|  |
| --- |
| Elmer Lukas, Heidt Christina, Steiner Diego, Treichler Delia, Waltenspül Remo  13. Mai 2011 |

|  |
| --- |
| SE2 Projekt MRT |
| Software Architektur |
|  |

****

# Dokumentinformationen

## Änderungsgeschichte

|  |  |  |  |
| --- | --- | --- | --- |
| Datum | Version | Änderung | Autor |
| 28.03.2011 | 1.0 | Erste Version des Dokuments | HC |
| 11.04.2011 | 1.1 | Vervollständigung des Inhaltsverzeichnisses | TD |

## Inhaltsverzeichnis

[1 Dokumentinformationen 1](#_Toc292885981)

[1.1 Änderungsgeschichte 1](#_Toc292885982)

[1.2 Inhaltsverzeichnis 1](#_Toc292885983)

[1.3 Abbildungsverzeichnis 3](#_Toc292885984)

[2 Einführung 4](#_Toc292885985)

[2.1 Zweck 4](#_Toc292885986)

[2.2 Gültigkeitsbereich 4](#_Toc292885987)

[2.3 Definitionen und Abkürzungen 4](#_Toc292885988)

[2.4 Referenzen 4](#_Toc292885989)

[3 Software Architektur 5](#_Toc292885990)

[3.1 Architektonische Ziele & Einschränkungen 5](#_Toc292885991)

[3.1.1 Ziele 5](#_Toc292885992)

[3.1.2 Einschränkungen 5](#_Toc292885993)

[3.2 Architekturübersicht 6](#_Toc292885994)

[3.3 Kernarchitektur Übertragung Zeiteintrag 7](#_Toc292885995)

[3.3.1 Sequenzdiagramm 7](#_Toc292885996)

[3.3.2 Zustandsdiagramm 8](#_Toc292885997)

[3.4 Systemstruktur 9](#_Toc292885998)

[3.4.1 Physische Sicht 9](#_Toc292885999)

[3.4.2 Logische Sicht Rails 9](#_Toc292886000)

[3.4.3 Logische Sicht Android 11](#_Toc292886001)

[3.4.4 Schnittstellen der Packages 15](#_Toc292886002)

[3.4.5 Architekturkonzepte 15](#_Toc292886003)

[4 Beschreibung der Packages 15](#_Toc292886004)

[4.1 Package Name 15](#_Toc292886005)

[4.1.1 Klassenstruktur 15](#_Toc292886006)

[4.1.2 Architekturkonzepte für Package (falls vorhanden) 15](#_Toc292886007)

[4.1.3 Klassenspezifikationen 15](#_Toc292886008)

[4.1.4 Interaktionen innerhalb Package (falls sinnvoll) 15](#_Toc292886009)

[4.2 Package Name 16](#_Toc292886010)

[4.2.1 Klassenstruktur 16](#_Toc292886011)

[4.2.2 Architekturkonzepte für Package (falls vorhanden) 16](#_Toc292886012)

[4.2.3 Klassenspezifikationen 16](#_Toc292886013)

[4.2.4 Interaktionen innerhalb Package (falls sinnvoll) 16](#_Toc292886014)

[4.3 Package Name 16](#_Toc292886015)

[4.3.1 Klassenstruktur 16](#_Toc292886016)

[4.3.2 Architekturkonzepte für Package (falls vorhanden) 16](#_Toc292886017)

[4.3.3 Klassenspezifikationen 16](#_Toc292886018)

[4.3.4 Interaktionen innerhalb Package (falls sinnvoll) 16](#_Toc292886019)

[5 Real Use Cases 17](#_Toc292886020)

[6 Prozesse und Threads 17](#_Toc292886021)

[7 Datenspeicherung 17](#_Toc292886022)

[7.1 Persistenz Rails 17](#_Toc292886023)

[7.2 Persistenz Android 17](#_Toc292886024)

[8 Grössen und Leistung 17](#_Toc292886025)

[9 Anhang 18](#_Toc292886026)

[9.1 Architekturentscheide 18](#_Toc292886027)

[9.1.1 Datenspeicherung / ORM 18](#_Toc292886028)

## Abbildungsverzeichnis

[Abbildung 1 - Architekturübersicht 5](file:///E:\EigeneDateien\Studium\WorkspaceEclipse\se2p_svn.elmermx.ch\doc\05_Design\Software%20Architektur.docx#_Toc292112116)

[Abbildung 2 - Sequenzdiagramm Übertragung TimeEntry 6](file:///E:\EigeneDateien\Studium\WorkspaceEclipse\se2p_svn.elmermx.ch\doc\05_Design\Software%20Architektur.docx#_Toc292112117)

[Abbildung 3 - Zustandsdiagramm Übertragung TimeEntry 7](#_Toc292112118)

[Abbildung 4 - Deployment Diagram MRT 8](#_Toc292112119)

[Abbildung 5 - Ablauf eines Requests, Quelle: http://gmoeck.github.com/2011/03/10/sproutcore-mvc-vs-rails-mvc.html 10](#_Toc292112120)

[Abbildung 6 - Übersicht über die Packages 11](#_Toc292112121)

[Abbildung 7 - Package interfaces 12](#_Toc292112122)

[Abbildung 8 - Package models 12](#_Toc292112123)

[Abbildung 9 - Package database 13](#_Toc292112124)

[Abbildung 10 - Package services 13](#_Toc292112125)

[Abbildung 11 - Package gui/gen 13](#_Toc292112126)

[Abbildung 12 - Package activities 14](#_Toc292112127)

# Einführung

## Zweck

Dieses Dokument beschreibt die Software Architektur für das Projekt MRT (Mobile Reporting Tool).

## Gültigkeitsbereich

Dieses Dokument ist während der gesamten Projektdauer gültig (21.02 bis 03.06.2011).

## Definitionen und Abkürzungen

Die Definitionen und Abkürzungen befinden sich in der ausgelagerten Datei doc/01\_Projektplan/glossar.docx.

## Referenzen

* doc/03\_Anforderungsspezifikationen/\*
* doc/04\_Domainanalyse/\*
* doc/05\_Design/\*
* doc/templates/template.dotx
* doc/templates/java\_formatting.xml
* doc/templates/ruby\_formatting\_settings.zip
* doc/media/logo.png

# Software Architektur

## Architektonische Ziele & Einschränkungen

<Beschreibt die Softwareanforderungen und Objekte, welche einen Einfluss auf die Architektur haben [Bspl: Safety, Security, Privacy, Distribution, …] Beinhaltet auch eine Beschreibung von Design und Implementationsstrategie, Teamstruktur, Entwicklungstools, Zeitplan, etc…>

### Ziele

* Es soll möglichst einfach möglich sein, den Android Client durch einen anderen Client (z.B. iPhone, Windows 7 Phone) zu ersetzen. Deshalb darf die Architektur keine Android-spezifischen Konzepte enthalten.
* Die Benutzeroberfläche des Android Clients soll sehr einfach und übersichtlich gestaltet werden. Deshalb stellt der Android Client nur die wichtigsten Funktionalitäten zur Verfügung.
* Mehrere Mitarbeiter müssen gleichzeitig Stundeneinträge machen können. Darum müssen die einzelnen Einträge voneinander getrennt werden.
* Die Daten müssen konsistent bleiben, auch wenn die Datenverbindung zwischen Client und Server unterbrochen wird. Das soll gezielt durch geplante Abläufe erreicht werden.
* Die Schnittstellen zwischen Client und Server sollen genau definiert werden, damit die zwei Teile unabhängig voneinander entwickelt werden könnten.

### Einschränkungen

* Der Android Client wird über den Touchscreen bedient. Deshalb muss darauf geachtet werden, dass beispielsweise die Buttons gross genug sind.
* Da für die Entwicklung des Prototyps nur wenig Zeit zur Verfügung steht und es in diesem Projekt nicht primär um das grafische Erscheinungsbild geht, wird dieses nicht ausgereift sein.

## Architekturübersicht

Wie anhand der nachstehenden Abbildung ersichtlich, besteht die Architektur aus einem Client-Server System. Als Clients werden Computer sowie Smartphones mit dem Android Betriebssystem eingesetzt. Die Android Mobiltelefone verwenden eine eigene kleine Datenbank, welche verwendet, wird falls ein Zeiteintrag nicht direkt übermittelt werden kann.



Abbildung 1 - Architekturübersicht

## Kernarchitektur Übertragung Zeiteintrag

### Sequenzdiagramm

Die folgende Abbildung zeigt den Ablauf beim Übertragen eines Zeiteintrags zwischen dem Client (Android) und dem Server.



Abbildung 2 - Sequenzdiagramm Übertragung TimeEntry

### Zustandsdiagramm

Das Zustandsdiagramm zeigt den Ablauf beim Übertragen eines Zeiteintrags mit allen Übergängen und Aktionen.



Abbildung 3 - Zustandsdiagramm Übertragung TimeEntry

## Systemstruktur

### Physische Sicht



Abbildung 4 - Deployment Diagram MRT

Da der Server die Rails Applikation hostet, wird er als Server Execution Node dargestellt. Der Client verbindet sich mit dem Server mit Hilfe des Protokolls HTTP/HTTPS. Deshalb wird der Client als Client Execution Node dargestellt.

Zu beachten ist, dass auf dem Server und dem Client je eine andere Applikation läuft. Auf dem Server ist dies eine Rails Applikation (rails\_app), auf dem Client eine Android Applikation (mrt.apk).

Server und Client kommunizieren über HTTP/HTTPS. Natürlich läuft HTTP/HTTPS über TCP/IP. Eine klare Abgrenzung ist hier wichtig, demzufolge ist für das Projekt die Schicht HTTP/HTTPS die tiefste Schicht.

Eine ausführliche Begründung für den Architekturentscheid ist im Anhang 7.1 Architekturentscheide zu finden.

### Logische Sicht Rails

Die Architektur der Rails Applikation ist zu einem grossen Teil vorgegeben. Diese wird nachfolgend beschrieben.

Rails setzt sich aus nachfolgenden individuellen Komponenten zusammen:

#### Action Pack

Das Action Pack besteht aus drei Komponenten, dem Action Controller, der Action View und dem Action Dispatch. Es ist der View Controller - Teil des Architekturmusters “MVC”.

##### Action Controller

Der Action Controller ist die Komponente, welche die Controller in einer Rails-Applikation steuert. Das Framework leitet eingehende Anfragen an die Applikation weiter, extrahiert die Parameter und verteilt diese an die zuständige Aktion. Der Service umfasst das Sitzungsmanagement, das Rendering von Templates und das Weiterleitungsmanagement.

##### Action View

Die Action View verwaltet die Ansichten der Rails-Applikation. Standardmässig kann HTML und XML Output generiert werden. Die Action View beinhaltet das Rendering von Templates, inklusive verschachtelten und Teil-Templates. Zudem schliesst es einen eingebauten AJAX Support ein.

##### Action Dispatch

Der Action Dispatch führt das Routing von Web-Anfragen aus und verteilt diese, je nach Wunsch, an die eigene oder eine andere Rack-Applikation.

#### Action Mailer

Der Action Mailer ist das Framework für einen eingebauten E-Mail Services. Er kann benutzt werden, um eingehende Nachrichten zu empfangen und zu verarbeiten. Einfache Textmeldungen oder komplexe, mehrteilige Emails, auf flexiblen Templates basierend, können versandt werden.

#### Active Model

Das Active Model stellt ein vordefiniertes Interface bereit, welches zwischen dem Service des Action Packs und Object Relationship Mapping-Komponenten wie Active Record fungiert. Es erlaubt Rails, andere ORM Frameworks anstelle des Active Records zu verwenden, sofern dies für die Applikation nötig ist.

#### Active Record

Der Active Record ist die Grundlage der Models einer Rails-Applikation. Er stellt Datenbank-Unabhängikeit, CRUD Funktionalitäten, fortgeschrittene Suchfähigkeiten und die Möglichkeit, Models miteinander oder anderen Services zu verknüpfen, bereit.

#### Active Resource

Die Active Resource bietet ein Framwork zur Verwaltung der Verbindung zwischen Business Objekten und RESTful Web Services. Es implementiert eine Methode, um webbasierte Ressourcen auf lokale Objekte mit CRUD Semantik abbilden zu können.

#### Active Support

Der Active Support ist eine umfangreiche Sammlung an Werkzeugklassen und Standard-Bibliothekserweiterungen von Ruby, welche in Rails, im Kerncode und den Applikationen, benutzt werden.

#### Railties

Railties ist der Kerncode von Rails, welcher neue Rails-Applikationen erzeugt und verschiedene Frameworks und Plugins in einer Rails-Applikation vereint.

#### Rails MVC

##### Models

Ein Model verkörpert die Informationen/Daten einer Applikation und die Regeln, um diese zu manipulieren. Bei Rails werden die Models vor allem verwendet, um die Interaktion mit einer dazugehörigen Datenbank Tabelle zu regeln. In den meisten Fällen wird sich eine Tabelle auf genau ein Model beziehen. Der Grossteil der Business Logik der Applikation befindet sich in den Models.

##### Views

Views stellen das User Interface einer Applikation dar. Bei Rails handelt es sich hierbei meist um HTML Dateien mit eingebettetem Ruby Code. Diese kümmern sich nur um die Präsentation der Daten. Views übergeben die Daten dem Webbrowser oder einem anderen Programm, welches Anfragen an die Applikation sendet.

##### Controllers

Die Controller könnten als Klebstoff zwischen Models und Views bezeichnet werden. Controllers sind in Rails dafür verantwortlich, die eingehenden Anfragen vom Webbrowser weiterzuleiten, von den Models Daten abzufragen und Daten an die Views weiterzuleiten.



Abbildung 5 - Ablauf eines Requests, Quelle: http://gmoeck.github.com/2011/03/10/sproutcore-mvc-vs-rails-mvc.html

##### Schritte

1. Der Browser sendet eine Anfrage an den Webserver.
2. Der Webserver verarbeitet die Anfrage, ermittelt die entsprechende Route und leitet die Anfrage an die dazugehörige Controller Methode weiter.
3. Der Controller erfragt danach all die benötigten Informationen vom Model Layer, die für den aktuellen Request benötigt werden.
4. Der Model Layer sammelt alle Informationen und gibt diese dem Controller zurück.
5. Der Controller gibt die evaluierten Informationen der View weiter und fordert sie auf, diese zu übersetzen.
6. Die View rendert sich selbst und übermittelt das übersetzte HTML dem Controller.
7. Der Controller erstellt daraus das gesamte HTML (inkl. Layout) der angefragten Seite und übergibt sie dem Webserver.
8. Der Webserver gibt die Seite an den Browser zurück.

### Logische Sicht Android

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

<Beschreibung mit Text und Diagramm der Architektur. Aufteilung in Packages (zum Beispiel: 3-Layer-Architektur mit GUI, Problem Domain und Datenhaltung)>

Diagramm mit Packages und Dependencies zwischen Packages

o Kurzbeschreibung der Packages

o Eventuell Kurzbeschreibung wesentlicher Klassen (Exportklassen, Facades)

o Einzelne Packages sind z.T. Technologiekomponenten wie GUI-Library

Kurze Beschreibung der gewählten Technologiekomponenten mit Begründung für ihre

Wahl (ev. Verweis auf einen Anhang, der Varianten diskutiert)

#### Übersicht über die Packages



Abbildung 6 - Übersicht über die Packages

Die jeweiligen Packages beinhalten Klassen, die Testzwecken dienen (grau hinterlegt).

Das Package „gui/gen“ läuft selbständig und benötigt keine Testklassen. Das Package interface benötigt ebenfalls keine Testklassen.

#### Kurzbeschreibung der Packages

##### Package interfaces

Das Package „interfaces“ beinhaltet die JSON-Serialisierung, welche von den Klassen im Package „models“ benötigt werden.



Abbildung 7 - Package interfaces

##### Package network

Das Package „network“ ist dafür verantwortlich, dass die Daten übertragen werden.

##### Package models

„models“ beinhaltet alle Daten, welche persistent gespeichert werden sollen.



Abbildung 8 - Package models

##### Package database

„database“ ist dafür verantwortlich, dass eine Verbindung zur Datenbank hergestellt wird. Ist es zuständig für die Erstellung der Datenbank, mit den einzelnen Tabellen.



Abbildung 9 - Package database

##### Packages service

Das Package „services“ ist dafür verantwortlich, dass Daten vom Server auf den Client und in die entgegengesetzte Richtung, also vom Client auf den Server, geladen werden.



Abbildung 10 - Package services

##### Package gui/gen

„gui/gen“ wird vom Android generiert. Durch dieses Package können die User Interfaces Angaben, welche in einem separaten XML-File gespeichert sind, identifiziert werden.



Abbildung 11 - Package gui/gen

##### Package activities

Dieses Package ist die Schnittstelle des Benutzers. In den „activities“ wird die Problem Domain abgehandelt. „activities“ nehmen Benutzerbefehle entgegen, bearbeiten „models“ und starten „services“.



Abbildung 12 - Package activities

### Schnittstellen der Packages

<Exportklassen der Packages mit allen public-Methoden

Architektur der Interaktion (z.B. zwischen Domain GUI: MVC-Architektur), Ev. Illustration mit Interaktionsdiagrammen>

### Architekturkonzepte

<Generell angewandte Lösungsmuster, die nicht einem Package zugeordnet werden, wie Errorhandling, Logging- und Debugging-Mechanismen, Speicherverwaltung u.a.m. mit Begründungen für gewählte Lösung.

Multitasking und Multithreading, d.h. Process View: aktive Objekte (wenn nötig)>

# Beschreibung der Packages

## Package interfaces

### Beschreibung

Das Package „interfaces“ dient als Schnittstelle zwischen Server und Client. Die verschiedenen Interfaces stellen Methoden zur Verfügung, welche die erfolgreiche Datenübertragung sicherstellen. Die Interfaces werden von verschiedenen Klassen des Packages „models“ implementiert. Verwendet werden sie von den Klassen im Package „network“, um die Daten zu übermitteln. Die Beschreibung der abstrakten Funktionalität vereinfacht es wesentlich, weitere Klassen dem Package „models“ hinzuzufügen.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

#### Confirmable

Das Interface Confirmable dient dazu einen Stundeneintrag auf dem Server zu bestätigen. Dadurch wird sichergestellt, dass Objekte erfolgreich an den Server übertragen wurden und diese auf dem Client entfernt werden können.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| getIdOnServer() | Gibt die ID zurück, welche vom Server gesetzt wurde. |
| processConfirmation(JSONObject jsonObject) | Gibt zurück ob die vom Server gesetzte ID mit der des erhaltenen JSONObjects übereinstimmt. |

#### Receivable

Das Interfaces Receivable stellt Methoden zur Verfügung um die Übertragung von Elementen vom Server an den Client zu ermöglichen.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| getIdOnServer() | Gibt die ID zurück, welche vom Server gesetzt wurde. |
| getUpdatedAt() | Gibt den Zeitpunkt der letzten Änderung in Form eines Timestamps zurück. |
| fromJSON(JSONObject jsonObject) | Dem erhaltenen JSONObject können die Attribute der implementierenden Klasse entnommen werden. Die Methode returniert ob dies erfolgreich verlaufen ist. |

#### Transmittable

Das Interfaces Transmittable ermöglicht es den implementierenden Klassen Objekte an den Server zu schicken. Zudem wird sichergestellt, dass Objekte nicht mehr als einmal auf dem Server gespeichert werden.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| toJSONObject() | Erstellt und ein neues JSONObject und setzt dort die jeweiligen Attribute der implementierenden Klasse. Danach wird dieses zurück gegeben. |
| isTransmitted() | Vergleicht die gespeicherte ID des |
| processTransmission(JSONObject jsonObject) | Durch das erhaltene JSONObject wird der Hashcode verglichen und die ID gesetzt. Stimmt der Hashcode auf Server und Client Seite überein, wird dieser danach serverseitig entfernt. Es wird zurückgegeben ob dies erfolgreich war. |

## Package network

### Beschreibung

Das Package „network“ ist dafür zuständig, Daten zu übertragen. Hierbei handelt es sich um die Daten die durch die Klassen des Packages „models“ gespeichert werden.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

### Interaktionen innerhalb Package (falls sinnvoll)

## Package models

### Beschreibung

Im Package „models“ befinden sich alle Klassen, die dazu dienen Daten persistent zu speichern. Dabei handelt es sich einerseits um verschiedene Daten welche der Client von der Server Seite benötigt, anderseits um Daten die an den Server übertragen werden müssen.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

Die Klassen Customer, GSP Position, TimeEntryType sowie User implementieren alle das Interface Receivable. Die damit implementierten Methoden werden im Abschnitt 4.1.4.2 Receivable erläutert. Die Klasse TimeEntry implementiert die Interfaces Confirmable (siehe 4.1.4.1 Confirmable) und Transmittable (siehe 4.1.4.3 Transmittable).

### Interaktionen innerhalb Package (falls sinnvoll)

## Package database

### Beschreibung

Das Package „database“ ist für die Verbindung zur Datenbank verantwortlich. Es ist zuständig für die Erstellung der Datenbank und deren Tabellen. Durch die Klasse DatabaseHelper können die verschiedenen Klassen der „activities“ auf die Datenbank zugreifen. Das weiteren kann darüber die Datenbank auch zurückgesetzt werden.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

### Interaktionen innerhalb Package (falls sinnvoll)

## Package services

### Beschreibung

Das Package „Services“ ist dafür verantwortlich, dass die Daten vom Server an den Client übertragen werden. Gleiches gilt für die entgegengesetzte Richtung, also vom Client an den Server.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

#### Synchronizer

#### Null Object

Bei der Klasse LocationListenerAdapter, welche das Interface LocationLister implementiert, handelt es sich um ein den Einsatz eines Null Objects. Hierbei wird anstatt, dass eine null Referenz ve

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

#### Synchronizer

Das Interface Synchronizer wird von fünf Klassen implementiert, welche für die Synchronisation der verschiedenen, im Package „models“ gespeicherten Objekte, zuständig ist.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| synchronize() | Synchronisiert für die vorgesehene Klasse jeweils die Datenbank. |

### Interaktionen innerhalb Package (falls sinnvoll)

## Package gui/gen

### Beschreibung

Das Package „gui/gen“ wird von Android automatisch generiert. Durch die darin befindliche Klasse R kann auf die in den XML-Dateien definierten User Interface Angaben zugegriffen werden. Dabei handelt es sich um beispielsweise um XML-Daten über die Darstellung der verschiedenen Layouts, Listen oder Menüs.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

### Interaktionen innerhalb Package (falls sinnvoll)

## Package activities

### Beschreibung

Das Package „activities“ stellt die oberste Schicht und somit die Schnittstelle zum Benutzer dar. Die Activities nehmen die Benutzerbefehle entgegen und leiten diese an die unteren Schichten weiter. Sie bearbeiten die Models und starten die verschiedenen Services.

### Klassenstruktur

<• Klassendiagramm mit Design-Information wie Zugriffsebenen, Navigationspfaden, Containment (By

Value, By Reference)>

### Architekturkonzepte für Package (falls vorhanden)

<• Umsetzung der generellen Architekturkonzepte

• Spezifische Architekturkonzepte für Package>

### Klassenspezifikationen

<• nur wo sinnvoll, insbesondere Schnittstellenklassen

o mit wesentlichen Attributen und Methoden (meist als Javadoc)

o Eventuell Datentypen für Attribute, Parameter und Rückgabewerte mit Datentypen für

Methoden, eventuell Zustandsdiagramm für Klasse (falls sinnvoll)>

### Interaktionen innerhalb Package (falls sinnvoll)

# Real Use Cases

<• Interaktionsdiagramme für Use Cases bzw. Systemoperationen

• Wenn nur Interaktionen, in einem Package gezeigt (z.B. Domain), dann auch als 4.1.4 Interaktionen innerhalb Package>

# Prozesse und Threads

<Wenn mehrere Prozesse oder Threads eingesetzt werden wird hier beschrieben, wie diese ablaufen, miteinander funktionieren, Daten austauschen, sich synchronisieren, etc...>

## Server

Der Ruby on Rails Server kann gleichzeitig mehrere Clients bedienen und funktioniert mit mehreren Prozessen. Hinter der Webapplikation steht ein MySQL Server, der die Daten persistent speichert. Auf der Datenbank können die einzelnen Transaktionen nur seriell erfolgen. Die Synchronisation wird vom Ruby on Rails Framework im Zusammenspiel mit der Datenbank übernommen.

Da die Webapplikations-Prozesse über die Datenbank und nicht direkt interagieren, werden diese nicht weiter beschrieben.

## Client

Auf dem Android Client gibt es zwei wichtige Prozesse in Bezug auf MRT. Der eine Prozess ist für die aktuelle Activity verantwortlich, der zweite Prozess für die Synchronisation. Die beiden Prozesse tauschen nicht direkt Daten aus, sondern nur indirekt über die SQLite Datenbank. Die Datenbank ist für die Synchronisation verantwortlich. Konkret heisst das, dass weder die Activity noch die Synchronisationsservices voneinander wissen, wenn etwas in der Datenbank geändert wurde.

### Activity

Die Activity legt neue Stundeneinträge an, die zum Server übertragen werden müssen. Die Übertragung wird zu einem späteren Zeitpunkt vom Synchronisationsservice gestartet.

Zusätzlich benötigt die Activity sowohl Stundeneintragstypen und Kunden, damit der Benutzer jeweils einen an einem Stundeneintrag zuweisen kann. Diese werden periodisch von der Datenbank gelesen und im GUI angezeigt.

### Synchronisationsservice

Der Synchronisationsservice ist für die Synchronisation der Stundeneinträge, der Kunden und der Stundeneintragstypen verantwortlich. Die Datensätze werden jeweils direkt von der Datenbank gelesen / in die Datenbank geschrieben bzw. vom Server gelesen / dem Server gesendet. Die Synchronisation wird mithilfe der Felder created\_at, updated\_at und deleted\_at realisiert. Also werden nur die jeweiligen Daten synchronisiert, die noch nicht übertragen wurden oder tatsächlich geändert haben.

Da der Service nichts von der Activity oder dem Server weiss, also auch nicht wann ein neuer Stundeneintrag / Kunde / Stundeneintragstyp angelegt wurde, muss die Synchronisation periodisch erfolgen.

# Datenspeicherung

<Beschreibung mit Diagramm der Datenspeicherung [Data Model]. (zum Beispiel: Datenbank)>

## Persistenz Rails

Auf dem Server werden sämtliche Daten persistent gespeichert. Um dies durchzuführen, werden alle Domänenmodelle mitsamt den Attributen unabhängig von der konkreten Datenbankimplementierung auf so genannte ActiveRecords abgebildet.

Um das Schema der Datenbank aufzubauen werden anhand sogenannter Migrationsdateien die passenden Tabellen in der Datenbank erzeugt. Jede Migrationsdatei enthält einen eindeutigen Timestamp, damit diese in genau einer Reihenfolge ausgeführt werden. Zusätzlich verfügt jede Datei über eine up und down Methode. Die up Methode ist zum Beispiel dafür verantwortlich, eine neue Tabelle anzulegen, während die down Methode jeweils immer das Gegenteil der up Methode machen soll, zum Beispiel eine Tabelle löschen. Dadurch ist es möglich, die Migrationen einzeln einzuspielen oder auch Rollbacks zu machen. Die Migrationen sind natürlich auch datenbankimplementationsunabhängig.

Die Datenbank wurde nach dem Prinzip „Convention over Configuration“ erzeugt. In Ruby on Rails bedeutet das folgende wichtigsten Dinge:

* Tabellennamen und Feldnamen werden klein\_und\_mit\_underscore geschrieben
* Jede Tabelle verfügt über folgende Attribute
  + id: Die automatisch inkrementierte ID des Records
  + created\_at: Datum und Zeit, wann der Record erstellt wurde
  + updated\_at: Datum und Zeit, wann der Record zuletzt editiert wurde
* Jeder Fremdschlüsselname setzt sich aus Fremdtabelle und „\_id“ zusammen. Zum Beispiel lautet der Fremdschlüssel für die Tabelle „time\_entry“ dann „time\_entry\_id“.

In MRT wird als Datenbankimplementierung die verbreitete und als performant bekannte Implementierung MySQL eingesetzt.

## Persistenz Android

Die Daten werden mithilfe des ORMLite in einer lokalen SQLite Datenbank gespeichert. Dieses ORM hat sich während der Evaluation sehr bewährt. Weitere Infos zum Entscheid sind unter 7.1.1 Datenspeicherung / ORM einsehbar.

# Grössen und Leistung

## Server

Die Speicherauslastung kann je nach Serverkonfiguration stark variieren. Oftmals bedeutet mehr Speicher natürlich bessere Performance.

Für die Ruby on Rails Applikation wird auf dem Server pro Prozess ca. 20-30 MB Ram benötigt. Dazu kommt noch der Apache, der je nach Konfiguration ca. 250 MB Ram benötigt.

## Client

<Einschränkungen der Applikation bezüglich Speicher, Leistung, etc…. (zum Beispiel: Verwaltung unterstützt maximal 20'000 Einträge)>

# Anhang

## Architekturentscheide

### Datenspeicherung / ORM

Dass die Daten temporär auf dem Client gespeichert werden müssen, ergibt sich durch die Anforderung, dass der Client auch ohne Serververbindung funktionieren muss. Die verschiedenen Varianten wurden nachfolgend untersucht und ausgewertet.

Schlussendlich wurde für ORMLite entschieden, da sich dieser OR Mapper am besten für die Android Plattform eignet und speziell dafür optimiert wurde. Ausserdem ist dieser OR Mapper schlank und die angebotene Grundfunktionalität reicht völlig aus.

#### Kein ORM, primitives SQLite

Jedes ORM ist ein wenig anders realisiert, verfolgt eine andere Strategie oder hat andere Befehle. Kurz gesagt: jedes ORM bedeutet Einarbeitungszeit. Da das Projekt nicht sehr gross ist und es in erster Linie ums Software Engineering geht, wurde diese Möglichkeit natürlich in Betracht gezogen.

Leider wurde schnell festgestellt, dass ohne ORM schnell redundanter Code entsteht, und dass schnell ein kleines ORM entwickelt werden müsste (mit Annotations oder Reflections), um Redundanzen zu vermeiden. Da dies viel Aufwand wäre oder ohne eigenes ORM den Prinzipien eines guten Software Engineerings wider­sprechen würde (DRY), wurde nach einer alternativen Lösung gesucht.

#### ORMLite

ORMLite ist ein leichtgewichtiger OR Mapper für Java Applikationen (Grösse: 234 KB). Er abstrahiert die Grundfunktionalitäten des SQL. Es werden viele verschiedene Datenbanken nativ unterstützt (volle Unterstützung für SQLite, MySQL, Postgres, Microsoft SQL Server, H2, Derby, HSQLDB, teilweise Unterstützung für DB2, Oracle, ODBC und Netezza) und ist leicht für andere Datenbanken erweiterbar, was für den modularen Aufbau des OR Mappers spricht. Zusätzlich bietet ORMLite eine einfache und strukturierte Integration in die Android Plattform.

Weitere Informationen unter http://ormlite.com/

#### DB4O

DB4O ist ein Schwergewicht in Sachen ORM (> 40MB!). Zusätzlich bietet DB4O sehr viel Funktionalität, die für dieses Projekt nicht genützt werden würde. Aus diesen zwei Gründen wurde gegen DB4O entschieden.

Weitere Informationen unter http://developer.db4o.com/

#### ActiveAndroid

ActiveAndroid ist ein schlankes aber mächtiges ORM, das speziell für Android entwickelt wurde. Leider ist dieses ORM nicht OpenSource und kostet für 5 Entwickler 60 USD. Aus diesem Grund wird ActiveAndroid nicht verwendet.

Weitere Informationen unter https://www.activeandroid.com/

#### Android-active-record

Android-active-record ist zwar grundsätzlich ein sehr gutes ORM, da es sehr klein ist und sich sehr einfach in das Android Framework integrieren lässt. Allerdings ist dieses ORM noch stark in der Entwicklungsphase und ist somit noch nicht stabil genug, um in einer produktiven Applikation eingesetzt zu werden.

Weitere Informationen unter http://code.google.com/p/android-active-record/