ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Implementation of a Multi-Touch, Gesture-based Process Modeling Component for Apple iPad

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Matthias Dapper
matthias.dapper@gmx.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Jens Kolb

2012

Abstract

In dieser Bachelorarbeit geht es um die Implementierung eines mobilen *proView-Clients* in Form einer iOS-App für das Apple iPad. Die Entstehung der im Rahmen dieser Arbeit zu entwickelnden App wird anhand der ihr zu Grunde liegenden Konzepte sowie der umzusetzenden Anforderungen beschrieben, erläutert und begründet. Neben einer ausführlichen Darstellung der einzelnen Entwicklungsschritte beinhaltet die Arbeit eine detaillierte Betrachtung einzelner Module und Klassen. Abschließend ist eine kritische Bewertung der Arbeit, vor allem aber bezüglich einer im Laufe der Entwicklung aufgetretenen Hürde, enthalten.

Inhaltsverzeichnis

1. Einleitung

Prozessmanagementsysteme sind in vielen Firmen, Ämtern und Bildungseinrichtungen ein unverzichtbarer Bestandteil der informationstechnischen Infrastruktur. Sie bilden nicht nur reale Arbeitsabläufe ab, sondern schaffen durch ihre Komplexität und Adaptierbarkeit neue Möglichkeiten, Daten gezielter und effektiver zu verarbeiten. Um Kosten und Zeit zu sparen, wird die Fähigkeit, Prozesse zu erfassen und zu implementieren, immer öfter diesbezüglich wenig geschulten oder gar fachfremden Mitarbeitern abverlangt.

Dabei offenbaren sich hinter den Potentialen die Fallstricke dieser Technologien. Denn neben der Sicherstellung der strukturellen Korrektheit eines Prozesses, welche von vielen gängigen Systemen nicht in ausreichendem Maße bereitgestellt wird, ist auch die mangelnde Übersichtlichkeit großer Prozesse ein Aspekt, der bei der Prozessmodellierung und -anpassung eine wichtige Rolle spielt.

Mit dem proView-Projekt [3, 4, 5, 6, 7, 8, 9, 10] werden diese beiden Aspekte adressiert. So kann bei der Modellierung von Prozessgraphen bereits von Anfang an, sowie mit jeder Manipulation, deren strukturelle Korrektheit gewährleistet werden. Gleichzeitig bietet es die Möglichkeit, verschiedene Sichten auf einen Prozess zu erstellen, die sich jeweils nur auf den Ausschnitt beschränken, der für eine bestimmte Aufgabe oder einen entsprechenden Zuständigkeitsbereich relevant ist.

In dieser Arbeit soll eine Implementierung der mobilen Benutzerschnittstelle zum *proView-Server* vorgestellt werden. Diese Implementierung soll neben der Bereitstellung der entsprechenden Manipulationsoperationen zusätzlich die komfortablen Eigenschaften des Apple iPads ausnutzen, um eine „intuitive“ Gestensteuerung zu ermöglichen.

In Kapitel 2 werden grundlegende Konzepte eingeführt. Dazu gehören die Begriffe *Prozessgraph* und *Gestensteuerung* sowie das *proView-Projekt* und *Entwicklungsparadigmen in iOS.* Kapitel 3 beinhaltet die Anforderungen an die zu entwickelnde App und eine Beschreibung der entsprechenden Vorgehensweisen während der Entwicklung. Es ist in drei Punkte gegliedert: die Darstellung des Prozessgraphen, die Interaktion mit dem proView-Server und die Manipulation des Prozessgraphen. Erläuterungen zu Klassenbeziehungen und -hierarchien sowie eine Übersicht über die Programm-Module finden sich in Kapitel 4. Kapitel 5 beschreibt während der Entwicklung aufgetretene

Probleme und enthält außerdem eine kritische Bewertung. Im sechsten und abschließenden Kapitel wird die Arbeit noch einmal zusammengefasst.

2. Grundlegende Konzepte

Zu Beginn müssen für das weitere Verständnis die Begriffe *Prozessgraph* (siehe Kapitel 2.1), *proView* (siehe Kapitel 2.2) und *Gestensteuerung* (siehe Kapitel 2.5) als zentrale Teile dieser Arbeit erläutert und abgegrenzt werden. Außerdem werden in Kapitel 2.3 wichtige Entwicklungsparadigmen in iOS vorgestellt. Kapitel 2.4 beschreibt die in dieser Arbeit verwendeten *hybriden Datenstrukturen*, ein Konzept zur Wiederverwendung geeigneter Klassen in Model und View.

2.1 Prozessgraph

Die Begriffe Geschäftsprozess, Prozessmodell, Prozessgraph oder Workflow werden häufig synonym verwendet. Sie bezeichnen Konzepte, die meist bereits informationstechnische Lösungen und praxisbezogene Arbeitsweisen im unternehmerischen Umfeld umfassen. Um Verwechslungen zu vermeiden, wird in dieser Arbeit nur der Begriff Prozessgraph – wie im Folgenden beschrieben – verwendet.

Ein *Prozessgraph* ist ein gerichteter Graph, der aus einer Menge von *Knoten* und einer Menge von *Kanten* besteht. Er besitzt genau einen *Startknoten*, welcher keine eingehende, und genau einen *Endknoten*, welcher keine ausgehende Kante besitzt. Er kann außerdem *Aktivitäts-*, *Verzweigungs-* und *Datenelementknoten* enthalten. *Verzweigungsknoten* können vom Typ *AND* oder *XOR* sein. *Datenelementknoten* werden nach ihrem *Datentyp* unterschieden. Alle Knoten können einen Namen besitzen.
Weiter wird die Menge der Kanten unterteilt in *Sequenzfluss-* und *Datenflusskanten*. Während *Sequenzflusskanten* ausschließlich Start-, End-, Aktivitäts- und Verzweigungsknoten untereinander verbinden, verbinden *Datenflusskanten* einen Aktivitäts- oder Verzweigungsknoten mit einem Datenelementknoten. Datenflusskanten können dabei entweder vom Typ *lesend* oder *schreibend* sein.

Abbildung 1 zeigt einen Prozessgraphen.

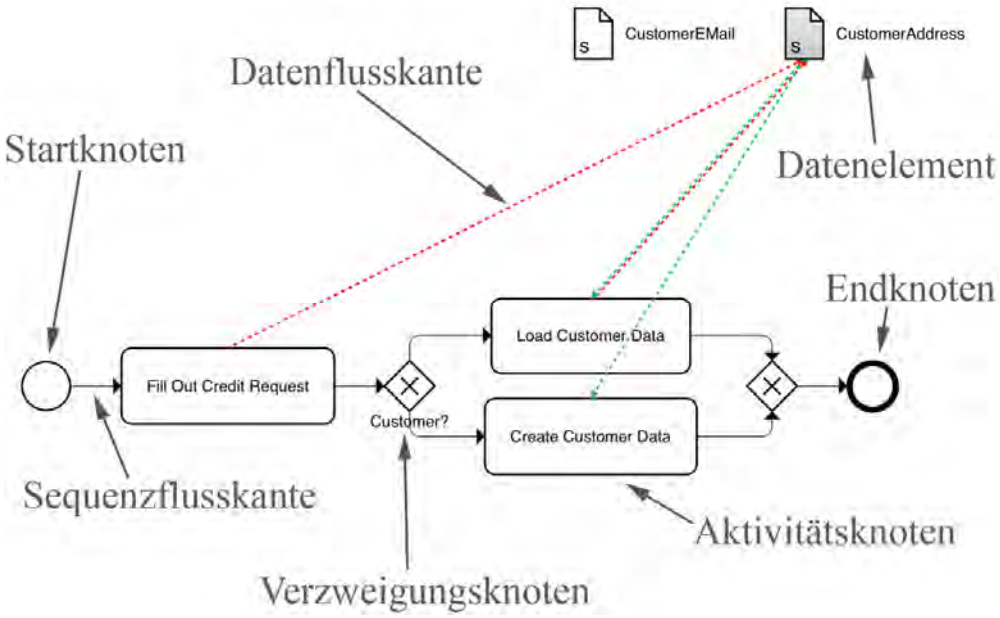

Abbildung 1: Prozessgraph

Ein Verzweigungsknoten tritt immer in Verbindung mit einem weiteren Verzweigungsknoten auf. Er besitzt genauso viele ausgehende Kanten, wie sein korrespondierender Verzweigungsknoten eingehende Kanten besitzt. Dadurch bilden beide Verzweigungsknoten eine Blockstruktur.

Wird ein späterer (ausgehend vom Startknoten) Verzweigungsknoten durch eine Sequenzflusskante mit seinem korrespondierenden, vorausgehenden Verzweigungsknoten verbunden, entsteht ein Loop (siehe Abbildung 2).

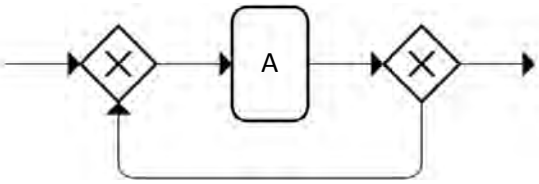

Abbildung 2: Loop

2.2 proView

Im proView-Projekt geht es um die Entwicklung eines Frameworks zur Verwaltung von Prozessmodellen, das die Erstellung unterschiedlicher Sichten auf ein und dasselbe Prozessmodell ermöglicht. [2]

2.2.1 Client-Server-Architektur

Das proView-Framework ist als Client-Server-System konzipiert. So beherbergt der *proView-Server* die komplette Programmlogik sowie alle Daten zu den verwalteten Prozessgraphen und bietet einheitliche Schnittstellen für verschiedene Client-Anwendungen an. Diese Schnittstellen umfassen neben der Bereitstellung der Informationen über die vorhandenen Prozessgraphen Operationen zum Lesen, Schreiben, Löschen und Erzeugen von *Templates* oder deren Elementen. Als *Templates* werden in diesem Zusammenhang einzelne *Ressourcen* bezeichnet, die einen Prozessgraphen oder eine bestimmte Sicht auf einen Prozessgraphen beschreiben. Sie werden in Kapitel 2.2.2 und 2.2.3 genauer beschrieben. Alle relevanten Daten können über eine REST-Schnittstelle abgefragt werden. Als Datenrepräsentationsformat wird dabei XML verwendet.

Abbildung 3 zeigt beispielsweise einen Request für eine Löschoperation.

```
<Request>
  <requestOperation>DELETE</requestOperation>
  <resourceID>882d35ec-d50f-41a9-bc20-5ee4d20466da</resourceID>
  <payloadFormat>XML</payloadFormat>
  <resourceType>VIEW</resourceType>
</Request>
```

Abbildung 3: Request für eine Löschoperation

Ein *proView-Client* hat im Wesentlichen die Aufgabe, die vom Server bereitgestellten Daten zu visualisieren und dem Benutzer zur Interaktion anzubieten. Er generiert anhand der Benutzereingabe entsprechende Lese- oder Änderungsanfragen an den *proView-Server* und zeigt die entsprechenden Resultate an.

2.2.2 Central Process Model

Das *Central Process Model* – kurz *CPM* – ist ein zentraler Bestandteil des *proView-Frameworks* [7, 10]. Es beinhaltet alle Informationen, die zur Repräsentation eines kompletten Prozessgraphen gehören und bietet dadurch die Grundlage für die Erstellung von Sichten (*Views*), welche in Kapitel 2.2.3 erklärt werden. Das CPM enthält Daten zu allen Knoten und Kanten und beinhaltet gegebenenfalls Programm-Logik, die zur Ausführung eines Geschäftsprozesses notwendig ist.

2.2.3 View

Eine *View* (deutsch: „Sicht“) ist neben dem CPM der zweite Template-Typ, der in proView verwendet wird. Die Grundidee dabei ist, dass sie nur einen Teil des kompletten Prozessgraphen beinhaltet bzw. beim entsprechenden Benutzer oder in dessen Bezugsrahmen sichtbar macht. Sie bezieht sich dabei aber trotzdem weiterhin auf ihr zugehöriges CPM. Das bedeutet, dass Änderungen, die innerhalb einer View vorgenommen werden, Auswirkungen auf das CPM haben können. In diesem Zusammenhang kann man zweierlei Arten von Operationen an Views unterscheiden:

Zur ersten Art, der sogenannten *Simplifizierungsoperationen*, gehören *Reduktion* und *Aggregation*. Eine Reduktion blendet einen Knoten innerhalb des Prozessgraphen der jeweiligen View aus. Dabei wird der Knoten im zu Grunde liegenden CPM jedoch nicht verändert oder gelöscht. Das Gleiche gilt für die Aggregation, bei der eine Menge von Knoten zu einem einzigen Knoten zusammengefasst wird (siehe Abbildung 4).

Neben den Simplifizierungsoperationen gibt es noch sogenannte *Update-Operationen*. Dazu gehören zum Beispiel das *Einfügen* und das *Löschen* von Knoten im Prozessgraphen einer View. Diese Operationen werden zwar auf einer View durchgeführt, haben aber gleichzeitig Auswirkungen auf das entsprechende CPM und dadurch auch folglich auf alle anderen Views, sofern diese die betroffene Region im Graphen nicht ohnehin verbergen oder vereinfachen. Ist eine View erst einmal erstellt, sind diese Operationen häufiger, da sie ein wichtiger Bestandteil der Weiterentwicklung eines Prozessgraphen sind. Gleichzeitig bewirken sie natürlich weitaus gravierendere Veränderungen der Daten [7].

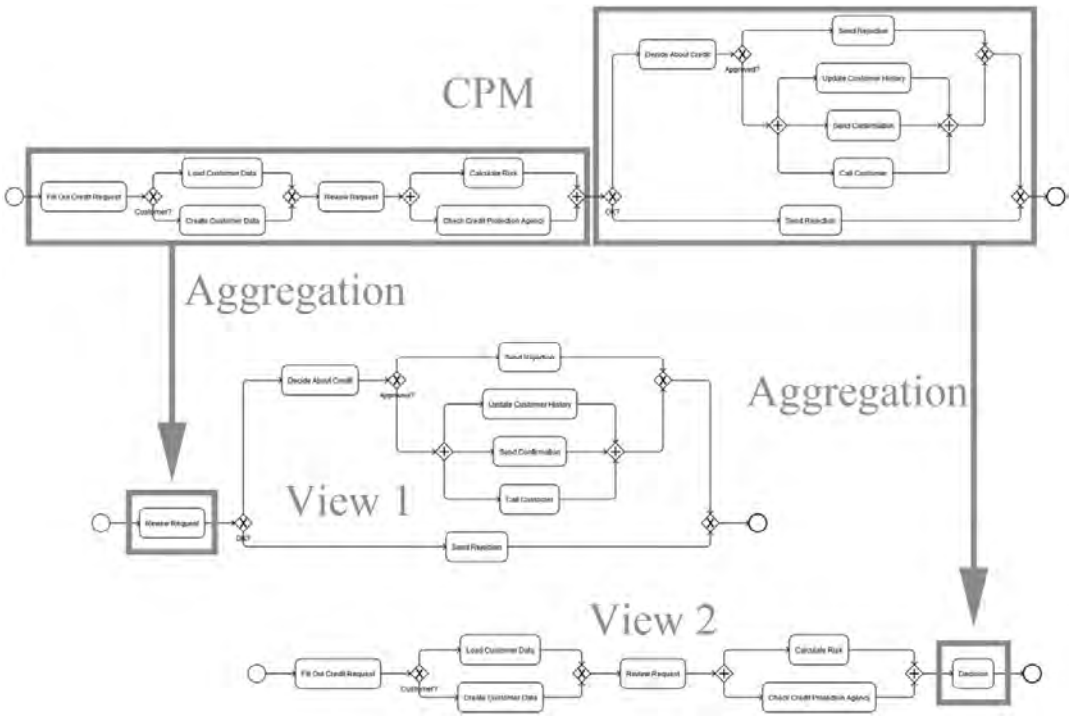

Abbildung 4: Zwei Views, verändert durch Aggregation

2.3 Wichtige Entwicklungsparadigmen in iOS

Das *iOS* ist ein Betriebssystem für mobile Endgeräte der Firma Apple. Wer Software für Apple-Produkte entwickeln möchte, muss zunächst einmal die Programmiersprache *Objective-C* beherrschen. Zusätzlich ist es aber auch erforderlich, einige Konzepte zu verstehen, die im Rahmen der Entwicklung für iOS häufig Verwendung finden. In den Kapiteln 2.3.1 und 2.3.2 werden dafür das *Model-View-Controller-Pattern* bzw. die Entwicklungsmuster *Delegates* und *Protokolle* vorgestellt.

2.3.1 Model-View-Controller-Pattern

„You can't really describe the development platform without including this design strategy which is (…) Model-View-Controller.“ [1]

Das *Model-View-Controller-Pattern* ist ein Entwicklungsmuster, das eine sehr strikte Trennung von Zuständigkeiten im Programmcode einer zu entwickelnden Software festlegt. Demnach gibt es in der Software oder einem Software-Modul jeweils eine gesonderte Schicht zur Darstellung (die *View*, nicht zu verwechseln mit der View aus Kapitel 2.2.3), zur Datenhaltung (das *Model*, nicht zu verwechseln mit dem Central Process Model aus Kapitel 2.2.2) und zur Steuerung der beiden Bereiche (den *Controller*).

Abbildung 5 zeigt bildhaft die Trennung der drei Schichten, die man abgekürzt *MVC* nennt. Dabei zeigen die gestrichelten Linien an, ob eine Kommunikation zweier Schichten in die entsprechende Richtung möglich ist, ähnlich einer Fahrbahnmarkierung mit einseitigem Spurwechselverbot. Nur der Controller hat Zugriff auf Model und View, welche ihrerseits nie direkt miteinander kommunizieren.

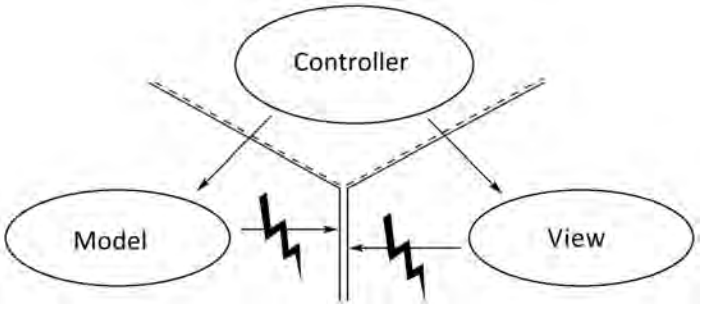

Abbildung 5: Model-View-Controller-Pattern (vgl. [1])

In den Frameworks zur Softwareentwicklung für iOS sind bereits viele Komponenten nach diesem Muster strukturiert. Besonders im Bereich der *Widgets* (grafische Bedienelemente) spielt der sogenannte *View Controller* eine bedeutende Rolle. Ein View Controller enthält bereits die beiden Schichten View und Controller und kann bei Bedarf mit einer sogenannten *Data-Source*-Eigenschaft ausgestattet werden, um das MVC-Pattern zu vervollständigen. Die *Data Source* übernimmt dann entsprechend die Aufgabe der Datenhaltung und bildet damit die Model-Schicht. Die View-Komponente ist für Zeichenoperationen bzw. die Darstellung der Software auf dem Display verantwortlich. Neben den in den Frameworks vorhandenen Views können eigene Views abgeleitet werden. Alle in dieser Arbeit verwendeten grafischen Bestandteile des Prozessgraphen sind eigens entwickelte Views, abgeleitet von der Superklasse *UIView*, welche die generischste Widget-Klasse im Entwicklungs-Framework von iOS ist.

Views werden über Controller gesteuert und mit für die Anzeige am Display relevanten Daten versorgt. Entscheidend dabei ist, dass die Verbindung im Allgemeinen nur in eine Richtung verläuft. So „kennt“ lediglich der Controller die View, aber nicht umgekehrt (siehe Abbildung 5).

Das gleiche Prinzip gilt auch für das Model. Es enthält im Wesentlichen die „puren“ Daten bzw. Datenverbindungen und keine Informationen über die Art der Darstellung. Auch hier ist der Controller für den Datentransport und die Manipulation von Daten verantwortlich. Ein Model kennt in der Regel „nur sich selbst“.

So übernimmt der Controller die Rolle des Vermittlers zwischen den anderen beiden Schichten. Ein neues *Xcode*-Projekt (Xcode ist die Standard-Entwicklungsumgebung für Apple-Betriebssysteme) verfügt in der Regel über einen zentralen View Controller, der von sich aus eine generische View bereitstellt. Der Einstiegspunkt für alle weiteren Aufrufe ist dessen Methode *viewDidLoad*.

Ein Beispiel für eine vollständige Implementierung des MVC-Patterns im *UI-Framework* (Sammlung von Klassen für die Entwicklung von User Interfaces, kurz UIs) von iOS ist der *UITableViewController*. Er enthält eine sogenannte *UITableView*, welche Daten tabellen- bzw. listenartig zur Anzeige bringt. Dabei übernimmt der Controller bereits wichtige Caching-Operationen, um auch mit großen Datenmengen performant umgehen zu können (vgl. dazu größere Listen von Alben und Songs im Apple iPod). Bei der Verwendung der *Data Source* kommt dabei ein weiteres wichtiges Entwurfsprinzip dazu: das der Delegates und Protokolle. Diese werden in Kapitel 2.3.2 behandelt.

2.3.2 Delegates und Protokolle

Delegates sind Objekte, die von anderen Objekten referenziert werden, um deren Methoden zu verwenden. Die Referenzierung kann auch zur Laufzeit geschehen. Daher ist es notwendig, eine gemeinsame Schnittstelle bzw. ein gemeinsames Protokoll zu finden, das von beiden beteiligten Klassen eingehalten wird. Solche *Protokolle* definieren, welche Eigenschaften oder Methoden von einer bestimmten Klasse implementiert werden können oder müssen.

Abbildung 6 zeigt eine gekürzte Version des UITableViewDelegate-Protokolls, das von der Beispiel-Klasse „SomeClass“ implementiert wird.

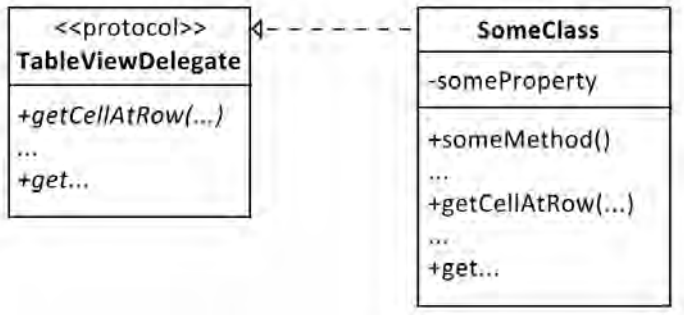

Abbildung 6: Protokoll und implementierende Klasse

Zum besseren Verständnis des Zusammenspiels der einzelnen Klassen und Protokolle zeigen die Abbildungen 7 bis 10 kleine Code-Fragmente aus *proViewMobile*.

Die Klasse *PVActionsMenuViewController* hat die Aufgabe, ein Menü darzustellen, mit dem Operationen an Templates durchgeführt werden können. Der Controller ist selbst nicht in der Lage, die tatsächliche Operation selbst durchzuführen, da er zur View (MVC) des Action-Menüs gehört und „nur“ für dessen Darstellung zuständig ist. Daher benötigt er einen „Ansprechpartner“ einer anderen Controllerklasse, der diese Aufgabe übernehmen kann. An diesen „delegiert“ er also diese Aufgabe (siehe Abbildung 7).

```
(…)
@interface PVActionsMenuViewController : UIViewController
@property (nonatomic, strong) id <PVActionsMenuDelegate> delegate;
@end
(…)
```

Abbildung 7: PVActionsMenuViewController verweist auf ein Delegate-Objekt

Der Delegate muss das Protokoll *PVActionsMenuDelegate* einhalten, das die Implementierung der Methode *deleteViewButtonClicked* vorschreibt. Abbildung 8 zeigt dieses Protokoll.

```
(…)
@protocol PVActionsMenuDelegate <NSObject>
- (void)deleteViewButtonClicked;
@end
(…)
```

Abbildung 8: Protokoll für einen Delegate, der für den PVActionsMenuViewController in Frage kommt

Der Controller *PVViewController* implementiert dieses Protokoll und zeigt dadurch an, dass er auf einen entsprechenden Methodenaufruf reagieren kann. Er muss dadurch also auch die vorgeschriebene Methode implementieren. Abbildung 9 zeigt, wie der *PVViewController* u. a. das *PVActionsMenuDelegate*-Protokoll implementiert:

```
(…)
@interface PVViewController () <(…) PVActionsMenuDelegate>
(…)
```

Abbildung 9: PVViewController implementiert PVActionsMenuDelegate

So kann der *PVActionsMenuViewController* den Methodenaufruf „bedenkenlos“ an seinen Delegate schicken, wenn er den *PVViewController* als solchen referenziert (siehe Abbildung 10).

```
(…)
- (IBAction)deleteViewButtonClicked:(id)sender {
  [self.delegate deleteViewButtonClicked];
}
(…)
```

Abbildung 10: PVActionsMenuViewController ruft die im Protokoll definierte Methode seines Delegate auf

Abbildung 11 zeigt ein Interaktionsdiagramm, das den beschriebenen Vorgang zusätzlich verdeutlichen soll.

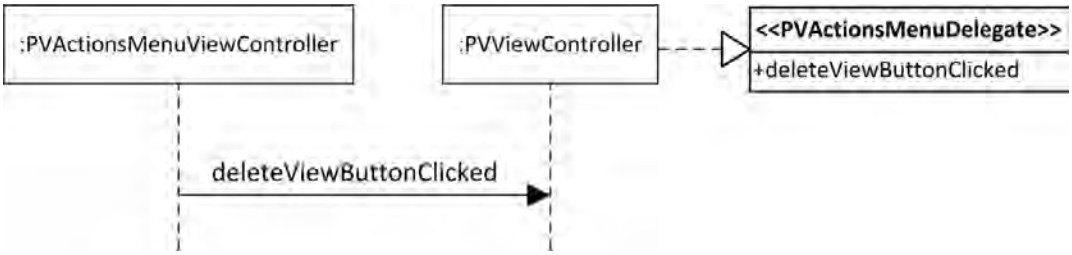

Abbildung 11: Interaktionsdiagramm

2.4 Hybride Datenstrukturen

Gemäß einer strikten Trennung von *Model* und *View* (siehe *MVC-Pattern* in Kapitel 2.3.1) müsste auch für die Repräsentation des Prozessgraphen eine Differenzierung der Klassen für Model und View erfolgen.

Demnach wäre beispielsweise für einen Aktivitätsknoten zum einen eine entsprechende View (Widget) und zum anderen ein passendes Model (Data Source) zu implementieren. Gleichzeitig wären dann die jeweiligen Beziehungen dieses Aktivitätsknotens zu anderen Knoten im Prozessgraphen ebenso auf beiden Ebenen zu realisieren. Ein solches redundantes Vorgehen erhöht jedoch grundsätzlich die Wahrscheinlichkeit von Programmierfehlern und Performance-Einbußen, wenn ein und dasselbe Problem doppelt gelöst werden muss.

Daher wird in dieser Arbeit für die Repräsentation des Prozessgraphen auf das MVC-Pattern verzichtet. Stattdessen werden Klassen zum Einen sowohl im Model als auch in der View zur Datenrepräsentation verwendet (*Fall Nr.1*). Zum Anderen gibt es Klassen, die anhand ihrer Methoden beides, View- und Model-Funktionalität, bereitstellen (*Fall Nr.2*).

Die Klasse *PVEdge* ist ein Beispiel für *Fall Nr.1*. Sie dient im Model – nach dem Parsen der XML-Repräsentationen der Templates – dem logischen Aufbau von Kantenverbindungen zwischen den Aktivitäts- und Verzweigungsknoten des Prozessgraphen. Auf der View-Seite findet sie aber auch Verwendung, nämlich beim Zeichenprozess der Datenflusskanten auf dem Bildschirm. Abbildung 12 illustriert diesen Sachverhalt.

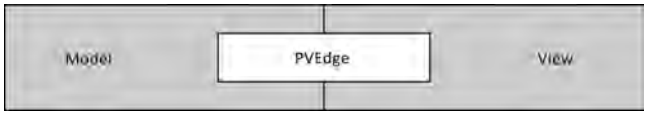

Abbildung 12: PVEdge in Model und View

Klassen, die das Protokoll *PVGraphContainer* implementieren, sind Beispiele für *Fall Nr.2*. Sie bilden zum Einen rekursiv die Struktur des jeweiligen Prozessgraphen ab, indem sie beispielsweise Verweise auf Folgeknoten speichern. Zum Anderen agieren sie bereits auf der Ebene der View, indem sie den Mechanismus der *impliziten Kantenzeichnung* (siehe Kapitel 3.1.1) bei jeder Verbindung von Knoten durchführen und auch

die Ausrichtung der Knoten zueinander jeweils autonom übernehmen. Dieses Verhältnis soll in Abbildung 13 veranschaulicht werden.

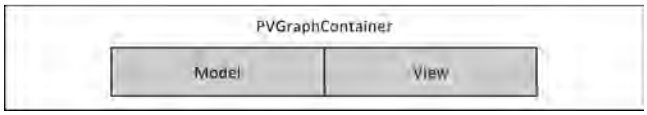

Abbildung 13: PVGraphContainer-Objekte übernehmen Funktionen von Model und View

2.5 Gestensteuerung

Der Begriff *Gestensteuerung* ist ein zentraler Aspekt dieser Arbeit, weshalb er zunächst in Kapitel 2.5.1 definiert und eingegrenzt wird. Anschließend werden in Kapitel 2.5.2 die sogenannten *Gesture Recognizer* sowie verschiedene *Multitouch-Gesten* vorgestellt.

2.5.1 Zum Begriff Gestensteuerung

Um den Begriff der Gestensteuerung im Bereich mobiler Endgeräte besser verstehen und einordnen zu können, ist es sinnvoll, sich zunächst die folgenden zwei Definitionen aus dem Bereich der Sprachwissenschaften anzusehen. Sie beziehen sich zunächst auf die Begriffe *Gestik* bzw. *Gesten* (siehe Definition 1 und 2).

Definition 1:

„Gesture is integral to human language. Its function within human communication is as much goal-directed, and subsequently as communicative, as is speech.“ [11]

Definition 2:

„Gestures are 'excursions': phrases of action recognized as 'gesture' move away from a 'rest position' and always return to a rest position“ [12]

Der Begriff Gestensteuerung tritt häufig im Kontext von Computern mit sogenannten *Multitouch-Displays* auf. Diese sind in der Lage, die Position mehrerer Finger zu erkennen, welche die Oberfläche des Multitouch-Displays berühren. Auf diese Weise ist es möglich, Gesten in einer Form von Daten zu repräsentieren, welche von einem Computer ausgelesen und berechnet werden können. So können sowohl die Richtung als auch Anfangs- und Endposition einer auf dem Multitouch-Display vollzogenen *Multitouch-Geste* bzw. der an ihr beteiligten Finger erfasst und als Informations-Einheit erkannt werden.

Somit ergibt sich im Kontext von Computern folgende Definition (siehe Definition 3).

Definition 3:

Gestensteuerung ist eine Form der Mensch-Computer-Interaktion. Sie erfolgt über Gesten (siehe Definition 1 und 2), deren messbare Informationen von einem Computer mittels dessen Sensorik wahrgenommen und verarbeitet werden können.

Die in dieser Arbeit implementierte Gestensteuerung verwendet das Multitouch-Display des Apple iPad. Auf diesem Display vollzogene Multitouch-Gesten werden mit Hilfe sogenannter *Gesture Recognizer* erkannt, welche in Kapitel 2.5.2 genauer beschrieben werden.

2.5.2 Gesture Recognizers

Ein *Gesture Recognizer* ist eine im *User-Interface-Framewo*rk für iOS bereitgestellte Komponente zur Erkennung von Gesten innerhalb eines ihm zugewiesenen Widgets. Ein Widget ist eine grafische Interaktions-Komponente mit einem rechteckigen Bereich, dessen Inhalt auf ein Display gezeichnet werden kann. In dieser Arbeit werden ein *Pan Gesture Recognizer* und ein *Pinch Gesture Recognizer* verwendet.

Der *Pan Gesture Recognizer* reagiert auf sogenannte Pan-Gesten (Schiebegesten), die auf einem Multitouch-Display ausgeführt werden. Er liefert die jeweiligen Koordinaten der Berührungspunkte sowie den dazu passenden Verschiebungsvektor. Abbildung 14 zeigt eine solche Pan-Geste.

Abbildung 14: Pan-Geste

Auf Gesten, bei denen mindestens zwei Finger beteiligt sind, die sich jeweils voneinander entfernen bzw. aufeinander zubewegen, reagiert der *Pinch Gesture Recognizer*.

Dieser liefert Informationen über die jeweiligen Berührungspunkte und deren Position sowie deren relative Abstandsveränderung als Float-Wert. Abbildung 15 zeigt eine Pinch-Geste.

Abbildung 15: Pinch-Geste

Außerdem sind folgende Gesture Recognizer im *User-Interface-Framework* enthalten:

- *Swipe Gesture Recognizer* (zur Erkennung von Wisch-Gesten)
- *Rotation Gesture Recognizer* (zur Erkennung von Dreh-Gesten)
- *Long Press Gesture Recognizer* (zur Erkennung von Druck- und Halte-Gesten)
- *Tap Gesture Recognizer* (zur Erkennung von Berührungs-Gesten)

3. Anforderungen und Vorgehensweisen

Dieses Kapitel beschreibt die geforderten Eigenschaften des zu entwickelnden Prototyps sowie die entsprechende Umsetzung dieser Vorgaben während des Entwicklungsprozesses. In Kapitel 3.1 geht es zunächst um die Darstellung des Prozessgraphen auf dem iPad. Kapitel 3.2 beschreibt die Interaktion der App mit dem proView-Server. Im letzten Schritt erklärt Kapitel 3.3, wie die Manipulation des Prozessgraphen mit Hilfe der zu implementierenden Gestensteuerung funktioniert.

3.1 Anforderung Nr. 1: Darstellung des Prozessgraphen auf dem Apple iPad

Die wichtigste Anforderung an die iPad-App ist die Darstellung des Prozessgraphen. Diese sollte sich an den bisherigen proView-Projekten orientieren und das Anzeigen von Datenflusskanten „on demand“ ermöglichen.

3.1.1 Umsetzung der Knoten-Darstellung

Knoten eines Prozessgraphen werden in *proViewMobile* in *Containern* positioniert. Ein Container beherbergt in der Regel ein oder zwei Knoten. Ursprünglich als Subklasse von *UIView* konzipiert (siehe Kapitel 5), besitzen diese Container die Eigenschaft *frame*. Diese speichert die Daten für ein Rechteck, das einen Ursprungspunkt mit x- und y-Koordinate sowie eine Breite und eine Höhe enthält. Anhand dieser Rechteckstruktur lässt sich ein Prozessgraph schrittweise aufbauen, wobei die einzelnen Elemente ineinander verschachtelt werden. Abbildung 16 zeigt eine solche Verschachtelung nach dem Hinzufügen eines einfachen Aktivitätsknoten-Containers (*PVSimpleNode*) zu einem Container, der Start- und Endknoten enthält (*PVMainGraphContainer*).

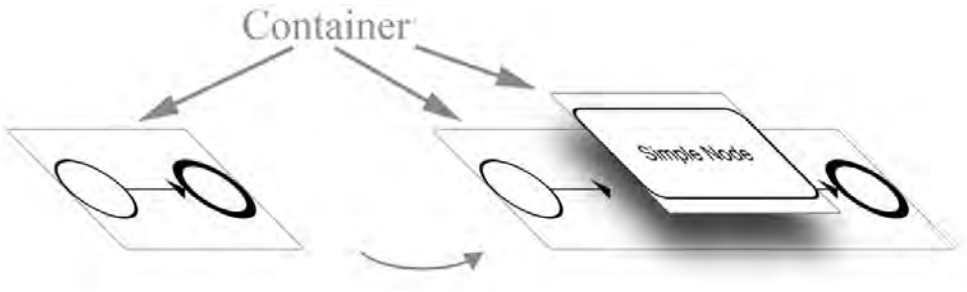

Abbildung 16: Verschachtelung von Containern

Die Kanten sind hier sozusagen bereits „von Haus aus“ vorhanden und müssen nicht noch einmal gesondert gezeichnet werden. Denn jeder Container enthält an dessen Anfang oder Ende bereits im Voraus „ins Nichts“ gezeichnete Kanten, die dann an den jeweils hinzugefügten Container grenzen. Dadurch werden also zwei Knoten **implizit miteinander verbunden** *(implizite Kantenzeichnung)*, da die Verbindung nur ein Nebenprodukt der Anordnung von Containern ist.

Diese Vorgehensweise ermöglicht gleichsam die Umgehung eines Problems, das in Abbildung 17 illustriert wird: Datentechnisch gleich repräsentierte Kanten müssen unter Umständen je nach Lage im Prozessgraphen völlig unterschiedlich visualisiert werden (siehe Kanten (a) und (b) in Abbildung 17). Durch die Verlagerung der Kanten auf Container, müssen diese aber letztlich gar nicht hinsichtlich ihres Verlaufs unterschieden werden. Sie befinden sich dann bereits mit der richtigen Optik an der richtigen Position.

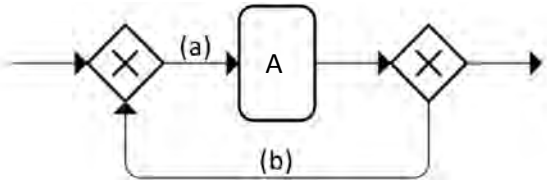

Abbildung 17: Unterschiedliche Visualisierung datentechnisch gleicher Kanten

Damit die Verschachtelung verschiedener Container gelingen kann, ist die Definition eines Protokolls notwendig, das von allen Containern eingehalten wird (siehe Kapitel 2.3.2). Dieses Protokoll heißt *PVGraphContainer* und verlangt die Implementierung der folgenden Eigenschaften:

*1) NSString *nodeId:*

Die *nodeId* identifiziert jeden Knoten eindeutig. Sie ist daher ein wesentlicher Bestandteil jedes Knotens und dient einerseits zum Auffinden eines Knotens im Prozessgraphen, andererseits zur Identifikation eines Knotens bei der Interaktion mit dem Server.

2) CGRect frame:

Wie oben bereits erwähnt, dient diese Eigenschaft der Bestimmung der Ausdehnung eines Containers und dessen Offsets relativ zu seinem Eltern-Container.

*3) PVConnectorOffset *connectorOffset:*

Dieses Objekt enthält die Verknüpfungspunkte eines vom Container beherbergten Knotens relativ zum Gesamtbezugssystem. Für jeden Knoten ist jeweils ein Punkt links, rechts, oben und unten definiert, der als Verknüpfungspunkt in Frage kommt. Diese Punkte sind wichtig bei der expliziten Kantenzeichnung (siehe Kapitel 3.1.2). Außerdem enthält das *connectorOffset* einen sogenannten *interactionFrame*. Dieses Rechteck gibt die Position des Knotens inklusive dessen Breite und Höhe an. So kann der Knoten aufgrund eines vom Benutzer berührten Punktes auf dem Bildschirm gefunden werden.

4) id parentContainer:

Hier wird ein Zeiger auf den Eltern-Container gespeichert. Diese Eigenschaft muss gesondert implementiert werden, da es sich bei den Containern nicht – wie ursprünglich eigentlich geplant – um Subklassen von *UIView* handelt (siehe Kapitel 5). Solche erben normalerweise bereits die Eigenschaft *superview*, welche die hierarchisch höher liegende *UIView* referenziert.

5) id <PVGraphContainer> nextContainerInChain:

Diese Eigenschaft gibt den jeweils folgenden Container an. Unter Umständen kann hier also auch „nil“ zurückgegeben werden, wenn kein weiterer Container existiert.

Außerdem werden folgende Methodenimplementierungen in *PVGraphContainer* vorausgesetzt:

1) - (void)addNextContainer:(id <PVGraphContainer>)nextContainer:

Jeder Container muss in der Lage sein, einen weiteren Container hinzuzufügen. Tatsächlich wird der weitere Container dem ursprünglichen einverleibt. Das bedeutet, dass dieser seine *frame*-Werte entsprechend erweitert, um den anderen Container zu umfassen oder zumindest am rechten Rand zu begrenzen. Außerdem wird der *frame* des einverleibten Containers entsprechend der notwendigen Offsets automatisch angepasst.

2) - (void)spread:

Diese Methode wird nach dem Hinzufügen eines Containers bei eben diesem aufgerufen und veranlasst den hinzugefügten Container, „sich auszubreiten“. Das bedeutet, dass sich dessen Größe daraufhin anpasst (z. B. weil er beschriftet ist). Die Methode wird meist rekursiv durch den Prozessgraphen propagiert, bis sie auf das jeweils letzte Kindelement stößt. Dieses breitet sich dann aus und gibt die Änderung seiner Größe mit der Methode *updateFrameWithSize* an sein Elternelement weiter.

3) - (void)updateFrameWithSize:(CGSize)size:

Diese Methode wird von einem Kindelement an dessen Elternelement aufgerufen, wenn es seine Größe geändert hat, um diesem seine neue Größe mitzuteilen.

4) - (void)createConnectorOffsetWithOrigin:(CGPoint)origin:

Wenn der Graph vollständig aufgebaut wurde, kann vom Wurzelelement aus diese Methode aufgerufen werden. Sie „bestückt“ daraufhin alle Elemente mit deren Verknüpfungspunkten relativ zum Bezugssystem. Dabei gibt jedes Elternelement rekursiv sein eigenes „Offset“ an seine Kindelemente weiter, wodurch diese ihre absolute Position erfahren.

Optional kann die folgende Methode implementiert werden:

*- (PVConnectorOffset *)getConnectorOffsetForNodeId:(NSString *)nodeId:*

Wird diese Methode innerhalb des Graphen aufgerufen, so wird nach dem Knoten mit der entsprechenden „nodeId“ gesucht. Der zuständige Container gibt dann ein „ConnectorOffset“ relativ zum aufrufenden Container zurück. Gerade bei Containern, die Verzweigungen beinhalten, ist dies eine wichtige Methode für die *implizite Kantenzeichnung*.

Die einfachste Implementierung des Protokolls *PVGraphContainer* ist die Klasse *PVSimpleNode*. Diese repräsentiert genau einen Aktivitätsknoten. Abbildung 18 zeigt zwei PVSimpleNodes, die miteinander verknüpft sind. Dabei spannt sich der erste Container so weit auf, dass er den Zweiten in sich aufnehmen kann. Die Kante kann danach einfach in den Zwischenraum gezeichnet werden.

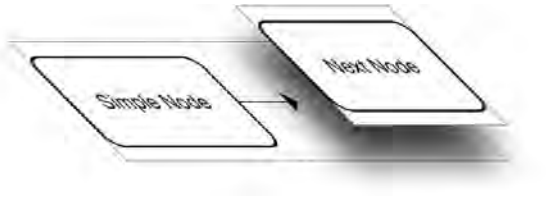

Abbildung 18: PVSimpleNode weiterem hinzugefügtem PVSimpleNode

Ein PVSimpleNode enthält ein *UILabel* (ein Widget zur Darstellung von Text) als Eigenschaft. Dieses wird jedoch nicht dazu verwendet, den Titel anzuzeigen, sondern lediglich um die erforderliche Breite des gesamten Containers festzustellen, die je nach Länge des jeweiligen Titels variieren kann.

Um einen *Loop* (siehe Kapitel 2.1) darzustellen, wird ein sogenannter *PVLoopContainer* verwendet. Er beinhaltet zwei Knoten, einen „XOR-Split“- und einen „XOR-Join“-Knoten. Zwischen beide Knoten kann ein beliebiger Container eingefügt werden. Je nach dessen Größe dehnt sich der PVLoopContainer daraufhin entsprechend aus. Alle Kanten können aufgrund der vorliegenden Blockstruktur nach einem einfachen Schema auf der Fläche des Containers gezeichnet werden (siehe Abbildung 19). Der innere Container beeinflusst dabei lediglich die Höhe des äußeren Containers und damit nur indirekt die zurückführende Kante, welche sich an dessen unterem Rand orientiert.

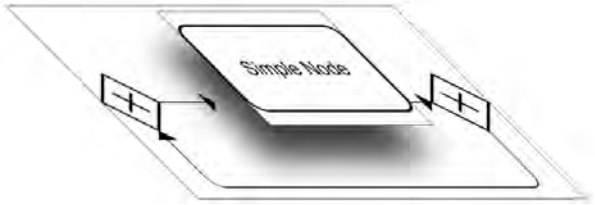

Abbildung 19: PVLoopContainer mit hinzugefügtem PVSimpleNode und bereits zuvor vorhandenen Kanten

Ein *PVLoopContainer* kann sowohl mit der Methode *addSubContainer* einen Container in sich aufnehmen als auch per *addNextContainer* (siehe PVGraphContainer-Protokoll oben) einen Container hinter sich anfügen.

Die komplexeste Container-Ausprägung ist die Klasse *PVSplitContainer*. Sie ist entweder vom Typ *XOR* oder *AND* und kann mehrere Container in Verzweigungen aufnehmen. Dabei „wächst er quasi mit seinen Aufgaben“ (siehe Abbildung 20). Auch hier können die Kanten relativ einfach an die inneren Container angefügt werden. Ein Spezialfall liegt vor, wenn ein Container auf Höhe der Raute positioniert wird (siehe Abbildung 20 rechts unten). Dann muss eine Kante direkt aus der Raute kommend gezeichnet und der betroffene Container unter Umständen bezüglich seines *Offsets* leicht angepasst werden, damit die Kante mittig ausgerichtet werden kann.

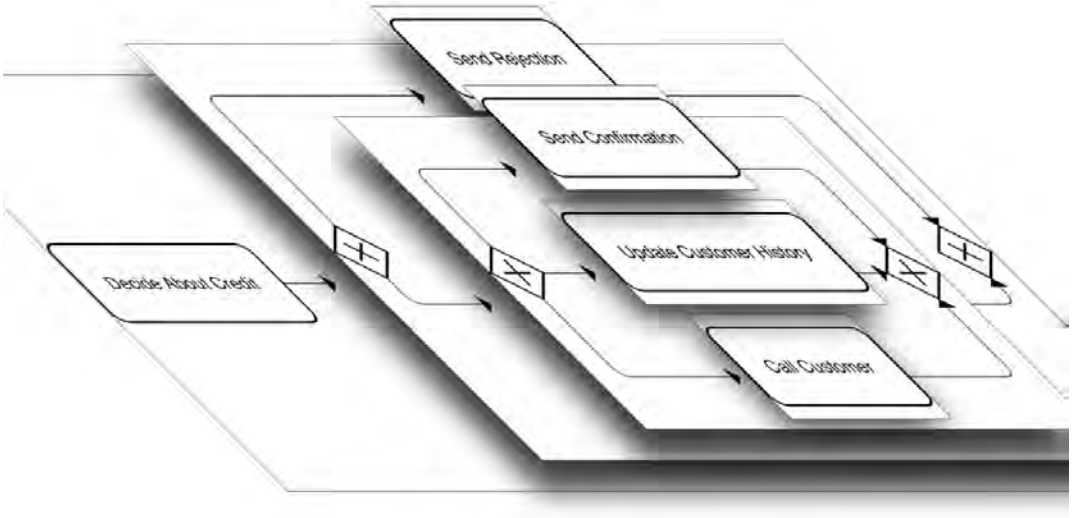

Abbildung 20: PVSplitContainer verschachtelt

Obwohl Datenelementknoten keine Sequenzflusskanten haben können, implementieren auch sie das PVGraphContainer-Protokoll. Sie benötigen nämlich u. a. die Eigenschaft „nodeId“ und das PVConnectorOffset-Objekt, das für die Zeichnung der Datenflusskanten wichtig ist. In Abbildung 21 ist ein *Screenshot* aus proViewMobile zu sehen, der vier Datenelemente mit entsprechenden Datenflusskanten zeigt. Grüne Kanten bedeuten in proViewMobile „Lesen“, rote Kanten „Schreiben“. Neben der Farbe ist der Kantentyp zusätzlich über die Pfeilrichtung codiert.

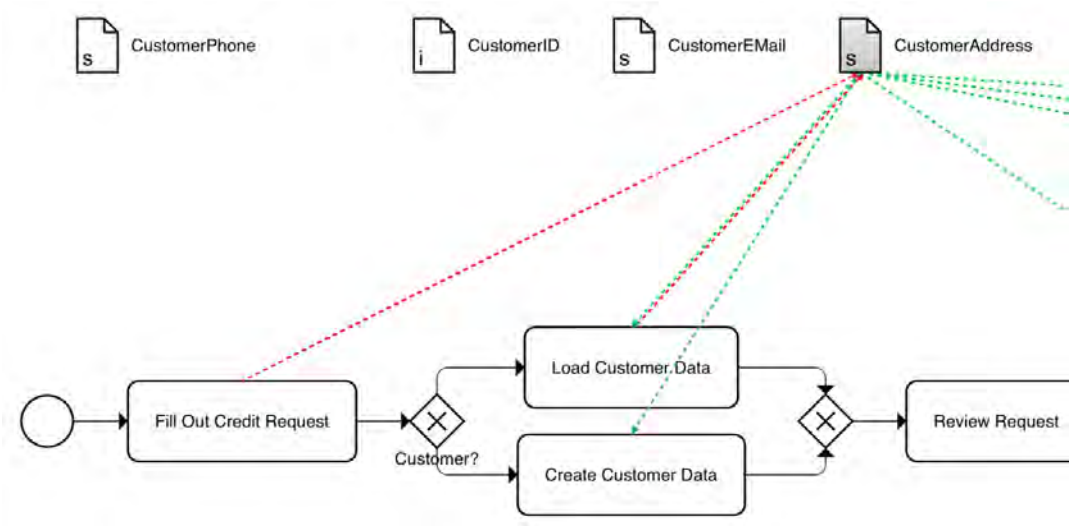

Abbildung 21: Datenelemente und Datenflusskanten

Anders als bei den anderen Containern liegen *PVDataElementNodes* auf einem gemeinsamen Eltern-Container, dem *PVDataElementContainer*. Dieser übernimmt anhand eines Algorithmus, der die Anzahl und Position ein- und ausgehender Kanten bzw. deren Zielknoten gewichtet, die Ausrichtung bezüglich des restlichen Prozessgraphen.

Genau wie bei der Klasse PVSimpleNode bestimmt auch ein *PVDataElementNode* seine jeweilige Größe mit Hilfe eines UILabels.

3.1.2 Umsetzung der Kanten-Darstellung

Mit Hilfe der Implementierungen des Protokolls PVGraphContainer ist es möglich, bestimmte Kanten implizit zu zeichnen (siehe Kapitel 3.1.1). Dennoch gibt es Kanten, die explizit gezeichnet werden müssen: die Datenflusskanten. Diese sind einfacher, da sie auf direktem Weg von einem Punkt zum anderen verlaufen. Das PVGraphContainer-Protokoll definiert für diesen Fall eine wichtige Schnittstelle: das *PVConnectorOffset*, mit dem bei jedem Knoten auf dessen mögliche Verknüpfungspunkte an allen Seiten zugegriffen werden kann. Bei Datenelementknoten wäre dies folglich ein Punkt an der unteren Seite, bei Aktivitäts- bzw. Verzweigungsknoten entsprechend ein Punkt an der oberen Seite.

Eine besondere Behandlung benötigt das gleichzeitige Auftreten von Lese- und Schreibekanten. Diese sollten sich beim Zeichnen nicht überlagern, da sie sich sonst gegenseitig verdecken würden. Erlaubt man außerdem eine Mehrfachauswahl verschiedener Knoten, muss sichergestellt werden, dass jede Kante nur einmal gezeichnet wird, auch dann wenn sie für mehrere Knoten relevant ist. Wird ein Knoten abgewählt, darf eine solche Kante auch nicht entfernt werden. ProViewMobile verwendet zur Realisierung dieser Funktion drei Klassen:

1) PVDataEdgeView

PVDataEdgeView erbt von UIView und ist damit ein UI-Element, das Zeichenoperationen ausführen kann. Seine Aufgabe ist das Zeichnen der Kanten, die vom Benutzer durch das An- und Abwählen von Knoten auf dem Display sichtbar gemacht werden. Dabei werden zu jedem angewählten Knoten dessen ein- und ausgehende Datenflusskanten dargestellt. PVDataEdgeView enthält ein Array (im Folgenden *Kantenpool* genannt) von *PVEdgeCollection*-Objekten, mit dessen Hilfe die nötigen Informationen über die zu zeichnenden Kanten abgerufen werden können.

2) PVEdgeCollection

Diese Klasse verwaltet eine variable Anzahl von Kanten, die sich in einem Merkmal gleichen: Sie haben alle die gleichen Quell- und Zielknoten. Um diese Klasse zu repräsentieren genügt also ein PVEdge-Objekt, das die benötigten Referenzen auf die passenden Knoten enthält. Wird ein bestimmter Knoten auf dem Display ausgewählt, so dass dessen Datenflusskanten angezeigt werden, wird für jede betreffende Kante eine solche PVEdgeCollection mit der entsprechenden Repräsentation erzeugt, oder – falls eine „ähnliche“ Kante bereits ausgewählt ist – eine neue Kante an einer bestehenden

PVEdgeCollection *registriert*. Dies funktioniert mit der dritten Klasse, *PVEdgeRegistration*. Sind mehrere Kanten registriert (z. B. durch eine Lese- und Schreibkante) wird auf Anfrage der Methode *getDominantEdge* eine Kante mit dem Typ *READ_AND_WRITE* zurückgegeben, die dann wie eine einzige Kante mit zwei Farben und zwei Pfeilen gezeichnet werden kann, was eine Überlagerung beider Kanten verhindert.

3) PVEdgeRegistration

In dieser Klasse wird der Bezug zwischen einer bestimmten Kante und demjenigen Knoten hergestellt, der für die Anzeige der Kante „verantwortlich“ ist. Wird der Knoten vom Benutzer also wieder abgewählt, wird aus dem Kantenpool nur dessen Registrierung gelöscht. Sollte ein anderer ausgewählter Knoten dieselbe Kante besitzen, so wird diese nicht entfernt.

3.1.3 Zeichnen des Graphen

Wie zuvor erwähnt, sind Klassen, die das Protokoll PVGraphContainer implementieren, im Allgemeinen keine UIViews. Das bedeutet, dass sie sich nicht selbst zeichnen, sondern lediglich mit entsprechenden Informationen zum Zeichnen beitragen können. Das tatsächliche Zeichnen übernimmt die Klasse *PVMainView*.

Die Klasse *PVMainView* ist eingebettet in eine *PVScrollView* (ein Widget, das Scrolling für seine Kind-Widgets ermöglicht). Sieht man von den notwendigen Verbindungen zu dieser Superview-Klasse ab, ist die PVMainView so etwas wie das oberste Wurzelelement des Prozessgraphen. Die meisten rekursiven Methodenaufrufe, die den Prozessgraphen durchlaufen, werden von ihr gestartet oder enden bei ihr. Sie enthält den *PVDataElementContainer*, der zur Verwaltung der Datenelemente dient, und den *PVMainGraphContainer*, der Start- und Endknoten des Prozessgraphen kapselt. Und sie ist verantwortlich für das Zeichnen des Prozessgraphen. Neben der rekursiven Verkettung des Prozessgraphen (ausgehend vom PVMainGraphContainer), die durch Zeiger-Verbindungen realisiert ist, existiert zusätzlich eine Liste aller Knoten, die zum Zeichnen traversiert werden kann. Außerdem liegen Referenzen aller Knoten in einem *NSDictionary*, welches Zeiger anhand von Schlüsseln verwaltet. So kann über die „nodeId“ auf einen Knoten bzw. dessen Container zugegriffen werden.

Zum Zeichnen wird die Knoten-Liste Schritt für Schritt durchlaufen. Dabei wird je nach Typ der Klasse eine andere Zeichenoperation angestoßen. Diese Zeichenoperation be-

kommt das jeweilige Container-Objekt als Parameter übergeben und liest aus diesem die erforderlichen Daten. Dazu gehören z. B. die Größe und Position des Knotens und der an ihn angrenzenden Kanten. Diese Daten sind zum Zeitpunkt des Zeichnens bereits durch das anfängliche Verknüpfen der Container berechnet worden.

Wie in Kapitel 3.1.2 beschrieben, werden auch die Datenflusskanten über eine spezielle UIView-Klasse gezeichnet: die PVDataEdgeView. Diese wird von der Klasse PVMainView mit den entsprechenden Daten über aktuell ausgewählte Knoten informiert. Dabei passt sie sich, was Größe und Position betrifft, den jeweiligen Erfordernissen an: Ihr *frame* befindet sich immer nur da, wo auch eine Kante gezeichnet werden muss.

3.2 Anforderung Nr. 2: Interaktion mit dem proView-Server

Nach der vollständigen Darstellung des Prozessgraphen ist die Interaktion mit dem proView-Server der nächste logische Schritt im Entwicklungsprozess von proViewMobile.

Zur Kommunikation zwischen einem proView-Client und einem proView-Server wird XML eingesetzt, was die Implementierung eines *XML-Parsers* erfordert. Mit dem sogenannten *Foundation Framework* steht mit der Klasse *NSXMLParser* auch bereits ein *eventbasierter XML-Parser* zur Verfügung. Dieser erzeugt beim Parsen einer XML-Datei nach der Erkennung einzelner *XML-Tags* Events, welche bei der Implementierung entsprechend behandelt werden können, um das eigene Datenmodell anhand der ausgelesenen Daten zu erstellen.

3.2.1 Abrufen und Erstellen von Templates

Das Abrufen von Templates ist über eine RESTful-Schnittstelle möglich. Dabei wird – REST-typisch – eine für jedes Template eindeutige URL (siehe Abbildung 22) per *GET-Request* abgefragt.

```
http://[proView-Server-Basis-URL]/process/16fa24a9-46c7-416a-a187-
e3cdd52610d9/views/340680e5-592f-4aff-8b1b-7bd7edb782d3
```

Abbildung 22: URL für eine View

Als Antwort wird die Ressource in ihrer XML-Repräsentation vom Server ausgegeben (siehe Abbildung 23).

```
<template id="0bf59cff-801f-4249-896b-4b9f8c50ac9a" version="16"
                    xsi:schemaLocation="(…)">
  <name>CreditApplication</name>
  <description/>
  <processType/>
  <supervisorAgent>(…)</supervisorAgent>
  <version/>
  <nodes>(…)</nodes>
  <dataElements>(…)</dataElements>
  <edges>(…)</edges>
  <dataEdges>(…)</dataEdges>
  <structuralData>(…)</structuralData>
  <instanceNameTemplate/>
  <pluginDataContainer>(…)</pluginDataContainer>
</template>
```

Abbildung 23: XML-Struktur eines Templates

Abbildung 23 zeigt, dass die Bestandteile des Prozessgraphen in verschiedenen Listen zur Verfügung stehen. So enthält das Element *nodes* (siehe Abbildungen 23 und 24) eine Liste der Knoten und das Element *edges* eine Liste der Kanten im Prozessgraph. Die Verbindungen der Knoten im Prozessgraph können nach dem Einlesen der Daten in einem weiteren Schritt erzeugt werden.

Abbildung 24 zeigt die Kindelemente des Elements *nodes*, welche in Abbildung 23 ausgelassen wurden und einzelne Knoten repräsentieren. Dabei kann von jedem Knoten dessen ID und der Name ausgelesen werden, nicht jedoch der Knotentyp. Erst die Einträge innerhalb des Elements *structuralData* klassifizieren den entsprechenden Knotentyp.

```
<nodes>
  (…)
  <node id="n2">
    <name>Fill Out Credit Request</name>
    <description>(…)</description>
    <staffAssignmentRule>(…)</staffAssignmentRule>
    <autoStart>false</autoStart>
  </node>
  (…)
</nodes>
```

Abbildung 24: Ausschnitt aus dem Element „nodes“

Abbildung 25 zeigt ein Beispiel für Kindelemente von structuralData.

```
<structuralData>
  (…)
  <structuralNodeData nodeID="n2">
    <type>NT_NORMAL</type>
    <topologicalID>1</topologicalID>
    <branchID>1</branchID>
    <splitNodeID>n0</splitNodeID>
    <correspondingBlockNodeID>n0</correspondingBlockNodeID>
  </structuralNodeData>
  (…)
  <structuralNodeData nodeID="n4">
    <type>NT_XOR_SPLIT</type>
    <topologicalID>2</topologicalID>
    <branchID>1</branchID>
    <splitNodeID>n0</splitNodeID>
    <correspondingBlockNodeID>n5</correspondingBlockNodeID>
  </structuralNodeData>
  (…)
</structuralData>
```

Abbildung 25: Ausschnitt aus dem Element „structuralData“

Im Template wird nicht nur klar definiert, um was für einen Knoten es sich jeweils handelt. Denn im Falle des XOR-Verzweigungsknotens ist darin noch eine weitere wichtige Information enthalten: die passende ID des Knotens, der den XOR-Block (siehe „Blockstruktur“ in Kapitel 2.1) wieder schließt (siehe Element „correspondingBlockNodeID“ in Abbildung 25).

Im Verlauf des Parsens eines Templates werden also zunächst beim Durchlaufen der *nodes*-Elemente sogenannte *Knoten-Prototypen* (*PVNodePrototype*) erzeugt. Diese werden dann mit den Daten der *structuralNodeData*-Elemente vervollständigt. Abschließend kann dann die Liste der Knoten-Prototypen durchlaufen und der jeweils passende Container erstellt werden.

Mit Hilfe der Kanten, die in den *edge*-Elementen definiert werden, kann dann schrittweise der Prozessgraph konstruiert werden, indem Quell- und Zielknoten (die jeweils auf den passenden Container verweisen) miteinander verbunden werden. Dies geschieht mit der Methode *addNextContainer*, die von allen Containern implementiert wird (siehe PVGraphContainer-Protokoll in Kapitel 3.1.1).

Da ein Container bis zu zwei Knoten beinhalten kann, ist es zusätzlich notwendig, diese Knoten voneinander zu unterscheiden. Bei einem Verzweigungs-Container (siehe Kapitel 2.1) macht es schließlich einen Unterschied, ob ein Container nach dem „Split“-Knoten oder nach dem „Join“-Knoten eingefügt wird. Bereits beim Erstellen des Graphen nach dem Parsen des Templates muss also eine Unterscheidung stattfinden. Hierfür wird zunächst ein sogenannter *PVJoinNodePromoter* verwendet. Dieser implementiert das *PVGraphContainer-Protokoll* und verfügt damit über die Operation *addNextContainer*. Außerdem besitzt er einen Zeiger auf den „Split“-Container. Wird er mit der passenden *nodeId* des „Join“-Knotens ausgestattet, kann er die entsprechende Operation an den „Split“-Container weiterleiten, welcher den einzufügenden Container dann richtig platziert.

Jeder Container mit zwei Knoten enthält für den weiteren Programmverlauf einen sogenannten „Geister-Container“, den *PVJoinGhostNode*, der als Verweis auf den Join-Knoten des Containers dient (siehe Abbildung 26). Er ist kein „echter“ Container, da er nie gezeichnet wird (der entsprechende Knoten wird schließlich bereits von einem anderen Container gezeichnet). Dennoch implementiert auch er das Protokoll *PVGraph-Container* und enthält damit die nötigen Informationen für den repräsentierten „Join“-Knoten. Dazu gehört neben der *nodeId* auch das *PVConnectorOffset*, was wiederum den *interactionFrame* bereitstellt. Auf diese Art und Weise können die Knoten trotz

gemeinsamem Container (der Container repräsentiert explizit nur den ersten Knoten) unterschieden und einzeln zu verschiedenen Operationen verwendet werden.

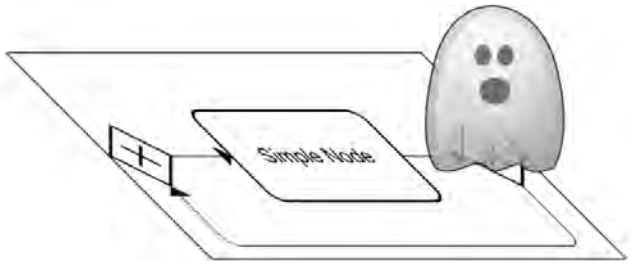

Abbildung 26: PVLoopContainer mit PVJoinGhostNode

Neben der Anfrage von Daten zur Darstellung eines Prozessgraphen kann der proView-Client auch Operationen an vorhandenen Templates durchführen oder neue Templates auf dem proView-Server erstellen.

Um eine View auf Basis eines CPM zu erzeugen, wird ein entsprechender Request an den proView-Server geschickt, welcher als Antwort die neue View-ID zurückliefert (siehe Abbildung 27).

```
<Request>
  <requestOperation>CREATE</requestOperation>
  <resourceID>0bf59cff-801f-4249-896b-4b9f8c50ac9a</resourceID>
  <payloadFormat>XML</payloadFormat>
  <resourceType>VIEW</resourceType>
  <query>
    viewName=Neue View
  </query>
</Request>
```

Abbildung 27: Request zum Erzeugen einer View

Bei einem solchen *Create-Request* wird als *resourceID* die ID des CPM übertragen (siehe Abbildung 27).

Das Abrufen von Prozessgraphen funktioniert in proViewMobile per Touch auf das Bookmark-Symbol in der linken unteren Ecke des Displays, das mit einem *UIBarButtonItem* (ein Button-Widget) auf einer *UIToolbar* (ein Toolbar-Widget) realisiert ist. Die Behandlung des Touch-Events öffnet einen *UIPopoverController*. Dieser enthält einen *UINavigationController*, der wiederum einen *UITableViewController* als sogenannten *rootViewController* verwendet (siehe Abbildung 28).

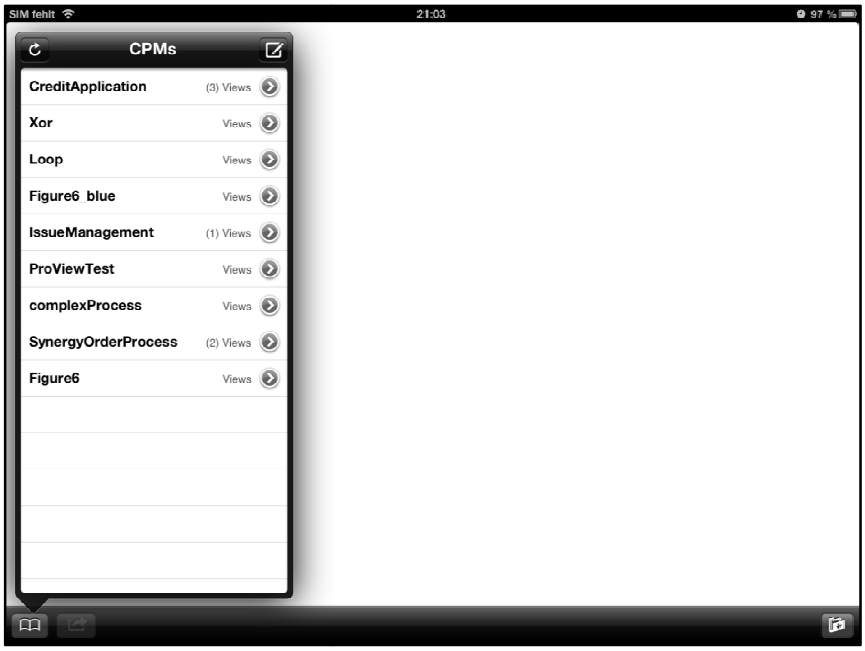

Abbildung 28: Übersicht der CPMs auf dem Server

Für die Übersicht wird besagter *UITableViewController* verwendet. So kann nun direkt ein CPM ausgewählt werden. Per Touch auf den blauen Pfeil-Button neben dem CPM-Namen gelangt man mit Hilfe des *UINavigationControllers*, der zwischen verschiedenen *View Controllern* hin- und herwechseln kann, zu einer Übersicht der zugehörigen Views. Dort kann man entweder eine neue View erstellen (Touch auf *UITabBarItem* mit Plus-Symbol, siehe Abbildungen 29 und 30) oder eine View zur Ansicht auswählen (Touch auf eine sogenannte *UITableViewCell,* siehe Abbildung 29)

Abbildung 29: Übersicht der Views

Abbildung 30 zeigt das Erstellen einer neuen View. Dabei wird die Eingabe des View-Namens mit Hilfe eines *UITextField* (Widget zur Eingabe von Text) realisiert.

Abbildung 30: Erstellen einer neuen View

Abbildung 31 zeigt das Löschen einer View im User Interface. Durch Touch auf das „Action“-Symbol (*UIBarButtonItem*) links unten öffnet sich ein *UIPopoverController* mit einem Löschen-Button (*UIButton*).

Abbildung 31: Löschen einer View

3.2.2 Änderungen an Templates

Die Templates werden nicht lokal geändert und hochgeladen, sondern es wird der Server über einen Request beauftragt, eine Änderung an einem bestimmten Template vorzunehmen. Abbildung 32 zeigt beispielhaft einen solchen Request. Dabei wurden zur besseren Lesbarkeit die Zeichen innerhalb des *query*-Elements dekodiert. Sonderzeichen liegen an der Stelle normalerweise in Form von *HTML Named Entities* vor.

```
<Request>
  <requestOperation>UPDATE</requestOperation>
  <resourceID>a23f5450-f29b-4842-870a-99d51efa7e47</resourceID>
  <payloadFormat>XML</payloadFormat>
  <resourceType>VIEW</resourceType>
  <query>
    <de.uniulm.proView.(…).view.operationset.ViewCreateOperation>
      <op class="de.uniulm.proView.(…).CreateChangeOperation">
        AGGREGATE_SESE
      <op>
      <nodeSet>
        <int>7<int>
        <int>27</int>
      </nodeSet>
      <optionSet>
        <entry>
          <string>nodeName</string>
          <string>Knoten-Name</string>
        </entry>
      </optionSet>
    </de.uniulm.proView.(…).view.operationset.ViewCreateOperation>
  </query>
</Request>
```

Abbildung 32: Request einer Aggregationsoperation

Nach der Verarbeitung des Requests antwortet der Server mit der Information, ob die Operation durchgeführt werden konnte oder nicht. Bei einer geglückten Operation (siehe Abbildung 33) kann der veränderte Prozessgraph durch einen erneuten Lese-Request neu geladen werden.

```
<de.uniulm.proView.api.entity.rest.Response>
  <request>
    <requestOperation>UPDATE</requestOperation>
    <resourceID>1d4645df-b740-41a6-a422-e69ff954438a</resourceID>
    <payloadFormat>XML</payloadFormat>
    <resourceType>VIEW</resourceType>
    <query>(…)query>
  </request>
  <status>SUCCESSFUL</status>
  <duration>0.0</duration>
  <result>
    <?xml version="1.0" encoding="UTF-8" standalone="no"?>
  </result>
</de.uniulm.proView.api.entity.rest.Response>
```

Abbildung 33: Erfolgsmeldung des proView-Servers

3.2.3 Zuständige Klassen

Für die Serverinteraktion gibt es bei proViewMobile zwei verschiedene Arten von Klassen. Zum Einen die XML-Parser. Davon gibt es einen *Template*- und einen *Template-Overview-Parser*. Der *Template-Parser* verarbeitet XML-Repräsentationen der Templates und erzeugt auf Anfrage ein *PVMainView*-Objekt, das alle notwendigen Daten des Prozessgraphen beinhaltet (siehe Kapitel 3.1.3). Der *Template-Overview-Parser* verarbeitet die Informationen über alle vorhandenen Templates auf dem proView-Server. Aus diesen generiert er sogenannte Template-Link-Container. Abbildung 34 soll einen solchen entsprechend veranschaulichen.

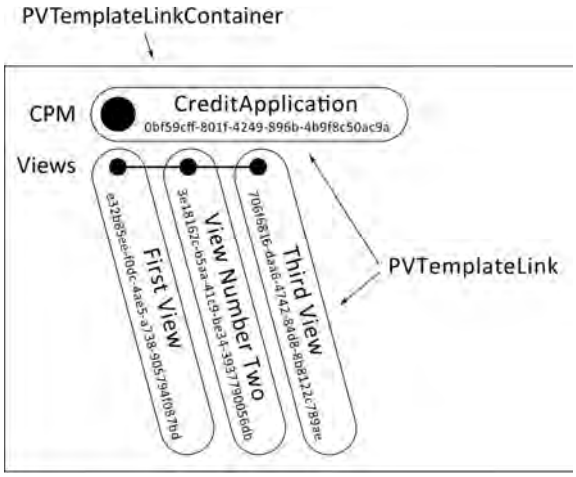

Abbildung 34: PVTemplateLinkContainer mit PVTemplateLinks

Die entstandenen *PVTemplateLinks* werden u. a. für die Übersichtsanzeige der CPMs auf dem proView-Server und für die Initialisierung des *PVTemplateParsers* verwendet.

Für das Versenden von Requests an den Server ist die Klasse *PVServerInteraction* zuständig. Sie erzeugt durch das dynamische Verknüpfen vorhandener String-Vorlagen mit den erforderlichen Daten für eine neue Operation komplette Requests und schickt diese an den proView-Server.

3.3 Anforderung Nr.3: Manipulation des Prozessgraphen durch Benutzereingaben

Der bereits vom proView-Server geladene und gezeichnete Prozessgraph soll in einem weiteren Entwicklungsschritt vom Benutzer verändert werden können. Dazu ist es zunächst einmal notwendig, die Eingaben des Benutzers abzufangen und den entsprechenden Elementen des Prozessgraphen zuzuordnen, mit denen der Benutzer interagieren möchte.

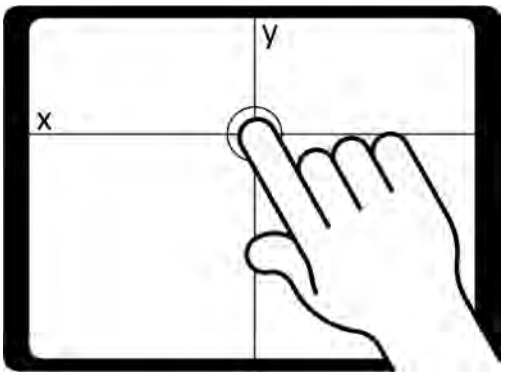

Abbildung 35: Benutzer tippt an irgendeine Stelle auf dem Bildschirm

Dies beginnt bei den Koordinaten der Berührungspunkte (siehe Abbildung 35). Dabei werden zunächst die Touch-Events im *PVMainView*-Objekt abgefangen. Danach wird die in Kapitel 3.1.1 beschriebene Klasse *PVConnectorOffset* wichtig. Diese enthält die Eigenschaft *interactionFrame*. Darin werden Größe und Position eines Knotens gespeichert. So kann beim Iterieren durch die Knotenliste der passende Knoten zum Berührungspunkt gefunden werden.

Als Repräsentation dient weiterhin der Container bzw. dessen *PVJoinGhostNode*, der den Knoten beinhaltet. Dennoch ist dieser unter Umständen viel größer als der eigentliche Knoten und kann daher nicht zur Touch-Event-Verarbeitung verwendet werden.

Bei einem einfachen „Touch“ wird lediglich der betroffene Knoten an- oder abgewählt. Dabei werden die passenden Datenflusskanten visualisiert (siehe Abbildung 21 auf Seite 25). Der Prozessgraph wird jedoch nicht verändert.

Vollzieht der Benutzer jedoch eine entsprechende Geste auf dem Display, kann er eine Veränderungsoperation am Prozessgraphen durchführen. Um solche Gesten zu erkennen, werden die in Kapitel 2.5.2 beschriebenen Gesture Recognizer eingesetzt.

3.3.1 Gestensteuerung

An den beiden Gesten, die in proViewMobile implementiert sind, sind zwei bzw. ein Finger beteiligt. In jedem Fall muss für eine potentielle Geste der entsprechende Knoten erfasst werden. Hierfür werden Slots verwendet (siehe Abbildung 36), in denen Zeiger auf Container-Objekte gehalten werden können.

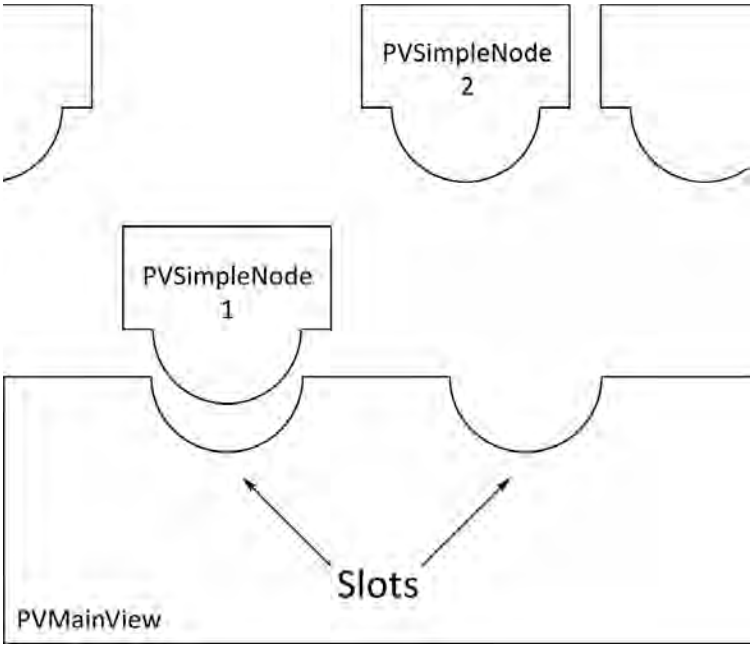

Abbildung 36: Slots für die Gestenerkennung

Ein PVMainView-Objekt hat genau zwei Slots zur Gestenerkennung. Diese werden zu Beginn einer Geste mit den entsprechend zugeordneten Knoten besetzt und so lange dort gehalten, bis eine Geste beendet oder abgebrochen wird.

Im Falle einer Pan-Geste ist nur ein Knoten betroffen (vorausgesetzt sie findet über einem Knoten statt). Der zweite Slot bleibt also leer. Wird eine Pinch-Geste ausgeführt, werden beide Slots besetzt. Dabei kommt der linke Knoten in den ersten, der rechte

Knoten in den zweiten Slot. Je nach vollzogener Geste werden dann unterschiedliche Folgeoperationen ausgeführt. Diese werden in den Kapiteln 3.3.2 und 3.3.3 beschrieben.

3.3.2 Kontext-Menü-Geste

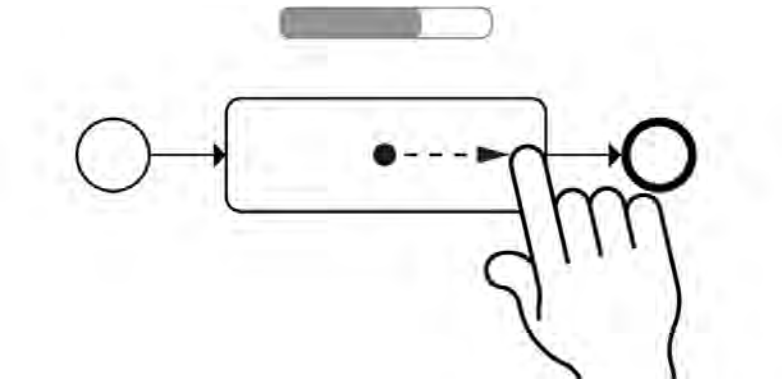

Abbildung 37: Pan-Geste auf einem Aktivitätsknoten

Abbildung 37 zeigt die Ausführung einer horizontalen Pan-Geste nach rechts an einem Aktivitätsknoten. Währenddessen wird eine Fortschrittsanzeige eingeblendet. Mit zunehmendem Abstand zum ersten Berührungspunkt erweitert sich die Anzeige und sie wird immer deutlicher sichtbar. Beendet der Benutzer die Geste, indem er den Finger vom Bildschirm nimmt, wird der aktuelle Fortschritt der Geste ausgewertet. Ist der Fortschrittsbalken unvollendet, wie in Abbildung 37, verschwindet dieser und es wird keine weitere Operation ausgeführt. Hat der Benutzer dabei die Geste so weit ausgeführt, dass der Fortschrittsbalken 100% anzeigt, öffnet sich rechts neben dem Knoten ein Kontext-Menü (siehe Abbildung 38).

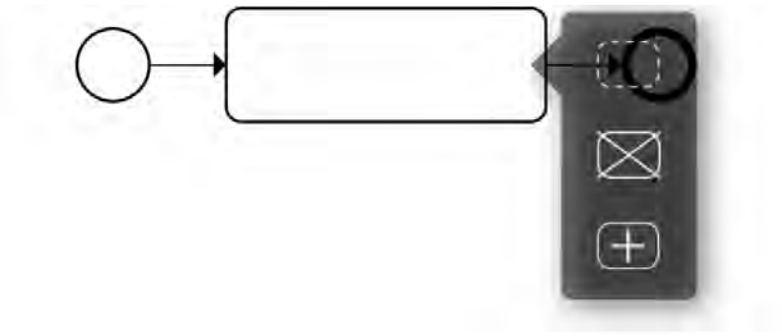

Abbildung 38: Kontext-Menü zum Bearbeiten/Hinzufügen eines Knotens

Die in Abbildung 38 abgebildeten Operationen lauten von oben nach unten wie folgt:

- Knoten reduzieren
- Knoten löschen
- Knoten (rechts) einfügen

Das sogenannte *PVNodeContextMenu* ist eine selbst entwickelte UIView-Komponente. Es ist eine Subview von PVMainView und wird von eben dieser gesteuert. Dabei richtet es sich entsprechend des *PVConnectorOffsets* des betreffenden Knoten je nach Richtung der Geste rechts oder links daneben aus.

Betätigt der Benutzer einen der Buttons, wird über *PVServerInteraction* ein Request erzeugt und an den proView-Server geschickt. Gelingt dieser, wird das veränderte Template neu geladen, eine neue PVMainView erstellt und die Alte dann mit einem Überblendungs-Effekt gegen die Neue ausgetauscht. Funktioniert die Operation nicht, bekommt der Benutzer eine Fehlermeldung angezeigt. Wählt der Benutzer den dritten Button aus, wird er nach dem Namen des neuen Knotens gefragt. Dabei blendet sich die iPad-Tastatur ein und es kann ein entsprechender Name eingegeben werden (siehe Abbildung 39).

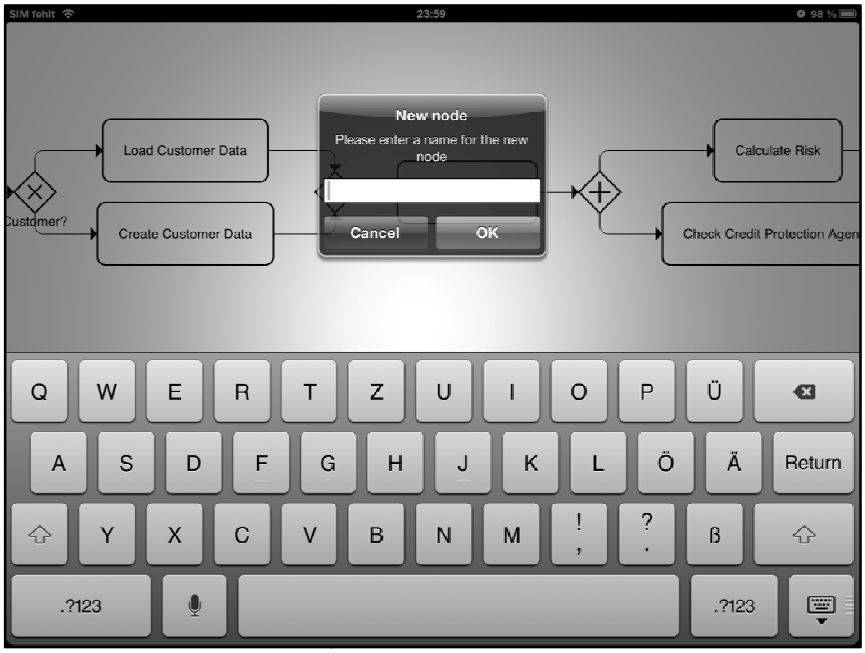

Abbildung 39: Beschriftung eines neuen Knotens

Ein Eingabedialog zur Namensgebung erscheint auch nach der Durchführung der Aggregations-Geste, welche im nächsten Kapitel beschrieben wird.

3.3.3 Aggregations-Geste

Während die Geste für das Kontext-Menü mit einem Finger ausgeführt werden kann, benötigt diese Geste zwei Finger. Dabei müssen diese jeweils von einem Knoten starten und aufeinander zubewegt werden. Die Knoten müssen auf gleicher vertikaler Höhe liegen, damit die Geste erkannt wird. Dies verhindert bereits im Voraus einige Kombinationen von Knoten, die gar nicht miteinander aggregiert werden können.

Abbildung 40 zeigt eine beginnende Aggregations-Geste. Die betroffenen Randknoten erhalten Klammern und darüber wird eine Fortschritts-Anzeige in Form zweier transparenter Boxen (als Repräsentation der betroffenen Knoten) eingeblendet. Je weiter die Finger aufeinander zubewegt werden, desto weiter nähern sich die Boxen in der Fortschritts-Anzeige einander an.

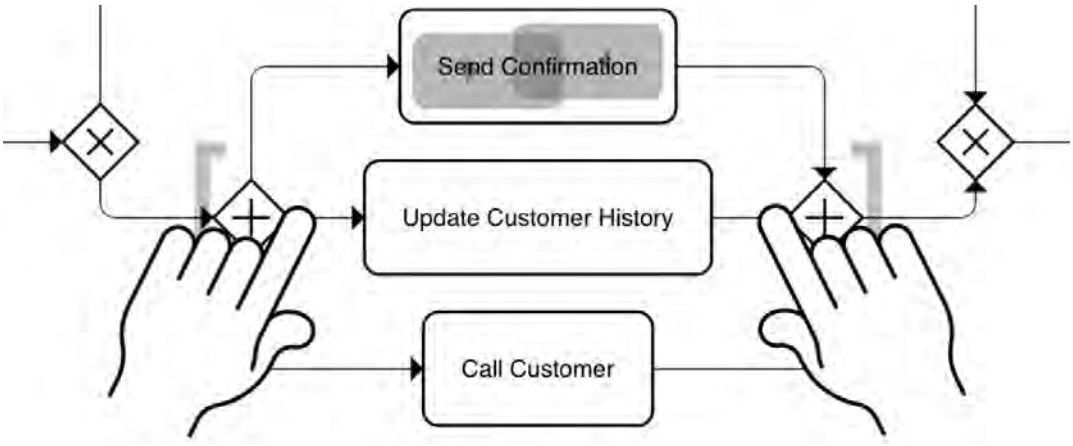

Abbildung 40: Aggregations-Geste

Unterschreitet der Abstand zwischen beiden Fingern einen Schwellenwert, sind auch die Boxen in der Fortschritts-Anzeige miteinander verschmolzen und blinken. Dieser Zustand ist in Abbildung 41 zu sehen.

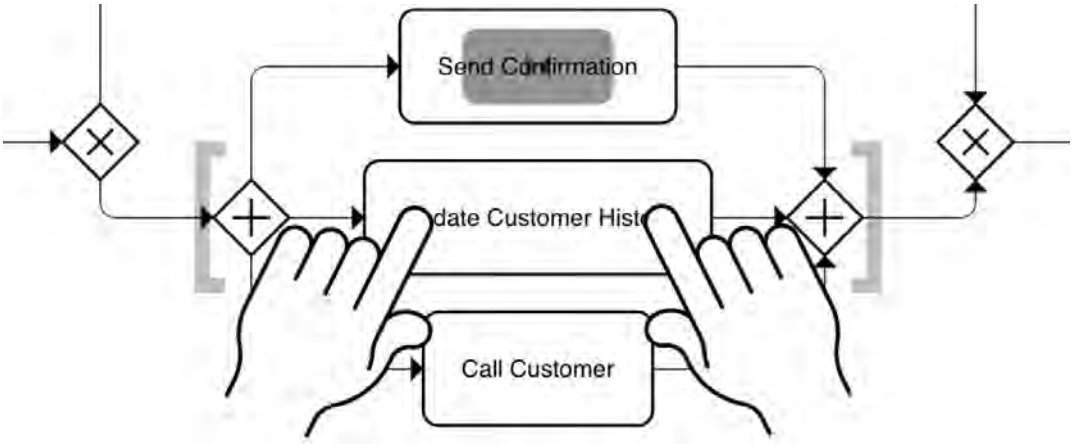

Abbildung 41: Aggregations-Geste erreicht „scharfen“ Zustand

Löst der Benutzer nun die Finger vom Display, öffnet sich ein Popup-Fenster, das die Eingabe eines Namens für den aggregierten Knoten verlangt. Mit dem Cancel-Button lässt sich die Operation abbrechen.

Da der *Pinch Gesture Recognizer* immer nur die aktuelle Position der Finger zurückgibt, muss zu Beginn der Geste für beide Finger der erste Berührungspunkt gespeichert werden.

Für besonders kleine, direkt nebeneinander liegende Knoten wird der Schwellenwert etwas verringert. Dadurch kann in diesem Fall selbst mit kleineren Bewegungen trotzdem noch eine vollständige Aggregations-Geste ausgeführt werden.

4. Aufbau von proViewMobile

Nachdem in Kapitel 4 bereits einige Klassen und deren Methoden beschrieben sind, soll in diesem Kapitel eine Übersicht über die wichtigsten Klassen und ihre Einordnung in die Gesamtstruktur von *proViewMobile* gegeben werden. Dabei werden in Kapitel 4.1 die einzelnen Programm-Module mit ihren Komponenten vorgestellt. Kapitel 4.2 beschreibt die Beziehungen der in *proViewMobile* verwendeten Klassen untereinander sowie deren Hierarchien und Datenstrukturen.

4.1 Programm-Module

Vor einer detaillierten Sicht auf die Beziehungen einzelner Bestandteile, ist es für das Verständnis einer komplexen Struktur oftmals empfehlenswert, einen Überblick über das „große Ganze“ zu bekommen. Daher steht an dieser Stelle zunächst eine Übersicht aller Programm-Module von *proViewMobile* (siehe Abbildung 42).

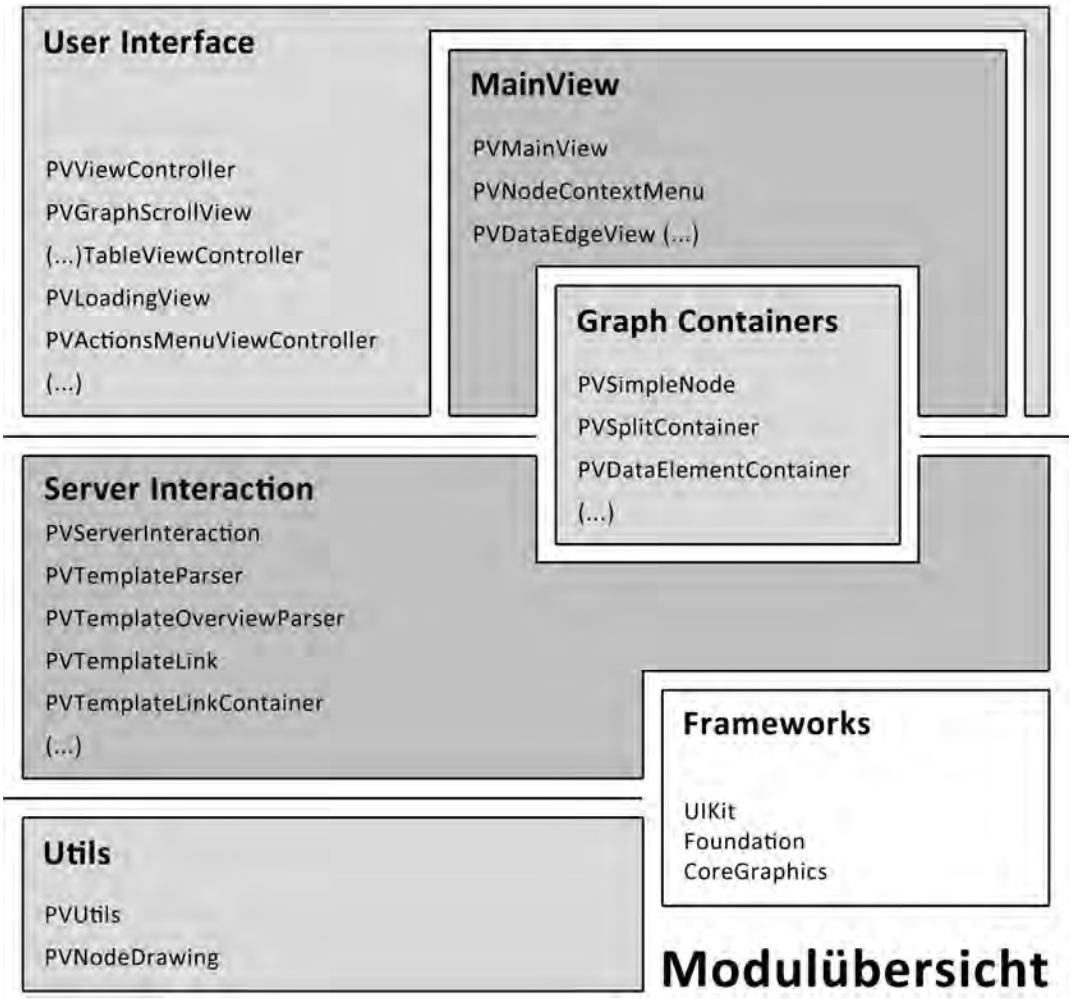

Abbildung 42: Übersicht über alle Programm-Module in proViewMobile

4.2 Klassenbeziehungen und -hierarchien

Eine der wichtigsten Klassen in *proViewMobile* ist der *PVGraphContainer*. In Abbildung 43 wird der *PVGraphContainer* in Beziehung zu den Klassen gesetzt, die sein Protokoll implementieren.

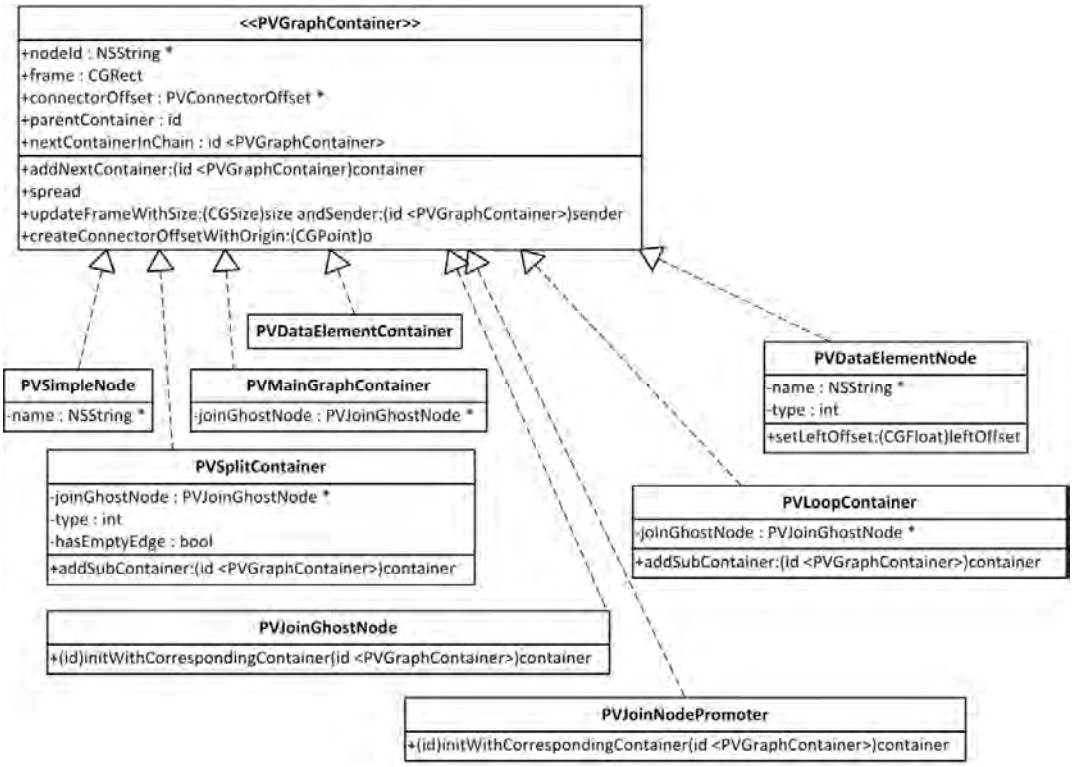

Abbildung 43: PVGraphContainer und implementierende Klassen

Die in Abbildung 43 abgebildeten Klassen implementieren alle das *PVGraphContainer*-Protokoll. Dies ist sinnvoll, da es sich jeweils um Datenklassen handelt, die miteinander interagieren. Bei den Controllerklassen ergibt sich dagegen ein völlig anderes Bild. Sie müssen untereinander nur schwache Verbindungen aufbauen, da sich ihr Aufgabenbereich klar abstecken lässt. Dafür übernehmen sie eine Vielzahl an Funktionen, die entsprechende Implementierungen anderer Protokolle erfordern. Abbildung 44 zeigt die beiden wichtigsten Controllerklassen in Bezug zu ihren Protokollen.

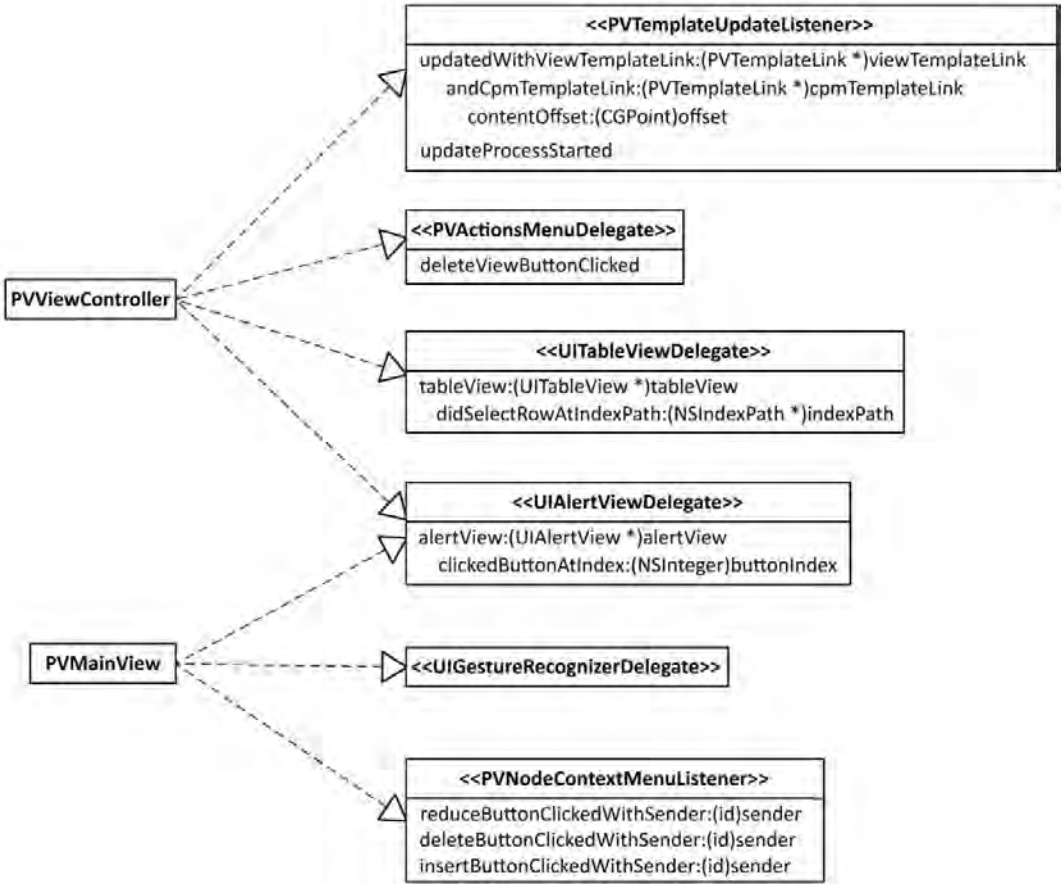

Abbildung 44: PVViewController und PVMainView als Delegates und Listener

Die Verbindung zwischen *PVViewController* und *PVMainView* erfolgt unter anderem über den *PVTemplateUpdateListener*, der von *PVViewController* implementiert wird (siehe Abbildung 45).

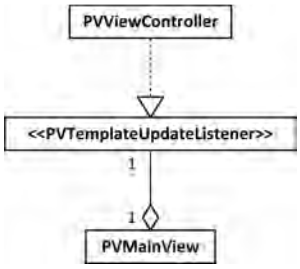

Abbildung 45: Beziehung zwischen PVViewController und PVMainView

Abbildung 46 macht die komplexe Struktur an Verknüpfungen bezüglich der Klasse PVMainView und ihrer Komponenten in Form eines UML-Klassendiagramms sichtbar. Dabei werden Assoziationen (z. B. bezüglich *PVMainView* und *PVDataEdgeView*), Vererbung (z. B. bezüglich *UIPanGestureRecognizer* und *UIGestureRecognizer*) und Implementierung (z. B. bezüglich *PVMainGraphContainer* und *PVGraphContainer*) dargestellt. Das Fehlen von Kardinalitäten ist in diesem Fall gleichbedeutend mit der Kardinalität Eins. Hierbei ist deutlich zu erkennen, dass es sich bei dieser Klasse eigentlich um einen *View Controller* und keine reine *View* handelt. Diese Problematik wird in Kapitel 5 entsprechend diskutiert.

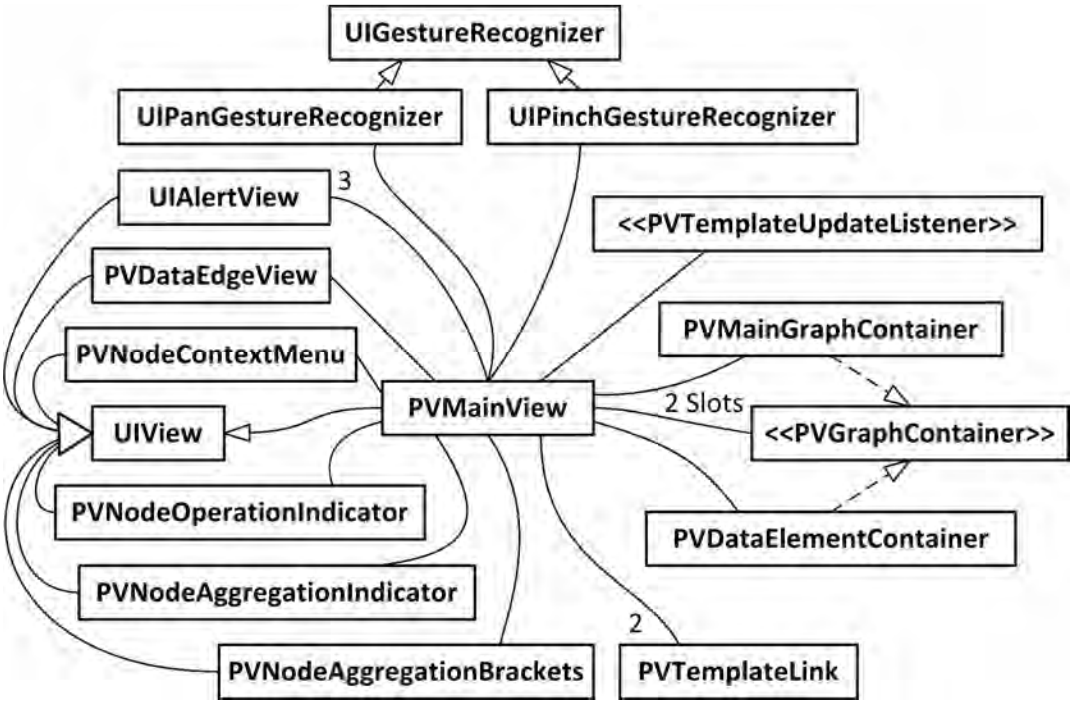

Abbildung 46: PVMainView mit Klassenbeziehungen

5. Probleme und Bewertung

Diese Arbeit wäre wohl eine große Ausnahme, wenn nicht im Laufe des Entwicklungsprozesses irgendwann ein Problem auftauchen würde. Tritt allerdings ein schwerwiegendes Problem erst am Ende der Entwicklung auf, kann dies gravierende Änderungen an bestehenden Konzepten zur Folge haben.

Dadurch, dass eine Testmöglichkeit auf dem iPad erst zu einem fortgeschrittenen Entwicklungsstand möglich war, entstand diese Arbeit zu über 90% unter Zuhilfenahme des iPad-Simulators der Entwicklungsumgebung Xcode 4. Doch was der Simulator nicht in ausreichendem Maße simulieren kann, ist die tatsächliche Speicherverwaltung des mobilen Endgeräts.

Die zum Zeitpunkt des Testbeginns verwendete UI-Struktur führte beim Testen großer Prozessgraphen zu regelmäßigen Abstürzen. Dabei war allein die Anzeige der UI-Elemente bereits so speicherintensiv, dass diese von starkem Ruckeln beim Scrollen begleitet wurde und ab einer gewissen Anzahl von Knoten auf dem Display eine Speicherüberlauf-Warnung in der Konsole hervorrief, meistens direkt gefolgt von einem Programmabsturz.

Die anschließende Verwendung der Analyse-Tools zur Speicherüberwachung ergab keinen Anhaltspunkt, wo am besten nach dem Fehler zu suchen wäre. Schließlich war der nächste logische Schritt die Umstrukturierung der zu diesem Zeitpunkt noch autonomen *UIView-Widgets*. Diese waren bis dato als eigenständige UI-Elemente entwickelt worden, die über Protokolle die Interaktion mit den restlichen UI-Klassen vollzogen hatten. Dabei waren sowohl PVMainView als auch PVViewController als Listener bei jedem Container-Objekt registriert. Zusätzlich war jeder Container mit einem eigenen *Gesture Recognizer* ausgestattet, der über die im Protokoll definierte Schnittstelle Informationen an die entsprechenden Controller übermittelte.

Die rein konzeptionell und algorithmisch gut funktionierende Schichtenstruktur der Container war im Zusammenhang mit der Darstellung auf dem Bildschirm eine ungeeignete Lösung. Um dem Problem letztlich aber doch noch Herr zu werden, war ein Kompromiss nötig, der in Abbildung 47 verdeutlicht werden soll.

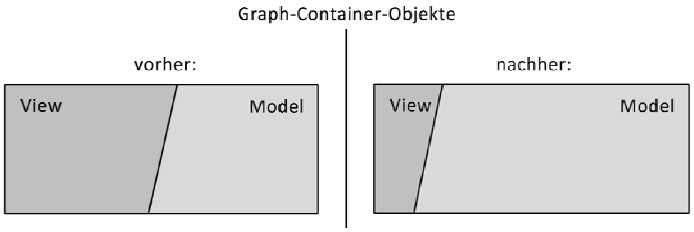

Abbildung 47: Implementierung des PVGraphContainer-Protokolls vorher und nachher im Vergleich

In der Abbildung schematisch zu sehen ist der notwendige Wegfall der Eigenschaft als UIView-Element bei jeder Klasse, die das PVGraphContainer-Protokoll implementiert.

Trotz dieses Wegfalls konnten die entsprechenden Klassen aber ihre *strukturelle* Komponente beibehalten. Dadurch konnten sie weiterhin als Datenobjekt für das Zeichnen des Prozessgraphen verwendet werden. Gleichzeitig wurde das Zeichnen des kompletten Prozessgraphen an eine einzige View-Klasse delegiert, die *PVMainView*. Nachdem die *Gesture Recognizer* nicht mehr in der bisherigen Form weiterverwendet werden konnten, musste die Klasse folglich auch diese Objekte verwalten. Und schließlich kam auch noch das An- und Abwählen von Knoten zum „Aufgabenbereich“ der *PVMain-View-Klasse* hinzu.

Aufgrund des späten Auftretens des Fehlers und der drastischen und zeitintensiven „Umbauarbeiten“ am Programmcode konnte letztlich auf die Einhaltung des MVC-Musters (siehe Kapitel 2.3.1) an dieser Stelle keine Rücksicht mehr genommen werden. Und so verbleibt die PVMainView vorerst als scheinbar reine View-Komponente „im falschen Gewand“.

Konsistent wäre dagegen die Aufspaltung dieser Klasse in eine „echte“ *PVMainView* und einen dazugehörigen *PVMainViewController*, was an dieser Stelle also einen ersten Anknüpfungspunkt für eine potentielle Weiterentwicklung des Prototyps ergibt.

6. Zusammenfassung

Diese Arbeit beschreibt die Implementierung einer auf Multi-Touch- und Gestensteuerung basierten Prozess-Modellierungs-Anwendung für das iPad. Nach einer Erklärung grundlegender Konzepte des proView-Projekts und der App-Entwicklung für iOS wird die Vorgehensweise während der Entwicklung entlang der Anforderungspunkte vorgestellt. Dabei wird der Entstehungsprozess in allen Phasen durchleuchtet und erklärt. Es werden wichtige Komponenten wie das PVGraphContainer-Protokoll und die PVMainView-Klasse genauer untersucht und in Bezug zum Gesamtsystem gesetzt. Außerdem werden die verwendete Gestensteuerung, die Entwicklung des User Interface und die Client-Server-Kommunikation beschrieben. Zuletzt werden Probleme während der Test-Phase und deren Bewältigung erläutert und die damit verbundenen Konsequenzen kritisch bewertet.

Mit dem Abschluss dieser Arbeit wird das „Software-Repertoire“ des proView-Projekts um eine Anwendung erweitert. Diese Anwendung zeichnet vor allem die gestenbasierte Bedienung auf einem mobilen Endgerät aus. Mit ihr ist ein Grundstein zu einer „spielerischen“ Prozessgraph-Modellierung gelegt, die aufgrund der Mobilität auch losgelöst vom normalen Bürobetrieb stattfinden und ein wenig frischen Wind in das manchmal vielleicht eher trockene Thema Prozessmanagement bringen kann.

Literaturverzeichnis

[1] Hegarty, Paul „Developing Apps for iOS, Lecture 2 September 2011“ in iTunesU, 2011 (https://itunes.apple.com/de/itunes-u/developing-apps-for-ios-sd/id395631522)

[2] Kolb, Jens „Personalized and Updatable Process Visualizations“, Institut DBIS auf www.uni-ulm.de, 2012

[3] Kolb, Jens and Reichert, Manfred and Weber, Barbara (2012) Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In: S-BPM ONE 2012, Vienna, April 2012, CCIS 284, Springer, pp. 237-251.

[4] Kolb, Jens and Rudner, Benjamin and Reichert, Manfred (2012) Towards Gesture-based Process Modeling on Multi-Touch Devices. In: 1st Int'l Workshop on Human-Centric Process-Aware Information Systems (HC-PAIS'12), Gdansk, Poland, 25th June 2012, LNBIP 112, Springer, pp. 280-293.

[5] Kolb, Jens and Hübner, Paul and Reichert, Manfred (2012) Model-Driven User Interface Generation and Adaptation in Process-Aware Information Systems. Technical Report UIB-2012-04, University of Ulm.

[6] Kolb, Jens and Hübner, Paul and Reichert, Manfred (2012) Automatically Generating and Updating User Interface Components in Process-Aware Information Systems. In: 20th International Conference on Cooperative Information Systems, Rome, Italy, September 12-14, 2012, LNCS 7565, Springer, pp. 444-454.

[7] Kolb, Jens and Kammerer, Klaus and Reichert, Manfred (2012) Updatable Process Views for User-centered Adaption of Large Process Models. In: 10th Int'l Conference on Service Oriented Computing (ICSOC'12), Shanghai, China, November 12-15, 2012, LNCS 7636, Springer, pp. 484-498.

[8] Kolb, Jens and Kammerer, Klaus and Reichert, Manfred (2012) Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: Demo Track of the 10th Int'l Conf on Business Process Management (BPM'12), Tallinn, Estonia, September 3-6, 2012.

[9] Kolb, Jens and Reichert, Manfred (2012) Supporting Business and IT through Updatable Process Views: The proView Demonstrator. In: Demo Track of the 10th Int'l Conference on Service Oriented Computing (ICSOC'12), Shanghai, China, November 12-15, 2012.

[10] Reichert, Manfred and Kolb, Jens and Bobrik, Ralph and Bauer, Thomas (2012) Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: 27th ACM Symposium On Applied Computing (SAC'12), 9th Enterprise Engineering Track (EE'12), Trento, Italy, March 25-29, 2012, ACM Press, pp. 1653-1660.

[11] Rossini, Nicola. „Reinterpreting Gesture as Language“, IOS Press US, 2012

[12] Schegloff, E.A. „On some gestures' relation to talk.“ In Structures of Social Action: Studies in Conversation Analysis. J.M. Atkinson and E. J. Heritage, eds. Pp. 266-296. Cambridge: Cambridge University Press. 1984