|  |
| --- |
| Elmer Lukas, Heidt Christina, Steiner Diego, Treichler Delia, Waltenspül Remo  31. Mai 2011 |

|  |
| --- |
| SE2 Projekt MRT |
| Codereview |
| Informales Codereview |

****

# Dokumentinformationen

## Änderungsgeschichte

|  |  |  |  |
| --- | --- | --- | --- |
| Datum | Version | Änderung | Autor |
| 04.05.2011 | 1.0 | Erste Version des Dokuments | WR |

## Inhaltsverzeichnis

[1 Dokumentinformationen 1](#_Toc294613512)

[1.1 Änderungsgeschichte 1](#_Toc294613513)

[1.2 Inhaltsverzeichnis 1](#_Toc294613514)

[2 Codemetriken 2](#_Toc294613515)

[2.1 Rails 2](#_Toc294613516)

[2.1.1 Metric\_fu 2](#_Toc294613517)

[2.1.2 Schlussfolgerung 2](#_Toc294613518)

[2.2 Android 2](#_Toc294613519)

[2.2.1 State of Flow Eclipse Metrics 2](#_Toc294613520)

[2.2.2 State 101 7](#_Toc294613521)

[2.2.3 Schlussfolgerung 9](#_Toc294613522)

# Codemetriken

Für die Metrikanalyse des implementierten Codes wurden unter Rails das Gem Metric\_fu und unter Android die Anwendungen State of Flow Eclipse Metrics sowie Structure 101 verwendet.

Die nachfolgend beschriebenen Metrik-Indikatoren wiederspiegeln nur die interessantesten Werte bzw. Auffälligkeiten. Um eine ausführliche Metrik-Analyse zu erhalten, wird auf die unter dem Kapitel 2.4 Referenzen aufgeführten Ordner verwiesen.

## Rails

### Metric\_fu

### Schlussfolgerung

## Android

Da die Testklassen für die Sicherstellung von diversen Funktionen verantwortlich sind, werden die Kohäsionswerte zum Teil geringer als es gewünscht wäre. In der Regel wurde für eine zu testende Klasse eine Testklasse entworfen. Ein weiteres aufteilen auf mehrere Klassen wäre nicht sinnvoll, da die Testklassen nur für den Test erzeugt wird und keinen weiteren Nutzen hat.

Aus diesen Gründen wird in den nachfolgenden Kapiteln eine Analyse des Codes exklusive der durchgeführten Tests vorgenommen.

### State of Flow Eclipse Metrics

Das Eclipse-Plugin State of Flow Eclipse Metrics wird verwendet um diverse Code Qualitätsindikatoren wie zum Beispiel McCabe’s Cyclomatic Complexity, Efferent Coupling, Feature Envy, Lines of Code in Method etc. aus dem Code zu berechnen. Diese werden zudem laufend während der Entwicklung neu ausgewertet und direkt in der IDE als Marker angezeigt.

#### Cyclomatic Complexity

Wie aus der nachfolgenden Abbildung 1 - Übersicht Cyclomatic Complexity ersichtlich, ist die zyklomatische Komplexität relativ gering. Sämtliche Klassen befinden sich innerhalb des gewünschten Bereichs von eins bis vier.

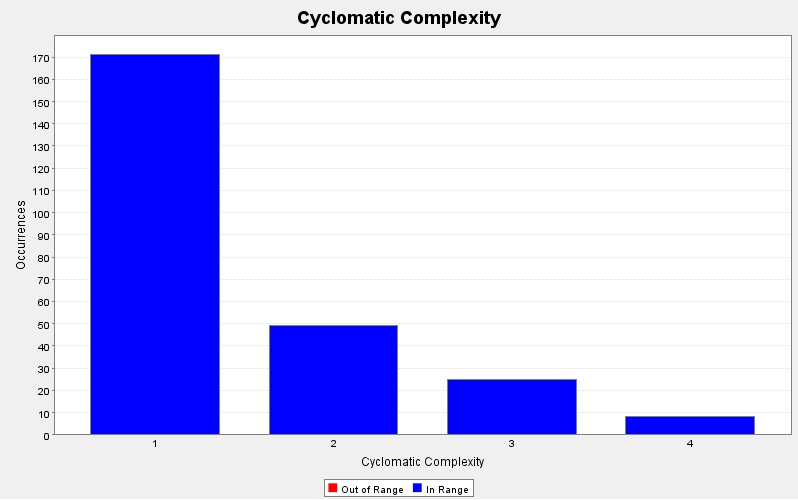


Abbildung - Übersicht Cyclomatic Complexity

#### Lines of Statements

Der Grenzwert für die Anzahl Anweisungen in einer Methode wurde auf 20 festgelegt. Alle Methoden liegen innerhalb des erwarteten Wertes (siehe Abbildung 2 - Übersicht Lines of Code).

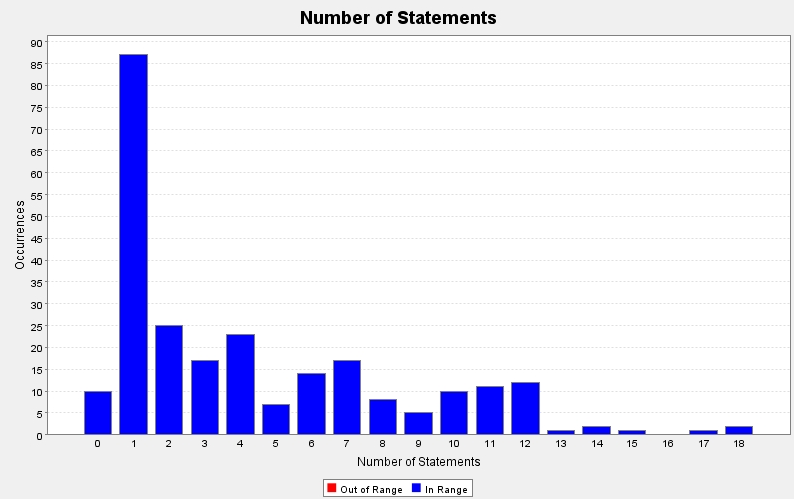


Abbildung - Übersicht Lines of Code

#### Weighted Methods per Class

Der Metrikindikator Weighted Methods per Class steht für die kumulierte Komplexität sämtlicher Methoden einer Klasse. Das Eclipse Metric Plugin verwendet für die Komplexitätsanalyse den MC Cabe’s Algorithmus. Je höher der Wert liegt, desto höher ist die Komplexität einer kompletten Klasse. Der gewünschte Bereich für diese Kenngrösse liegt zwischen 0 und 40.

Wie aus der nachstehenden Abbildung 3 - Übersicht Weighted Methods per Class ersichtlich, existieren zwei Klassen, welche den definierten Grenzwert knapp überschreiten.

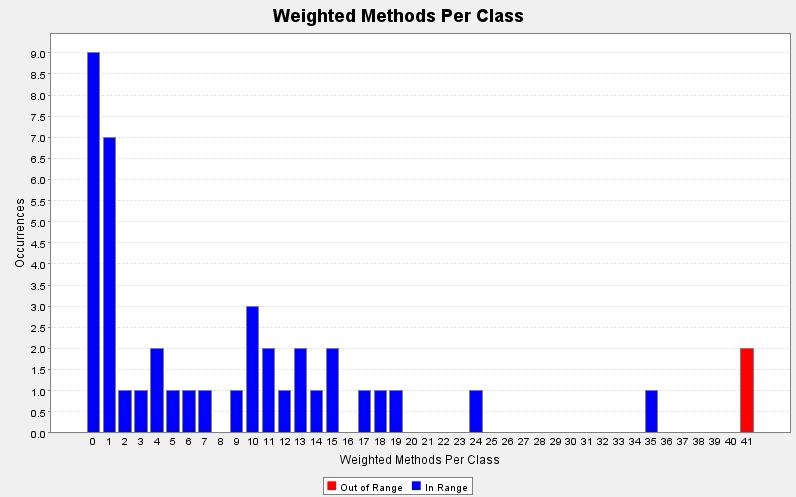
****

Abbildung - Übersicht Weighted Methods per Class

**Erklärung hohe Komplexität**

Bei den Klassen mit überschrittener Komplexität handelt es sich um TimeEntryActivity sowie Customer. Der Schwellwert wird nur knapp um eins übertreten. Dies in erster Linie aufgrund der vielen kurzen Methoden, da jede Methode die Komplexität um eins inkrementiert. Indes wurde darauf verzichtet diese zwei Klassen weiter auf neue Klassen aufzuteilen, da die Kohäsionswerte im grünen Bereich liegen, sowie viele Methoden wie zum Beispiel die Lifecycle Methoden fast unumgänglich sind (siehe Abbildung 4 - Codebeispiel Lifecycle Methoden).

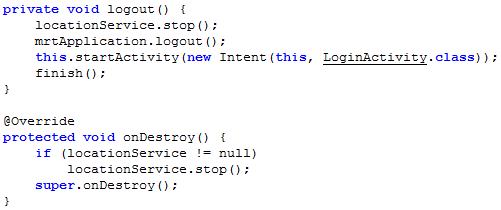


Abbildung - Codebeispiel Lifecycle Methoden

Im Falle der zweiten Klasse Customer ist der Hauptgrund für die hohe Komplexität der Einsatz von Settern bzw. Gettern für diverse Attribute.

#### Efferent Coupling

Der Efferent Coupling Wert beschreibt die Anzahl verwendeter Typen in einer Klasse. Dies kann ein Indikator für die Kopplung einer Klasse darstellen.

Die nachfolgende Abbildung 5 - Übersicht Efferent Couplings verdeutlicht, dass abgesehen von einer Klasse sämtliche Klassen im gewünschten Bereich von eins bis 25 liegen.

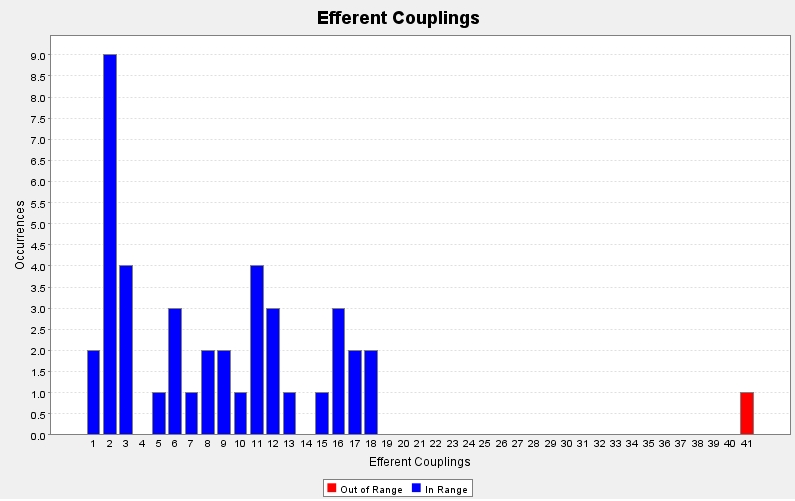


Abbildung - Übersicht Efferent Couplings

**Erklärung Hohe Kupplung Klasse TimeEntryActivity**

Die Klasse TimeEntryActivity ist für die Visualisierung der Hauptanzeige (User Interface) verantwortlich. Dementsprechend enthält diese Klasse eine grosse Anzahl referenzierter GUI Typen, diese sind in erster Linie der Grund für die hohe Kopplung (siehe Abbildung 6 - Ausschnitt Importliste Klasse TimeEntryActivity). Diese Anbindung an diese Komponenten ist jedoch nicht weiter tragisch, da es sich bei den GUI Typen um Klassen handelt, die im Android Framework integriert sind und stabil laufen.

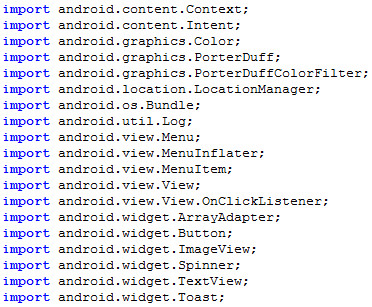


Abbildung - Ausschnitt Importliste Klasse TimeEntryActivity

### State 101

#### Übersicht

Bei der Entwicklung der Anwendung MRT wurde darauf geachtet, dass keine zirkuläre Abhängigkeiten zwischen den Packages bestehen. Wie aus der nachstehenden Abbildung 7 - Abhängigkeiten zwischen Packages ersichtlich wird, bestehen keine zirkulären Abhängigkeiten auf Packageebene.

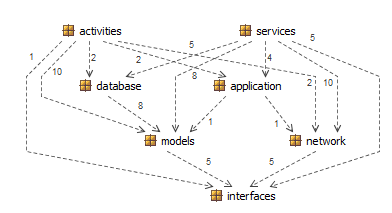


Abbildung - Abhängigkeiten zwischen Packages

#### Abhängigkeiten in Package Services

Das Package Services enthält die Klassen zur Synchronisation von Stundeneinträgen, Stundeneintragstypen sowie Kunden. Wie aus der Abbildung 8 - Abhängigkeiten in Package Services hervorgeht, existiert in diesem Packet eine zirkuläre Abhängigkeit zwischen den zwei Klassen SynchronizationServiceTask und SychnronizationService. Ansonsten verhalten sich die Abhängigkeiten hierarchisch, wobei die Klasse SynchronizationService die Wurzel darstellt.

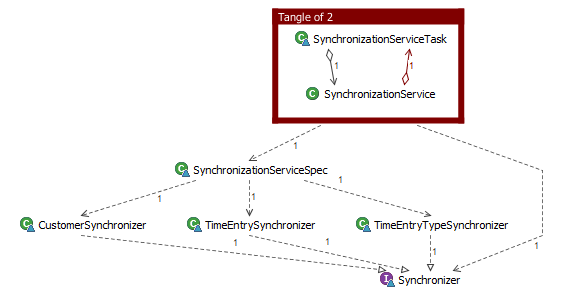


Abbildung - Abhängigkeiten in Package Services

**Erklärung zirkuläre Abhängigkeit**

Die Klasse SynchronizationService enthält die Lifecycle Methoden wie zum Beispiel onCreate(), onStart() etc., welche beim Starten des Dienstes aufgerufen werden. Diese Klasse instanziiert ein Objekt der SynchronisationServiceTask, damit die Klasse möglichst kohäsiv bleibt. Das neu erstellte Objekt benötigt jedoch eine Referenz auf den Service, da der Kontext bzw. die Datenbankhelper an den Service gebunden sind.

#### Abhängigkeiten in Package Activities

Das Package Activities enthält Klassen, welche für die Darstellung der Benutzeroberfläche zuständig sind. Anhand der nachstehenden Abbildung 9 - Abhängigkeiten in Package Activity wird eine zirkuläre Abhängigkeit sichtbar, die im folgenden Kapitel begründet wird.

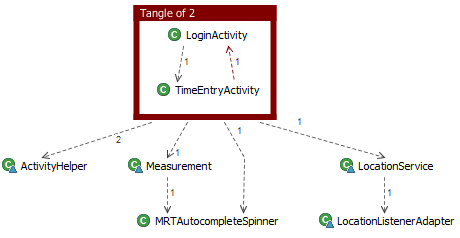


Abbildung - Abhängigkeiten in Package Activity

**Erklärung zirkuläre Abhängigkeit**

Diese zirkuläre Abhängigkeit ist aufgrund des Android Frameworks vorgegeben, da beim Aufruf einer neuen Activity der Context einer Applikation übergeben werden muss. Deshalb wird die Referenz auf das eigene Objekt aufgrund der Ableitung von Context übergeben.

#### Abhängigkeiten in Package Models

**Luke fragen zirkuläre Abhängigkeit aufgrund von TAG (ev. TAG falsch gesetzt in GpsPosition)??**

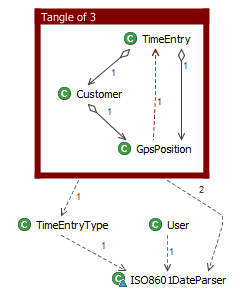


Abbildung - Abhängigkeiten in Package Models

#### Abhängigkeiten in Package Network

Im Package Network sind keine zirkulären Abhängigkeiten zwischen den Klassen vorhanden. Es herrscht eine strenge Top-Down Hierarchie.

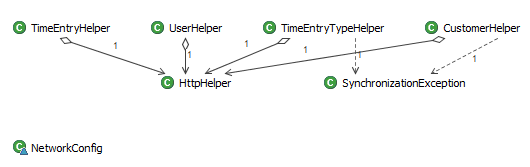


Abbildung - Abhängigkeiten in Package Network

### Schlussfolgerung

Wie aus den vorangehenden Kapiteln ersichtlich, sind bis auf wenige Ausnahmen keine zirkulären Abhängigkeiten in der Anwendung präsent. Zudem befinden sich die Codemetrik-Indikatoren, abgesehen von zwei Klassen, in einem erwarteten Bereich.

Aufgrund dieser Messungen werden keine weiteren Änderungen an der Architektur bzw. dem Code durchgeführt.