MODELLGETRIEBENE ENTWICKLUNG EINER MOBILEN APPLIKATION MIT JUSE4ANDROID

Jano Espenhahn, Tobias Franz and Franziska Krebs
Fachhochschule Brandenburg, Fachbereich Informatik und Medien
{*espenhah, franzt, krebsf*}*@fh-brandenburg.de*

Keywords: MDA, UML, USE, OCL, Android

Abriss: ein deutsches Abstract: Entwicklung einer Applikation, USE-Spezifikation zur Definition eines Klassendiagramms; OCL-Constraints oben drauf;Im Anschluss Generierung der Anwendung mit Hilfe von JUSE4Android. Untersuchung der Anwendung anhand bestimmter Fragestellungen

Abstract: ein englisches Abstract

1 EINLEITUNG

Zitat Test (**?**)

2 BESCHREIBUNG DER ANWENDUNG

Das Beispiel wurde aus dem Artikel (**?**) entnommen. Es handelt sich um ein fiktives Programm der Regierung zur Kontrolle der Eispartikel in der Luft. Wenn die Konzentration zu niedrig ist, bedeutet das, dass die Bevölkerung zu wenig Eiscreme isst, was eine Menge an Risiken für die Umwelt und die öffentliche Ordnung darstellt. Um die Eispartikel in der Luft zu überwachen, hat der Staat Kontrollstationen im gesamten Land verteilt aufgestellt. Für jede Station gibt es einen festgelegten Zielwert der Eispartikel. Der aktuelle Wert weicht in der Regel vom Zielwert ab.
Die Anwendung ermöglicht es neue Stationen mit Zielwerten aufzunehmen und alte Stationen zu löschen. Außerdem gibt es die Möglichkeit eine Adresse zu einer Station anzugeben. Eine Adresse ist im Nachhinein auch wieder entfernbar. Die Erfassung von beliebig vielen Einträgen zu einer Station ist ebenfalls möglich. Auch Einträge lassen sich im Nachhinein wieder entfernen. Zudem wird für jeden Eintrag, nach Eingabe des aktuellen Wertes die Abweichung zum Zielwert angezeigt.

3 VORSTELLUNG USE

UML based Specifiation Environment (USE) wird zur Spezifikation von Informationssystemen verwendet und wurde an der Universität Bremen entwickelt. Neben dem Einsatz für Fallstudien, wird USE vor allem in der Lehre an Hochschule wie z. B. MIT, Cambridge, University of Edinburgh und University of Lisbon eingesetzt. USE basiert auf einer Teilmenge der Unified Modeling Language (UML) und der Object Constraint Language (OCL). Eine USE-Spezifikation besteht aus einer textuellen Beschreibung eines Modells, bei der Eigenschaften aus UML-Diagramm verwendet werden. Weitere Integritätsausdrücke für ein Modell können durch die OCL definiert werden. (**?**)

Die Abbildung 1 veranschaulicht den Workflow für eine USE-Spezifikation. Ein Entwickler spezifiziert ein plattformunabhängiges USE-Modell, welches ein System beschreibt und nutzt dafür UML- und OCL-Ausdrücke. Mithilfe von USE ist es ihm möglich, die bestimmten Anforderungen an sein System auf Erfüllung mit dem Modell zu validieren.

3.1 Spezifikation

Die textuelle Beschreibung eines Modells mit USE beginnt immer mit der Definition eines Modell-Namens. In diesem Fall ist das *IceCream*. Im Anschluss folgen Klassendefinitionen mit ihren jeweiligen Attributen und Methoden. Im Beispiel hat die

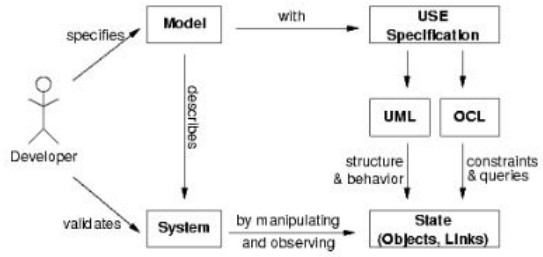

Abbildung 1: Workflow einer USE-Spezifikation (**?**)

Klasse *Station* das Attribut *name* und die Operation *entries* ohne Übergabeparameter. Die nachfolgenden Code-Ausschnitte verwenden lediglich UML.

Listing 1: USE-Spezifikation der Klasse Station im Modell IceCream

```
model IceCream

class Station
   attributes
      name         : String
      target       : Integer
      numberOfEntries : Integer
      meanActualValue : Real
      meanVarianceValue :Real
   operations
      entries()    : Set(Entry)
      calculateMeanActualValue() : Real
      calculateMeanVarianceValue() :
          Real

end
```

Klassen können untereinander in Abhängigkeit stehen. Für diese Abhängigkeiten sind Assoziationen vorgesehen. Um eine Assoziation auszudrücken, wird zuerst eine weitere Klasse *Address* eingeführt.

Listing 2: USE-Spezifikation der Klasse Adresse

```
class Address
   attributes
      street   : String
      postCode : Integer
end
```

Für das dem Artikel zugrunde liegende Beispiel kann eine Station entweder eine oder keine Adresse haben.

Listing 3: USE-Spezifikation der Assoziation zwischen einer Station und einer Adresse

```
association Station_Address between
   Station[ 1 ]
   Address[ 0..1 ] role place
end
```

Station_Address ist dabei der Name der Assoziation und das Attribut *place* nimmt in der Klasse *Station* die Rolle für die Adresse ein. Zum gesamten USE-Modell gehören weiterhin noch die Klasse *Entry* und die Assoziation *Station_Entry*.

Listing 4: USE-Spezifikation der Klasse Entry und der Assoziation zwischen einer Station und deren Entries

```
class Entry
   attributes
      date     : CalendarDate
      actual   : Integer
      variance : Integer
   operations
      calculateVariance(): Integer
end

association Station_Entry between
   Station[ 1 ] role station
   Entry[ * ] role records
end
```

Zur Vervollständigung des Modells gehört außerdem eine aus der Arbeit (**?**) entnommene Klasse *CalendarDate*.

3.2 Erweiterung durch OCL

Als Bestandteil der UML ist die OCL ebenfalls als Spezifikation zur Modellierung von Softwareartefakten zu verstehen. Die Entwicklung der OCL wurde angetrieben durch den Wunsch, zusätzliche Modelleigenschaften - welche nicht mithilfe grafischer Elemente ausgedrückt werden können - festlegen zu können. (**?**, S.5f) Da diese Aspekte eindeutig und für alle Akteure verständlich sein sollen, wurde die OCL als eine formale und dennoch gut lesbare Sprache konzipiert. Das Vokabular der aktuell in Version 2.4. bereitgestellten Spezifikation ist sehr umfänglich und wird u.a. für die folgenden Zwecke genutzt:

- zur Definition von Restriktionen für Operationen
- zur Beschreibung von Vor- und Nachbedingungen von Operationen
- zur Definition von Invarianten
- zur Definition von Ableitungsregeln für Attribute

Im Folgenden werden die OCL-Konstrukte erläutert, welche zur textuellen Beschreibung von Bedingungen im IceCream Beispiel verwendet wurden. Da diese innerhalb der Klassen spezifiziert werden, kann auf die Einleitung durch das Schlüsselwort *context* verzichtet werden.
Zunächst werden die in den Klassen Station und Entry deklarierten Operationen näher durch die OCL definiert und es werden Vorbedingungen festgelegt. Listing 5 zeigt, wie die Einträge zu einer Station durch die Methode *entries()* gesammelt werden und wie einfache Berechnung des Mittelwerts für gemessene Werte (*calculateMeanActualValue()* erfolgt.

Listing 5: OCL-erweiterte Operationen der Klasse Station

```
entries(): Set(Entry) = self.records->
    asSet

calculateMeanActualValue() : Real =
    entries()->iterate(iterator : Entry
    ; result : Real = 0 | result +
    iterator.actual)/(numberOfEntries)
   pre:numberOfEntries>0
```

Das Schlüsselwort *self* wird genutzt, um auf eine Instanz der Klasse Bezug zu nehmen. Die Einträge eines Objektes der Klasse Station werden über den Rollennamen *records* referenziert. Die Kollektionsoperatoren *asSet* und *iterate* überführen die Einträge in eine Menge bzw. iterieren über diese, um die jeweiligen gemessenen Werte zu addieren. Um sicherzustellen, dass keine Teilung durch Null erfolgt, wird innerhalb einer Vorbedingung (eingeleitet durch *pre*) festgelegt, dass mindestens ein Eintrag vorhanden sein muss.

Für die Klassen Station und Entry werden die Invarianten *TargetValueCannotBeNegative* (siehe Listing 6), *ActualValueCannotBeNegative* und *SelectedDateCannotBeInTheFuture* (siehe Listing 7) definiert. Diese repräsentieren Aussagen, welche für die Instanzen der jeweiligen Klasse zu jeder Zeit wahr sein müssen. (**?**, S.188)

Listing 6: Invariante in der Klasse Station

```
inv TargetValueCannotBeNegative:
   target>=0
```

Listing 7: Invarianten in der Klasse Entry

```
inv ActualValueCannotBeNegative:
   actual>=0
inv SelectedDateCannotBeInTheFuture:
   date.isBefore(date.today()) or date.
       isEqual(date.today())
```

Nach dem Schlüsselwort *inv* folgt der Bezeichner der Invariante und anschließend der OCL-Ausdruck. Auf diese Art und Weise kann formuliert werden, dass der Zielwert einer Station (*target*) sowie der tatsächlich gemessene Wert (*actual*) stets im positiven Zahlenbereich liegen sollen. Ebenfalls kann spezifiziert werden, dass ein Eintrag niemals in der Zukunft vorgenommen werden kann. Da OCL Sichtbarkeiten ignoriert, können problemlos Zugriffe auf Methoden anderer Klassen oder deren Objekte (in Listing 7 beispielsweise die Methoden der Klasse CalendarDate) definiert werden. (**?**, S.71)
Mit Hilfe des *derive*-Konstruktes können Ableitungen für Attribute formalisiert werden. Listing 8 zeigt, wie sich die Attributwerte *numberOfEntries* und *meanActualValue* entweder aus dem Rückgabewert einer Methode oder durch darauf angewandte Operationen ergeben.

Listing 8: Abgeleitete Attribute der Klasse Station

```
numberOfEntries : Integer derive:
    entries()->size()
meanActualValue : Real derive:
    calculateMeanActualValue()
```

Die vollständige USE-Spezifikation kann in Anhang REFERENZ eingesehen werden.

3.3 USE-Tool

Um eine Spezifikation auf nicht-formale Anforderungen zu validieren, kann ein Modell mithilfe des USE-Tools animiert werden. Direkt nach dem Import eines Modells erhält man vom Tool ein Feedback über die Validität der UML- und OCL-Definitionen. Neben der Validierung bietet das Tool weitere Möglichkeiten, wie z. B. die Visualisierung eines Klassen-, Sequenz- oder Objektdiagramms. In der Abbildung 2 finden sich die im Kapitel 3.1 definierten Klassen und Assoziationen als Klassendiagramm wieder.

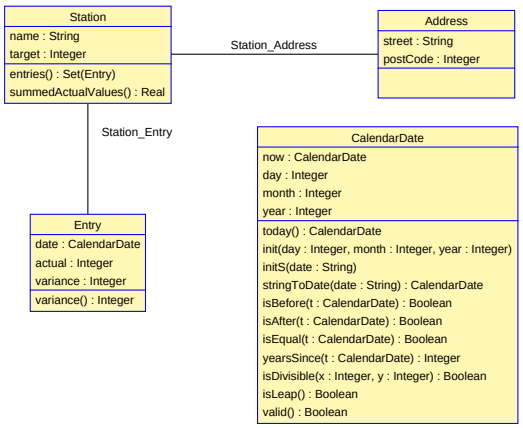

Abbildung 2: Klassendiagramm für das Beispiel

4 JUSE4ANDROID

Dieser Abschnitt befasst sich mit der Anwendung Juse4Android und untersucht die erstellte Anwendung, sowie dessen Quellcode.

4.1 Vorstellung

Das Tool Juse4Android ist im Rahmen der Master-Arbeit *Model-Driven Generative Programming for BIS Mobile Applications* von Luís Miguel Pires Teixeira da Silva entstanden und ermöglicht die Erstellung eine Mobilen BIS(Business Information System)

Anwendung für das Betriebssystem Android aus einer Use-Datei. Zum persistieren von Daten wird das Objekt-Orientierte Datenbank-Framework *db40* verwendet. Dieses wird nur bis zur Java-Version 1.6 unterstützt stellt dementsprechend spezielle Anforderungen an das *Android Software Development Kit*. Konkret bedeutet das, dass die generierte BIS-Apps nur bis zum Android API-Level 17 lauffähig sind und mit der Betriebssystemversion Android 4.2 (*Jelly Bean*) betrieben werden müssen.

To Do: Bremen Tool sagt, das Modell sei valide; JUSE4Anroid: keine Generierung aufgrund der Definition von abgeleiteten Attributen

4.2 Untersuchung der erstellten Applikation

Die generierte Anwendung wird durch die Folgenden Schichten definiert: **Persistenz-Schicht, Präsentations-Schicht, Model-Schicht(Business Logik)**. Wie eine tiefere Analyse des Codes zeigte, wurden die Funktionen der Schichten nicht durchgängig eingehalten, was zu Brüchen der Schichtenarchitektur führt. So sind die eigentlichen Datenhaltungsobjekte (POJOsPOJO = Plain Old Java Object) selbst für das Speichern und Löschen ihrer Daten aus der Datenbank verantwortlich und greifen dafür auf entsprechende Zugriffsobjekte zu. Bei einem lesenden Zugriff wird gänzlich auf die Delegierung an Zugriffsobjekte verzichtet und die Datenbank direkt angesprochen. Das einbringen einer zusätzlichen Datenzugriffsschicht, sowie die Verwendung des Fabrikmethoden-Entwurfsmuster zur Erstellung der POJOs würde die Lesbarkeit deutlich erhöhen. **Ein entwickler hätte es anders gemacht???** Laut des Entwicklers wird konsequent das Model-View-View-Model Entwurfsmuster verwendet. Jedoch ist dies bei genauer Betrachtung nicht komplett Richtig. Zwar wird dies Business-Logik und die View getrennt, jedoch werden die *Activity Klassen*[1] als ViewModel verwendet. Diese beherbergen allerdings die Interaktionslogik für die korrespondierenden XML-Oberflächen Definitionen, genannt Layouts. Somit sind die, vom Entwickler gewählten Viewmodels teil der Presentationsschicht, was im engeren sinne, nicht in die MVVM Definition passt. Dennoch wird eine lose Kopplung zwischen der eigentlichen Oberfläche und dem Model geschaffen, welche die Möglichkeit eröffnet, teile der Oberfläche zu ändern oder auszutauschen.

[1] Kssen, die vom Typ Activity erben, Übernehmen die Interaktionslogik für die Oberfläche und sind vergleichbar mit den sog. CodeBehind Klassen, der xaml-Oberflächendefinitionen in Microsofts WPF-Framework

Das vorherrschende Architekturmuster ist das Naked-Object-Pattern. Dieses definiert 3 Prinzipien. 1.) Die gesamte Geschäftslogik wird in Domänen Objekten gekapselt, 2.) Die Benutzeroberfläche ist eine direkte Repräsentation dieser Objekte und 3.) die Benutzeroberfläche kann oder wird direkt aus der Definition dieser Objekte erstellt. Diese 3 Prinzipien, werden in JUSE4Android vollständig umgesetzt. Zudem werden die Domänen Objekte direkt in eine Objekt Orientierte Datenbank gespeichert. Die Abbildung 3 verdeutlicht die Grundarchitektur des Naked-Objects-Patterns.
Der Programmfluss wird über die 4 grundlegenden Datenbankoperationen **C**reate **R**ead **U**pdate **D**elete gesteuert. Dabei werden neue Objekte sowohl in der Datenbank als auch der View verändert oder neu erstellt.
Die generierten Oberflächen, verwenden das*Master/Detail Flow* Layout, welches in der Lage ist eine Liste von Items auf dem sogenannten *Master* anzuzeigen. Und bei Berührung eines Items in die sogenannten *Detail* Ansicht umschaltet, in der Daten zu dem dazugehörigen Item präsentiert werden. Dieses Layout ermöglicht eine Anpassung der Oberfläche an verschiedene Display-Größen.
In der generierten Anwendung sind die Oberflächen eine Eins-zu-eins Repräsentation der Business-Objekte, so wie es das Naked-Object-Pattern fordert.

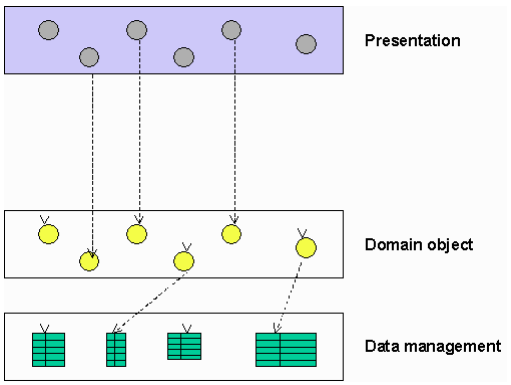

Abbildung 3: Richard Pawson *„Naked objects“*; University of Dublin, Trinity College; 2004; Seite 24)

4.2.1 Abbildung von Assoziationen und OCL-Constraints

Wie bilden sich Assoziationen ab? Wie bilden sich Constraints ab?

4.2.2 Generierter Code und zusätzliche Funktionalität

Der aus der USE-Definition generierte Code bedarf in den meisten Fällen einer manuellen Nachbearbeitung, um syntaktische Fehler der Generierung zu beheben. Diese äußern sich u.a. durch den Aufruf nicht deklarierter Variablen oder durch eine fehlerhafte Kommasetzung in Parameterlisten. Ebenfalls zeigten sich Probleme bei der Konfiguration des Build Paths in Form von nicht hinzugefügten Libraries.
Die Anwendung ist in aussagekräftigen Paketstruktur gegliedert, wie in Abbildung 4 zusehen ist. Außerdem hält sie die gängigen Code-Konventionen, wie beispielsweise die *camelcase* Schreibweise ein. Dabei bilden die Getter-Methoden eine Ausnahme, diese besitzen nicht das Get-Präfix. Negativ muss ebenfalls erwähnt werden, dass der Code nur geringfügig kommentiert wurde und teilweise verschiedene Sprachen für die Kommentare verwendet wurden. Alles in allem ist der Code für einen Fortgeschrittenen Java/Android Entwickler nachvollziehbar und verständlich. Nur nach der Generierung sind manuelle Anpassungen oder das Hinzufügen komplexerer Funktionalität möglich. Diese werden dementsprechend bei einer erneuten Generierung wieder überschrieben. Ebenfalls ist keine Möglichkeit vorgesehen, geschützte Bereiche zu definieren. Auch das Auslagern von manuellem Code in separate Dateien kann nicht vorgenommen werden, da der gesamte Code, mit Ausnahme von wenigen vordefinierten Standardklassen dynamisch durch die Applikation generiert wird. Somit ist eine Trennung von generiertem Code und manuellem Code nicht möglich.

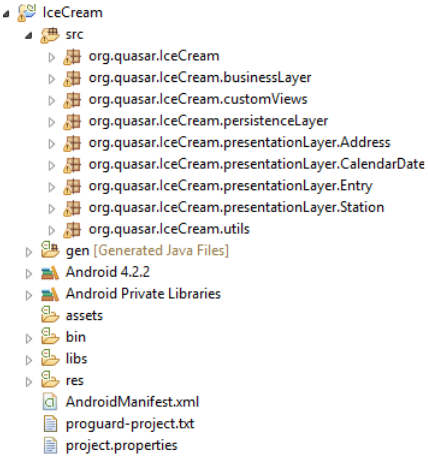

Abbildung 4: Paketstruktur einer von Juse4Android generierten App

4.2.3 Veränderungen der Benutzeroberfläche

Wie bereits in Abschnitt 4.2 beschrieben, ist das verwendetet Entwurfsmuster MVVM nicht vollständig erfüllt, dennoch wurde festgestellt, dass es eine lose Kopplung zwischen der Business-Schicht und der Präsentations-Schicht gibt. Damit ist der Austausch vereinzelter Oberflächenkomponenten gewährleistet. Die Anpassung des Designs ist schon durch die Gestaltungsmechanismen des Betriebssystems Android ohne Probleme möglich. Ebenfalls können verwendete Grafiken einfach ausgetauscht werden. Theoretisch wäre der Austausch der gesamten Oberfläche auch möglich, jedoch entsteht dort ein Mehraufwand durch die nicht konsequente Verwendung des MVVM Architekturmusters. Bei der Auswechselung der Oberfläche müssten zum einen die die definierten in XML definierten Layout-Datein verändert oder ausgetauscht werden, zum anderen müssten auch die *Activity-Klassen* mindestens angepasst werden. Durch die Verwendung des Master/Detail Flow Layouts, gehören zu jeder Oberflächenrepräsentation eines Objektes drei Klassen, welche die Anzeige steuern. Somit ist es nicht auszuschließen, dass es zu Inkonsistenzen im Programmfluss kommen könnte. Das Adaptieren einer neuen Oberflächengestaltung für eine durch Juse4Android generierte Applikation, ist demzufolge mit einem erheblichen Aufwand verbunden.
Im Rahmen dieser Arbeit wurde exemplarisch gezeigt, wie die Anpassung der Oberfläche möglich ist. Dabei wurde die Darstellung für die Darstellung der Varianz leicht verändert. Wenn diese Außerhalb des tolerablen Bereichs ist, wird die das angezeigte Feld Rot eingefärbt, innerhalb des tolerablen Bereiches erscheint sie grün. Um dies umzusetzen bedarf es einen manuellen Eingriffes in die *EntryDetailFragment-Klasse*. Es wurde lediglich die Varianz mit dem Grenzwert verglichen und dann die Änderung der Textfarbe der Anzeigekomponente vorgenommen. Die Anpassung des ViewModel-Codes für die Umsetzung dieses Beispiels, zeigt ebenfalls, dass hier das MVVM-Muster verletzt wurde.

JUSE4Android bietet Annotationen an, welche im USE-File verankert sind und durch welche geringfügig auf die View Einfluss genommen werden kann. SIEHE MASTERARBEIT Seite

5 Zusammenfassung und Fazit

JUSE4Anroid zu starr; möglicherweise geht es in die Richtung der Generierung von Softwarekomponenten, im Gegensatz zur Generierung einer

vollständigen Applikation

Da somit eine Menge moderne Mobilen Geräte mit den generierten BIS-Apps kompatibel sind, ist dies als erheblicher Nachteil ein zu schätzen.

ANHANG

USE-Spezifikation, welche zur Validierung durch das USE-Tool genutzt wird

Diese weicht ab von der tatsächlich zur Generierung verwendeten USE-Spezifikation Annotationen sind JUSE4Android spezifisch und nicht Bestandteil der OCL $\backslash section*\{APPENDIX\}$