Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 6

Frameworkentwicklung, Multi-Plattform

Autoren: E. Gailus (B2M),

G. Hübsch (CAS),

A. Schwichtenberg (CAS),

H. Groenda (FZI)

Fertiggestellt am: 30.09.2013

Schlagworte: Frameworkentwicklung, Architekturdokumentation

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
0.1	Eric Gailus	03.09.2013	Initiale Fassung
0.2	Eric Gailus	20.09.2013	Erstellung der „Javadoc“ Dokumentation
0.3	Groenda	26.09.2013	Kapitel 3
0.4	Groenda	27.09.2013	Kapitel 5
0.5	Eric Gailus	27.09.2013	Kapitel 2
0.6	Eric Gailus	27.09.2013	Kapitel 6
0.41	Groenda	27.09.2013	Review & Typos

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Inhalt

Inhalt 3

Abbildungen 4

1 Einleitung 5

2 Architekturprinzipien und Entwurfsentscheidungen 6

2.1 ORM vs. SQL 6

2.2 Warum wir keine Threads im Framework verwenden 8

2.3 Warum wir kein EMF verwenden 8

3 Framework-Architektur 9

4 Detaillierte API Beschreibung 11

5 Umgang am Beispiel 52

6 Zusammenfassung 54

7 Literaturverzeichnis 55

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Abbildungen

Abbildung 1: Frameworkanteile und deren Beziehung 9

Abbildung 2: Konzept für den projektunabhängigen Zugriff im Framework 11

Abbildung 3: HereIAmModel zur Beschreibung der Position interessanter Orte und Aufenthaltsposition von Freunden..52

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

1 Einleitung

Das vorliegende Dokument stellt die Architekturdokumentation des Mohito-Frameworks dar. Dies ist ein Framework zur homogenisierten Datenhaltung bei der Entwicklung von Multi-Plattformanwendungen. Die Architektur des Frameworks basiert auf den Dokumenten aus Arbeitspaket 3. Diese Dokumente zeigen eine generelle Referenzarchitektur für das Problem der homogenisierten Datenhaltung auf. Das vorliegende Dokument stellt eine Fortführung und Erweiterung des Dokuments 6.1.2 (Mohito, Frameworkentwicklung, Multi-Plattform, 2013) aus Arbeitspaket 6 dar. Somit ist in diesem Dokument auch sehr schön die Weiterentwicklung des Mohito-Frameworks nachvollziehbar, da einige zuvor getroffene Entwurfsentscheidungen revidiert, oder an die Bedürfnisse der Softwaregenerierung angepasst werden mussten. War die erste Version des Frameworks noch weitestgehend im Hinblick auf die Notwendigkeiten und Anforderungen der Referenzimplementierung entstanden, ist die hier weitestgehend finale Version des Frameworks um die Anforderungen der eigentlichen Softwaregeneration erweitert bzw. angepasst.

Das vorgestellt Framework implementiert alle in Arbeitspaket 2 identifizierten Anforderungen, die wiederum Einfluss auf die Referenzarchitektur und die hier beschriebenen Architekturentscheidungen hatten. Es stellt den nicht-generierten Teil der Datenhaltungsschicht in einer plattformspezifischen Ausprägung dar, realisiert Offlinesynchronisation und – Datenhaltung und kann auf variable Konnektivität und mögliche Bandbreitenbeschränkung reagieren.

In diesem Kapitel sollen zuerst unterschiedliche Architekturprinzipien diskutiert werden, um dann im Folgenden die Entscheidungen der gewählten Architektur zu erklären. Des Weiteren wird das Mohito-Framework genauer in seiner Funktionalität vorgestellt und in Form eines aus dem Quellcode erzeugten „Javadoc“ Dokuments detailliert beschrieben. Abschließend wird ein Beispiel angeführt an Hand dessen die Nutzung des in den vorangegangenen Kapiteln beschriebenen Frameworks veranschaulicht wird.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

2 Architekturprinzipien und Entwurfsentscheidungen

Das Ziel des Mohito-Projekts ist die einheitliche und zentrale, modellgetriebene Entwicklung des Datenmodells bzw. der Datenhaltungsschicht einer Anwendung und die Erzeugung von Programmierschnittstellen, die Entwicklern bei der Erstellung eines Softwaresystems den konsistenten Zugriff auf diese Datenhaltungsschicht ermöglichen. Die Modelle sollen eine plattformübergreifende Lösung beschreiben, zu dem das Mohito-Framework den nicht-generierten Teil der Datenhaltungsschicht in plattformspezifischer Ausprägung vorhält, Offlinesynchronisation und – Datenhaltung ermöglicht, sowie auf variable Konnektivität und mögliche Bandbreitenbeschränkung reagiert. Die entsprechenden Anforderungen, welche die Mohito Plattform erfüllen soll, wurden in Dokument *L 2.1: Dokumentation der Anwendungsszenarien* (Mohito, Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik, 2013), spezifiziert. Basis der Frameworkentwicklung sind die Überlegungen, die im Dokument L 3.2 (Mohito, Erweiterte Referenzarchitektur für Multi-Plattformhomogenisierung von Datenhaltungsschichten, 2013) angestellt wurden. Hier wurde das Mohito-Framework als eine Bibliothek spzifiziert, die den technischen Code beinhaltet, welcher die Verwendung der zugrunde liegenden spezifischen Plattform Komponenten, unabhängig von den anwendungsspezifischen fachlichen Datentypen, ermöglicht. Da als Ziel von Mohito eine einheitliche und homogene Zugriffstruktur auf den Datenhaltungs-Stack auf allen unterstützen Zielplattformen verfolgt wird, bedeutet dies, dass das Mohito-Framework in seiner jeweiligen plattformspezifischen Ausprägung genau die Funktionalitäten zur Verfügung stellten muss, die nicht durch eine jeweilige plattformspezifische Komponente angeboten werden kann. Das kann beispielsweise auf mobilen Plattformen wie Android bedeuten, dass das Mohito-Framework Mechanismen und Funktionalitäten zur Verfügung stellen muss, die auf Desktop-Plattformen schon durch existierende Komponenten abgedeckt werden.

Im Folgenden sollen einzelne Architekturprinzipien und Entwurfsalternativen, die bei der Entscheidung zur gewählten Architektur zur Verfügung standen, erklärt und gegenübergestellt werden, um die Entscheidungen zu der gewählten Architektur besser verstehen und nachvollziehen zu können.

2.1 ORM vs. SQL

Ein Problem mit welchem wir uns bei dem Entwurf des Zugriffs auf den Datenhaltungs-Stacks konfrontiert sahen, war die schon seit Anfang der 1980er Jahre als „Impedance Mismatch“ (Copeland & Maier, 84) bezeichnete objektrelationale Unverträglichkeit. Diese Unverträglichkeit beschreibt das Problem, welches durch die Verwendung von objektorientierten Programmiersprachen in Verbindung mit Daten entsteht, die in relationalen Datenbanken gespeichert werden. Objektorientierte Anwendungen stellen ihre Daten in Form von Objekten dar. Diese sind typischerweise durch vier Eigenschaften gekennzeichnet: Identität, Zustand, Verhalten und Kapselung. Relationale Datenbanksysteme hingegen bedienen sich bei Datenzugriff und Datenspeicherung der Prädikatenlogik bzw. des Relationenkalkül, eine der theoretischen Grundlagen von Datenbankabfragesprachen wie etwa SQL. Beide Welten sind demnach sehr unterschiedlich und man wird immer irgendeine Form von Zuordnungen von Daten zu Objekten

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

benötigen, wenn man von objektorientierten Programmiersprachen auf Daten in relationalen Datenbanken zugreifen möchte. (Neward, 2006)

Zu diesem Problem gibt es mannigfaltige Lösungen. Im Kern lassen sich hier drei Gruppen von Lösungsvarianten unterscheiden, die unterschiedliche Kompromisse eingehen und alle Ihre individuellen Vor- und Nachteile haben. Der erste Ansatz wäre eine manuelle Implementierung der Datenbankzugriffslogik unter Verwendung relationaler Werkzeuge wie JBDC unter Java oder ADO.NET in .NET Umgebungen. So kann auf relationale Daten zugegriffen werden und es können diese, wo nötig, auf Objekte abgebildet werden.

Ein anderer Ansatz wird als „Table Data Gateway" (Fowler) bezeichnet. Hier agiert ein Objekt als Zugangsschnittstelle zu einer Datenbanktabelle. Eine Instanz behandelt alle Zeilen einer Tabelle. In der entsprechenden Klasse sind damit alle SQL Anfragen an eine Tabelle versammelt. Damit erreicht man eine klare Trennung zwischen SQL und Anwendungslogik. In Kombination mit der Generation von immer wiederkehrenden Codefragmenten kann über dieses Entwicklungsmuster eine einfache Zugriffsschicht zur Datenbank realisiert werden.

Der dritte Ansatz stellt die Verwendung automatisierter objektrelationaler Bibliotheken zur gegenseitigen Abbildung, wie beispielsweise Hibernate oder ORMLight, dar. Diese Bibliotheken übernehmen neben grundlegenden Operationen wie dem Lesen, Schreiben und Suchen von Daten häufig auch Funktionalitäten wie das Caching von gerade angefragten Datensätzen. Darüber hinaus stellen OR-Mapper häufig eine Anbindung an eine Vielzahl unterschiedlicher SQL Server sowie deren spezifischen Daten- und Anfrageformaten zur Verfügung. Wichtig an dieser Stelle ist jedoch, dass sich OR-Mapper auf die Verbindung zwischen objektorientierter Software und relationalen Datenbanken fokussieren. Eine generelle, plattformübergreifende Datenmodellierung inklusive der Kommunikation und technischen Unterstützung auf unterschiedlichen Plattformen, die über SQL Server hinausgehen, ist hierbei nicht vorgesehen.

Hier kommt Mohito ins Spiel. Das Mohito-Framework soll eine Vielzahl von OR-Mappern auf unterschiedlichsten Plattformen unterstützen, um dadurch auf den zugrunde liegenden Daten-Stack zugreifen zu können, ohne selbst entsprechende Anbindungen an unterschiedlichste Datenbanken bereitstellen zu müssen. Auf diesem Weg ist es recht einfach möglich ein breites Spektrum an bestehenden Datenbanken und damit verbunden ein breites Spektrum an Bestandssoftware zu bedienen. Daher haben wir uns im Mohito Projekt für diese Alternative entschieden.

Doch der Ansatz hat auch Nachteile. OR-Mapper sind und können per Entwurf nicht so mächtig sein wie SQL. Das heißt, es wird immer Randbereiche und Problemstellungen geben, die sich durch den Einsatz von OR-Mappern nicht korrekt abdecken lassen. Ferner ist die Art der Abbildung von Beziehungen zwischen Elementen des Datenmodells nicht eindeutig und wird von OR-Mappern unterschiedlich durchgeführt. Gerade OR-Mapper für mobile Plattformen haben häufig Einschränkungen bezüglich der unterstützten Beziehungen, welche im Rahmen des Mohito-Frameworks vor Entwicklern verborgen werden müssen. Darüber hinaus gibt es auch Bestandssoftware, die auf Mohito migriert werden soll, aber bei ihrem aktuellen Datenzugriff keinen OR-Mapper nutzen.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

2.2 Warum wir keine Threads im Framework verwenden

Wir haben uns im Mohito Projekt von Anfang an Gedanken darüber gemacht, ob wir uns für eine rein synchrone Lösung entscheiden, oder ob das Mohito-Framework alle Zugriffe auf die Datenbank asynchron realisiert. Die erste Intuition war die zeitbeanspruchenden Datenbankzugriffe im Framework asynchron auszuführen und die Multithreading Komplexität vor dem Programmierer zu verbergeb. Es wurde schon sehr schnell klar, dass dies kein guter Ansatz ist. Zum einen hätten wir uns diese Lösung durch komplexere Schnittstellen erkaufen müssen, da man zu jeder asynchron ausgeführten Aktion auch Callback Routinen zur Verfügung stellen müsste. Zum anderen, und dies war auch der entscheidende Grund, ist es gerade auf mobilen Plattformen häufig nötig, die Ausführung der gesamten Anwendung zu pausieren. Dies passiert unter Android beispielweise beim Drehen des Gerätes, was ein pausieren und einen darauf folgenden kompletten Neuaufbau der Anwendung erfordert. In diesem Zusammenhang braucht der Anwendungsentwickler volle Kontrolle über alle laufenden Threads, um diese korrekt beenden und gegebenenfalls neu starten zu können. Verbirgt man allerdings die komplette Threading-Logik vor Ihm, wär das nicht mehr möglich. Man müsste also zusätzlich zu den Callback Routinen noch weitere Schnittstellen schaffen, um die intern ablaufenden Threads von außen kontrollieren zu können. Dies wiederum würde allerdings die Gesamtkomplexität des Frameworks eher erhöhen und mehr Aufwand bereiten, als den Benutzer des Mohito-Frameworks das Auslagern relevanter Aufrufe in Threads selbst zu überlassen. Im Sinn der Kapselung und Trennung von Aspekten wurde deswegen die synchrone Variante gewählt.

2.3 Warum wir kein EMF verwenden

Eclipse bietet mit dem Eclipse Modeling Framework (EMF) ein vollständiges System zum Modellieren und Verwalten komplexer Modelle. Modelle werden mittels EMOF spezifiziert. Die Persistierung kann beispielsweise mittels XMI, einem offenen Austauschformat für Modelle, erfolgen. EMF stellt eine komplette Verarbeitungskette und Laufzeitunterstützung für die Modifikation, Verifikation und Interpretation der Modelle und ihrer Metadaten zur Verfügung. Im Mohito-Projekt nutzen wir teilweise Tools und Technologien der EMF Infrastruktur, wie beispielsweise das Ecore Metamodell und die Ecore Tools zur Modellierung und Validation der Modelle oder Codegenerator Mechanismen durch Xtend, ohne aber dadurch eine direkte Abhängigkeit von generiertem Quellcode für Zielplattformen zu EMF zu schaffen. Dies ist daher wichtig, da wir die Modelle auch auf Plattformen und in Bereichen nutzen wollen, die nicht inhärent mit EMF arbeiten, wie beispielsweise bei der iOS/iPhone Entwicklung. Eine Verwendung von EMF würde außerdem die volle Unterstützung des Standards auf alle Plattformen erfordern. Viele Anteile und Funktionen von EMF sind im Anwendungskontext von MOHITO nicht notwendig und würden den Umgang mit Modelle und die Umsetzung für neue Plattformen erschweren.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

3 Framework-Architektur

Dieses Kapitel beschreibt die Architektur des Frameworks. Die Beschreibung basiert auf dem in (L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform, 2013) beschriebenen technischen Aufbau und beschreibt die Bausteine des Frameworks sowie deren Zusammenspiel.

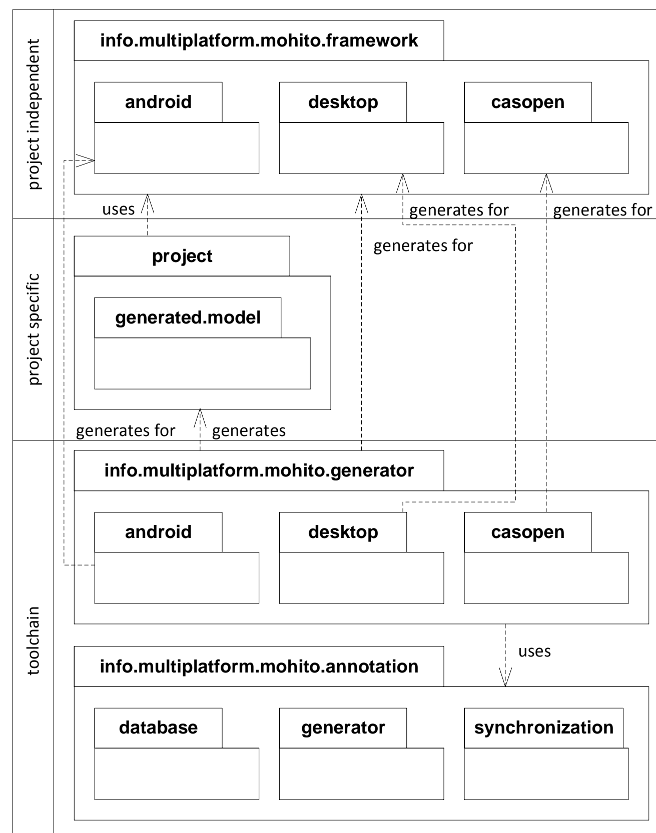

Abbildung 1: Frameworkanteile und deren Beziehung

Das Framework wird in 3 Anteile geteilt: Einen Anteil für die Nutzung von MOHITO-Modellen sowohl allgemein als auch auf Zielplattformen (*project independent*), einen Anteil für ein konkretes Projekt inklusive erzeugter Implementierung für ein bestimmtes Modell (*project specific*) und einen Anteil für die Erzeugung von Implementierungen für ein Modell sowie der Verwaltung von MOHITO-Annotationen (*toolchain*). Diese Anteile sind in Abbildung 1 dargestellt.

Der projektunabhängige Anteil enthält die Implementierung zur Nutzung von MOHITO-Modellen. Dies erlaubt sowohl den einfachen Zugriff, eine in sich geschlossene Dokumentation des Frameworks für Entwickler direkt im Quellcode und erleichtert die Wartung von Anteilen, welche unabhängig von einem konkreten Modell sind. Es findet lediglich eine Trennung in allgemeine und plattformspezifische Anteile statt. Die notwendigen allgemeinen Klassen für Java-basierte Umgebungen liegen im Paket *info.multiplatform.mohito.framework*. Die Implementierung für die einzelnen Plattformen liegt in den ent-

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

sprechenden Unterpaketen, beispielsweise *android*, *desktop* und *casopen*. Technische Details sind in Kapitel 3.3 von (L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform, 2013) beschrieben. Zusätzlich beschreibt Kapitel 4 des vorliegenden Dokuments die API im Detail.

Der projektspezifische Anteil enthält die generativ erzeugte Implementierung für ein konkretes MOHITO-Modell. Dieser liegt typischerweise innerhalb der Anwendung, welche auf die Daten des Modells zugreifen möchte. Im Beispiel ist dieser Anteil durch das Paket *project* dargestellt. Der generierte Anteil liegt im Beispiel im Paket *generated.model*.

Der Werkzeugkettenanteil enthält die Implementierung zur Erstellung und dem Umgang mit MOHITO-Modellen. Diese lässt sich in zwei Anteil gliedern: Den Anteil zur Erzeugung der projektspezifischen Anteile sowie den Anteil zur Erzeugung und Verwaltung von MOHITO-Annotationen. Ersterer liegt im Paket *info.multiplatform.mohito.generator* und enthält für die jeweiligen Zielplattformen angepasste Implementierungen in den Unterpaketen. Im Beispiel sind dies *android*, *desktop* und *casopen*. Der Anteil zum Umgang mit Annotationen liegt im Paket *info.multiplatform.mohito.annotation*. Die Verwendung und technischen Details sind in Kapitel 3.1 von (L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform, 2013) beschrieben. Unterpakete enthalten die notwendigen Definitionen und Erweiterungen für einzelne Kategorien. Im Beispiel sind dies *database*, *generator* und *synchronization*. Die Verwendung und technische Details sind in Kapitel 3.2 von (L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform, 2013) beschrieben.

Insgesamt sind die einzelnen Anteile des Frameworks so lose gekoppelt wie möglich. Änderungen bleiben damit möglichst lokal und die einzelnen Anteile können bis auf die definierten Schnittstellen unabhängig voneinander gepflegt und weiterentwickelt werden. Änderungen bei der Erzeugung erfordern beispielsweise keine Anpassung von Annotationen oder allgemeinem Quellcode zur Modellverwaltung. Jeder Anteil kann für sich geprüft und kompiliert werden, es gibt keine beliebigen Einsprungs- und Erweiterungspunkte. Die Dokumentation für Entwickler kann direkt in der Entwicklungsumgebung erfolgen und erlaubt einfachen Zugriff.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

4 Detaillierte API Beschreibung

Im Folgenden wird der nicht generierte Framework Anteil des Mohito-Projekts detailliert betrachtet. Das Framework besteht im Wesentlichen aus den Mohito Entitäten, welche die zu persistierenden Daten darstellen und entsprechender DAO Objekten, welche die Funktionalitäten zum Zugriff auf Entitäten zur Verfügung stellen. Verwaltet werden die DAO Objekten durch entsprechende DAO Manager, die wiederum über Storage Manager erzeugt und verwaltet werden.

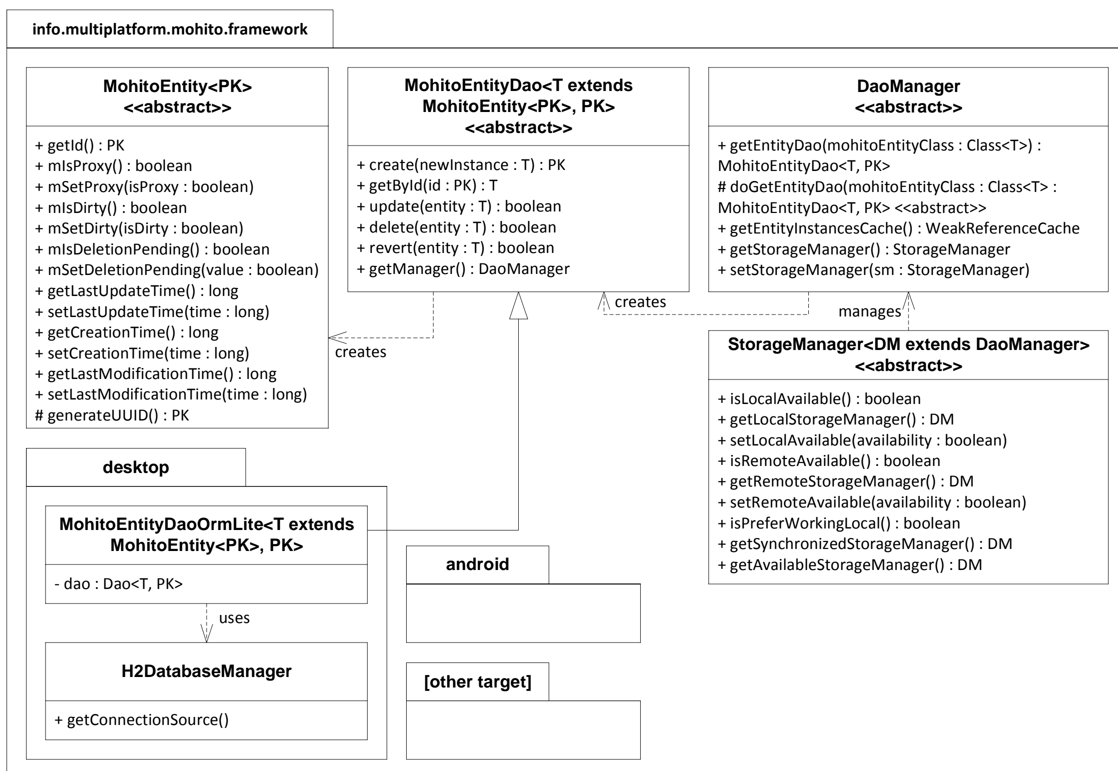

Abbildung 2: Konzept für den projektunabhängigen Zugriff im Framework

Abbildung 2 gibt einen Überblick über das zugrundeliegende Konzept des Frameworks. Auf den nachfolgenden Seiten werden die einzelnen Klassen in Form einer „Javadoc" Dokumentation detailliert beschrieben.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Package info.multiplatform.mohito.framework

Class Summary		Page
DaoManager	Management interface for all types of entities of a model.	12
MohitoEntity<PK>	Interface and convenience functionality for handling MOHITO model entities.	16
MohitoEntityDao<T extends MohitoEntity<PK>,PK>	Data access object for mohito entities.	23
MohitoList<T>	Manages list, which are part of MOHITO-Entities.	26
StorageManager<DM extends DaoManager>	Manages direct access to the available storages.	35
WeakReferenceCache	Cache storing weak references to instances of MOHITO-entities.	40

Exception Summary		Page
NoInformationSourceAvailableException	Notification that there was no information source available for providing mandatory information.	34

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class DaoManager

info.multiplatform.mohito.framework

java.lang.Object

└ **info.multiplatform.mohito.framework.DaoManager**

abstract public class **DaoManager**

extends Object

Management interface for all types of entities of a model.

Field Summary		Page
protected WeakReferenceCache	**mohitoEntityInstancesCache** Cache managing access to instance of all elements connected to this storage.	13
protected StorageManager<DaoManager>	**storageManager** The storage manager using this dao manager.	14

Constructor Summary	Page
DaoManager(boolean useCache) Creates a new instance.	14

Method Summary		Page
protected abstract <T extends MohitoEntity<PK>,PK> MohitoEntityDao<T,PK>	**doGetEntityDao**(Class<T> mohitoEntityClass) Actually returns the DAO for the given class.	14
<T extends MohitoEntity<PK>,PK> MohitoEntityDao<T,PK>	**getEntityDao**(Class<T> mohitoEntityClass) Request the entity manager responsible for a an entity.	14
WeakReferenceCache	**getEntityInstancesCache**()	15
StorageManager<DaoManager>	**getStorageManager**()	15
void	**setStorageManager**(StorageManager<DaoManager> storageManager)	15

Field Detail

mohitoEntityInstancesCache

protected final WeakReferenceCache **mohitoEntityInstancesCache**

Cache managing access to instance of all elements connected to this storage.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

storageManager

protected StorageManager<DaoManager> **storageManager**

The storage manager using this dao manager.

Constructor Detail

DaoManager

public **DaoManager**(boolean useCache)

Creates a new instance.

Parameters:

useCache - Determines if all data objects returned by any managed DAO are cached. If caching is enabled, references to existing objects are returned instead of the generation of new instances.

Method Detail

getEntityDao

pub-
lic <T extends MohitoEntity<PK>,PK> MohitoEntityDao<T,PK> **getEntityDao**(Class<T> mohitoEntityCl
ass)

Request the entity manager responsible for a an entity.

Parameters:

mohitoEntityClass - The entity class.

Returns:

The entity manager responsible for the class.

---

doGetEntityDao

protect-
ed abstract <T extends MohitoEntity<PK>,PK> MohitoEntityDao<T,PK> **doGetEntityDao**(Class<T> mo
hitoEntityClass)

Actually returns the DAO for the given class.

Parameters:

mohitoEntityClass - The class.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Returns:

The DAO responsible for the class.

---

getEntityInstancesCache

public `WeakReferenceCache` **getEntityInstancesCache**()

Returns:

The cache for instances created by any DAO managed by this manager. Returns `null` if no cache should be used.

---

getStorageManager

public `StorageManager`<`DaoManager`> **getStorageManager**()

Returns:

the storageManager

---

setStorageManager

public void **setStorageManager**(`StorageManager`<`DaoManager`> storageManager)

Parameters:

storageManager - the storageManager to set

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class MohitoEntity<PK>

info.multiplatform.mohito.framework

java.lang.Object
└ **info.multiplatform.mohito.framework.MohitoEntity<PK>**

abstract public class **MohitoEntity<PK>**

extends Object

Interface and convenience functionality for handling MOHITO model entities.

Field Summary		Page
protected Long	`creationTime` Entity creation date.	18
protected Long	`lastModificationTime` Entity was last modified on this date.	18
protected Long	`lastUpdateTime` Entity was last updated or synchronized on this date.	18
static Logger	`logger` Logger for this class.	17
protected boolean	`mDeletionPending` Flag if this entity is pending deletion upon the next update.	18
protected boolean	`mIsDirty` Flag if this entity has been modified since the last update.	17
protected boolean	`mIsProxy` Flag if this entity is a proxy and does not contain the real values.	17

Constructor Summary	Page
`MohitoEntity`() Creates a new instance and assigns the administrative flags and general information.	18

Method Summary		Page
protected void	`checkProxyAndResolve`() Checks if this MOHITO-Entity is a proxy and loads the data if necessary.	19
protected abstract void	`doCheckProxyAndResolveAssignment`(`MohitoEntity`<PK> reference) Actual assignment of data for each subclass for `checkProxyAndResolve()`.	19
protected abstract `MohitoEntity`<PK>	`doCheckProxyAndResolveGetReferenceEntity`() Retrieves the DAO for the actual handled entity for `checkProxyAndResolve()`.	19

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

abstract boolean	**domainContentEquals**(MohitoEntity<PK> reference) Compares this entity with a given reference for equality.	*19*
PK	**generateUUID**() Generator for UUIDs.	*18*
Long	**getCreationTime**()	*20*
abstract PK	**getId**()	*19*
Long	**getLastModificationTime**()	*21*
Long	**getLastUpdateTime**()	*21*
boolean	**mIsDeletionPending**()	*21*
boolean	**mIsDirty**()	*20*
boolean	**mIsProxy**()	*20*
void	**mSetDeletionPending**(boolean deletionPending)	*22*
void	**mSetDirty**(boolean mIsDirty) Marks if this entity has been modified by a user since it was create or loaded.	*20*
void	**mSetProxy**(boolean mIsProxy) Marks if this entity is only a proxy for the object and does not contain the correct values (yet).	*20*
void	**setCreationTime**(Long creationTime)	*20*
void	**setLastModificationTime**(Long lastModificationTime)	*21*
void	**setLastUpdateTime**(Long lastUpdateTime)	*21*

Field Detail

logger

public static final Logger **logger**

Logger for this class.

---

mIsProxy

protected boolean **mIsProxy**

Flag if this entity is a proxy and does not contain the real values. Transient value, which does not need to be persisted.

---

mIsDirty

protected boolean **mIsDirty**

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Flag if this entity has been modified since the last update.

---

mDeletionPending

protected boolean **mDeletionPending**

Flag if this entity is pending deletion upon the next update.

---

lastUpdateTime

protected Long **lastUpdateTime**

Entity was last updated or synchronized on this date. Time zone is UTC, reference point in time January 1, 1970, resolution milliseconds. The time must be set by the server before starting the data transmission.

---

creationTime

protected Long **creationTime**

Entity creation date. Time zone is UTC, reference point in time January 1, 1970.

---

lastModificationTime

protected Long **lastModificationTime**

Entity was last modified on this date. Time zone is UTC, reference point in time January 1, 1970, resolution milliseconds.

Constructor Detail

MohitoEntity

public **MohitoEntity**()

Creates a new instance and assigns the administrative flags and general information.

Method Detail

generateUUID

public PK **generateUUID**()

Generator for UUIDs. Must be overwritten if customized generation strategies are used for MOHITO-Entities.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Returns:

UUID.

---

checkProxyAndResolve

protected void **checkProxyAndResolve**()

Checks if this MOHITO-Entity is a proxy and loads the data if necessary.

---

doCheckProxyAndResolveGetReferenceEntity

protected abstract `MohitoEntity`<PK> **doCheckProxyAndResolveGetReferenceEntity**()

Retrieves the DAO for the actual handled entity for `checkProxyAndResolve()`.

---

doCheckProxyAndResolveAssignment

protected abstract void **doCheckProxyAndResolveAssignment**(`MohitoEntity`<PK> reference)

Actual assignment of data for each subclass for `checkProxyAndResolve()`.

---

domainContentEquals

public abstract boolean **domainContentEquals**(`MohitoEntity`<PK> reference)

Compares this entity with a given reference for equality.

Parameters:

reference - Reference entity.

Returns:

`true` if the provided entity is assignment compatible and the domain content is equal.

---

getId

public abstract PK **getId**()

Returns:

the identifier of this entity.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

mIsProxy

public boolean **mIsProxy**()

Returns:

If this entity is a proxy.

---

mSetProxy

public void **mSetProxy**(boolean mIsProxy)

Marks if this entity is only a proxy for the object and does not contain the correct values (yet).

Parameters:

mIsProxy - `true` if it was modified, `false` otherwise.

---

mIsDirty

public boolean **mIsDirty**()

Returns:

If this entity is a proxy.

---

mSetDirty

public void **mSetDirty**(boolean mIsDirty)

Marks if this entity has been modified by a user since it was create or loaded.

Parameters:

mIsDirty - `true` if it was modified, `false` otherwise.

---

getCreationTime

public final Long **getCreationTime**()

Returns:

the creationTime

---

setCreationTime

public final void **setCreationTime**(Long creationTime)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Parameters:

creationTime - the creationTime to set

---

getLastModificationTime

public final Long **getLastModificationTime**()

Returns:

the lastModificationTime

---

setLastModificationTime

public final void **setLastModificationTime**(Long lastModificationTime)

Parameters:

lastModificationTime - the lastModificationTime to set

---

getLastUpdateTime

public final Long **getLastUpdateTime**()

Returns:

the lastUpdateTime

---

setLastUpdateTime

public final void **setLastUpdateTime**(Long lastUpdateTime)

Parameters:

lastUpdateTime - the lastUpdateTime to set

---

mIsDeletionPending

public final boolean **mIsDeletionPending**()

Returns:

the mDeletionPending

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

mSetDeletionPending

public final void **mSetDeletionPending**(boolean deletionPending)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class MohitoEntityDao<T extends MohitoEntity<PK>,PK>

info.multiplatform.mohito.framework

java.lang.Object

└ **info.multiplatform.mohito.framework.MohitoEntityDao<T,PK>**

Type Parameters:

T - Type of MOHITO business entity accessed by this DAO.

PK - Type of primary key.

---

abstract public class **MohitoEntityDao<T extends** MohitoEntity**<PK>,PK>**

extends Object

Data access object for mohito entities.

Constructor Summary	Page
MohitoEntityDao(DaoManager manager) Default constructor setting the MOHITO-storage manager.	23

Method Summary		Page
abstract PK	create(T newInstance) Stores the provided entity.	24
abstract boolean	delete(T entity) Delete an entity specified by the supplied entity.	24
abstract T	getById(PK id) Retrieve an entity by its identifier.	24
DaoManager	getManager()	25
abstract boolean	revert(T entity) Reverts the values of the provided entity to the available values in the local or remote storage.	25
abstract boolean	update(T entity) Persists the entity.	24

Constructor Detail

MohitoEntityDao

public **MohitoEntityDao**(DaoManager manager)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Default constructor setting the MOHITO-storage manager.

Method Detail

create

public abstract `PK` **create**(`T` newInstance)

Stores the provided entity.

Returns:

The automatically generated identifier.

---

getById

public abstract `T` **getById**(`PK` id)

Retrieve an entity by its identifier. If the same entity is requested multiple times, the same object is returned. If an entity is modified between different calls to this method then the most recent version, which may include modifications, is returned. Use `revert(MohitoEntity)` on an existing entity to restore the contained information according to the available storage(s).

Parameters:

id - The internal identifier.

Returns:

The entity or `null`.

---

update

public abstract boolean **update**(`T` entity)

Persists the entity. The meta-data of the entity is updated accordingly.

Parameters:

entity - The entity to update.

Returns:

`true` if the update succeeded, `false` otherwise.

---

delete

public abstract boolean **delete**(`T` entity)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Delete an entity specified by the supplied entity.

Parameters:

entity - The entity to delete.

Returns:

`true` if the deletion was successful, `false` otherwise.

---

revert

public abstract boolean **revert**(`T` entity)

Reverts the values of the provided entity to the available values in the local or remote storage.

Parameters:

entity - The entity to revert.

Returns:

`true` if the reversion was successful, `false` otherwise.

---

getManager

public `DaoManager` **getManager**()

Returns:

the manager for this DAO.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class MohitoList<T>

info.multiplatform.mohito.framework

java.lang.Object
└ **info.multiplatform.mohito.framework.MohitoList<T>**

Type Parameters:

T - Type of elements contained in the list.

All Implemented Interfaces:

Collection<T>, Iterable<T>, List<T>

---

public class **MohitoList<T>**

extends Object

implements List<T>

Manages list, which are part of MOHITO-Entities. Takes care of assigning containment relations for added/removed objects.

Field Summary		Page
static Logger	`logger` Logger for this class.	27

Constructor Summary	Page
`MohitoList`(Class<`T`> managedType, `MohitoEntity`<?> parent, String javaFieldNameForInverseRelation) Creates a new list contained in a MOHITO-Entity.	27

Method Summary		Page
void	`add`(int index, `T` element)	28
boolean	`add`(`T` e)	28
boolean	`addAll`(int index, Collection<? extends `T`> c)	28
boolean	`addAll`(Collection<? extends `T`> c)	28
void	`clear`()	29
boolean	`contains`(Object o)	29
boolean	`containsAll`(Collection<?> c)	29
boolean	`equals`(Object o)	29

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

T	get(int index)	29
int	hashCode()	30
int	indexOf(Object o)	30
boolean	isEmpty()	30
Iterator<T>	iterator()	30
int	lastIndexOf(Object o)	31
ListIterator<T>	listIterator()	31
ListIterator<T>	listIterator(int index)	31
T	remove(int index)	31
boolean	remove(Object o)	31
boolean	removeAll(Collection<?> c)	31
boolean	retainAll(Collection<?> c)	32
T	set(int index, T element)	32
int	size()	32
List<T>	subList(int fromIndex, int toIndex)	32
Object[]	toArray()	32
<T> T[]	toArray(T[] a)	33

Field Detail

logger

public static final Logger **logger**

Logger for this class.

Constructor Detail

MohitoList

public **MohitoList**(Class<T> managedType,
MohitoEntity<?> parent,
String javaFieldNameForInverseRelation)

Creates a new list contained in a MOHITO-Entity.

Parameters:

managedType - Type of list elements.

parent - The instance of the MOHITO-Entity containing the list.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

javaFieldNameForInverseRelation - Name of the java field used to set the inverse relation. Must be `null` for 1-way references. Must be set for 1-way containment relations between two MOHITO-Entities. Should be `null` for 1-way containment relations between 1 MOHITO-Entity and another entity, e.g. java.lang.String. Inverse relations allow to access the parent in a containment relation between two MOHITO-Entities.

Method Detail

add

public void **add**(int index,
    T element)

Specified by:

add in interface List<E>

---

add

public boolean **add**(T e)

Specified by:

add in interface Collection<E>

add in interface List<E>

---

addAll

public boolean **addAll**(Collection<? extends T> c)

Specified by:

addAll in interface Collection<E>

addAll in interface List<E>

---

addAll

public boolean **addAll**(int index,
    Collection<? extends T> c)

Specified by:

addAll in interface List<E>

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

clear

public void **clear**()

Specified by:

clear in interface Collection<E>

clear in interface List<E>

---

contains

public boolean **contains**(Object o)

Specified by:

contains in interface Collection<E>

contains in interface List<E>

---

containsAll

public boolean **containsAll**(Collection<?> c)

Specified by:

containsAll in interface Collection<E>

containsAll in interface List<E>

---

equals

public boolean **equals**(Object o)

Specified by:

equals in interface Collection<E>

equals in interface List<E>

Overrides:

equals in class Object

---

get

public T **get**(int index)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Specified by:

get in interface List<E>

---

hashCode

public int **hashCode**()

Specified by:

hashCode in interface Collection<E>

hashCode in interface List<E>

Overrides:

hashCode in class Object

---

indexOf

public int **indexOf**(Object o)

Specified by:

indexOf in interface List<E>

---

isEmpty

public boolean **isEmpty**()

Specified by:

isEmpty in interface Collection<E>

isEmpty in interface List<E>

---

iterator

public Iterator<T> **iterator**()

Specified by:

iterator in interface Iterable<T>

iterator in interface Collection<E>

iterator in interface List<E>

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

lastIndexOf

public int **lastIndexOf**(Object o)

Specified by:

lastIndexOf in interface List<E>

---

listIterator

public ListIterator<T> **listIterator**()

Specified by:

listIterator in interface List<E>

---

listIterator

public ListIterator<T> **listIterator**(int index)

Specified by:

listIterator in interface List<E>

---

remove

public T **remove**(int index)

Specified by:

remove in interface List<E>

---

remove

public boolean **remove**(Object o)

Specified by:

remove in interface Collection<E>

remove in interface List<E>

---

removeAll

public boolean **removeAll**(Collection<?> c)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Specified by:

removeAll in interface Collection<E>

removeAll in interface List<E>

---

retainAll

public boolean **retainAll**(Collection<?> c)

Specified by:

retainAll in interface Collection<E>

retainAll in interface List<E>

---

set

public T **set**(int index,
T element)

Specified by:

set in interface List<E>

---

size

public int **size**()

Specified by:

size in interface Collection<E>

size in interface List<E>

---

subList

public List<T> **subList**(int fromIndex,
int toIndex)

Specified by:

subList in interface List<E>

---

toArray

public Object[] **toArray**()

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Specified by:

toArray in interface Collection<E>

toArray in interface List<E>

---

toArray

public <T> T[] **toArray**(T[] a)

Specified by:

toArray in interface Collection<E>

toArray in interface List<E>

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class NoInformationSourceAvailableException

info.multiplatform.mohito.framework

java.lang.Object
 └ java.lang.Throwable
  └ java.lang.Exception
   └ java.lang.RuntimeException
    └ **info.multiplatform.mohito.framework.NoInformationSourceAvailableException**

All Implemented Interfaces:

Serializable

---

public class **NoInformationSourceAvailableException**

extends RuntimeException

Notification that there was no information source available for providing mandatory information.

Constructor Summary	Page
`NoInformationSourceAvailableException`(String message) Creates a new exception.	34

Constructor Detail

NoInformationSourceAvailableException

public **NoInformationSourceAvailableException**(String message)

Creates a new exception.

Parameters:

message - Message with information on the error.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class StorageManager<DM extends DaoManager>

info.multiplatform.mohito.framework

java.lang.Object

└ **info.multiplatform.mohito.framework.StorageManager<DM>**

abstract public class **StorageManager<DM extends** DaoManager**>**

extends Object

Manages direct access to the available storages.

Field Summary		Page
protected boolean	autoSyncOnRemoteAvailable Flag if synchronization should be started automatically if the remote storage becomes available.	37
protected boolean	handleConflictsAutomatically Flag if synchronization conflicts should be handled automatically or manually.	37
protected boolean	localAvailable Flag if the local instance is available and should be used for lookups.	37
protected DM	localInstance Manager for access to local storage.	36
protected boolean	preferWorkingLocal Flag if working on the local instance is preferred and synchronization is invoked explicitly.	37
protected boolean	remoteAvailable Flag if the remote instance is available and should be used for lookups.	37
protected DM	remoteInstance Manager for access to remote storage.	36
protected boolean	serverWinsInAutomatedConflictResolution Flag if the server-side always wins in automated synchronization conflict resolution.	37
protected DM	synchronizedInstance Manage for synchronized storage access.	36
protected info.multiplatform.mohito.framework.synchronization.MohitoEntitySynchronizer	synchronizer Synchronization mechanism for the managed classes.	37

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Constructor Summary	Page
StorageManager(DM localInstance, DM remoteInstance, DM synchronizedInstance, boolean preferWorkingLocal, boolean handleConflictsAutomatically, boolean serverWinsInAutomatedConflictResolution, boolean autoSyncOnRemoteAvailable, Class<? extends MohitoEntity<?>>... managedClasses) Create a new instance.	*38*

Method Summary		Page
DM	**getAvailableStorageManager**()	*38*
DM	**getLocalStorageManager**()	*39*
DM	**getRemoteStorageManager**()	*39*
DM	**getSynchronizedStorageManager**()	*39*
boolean	**isLocalAvailable**()	*38*
boolean	**isPreferWorkingLocal**()	*39*
boolean	**isRemoteAvailable**()	*38*
void	**setLocalAvailable**(boolean isLocalAvailable)	*39*
void	**setRemoteAvailable**(boolean isRemoteAvailable)	*38*

Field Detail

localInstance

protected final DM **localInstance**

Manager for access to local storage.

---

remoteInstance

protected final DM **remoteInstance**

Manager for access to remote storage.

---

synchronizedInstance

protected final DM **synchronizedInstance**

Manage for synchronized storage access.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

localAvailable

protected boolean **localAvailable**

Flag if the local instance is available and should be used for lookups.

---

remoteAvailable

protected boolean **remoteAvailable**

Flag if the remote instance is available and should be used for lookups.

---

preferWorkingLocal

protected final boolean **preferWorkingLocal**

Flag if working on the local instance is preferred and synchronization is invoked explicitly.

---

handleConflictsAutomatically

protected final boolean **handleConflictsAutomatically**

Flag if synchronization conflicts should be handled automatically or manually.

---

serverWinsInAutomatedConflictResolution

protected final boolean **serverWinsInAutomatedConflictResolution**

Flag if the server-side always wins in automated synchronization conflict resolution.

---

autoSyncOnRemoteAvailable

protected final boolean **autoSyncOnRemoteAvailable**

Flag if synchronization should be started automatically if the remote storage becomes available.

---

synchronizer

protected final info.multiplatform.mohito.framework.synchronization.MohitoEntitySynchronizer **synchronizer**

Synchronization mechanism for the managed classes.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Constructor Detail

StorageManager

public **StorageManager**(DM localInstance,
DM remoteInstance,
DM synchronizedInstance,
boolean preferWorkingLocal,
boolean handleConflictsAutomatically,
boolean serverWinsInAutomatedConflictResolution,
boolean autoSyncOnRemoteAvailable,
Class<? extends `MohitoEntity`<?>>... managedClasses)

Create a new instance.

Parameters:

localInstance - DAO manager for local access.

remoteInstance - DAO manager for remote access.

Method Detail

getAvailableStorageManager

public DM **getAvailableStorageManager**()

Returns:

An available dao manager (either local or remote).

---

isRemoteAvailable

public boolean **isRemoteAvailable**()

---

setRemoteAvailable

public void **setRemoteAvailable**(boolean isRemoteAvailable)

---

isLocalAvailable

public boolean **isLocalAvailable**()

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

setLocalAvailable

public void **setLocalAvailable**(boolean isLocalAvailable)

---

getLocalStorageManager

public DM **getLocalStorageManager**()

---

getRemoteStorageManager

public DM **getRemoteStorageManager**()

---

getSynchronizedStorageManager

public DM **getSynchronizedStorageManager**()

---

isPreferWorkingLocal

public boolean **isPreferWorkingLocal**()

Returns:

the preferWorkingLocal

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class WeakReferenceCache

info.multiplatform.mohito.framework

java.lang.Object

└ **info.multiplatform.mohito.framework.WeakReferenceCache**

---

public class **WeakReferenceCache**

extends Object

Cache storing weak references to instances of MOHITO-entities. The weak references are required in order to allow garbage collection of instances referenced by this cache. Supports deferred loading of information for objects and references between objects.

See Also:

http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/ref/WeakReference.html

---

Constructor Summary	***Page***
WeakReferenceCache()	*40*

Method Summary		***Page***
`<T extends MohitoEntity<PK>,PK> T`	**get**(Class<T> clazz, PK id) Retrieves the instance with the given id and MOHITO-Entity class from the cache.	*41*
`<T extends MohitoEntity<PK>,PK> T`	**put**(Class<T> clazz, PK id, T instance) Puts a new instance of the given class and id in the cache.	*41*
`WeakReferenceCache`	**registerClass**(Class<? extends `MohitoEntity`<?>> clazz) Register handling instance of the provided class representing a MOHITO-Entity by this cache.	*41*
`<T extends MohitoEntity<PK>,PK> void`	**remove**(Class<T> clazz, PK id) Removes an instance of the given class from the cache.	*42*
`WeakReferenceCache`	**unregisterClass**(Class<? extends `MohitoEntity`<?>> clazz) Unregisters handling instance of the provided class by this cache.	*41*

Constructor Detail

WeakReferenceCache

public **WeakReferenceCache**()

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Method Detail

registerClass

public synchronized `WeakReferenceCache` **registerClass**(Class<? extends `MohitoEntity`<?>> clazz)

Register handling instance of the provided class representing a MOHITO-Entity by this cache.

Parameters:

clazz - The class.

---

unregisterClass

public synchronized `WeakReferenceCache` **unregisterClass**(Class<? extends `MohitoEntity`<?>> clazz)

Unregisters handling instance of the provided class by this cache.

Parameters:

clazz - The class.

---

get

public <T extends `MohitoEntity`<PK>,PK> T **get**(Class<T> clazz,
PK id)

Retrieves the instance with the given id and MOHITO-Entity class from the cache.

Parameters:

clazz - The class.

id - The id.

Returns:

The object or `null` if no object is cached for the id.

---

put

public <T extends `MohitoEntity`<PK>,PK> T **put**(Class<T> clazz,
PK id,
T instance)

Puts a new instance of the given class and id in the cache.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Parameters:

clazz - The class representing the MOHITO-Entity.

id - The id of the instance.

instance - The instance.

Returns:

The prior cached value or `null`.

---

remove

public <T extends `MohitoEntity`<PK>,PK> void **remove**(Class<T> clazz,
PK id)

Removes an instance of the given class from the cache.

Parameters:

clazz - The class representing the MOHITO-Entity.

id - The id of the instance.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Package info.multiplatform.mohito.framework.synchronization

Class Summary		Page
MohitoEntityDaoSynchronizingImpl<T extends MohitoEntity<PK>,PK>	MOHITO-Entity DAO responsible for synchronization-aware access to local and remote storage.	*12*
MohitoEntitySynchronizer	Synchronization mechanism for the MOHITO-Entities managed by a storage manager.	*18*
SynchronizationConflicts	Informing about synchronization conflicts between a local and remote MOHITO-Entity.	*18*

Exception Summary		Page
SynchronizationRequiredException	Notification that a synchronization is required in order for a successful execution.	*18*

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class MohitoEntityDaoSynchronizingImpl<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

info.multiplatform.mohito.framework.synchronization

java.lang.Object

└ info.multiplatform.mohito.framework.MohitoEntityDao<T,PK>

└ **info.multiplatform.mohito.framework.synchronization.MohitoEntityDaoSynchronizingImpl<T,PK>**

Type Parameters:

T - MOHITO-Entity type.

PK - Type of identifier for the entity.

---

public class **MohitoEntityDaoSynchronizingImpl<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>**

extends info.multiplatform.mohito.framework.MohitoEntityDao<T,PK>

MOHITO-Entity DAO responsible for synchronization-aware access to local and remote storage. Automatically selects the available storage and throws `SynchronizationRequiredException` if the access would lead to inconsistent data and a prior synchronization is needed. The operations throw

info.multiplatform.mohito.framework.NoInformationSourceAvailableException if operations require access to data and no storage is available.

Constructor Summary	Page
`MohitoEntityDaoSynchronizingImpl`(Class<`T`> type, info.multiplatform.mohito.framework.DaoManager daoManager) Creates a new instance.	14

Method Summary		Page
`PK`	`create`(`T` newInstance) Stores the provided entity.	14
`boolean`	`delete`(`T` entity) Delete an entity specified by the supplied entity.	15
`T`	`getById`(`PK` id) Retrieve an entity by its identifier.	14

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

boolean	revert(T entity) Reverts the values of the provided entity to the available values in the local or remote storage.	*15*
boolean	update(T entity) Persists the entity.	*15*

Methods inherited from class info.multiplatform.mohito.framework.MohitoEntityDao
getManager

Constructor Detail

MohitoEntityDaoSynchronizingImpl

public **MohitoEntityDaoSynchronizingImpl**(Class<T> type,
info.multiplatform.mohito.framework.DaoManager daoManager)

Creates a new instance.

Parameters:

type - The type managed by this DAO.

daoManager - The manager of this DAO.

Method Detail

create

public PK **create**(T newInstance)

Stores the provided entity.

Overrides:

create in class info.multiplatform.mohito.framework.MohitoEntityDao<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

Returns:

The automatically generated identifier.

---

getById

public T **getById**(PK id)

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Retrieve an entity by its identifier. If the same entity is requested multiple times, the same object is returned. If an entity is modified between different calls to this method then the most recent version, which may include modifications, is returned.

Use “info.multiplatform.mohito.framework.MohitoEntityDao.revert(MohitoEntity) on an existing entity to restore the contained information according to the available storage(s).

Overrides:

getById in class info.multiplatform.mohito.framework.MohitoEntityDao<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

Parameters:

id - The internal identifier.

Returns:

The entity or `null`.

---

update

public boolean **update**(T entity)

Persists the entity. The meta-data of the entity is updated accordingly.

Overrides:

update in class info.multiplatform.mohito.framework.MohitoEntityDao<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

Parameters:

entity - The entity to update.

Returns:

`true` if the update succeeded, `false` otherwise.

---

delete

public boolean **delete**(T entity)

Delete an entity specified by the supplied entity.

Overrides:

delete in class info.multiplatform.mohito.framework.MohitoEntityDao<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Parameters:

entity - The entity to delete.

Returns:

`true` if the deletion was successful, `false` otherwise.

---

revert

public boolean **revert**(T entity)

Reverts the values of the provided entity to the available values in the local or remote storage.

Overrides:

revert in class info.multiplatform.mohito.framework.MohitoEntityDao<T extends info.multiplatform.mohito.framework.MohitoEntity<PK>,PK>

Parameters:

entity - The entity to revert.

Returns:

`true` if the reversion was successful, `false` otherwise.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class MohitoEntitySynchronizer

info.multiplatform.mohito.framework.synchronization

java.lang.Object

└ **info.multiplatform.mohito.framework.synchronization.MohitoEntitySynchronizer**

---

public class **MohitoEntitySynchronizer**

extends Object

Synchronization mechanism for the MOHITO-Entities managed by a storage manager.

Field Summary		*Page*
protected info.multiplatform.mohito.framework.StorageManager<?>	**storageManager** The storage manager used to access the data stores.	*17*
protected Class<? extends info.multiplatform.mohito.framework.MohitoEntity<?>>[]	**synchronizedEntityTypes** Synchronized entity types.	*17*

Constructor Summary	*Page*
MohitoEntitySynchronizer(info.multiplatform.mohito.framework.StorageManager<?> storageManager, Class<? extends info.multiplatform.mohito.framework.MohitoEntity<?>>[] managedClasses) Creates a new instance.	*17*

Method Summary		*Page*
Collection<SynchronizationConflicts>	**synchronize**() Starts the synchronization for all entities.	*18*

Field Detail

storageManager

protected final info.multiplatform.mohito.framework.StorageManager<?> **storageManager**

The storage manager used to access the data stores.

---

synchronizedEntityTypes

protected final Class<? extends info.multiplatform.mohito.framework.MohitoEntity<?>>[] **synchronizedEntityTypes**

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Synchronized entity types.

Constructor Detail

MohitoEntitySynchronizer

pub-
lic **MohitoEntitySynchronizer**(info.multiplatform.mohito.framework.StorageManager<?> storageManag
er,

Class<? extends info.multiplatform.mohito.framework.MohitoEntity<?>>[] managedClasses)

Creates a new instance.

Parameters:

storageManager - Manager used to access local and remote data stores.

managedClasses - The entity types to synchronize.

Method Detail

synchronize

public Collection<`SynchronizationConflicts`> **synchronize**()

Starts the synchronization for all entities.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class SynchronizationConflicts

info.multiplatform.mohito.framework.synchronization

java.lang.Object

└ **info.multiplatform.mohito.framework.synchronization.SynchronizationConflicts**

---

public class **SynchronizationConflicts**

extends Object

Informing about synchronization conflicts between a local and remote MOHITO-Entity.

Constructor Summary	*Page*
`SynchronizationConflicts`()	*18*

Constructor Detail

SynchronizationConflicts

public **SynchronizationConflicts**()

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Class SynchronizationRequiredException

info.multiplatform.mohito.framework.synchronization

java.lang.Object

└ java.lang.Throwable

└ java.lang.Exception

└ java.lang.RuntimeException

ino.multiplatform.mohito.framework.synchronization.SynchronizationRequiredException

All Implemented Interfaces:

Serializable

---

public class **SynchronizationRequiredException**

extends RuntimeException

Notification that a synchronization is required in order for a successful execution. The data would otherwise be in an inconsistent state.

Constructor Summary	*Page*
`SynchronizationRequiredException`()	*18*

Constructor Detail

SynchronizationRequiredException

public **SynchronizationRequiredException**()

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

5 Umgang am Beispiel

Dieses Kapitel beschreibt anhand eines Beispiels wie auf Daten in MOHITO-Modellen zugegriffen werden kann. Es wird beschrieben, wie die Daten für ein bestimmtes Objekt auf einem Client vom Server geholt werden können und wie anschließend über die Beziehungen im Modell navigiert werden kann.

Die Beschreibung der im Detail verwendeten API und der verfügbaren Funktionen befindet sich in Kapitel 4, inklusive einer groben Übersicht in Abbildung 2. Eine grobe Übersicht und Beschreibung der Beziehungen der unterschiedlichen architekturellen Elemente befindet sich ferner in Kapitel 3.3 von (L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform, 2013).

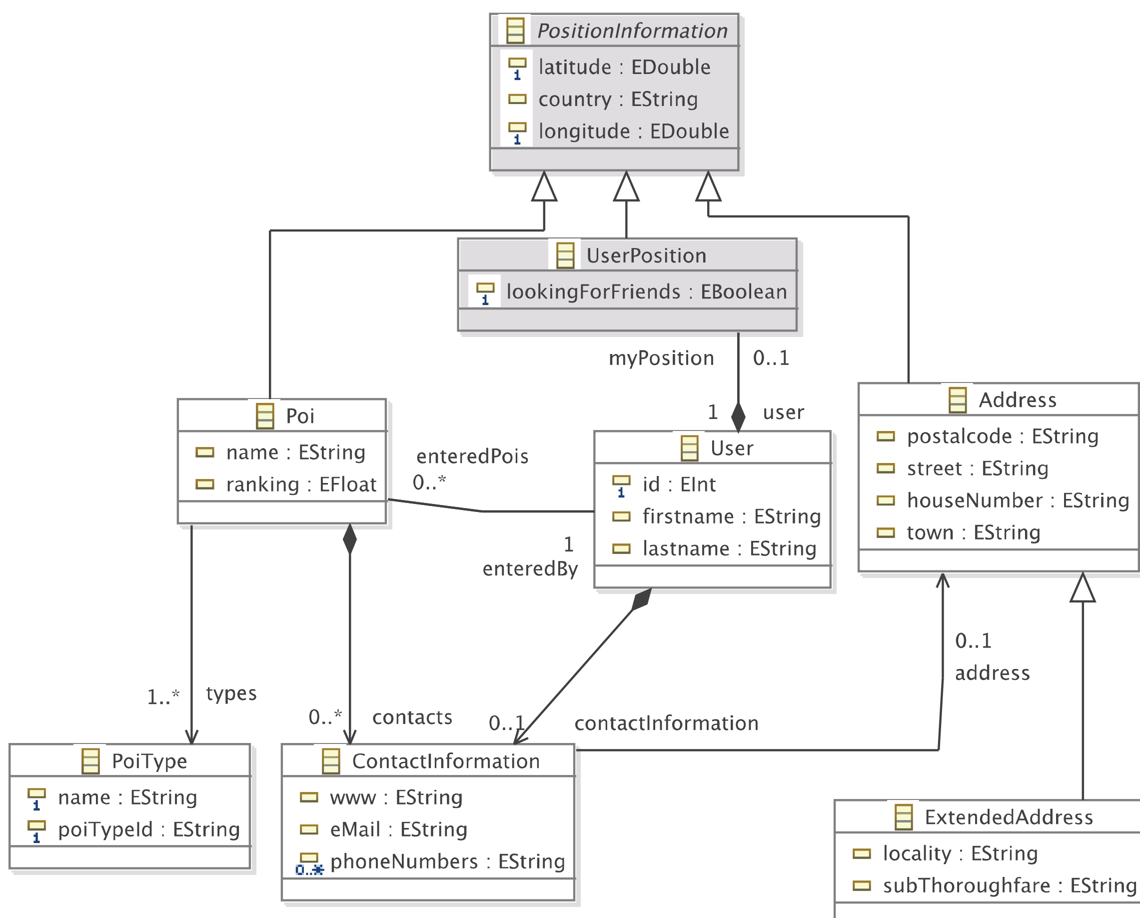

Abbildung 3: HereIAmModel zur Beschreibung der Position interessanter Orte und Aufenthaltsposition von Freunden

Abbildung 3 zeigt das für die Veranschaulichung von Zugriff und Nutzung verwendete MOHITO-Modell. Im Folgenden wird beschrieben wie auf die Namen der Typen des interessanten Ortes mit dem technischen Bezeichner *25* zugegriffen werden kann.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

Zuerst muss ein DAO für den Zugriff auf Daten für Elemente des Typs *Poi* vom *StorageManager* geholt werden. Das Framework soll sich selbstständig um den Zugriff auf die Daten kümmern, unabhängig davon, ob sie bereits lokal zwischengespeichert sind oder nur auf dem Server vorliegen. Aus diesem Grund wird eine Anfrage über den *SynchronizedStorageManager* gestellt. Dies erfolgt mittels *MohitoEntityDao<Poi, String> poiDao = HereIAmModelStorageManager.mINSTANCE.getSynchronizedStorageManager().getPoiDao();*.

Der interessante Ort mit dem Bezeichner 25 lässt sich über das DAO wie folgt beziehen: *Poi poi = poiDao.getById("25")*;

Die Navigation über die Beziehung *types* zu den zugehörigen Instanzen von *PoiType* kann direkt am Objekt erfolgen: *MohitoList<PoiType> types = poi.getTypes();*

Die einzelnen Namen können direkt über die in der List enthaltenen Objekte abgefragt werden. Sollte ein Objekt dabei noch nicht zwischengespeichert vorliegen, wird dies automatisch vom Server abgefragt ohne dass ein Entwickler dies explizit angeben müsste. Ein Beispiel für die Abfrage ist wie folgt: *for (PoiType type : types) { type.getName(); }*.

Möchte ein Entwickler wissen, ob die Daten für einen Typen bereits vorliegen, kann er dies innerhalb obiger Schleife mittels *type.mIsProxy();* im Vorfeld prüfen.

Soll der Name des interessanten Ortes selbst geändert und anschließend permanent gespeichert werden, lässt sich dies mit *poi.setName("Neuer Name"); poiDao.update(poi);* umsetzen.

Soll ein weiterer *PoiType* eingeführt und mit dem interessanten Ort verbunden werden, lassen sich diese beiden Schritte mittels *MohitoEntityDao<PoiType, String> poiTypeDao = HereIAmModelStorageManager.mINSTANCE.getSynchronizedStorageManager().getPoiTypeDao(); PoiType poiType = new PoiType(); poiType.setName("Neuer Typ"); poiTypeDao.create(poiType);* sowie anschließendem *poi.getTypes().add(poiType); poiDao.update(poi);* umsetzen.

Zusammenfassend lässt sich sagen, dass Entwickler unter Kenntnis des Modells selber sehr leicht und direkt zwischen den einzelnen MOHITO-Entitäten, welche eine Instanz der Elemente des Modells darstellen, navigieren können. Die tatsächliche Speicherung wird verborgen und die gewünschte Information kann direkt beschafft werden ohne aufwändige technische Zwischenschritte gehen zu müssen.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

6 Zusammenfassung

Abschließend lässt sich feststellen, dass das im Dokument 6.1.2 (Mohito, Frameworkentwicklung, Multi-Plattform, 2013) beschriebene Framework noch an die Bedürfnisse des Generators angepasst und um geeignete Konzepte erweitert werden musste. Der erste Ansatz des Mohito Frameworks war in seinem grundlegenden Entwurf im Hinblick auf die Bedürfnisse der beiden Referenzimplementierungen entstanden. Bei der Analyse der Referenzimplementierungen im Rahmen der Generatorableitung wurden jedoch Anpassungen notwendig, die zu dem jetzigen Mohito Framework geführt haben. Die vorgenommenen Änderungen umfassen vor allem die generatorfreundliche Aufspaltung der CRUD Operationen auf die entsprechenden Klassen *MohitoEntitiy* und *MohitoEntitiyDAO* und deren einheitliche Erzeugung und Zugriff über die entsprechenden *StorageManager*. Dies war vorher weniger generatorfreundlich über die *IDataAccess* und *IDataAccessControl* Schnittstellen gelöst. Das in diesem Dokument beschriebene Framework kann in seinem grundlegenden Entwurf als final angesehen werden, allerdings wird es im Detail sicher noch im Verlauf des Projekts Änderungen wie beispielsweise Erweiterungen und Ergänzungen erfahren.

Des Weiteren liegen die, in den vorangegangenen Kapiteln detailliert dargestellten Schnittstellen, in der momentanen Form in einer plattformspezifischen Ausprägung in Java vor, da beide Referenzimplementierungen mit Java entwickelt wurden. Es ist allerdings konzeptionell möglich die entsprechenden Interfaces und deren Implementierungen auch auf andere Plattformen und andere Programmiersprachen zu übertragen, um auch dort die Funktionalitäten des MOHITO-Frameworks zur Verfügung stellen zu können.

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.

7 Literaturverzeichnis

Copeland, G., & Maier, D. (84). Making smalltalk a database system. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* .

Fowler, M. (n.d.). *Catalog of Patterns of Enterprise Application Architecture*. Retrieved 2013 йил 25-9 from http://martinfowler.com/eaaCatalog/tableDataGateway.html

(2013). *L 8.1: Initiale Version der Generatortemplates und des Modellinterpreters für eine clientseitige und eine serverseitige Plattform.* MOHITO Konsortium.

Mohito, K. (2013). *Erweiterte Referenzarchitektur für Multi-Plattformhomogenisierung von Datenhaltungsschichten* .

Mohito, K. (2013). Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik.

Mohito, K. (2013). Frameworkentwicklung, Multi-Plattform.

Neward, T. (2006 йил 6). *Ted Neward's Technical Blog*. Retrieved 2013 йил 25-9 from http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

© 2013 MOHITO Projektkonsortium. Alle Rechte vorbehalten.