Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 7

Dokumentation: DSL

Autoren: Antonia Schwichtenberg (CAS),

Martin Küster (FZI),

Gailus (B2M)

Fertiggestellt am: 30.06.2013

Schlagworte: DSL, Modellierung, Datenhaltung

Projektpartner

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

© MOHITO Projektkonsortium. Alle Rec

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
1.0	Küster	17.06.2013	Initiale Fassung
1.1	Schwichten-berg	20.06.2013	Ausarbeitung Kap. 1,2
1.2.	Schwichten-berg	26.6.2013	Ausarbeitung Einleitung Kap. 3 und Kap. 4
1.3	Gailus	27.6.2013	Ausarbeitung Kap. 3
1.4	Klatt	27.6.2013	1. Review
1.5	Hübsch	28.6.2013	2. Review
2.0	Schwichten-berg	28.6.2013	Finale Version

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Inhalt

Abbildungen

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

1 Einleitung

Ziel des Projektes MOHITO ist die modellgetriebene Generierung von Datenhaltungsschichten für verschiedene – vor allem mobile – Plattformen. Der Generator liest das Modell ein und generiert auf dessen Grundlage den Plattform-spezifischen Code. Das Modell wird in einer geeigneten Sprache (der DSL – Domain Specific Language) geschrieben.

Im Folgenden wird die im Rahmen des Projektes MOHITO entwickelte DSL (Domain Specific Language) beschrieben. Das vorliegende Dokument wird durch Erkenntnisse während der Umsetzung des MOHITO Ansatzes in den Anwendungsfällen weiter ergänzt und als finale Version [1] veröffentlicht. Die Notwendigkeit einer eigenen DSL ergibt sich aus der Tatsache, dass typische Modellierungs-Sprachen wie Ecore (vgl. [2] und [3]), UML (Unified Modelling Language, vgl. [4]) oder ER/ERM (Entity Relationship Modell, vgl. [5]) nicht aussagekräftig genug sind, um daraus plattformspezifischen Code für die Datenhaltung generieren zu können. Diese allgemeinen Modellierungssprachen verfügen nicht über jene Sprachelemente, die gebraucht werden um zu einem Datenmodell Informationen über die Datenhaltung zu definieren. Das Hauptziel war also auf einem geeigneten Metamodell aufzubauen und dieses um die nötigen Elemente zu ergänzen. Das Metamodell definiert, welche allgemeinen Elemente in der DSL vorkommen können und wie diese in Beziehung zueinander stehen. Vergleicht man die DSL mit der deutschen Sprache, so würde im Metamodell festgelegt sein, welche grammatikalischen Einheiten es gibt, z.B. Worte, wobei diese Substantive, Verben, Adjektive sein können, sowie Punkte, Kommata etc.

Im Rahmen des MOHITO Projektes wurde:

1. ein Metamodell festgelegt
2. Annotationen definiert
3. eine DSL entwickelt, die die Annotationen enthält und Aussagen, an welche Elemente die Annotationen angehängt werden dürfen
4. Für die beiden Anwendungsfälle – die offline-fähige xRM App (CAS) und die "Here-I-am-App" (B2M) wurden Applikationsmodelle entworfen, die vor allem die annotierten Datentypen umfassen

In Kapitel 2 Architektur der DSL als Annotationssprache werden die Zusammenhänge und der Aufbau der wesentlichen Modellierungsteile im Detail erläutert. Alles zusammen wird von dem Generator ausgewertet (der im Rahmen von Arbeitspaket 8 entwickelt wird), um die Datenhaltungsschicht, bzw. Teile davon zu generieren. Die Annotationen der DSL bestimmen u.a., wie die Datenobjekte und Attribute gespeichert und synchronisiert werden.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Eine wichtige Design Grundlage für die in Kapitel 2 Architektur der DSL als Annotationssprache vorgestellte Lösung ergibt sich aus den im Rahmen von Arbeitspaket 2 definierten Anforderungen (s. [6]). Die dort definierten Anforderungen betreffen einerseits die beiden Demonstratoren "offline-fähige xRM App" und "Here-I-am-App", sowie generelle Anforderungen an das im Rahmen von Arbeitspaket 6 entwickelte Framework. Als generelle, übergreifende Anforderungen gelten: Integrierbarkeit, Erweiterbarkeit, Wartbarkeit und Flexibilität. Ferner wurden in [7] Kriterien definiert, wie diese Aspekte geprüft werden können. Die Entwicklung der DSL war und ist daher maßgeblich durch die Arbeiten in den anderen Arbeitspaketen bestimmt und wird andererseits auch die Ergebnisse dieser beeinflussen.

2 Architektur der DSL als Annotationssprache

Eine grundlegende Entscheidung beim Entwurf der DSL war die Verwendung, bzw. der Aufbau auf EMF. EMF (auch als EMF Core bekannt) ist ein weitverbreitetes Modellierungs- und Code-Generierungs-Framework, das insbesondere im Umfeld der Entwicklungsumgebung von Eclipse von tragender Bedeutung ist. Die Entwicklungsumgebung Eclipse ist mit 6 Millionen Entwicklern, die diese benutzen die am weitesten verbreitete Entwicklungsumgebung im Java Umfeld – danach kommen IntelliJ IDEA und NetBeans (vgl. [8] und [9]). Eclipse selbst setzt auf EMF auf und bietet auch deswegen eine umfassende Unterstützung sowohl für die Entwurfsphase als auch für die Laufzeit.

Die wesentlichen Teile von EMF (core) sind:[1]

1. Ein Metamodell (Ecore) zum Beschreiben von Modellen. Im Metamodell ist definiert, welche (Sprach)-Konstrukte es überhaupt gibt, z.B.: Package, Class, Data Types, Feature (Reference oder Attribute), etc. Aufgrund der Nähe zu UML, der am weitesten verbreiteten Modellierungssprache, eignet sich Ecore als Modellierungssprache, da somit der Lernaufwand gering ist.
2. Laufzeitunterstützung, sodass die Modelle ausgewertet und auf ihnen gearbeitet werden kann
3. Change-Notification, also ein Framework zur Benachrichtigung von Modell Elementen, wenn – zur Laufzeit – Änderungen aufgetreten sind
4. Persistence Support, also Unterstützung zum Speichern und Serialisieren der Modelle im XMI (XML Metadata Interchange) Format
5. API zur generellen Manipulation von EMF Objekten

[1] Vgl. [13]

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Hiervon werden im MOHITO Projekt primär Teil 1 als Grundlage der DSL Entwicklung und zur Modellierung durch den Entwickler, Teil 2 zur Verarbeitung der Modelle im Generator und Teil 4 zur Speicherung und zum Austausch der Modelle verwendet.

Diese von EMF (Core) zur Verfügung gestellten Mechanismen ermöglichen a) Java Klassen zu erzeugen, b) eine Menge an Adapter Klassen zu erzeugen, die vor allem zur Ansicht benötigt werden und c) einen einfachen Editor zu generieren, der das Bearbeiten des Modells ermöglicht. Die durch EMF zu interpretierenden Modelle können in verschiedenen Sprachen ausgedrückt werden. Im MOHITO Projekt XMI (MXL Metadata Interchange, vgl. [10]) als Format für die Modellierung verwendet.

Aufgrund der weiten Verbreitung und einer ausgiebigen Evaluierung bestehender Modellierungssprachen und -werkzeuge hat sich EMF als Framework und Ecore als Metamodell als Basis für die Modellierung im MOHITO Projekt herauskristallisiert. Da die von EMF und dem Ecore Modell unterstützen Elemente – wie bereits erwähnt – für die Generierung von Datenhaltungsschichten nicht hinreichend sind, wurden sie um datenhaltungsspezifische Annotationen ergänzt; dieser Ansatz ist im Ecore Modell bereits vorgesehen. Kapitel 0 Für die Erreichung der Ziele des MOHITO Projektes wird kein Modell zur Laufzeit benötigt, da die Datenhaltungsschicht von dem Generator in der Entwicklungsphase generiert wird. Das Arbeiten auf den Daten erfolgt durch das im Rahmen von Arbeitspaket 6 entwickelte Framework, das gewissermaßen klassisch – also nicht modellbasiert – implementiert wurde.

Die DSL ist eine spezifische Sprache, die vorgibt welche Elemente zur Definition eines Applikationsmodells zur Verfügung stehen. Das Modell wird, um es dem Generator als Input zu geben, zu Text transformiert. Für diesen Prozess wird im MOHITO Projekt auf Xtend aufgesetzt. Weitere Details zur Transformation und die Entwicklung des Generator werden in beschrieben.

Zum Entwerfen und Bearbeiten von Ecore Modellen stehen durch EMF verschiedene Editoren zur Verfügung. Abbildung 2: Modellierungseditoren gibt einen Überblick über die in Eclipse mit EMF zur Verfügung stehenden Editoren.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

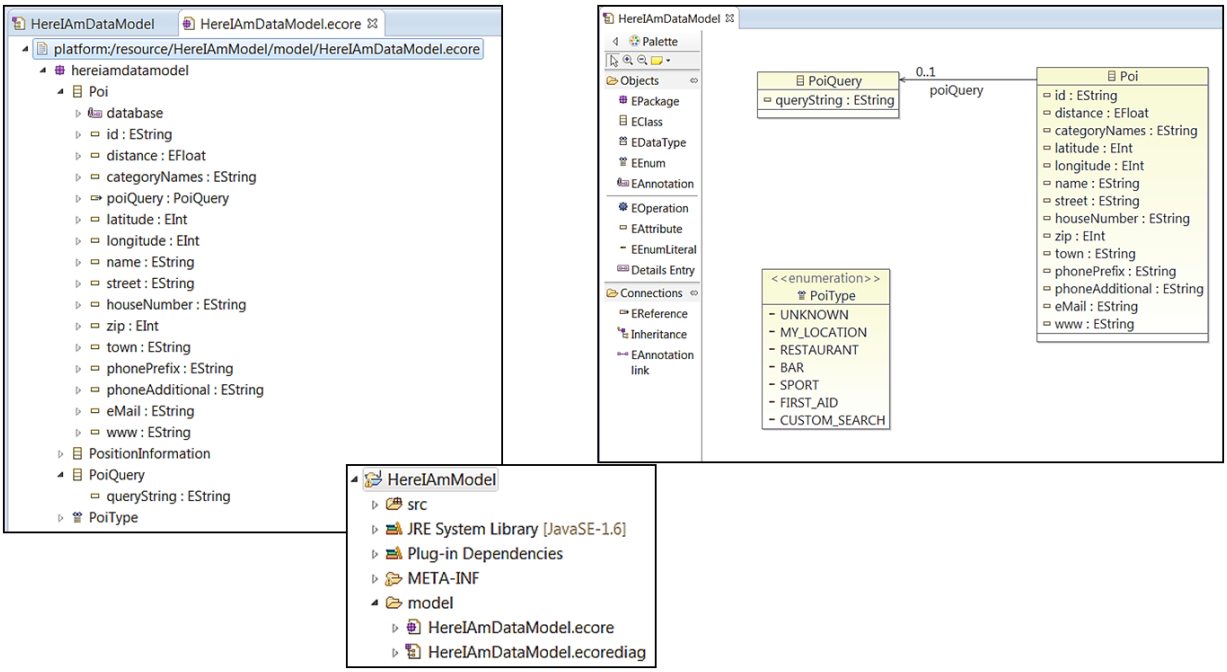

Abbildung 2: Modellierungseditoren

MOHITO Annotations – Metamodell der DSL gibt einen Überblick über die im Rahmen des MOHITO Projektes definierten Annotationen und ihre Bedeutung für die Generierung der Datenhaltungsschichten.

Die folgende Abbildung 1: Modellierungssprache gibt einen Überblick über die bereits erläuterten Zusammenhänge von Metamodell, DSL (Annotationen), Applikationsmodell und Laufzeitmodell.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

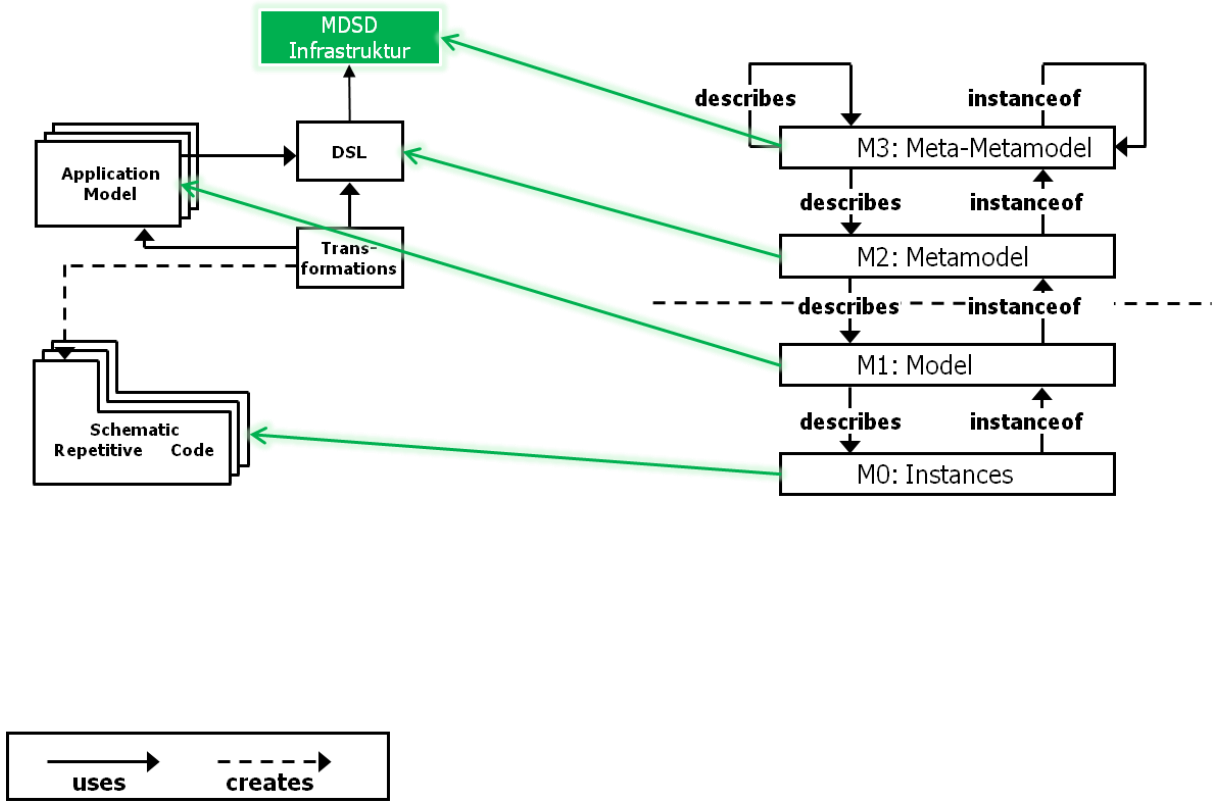

Abbildung 1: Modellierungssprache

Für die Erreichung der Ziele des MOHITO Projektes wird kein Modell zur Laufzeit benötigt, da die Datenhaltungsschicht von dem Generator in der Entwicklungsphase generiert wird. Das Arbeiten auf den Daten erfolgt durch das im Rahmen von Arbeitspaket 6 entwickelte Framework, das gewissermaßen klassisch – also nicht modellbasiert – implementiert wurde.

Die DSL ist eine spezifische Sprache, die vorgibt welche Elemente zur Definition eines Applikationsmodells zur Verfügung stehen. Das Modell wird, um es dem Generator als Input zu geben, zu Text transformiert. Für diesen Prozess wird im MOHITO Projekt auf Xtend[2] aufgesetzt. Weitere Details zur Transformation und die Entwicklung des Generator werden in [11] beschrieben.

Zum Entwerfen und Bearbeiten von Ecore Modellen stehen durch EMF verschiedene Editoren zur Verfügung. Abbildung 2: Modellierungseditoren gibt einen Überblick über die in Eclipse mit EMF zur Verfügung stehenden Editoren.

[2] S. [12]

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

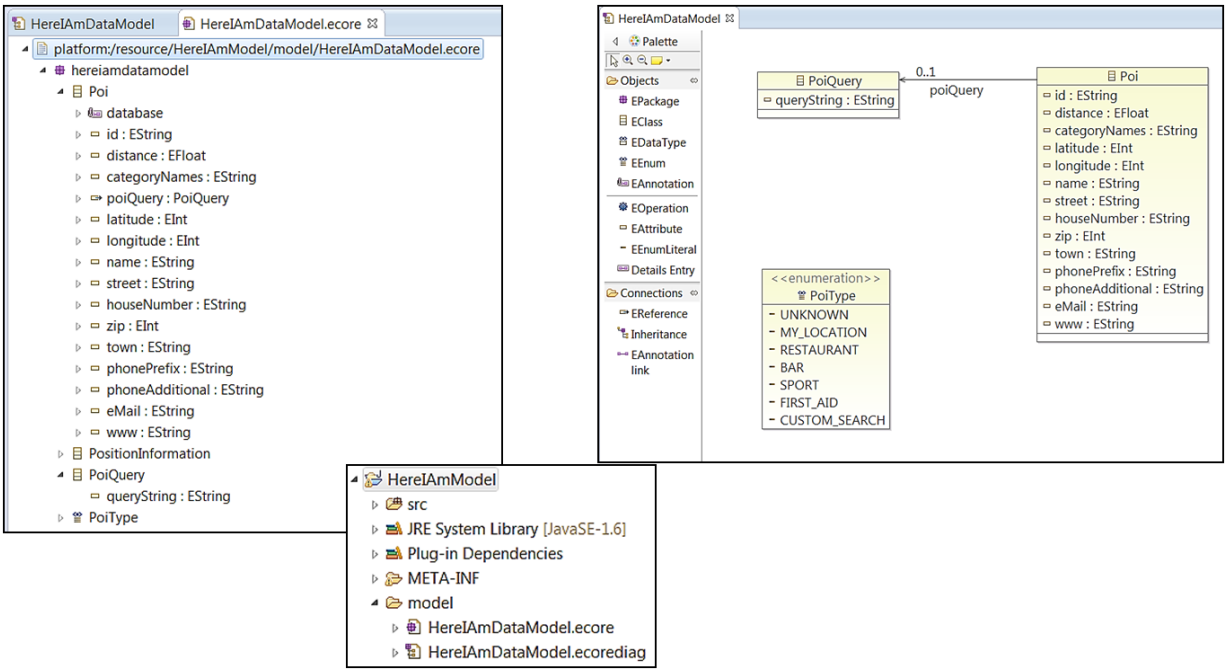

Abbildung 2: Modellierungseditoren

3 MOHITO Annotations – Metamodell der DSL

Wie bereits erwähnt, wurde im Rahmen des Projektes auf Annotationen als Erweiterung von Ecore und somit als Modellierungssprache gesetzt. Annotationen haben den Vorteil, dass sie sehr einfach erweiterbar und flexibel sind. Ferner können sie gerade im Zusammenhang mit Ecore "out of the box" (also ohne zusätzliche Module) genutzt werden. Annotationen haben aber zwei generelle Nachteile:

1. Annotationen können nicht strukturiert werden; es gibt keine Hierarchie und keine Klassifizierung von Annotationen.
2. Annotationen sind einfache Textfelder; es gibt vorerst keine Entwicklungsunterstützung. Hierin liegt potentiell eine beachtliche Fehlerquelle, da Entwickler nicht wissen, welche Annotationen zur Verfügung stehen und was genau sie bedeuten.

Aus diesem Grund wird im MOHITO Projekt ein Ansatz verfolgt, der sich als „Annotation Definition“ Ansatz bezeichnen lässt. Hierbei geht es zum einen darum, Mechanismen zu definieren, die im Zusammenspiel mit der Entwicklungsumgebung Eclipse verwendet werden können, um Annotationen zu strukturieren. Zum anderen geht es darum, eine Sprache zu definieren, mit der sich ausdrücken lässt, welche Annotationen an welche Ecore Elemente angehängt werden können. Wesentliche Elemente dieser DSL sind:

- Namespace (z.B. für Datenbankbezogenen Annotationen)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

- Scope (z.B. die Ecore Elemente eclass, eAttribute, etc.)
- Key (Innerhalb eines Namespaces eindeutiger Identifikationsschlüssel, der z.B. von dem Generator erkannt wird)
- Type (zur Definition der möglichen Eingaben des Entwicklers, z.B. Boolean, String, etc.)

Ferner wurde basierend auf dem Konzept von Eclipse Plugin Extension Points eine Infrastruktur entwickelt, um eigene Annotationen definieren zu können. Das hierfür entwickelte Plugin liefert entsprechend der oben genannten DSL Definitionsobjekte zurück, die von den MOHITO Modellierungstool und Generatoren verarbeitet werden können.

Ergänzt wird die Infrastruktur um einen sogenannten Annotation View, der basierend auf der Eclipse RCP (Rich Client Platform) entwickelt wurde. Der Annotation View ermöglicht dem Benutzer die Angabe von Annotationen indem er einfach ein Element im Modellierungstool auswählt und passend hierzu ein Formular zur Angabe der möglichen Annotationen angeboten bekommt. Dieses Formular wird dynamisch aus den, in der Eclipse Entwicklungsumgebung aktuell registrierten "Annotation Definitions", abgeleitet.

Scope	Key	Description	Type
`http://www.multiplatform.info/mohito/annotations/database`			
Class	databasetable	Zeigt an, dass eine Datenklasse als eine Tabelle in der Datenbank repräsentiert wird	Boolean
Attribute	Databasefield	Zeigt an, dass ein Klassenattribut als Feld der zugehörigen Datenbank Tabelle repräsentiert wird	Boolean
Attribute	databasefield. Columnname	Definiert den Namen der Datenbankspalte des entsprechenden Klassenattributes	String
Attribute	databasefield.id	Zeigt an, dass ein Feld Teil des Tabellenschlüssels ist.	Boolean
Attribute	databasefield.index	Zeigt an, dass das zugehörige Datenbankfeld von der Datenbank indiziert werden soll.	Boolean
Attribute	databasefield.foreign	Definiert das zugehörige Datenbankfeld als Fremdschlüssel	Boolean
Attribute	databasefield.nullable	Zeigt an, dass das zugehörige Datenbankfeld NULL Werte enthalten darf	Boolean
Attribute	databasefield. Generatedid	Markiert ein Datenbankfeld als generierte ID	Boolean

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

http://www.multiplatform.info/mohito/annotations/cache			
Class	cache.cachedobject	Markiert eine Klasse als cachbares Objekt	Boolean
Attribute	cache.key	Markiert ein Attribut als den Identifikator zu einem gecachten Objekte.	Boolean

Tabelle 1: Übersicht der MOHITO Annotationen

3.1 Annotationen für Datenbankeinträge

Die Annotationen für die Datenbankeinträge werden verwendet, um auszudrücken wie die Daten des Applikationsmodells gespeichert werden sollen. Zur Speicherung der Daten, werden relationale Datenbanken verwendet. Dementsprechend beziehen sich die Annotationen – wie in Tabelle 1: Übersicht der MOHITO Annotationen beschrieben – auf Tabelleneigenschaften, bzw. -elemente relationaler Datenbanken. So können beispielsweise die Namen der Datenbankspalten definiert werden, auf welche die Attribute der Datenobjekte gemappt werden. Desweiteren können Primärschlüssel für Datenbanktabellen festgelegt, oder Felder als indizierbar gekennzeichnet werden. Auch lassen sich Fremdschlüssel Beziehungen durch Annotationen abbilden.

Die Definition, für welche Art von relationaler Datenbank (z.B. SQLite für Android und iOS Plattformen) die Datenhaltungsschicht generiert werden soll, wird im Generator konfiguriert.

3.2 Annotationen für den Cache

Wie auch in [6] beschrieben, soll es möglich sein, alle bereits verwendeten Daten offline zur Verfügung zu stellen (einfaches Caching). Hierfür wird vom Framework eine Verdrängungsstrategie zur Verfügung gestellt, die dafür sorgt, dass der Datenspeicher des Endgerätes nicht überläuft. Desweiteren sollen die als Prefetching bezeichneten Mechanismen bereitgestellt werden, die ein komplexeres Caching ermöglichen. Beim Prefetching werden gezielt bestimmte Daten offline zur Verfügung gestellt. Über die in Tabelle 1: Übersicht der MOHITO Annotationen definierten Annotationen wird die durch MOHITO realisierte Cacheimplementierung gesteuert. Datenobjekte können durch die entsprechende Annotation als cachebar definiert werden. Auch die zur internen Cachelogik notwendigen Cache-Schlüssel lassen sich so festlegen.

Im Zuge der Weiterentwicklung der Prototypen und aufgrund der damit gewonnenen Erkenntnisse werden weitere Annotationen hinzukommen und die bestehenden werden u.U. inhaltlich verfeinert. Dies wird in [1] beschrieben.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

4 Fazit und weiteres Vorgehen

Zu Testzwecken wurde das Applikationsmodell der Here-I-am-App mit den aktuell zur Verfügung stehenden Annotationen annotiert und als Ecore Modell instanziiert. Ein wesentliches Resultat dieser Umsetzung ist, dass der hier beschriebene Ansatz (Ecore um datenhaltungsspezifische Annotationen zu erweitern) praktikabel ist. Die Modellierung war durch die entwickelte Infrastruktur (Eclipse Erweiterungen) intuitiv und mit nur geringem Zeitaufwand zu bewerkstelligen. Im Rahmen der prototypischen Umsetzung für die offline-fähige xRM App, deren Datenmodell noch umfangreicher ist, werden sich weitere Erkenntnisse gewinnen lassen, die in dem Folgedokument beschrieben werden [1]. Zusätzlich werden die Arbeiten am Generator weitere Erkenntnisse für die DSL und dem hier verfolgten Ansatz bringen. Basierend auf dem aktuellen Stand lässt sich jedoch bereits jetzt festhalten, dass der hier dargestellte Ansatz sehr zufriedenstellend ist und zu guten Ergebnissen führt.

5 Literaturverzeichnis

[1] „L7.2 Dokumentation: verfeinerte DSL“.

[2] „Eclipse: EMF,“ [Online]. Available: http://www.eclipse.org/modeling/emf/?project=emf.

[3] „Eclipse: Ecore,“ [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details.

[4] „Wikipedia: UML,“ [Online]. Available: http://de.wikipedia.org/wiki/Unified_Modeling_Language.

[5] „Wikipedia: ERM,“ [Online]. Available: http://de.wikipedia.org/wiki/Entity-Relationship-Modell.

[6] „L 2.1: Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik“.

[7] „L9.1 QM-Maßnahmen und Tests“.

[8] [Online]. Available: http://www.heise.de/developer/meldung/10-Jahre-Eclipse-Konsolidierung-des-Java-IDE-Markts-1370644.html.

[9] „Heise: Ecplise Markt,“ [Online]. Available: http://www.heise.de/developer/meldung/10-Jahre-Eclipse-Konsolidierung-des-Java-IDE-Markts-1370644.html.

[10 „Wikipedia: XMI,“ [Online]. Available: http://de.wikipedia.org/wiki/XML_Metadata_Interchange.

[11 „L8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform“.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

[12 „Wikipedia: Xtend,“ [Online]. Available: http://de.wikipedia.org/wiki/Xtend .

[13 „Eclipse: EMF,“ [Online]. Available:
http://www.eclipse.org/modeling/emf/?project=emf#emf.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.