl von Clients in unterschiedlichen Umgebungen angewandt werden. Die Festlegung auf eine spezifische Ausprägung erleichtert detailliertere Betrachtungen einer konkreten Implementierung und eine entsprechend Darstellung der Migrationsschritte anhand der im Projekt entstandenen Referenzimplementierungen.

Bei Client-Server Systemen ist die Kommunikationsschicht, welche den Austausch der Daten zwischen Client und Server organisiert, von besonderem Interesse. Die hier auftretenden Herausforderungen sind ein standardisierter Zugriff auf die von Servern angebotenen Dienste und das asynchrone Verarbeiten der Anfrageergebnisse. Diesen Herausforderungen wird in der Android-Welt häufig mit einem Entwurfsmuster begegnet, welches die eigentliche Kommunikation mit dem Server in einen *Service* auslagert und als Kommunikationsschnittstelle das *Content-Provider / Resolver* Prinzip nutzt.

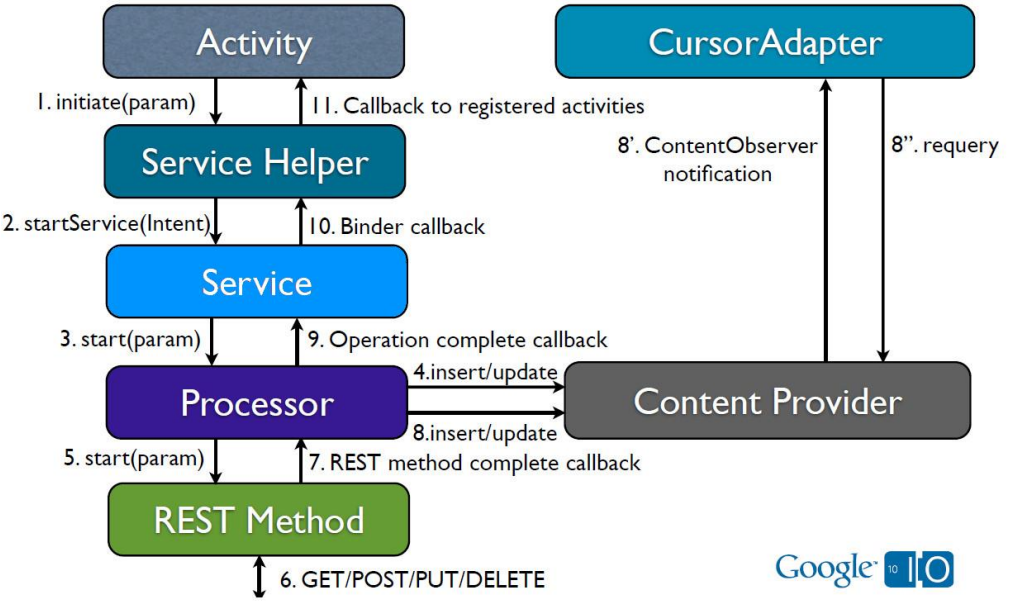

Abbildung 3 Zugriffs- und Datenkommunikation unter Android (Dobjanschi, 2010)

Abbildung 3 verdeutlicht den beschriebenen Zugriffs- und Datenkommunikationsmechanismus. Hervorzuheben ist die Mohito API Komponente, welche die eigentliche Kommunikation

zum Server über eine REST Schnittstelle realisiert und Synchronisationsmechanismen zwischen lokalen und entfernten Kopien eines Datenobjekts implementiert. Hier setzt MOHITO an, dessen API eine homogene und transparente Zugriffsschicht sowohl auf lokale als auch entfernte Daten zur Verfügung stellt. Das Design einer nach MOHITO migrierten Anwendung würde dementsprechend folgendermaßen aussehen.

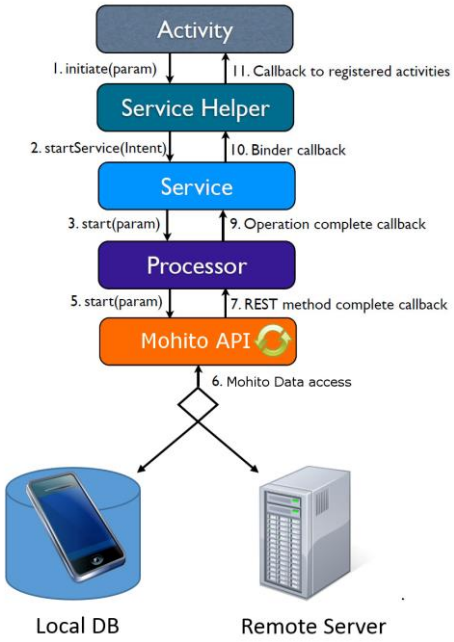

Abbildung 4 Datenzugriff mit MOHITO

Die Abbildung beschreibt den reinen, plattformunspezifischen Datenzugriff einer Client-Anwendung über die MOHITO API. Dieser Ansatz ist so auch in einer Android Umgebung einsetzbar, ohne die explizite Nutzung der Android-spezifischen Content-Provider / Resolver Kommunikationsschnittstellen. Das Content-Provider / Resolver Prinzip vereinfacht und vereinheitlicht als grundlegender Baustein der Datenkommunikation unter Android den Datenzugriff. Für einen Prozess und anwendungsübergreifende Datenkommunikation ist es sogar zwingend erforderlich.

Für einfache Anwendungen, die ihre Daten keinen anderen Prozessen oder Anwendungen zur Verfügung zu stellen ist die einfache Benutzung der MOHITO API, wie oben beschrieben, ausreichend. Sollten allerdings Anforderungen an verteilte Datenhaltung über Prozessgrenzen hinaus bestehen, oder einfach eine bestehende Anwendung migrieren wollen, welche schon auf dem Content-Provider / Resolver Prinzip basiert, ist ein Android-spezifischer ContentResolver in der MOHITO API Abstraktionsschicht zusätzlich zu implementieren. Dies ist einfach zu realisieren, da die MOHITO API und die ContentResolver Schnittstelle sehr ähnlich konzipiert sind und beide die grundlegenden CRUD Mechanismen implementieren. So kann ein ContentResolver alle an ihn gerichteten CRUD Operationen einfach an die MOHITO API weiterleiten und die entsprechenden Antworten wieder an den Aufrufer zurück lie-

fern. Solch ein ContentResolver kann als einfacher Proxy zur MOHITO API umgesetzt werden, was den Entwicklungsaufwand und die Komplexität in akzeptablen Grenzen hält und dennoch die Vorteile der MOHITO API nutzbar macht. Die MOHITO API unterstützt durch ihren Abstraktionsgrad eine einfache Anpassung an plattformspezifische Besonderheiten.

3 Migration der Serveranbindung

3.1 Kapselung Konfliktlösung

Um eine konsistente lokale Datenpersistenz und Cachelogik in Clientanwendungen gewährleisten zu können, müssen Daten, die während einer offline Phase modifiziert wurden, wieder mit den Datensätzen eines Servers, bzw. anderer Clients synchronisiert werden. Hierbei können Konflikte auftreten, die mit geeigneten Konfliktlösungsstrategien beseitigt werden müssen. Solche Strategien sind Bestandteil des MOHITO Frameworks und werden während entsprechender Synchronisationsvorgänge angewandt. Hat eine Bestandsanwendung eigene Konfliktlösungsstrategien implementiert müssen diese durch das MOHITO Framework ersetzt werden. Daher ist es sinnvoll in diesem Migrationsschritt sämtliche Synchronisationsvorgänge in einer Schicht zu kapseln, um sie später einfach ersetzen zu können. Ist eine von der Bestandssoftware genutzte Konfliktlösungsstrategie nicht durch MOHITO adäquat ersetzbar, bietet MOHITO dafür entsprechende Erweiterungspunkte an, um eigene Strategien zu realisieren.

3.2 Konkrete Umsetzung einer Serveranbindung

Die MOHITO Datenhaltungsschicht stellt für den Zugriff auf Daten zwei verschiedene Wege zur Verfügung. Der erste Weg ist der lokale Zugriff auf Daten, die auf dem gleichen System persistiert werden, auf dem auch die Anwendung selbst ausgeführt wird. Der zweite Weg unterstützt den Zugriff auf Daten, die von einem entfernten System (Server) bereitgestellt werden.

Bei der Migration einer beliebigen Clientanwendung nach MOHITO wird man immer mit der Frage konfrontiert sein, wie eine existierende Serverschnittstelle, die nicht für einen einzelnen Client angepasst oder erweitert werden soll, mit dem MOHITO Framework integriert werden kann. Im Unterschied zum lokalen Zugriff ist dadurch kein direkter Austausch von Mohito-Entities zwischen Client und Server, z.B. als serialisierte Java Objekte, möglich. Der existierende Server bietet stattdessen üblicherweise eine auf REST oder SOAP Prinzipien basierende Schnittstelle an, für die clientseitig

- Konvertierungsmechanismen zwischen dem vom Server verwendeten Darstellungsformat für Datenobjekte und der Clientrepräsentation (Mohito-Entities) bereitgestellt werden müssen.
- CRUD Operationen des MohitoEntityDAO (create, getById, getByCriteria, update, delete) auf passende Serveroperationen und -aufrufe abgebildet werden müssen.

Im Folgenden wird die Umsetzung der Serveranbindung eines Android xRM Clients an den CAS Open Server über dessen REST Schnittstelle vorgestellt, die JSON als Datenformat nutzt. Dieses Szenario ist aufgrund des leichtgewichtigen Protokolls besonders für mobile Clients relevant. Die vorgestellte Lösung lässt sich aufgrund ihres generischen Charakters leicht Verallgemeinern und auf andere Clientanwendungen übertragen.

3.2.1 Konvertierungsmechanismen

Die Umsetzung des Konvertierungsmechanismus erfolgt durch die Einführung eines Konverters (*MohitoEntityConverter*, Abbildung 5), der Instanzen von Mohito-Entities und JSON Objekten ineinander überführt, sowie eine Liste von JSON Objekten in eine Liste von Mohito-Entities. Dafür sind die jeweiligen *convert*-Methoden zuständig.

Die Konvertierung primitiver Datentypen zwischen JSON Properties und Attributen eines MohitoEntity wird durch den *JavaTypeConverter* realisiert. Die Methode *convertFromString* überführt eine primitive JSON Property in ein Java Objekt. Die *convert*-Methode überführt einen Java-Attributwert in ein Objekt, welches anschließend der vom Client verwendeten JSON-API zum Setzen des Feldes übergeben werden kann. Im Falle eines Datumswerts bildet diese Methode beispielsweise ein Objekt vom Typ *java.util.Date* auf einen String ab, dessen Format der vom Open Server für JSON verwendeten Repräsentation für Datumswerte entspricht.

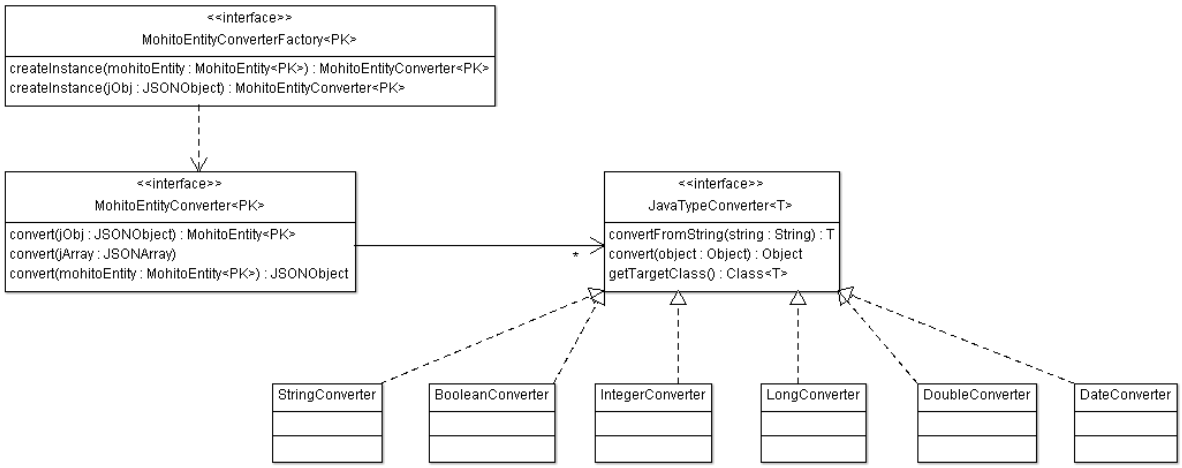

Abbildung 5 Konvertierungsframework zur Serveranbindung

Die Ermittlung des anzuwendenden JavaTypeConverters erfolgt durch Introspection der Attribute des zu erzeugenden MohitoEntity. Die Identifikation aufeinander abzubildender JSON Properties und MohitoEntity Attribute ist im Fall des Open Servers über Namenskonventionen (Namensgleichheit von JSON Property und MohitoEntity Attributen) möglich. Dadurch ist eine weitestgehend generische Implementierung der MohitoEntityConverter für den Open-Server möglich.

Der modellspezifische und damit zu generierende Teil des Konvertierungsframeworks beschränkt sich auf die Festlegung der Abbildung der vom Open Server in JSON gesetzten Typinformation (Property objectType, z.B. „ADDRESS“) auf die korrespondierende MohitoEntity Klasse (z.B. model.ADDRESS). Alternativ könnte durch eine geeignete Generierung größerer Teile des MohitoEntityConverter auf die Verwendung von Introspection und Namenskonventionen verzichtet werden.

3.2.2 Abbildung von CRUD Operationen

Der Zugriff auf Mohito-Entities innerhalb der Clientanwendung erfolgt, wie auch im lokalen Fall, über *MohitoEntityDao* des MOHITO Frameworks (vgl. Abbildung 6). Für entfernte Zugriffe wird eine Spezialisierung (*MohitoEntityOpenRemoteDao*) eingesetzt. Für deren Bereitstellung ist der *OpenRemoteDaoManager* zuständig, der von allen durch ihn erzeugten MohitoEntityOpenRemoteDao referenziert wird. Er fügt dem *DaoManager* des MOHITO Frameworks generische CRUD-Operationen für den entfernten Zugriff auf den Open Server hinzu, an die MohitoEntityOpenRemoteDao Aufrufe der durch sie angebotenen CRUD-Operationen delegieren.

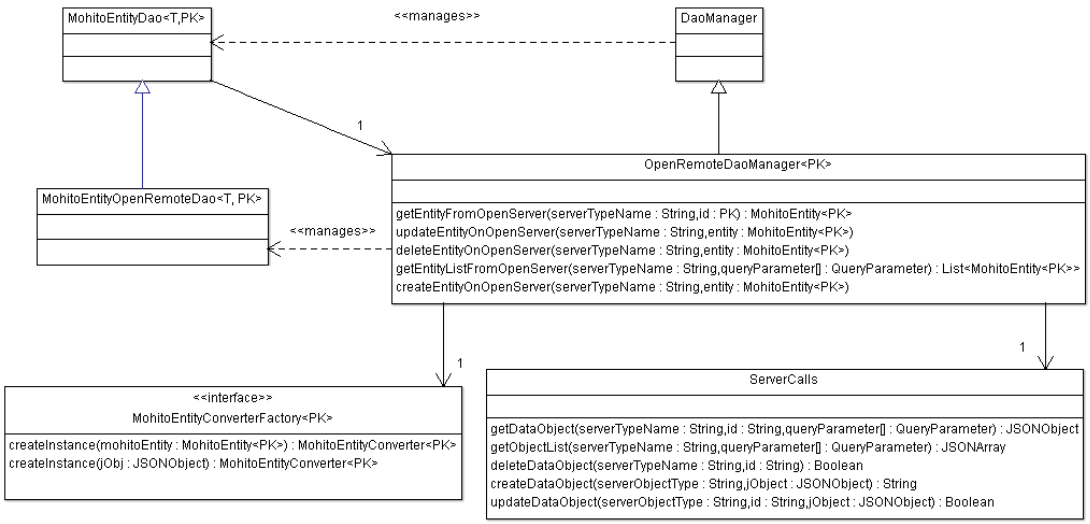

Abbildung 6 Klassen für den Remotezugriff auf den CAS Open Server

Für die eigentlichen REST Zugriffe auf den Open Server verwendet der OpenRemoteDaoManager die Util-Klasse *ServerCalls*. Diese Klasse stellt alle CRUD-Operationen des

Servers für die von ihm verwendeten JSON Objekte bereit. ServerCalls konstruiert entsprechenden REST Aufrufe, setzt sie ab und wertet die Antworten des Servers aus. Bei einer Migration auf das MOHITO Framework kann eine solche Klasse im Idealfall direkt aus der zu migrierenden Clientanwendung übernommen werden.

Für die Konvertierung zwischen MohitoEntity Objekten und den von ServerCalls erwarteten JSON Objekten nutzt der OpenRemoteDaoManager die durch die *MohitoEntityConverter-Factory* bereitgestellten Konvertierungsmechanismen (siehe 3.2.1).

4 Migration des Datenmodells

4.1 Abgleich Datenmodelle der Teilsysteme

Von zentraler Bedeutung sind die Datenmodelle der Systemlandschaft. Für den MOHITO Ansatz wird clientseitig von einem einheitlichen Datenmodell ausgegangen - das mit dem Datenmodell des Servers kompatibel aber nicht identisch sein muss. Um gemäß des MOHITO Ansatzes die Datenhaltungsschicht für verschiedene Plattformen generieren zu können, muss das Datenmodell in Ecore definiert sein.

4.2 Erstellung Ecore Datenmodell

Prinzipiell gibt es drei Wege, ein in Ecore definiertes Datenmodell zu erzeugen:

- Manuelle Erzeugung (Verwendung von Eclipse + EMF Modellierungswerkzeugen)
- Annotieren des Java-Codes existierender Datenklassen und anschließender Import über den Model Import Wizard
- Transformationen von XMl basierten Datenmodellen (Verwendung von XSLT oder Modell-zu-Modell Transformationen)

Für die manuelle Modellierung des Datenmodells eignet sich Eclipse zusammen mit den EMF Modellierungswerkzeugen.

Sofern das Datenmodell der Bestandsanwendung in Java explizit definiert ist, bietet sich die Verwendung des Ecore Model Import Wizards an, vgl. (A. Schwichtenberg G. H., 2013).

Im Rahmen des MOHITO Projektes wurden ferner Möglichkeiten der Transformation von XML basierten Datenmodellen in Ecore Datenmodelle untersucht. Dies ist klarerweise nur dann ein sinnvoller Weg, wenn das zu migrierenden Datenmodell sehr umfangreiches ist – andernfalls ist die manuelle Modellierung des Ecore Datenmodells der einfachere Weg.

Es hat sich gezeigt, dass der Einsatz von QVTo, einer Sprache für Modell-zu-Modell Transformationen) gut geeignet ist. In (A. Schwichtenberg G. H., 2013) ist beschrieben, wie eine

solche Transformation durchgeführt werden kann. Resultat der Transformation ist ein Datenmodell in Ecore, das mit den Annotationen der MOHITO DSL, sowie ggf. anwendungsspezifischen Annotationen so ausgezeichnet wird, dass auf der Basis dieses Modells alle für die Datenhaltung relevanten Informationen enthalten sind.

4.3 Datenmodellanreicherung

Ziel des MOHITO Ansatzes ist die Generierung der Datenhaltungsschicht für verschiedene Plattformen. Dies setzt voraus, dass die Information, welche Teile des Datenmodells auf welche Art persistiert werden sollen, bereits im Modell enthalten ist. Im Rahmen des MOHITO Projekts wurde für diesem Zweck eine eigene DSL definiert, die all jene Informationen, die generell für die Datenhaltung relevant sind, in Form von Annotationen umfasst, s. (A. Schwichtenberg G. H., 2013). Zusätzlich zu den generellen Datenhaltungsspezifischen Annotationen wurden für die beiden Anwendungsfälle (xRM App und Here-I-Am bzw. Library App) zusätzliche Annotationen definiert, über die anwendungsspezifische Informationen, beispielsweise zu Feldtypen der Datenbank ausgedrückt werden können.

Im Zuge der XML-nach-Ecore Transformationen können diese Annotationen klarerweise bereits verwendet werden, so dass direkt ein Ecore Datenmodell mit allen MOHITO relevanten Informationen resultiert.

5 Generierung und Austausch der Datenzugriffsschicht

Der Kernschritt der Migration ist der eigentliche Austausch der Datenzugriffsschicht der Bestandsanwendung durch den in MOHITO implementierten Datenhaltungs-Stack. Dazu muss zuerst ein passendes Datenmodell unter Verwendung des Ecore Diagram-Editors, entworfen werden. Auf dieser Basis können dann die MOHITO-spezifischen Annotationen angewandt werden, welche die Datenhaltungsschicht im Einzelnen beschreiben. Eine detaillierte Beschreibung des Vorgehens ist in der Beschreibung der Generatortemplates (Schwichtenberg, Groenda, Klatt, Küster, & Gailus, 2013) Kapitel 3 und (A. Schwichtenberg G. H., 2014) zu finden.

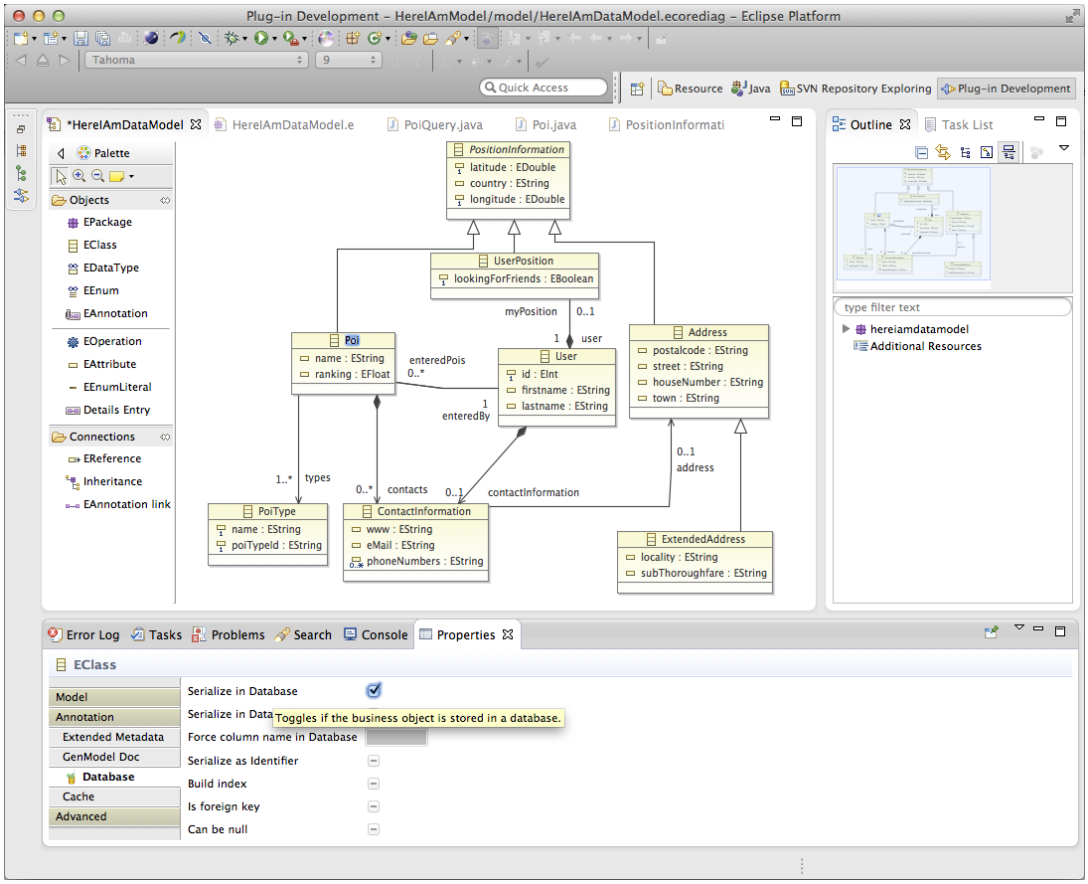

Abbildung 7 Integration der Annotationen in die Entwicklungsumgebung

Die definierten Annotationskategorien und Annotationen werden in der *Properties* Ansicht im unteren Teil der Entwicklungsumgebung dargestellt. Am Beispiel der Klasse *Poi* und der Kategorie *Database* ist dies in Abbildung *7* dargestellt. Verweilt der Cursor auf einem Eintrag wird der Beschreibungstext der Annotation als Tooltip angezeigt.

Nach Erstellung eines validen MOHITO-Datenmodells kann Code für die unterschiedlichen unterstützen Plattformen aus dem Modell heraus generiert werden.

Das Erzeugen von Code kann für einzelne Modelle mittels des mit MOHITO ausgelieferten Wizard angestoßen werden.

Die Erzeugung wird mit den folgenden Schritten durchgeführt:

1 Auswahl des MOHITO Domänenmodells (**Fehler! Verweisquelle konnte nicht gefunden werden.**)

2 Auswahl von *Export...* (**Fehler! Verweisquelle konnte nicht gefunden werden.**) und bestätigen mit *Next >*

3 Auswahl von *MOHITO Development* (**Fehler! Verweisquelle konnte nicht gefunden werden.**)

4 Auswahl von *Platform-dependent code* (**Fehler! Verweisquelle konnte nicht gefunden werden.**) und bestätigen mit *Next* >

5 Auswahl der Zielumgebung (**Fehler! Verweisquelle konnte nicht gefunden werden.**) und bestätigen mit *Next* >

6 Auswahl der lokalen Speichertechnik (**Fehler! Verweisquelle konnte nicht gefunden werden.**) und bestätigen mit *Next* >

7 Auswahl der entfernten Zugriffstechnik (**Fehler! Verweisquelle konnte nicht gefunden werden.**) und bestätigen mit *Finish*

Nach Schritt 7 wird das Projekt gemäß der in der DSL am Domänenmodell mittels MOHITO-Annotationen festgelegtem Namen erzeugt. Für das Referenzbeispiel ist dies das *HereIAm* Projekt.

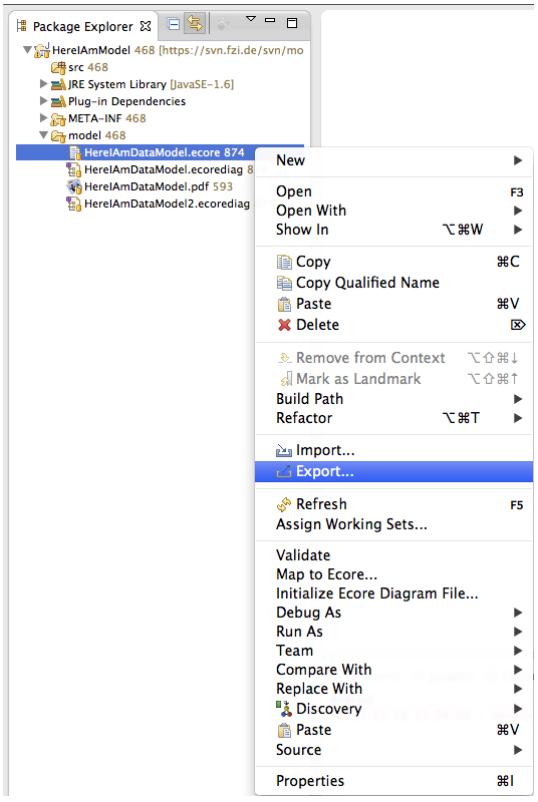

Abbildung 8 Auswahl des Domänenmodells und Export

Die Einstellungen des Generators können mittels der Annotationen der Kategorie *Generator* am Paket, welches alle Klassen des Domänenmodells enthält, übergeben werden. Die generierten Klassen sind im aktuellen Eclipse-Workspace sicht- und bearbeitbar. (Schwichtenberg, Hübsch, Groenda, & Gailus, 2013)

Der so generierte Code kann nun in einem ersten Schritt in das zu migrierende Projekt übernommen werden. Auf dieser Basis kann der Austausch der vorher gekapselten Datenzugriffsschicht vorgenommen werden. Dabei müssen unter Umständen verwendete Entwurfsmuster und Komponenten angepasst werden.

6 Literaturverzeichnis

(kein Datum). Von http://www.heise.de/developer/meldung/10-Jahre-Eclipse-Konsolidierung-des-Java-IDE-Markts-1370644.html abgerufen

A. Schwichtenberg, G. H. (2013). *L7.2 Dokumentation: verfeinerte DSL.*

A. Schwichtenberg, G. H. (2014). *Finale Version der Generatortemplates und des Modellinterpreters für die unterstützten Plattformen.*

B. Klatt, E. G. (2013). *L 5.1: Migrationsstrategie für Bestandsanwendungen zur MOHITO Referenzarchitektur.*

B. Klatt, K. (2013). *L 3.2: Erweiterte Referenzarchitektur für Multi-Plattformhomogenisierung von Datenhaltungsschichten.*

Dobjanschi, V. (2010, 5 20). Developing Android REST Client Applications.

E. Gaillus, G. H. (2013). *Frameworkentwicklung, Multi-Plattform.*

Eclipse: Ecore. (kein Datum). Von http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details abgerufen

Eclipse: EMF. (kein Datum). Von http://www.eclipse.org/modeling/emf/?project=emf abgerufen

Eclipse: EMF . (kein Datum). Von http://www.eclipse.org/modeling/emf/?project=emf#emf abgerufen

Gailus, E., Klatt, B., Hübsch, G., Krogmann, K., & Schwichtenberg, A. (2013). *L 2.1: Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik.*

Heise: Ecplise Markt. (kein Datum). Von http://www.heise.de/developer/meldung/10-Jahre-Eclipse-Konsolidierung-des-Java-IDE-Markts-1370644.html abgerufen

(kein Datum). *L7.1 Dokumentation DSL.* Karlsruhe.

Schwichtenberg, A., Gailus, E., Klatt, B., & Hübsch, G. (2013). *L9.1 QM-Maßnahmen und Tests.*

Schwichtenberg, A., Groenda, H., Klatt, B., Küster, M., & Gailus, E. (2013). *Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform.*

Schwichtenberg, A., Hübsch, G., Groenda, H., & Gailus, E. (2013). *L8.2: FInale Version der Generatortemplates und des Modellinterpreters für die unterstützten Plattformen.*

Schwichtenberg, A., Küster, M., & Gailus, E. (2013). *L7.2 Dokumentation: DSL.*

Wikipedia: ERM. (kein Datum). Von http://de.wikipedia.org/wiki/Entity-Relationship-Modell abgerufen

Wikipedia: UML. (kein Datum). Von http://de.wikipedia.org/wiki/Unified_Modeling_Language abgerufen

Wikipedia: XMI . (kein Datum). Von http://de.wikipedia.org/wiki/XML_Metadata_Interchange abgerufen

Wikipedia: Xtend . (kein Datum). Von http://de.wikipedia.org/wiki/Xtend abgerufen

