UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

App-Entwicklung für Apple-iOS
am Beispiel eines Head-mounted Displays

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von
Grégory Catellani
gcatellani@uni-koblenz.de
Version 1.0

Erstgutachter:	Prof. Dr. Dieter Zöbel, Universität Koblenz-Landau, Institut für Softwaretechnik
Zweitgutachter:	Dr. Merten Joost, Universität Koblenz-Landau, Institut für Integrierte Naturwissenschaften – Abteilung Physik

Frankfurt am Main, im Juni 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Frankfurt am Main, den . . . . . .

"Being the richest man in the cemetery doesn't matter to me ... Going to bed at night saying we've done something wonderful... that's what matters to me."

[Steven Paul Jobs im The Wall Street Journal, 25. Mai 1993]

Inhaltsverzeichnis

Typographische Konventionen

Die nachstehenden Textvorlagen werden im Dokument genutzt um Quellcode und Syntaxelemente von normaler deutschen Sprache zu trennen. Die Konventionen werden allerdings nicht in Überschriften verwendet, wo eine Unterscheidung nicht nötig ist.

Palatino Linotype – 11pt.: Normaler Text

Palatino Linotype – 11pt.: Hervorzuhebende Wörter

Palatino Linotype – 11pt.: Fremdwörter sowie Namen von Firmen, Projekten, Notationen, Programmiersprachen, Frameworks, Computerprogramme, Dateiformate, oder Dokumenten wie Spezifikationen und Publikationen im Fließtext

PALATINO LINOTYPE – 11PT.: Nachnamen von Fachleuten im Fließtext

`Courier New – 11pt.:` Quellcode sowie Namen von Klassen, Objekten, Funktionen, Methoden, Nachrichten, Variablen und Attributen im Fließtext

Teil I.

Einführung in die Thematik

1. Einleitung in den Bereich des wearable computing

Computer unterstützen Menschen zunehmend in Alltagssituationen. Ihre fortwährend voranschreitende Miniaturisierung erlaubt die Erschließung weiterer Einsatzmöglichkeiten und trägt zu einer wachsenden Bedeutung und Verbreitung in der Gesellschaft bei. Längst sind solche Systeme in vielerlei Alltagsgegenständen Realität und ihre Mobilität spielt eine immer größer werdende Rolle: von Laptops über Smartphones und Tablets hin zu am oder im Körper tragbaren Systemen (engl. *wearable computing*) wie Hörgeräte und Pulsuhren, unterstützen sie den Menschen aktiv und kontextsensitiv in ihrem Alltag.

Als ein Teilgebiet des *wearable computing* gilt die Entwicklung von *Head-mounted Displays* (kurz HMD). Das sind Helme oder Brillen die dem Benutzer auf einer oder mehreren Darstellungsflächen computergenerierte Bilder (*virtual reality*) oder mit Zusatzinformationen versehene Aufnahmen der Umgebung (*enhanced reality*) präsentieren. Aktuell sind *HMDs* welche *LC-Displays* nutzen weit verbreitet, innovativer sind solche welche das Bild direkt auf die Netzhaut des Benutzers projizieren. Neuste Entwicklungen deuten auf weitere Miniaturisierung des Konzepts, hin zu Kontaktlinsen mit integriertem Bildschirm[1]. Für die Sammlung dazustellender Daten werden eine Vielzahl Sensoren hinzugezogen, wie *Head-Tracker* oder *GPS*. Auch durch Erweiterungen im Bereich der Bedienbarkeit – zum Beispiel durch Spracheingabe über ein Mikrofon – und die Steigerung des Immersionsgefühls – zum Beispiel durch stereoskopische Bilder oder integrierte Kopfhörer – wandeln sich *HMDs* von einfachen Anzeigen hin zu stetig besser ausgerüsteten Geräten mit ständig wachsenden Einsatzgebieten. Längst werden *HMDs* in einer Vielzahl von Feldern benutzt. Vom Militär für die Ausbildung von Soldaten auf simulierten Schlachtfelder und Luftwaffenpiloten mit *HMDs* in Helmen. Als Konzepte in der Medizin zur Diagnose von Patienten oder der Unterstützung in der Chirurgie. Im Sport mit der Anzeige von Zusatzinformationen im Helm von Formel-1 Fahrern[2] oder in Skibrillen[3]. Oder in der Spiele- und Multimediabranche für den immersiven Konsum von Computerspielen und (bewegtem) Bildmaterial.

Die vielfältigen Einsatzmöglichkeiten von *HMDs* und die ständig sinkenden Kosten bei gleichzeitigem Leistungszuwachs von Komponenten für diesen Einsatzzweck, führten das *Institut für Integrierte Naturwissenschaften der Universität Koblenz-Landau*, zur Entwicklung eines *HMDs* auf Basis von *Apple iOS*-Mobilgeräten mit hochauflösenden *Retina Displays*. Kern des Ansatzes stellt die hohe Pixeldichte der Displays dar, welche gepaart mit Kondensorlinsen dem Benutzer ein hohes Immersionsgefühl vermitteln, während andere Systeme lediglich eine Bildschirmfläche bei gerin-

[1] `http://spectrum.ieee.org/biomedical/bionics/augmented-reality-in-a-contact-lens` , Stand: 25.05.2012

[2] `http://findarticles.com/p/articles/mi_m0EIN/is_2002_Sept_30/ai_92199883`, Stand: 25.05.2012

[3] `http://www.reconinstruments.com/products/recon-ready`, Stand: 25.05.2012

ger Auflösung auf mehreren Metern vor dem Auge des Betrachters simulieren[4]. Zusätzlich bergen Geräte wie das *iPod Touch*, *iPhone* oder *iPad* hohes Potential, durch die Rechenleistung und die Vielfalt integrierter Sensoren, darunter u.a. für *HMDs* interessant, das Gyroskop, der Beschleunigungssensor, die Kamera, diverse Vernetzungsmöglichkeiten oder die Audioausgabe. Diese Charakteristika und die sich hierdurch ergebenden Möglichkeiten lassen sich in anderen Konzepten für Videobrillen[5][6] oft nur durch einen hohen Preise erkaufen, während zwei *iPod Touch* der vierten Generation samt nötigen Komponenten für das Gestell, aktuell im niedrigen dreistelligen Bereich liegen. Außerdem sprechen die hohe Verbreitung von *Apple*-Mobilgeräten für ihren Einsatzzweck in einem *HMD*. Mit geeigneter, an die diverse Gerätemodelle angepasster Software steigt die Wiederverwendung der Lösung, so könnte später ein neueres, potenteres Mobilgerät mit wenig Aufwand eingesetzt werden und der Benutzer kann seine eigenen Mobilgeräte im Gestell des *HMDs* nutzen, wodurch die Kosten zusätzlich sinken.

1.1. Problemstellung und Lösungsidee

Im Vordergrund dieser Arbeit steht der Entwurf und die Erstellung einer Anwendung für das Betriebssystem *Apple iOS 5*, als softwaretechnischen Beitrag zur Entwicklung eines *HMD* auf Basis von *Apple*-Mobilgeräten mit hochauflösendem *Retina Display*. Das RollerCoaster2000-Projekt[7] soll zur Darstellung des grafischen Inhalts genutzt werden, aus Gründen einer freien Lizenz, frei zugänglichen Quellcodes sowie einer effizienten *OpenGL-ES*-Portierung. Dabei soll ein leichter Austausch oder die Erweiterung dieses grafischen Grundgerüsts gegeben sein um die Einsatzmöglichkeiten des *HMD* zu erhöhen. Über die reine Nutzung als Darstellungsmedium soll die erstellte Anwendung die vielfältigen, sensorischen Fähigkeiten und Vernetzungsmöglichkeiten aktueller *Apple*-Endgeräte nutzen um mehrere Geräte zu einer Datenbrille zu verbinden und die Kopfbewegungen des Benutzers auszuwerten, in der virtuellen Szene abzubilden und zwischen den Geräten synchron zu halten. Zur leichten Orientierung des Benutzers während der Bedienung soll außerdem auf die integrierte Gerätekamera zugegriffen werden.

Um die Realisierung einer den Kriterien gerecht werdenden Anwendung zu ermöglichen wird in dieser Arbeit in den Bereich der *iOS*-Entwicklung eingeführt. Zuerst erfolgt eine Einleitung in die historischen Entwicklungen auf dem Gebiet der mobilen Entwicklung für *Apple*-Mobilgeräte, von der Entstehung des objektorientierten Paradigmas hin zur heutigen Ausprägung des *iOS-SDK* (siehe Kapitel 2). Letzteres ergibt sich dabei aus der Summe einer Vielzahl, stark kohärenter Komponenten. Dazu gehören die Grundsteine einer jeden *iOS*-Applikation, die objektorientierte, dynamische und streng typisierte Sprache *Objective-C* sowie den durch *Apple* an die eigenen Bedürfnisse angepassten *LLVM*-Compiler und die von Apple angepasste *LLVM*-Laufzeitumgebung (siehe Teil II). Zudem sind das *Cocoa Framework* und die Entwicklungsumgebung *Xcode*, die zur Erstellung mobiler Anwendungen von Apple entwickelt wurden, Gegenstand dieser Arbeit (siehe Teil III). Anhand der gewonnenen Kenntnisse im Bereich der mobilen Anwendungsentwicklung

[4] `http://www.siliconmicrodisplay.com/st1080.html`, Stand: 25.05.2012
[5] `http://www.sony.de/product/head-mounted-display/hmz-t1`, Stand: 25.05.2012
[6] `http://cinemizer.zeiss.com/cinemizer-oled/de_de/cinemizer-oled.html`, Stand: 25.05.2012
[7] `http://plusplus.free.fr/rollercoaster/index.html`

werden Anforderungen an eine Anwendung für ein *HMD* erhoben und die Erstellung einer, diesen Kriterien genügenden, Software beschrieben (siehe Teil IV).

2. Historische Einführung in die Sprache Objective-C und das iOS-SDK

Seit dem Ende der 50er Jahre und durchgehend bis in die späten 80er Jahre des letztens Jahrhunderts setzten fast alle Anwendungsentwicklungen auf den Ansatz der prozeduralen Programmierung als Erweiterung des imperativen Paradigmas (vgl.[Pro09] S.10ff). Durch die darin enthaltene Idee der Programmstrukturierung erhoffte man sich, immer weiter anwachsende Entwicklungen in kleinere und besser zu handhabende Teile zerlegen zu können (vgl. [Seb11], S.23). Aber auch dieses Paradigma geriet Ende der 80er Jahre durch den stetig ansteigenden Umfang der Anwendungen an seine Grenzen, eine wirksame Wiederverwendung von Code war kaum möglich (vgl. [Seb11], S.23). Mit der ansteigenden Programmkomplexität stieg die Anzahl benötigter Prozeduren und Kontrollstrukturen, was den Quellcode für die Entwickler kaum überschaubar und anfällig für Fehler machte.

Als eine Lösung für diese Probleme entstand der Ansatz der objektorientierten Programmierung, welche in Aussicht stellte, auch große Projekte handhabbar zu unterteilen und die Codewiederverwendung erheblich zu vereinfachen. Ende der 60er Jahre des letzten Jahrhunderts schlossen sich die Arbeitskollegen ALAN KAY, DAN INGALLS und ADELE GOLDBERG zusammen um eine der ersten, objektorientierten Programmiersprachen am *Xerox PARC* Forschungszentrum zu entwickeln, die Sprache *SmallTalk*. Für die Firma *Xerox* entwickelten sie gleichzeitig weitere, für damalige Verhältnisse bahnbrechende, Ansätze wie die erste graphische Benutzungsoberfläche, die auf Basis der neuen Sprache basierte. Diese beiden zusammenhängenden Entwicklungen galten der Verfolgung eines ehrgeizigen Ziels, der der Fertigstellung und Marktdurchsetzung der ersten *Personal Computer*. Diese Vision soll ALAN KAY 1970, also zu Beginn seiner Zeit bei *Xerox*, auf die Frage „Was wird ihre wichtigste Errungenschaft hier sein?" formuliert haben. Zu dieser Zeit hatte auch die Firma *Apple* mit STEVE JOBS bei einem Besuch bei *Xerox* den ersten Kontakt mit den neuen Konzepten. Allerdings war STEVE JOBS dermaßen von der graphischen Oberfläche fasziniert, dass er erst nach 1988, mit seinem damals frisch gegründeten Unternehmen *NeXT,* den objektorientierten Ansatz für sich entdeckte. (vgl. [Seb11], S. 23ff)

Der größte Nachteil der Sprache *SmallTalk* war ihre Ausführungsgeschwindigkeit. Darin geschriebener Code wurde nicht direkt ausgeführt sondern in *Bytecode* übersetzt, der anschließend in einer virtuellen Maschine ausgeführt wurde. Dieser Ansatz, der später u.a. von den Entwicklern der Firma *Sun* als Konzept für ihre Programmiersprache *Java* aufgegriffen wurde, hatte den entscheidenden Nachteil auf den damaligen Rechnern langsam zu laufen.

Die Nachteile von *SmallTalk*, aber auch ihre fundamentalen Neuerungen, hatte die Entwicklung weiterer, objektorientierten Sprachen zur Folge. Nicht nur die Programmiersprache *C++* von BJARNE STROUSTRUP wurde in Anlehnung an *SmallTalk* entwickelt, sondern auch die Sprache *Objective-C* von BRAD COX und TOM LOVE. Beide Ansätze verfolgten die Idee die Vorzüge von *SmallTalk*

mit den Vorteilen von *C* zu kombinieren, mit ganz unterschiedlichen Ergebnissen. Bei der Entwicklung von *C++* stand die Effizienz im Vordergrund, was zu einer eher statischen, objektorientierten Erweiterung von *C* führte. Beim Entwurf von *Objective-C* war das Leitmotiv hingegen, weniger die Effizienz als vielmehr der Gedanke, sich den dynamischen Aspekten, also der Flexibilität, von *SmallTalk* anzunähern. Die erste Spezifikation von *Objective-C* erschien 1986 mit dem Buch „*Object-Oriented Programming, An Evolutionary Approach*" von BRAD COX und TOM LOVE[1]. (vgl. [Seb11], S.23ff)

Als STEVE JOBS *Apple* verließ und 1988 die Firma *NeXT* gründete, lizensierte er *Objective-C* von der Firma *StepStone*, der 1983 gegründeten, ursprünglich *Productivity Products International* (*PPI*) genannten, Software-Entwicklungsfirma von BRAD COX und TOM LOVE. Mit *NeXT* wollte STEVE JOBS seine Vision eines *Personal Computers* ganz nach seinen eigenen Vorstellungen realisieren. Hierzu wurden der *GCC*-Compiler erweitert und die ersten Versionen der *AppKit* und *Foundation Kit* Bibliotheken entwickelt, die sich noch heute, wenngleich in stark abgeänderter Form, in *Mac OS X* und *iOS* bzw. in deren *SDKs* wiederfinden[2]. Zuerst versuchte *NeXT* selbst Computer herzustellen, auf Basis des mit *Objective-C* geschriebenen, graphischen Betriebssystems *NeXTSTEP*. Nachdem das Hardware-Projekt scheiterte konzentrierte sich *NeXT* vollständig auf die selbst entwickelte Software, die Bibliotheken und die dazugehörigen Entwicklungswerkzeuge. (vgl. [Seb11], S.26)

1996 kaufte Apple die Firma *NeXT* und setzte STEVE JOBS als Firmenberater ein, der später die Rolle als Geschäftsführer übernehmen sollte. Noch im selben Jahr stellte *Apple* die erste Version des Betriebssystems *Mac OS X* auf Grundlage von *NeXTSTEP* fertig. Hierzu gehörten auch die Sprache *Objective-C* und die Entwicklungswerkzeuge *Project Builder* und *Interface Builder*. Aus *Project Builder* wurde später die Entwicklungsumgebung *Xcode* und der *Interface Builder* wurde mit der *Xcode* Version *4* in die Entwicklungsumgebung integriert.

Einem weiterer Anstieg der Sprachpopularität kam die 2006 durch *Apple* vorgestellte Version *2.0* zu Gute. Der späte Erscheinungstermin erlaubte den *Apple*-Ingenieure aus den Fortschritten und Fehlern auf dem Gebiet des objektorientierten Sprachparadigmas zu lernen. Neben einer verbesserten Performance bot die, bis heute aktuelle, Revision der Sprache u.a. einen *Garbage Collector*[3] und erweiterte Sprachmittel wie u.a. *Properties*. (vgl. [Seb11], S.26f)

Nach der Veröffentlichung des ersten und der Vermarktung des zweiten *iPhone*-Modells im Juni 2007 beziehungsweise Juni 2008 sowie dem damit einhergehenden, raschen Zugewinn an Popularität von Mobilgeräte, veröffentliche Apple am 11. Juli 2008 das *iPhone SDK*. Seit dem stehen Entwicklern mit *Objective-C 2.0* und einer auf die Bedürfnisse des Einsatzes im Mobilbereich zugeschnittenen Version der *Cocoa Bibliothek*, dem Entwicklungs-*Framework Cocoa Touch*, alle Mittel zur Erstellung von Anwendungen zur Verfügung, die auch *Apple* für ihre Programme nutzt. Mit der Vorstellung der ersten *iPad*-Generation im Januar 2010 wurde das Betriebssystem *iPhone OS* und das *iPhone SDK* in *iOS* respektive *iOS-SDK* umbenannt. Bis heute fungiert *Objective-C 2.0*, neben der ihr zugrunde liegenden prozeduralen Sprache *ANSI C*, auf der *iOS*-Plattform als exklusives Sprachmittel (vgl. [Seb11], S. 27).

[1] Siehe `http://books.google.de/books?id=deZQAAAAMAAJ&dq=editions:ISBN0201548348`, letzter Zugriff: 04.04.2012 um 12:21Uhr

[2] Klassen deren Namen mit dem Präfix `NS` anfangen, das für *NeXTSTEP* steht, sind hierbei die prominentesten Zeugen für den historischen Erhalt einiger, ursprünglicher Sprachmittel.

[3] In *Mac OS X* integriert, aber bis heute nicht in *iOS* – aktuell Version *5*.

Das Gesamtkonzept aus hochintegriertem *System-on-Chip* (*SoC*), knappen und effizienten Programmen, hilfreichen Handbüchern zum Design von Programmen und Benutzungsoberflächen, kategorischem Weglassen effizienzmindernder Funktionalität sowie raffinierten Hard- und Software-Tricks, lässt die Nachfrage nach Mobilgeräten ständig steigen (vgl. [Ben12], S. 102ff). Dieser Erfolg dürfte ein Garant für zukünftige Neu- und Weiterentwicklung auf dem Gebiet der mobilen Systeme sein. Somit sind auf absehbare Zeit auch mit weiteren Neuerungen im Bereich des *iOS-SDK* und der Sprache *Objective-C* zu rechnen.

Teil II.

Einführung in Objective-C

3. Die objektorientierte Sprache Objective-C

3.1. Einleitung

Bevor die Funktionalität des *iOS-SDKs* vorgestellt und eine Anwendung für die Zwecke eines *Head-mounted Displays* auf Basis von *Apple*-Mobilgeräten entwickelt werden kann, muss die, primär zur Entwicklung von auf dem *Apple Cocoa API* basierenden Anwendungen, verwendete Programmiersprache *Objective-C* mit ihren Konzepten und Sprachmitteln festgehalten werden (vgl. [Seb11], S.15). Dieses Kapitel beschreibt die Hauptaspekte und Sprachelemente der objektorientierten Sprache *Objective-C 1.0* und der Sprachrevision *2.0*.

Die folgenden Unterkapitel bestehen aus einer Einführung in die Entwicklung mit *Objective-C*, angeführt von einem einfachen Notationsbeispiel (siehe Unterkapitel 3.1.1), gefolgt von einer Beschreibung der Sprachkonzepte und Notationselemente (siehe Unterkapitel 3.2). Die Beispiele in den jeweiligen Unterkapiteln basieren auf dem Notationsbeispiel oder sind Auszüge davon.

Anzumerken ist, dass die Struktur der Unterkapitel zur Beschreibung der Sprachelemente sich nicht an der Klassifizierung der Elemente aus den offiziellen *Apple* Dokumentationen orientiert (siehe u.a. [App11b]). Stattdessen findet die Einführung der Sprachkonzepte grob am Programmverlauf ausgerichtet statt. So wird zuerst der prozedurale *C*-Unterbau der Sprache erläutert, um dann einige Erweiterungen dieser Grundkonzepte zu erklären, wie das Einbinden von Quelltext in eine andere Datei via der neu hinzugekommenen Präprozessordirektive `#import` sowie die Bedeutung der Dateitypen *.h* und *.m* (siehe Unterkapitel 3.2.1). Es wird die Bedeutung und der Aufbau von Klassendeklarationen und -implementierungen dargelegt (siehe Unterkapitel 3.2.2), bevor Variablen (siehe Unterkapitel 3.2.3) und das aus der objektorientierten Programmiersprache Smalltalk entliehene Konzept der Methoden und Nachrichten betrachtet werden (siehe Unterkapitel 3.2.4)(vgl. [App11b], S.7). Abschließend wird über die erweiterten Sprachkonzepte Kategorien und Klassenerweiterungen (siehe Unterkapitel 3.2.5) sowie Protokolle (siehe Unterkapitel 3.2.6) referiert .

3.1.1. Notationsübersicht

Zur Einführung in *Objective-C* wird der Aufbau einer fiktiven Klasse dargestellt, welche darauf abzielt, möglichst viele Notationselemente zu vereinen. Klassenbeschreibungen setzen sich in *Objective-C* wie in *C* oder *C++* aus zwei Teilen zusammen: einer Klassendeklaration (siehe Codelisting 3.1) und einer Klassendefinition (siehe Codelisting 3.2).

```
1  #import "ProtocolX.h"
2  @class ClassB;
3
4  // Class declaration
5  @interface ClassA : NSObject <ProtocolX>
6  {
7      int anInt;
8    @public
9      id someObject;
10 }
11 @property(nonatomic, copy) NSString *aString;
12 @property(strong, readonly) NSString *anotherString;
13
14 +(void) aClassMethod;
15 -(BOOL) anInstanceMethodWithAParam:(ClassB *)classBInstance;
16 @end
```

Tabelle 3.1.: Deklaration der Klasse `ClassA`.

Den Konventionen entsprechend steht eine Klassendeklaration in einer eigenen Datei, die *Header*-Datei genannt wird und deren Name sich aus dem Namen der Klasse gefolgt von der Dateiendung *.h* ergibt (siehe Unterkapitel 3.2.1.2). Eine Klassendeklaration beschreibt die Schnittstelle einer Klasse, d.h. sie beschreibt die, durch die Klasse nach außen zugänglich gemachte, Datenstruktur und Funktionalität. Weitere Klassen können Klassenschnittstellen einbinden und auf als sichtbar markierte Methoden und Variablen zugreifen (siehe Unterkapitel 3.2.3.4).

Die Klassendeklaration der Klasse `ClassA` bindet die Schnittstellen des Protokolls `ProtocolX` (siehe Unterkapitel 3.2.6) und der Klasse `ClassB` in den ersten zwei Codezeilen ein. Hierzu bietet *Objective-C* die aus *C* bekannten Präprozessordirektiven zum Einbinden von Quellcode, welche für *Objective-C* um weitere Schlüsselwörter ergänzt wurden, an (siehe Unterkapitel 3.2.1.3 und 3.2.2.2). Die in der ersten Zeile benutzte Direktive `#import` garantiert**,** im Gegensatz zu dem aus *C*-Präprozessoren bekannten Schlüsselwort `#include`, dass keine Datei mehr als einmal eingebunden wird. Das Schlüsselwort `@class` (siehe Unterkapitel 3.2.2.2) wird hingegen dann benutzt, wenn nicht auf Implementierungsdetails der hinter dem Schlüsselwort angegebenen Klasse zugegriffen werden muss, zum Beispiel dann, wenn der Klassenbezeichner nur als Datentyp dient. Hiermit wird der Compiler lediglich darüber informiert, dass der hinter `@class` angegebene Klassenbezeichner ein Klassenname ist. Der Quellcode dieser Klasse, genauer genommen das Interface dieser Klasse, wird aber nicht in die verwendete Klasse eingebunden.

Eine Klassendeklaration beginnt, wie in der fünften Zeile des Beispiels ersichtlich, mit dem *Objective-C* Schlüsselwort `@interface`, worauf die Angabe eines Klassennamens, ggf. eine Oberklasse und einer optionalen Liste Protokolle folgen (siehe Unterkapitel 3.2.2.1). Im Gegensatz zu anderen objektorientierten Sprachen wie *C++*, bei denen eine Klasse von mehreren Oberklassen abstammen und erben kann, begrenzt *Objective-C* die Vererbungshierarchie auf einfache Vererbung. Um weitere Funktionalität einzubinden, welche mehrere Klassen in unterschiedlichen Ästen einer Vererbungshierarchie aufweisen können, bietet *Objective-C* die Protokolle an (siehe Unterkapitel 3.2.6). Dieses Konzept erinnert stark an die aus *Java* bekannten Interfaces. Eine Liste der von `ClassA`

unterstützten Protokolle wird in spitzen Klammern angegeben, die Implementierung der darin beschriebenen Funktionalität findet hingegen erst in der Klassendefinition statt.

Auf die Eröffnungsanweisung aus Zeile fünf folgt die Angabe einer **o**ptionalen Liste von Instanzvariablen in einem durch geschweifte Klammern umgebenen Block (siehe Unterkapitel 3.2.2.1 und 3.2.3.1). Instanzvariablen beschreiben den internen Zustand eines Objekts. Als mögliche Datentypen stehen die aus *C* bekannten Typen, sowie Objekttypen in Form der Bezeichner, der im Programm deklarierten *Objective-C* Klassen, zur Verfügung. Neben statischer Typisierung bietet *Objective-C* die aus der Sprache *SmallTalk* bekannte dynamische Typisierung und bietet die generischen Datentypen `Class`, `Protocol` und `id` an (siehe Unterkapitel 3.2.3.3). Neben dem Namen und Typen einer Variable ist in Objective-C auch die Möglichkeit gegeben den Gültigkeitsbereich einer Instanzvariable mit geeigneten Schlüsselwörtern wie u.a. dem in der achten Zeile genutzten `@public` festzulegen (siehe Unterkapitel 3.2.3.4). Der Gültigkeitsbereich einer Variable legt fest an welchen Stellen im Quellcode die Variable zugänglich ist. *Objective-C* bietet nur Instanzvariablen, d.h. Variablen welche für genau ein Instanzobjekt einer Klasse gültig sind. Variablen für Klassenobjekte sind hingegen nicht vorgesehen, lassen sich aber nachahmen (siehe Unterkapitel 3.2.3.2).

Auf die Liste der Instanzvariablen folgt in den Zeilen 11 und 12 des Codelistings die Deklaration von *Properties* (siehe Unterkapitel 3.2.2.1 und 3.2.3.5). *Properties* wurden in *Objective-C 2.0* eingeführt und sollen die Einhaltung des Abkapselungsprinzips, auch Prinzip des Information Hidings, sicherstellen. Dieses Konzept gibt vor, dass nur diejenigen Implementierungsdetails eines Objekts nach außen hin sichtbar sein sollen, die zum korrekten Verhalten **e**ines anderen, externen Objekts dringend benötigt werden. Der direkte Zugriff auf Instanzvariablen von außen, und die damit einhergehende Möglichkeit den Variablenwert extern zu manipulieren, birgt ein hohes Sicherheitsrisiko. Um dieses Problem zu umgehen, wird die Festlegung spezieller Objektfunktionen empfohlen, welche den Zugriff auf Instanzvariablen kontrollieren. Solche Kontrollinstanzen sind als Akzessoren bzw. als *Getter* und *Setter* bekannt (vgl. [Som07], S. 507f). *Objective-C* erleichtert die Erstellung dieser Akzessormethoden mit dem Konzept der Properties. Durch das Schlüsselwort `@property` wird eine *Property*, welche eine Instanzvariable mit dem angegebenem Typ und Namen kapselt, sowie dazugehörige Akzessormethoden deklariert. Anhand optionaler *Property*-Attribute lässt sich das Implementierungsverhalten der Akzessormethoden genauer festlegen. Die indirekte Deklaration von Instanzvariablen anhand von *Properties* bringt den zusätzlichen Vorteil, dass Nutzern beim Lesen der Properties und Property-Attribute vermittelt wird, dass für den Zugriff auf die, durch die *Properties* gekapselten, Instanzvariablen bestimmte Akzessor-Methoden zur Verfügung stehen und wie sich diese in der Implementierung verhalten.

Abschließend werden in den Zeilen 14 und 15 der Klassendeklaration zwei Methoden deklariert (siehe Unterkapitel 3.2.2.1 und 3.2.4). Methoden in *Objective-C* unterscheiden sich von Funktionen aus *C* dahingegen, dass sie an Objekte gebunden sind. Methoden deren Signatur von einem Pluszeichen angeführt werden, deuten auf eine Klassenmethode hin. Solche mit einem Minuszeichen stellen Instanzmethoden dar. Auf die Bindung einer Methode an ein Klassen- oder Instanzobjekt (siehe Unterkapitel 3.2.2.4) folgt der Rückgabetyp. Er wird in runden Klammern angegeben. Darauf folgen der Methodenname und optionale Methodenparameter. Bei der Schreibweise von Methodensignaturen zeigt sich ein gewaltiger Unterschied zwischen *Objective-C und* ***C*** sowie anderen objektorientierten Sprachen wie ***C++*** oder *Java* (siehe Unterkapitel 3.2.4.2). *Objective-C* orientiert sich hierbei an *SmallTalk*. Werden Parameter angegeben, wechselt sich deren Angabe mit Teilen des Methodennamens ab. Der Teil eines Methodennamens vor einem Parameter wird Parame-

terbezeichner genannt und gibt Aufschluss über die Art des Parameters und dessen Rolle in der Methodenimplementierung.

Die Klassendeklaration endet in Zeile 16 mit dem *Objective-C*-Schlüsselwort `@end`.

```
1  #import "ClassA.h"
2  #import "ClassB.h"
3
4  // Class Extension
5  @interface ClassA ()
6  {
7    @private
8      float someFloat;
9  }
10 @property BOOL someBool;
11 @property(strong, readwrite) NSString *anotherString;
12 @end
13
14 // Class definition/ implementation
15 @implementation ClassA
16 {
17   @protected
18     ClassB *aClassBInstance;
19 }
20 @synthesize aString, someBool, anotherString;
21
22 +(void) aClassMethod{
23   // Method implementation
24 }
25
26 -(void) aMethodDeclaredInProtocolXWithOneParam:(id)someObjectParam{
27   // Implementation of Method declared in ProtocolX
28 }
29
30 -(float) anInstanceMethodWithAParam:(ClassB *)classBInstance andAFloatParam:(float)
       someFloatParam{
31   // Method implementation
32   [classBInstance someClassBMethodWithAnIntParam:anInt];
33   self->someFloat = someFloatParam;
34 }
35 @end
```

Tabelle 3.2.: Klassendefinition der Klasse `ClassA`, welche die Funktionalität der Klasse `ClassB` nutzt und die Funktionalität des Protokolls `ProtocolX` implementiert.

In der Implementierungsdatei (siehe Codelisting 3.2) wird das Verhalten der Klasse `ClassA` definiert. In der ersten Zeile wird zuerst der Quellcode der, zur Definition gehörenden, Klassenschnittstelle über die entsprechenden Header-Dateien importiert. Zusätzlich können an dieser Stelle weitere Klassenschnittstellen eingebunden werden – so die der Klasse `ClassB` in der zweiten Zeile. Während in der Klassendeklaration lediglich Referenzen auf diese Klasse benötigt wurden und daher die Angabe von `@class` genügte, wird in der Klassendefinition auf Implementierungsde-

tails von `ClassB`-Objekten zugegriffen weshalb der Quellcode der Klasse `ClassB` eingebunden werden muss (siehe Unterkapitel 3.2.2.2).

Vor dem eigentlichen Anfang der Klassenimplementierung, welche in Zeile 15 beginnt, wird die Funktionalität der Klasse in der fünften Zeile um eine Klassenerweiterung ergänzt (siehe Unterkapitel 3.2.5.2). Eine Klassenerweiterung stellt eine spezielle Unterart einer Kategorie dar (siehe Unterkapitel 3.2.5.1). Kategorien werden dazu verwendet, das Verhalten von Klassen zu erweitern, auch von solchen Klassen, die nur in compilierter Form vorliegen oder deren Quellcode nicht vorliegt. In einer Klassenerweiterung können, nach gleicher Manier wie in Klassendeklarationen, weitere Instanzvariablen, Properties und Methoden deklariert werden. Die Sichtbarkeit dieser ergänzenden Funktionalität ist bis auf einzelne Ausnahmen (wenn das Schlüsselwort `@public` in Kombination mit Instanzvariablen genutzt wird) auf die Klassenimplementierung beschränkt. Dieses Konzept wird häufig verwendet, um *Properties*, auf die extern nur lesend zugegriffen werden darf, neu zu deklarieren, so dass sie innerhalb der Klassenimplementierung auch den schreibenden Zugriffe über geeignete Akzessormethoden ermöglichen. Diese Praxis wird auch in Zeile 11 angewandt, wo der Zugriffsschutz für die *Property* `anotherString` mit dem *Property*-Attribut `readwrite` gelockert wird, im Vergleich zum `readonly`-Attribut in der Klassenschnittstelle (siehe 3.1, Zeile 12).

Zeile 15 markiert mit dem Schlüsselwort `@implementation` den Anfang der Klassenimplementierung (siehe Unterkapitel 3.2.2.1). Im darauf folgenden, optionalen Variablenbereich wird eine neue, nicht-öffentliche Variable vom Typ `ClassB` angelegt (siehe Unterkapitel 3.2.2.1 und 3.2.3.1). Im Anschluss wird in Zeile 20 mit dem Schlüsselwort `@synthesize` dafür gesorgt, dass der Compiler selbsttätig Getter- und Setter-Methoden für die, in dieser Zeile angegeben, *Properties* bei der Kompilierung erzeugt (siehe Unterkapitel 3.2.3.5, Paragraph „Erstellen von Properties").

Abschließend folgt die Implementierung von Methoden, welche das Verhalten der Klasse darstellen (siehe Unterkapitel 3.2.2.1 und 3.2.4). Die Methoden in den Zeilen 23 und 30 stellen Implementierungen der im Klasseninterface von `ClassA` deklarierten Methoden dar. Die Methode in Zeile 26 stellt hingegen die Implementierung einer Methode dar, welche im Protokoll `ProtocolX` deklariert wurde. Durch die Implementierung aller nicht-optionalen Methoden aus einem Protokoll garantiert eine Klasse, dass sie diesem Protokoll genügt, also dessen Funktionalität anbietet (siehe Unterkapitel 3.2.6).

Die Methode aus Zeile 30 erwartet zwei Parameter, ein Instanzobjekt der Klasse `ClassB` und eine Gleitkommazahl `someFloatParam`. In der Implementierung der Methode wird eine Nachricht an das Empfängerobjekt classBInstance geschickt, welche zur Folge hat, dass ein Aufruf der Methode `someClassBMethodWithAnIntParam:` auf dem Objekt `classBInstance` stattfindet (siehe Unterkapitel 3.2.4.1). Auffallend ist die Syntax eines Methodenaufrufs, die sich, wie die der Methodendeklaration, stark von der aus Sprachen der *C*-Familie und anderen objektorientierten Sprachen unterscheidet (siehe Unterkapitel 3.2.4.2). In Zeile 33 findet ein direkter, schreibender Zugriff auf die Instanzvariable `someFloat` anhand des, aus der Sprache *C* bekannten, Pfeiloperators statt (siehe Unterkapitel 3.2.3.1).

3.2. Objective-C Sprachspezifikation

In den folgenden Unterkapiteln werden die Sprachkonzepte und Notationselemente von *Objective-C* beschrieben. Die Kapitelstruktur sieht keine explizite Trennung zwischen Sprachmitteln von *Objective-C 1.0* und *2.0* vor. Stattdessen wird lediglich im Fall der Funktionalität, die im Rahmen der Sprachrevision eingeführt wurde, im Fließtext darauf hingewiesen.

Zuerst werden in Unterkapitel 3.2.1 die Grundzüge von *Objective-C* als Erweiterung zur prozeduralen Sprache *C* eingeführt. Dann wird in Unterkapitel 3.2.2die Umsetzung des Klassen-Konzepts aus dem objektorientierten Paradigma in *Objective-C* vorgestellt. Anschließend werden Instanzvariablen und *Properties* sowie Methoden und Nachrichten behandelt (siehe Unterkapitel 3.2.3 und 3.2.4). Zuletzt werden die erweiterten Sprachkonzepte der Kategorien, Klassenerweiterungen und Protokolle beschrieben (siehe Unterkapitel in 3.2.5.1, 3.2.5.2 und 3.2.6).

Viele Konzepte sind nicht in der Sprachdefinition selbst festgelegt, sondern werden erst durch den verwendeten Compiler, die benutzte Laufzeitumgebung und das eingesetzte Framework ermöglicht. Wo es sich in diesem Kapitel anbietet, wird auf die Rolle des Compilers und der Laufzeitumgebung hingewiesen. Die Rolle des Frameworks wird hingegen erst in Kapitel 4 näher betrachtet.

3.2.1. Der prozedurale Unterbau von Objective-C

3.2.1.1. Objective-C als Obermenge von C

Als Fundament von *Objective-C* dient die Hardware-nahe Sprache *ANSI-C*. Die prozedurale Syntax von *Objective-C* ist daher identisch zu *C*. Genauer gesagt ist *Objective-C* eine strikte Obermenge von *C*, wodurch beliebiger *ANSI-C*-Code an jeder gültigen Stelle in den *Objective-C*-Code eingebettet werden kann und das Ergebnis von jedem *Objective-C*-Compiler kompilierbar ist.

Objective-C pflegt viele objektorientierte Konzepte in *C* nach und lehnt sich dabei stark an der Syntax und Konzeption von *SmallTalk* an. Die neuen, objektorientierten Sprachmittel und Konzepte sind strikt von der prozeduralen *C*-Syntax getrennt. Diese Trennung ermöglicht es, dasselbe Erweiterungskonzept auch auf andere imperative Sprachen anzuwenden. So existiert u.a. auch eine objektorientierte Version von *Pascal*, genannt *Objective-Pascal*. Reine *Objective-C*-Schlüsselwörter lassen sich am vorangestellten `@` leicht erkennen.

Als Laufzeitumgebung wird bei *Objective-C* auf eine leichtgewichtige, effiziente und in *C* geschriebene Laufzeitumgebung zurückgegriffen. Damit wird ein *Objective-C*-Programm über die Summe seines Codes und seiner eingebundenen Bibliotheken nicht unnötig vergrößert. Somit braucht es wesentlich weniger Speicher sowie Ausführungszeit als vergleichbare *SmallTalk*-Programme, welche in einer virtuellen Maschine ausgeführten werden (vgl. Kapitel 2). Anzumerken ist aber, dass compilierte *Objective-C*-Programme äquivalente *C*- und *C++*-Programme bezüglich ihrer Größe übersteigen. Ein Umstand der, dem Konzept der dynamischen Typisierung, wie sie in Objective-C implementiert ist, geschuldet ist[1]. (vgl. [Seb11], S.27f)

[1] Durch die dynamische Typisierung kann zur Compilezeit keine sichere Aussage darüber getroffen werden ob eine Methode zur Laufzeit benutzt wird. Hierdurch können qualifizierte Methoden beim kompilieren nicht gänzlich entfernt oder ihr Code durch *Inline-Expansion* an geeigneten Stellen eingefügt und die entsprechenden Methoden dann weggelassen werden.

3.2.1.2. Die Dateitypen .h und .m

Wie in der Einleitung zu diesem Kapitel bereits erwähnt, orientiert sich die Kapitelstruktur grob am Programmverlauf. Aus diesem Grund soll ein erster Unterschied zwischen *C*, *C++* und *Objective-C* anhand der Dateien erläutert werden, in denen sich der Quellcode befindet und deren Erstellung den Anfang jeder Programmierarbeit ausmacht.

Eine Quellcodedatei mit Dateiendung *.h* steht im Kontext der Sprachen *C*, *C++* und *Objective-C* gleichermaßen für eine *Header*-Datei[2]. Diese enthält viele, vielleicht sogar alle, Variablen und Methodendeklarationen, die die repräsentierte Klasse ausmachent. Eine *.h*-Datei kann von beliebig vielen Klassendeklarationen und -definitionen importiert werden (siehe Unterkapitel 3.2.2.2). So kann u.a. aus einer Instanz einer Klasse auf entsprechend deklarierte Inhalte einer Instanz einer importierten Klasse zugegriffen werden. Zudem können mehrere Klassenimplementierungen eine gemeinsame *Header*-Datei als Klassendefinition nutzen. Das importieren einer Klassendeklaration oder -definition kann in *Objective-C* mit dem, aus *C* bekannten, Schlüsselwort `#include` oder mit dem, in *Objective-C* hinzugefügten, Schlüsselwort `#import` stattfinden (siehe Unterkapitel 3.2.1.3).

Die Implementierung einer Klasse steht üblicherweise in einer weiteren Datei, die in *Objective-C* den Konventionen entsprechend auf *.m* endet. Die entsprechenden Repräsentanten in *C* und *C++* enden hingegen auf *.c* respektive *.cpp*. In eine *.m*-Datei können Klassendeklarationen und Implementierungen anderer Klassen mit `#include` oder in *Objective-C* `#import` eingebunden werden. Sie kann zudem weitere Attribute enthalten und implementiert zumeist die Methoden, die in der, zur Klassenimplementierung passenden, *Header*-Datei deklariert wurden.

Der weiteren Bedeutung und der genaue Kennzeichnung von *.h* und *.m* Dateien in *Objective-C* widmet sich das Unterkapitel 3.2.2.

3.2.1.3. Präprozessordirektiven

Da *Objective-C* auf *C* basiert und der Code durch einen von *NeXT* und später von *Apple* erweiterten *C*-Compiler behandelt wird, steht auch eine erweiterte Version des *C*-Präprozessors und damit alle aus *C* bekannten Präprozessordirektiven zur Verfügung. Der Präprozessor kümmert sich um das bedingungslose und an Bedingungen gebundene Einbinden von Quellcode sowie die Definition, oder genauer die Expansion, von Konstanten und Makros. Auch in diesem Bereich bietet *Objective-C* bzw. der erweiterte Präprozessor weitere Schlüsselwörter. In *Objective-C* stehen u.a. folgende Präprozessordirektiven zur Verfügung:

- **`#include`** und **`#import`**

  Mit beiden Schlüsselwörtern kann der Quelltext einer Datei in eine andere Datei eingebunden werden (siehe Beispiel 3.3). Während der Befehl `#include` auch unter *C* und *C++* zur Verfügung steht, wird `#import` nur von Präprozessoren für *Objective-C* unterstützt. Beide Direktiven verhalten sich sehr ähnlich, jedoch unterscheidet sich `#import` dahingehend, dass der Quellcode einer mehrfach eingebundene Datei nicht multiple Male in die Klasse

[2] Der *Objective-C*-Compiler erlaubt, dass eine Klasse in einer Datei mit beliebiger Endung deklariert wird. In Anlehnung an die Bedeutung der Klassendeklaration aus *C*, wird aber üblicherweise die vom Namen *Header* abgeleitete Dateiendung *.h* bevorzugt. (vgl. [App11b], S. 33)

eingefügt wird, Fehlern bei der Kompilierung vorgebeugt wird. In *C/C++* muss ein ähnliches Konstrukt manuell nachgebildet werden[3] was eine häufige Fehlerquelle darstellt und die Lesbarkeit verschlechtert (siehe 3.4). (vgl. [Seb11], S.28f)

```
1 /* In C/C++ */
2 #include "someClassDeclaration.h"
3 #include "someOtherClassImplementation.m"
4
5 /* In Objective-C */
6 #import "anotherClassDeclaration.h"
7 #import "yetAnotherClassDefinition.m"
```

Tabelle 3.3.: Bedingungslose Einbindung von zwei Dateien mittels `#include` und `#import`.

```
1 /* In Objective-C */
2 #import _SOMEFILE_H
3
4 /* Equivalent behavior in C/C++ */
5 #ifndef _SOMEFILE_H_
6 #define _SOMEFILE_H_
7   // Actual Code
8 #endif /*_SOMEFILE_H_*/
```

Tabelle 3.4.: Nachbildung der Verhaltensweise von `#import` in *C/C++*.

- **`#define`** und **`#undef`**

  Mit den Präprozessordirektiven `#define` und `#undef` lassen sich unter *C*, *C++* sowie *Objective-C* Symbole und Makros definieren und wieder löschen (siehe Beispiel 3.5). Der Präprozessor erweitert diese zur Compilezeit. An Stellen im Code wo sie verwendet werden, wird die entsprechende Quellsprachanweisung eingefügt. Makros verhalten sich dabei ähnlich wie *C*-Methoden und können Argumente annehmen. Der Präprozessor evaluiert den Ausdruck und liefert das Ergebnis als Quellsprachanweisung zurück, das dann an den entsprechenden Stellen im Code eingefügt wird.

```
1 #define FOO 1
2   // do sth. with FOO
3 #undef FOO
4 #define FOO @"Bar"
5   // do sth. else with a new FOO
```

Tabelle 3.5.: Definition und Neudefinition eines Symbols.

[3] Alternativ kann unter *C* oder *C++* die nicht standardisierte, aber von vielen Compilern unterstützte, Anweisung `#pragma once` genutzt werden.

- **`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, und `#endif`**

  Ebenfalls aus der Syntax des *C*-Präprozessors stammen die Schlüsselwörter `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` und `#endif`, welche sich in *Objective-C* exakt wie unter *C* und *C++* verhalten (siehe Beispiel 3.6). (Details unter [Ale])

```
1  #ifdef DEBUG
2   #undef DEBUG
3   #define DEBUG 2
4  #else
5   #define DEBUG 1
6  #endif
7
8  #ifndef _SOME_FILE_H_
9   #define _SOME_FILE_H_
10   // Put code here
11 #endif
12
13 #if DEBUG == 1
14  #import "someDebugAssistanceFile.h"
15 #elif DEBUG == 2
16  #import "someOtherDebugAssistanceFile.h"
17 #else
18  #import "someNormalFile.h"
19 #endif
```

Tabelle 3.6.: Dieser Quellcode-Auszug zeigt die Nutzung diverser Präprozessordirektiven für an Bedingungen gebundene Codeeinbindungen. Im ersten Block wird zur Compilezeit geprüft, ob die Zeichenfolge `DEBUG` bereits an vorausgegangener Stelle mit der entsprechenden Präprozessordirektive `#define` definiert wurde. Wenn dies der Fall ist, wird die Definition aufgehoben und anschließend mit dem Wert 2 neu angelegt. War das Symbol `DEBUG` hingegen noch nicht definiert, wird es in der fünften Zeile mit dem Wert 1 angelegt. Im zweiten Block, ab der achten Zeile, wird geprüft ob die Zeichenfolge, welche in diesem Fall dem Namen einer Quellcodedatei entsprechen soll, noch nicht definiert wurde. Ist dies nicht der Fall, hierbei wird davon ausgegangen dass diese Datei und damit ihr Code noch nicht in die aktuelle Datei eingebunden wurde, wird das Symbol angelegt und der Code für diese Datei festgelegt. Im dritten Block, ab Zeile 13, wird eine bestimmte Datei importiert, je nachdem ob der dem Symbol `DEBUG` zugewiesene Wert dem Zeichen `1`, `2` oder einem anderen Wert entspricht.

- Kommentare mit **`//`**, **`/*`** und **`*/`**

  Wie in *C* oder *C++* kann der Code in *Objective-C* kommentiert werden. Kommentare werden zur Compilezeit vom Präprozessor entfernt. Sie tauchen somit nicht im kompilierten Code auf. Bei der Kommentierung wird zwischen Zeilen- und Blockkommentaren unterschieden (siehe Beispiel 3.7). Zeilenkommentare erstrecken sich ab der Zeichenfolge `//` über eine einzige Zeile. Ein Blockkommentar grenzt hingegen einen Teil des Quellcodes mit den Zeichenfolgen `/*` als Anfang und `*/` als Ende ein und deklariert die Zeichen in der Region dazwi-

schen als Kommentar. Blockkommentare können mehrzeilig sein und dürfen in Objective-C im Vergleich zu anderen Sprachen nicht rekursiv verschachtelt werden.

```
int i = 0; // a variable definition
while(/*true*/i++ <= 10){
  /* performant sth. important
       in this important loop*/
}
```

Tabelle 3.7.: Unterschiedliche Einsatzszenarien für Zeilen- und Blockkommentare.

- **`#pragma`**

  Das Schlüsselwort Pragma, vom griechischen „Handlung", erlaubt in *C*, *C++* und *Objective-C* die Überlieferung von über den Code hinausgehenden Anweisungen und Informationen an den Compiler oder das Entwicklungswerkzeug. Bis auf wenige Ausnahmen unterstützten verschiedene Compiler- und *IDE*-Modelle unterschiedliche Pragma-Anweisungen. Wird ein Pragma nicht unterstützt wird kein Fehler generiert sondern das Pragma ignoriert (vgl. [Mar09], S.119).

  Erwähnenswert ist bei Nutzung der Entwicklungsumgebung *Xcode* das Schlüsselwort `#pragma` mit dem Zusatz `mark`. Hierdurch wird der IDE signalisiert, die darauffolgenden Methoden in der *Xcode Jump Bar* unter einer, hinter `mark` angegebenen, Überschrift zu gruppieren (siehe Codebeispiel 3.8 und Illustration 3.1) (weitere Details zu `#pragma mark` und der *Jump Bar* im Unterkapitel 4.3.4).

```
#pragma mark - Initialiazion methods

- (id) init{
  if(self = [super init]){
    // Custom initialization
  }
  return self;
}

- (id) initWithParam:(id)aParam{
  self = [self init];
  // Do sth. with aParam
  return self;
}
```

Tabelle 3.8.: Mit dem Schlüsselwort `#pragma` und dem Zusatz `mark` werden alle auf diese Anweisung folgenden Methoden bis zum Dateiende oder einer weiteren „`#pragma mark`"-Anweisung in der *Xcode Jump Bar* gruppiert. Die *Jump Bar* dient der bequemen Navigation in Quellcodedateien. Wird darin eine Methode angeklickt springt der Editor zu der entsprechenden Stelle im Code.

Abbildung 3.1.: Methodengruppierung mittels #pragma mark in der *Xcode Jump Bar*. Bei Angabe eines Bindestrichs in der Überschrift, kann eine horizontale Trennlinie in die *Jump Bar* eingefügt werden.

- **`#warning und #error`**

  Anhand der Schlüsselwörter `#warning` und `#error` lassen sich explizit Fehler- bzw. Warnmeldungen im Code platzieren, auf die beim Compilieren hingewiesen wird. Diese Erweiterungen, die sich nicht in jedem Compiler wiederfinden, so aber z.Bsp. in solchen für *Objective-C*, haben entsprechend eingesetzt durchaus ihre Daseinsberechtigung. So können sie in Kombination mit *Xcode* u.a. eingesetzt werden um Xcode Schlüsselwörtern wie `TODO:` und `FIXME:` durch zusätzliche Hervorhebung beim Compiliervorgang mehr Nachdruck zu geben.

3.2.2. Der Objektorientierte Ansatz von Objective-C: Klassen und Objekte

3.2.2.1. Rolle und Aufbau von Objekten und Klassen

Dem Paradigma der objektorientierten Programmierung folgend sind *Objective-C*-Programme um Objekte herum aufgebaut. Ein Objekt wird als abgeschlossene Programmeinheit definiert, welche eine Datenstruktur (Instanzvariablen, auch *Member*-Variablen genannt) mit einer Gruppe Prozeduren (Methoden) bündelt, die sich auf diese Datenstruktur berufen können. Wie in anderen objektorientierten Sprachen, deklariert man in *Objective-C*-Objekte durch die Definition ihrer Klasse. Dabei stellt die Klassendefinition einen Prototypen für eine Art von Objekt dar. Für jede Klasse erzeugt der Compiler genau ein zugängliches Objekt, welches eine kompilierte Repräsentation dieser Klasse darstellt. Anhand dieses Objekts, auch Klassenobjekt genannt, lassen sich weitere Objekte dieser Klasse erzeugen. Diese besitzen unabhängige Instanzen ihrer Variablendefinitionen, teilen sich aber die Implementierung der Methoden ihrer Klasse (siehe Unterkapitel 3.2.2.4). In diesem Zusammenhang wird das Klassenobjekt auch als *Factory-object* und die hierdurch erzeugten Objekte als Klasseninstanzen oder Instanzobjekte bezeichnet (vgl. [App11b], S.10 und S.19).

Der *Objective-C*-Compiler erfordert für die Erstellung eigener Klassen, die Deklaration eines Klasseninterfaces und die Definition einer zugehörigen Klassenimplementierung. Üblicherweise werden Deklaration und Definition in zwei separat deklarierten Dateien vorgenommen, in einer *Header*-Datei für die Deklaration und einer Implementierungsdatei für die Definition. Vom Compiler her ist diese Trennung allerdings nicht zwingend erforderlich (vgl. [App11b], S.33). Per Konvention wird in *Header-*/ Implementierungsdateien genau eine Klasse beschrieben Auch dies ist nicht ausdrücklich vom *Objective-C*-Compiler gefordert (siehe Unterkapitel 3.2.1.2).

Im folgenden werden die Strukturen welche Klassendeklaration und -implementierung ausmachen beschrieben. Hierbei werden einige Sprachmittel kurz eingeführt welche dem Verständnis

und der Vollständigkeit der Beschreibungen dienen. Diese Sprachmittel werden in weiteren Kapiteln genauer beleuchtet, die entsprechenden Kapitel sind angegeben. Anzumerken bleibt, dass sich der Aufbau von Klassendeklaration und -definition in *Objective-C* bis auf kleine Abweichungen sehr ähnlich sind. Aus diesem Grund werden manche Sprachelemente in Bezug zu Deklaration und Definition zusammengefasst behandelt. Hierauf wird im Fließtext hingewiesen.

Klassendeklaration:

Die Klassendeklaration findet üblicherweise in einer *Header*-Datei statt, welche im Regelfall auf *.h* endet (siehe Unterkapitel 3.2.1.2). Der Name der Datei ist dabei eng an den Namen der Klasse angelehnt, welche in der Datei deklariert wird. Die Klassendeklaration beginnt mit dem *Objective-C*-Schlüsselwort **`@interface`** und endet mit **`@end`**. Zwischen diesen Schlüsselwörtern wird der Funktionsumfang einer Klasse festgelegt. Implementierende Klassendefinitionen erben diese Funktionalität und müssen sie ggf. implementieren.

Die Einleitungszeile eines Klasseninterfaces setzt sich aus nachfolgenden Teilen zusammen (siehe Beispiel 3.9):

1. Das Schlüsselwort `@interface` stellt den Beginn der Klassendeklaration dar.
2. Darauf folgt die Klassenbezeichnung, also der Name der Klasse deren Interface deklariert wird.
3. Anschließend folgt die Angabe der Oberklasse hinter einem Doppelpunkt.

   Die Angabe der Superklasse legt die Position der neuen Klasse in der Ableitungshierarchie fest. Eine Klasse erbt entsprechend deklarierte Variablen und Methoden von ihren direkten und indirekten Oberklassen. Im Vergleich zu *C++* erlaubt *Objective-C* keine multiple Vererbung. Jede Klasse hat also maximal eine Oberklasse. Um dieses Defizit auszugleichen, können in *Objective-C* Kategorien und Protokolle eingesetzt werden, welche funktionell dem aus *Java* bekannten Konzept der Interfaces ähneln (siehe Unterkapitel 3.2.5 und 3.2.6).
4. Abschließen kann eine Liste von Protokollen angegeben werden, deren Funktionalität von der Klasse oder von Unterklassen dieser Klasse implementiert wird (siehe Unterkapitel 3.2.6). Die Protokolle werden in spitzen Klammern angegeben und durch Kommata getrennt.

```
1 @interface ClassA : NSObject <ProtocolX>
2   // Interface declaration
3 @end
```

Tabelle 3.9.: Beispiel der Deklaration einer Klasse mit Superklasse – Auszug aus dem Einführungsbeispiel 3.1.

Alternativ zur Angabe einer Superklasse kann auch eine Kategorie in Klammern hinter der Klassenbezeichnung angegeben werden (siehe Beispiel 3.10). Diese Angabe sagt aus, dass eine bereits existierende Klasse mit gleichem Klassenbezeichner um weitere, in dieser Datei beschriebene,

Funktionalität erweitert wird (siehe Unterkapitel 3.2.5). Die zu erweiternde Klasse kann dabei in anderen Dateien deklariert und definiert und sogar bereits erweitert worden sein. Zudem ist es nicht nötig den Inhalt der zu erweiternden Klasse als Quellcode vorliegen zu haben. So können auch bereits compilierte Klassen nachträglich beliebig erweitern werden.

```
1 @interface SomeExistingClass (SomeCategoryAsExtensionOfSomeExistingClass)
2   // Extension of SomeExistingClass' Interface declaration
3 @end
```

Tabelle 3.10.: Beispiel einer Klassendeklaration, welche eine bereits existierende Klasse um neue Funktionalität erweitert.

Klassendefinition:

Die Definition einer Klasse findet in einer eigenen Implementierungsdatei statt, deren Namen sich am Klassennamen orientiert (siehe Unterkapitel 3.2.1.2). Die Implementierung beginnt mit dem *Objective-C* Schlüsselwort **`@implementation`** und endet mit **`@end`**. Zwischen diesen Anweisungen steht die Ausformulierung der in dem oder den dazugehörigen Interface(s) und optionalen Kategorien/ Protokollen (siehe Unterkapitel 3.2.5 und 3.2.6) deklarierten Prozeduren. Dazu ist die Einbindung der zur Klassendefinition gehörenden *Interface*-Datei sowie der *Header*-Dateien weiterer benutzten Klassen und Interfaces zwingend erforderlich (siehe Unterkapitel 3.2.2.2).

Die Einleitungszeile einer Klassenimplementierung setzt sich aus nachfolgenden Teilen zusammen (siehe Beispiel 3.11):

1. Das Schlüsselwort `@implementation` stellt den Beginn der Klassendefinition dar.
2. Es folgt die Bezeichnung der Klasse, also der Name der Klasse, deren Verhalten implementiert wird.

Weitere Elemente sind nicht nötig. Die Implementierung muss importierte Deklarationen nicht wiederholen, wodurch die Angabe der Superklasse entfallen kann.

```
1 @implementation ClassA
2   // Implementation of ClassA
3 @end
```

Tabelle 3.11.: Anfangs- und Endzeile einer Klassenimplementierung.

Instanzvariablen und Properties:

Nach Einleitung der Deklaration oder Definition kann eine Liste von Instanzvariablen in einem Block mit geschweiften Klammern folgen (siehe Beispiel 3.12). Der Zugriff auf Instanzvariablen ist

für jedes Instanzobjekt (siehe Unterkapitel 3.2.2.4) dieser Klasse zulässig, jedoch besitzt jedes Instanzobjekt eine eigenständige Instanz jeder in der Klasse enthaltenen *Member*-Variable. Beim Typ einer Variable stehen alle Typen aus *ANSI-C* zur Verfügung und zusätzlich alle deklarierten und importierten Klassenbezeichner sowie die dynamischen Datentypen `id`, `Class` und `Protocol` (siehe Unterkapitel 3.2.3.3). Die Sichtbarkeit jeder Instanzvariable kann individuell mit entsprechenden Schlüsselwörtern festgelegt werden (siehe Kapitel 3.2.3.4).

```
1 {
2     int anInt;
3   @public
4     id someObject;
5 }
```

Tabelle 3.12.: Deklaration von zwei Instanzvariablen.

Weitere Instanzvariablen lassen sich mit dem in *Objective-C 2.0* eingeführten Sprachkonzept der *Properties* und dem entsprechenden Schlüsselwort `@property` deklarieren (siehe Beispiel 3.13). Obwohl Instanzvariablen sowohl in Klasseninterface wie auch in der Klassenimplementierung zulässig sind, stellen sie ein Implementierungsdetail dar und sollten daher typischerweise nicht direkt von außen zugänglich sein. Es wird daher die exklusive Benutzung von *Properties* im Klasseninterface empfohlen, welche dafür Sorge tragen, dass entsprechende *Getter*- und *Setter*-Methoden für den Zugriff auf diese Variablen existieren. So ist kein direkter Zugriff auf die Variable von außen möglich, wodurch die Fehleranfälligkeit des Programms reduziert wird (siehe Unterkapitel 3.2.3.5, Paragraph „Die Rolle von *Properties*").

Während *Properties* ausschließlich in Klassendeklarationen deklariert werden, ist es Aufgabe der Klassenimplementierung die durch eine *Property* gekapselte Instanzvariable und die dazugehörigen Akzessormethoden mit dem Schlüsselwort `@synthesize` zu generieren (siehe Beispiel 3.14). Optional können *Getter*- und *Setter*-Methoden auch per Hand angelegt oder zur Laufzeit eingebunden werden (siehe Unterkapitel 3.2.3.5, Paragraph „Erstellen von *Properties*").

Die *Objective-C*-Syntax unterstützt optionale Attribute, welche die Zugriffsmethoden für *Properties* näher definieren. Die Attribute lassen sich hinsichtlich des Schreibschutzes, des Referenzmodells und der Atomarität unterscheiden (siehe Unterkapitel 3.2.3.5, Paragraph „*Property*-Attribute").

```
1 @property(nonatomic, copy) NSString *aString;
```

Tabelle 3.13.: Deklaration einer Property für eine Zeichenkette in einer Klassendeklaration. Das Attribut `nonatomic` gibt vor, dass der Zugriff auf die Variable durch die Akzessormethoden nicht thread-sicher implementiert ist. Das Attribut `copy` hingegen besagt, dass eine Kopie der, an die Instanzvariable übertragene, Zeichenkette gespeichert werden soll. Es wird eine neue Instanz der Zeichenkette im Speicher abgelegt und unter diesem Variablenbezeichner referenziert, statt auf die Speicherstelle der ursprünglich zugewiesenen Zeichenkette zu verweisen.

```
1 @synthesize aString, someBool, anotherString;
```

Tabelle 3.14.: Synthese von drei *Properties* in einer Klassenimplementierung. Hierbei werden zur Compilezeit vom Compiler selbsttätig den *Property*-Attributen genügende *Getter*- und ggf. *Setter-Methoden* für den Zugriff auf die, durch die *Property* gekapselten, Variablen generiert.

Methoden:

Auf die Deklaration der Variablen folgt die Deklaration bzw. Implementierung von Klassen- und Instanzmethoden (siehe Beispiele 3.15 und 3.16). Dabei müssen in der Klassendefinition alle im Interface deklarierten sowie in allen eingebundenen Protokollen (siehe Unterkapitel 3.2.6) als nicht-optional markierten, deklarierten Methoden, implementiert werden.

Ob es sich um eine Klassen- oder eine Instanzmethode handelt, verrät das vorangestellte Zeichen '+' bzw. '−'. In Klammern folgt der optionale Rückgabetyp, bei dem die gleichen Datentypen angegeben werden können wie bei Variablen, zuzüglich des Typs `void` (siehe Unterkapitel 3.2.3.3 und 3.2.4). Wird der Rückgabetyp weggelassen, dann entfallen auch die Klammern und es wird als Rückgabe nicht wie in *C* von einem *Integer*-Wert sondern von einem Wert vom Typ `id` ausgegangen[4] (vgl. [App11b], S. 11). Ein eklatanter Unterschied zwischen *C/C++* und *Objective-C* stellt die Syntax der Methodensignatur dar (siehe Beispiel 3.15), worauf in Unterkapitel 3.2.4.2 genauer eingegangen wird.

Objective-C unterscheidet zwischen **Nachrichten und Methoden**. Bei der Implementierung von Prozeduren spricht man von Methoden, sie führen wie in *C* oder *C++* Anweisungen nach und nach aus, können Übergabeparameter erhalten, greifen auf Klassenstrukturen zurück und liefern optional ein Ergebnis zurück. Beim Aufruf von Prozeduren orientiert sich *Objective-C* hingegen am Nachrichtenkonzept von *SmallTalk*. Ein **Sender**, in Form eines Klassen- oder Instanzobjekts (siehe Unterkapitel 3.2.2.4), sendet eine **Nachricht** an einen **Empfänger**, welcher ein beliebiges weiteres Klassen- oder Instanzobjekt sein kann (siehe Beispiel 3.16, siebte Zeile). Spezielle durch den Compiler generierte *C*-Methoden in den Klassen greifen die Nachricht ab und versuchen die dem Nachrichtentyp entsprechenden Implementierung einer gleichnamigen Methode des Objekts aufzurufen. Dieses Nachrichtenkonzept spielt eine entscheidende Rolle bei der dynamischen Bindung (siehe Unterkapitel 3.2.4.1).

```
1 +(void) aClassMethod;
2 -(float) anInstanceMethodWithAParam:(ClassB *)classBInstance andAFloatParam:(float)
      someFloatParam;
```

Tabelle 3.15.: Deklaration einer Klassen- und einer Instanzmethode.

[4] Für strikte *C*-Konstrukte bleibt hingegen `int` der Standardrückgabetyp (vgl. [App11b], S. 11).

```
+(void) aClassMethod{
  // Method implementation
}

-(float) anInstanceMethodWithAParam:(ClassB *)classBInstance andAFloatParam:(float)
    someFloatParam{
  // Method implementation
  [classBInstance someClassBMethodWithAnIntParam:anInt];
  self->someFloat = someFloatParam;
}

-(void) aMethodDeclaredInProtocolX{
  // Implementation of Method declared in ProtocolX
}
```

Tabelle 3.16.: Implementierung einer Klassenmethode, einer im Klasseninterface deklarierten und einer in einem Protokoll deklarierten Instanzmethode. In der zweiten Methode wird einem `classBInstance`-Instanzobjekt eine Nachricht vom Typ `someClassBMethodWithAnIntParam:` mit dem Parameter `anInt` zugesendet. Zur Laufzeit wird dann versucht eine zum Nachrichtentyp passende Methode im Empfänger-Objekt `classBInstance` auszuführen.

3.2.2.2. Verwendung anderer Klassen

Um in einer Klasse andere Klassen/ Protokolle als Typ nutzen oder gar auf Implementierungsdetails anderer Klassen/ implementierter Protokolle zurückgreifen zu können, müssen vor dem Kompiliervorgang die entsprechenden Klassen-/ Protokollinformationen eingebunden sein. Hierfür stehen je nach Fall drei leicht unterschiedliche Sprachelemente zur Verfügung.

`#import` und `#include`:

Bei den Schlüsselwörtern `#import` und `#include` handelt es sich um Präprozessordirektiven, welche bereits in Unterkapitel 3.2.1.3 behandelt wurden. Die Schlüsselwörter kommen immer dann zur Anwendung, wenn Implementierungsdetails externer Klassen benötigt werden. Sie werden z.Bsp. dann eingesetzt, wenn Instanzen einer externen Klasse erstellt, auf Instanzvariablen einer externen Klasse zugegriffen oder einem Klassen- oder Instanzobjekt einer anderen Klasse eine Nachricht zugesendet werden soll (siehe Beispiel 3.17). Auch wenn eine Klasse ein Protokoll implementiert, muss die entsprechende Deklaration dieses Protokolls importiert werden.

Der Quellcode aller in einer Klasse benötigten externen Klassen und implementierten Protokolle werden typischerweise am Anfang einer Klassendeklaration oder -definition eingebunden. Um widerzuspiegeln, dass die Definition einer Klasse auf der Definition geerbter Klassen beruht, beginnt die Klassenimplementierung üblicherweise mit dem Einbinden des zur Implementierung passenden Interfaces. Hierdurch werden auch die Klasseninterfaces aller, der aktuell definierten

Klasse, übergeordneten Klassen in der Vererbungshierarchie mit eingebunden – d.h. jede Klassendefinition enthält indirekt alle Interfaces aller geerbten Klassen.

Das Schlüsselwort `#import` verhält sich identisch zu `#include`, bis auf den Unterschied, dass es garantiert, dass keine Datei mehr als einmal eingebunden wird (siehe Unterkapitel 3.2.1.3).

```
1 #import "ClassA.h"
2 #import "ClassB.h"
```

Tabelle 3.17.: Einbindung der Quelltexte zweier Klassendeklarationen. In Klassendefinitionen wird den Konventionen entsprechend zuerst das zur Implementierung dazugehörige Interface importiert. Dann folgt der Import weiterer Klassen und Protokolle.

`@class`:

Greift eine Klassendeklaration oder -definition nicht auf die Implementierungsdetails einer anderen Klasse/ eines anderen Protokolls zurück, sondern benutzt dessen Bezeichner lediglich als Typ, kann auf das Schlüsselwort `@class` zurückgegriffen werden (siehe Codeauszug 3.18). Hiermit wird der Compiler lediglich darüber informiert, dass der hinter `@class` angegebene Bezeichner ein Klassen-/ Protokollname ist. Der Quellcode dieser Klasse, genauer genommen das Interface dieser Klasse, wird aber nicht in die verwendende Klasse eingebunden. Eine Verwendung von `@class` reduziert demnach die Menge Code, die Compiler und Linker bearbeiten, und stellt die einfachste Methode dar um Vorwärtsdeklarationen eines Klassennamens anzugeben (vgl. [App11b], S.36f).

```
1 @class ClassB;
```

Tabelle 3.18.: Auszug aus dem Einführungsbeispiel 3.2, bei dem im Interface der Klasse `ClassA` dem Compiler die Klasse `ClassB` lediglich als Klassenbezeichner bekannt gemacht wird, da das Interface selbst nicht auf Implementierungsdetails der Klasse `ClassB` zugreift. In der Implementierung der Klasse `ClassA` wird dann `ClassB` explizit eingebunden, weil hier auf Implementierungsdetails von `ClassB` zugegriffen wird.

Anzumerken ist, dass wenn das Interface einer Klasse/ eines Protokolls jedoch tatsächlich genutzt wird, dieses explizit mit `#import` oder `#include` eingebunden werden muss. Das wäre zum Beispiel dann nötig, wenn Instanzen dieser Klasse erstellt werden sollen, auf Instanzvariablen zugegriffen werden soll oder Nachrichten an Objekte dieser Klasse gesendet werden sollen.

Typischerweise werden einer Klassendeklaration die Bezeichner von Klassen/ Protokollen mit `@class`-Anweisungen bekannt gemacht und für den Zugriff auf Implementierungsdetails in der dazugehörigen Klassenimplementierungen explizit mit `#import` der Quellcode der Klassen/ Protokolle eingebunden. Durch diese Vorgehensweise lassen sich mögliche Probleme umgehen, wel-

che auftreten, wenn Klassen weitere Klassen importieren und hierdurch unter Umständen zirkuläre Abhängigkeiten entstehen. Beispielsweise kann es vorkommen, dass der Compiler mit einem Fehler abbricht, wenn eine Klasse die Klassenbezeichner einer anderen Klasse als Typ einer Instanzvariable nutzt und sich beide implizierten Klassen gegenseitig importieren (siehe Beispiel 3.19 und Codelisting 3.20 für eine mögliche Lösung).

```
1  /* Header-file Foo.h */
2  #import "Bar.h"
3
4  @interface Foo : NSObject
5  @property Bar *bar;
6  @end
7
8
9  -----
10
11
12 /* Implementation-file Foo.m */
13 #import "Foo.h"
14
15 @implementation Foo
16 @synthesize bar;
17 @end
18
19
20 -----
21
22
23 /* Header-file Bar.h */
24 #import "Foo.h"
25
26 @interface Bar: NSObject
27 @property Foo *foo;
28 @end
29
30
31 -----
32
33
34 /* Implementation-file Bar.m */
35 #import "Bar.h"
36
37 @implementation Bar
38 @synthesize foo;
39 @end
```

Tabelle 3.19.: Problembehaftete Deklaration zweier zirkulär abhängiger Interfaces. Führt beim Kompilieren zu Fehlern.

```objc
/* Header-file Foo.h */
@class Bar;

@interface Foo: NSObject
@property Bar *bar;
-(void)fooMethod;
@end

-----


/* Implementation-file Foo.m */
#import "Foo.h"
#import "Bar.h"

@implementation Foo
@synthesize bar;
@end

-----


/* Header-file Bar.h */
@class Foo;

@interface Bar: NSObject
@property Foo *foo;
@end

-----


/* Implementation-file Bar.m */
#import "Bar.h"
#import "Foo.h"

@implementation Bar
@synthesize foo;
@end
```

Tabelle 3.20.: Wenn darauf verzichtet wird in beiden zirkulär abhängigen Klasseninterfaces den Quellcode des jeweilig andere Klasseninterface zu importieren, wird dem Compiler also stattdessen mit `@class` lediglich die Information geliefert, dass es sich bei der jeweilig anderen Klasse um einen Klassenbezeichner handelt, und werden aus den jeweilig anderen Klassen keine Implementierungsdetails in den Interfaces benötigt/ benutzt, dann führt ein Compiler-Lauf nicht zum Abbruch.

3.2.2.3. Vererbung

Eine prominente Eigenschaft im Zusammenhang mit objektorientierter Programmierung ist die Vererbung von Klassenfunktionalität von Oberklassen an ihre Unterklassen. Wie in *C++* sind Klassendefinitionen auch in *Objective-C* additiv, d.h. neue Klassen basieren auf der Funktionalität ihrer Oberklassen, erben ihre Methoden und Instanzvariablen und können Funktionalität hinzufügen oder ändern. Anzumerken ist jedoch, dass ausschließlich Methoden neu definiert werden können. Geerbte Instanzvariablen können hingegen nicht überschrieben werden. (vgl. [App11b], S. 19ff)

Im Unterschied zur Sprache *C++* kann jede *Objective-C* Klasse nur eine Oberklasse besitzen. Jede Klasse kann aber die Oberklasse einer beliebigen Anzahl Unterklassen sein. Hieraus ergibt sich für die Vererbung ein hierarchischer Baum mit einer gemeinsamen Oberklasse. So steht zumeist in Code auf Basis des *Cocoa*-Frameworks, die Klasse `NSObject` als Wurzel der Vererbungshierarchie einer beliebigen Klasse (siehe Unterkapitel 4.1.1). Um die fehlende Multiple-Vererbung auszugleichen, kann auf die Objective-C Sprachkonzepte der Kategorien und Protokolle zurückgegriffen werden, welche funktionell den aus *Java* bekannten Interfaces ähneln (siehe Unterkapitel 3.2.5 und 3.2.6). (vgl. [App11b], S. 19ff)

Wie in *Java* und anderen objektorientierten Sprachen gibt es in *Objective-C* die Notion von abstrakten Klassen. Im Unterschied zu anderen Sprachen sieht *Objective-C* aber keine Syntax vor um abstrakte Klassen als solche zu markieren und unterbindet auch nicht ihre Instanziierung. Vielmehr stellen abstrakte Klassen unvollständig implementierte Klassen dar, welche Methoden und Instanzvariablen zu einer praktischen Sammlung von Funktionalität gruppieren und anderen Klassen zur Verfügung stellen. Da abstrakte Klassen Unterklassen haben müssen, die von ihnen erben um nutzbar zu sein, spricht man auch von abstrakten Superklassen. Im *Cocoa-Framework* stellt die Klasse `NSObject` ein gutes Beispiel für eine abstrakte Klasse dar, welche als Basisklasse vieler Unterklassen dient, deren Benutzung als Objektinstanz aber meist wenig Sinn ergibt. Die Klasse `NSView`, welche ebenfalls dem *Cocoa-Framework* entstammt, stellt hingegen das Beispiel einer abstrakten Klasse dar, von welcher aber auch selbständige Objektinstanzen anzutreffen sind. (vgl. [App11b], S. 23)

3.2.2.4. Klassen- und Instanzobjekte

Im Zusammenhang mit Objektinstantiierungen entstehen gerne Missverständnisse beim Begriff Klasseninstanz. Der Begriff „Exemplar einer Klasse" ist nämlich mehrdeutig und kann sowohl als **Klassenobjekt** wie als **Instanzobjekt** verstanden werden. Somit sollten in diesem Zusammenhang ausschließlich die beiden zuletzt genannten Termen verwendet werden (vgl. [App11b], S.19).

Ein Klassenobjekt stellt das einzig zugängliche Objekt einer Klasse dar, welche vom Compiler als kompilierte Version dieser Klasse generiert wird. Dieses Objekt ist selbst vom *Objective-C*-Typ `Class`, wodurch auch Klassenobjekte einer Variablen zugewiesen werden und sie als Empfänger von Nachrichten dienen können – als Folge eines Nachrichtenempfangs wird eine Klassenmethode, welche dem Nachrichtentyp entspricht, aufgerufen (siehe Beispiel 3.21). Klassenobjekte können in *Objective-C* nur Klassenmethoden aber keine Member-Variablen besitzen (siehe Unterkapitel 3.2.3.2). Zudem beherbergt das Klassenobjekt die Logik zur Erstellung neuer Instanzobjekte und wird daher in diesem Kontext auch als *Factory*-Objekt bezeichnet. (vgl. [App11b], S. 19)

```
1 // Variable holding a Class-object
2 Class someClassObject;
3
4 // A method that sets the someClassObject-variable to a given value
5 - (void) setClassObject:(Class)newClassObject{
6   someClassObject = newClassObject;
7   [someClassObject someClassMethod];
8 }
```

Tabelle 3.21.: Beispiel für die Verwendung eines Klassenobjekts. Da Klassenobjekte vom *Objective-C*-Typ `Class` sind, können sie Variablen zugeordnet werden und Nachrichten empfangen. In der sechsten Zeile wird die Variable `someClassObject` mit dem Klassenobjekt `newClassObject` belegt. In der siebten Zeile wird eine Nachricht an die eben neu belegte Variable geschickt, welche den Aufruf der Klassenmethode `someClassMethod` zur Folge hat.

Von einem Klassenobjekt erzeugte Instanzobjekte tragen als Typnamen den Namen ihrer Klasse. Alle Instanzobjekte einer Klasse teilen sich die Methodenimplementierungen, jede Instanz hat aber ihre eigenen Abbilder der Instanzvariablen.

3.2.2.5. Objekterzeugung

Die *Objective-C* Sprachspezifikation beschreibt im Vergleich zu anderen objektorientierten Sprachen nur „was" bei der Objekterzeugung passieren muss. Das „Wie", also die Implementierung, ist weder in der Sprache noch in der Laufzeit vorgegeben, sondern ist Bestandteil des Frameworks und damit direkter Bestandteil des von Anwendungsentwicklern erstellten Programmcodes. Zu diesem Zweck implementiert das zur Erstellung von *iOS*-Anwendungen genutzte Framework *Cocoa Touch* geeignete Methoden zur Erstellung und zum Freigeben von Objekten, welche dem Programmierer zur Verfügung gestellt werden. Der Anwendungsentwickler kann die Vorgänge bei der Objekterzeugung durch die Auslagerung des „Wie" in den Quellcode komplett selbst in die Hand nehmen und hat somit mehr Freiheiten als bei anderen objektorientierten Sprachen. Da die Implementierung im Framework stattfindet, wird in Unterkapitel 4.2.2 näher auf die Objekterzeugung eingegangen.

Die Sprachspezifikation erfordert lediglich, dass bei erfolgreicher Objekterzeugung ein Zeiger auf den Startpunkt des, für dieses Objekt angelegten, Bereichs im virtuellen Speicher einer Applikation zurückgegeben wird. Bei nicht-erfolgreicher Instantiierung wird ein *Nullpointer* zurückerwartet. In *Objective-C* repräsentiert das Schlüsselwort ***nil*** einen Nullpointer und wird bei Zeigern auf Objekte verwendet. Für primitive Typen existiert das aus *C bekannte* Schlüsselwort *NULL*.

3.2.3. Variablen

3.2.3.1. Die Rolle von Variablen in Objective-C

In *Objective-C* wie auch in anderen Sprachen, welche dem objektorientierten Paradigma folgen, bilden Variablen den internen Zustand des Objekts dem sie angehören und die Datenstruktur auf denen Prozeduren dieses Objekts operieren ab. Da es sich bei Objekten sowohl um Klassen- wie auch um Instanzobjekte handeln kann (siehe Unterkapitel 3.2.2.4), findet in vielen objektorientierten Sprachen eine Unterscheidung zwischen Klassenvariablen und Instanzvariablen statt. Während Klassenvariablen für alle Instanzen einer Klasse eine gemeinsame Gültigkeit haben und sie über den Klassenbezeichner angesprochen werden, sind Instanzvariablen nur innerhalb eines Instanzobjekts gültig. Jedes Instanzobjekt besitzt eigenständige Instanzen jeder in der Klasse deklarierter Instanzvariablen.

Die Sprache *Objective-C* verfügt nur über das Konzept von Instanzvariablen. Auch mit Version *2.0* wurden **keine Klassenvariablen** eingeführt. Alternativ kann zur Nachahmung der Funktionalität auf statische Variablen und in geeignetem Fall auf die Präprozessordirektive `#define` zurückgegriffen werden (siehe Unterkapitel 3.2.3.2).

Intern wird der Zustand eines Objekts als *C*-Struktur repräsentiert, welche für jede Klasse die direkten sowie geerbten Instanzvariablen enthält (siehe Beispiel 3.22). Die Reihenfolge der Variablen in der Struktur ergibt sich aus der Vererbungshierarchie, absteigend von der Wurzelklasse hin zur aktuellen Klasse. Wird ein Instanzobjekt erzeugt, wird durch die Laufzeitumgebung im Hauptspeicher Platz für die passende Struktur reserviert. Das Instanzobjekt repräsentiert einen Zeiger auf diese Struktur im Speicher (vgl. [App11b], S. 39 und [Seb11], S. 58f).

```
1 struct ClassA_structure{
2   Class isa;
3   int anInt;
4   id someObject;
5   NSString* aString;
6   float someFloat;
7   BOOL someBool;
8   ClassB* aClassBInstance;
9 };
```

Tabelle 3.22.: Interne *Objective-C* Repräsentation der Speicherstruktur aller direkten und geerbten Instanzvariablen eines Instanzobjekts aus dem Einführungsbeispiel 3.1.1 als *C*-Struktur. Jedes Instanzobjekt besitzt in *Objective-C* eine Variable `isa`, welche auf das Klassenobjekt verweist und aus Gründen der dynamischen Typisierung und Reflexion an oberster Stelle in der Struktur steht (siehe Unterkapitel 3.2.3.3). `Isa` steht für „*is a*", was für ein beliebiges Instanzobjekt angibt, von welchem statischen Klassentyp es ist.

In *Objective-C* können Instanzvariablen sowohl in der Klassendeklaration als auch in der Klassendefinition festgelegt werden. Hierfür stehen drei sich teilweise unterscheidende Methoden zur Verfügung:

1. Bei der ersten, bereits in *Objective-C 1.0* existenten Methode, wird die Deklaration von Instanzvariablen in geschweiften Klammern hinter der Anweisung `@interface` durchgeführt. Die hier deklarierten Instanzvariablen sind innerhalb des Instanzobjekts an beliebiger Stelle nutzbar und je nach Gültigkeitsbereich auch von außerhalb zugänglich. Die Definition von Instanzvariablen ist nur innerhalb von Methodenimplementierungen möglich.

2. Die zweite Methode datiert aus der Zeit vor *Objective-C* zurück und ist durch den *C*-Unterbau der Sprache möglich. In der Klassendeklaration können Variablen an beliebiger Stelle außerhalb des durch `@interface` und `@end` begrenzten Bereichs deklariert und auch definiert werden. Innerhalb des Bereichs können Variablen nur wie in der ersten Methode in geschweiften Klammern oder wie in der dritten Methode beschrieben, deklariert werden.

   In der Klassenimplementierung hingegen können Variablen auf diese Art an beliebiger Stelle außerhalb der geschweiften Klammern des Variablendeklarationsblocks deklariert und definiert werden.

   Die auf diese Art im Klasseninterface oder in der Klassenimplementierung deklarierten oder definierten Variablen können in Methoden ganz normal verwendet werden. So deklarierte oder definierte Variablen können außerhalb von Methodenimplementierungen allerdings nicht bei der Definition weiterer Variablen genutzt werden – dies hat einen Compilerfehler zur Folge. Anzumerken ist, dass auf die hier beschriebene Weise deklarierte oder definierte Variablen solche Variablen, die nach der ersten oder dritten Methode festgelegt werden, überdecken. D.h. gibt es eine auf die erste Weise deklarierte Variable `foo` und eine nach der zweiten Methode gleichnamigen Variable, so ist nur die letztere sichtbar. Auch dann wenn die Definition der Variablen nach der zweiten Art programmablauftechnisch vor der Deklaration der Inkarnation von `foo` erfolgt, welche nach der ersten Art erstellt wurde.

   Die hier beschriebene Methode kann in manchen Fällen nützlich sein, zum Beispiel bei der Emulation von Klassenvariablen (siehe Unterkapitel 3.2.3.2). Nach Meinung des Autors sollte bis auf diese Ausnahmesituationen auf die Verwendung dieser Methode verzichtet werden. Stattdessen sollten Variablen anhand der ersten und dritten Methode festgelegt werden, da dies durch die Umsetzung des objektorientierten Ansatzes durch *Objective-C* so eingeführt wurde und die in dieser Methode beschriebene „*legacy C*"-Variante zu Verständnis- und Lesbarkeitsproblemen sowie zu Fehler führen kann.

3. Seit *Objective-C 2.0* kann die Deklaration von Instanzvariablen auch anhand des ***Property***-Konzepts erfolgen (Details in Unterkapitel 3.2.3.5). Dieses Konzept beruht auf dem Abkapselungsprinzip, welches vorgibt, dass nur diejenigen Implementierungsdetails nach außen hin sichtbar sein sollen, die auch wirklich extern benötigt werden. Weitere Aspekte dieses Prinzips, das auch *Information Hiding* genannt wird, sind die Implementierung von Akzessor-Methoden für jede nach außen hin zugänglich zu machende Variable und die Unterbindung eines direkten, externen Zugriffs auf die Variablen durch Einschränkung der Sichtbarkeit (vgl. [Som07], S. 507f). Mit dem Schlüsselwort `@property` werden *Properties* innerhalb eines Klasseninterfaces deklariert. In der Klassenimplementierung muss der Compiler oder der Programmierer dafür Sorge tragen, dass entsprechende Akzessor-Methoden, gemeinhin auch *Getter*- und *Setter*-Methoden genannt, für jede *Property* existieren.

Auch der Zugriff auf Instanzvariablen eines Instanzobjekts kann auf verschiedene Arten erfolgen:

1. Bei Instanzvariablen, die nach der ersten oder zweiten Methode festgelegt wurden, erfolgt der Zugriff ausschließlich anhand des aus *C* bekannten Pfeiloperators „->". Beim Zugriff auf Variablen von Objekten externer Klassen, muss der Bezeichner des Objekts explizit vor dem Pfeiloperator und dem Variablennamen angegeben werden (siehe Beispiel 3.23, Zeile 11). Entfallen kann die Angabe des Pfeiloperators hingegen beim Zugriff auf Instanzvariablen, welche innerhalb der Klasse deklariert sind (siehe ebd., Zeile 14). Im zweiten Fall kann als Objektbezeichner vor dem Pfeiloperator optional auch das Schlüsselwort `self` stehen, welches innerhalb eines Instanzobjekts eben dieses Instanzobjekt referenziert (siehe ebd., Zeile 15). Anzumerken ist, dass wenn das Schlüsselwort `self` nicht explizit angegeben ist, es trotzdem implizit angenommen wird und vom Compiler bei der Kompilierung eingefügt wird. Die drei hier beschriebenen Fälle werden in Beispiel gezeigt.

```
1  #import "ClassB.h"
2  #import "ClassA.h"
3
4  @implementation ClassB
5  {
6    float aFloat;
7  }
8
9  - (void) test:(ClassA *)classAInstance{
10   // Accessing an external member-variable
11   classAInstance->anInt = 5;
12
13   // Accessing a local member-variable in two different ways with the same result
14   aFloat = 1.0;
15   self->aFloat = 2.0;
16 }
17 @end
```

Tabelle 3.23.: Multiple Einsatzszenarien für den Pfeiloperator.

2. Auf eine Instanzvariable die über die dritte Methode deklariert wurde, also durch die Deklaration einer *Property* die diese Instanzvariable kapselt, kann direkt und indirekt zugegriffen werden. Für den direkten Zugriff auf die Variable wird der Pfeiloperator verwendet. Das Schlüsselwort `self` kann angegeben werden. Die Akzessor-Methoden werden in diesem Fall umgangen.

   Ein direkter Zugriff über den Pfeiloperator ist aber nur innerhalb der Klasse in der die *Property* deklariert wurde möglich, die gekapselte Variable wird bezüglich ihrer Sichtbarkeit eingeschränkt. Um von außerhalb der Klasse auf die Variable zuzugreifen, müssen die Akzessor-Methoden benutzt werden (siehe Unterkapitel 3.2.3.4). Hierzu wird die *Dot*-Notation in Kombination mit dem *Property*-Bezeichner verwendet (siehe Unterkapitel 3.2.3.5).

3.2.3.2. Emulation von Klassenvariablen

Manche Programmiersituationen erfordern Variablen die nicht an Instanzen sondern an Klassen gebunden sind. Solche Variablen werden Klassenvariablen genannt. Während objektorientierte Sprachen wie *Java* dieses Konzept in der Sprachspezifikation unterstützen, kennt *Objective-C* dieses Konzept nicht[5]. Da aber jedes Klassenobjekt für jede Klasse in *Objective-C* einmalig ist, kann anhand des *C*-Schlüsselworts `static`, das auch in *Objective-C* verwendbar ist, ein ähnliches Konzept realisiert werden.

`Static` ist ein in *C* reserviertes Schlüsselwort und stellt eine *Storage*-Klasse dar, welche eine Aussage über die Lebensdauer und Gültigkeit von damit versehenen Variablen und Methoden macht. Statische Methoden und Variablen in Sprachen aus der *C*-Familie, zu der auch *Objective-C* gehört, existieren vom Laden der Klasse bis zum Beenden des Programms.

Statische Variablen lassen sich in *Objective-C*, wie in der zweiten Methode zur Deklaration von Instanzvariablen beschrieben, an beliebiger Stelle außerhalb des durch `@interface` und `@end` begrenzten Bereichs deklarieren und definieren (siehe Unterkapitel 3.2.3.1). Innerhalb der Implementierungsdatei einer Klasse können statische Variablen überall deklariert und definiert werden, außer im durch geschweifte Klammern begrenzten Bereich für die Deklaration von Instanzvariablen. Eine solche Deklaration/ Definition unterscheidet sich syntaktisch nur durch das vorgesetzte Schlüsselwort `static`.

Eine Einschränkung gilt jedoch für statische Variablen welche als Typ ein Objekt haben. Die Deklaration einer solchen statischen Variablen kann überall erfolgen, deren Definition aber nicht. Die Definition einer statischen Variable vom Typ eines Objekts muss zwingend in einer Klassenmethode erfolgen. Meist empfiehlt sich die Definition solcher statischen Variablen innerhalb der Klassenmethode `initialize`, welche von der *Objective-C* Sprachspezifikation für jedes, auf *Objective-C* basierendes, Framework vorgeschrieben wird (siehe Unterkapitel 4.2.2). Die `initialize`-Methode wird nach der Speicherallokation eines Klassenobjekts aufgerufen. Die Speicherallokation erfolgt für jede während der Programmlaufzeit benutzten Klasse einmalig.

Statische Variablen die innerhalb der Implementierung einer Klassen- oder Instanzmethoden deklariert werden, haben einen Gültigkeitsbereich, der sich von der Deklarationsstelle innerhalb der Methode bis zum Methodenende erstreckt. Außerhalb dieser Methode ist die Variable ungültig. Der Wert der Variable bleibt beim Verlassen der Methode erhalten und kann bei einem weiteren Aufruf dieser Methode mit gleichem Wert weiterverwendet werden.

Insbesondere bei der Nutzung von statischen Variablen durch Instanzmethoden ist auf unbeabsichtigte *Race Conditions* zu achten. Aus diesem Grund sollte der Programmierer für einen threadsicheren Zugriff auf statische Variablen anhand von Semaphoren sorgen. Außerdem sollte, auch bei Verwendung von statischen Variablen, mit der Idee des *Information Hiding*s im Hinterkopf und um unerwünschten Nebenwirkungen vorzubeugen, nicht auf den Einsatz geeigneter Akzessor-Methoden verzichtet werden.

Will der Programmierer hingegen reine Konstanten, wie die aus der Programmierung mit *Java* bekannten Variablen mit den Deklarationsmodifikatoren `static final`, in *Objective-C* festlegen, kann er statt eine statische Variable zu definieren, die Präprozessordirektive `#define` nutzen.

[5] Dies ist insbesondere im Hinblick darauf überraschend, dass *Objective-C* das Konzept von Klassenmethoden kennt (siehe Unterkapitel 3.2.4).

3.2.3.3. Typen

Variablen unterscheiden sich nicht nur durch ihre Bezeichner sondern auch durch ihren Typ. *Objective-C* bietet die folgende Auswahl an Typen:

1. *Objective-C* unterstützt als primitive Typen alle aus *ANSI C* bekannten Datentypen und fasst sie unter dem Begriff *Plain Old Data* (kurz *POD*-Typen) zusammen (vgl. [App11b], S.69). Hierunter fallen u.a. die Ganzzahl-Typen `short`, `int` und `long`, jeweils in einer Variante mit und einer ohne Vorzeichen. Zudem enthalten sind Datentypen für Gleitkommazahlen wie `float` und `double`, auch jeweils in einer Variante mit und ohne Vorzeichen. Außerdem fallen noch Typen wie `char` und dem später zur *ANSI C* Spezifikation hinzugefügten Typ `_Bool` unter diese Bezeichnung. Überdies kennt *Objective-C* auch Aufzählungstypen wie *Enumerations* (`enum`), aggregierte Klassen (struct) und aggregierte Vereinigungen (union) sowie den aus *C++* bekannten Typ **`bool`**. Letzterer ist wie `_Bool` fester Bestandteil aller neueren *Objective-C*-Compilern (vgl. [Seb11], S. 30).
2. Zusätzlich zu den logischen Variablentypen `_Bool` und `bool` bietet *Objective-C* noch den Datentyp `BOOL`. Dieser Typ stellt, wie auch die Typen `_Bool` und `bool`,nur eine Typdefinition auf `signed char` dar (siehe 3.24). Es werden zudem zwei Konstanten für die Wahrheitswerte „Wahr" (`YES`) und „Falsch" (`NO`) definiert.

```
1 typedef signed char BOOL;
2
3 #define YES (BOOL)1
4 #define NO (BOOL)0
```

Tabelle 3.24.: Definition des logischen *Objective-C*-Datentyps `BOOL`, sowie der Wahrheitswerte `YES` und `NO`.

   Die drei logischen Typen unterscheiden sich jedoch durch ihre Implementierung im Compiler. Durch die Typdefinition von `BOOL` als `signed char`, kann einer als `BOOL` typisierten Variable jeder Wert zugewiesen werden, der auch einer Variable vom Typ `signed char` zugewiesen werden kann. Dieses Verhalten birgt den Nachteil, dass sich durch Unaufmerksamkeiten des Programmierers Fehler in Bezug auf Wahrheitswerte vom Typ `BOOL` in ein Programm einschleichen können. So führt ein Vergleich zwischen einer `BOOL`-Variable mit Wert größer Eins und dem Wahrheitswert `YES` zum Ergebnis, dass beide nicht identisch sind. Grund dafür ist, dass `YES` als genau Eins dargestellt wird. Bei den Typen `_Bool` und `bool` wird ein solches, unerwartetes Verhalten dadurch entschärft, dass einer Wertzuweisung an eine so typisierte Variable stets eine implizite Überprüfung vorgeschaltet ist. Es wird geprüft ob der zuzuweisende Wert größer Eins ist, falls ja wir der Variable der Wert von `true` zugewiesen, genau Eins. Ist ein zuzuweisender Wert kleiner Null, wird die Variable auf den Wert von `false`, genau Null, gesetzt. (vgl. [Seb11], S. 29ff)
3. Als objektorientierte Sprache unterstützt *Objective-C* für Variablen neben primitiven Datentypen auch Objekttypen. Die Variable stellt dann einen Zeiger auf den Anfang des Speicherbereichs dar, in dem das Objekt abgelegt ist. Bei Objekttypen in *Objective-C* kommt die An-

gliederung an *SmallTalk* stark zum Vorschein, denn neben einer statischen Typisierung wie in *C/C++* oder *Java* unterstützt *Objective-C* auch eine dynamische Typisierung.

a) Bei der **statischen Typisierung** wird der Bezeichner einer Klasse oder eines formellen Protokolls (siehe Unterkapitel 3.2.6), also der Klassen- bzw. Protokollname, als Typbezeichner verwendet (siehe Beispiel 3.25). Die Bekanntmachung der konkreten Klasse eines Objekts ermöglicht dem Compiler mehr Informationen über das zu verwaltende Objekt zu sammeln. Diese enthalten u.a. den Klassen-/ Protokollnamen, die Superklassen der Klasse, die darin deklarierten und geerbten Instanzvariablen sowie die Deklaration und Implementierung aller (geerbten) Methoden dieser Klasse samt Rückgabetyp und Parametertypen. Die Informationen können vom Compiler verwendet werden um u.a. vor der Laufzeit Typprüfungen durchzuführen und sicherzustellen, dass im Quellcode aufgerufene Methoden in den zugehörigen Klassen**-** oder Instanzobjekten schon vor der Laufzeit **i**mplementiert sind. (vgl. [App11b], S. 23f)

   Syntaktisch folgt einem statischen Typbezeichner ein Stern-Zeichen '*', als Symbol dafür, dass es sich bei der Variable um einen Zeiger auf eine Speicherstelle handelt.

```
1 ClassB *aClassBInstance;
```

Tabelle 3.25.: Eine dem Einführungsbeispiel 3.1.1 entnommene, statisch typisierte Instanzvariable vom Objekttyp `ClassB`.

b) Bei der **dynamischen Typisierung** hingegen wird der exakte Objekttyp der Variable nicht genannt. Stattdessen kann für jedes Klassenobjekt einer beliebigen Klasse der Datentyp `Class`, für jedes Protokollobjekt eines beliebigen Protokolls der Datentyp `Protocol` und allgemeiner für jedes Klassen-, Protokoll- oder Instanzobjekt einer beliebigen Klasse/ eines beliebigen formellen Protokolls der Datentyp **`id`** verwendet werden. Hierbei handelt es sich um generische Zeiger auf eine Speicherstelle, wobei die konkrete Klassenimplementierung, die statisch gesehen hinter dem Variablentyp steht, ignoriert wird. Da die Typbezeichner `Class`, `Protocol` und `id` die unter (a) genannten klassenspezifischen Informationen dem Compiler vorenthalten, muss jedes Objekt die benötigten Informationen zur Laufzeit liefern. Um zur Laufzeit an die benötigten Informationen zu gelangen, kann eine *Objective-C*-Laufzeitumgebung auf den, durch die Sprachspezifikation fest vorgegebenen, `isa`-*Pointer* zugreifen. (vgl. [App11b], S. 11)

   Der Begriff `isa`-*Pointer* stammt von der `isa`-Variable, welche den existenziellen Kern der Datenstruktur hinter einem Objekt vom Typ `id` ausmacht (siehe Typdefinition 3.26). Jedes Objekt in *Objective-C* hat an erster Stelle seiner, ihm zugrunde liegenden, *C*-Datenstruktur, die Variable **`isa`**, welche vom Typ `Class` ist (siehe Codelisting 3.26) und die Relation eines Objekts zu seiner Klasse festhält (siehe Unterkapitel 3.2.3.1 und das dort aufgeführte Beispiel 3.22). Da auch der dynamische Datentyp `Class` als Zeiger definiert ist (siehe Typdefinition 3.27), wird statt von `isa`-Variable auch von `isa`-*Pointer* gesprochen. (vgl. [App11b], S. 11f)

```
1 typedef struct objc_object{
2   Class isa;
3 } *id
```

Tabelle 3.26.: Typdefinition des dynamischen Objekttyps `id`, welcher für ein beliebiges Instanzobjekt stehen kann. Der Variablenname `isa` entstammt dem Englischen *„is a"* und spricht die Beziehung zwischen Objekt und der Klasse des Objekts aus.

```
1 typedef struct objc_class *Class;
```

Tabelle 3.27.: Typdefinition des dynamischen Objekttyps `Class`, welcher für ein beliebiges Klassenobjekt stehen kann.

Greift die Laufzeitumgebung bei Programmausführung auf den `isa`-*Pointer* zu, erfährt sie um welche explizite Klasse es sich bei einem Objekt vom Typ `id` oder `Class` handelt. Hierdurch wird die Datenstruktur und die durch die Klasse unterstützten Prozeduren aufgedeckt. Das Konzept der dynamischen Typisierung dient als Fundament der **dynamischen Bindung**, einem weiteren aus *SmallTalk* entliehenen Konzept (siehe Unterkapitel 3.2.4.1). (vgl. [App11b], S. 11f)

Außerdem spielt der `isa`-Pointer in der Erfüllung anderer Konzepte wie z.Bsp. der ***Reflexion*** – auch **Introspektion** genannt – eine zentrale Rolle. Anhand dieses Konzepts kann ein Programmierer in seiner Anwendung z.Bsp. die Anfrage an die Laufzeitumgebung stellen ob eine Klasse eine bestimmte Methode implementiert, welche die Superklasse einer Klasse ist oder ob eine Klasse ein gewisses Protokoll unterstützt. Dies wird durch die Informationen über ein Objekt ermöglicht, die die Laufzeitumgebung über die `isa`-Variable des Objekts erhält und zur Laufzeit an den Programmierer weitergeben kann (für weitere Details zum Thema Introspektion siehe u.a. [App11b], S. 24f).

Zusätzlich ist anzumerken, dass der dynamische Typ `id` den A*NSI-C*-Typen `int` als Standardtyp für objektorientierte *Objective-C*-Konstrukte, wie z.Bsp. Methodenrückgabewerte, ersetzt[6] (vgl. [App11b], S. 11).

Abschließend ist anzumerken, dass auch dynamisch typisierte Objekte in *Objective-C* stets einer konkreten Klasse angehören. *Objective-C* typisiert also streng, aber dynamisch.

3.2.3.4. Gültigkeitsbereiche

Zusätzlich zum Bezeichner und Datentyp können sich Variablen auch durch ihren Gültigkeitsbereich voneinander unterscheiden. Der Gültigkeitsbereich einer Variable legt den Programmabschnitt fest, in dem die Variable sichtbar und nutzbar ist.

[6] Für strikte *C*-Konstrukte bleibt `int` aber auch weiterhin der Standarddatentyp.

Die Sichtbarkeit von Instanzvariablen lässt sich auf zwei Arten festlegen, je nachdem ob die Variable in dem mit geschweiften Klammern umgebenen Bereich oder über das Konzept der Properties deklariert wird (siehe erste respektive dritte Methode zur Deklaration von Variablen in Unterkapitel 3.2.3.1).

Werden Instanzvariablen nicht durch *Properties* festgelegt, kann ihr Gültigkeitsbereich feingranular vom Programmierer anhand geeigneter Schlüsselwörter festgelegt werden. Für alle Variablen, die auf ein solches Schlüsselwort folgen, gilt dieser Sichtbarkeitsbereich, bis zur Angabe eines anderen Sichtbarkeitmodifikators. Objective-C unterstützt die folgenden vier Gültigkeitsbereiche (siehe Abb. 3.2):

- Durch das Schlüsselwort **`@private`** wird die Sichtbarkeit einer Instanzvariable auf die aktuelle Klasse reduziert. Methodenimplementierungen dieser Klasse können darauf zugreifen. Der Zugriff aus Unterklassen und externen Klassen ist hingegen untersagt.
- Auf Instanzvariablen mit dem Sichtbarkeitsbereich **`@protected`** kann über den von `@private` vorgeschriebenen Gültigkeitsbereich hinaus, auch aus Methodenimplementierungen aus Unterklassen zugegriffen werden.

  Wird kein Schlüsselwort bei der Deklaration einer Instanzvariable angegeben, wird vom Compiler implizit `@protected` angenommen. Aus softwaretechnischer Sicht ist diese Restriktion aus Gründen des Abstraktionsprinzips sinnvoll. Programmkomponenten sollten nur auf Daten zugreifen können, die sie zu ihrer Implementierung wirklich benötigen und dieser Zugriff sollte wenn möglich über Akzessor-Methoden abgesichert sein. Das Verbergen von Informationen soll verhindern, dass Klassenkomponenten von außenstehenden Programmkomponenten verfälscht werden, für die sie nicht vorgesehen sind. (vgl. [Som07], S. 507f)
- Soll der Zugriff auf eine Instanzvariable aus jeder externen Klasse ermöglicht werden, die die Klasse in der die Variable deklariert wird importiert, muss das Schlüsselwort **`@public`** verwendet werden.

  Mit Verweis auf das Abstraktionsprinzip, das bereits im Abschnitt über den Sichtbarkeitsmodifikator `@protected` erwähnt wurde, ist davon abzuraten den direkten Zugriff auf Instanzvariablen zu erlauben, da ein falscher Zugriff zu unvorhergesehenen Problemen und schwer nachvollziehbaren Fehlern während der Laufzeit führen kann. Stattdessen sollten Variablen möglichst mit den anderen Gültigkeitsbereichsmodifikatoren versehen und Informationen über entsprechende Akzessor-Methoden zugänglich gemacht werden, welche von außenstehenden Klassen benötigt werden. Akzessor-Methoden erlauben vor einer Preisgabe oder Veränderung der gekapselten Information u.a. die Zugriffsberechtigung auf diese Information zu Überprüfen, die Rückgabe ggf. an den Anfragesteller anzupassen oder Plausibilitätstests auf zuzuweisenden Werten durchzuführen und gegebenenfalls die Zuweisung zu verweigern oder abzuändern.
- Nur für die Verwendung in 64-Bit Programmen ist der Gültigkeitsbereichsmodifikator **`@package`** vorgesehen. Hiermit versehene Variablen haben innerhalb des Programmkompilats dieselbe Gültigkeit als hätten sie den Gültigkeitsbereichsmodifikator `@public`. Außerhalb des Kompilats verhalten sie sich hingegen wie Variablen mit einer Sichtbarkeit von `@private`. Die Benutzung dieser Sichtbarkeitsstufe ist vor allem für Variablen in Framework-Klassen

nützlich, wo `@private` zu restriktiv aber `@protected` und `@public` zu freizügig sind. (vgl. [Seb11], S. 57 und [App11b], S. 40)

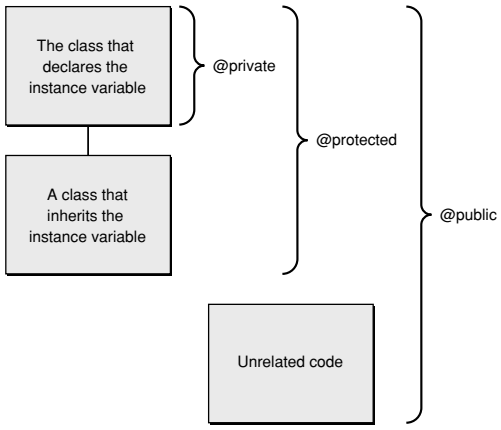

Abbildung 3.2.: Übersicht der Gültigkeitsmodifikatoren in *Objective-C.* (Bildquelle: [App11b], S. 41)

Werden Instanzvariablen hingegen anhand von *Properties* (siehe Unterkapitel 3.2.3.5) festgelegt, gelten etwas andere Regeln für ihren Sichtbarkeitsbereich. Für durch *Properties* gekapselte Variablen, welche in einem öffentlichen Interface deklariert werden, gilt, dass kein direkter Zugriff anhand des Pfeiloperators aus externen Klassen möglich ist (siehe Unterkapitel 3.2.3.1). Für den Zugriff müssen stattdessen die dafür vorgesehenen Akzessor-Methoden genutzt werden. Werden *Properties* hingegen in einer Klassenerweiterung deklariert (siehe Unterkapitel 3.2.5.2), kann auf die, durch eine *Property*, gekapselte Variable sowohl über die Akzessor-Methoden wie auch über den Pfeiloperator direkt zugegriffen werden. Dies ist dem Umstand zuzuschreiben, dass durch *Properties* gekapselte Variablen bei ihrer Definition implizit mit dem Gültigkeitsbereichsmodifikator `@private` versehen werden und somit ein direkter Zugriff innerhalb der Klasse möglich ist.

3.2.3.5. Properties und die Dot-Notation

Die Rolle von Properties:

Mit der Veröffentlichung von *Objective-C 2.0* wurde die bis dahin übliche Deklaration und Definition von Instanzvariablen (siehe erste und zweite Methode in Unterkapitel 3.2.3.1) an das Prinzip des *Information Hidings* angepasst. Dieses fundamentale softwaretechnische Prinzip, auch als Abstraktionsprinzip bekannt, schreibt vor, das Programmkomponenten nur auf diejenigen Daten Zugriff haben sollten, welche sie für ihren korrekten Funktionsablauf benötigen. Im Umkehrschluss sollen Programmkomponenten nach außen hin nur die Informationen preisgeben, welche unabdingbar für die Implementierung anderer Klassen sind. Als zusätzliches Sicherheitskonzept sollte der Zugriff nicht direkt auf die freigegebenen Informationen möglich sein, sondern stattdessen über Akzessor-Methoden erfolgen. Solche landläufig *Getter-* und *Setter*-Methoden genannte

Akzessor-Methoden können dann vor dem Zugriff auf die Variable u.a. überprüfen, ob der Anfragesteller Zugriffsrechte hat, ggf. die Rückgabe an den Anfragesteller anpassen oder Plausibilitätsprüfungen des, der Variable zuzuweisenden, Wertes durchführen und ggf. Anpassungen vornehmen. So soll die Sicherheit und Konsistenz des internen Zustands eines Objekts gewahrt werden. (vgl. [Som07], S. 507f)

Obwohl die Benutzung solcher *Getter-* und *Setter*-Methoden die Sicherheit erhöht, stellt ihre manuelle Erstellung durch den Programmierer eine lästige Arbeit dar. Zudem sind unter Umständen wichtige Implementierungsdetails, also die internen Abläufe, dieser Akzessor-Methoden in der öffentlich zugänglichen Methodendeklaration für die Benutzer der Methoden oft nicht ersichtlich. Es ist der Deklaration einer Akzessor-Methode zum Beispiel nicht anzusehen, ob ihre Implementierung thread-sicher ist, oder ob ein zuzuweisender Wert der gekapselten Variable sofort zugewiesen, oder zuvor kopiert und dann die Kopie zugewiesen wird.

Um die beschriebenen Bedenken zu adressieren, wurde in *Objective-C 2.0* das Konzept der *Properties* in die Sprachspezifikation eingeführt. Der Name soll darauf hinweisen, dass eine Variable als Teil des internen Zustandes eines Objekts aus softwaretechnischer Sicht eine Eigenschaft (engl. *Property*) dieses Objekts beschreibt. *Properties* bieten folgende Vorteile: (vgl. [App11b], S. 62)

- *Property*-Deklarationen bieten eine klare, explizite Spezifikation der Verhaltensweise von Akzessor-Methoden (siehe Paragraph „*Property*-Attribute" für Details),
- der Compiler kann dem Programmierer zur Compilezeit die Arbeit abnehmen die Akzessor-Methoden gemäß der Spezifikation zu implementieren (siehe Paragraph „Erstellen von *Properties*" für Details), und
- *Properties* werden syntaktisch als Bezeichner dargestellt und sie haben einen Gültigkeitsbereich, so dass der Compiler die Benutzung nicht deklarierter *Properties* im Quellcode zur Compilezeit erkennen kann.

Erstellen von Properties:

Properties werden in zwei Schritten angelegt. Zuerst müssen sie deklariert und anschließend implementiert werden[7]. *Properties* werden bei geeigneter Sichtbarkeit an Unterklassen der Klasse, in der sie deklariert wurden vererbt. Dort können sie neu deklariert und sogar neu definiert werden.

Die Deklaration von Properties kann innerhalb eines Klasseninterfaces (`@interface`, siehe Beispiel 3.29) oder eines Protokolls (`@protocol`, siehe Unterkapitel 3.2.5) erfolgen. In Klassendeklarationen kann die Deklaration von Properties an beliebiger Stelle zwischen den Schlüsselwörtern `@interface` und `@end`, aber außerhalb des optionalen mit geschweiften Klammern umgebenen

[7] In alten Beispielen zu *Objective-C*, in denen die alte Laufzeitumgebung verwendet wurde, beinhaltet dieser Vorgang einen weiteren Schritt: das Anlegen der Instanzvariable welche von der *Property* gekapselt wird. Mit der Umstellung des Compilers und der Laufzeitumgebung vom *GCC* zur ***Apple-LLVM*** entfiel der Schritt. Ab *Xcode 4* sind der *Apple-LLVM*-Compiler und die *Apple-LLVM*-Laufzeitumgebung die Standardwerkzeuge. Auf alten Plattformen ist der Schritt aber noch nötig, insbesondere bei der Entwicklung von *32-bit*-Anwendungen auf einer *Mac-OS-X-Version* älter als Version *10.5*, wo heute noch die alte Laufzeitumgebung zum Einsatz kommt. (vgl. [App11b], S. 68 und [App09b], S. 7 sowie [Dav11], S. 50)

Blocks für die Deklaration von Instanzvariablen, erfolgen. In Protokollen erfolgt die Deklaration von Properties an beliebiger Stelle innerhalb des Bereichs welcher durch die Schlüsselwörter `@protocol` und `@end` begrenzt wird.

Das Aussehen einer *Property*-Deklaration folgt dem Schema in Codelisting 3.28 und sie besteht aus den folgend beschriebenen Teilen:

```
1 @property (attribute1, attribute2, ...) type name, name2, ...;
```

Tabelle 3.28.: Schematisches Aussehen einer *Property*-Deklaration.

- die Anweisung beginnt mit dem Schlüsselwort **`@property`**,
- es folgt eine optionale, kommaseparierte Liste von *Property*-Attributen, die die Zugriffsmethoden für *Properties* näher definieren (Details im Paragraphen „*Property*-Attribute" dieses Unterkapitels),
- dann folgt der Typ[8] der *Property* oder *Properties* die mit der aktuellen Deklaration festgelegt wird/ werden,
- anschließend folgt der *Property*-Bezeichner oder eine durch Kommas getrennte Aufzählung der Bezeichner, wenn die Deklaration mehrerer *Properties* mit gleichen Attributen und Typ in einer Anweisung vorgenommen werden soll, und
- abschließend folgt im Vergleich zu anderen in *Objective-C* eingeführten Kommandos die mit `@` anfangen, ein Semikolon das das Ende der Anweisung markiert.

```
1 @property(nonatomic, copy) NSString *aString;
```

Tabelle 3.29.: Deklaration einer *Property* für eine Zeichenkette aus dem Einführungsbeispiel 3.1.1. Das Attribut `nonatomic` gibt vor, dass der Zugriff auf die Variable durch die Akzessormethoden nicht thread-sicher implementiert ist. Das Attribut `copy` hingegen besagt, dass eine Kopie des an die Instanzvariable übertragenen Strings gespeichert werden soll. Es wird in diesem Fall eine neue Instanz der Zeichenkette im Speicher abgelegt und unter diesem Variablenbezeichner referenziert, statt auf die Speicherstelle der ursprünglich zugewiesenen Zeichenkette zu verweisen.

Die Implementierung von Properties erfolgt stets in einer Klassenimplementierung, zwischen den Schlüsselwörtern `@implementation` und `@end`. Die Auswirkung einer *Property*-Implementierung ist die Erstellung der zu kapselnden Variable sowie entsprechender Akzessor-Methoden für den Zugriff auf diese Variable. Eine *Property*-Definition kann auf drei Weisen vorgenommen werden: (vgl. hierzu auch [App11b], S. 62ff)

[8] Für eine *Property* kann jeder auch für Instanzvariablen gültige Typ verwendet werden (siehe Unterkapitel 3.2.3.3).

1. *Properties* können in der Klassenimplementierung mit dem Schlüsselwort **`@synthesize`** implementiert werden. Zur Compilezeit erstellt der Compiler dann **automatisch** die zu kapselnde Variable sowie entsprechende Akzessor-Methoden. Dabei wird auf die Einhaltung der angegeben Attribute geachtet. *Properties* können nur ein Mal innerhalb derselben Klassenimplementierung definiert werden.

   Die Namen der Akzessor-Methoden, welche mit einer *Property* assoziiert sind, orientieren sich an dem Bezeichner der *Property*. *Getter*-Methode tragen nur den Namen des *Property*-Bezeichners. *Setter*-Methoden wird dem *Property*-Bezeichner der Zusatz `set` vorangestellt. Für eine *Property* mit dem Bezeichner `aString` (siehe Beispiel 3.30) wird zum Beispiel eine *Getter*-Methode namens `aString` und eine *Setter*-Methode `setAString:` erstellt (siehe Beispiel 3.30). Auf die Namensgebung der generierten Methoden kann anhand geeigneter *Property*-Attribute Einfluss genommen werden (siehe Paragraph „*Property*-Attribute" in diesem Unterkapitel).

```
1 - (NSString *) aString;
2 - (void) setAString:(NSString *) newAString;
```

Tabelle 3.30.: Auswirkung der Implementierung der Property aus 3.29 mit dem Schlüsselwort `@synthesize`. Der Compiler erstellt diese Akzessor-Methoden zur Compilezeit und verhindert direkte Zugriffe auf die Variable von außerhalb der Klasse.

Wird eine *Property* in einer Klassendeklaration deklariert und wird die Definition der Property durch Angabe von `@synthesize` dem Compiler überlassen, sind die automatisch erstellten Akzessor-Methoden innerhalb der Klasse, aus Subklassen heraus und von jeder externen Klasse aus aufrufbar. Auf die, durch die *Property* gekapselte, Instanzvariable kann aber von außen und aus Unterklassen nicht direkt mit dem Pfeiloperator (siehe Unterkapitel 3.2.3.1) zugegriffen werden. Hierfür sorgt die implizit vom Compiler für die Variable angegebene `@private`-Sichtbarkeit (siehe Unterkapitel 3.2.3.4). Innerhalb der Klasse ist der direkte Zugriff möglich.

Properties können in Unterklassen einer Klasse und Kategorien, welche die Funktionalität einer Klasse erweitern (siehe Unterkapitel 3.2.5), neu deklariert werden. Abweichungen bei den Attributen zur ursprünglichen Deklaration sind zulässig. Diese Möglichkeit lässt sich zum Beispiel in dem häufig angewandten Szenario nutzen, bei dem der Zugriff auf eine *Property* nach außen nur lesend und innerhalb der Klasse schreibend ermöglicht werden soll (vgl. [App11b], S. 74). Hierzu wird die *Property* in der Deklaration der Klasse mit dem Attribut `readonly` versehen und in einer Klassenerweiterung (siehe Unterkapitel 3.2.5.2) innerhalb der Klassenimplementierung mit dem Attribut `readwrite` neu deklariert (siehe Beispiel 3.31).

```
/* Excerpt from the Header-file "ClassA.h" */
...

@interface ClassA : NSObject <ProtocolX>

...

@property(strong, readonly) NSString *anotherString;

...

@end


-----


/* Excerpt from the Implementation-file "ClassA.m" */
#import "ClassA.h"

...

@interface ClassA()

...

@property(strong, readwrite) NSString *anotherString;

...

@end

@implementation ClassA

...

@synthesize aString, someBool, anotherString;

...

@end
```

Tabelle 3.31.: Neudeklaration einer *Property* aus dem Einleitungsbeispiel *3.1.1.*

Properties können in Subklassen außerdem auch neu definiert werden, wodurch die Definition aus einer Oberklasse überschrieben wird. In der Subklasse werden Akzessor-Methoden generiert, die unabhängig von den *Getter*- und *Setter*-Methoden für die namensgleiche *Property* aus der Oberklasse existieren. Im Grunde handelt es sich dann um zwei Instanzvariablen mit gleichem Bezeichner, welche parallel in der Oberklasse und der Unterklasse existieren. Mit dem Schlüsselwort `super` kann auf die Akzessor-Methoden der Oberklasseninstanzvariable zugegriffen werden. Ein direkter Zugriff auf die Variable in der Oberkasse ist

aber weiterhin, wie im vorletzten Abschnitt beschrieben, unmöglich.

Abschließend ist anzumerken, dass wenn eine Property mit `@synthesize` definiert wird, der Programmierer aber zugleich Akzessor-Methoden für diese *Property* von Hand angelegt (siehe dritte Methode), der Compiler die bereits von Hand angelegten Akzessor-Methoden zur Compilezeit nicht generiert/ überschreibt. D.h. beim Aufruf einer Akzessor-Methode einer, diesen Kriterien entsprechenden, *Property* wird nicht etwa eine generierte, sondern die von Hand angelegte Methodenimplementierung ausgeführt.

2. Alternativ zur Synthese können *Properties* auch mit dem Schlüsselwort **`@dynamic`** definiert werden. Dieses Schlüsselwort und seine Wirkung sind darauf zurückzuführen, dass *Objective-C* in der Bestrebung Eingenschaften von *SmallTalk* auf die Sprache *C* anzuwenden, auch das Konzept der dynamischen Bindung übernommen hat (siehe Unterkapitel 3.2.4.1). Kurz zusammengefasst erlaubt die dynamische Bindung im Zusammenhang mit *Properties*, dass die zu einer *Property* gehörenden Akzessor-Methoden nicht zur Compilezeit vorliegen müssen, sondern erst zur Laufzeit dynamisch eingebunden werden können. Ist eine (Akzessor-)Methode zur Laufzeit nicht auffindbar, wird das Programm terminiert, sofern keine Fehlerbehandlungsroutinen vorgesehen sind.

   Auch bei Definition von Properties via `@dynamic` kann der Programmierer nach Belieben eine oder beide Akzessor-Methoden per Hand im Quellcode implementieren. Diese Akzessor-Methoden sind dann zu Beginn der Laufzeit bereits eingebunden.

3. Die Getter- und/ oder Setter-Methode für Properties können auch vom Programmierer **manuell** angelegt werden, zum Beispiel dann, wenn er weiterführende Funktionalität in der Implementierung der jeweiligen Akzessor-Methode beschreiben will als es die Synthese mit `@synthesize` ermöglicht. Werden alle Akzessor-Methoden für eine *Property*, entsprechend ihren Attributen per Hand erstellt, kann auf die Definition dieser *Property* anhand der Schlüsselwörter `@synthesize` und `@dynamic` verzichtet werden. So muss für eine `readonly` *Property* lediglich eine *Getter*-Methode, für eine `readwrite` *Property* zusätzlich noch eine *Setter*-Methode angelegt werden. Die Namen der selbst implementierten Akzessor-Methoden müssen sich exakt an den Namenskonventionen für automatisch generierte *Property*-Akzessor-Methoden oder an den durch geeignete *Property*-Attribute festgelegten Bezeichnern orientieren (siehe Paragraphen „*Property*-Attribute" in diesem Unterkapitel). Fehlt eine Akzessor-Methode einer manuell implementierten *Property* zur Compilezeit, äußert der Compiler eine Warnung. Der Compiliervorgang läuft trotzdem durch, die fehlende(n) Methode(n) können noch zur Laufzeit dynamisch eingebunden werden.

   Anzumerken ist, dass die hier vorgestellte Methode der manuellen Implementierung von Akzessor-Methoden mit den beiden anderen Methoden kombiniert werden kann (siehe erste Methode, letzter Abschnitt). Eine *Setter*-Methode kann zum Beispiel von Hand implementiert werden und zugleich die *Getter*-Methode dieser *Property* via `@synthesize` durch den Compiler erstellt werden lassen.

Property-Attribute:

Bei der Deklaration von *Properties* können optionale Attribute angegeben werden. Sie ermöglichen die Akzessor-Methoden einer *Property* bezüglich ihrer Implementierung näher zu definieren und

gewähren Methodennutzern in übersichtlicher Form ein besseres Verständnis über die Funktionsweise der Methoden. *Property*-Attribute werden in einer durch Kommas getrennte Liste hinter dem Schlüsselwort `@property` angegeben (siehe Codelisting 3.28). Sie gelten für alle Bezeichner die am Ende der *Property*-Deklaration stehen.

Property-Attribute lassen sich hinsichtlich ihrer Bedeutung in folgende Gruppen unterteilen: (vgl. [App11b], S. 64ff)

- Attribute zur Modifikation der Namensgebung von Akzessor-Methoden:
  - Das Attribut **`getter=getterName`** legt den Namen der *Getter*-Methode fest. Wird die dazugehörige *Property* mit dem Befehl `@synthesize` vom Compiler generiert, entspricht der Methodenbezeichner diesem Schema. Bei `@dynamic` wird zur Laufzeit eine entsprechend benannte Methode erwartet und auch bei manueller Implementierung der *Getter*-Methode, muss das hier angegebene Namensschema befolgt werden.
  - Das Attribut **`setter=setterName`** legt den Namen der Setter-Methode fest. Es gelten, in angepasster Form für *Setter*-Methoden, die gleichen Regeln wie beim zuletzt beschriebenen Attribut.
- Attribute zur Festlegung der Schreibbarkeit von *Properties*:
  - Das Attribut **`readwrite`** gibt an, dass die *Property* sowohl gelesen als auch verändert werden kann. Wird die *Property* mit `@synthesize` definiert, werden entsprechende *Getter*- und *Setter*-Methoden vom Compiler erstellt. Bei Verwendung von `@dynamic` müssen entsprechende Akzessor-Methoden zur Laufzeit und bei manueller Erstellung zur Compilezeit vorliegen.

    Wird kein Attribut zur Festlegung der Schreibbarkeit einer *Property* angegeben, wird das Attribut `readwrite` implizit angenommen.
  - Das Attribut **`readonly`** gibt an, dass eine *Property* schreibgeschützt ist. Wird die Property mit `@synthesize` definiert, erstellt der Compiler nur eine *Getter*-Methode. Bei `@dynamic` wird eine *Getter*-Methode zur Laufzeit und bei der manuellen Implementierung zur Compilezeit erwartet.

    Beim Versuch einer schreibgeschützten *Property* per *Dot*-Syntax einen Wert zuzuweisen, was dem Aufruf der *Setter*-Methode für diese *Property* entspricht, meldet der Compiler einen Fehler. Anzumerken ist jedoch, dass der Schreibschutz umgangen werden kann indem innerhalb der Klasse, in der die *Property* definiert ist, per Pfeiloperator direkt auf die gekapselte Variable zugegriffen wird. Der Schreibschutz ist also nur außerhalb der Klasse wirksam, was durchaus gewollt sein kann.

    Anzumerken und zugleich abzuraten ist von der Möglichkeit trotz der Angabe von `readwrite` eine, den Namenskonventionen entsprechende, *Setter*-Methode für diese *Property* anzulegen oder zur Laufzeit einzubinden. Zwar ist die *Setter*-Methode dann für diese *Property* ganz normal nutzbar und der Compiler äußert keine Bedenken, doch schadet dieses Vorgehen ungemein der Lesbarkeit und Verständlichkeit des Quellcodes.
- Attribute welche Aussagen zur Atomarität machen:

  - Das einzige Schlüsselwort dieser Kategorie, **`nonatomic`** sagt aus, dass die Akzessor-Methoden einer so typisierten *Property* nicht thread-sicher implementiert sindist. *Race Conditions* beim Zugriff auf die gekapselte Variable können auftreten.
  - Es existiert in *Objective-C* kein explizites Schlüsselwort als Gegenstück zu `nonatomic`. Wird auf die Angabe von `nonatomic` als Attribut verzichtet, wird die *Property* implizit als atomar angesehen. Der Zugriff wird dann anhand von Semaphoren geregelt, wodurch parallele Zugriffe gefahrlos durchgeführt werden können.

- Attribute die sich gegenseitig ausschließen und welche im Zusammenhang mit der Speicherverwaltung für *Properties* stehen, die Objekte kapseln:
  - Das Schlüsselwort **`strong`** gibt an, dass es eine starke Kohärenz zwischen dem Objekt, in dem die *Property* deklariert wurde, und dem Objekt, welches durch die *Property* gekapselt wird, existiert. Dieses Attribut spielt eine wichtige Rolle bei der *Garbage Collection* für *Mac-OS-X*-Programme und dem *Automatic Reference Counting* (kurz ARC) für *iOS*-Programme. Wird *ARC* verwendet, wird bei der Zuweisung eines Objekts an eine, mit diesem Attribut versehene *Property*, ein spezieller Zähler inkrementiert, der *Retain Count* genannt wird. Erreicht dieser Zähler Null, wird das Objekt freigegeben (siehe 4.2.3).

    Das Attribut `strong` übernimmt bei der Programmierung für *iOS 5* mit aktiviertem *ARC* die Rolle des Schlüsselworts `retain` (siehe weiter unten), welches dann nicht mehr genutzt werden sollte. Zusätzlich kümmert sich das *ARC* bei Benutzung des Attributs `strong` um die automatische Zerstörung des Objekts wenn der *Retain Count* auf Null fällt.
  - Das Schlüsselwort **`weak`** sagt, im Gegensatz zum letzten Attribut, aus, dass es eine schwache Beziehung zwischen dem Objekt, welches die *Property* deklariert, und dem Objekt, welches durch die *Property* gekapselt wird, gibt. Auch dieses Attribut spielt bei der Speicherverwaltung eine zentrale Rolle. Bei der Zuweisung eines Objekts an eine *Property* mit `weak`-Attribut, wird der *Retain Count* für das zugewiesene Objekt nicht erhöht. Obwohl *ARC* durch automatische Verfolgung von Beziehungen zwischen Objekten eine Erleichterung bei der Programmierung bringt, ist es aber aufgrund von Beschränkungen nicht in der Lage zyklische Abhängigkeiten zwischen Objekten aufzudecken (siehe Unterkapitel 4.2.3). Zur manuellen Auflösung von Abhängigkeitszyklen muss der Programmierer selber sorgen, indem er an geeigneten Stellen die Schlüsselwörter `weak` und sein Gegenstück `strong` einsetzt (siehe Bsp. 3.32).

```
/* Declaration of a fictive class "ClassX" */
#import "ClassY.h"

@interface ClassX
@property (strong) ClassY *intanceOfClassY;
@end


-----


/* Declaration of another fictive class "ClassY" in another file */
#import "ClassX.h"

@interface ClassY
@property (weak) ClassX *instanceOfClassX;
@end
```

Tabelle 3.32.: Beispiel für einen korrekten Einsatz der *Property*-Attribute `strong` und `weak` um zyklische Abhängigkeiten zu vermeiden. Bei der Zuweisung eines Instanzobjekts vom Typ `ClassY` in einem `ClassX`-Instanzobjekt wird der *Retain Count* um eins erhöht um zu symbolisieren, dass ein anderes Objekt eine Beziehung zu diesem Objekt besitzt. Bei der umgekehrten Zuweisung eines `ClassX`-Objekts in einem `ClassY`-Instanzobjekt wird der *Retain Count* des zugewiesenen Objekts hingegen nicht erhöht.
Würde zwischen beiden Klassen eine beidseitige starke Abhängigkeitsbeziehung existieren, könnte kein Instanzobjekt der beiden Klassen je aus dem Speicher entfernt werden, weil keiner ihrer Zähler jemals Null erreichen könnte. Hierfür müsste erst das jeweils andere Objekt zerstört werden, was durch die gegenseitige Beziehung aber nicht möglich wäre.

Um Laufzeitfehler zu vermeiden, muss bei Nutzung des `weak`-Attributs aufgepasst werden. Fällt der *Retain Count* eines `ClassX`-Instanzobjekts aus dem Beispiel 3.32 auf Null, wird das Objekt unweigerlich zerstört. Versucht ein Instanzobjekt der Klasse `ClassY` auf das nunmehr zerstörte Objekt zuzugreifen, kommt es zu einem Laufzeitfehler.

Abschließend ist anzumerken, dass wenn weder `strong` noch `weak` angegeben wird, implizit `weak` für *Properties*, welche Objekte kapseln, angenommen wird.

– Das Attribut **`copy`** legt die Implementierung der Setter-Methode für den schreibenden Zugriff auf eine, durch eine *Property* gekapselte, Variable wie folgt fest. Zuerst wird eine Kopie des zuzuweisenden Wert erstellt. Dann wird diese Kopie der Variable zugewiesen.

  Dieses Attribut ist nur in Kombination mit dem *Cocoa-Framework* nutzbar und dort nur mit Objekten, die das Protokoll `NSCopying` implementieren (vgl. [App11b], S. 65). Bei der Erstellung der Kopie wird der *Retain Count* der neu erstellten Objektkopie initialisiert und um Eins inkrementiert.

– Die mit der Einführung vom *Automatic Reference Counting* (kurz *ARC*) in *iOS 5* (siehe Unterkapitel 4.2.3) überholten Schlüsselwörter **`retain`** und **`assign`** können parallel zu `strong` und `weak` verwendet werden. `Retain` erfüllt nahezu den selben Zweck wie `strong` und `assign` entspricht dem neuen Attribut `weak`. Bei der Nutzung von `strong`, für eine *Property*, in Kombination mit *ARC* wird das, durch diese *Property* gekapselte, Objekt automatisch aus dem Speicher entfernt, wenn sein *Retain Count* auf Null fällt. Bei der Nutzung des Attributs `retain`, muss die Entfernung des Objekts aus dem Speicher dagegen vom Programmierer manuell vorgenommen werden.

  Die Namen der Schlüsselwörter stehen in Verbindung mit der, seit der Einführung von *ARC* überflüssigen, `retain`-Methode, welche wenn verwendet den *Retain Count* erhöhte, um eine Beziehung zwischen dem erstellten Objekt und dem Erzeuger des Objekts zu repräsentieren. Seit *iOS 5* verbietet *Apple* bei aktiviertem *ARC* die Nutzung dieser Methode und führte zugleich die neuen, aussagekräftigeren Schlüsselwörter ein. Aus Kompatibilitätsgründen wurden die alten Bezeichner beibehalten.

Abschließend ist anzumerken, dass die, in der letzten Attribute-Kategorie vorgestellten Schlüsselwörter auch bei der Deklaration von Instanzvariablen verwendet werden können. Die Attribute werden, vorangegangen durch zwei Unterstriche, vor den Variablentyp geschrieben. Die Bedeutung der Schlüsselwörter bleibt unverändert. Beispiel: `__strong ClassY *instanceOfClassY;`

Die Dot-Notation:

Als logische Ergänzung zum Konzept des sicheren Zugriffs auf Instanzvariablen durch *Properties*, führte *Apple* mit *Objective-C 2.0* die ***Dot*-Syntax** ein. Statt eines weniger leserlichen und den Quellcode unnötig aufblähenden Aufrufs einer Akzessor-Methode über eine Nachricht in Klammernotation (siehe Unterkapitel 3.2.4.2), kann anhand des neu hinzugekommenen ***Dot*-Operators** die benötigte Akzessor-Methode einfach und schnell aufgerufen werden. Die *Dot*-Syntax folgt dabei demselben Schema wie beim Zugriff auf *C*-Strukturelemente (siehe Beispiel 3.33). (vgl. [App11b], S. 17f)

```
1 myInstance.value1 = 10;
2 myInstance.value2 = myInstance.value1 + 2;
3
4
5 -----
6
7
8 [myInstance setValue1:10];
9 [myInstance setValue2:[myInstance value1] + 2];
```

Tabelle 3.33.: Im obigen Block wird die *Dot*-Syntax verwendet. Im unteren wird die gleiche Funktionalität ohne *Dot*-Syntax dargestellt. Bei Zuweisungen unter Benutzung der *Dot*-Notation wird automatisch die *Setter*-Methode aufgerufen. Beim lesenden Zugriff wird die *Getter*-Methode benutzt.

Soll mit der *Dot*-Syntax innerhalb einer Klasse auf eine, in derselben Klasse definierte, *Property* zugegriffen werden, muss das Schlüsselwort `self` vor dem *Dot*-Operator angegeben werden. Erfolgt ein Zugriff auf eine Variable ohne Angabe von `self` und *Dot*-Operator, handelt es sich um einen direkten Zugriff auf die Instanzvariable via Pfeiloperator (siehe Unterkapitel 3.2.3.1), unter Umgehung der Akzessor-Methoden.

Abschließend ist anzumerken, dass für den Aufruf von Akzessor-Methoden über die *Dot*-Notation nicht zwingend eine *Property* deklariert werden muss. Der Aufruf selbst geschriebener Akzessor-Methoden für eine selbst angelegte Instanzvariable ist mit der *Dot*-Notation genau so möglich.

3.2.4. Methoden in Objective-C

3.2.4.1. Die Rolle von Funktionen, Methoden und Nachrichten

Während sich das Verhalten von Programmen in der Sprache *C* in **Funktionen** beschreiben lässt, welche sich auf Module aufteilen, eröffnen sich beim objektorientierten Paradigma durch Klassen und Objekte andere Möglichkeiten Programmprozeduren festzulegen. Das objektorientierte Pendant zur Funktion heißt in *Objective-C* **Methode**. Eine Methode lässt sich im Vergleich zu einer Funktion einem Objekt zuteilen und legt dessen Verhalten fest. Da *Objective-C* auf *C* basiert, lässt sich aber weiterhin exklusiv mit prozeduralen Funktionen oder mit einer Mischung aus Funktionen und Methoden arbeiten. Eine Methode kann also eine Funktion aufrufen. In der Sprachspezifikation ist der Aufruf einer Methode durch eine Funktion aber nicht ohne Weiteres vorgesehen. Dies ist verwunderlich, da der Compiler *Objective-C*-Methoden zur Compilezeit in *C*-Funktionen umwandelt. Um einer so erstellten *C*-Funktion den Zugriff auf die Instanzvariablen des Objekts, auf dem die ursprüngliche Methode aufgerufen wurde, zu ermöglichen, fügt der Compiler einen weiteren, versteckten Parameter an die *C*-Repräsentation der Methode an. Als Parameterwert wird ein Zeiger auf die *C*-Datenstruktur des Objekts, auf dem die Methode aufgerufen wurde, übergeben (siehe 3.2.3.1). Die so gewonnene Funktion verhält sich zur Laufzeit wie eine reine *C*-Funktion. Der einzige Unterschied besteht darin, dass in der Implementation der Funktion nicht mit Variablen des Moduls, in dem die Funktion definiert ist, gearbeitet wird, sondern mit den Variablen aus der Struktur des Objekts, auf dem die ursprüngliche *Objective-C*-Methode aufgerufen wurde. (vgl. [App11b], S. 12ff)

Während sich Methoden in *Objective-C* und Funktionen aus *C* in ihrer Funktionsweise nicht groß unterscheiden, gibt es einen größeren Unterschied bei der Syntax ihrer Deklaration. Dieser Unterschied ergibt sich nicht etwa aus den verschiedenen Paradigmen, sondern entsteht, da sich die Syntax von *Objective-C* Methoden stark an die aus *SmallTalk* anlehnt und daher auch zu der aus anderen objektorientierten Sprachen wie *C++* oder *Java* divergiert (siehe Unterkapitel 3.2.4.2).

C und *Objective-C* unterscheiden sich auch in der Art und Weise wie Programmverhalten aufgerufen wird und welche Syntax dazu verwendet wird (siehe Unterkapitel 3.2.4.2). Während ein Funktionsaufruf syntaktisch in *C* aus dem Funktionsnamen, gefolgt von Aufrufparametern in runden Klammern geprägt ist, wird eine Methode eines Objekts in *Objective-C* anhand einer **Nachricht** aufgerufen. Auch dieses Konzept ist aus *SmallTalk* entliehen und unterscheidet sich fundamental vom gleichen Konzept in objektorientierten Sprachen wie *C++* oder *Java*. Für einen Nachrichtenaufruf werden drei Elemente gebraucht. Es sind ein **Sender** und ein **Empfänger** sowie eine **Nachricht**

erforderlich. Der Sender stellt das Objekt dar, aus dem, aus einer beliebigen Methode heraus, die Funktionalität eines anderen Objekts in Anspruch genommen wird. Der Empfänger stellt dagegen das Objekt dar, dessen Verhalten aufgerufen wird. Die Nachricht stellt eine Repräsentation der aufzurufenden Methode samt Parametern dar. (vgl. [App11b], S. 12f)

Wie *Objective-C*-Methoden zur Laufzeit als erweiterte *C*-Funktionen abgebildet werden, werden die Stellen, an denen Methodenaufrufe im Quellcode stattfinden, durch Aufrufe einer spezielle *C*-Funktion ersetzt, die Funktion `objc_msgSend`. Diese Funktion erhält als Parameter das Empfängerobjekt, bzw. einen Zeiger auf dessen Datenstruktur und den Namen der Methode welche in der Nachricht erwähnt wird (siehe Codeauszug 3.34 und 3.35). Beim Namen einer Methode spricht *Objective-C* auch von Methodenselektor, wobei es sich um eine einfache Zeichenkette handelt. Zur Laufzeit wertet die `objc_msgSend`-Funktion den Methodenselektor sowie mögliche Parameter der Nachricht aus und versucht eine geeignete Methodenimplementierung im Empfängerobjekt aufzurufen (vgl. [App11b], S. 17). Hierzu folgt die Laufzeitumgebung der `isa`-Variable, die in der *C*-Datenstruktur jedes *Objective-C*-Objekts gespeichert wird, zur Klassenimplementierung und bedient sich dort der *Dispatch Table*. Die *Dispatch Table* einer Klasse besitzt für jede Methode einen Eintrag, welcher den Selektor einer Methode mit der Speicherstelle der Methodenimplementierung in der Klasse verbindet. Bei der Suche nach einer geeigneten Methodenimplementierung wird zuerst in der *Dispatch Table* des Empfängerobjekts gesucht. Wird die Laufzeitumgebung hier nicht fündig, folgt sie einem Zeiger auf die Oberklasse des Empfängerobjekts, welcher in der *C*-Datenstruktur eines jeden *Objective-C*-Objekts gespeichert wird. Nun sucht sie dort nach einer geeigneten Methodenimplementierung. Die Laufzeitumgebung traversiert dabei die Vererbungshierarchie so lange nach oben, bis das Wurzelobjekt erreicht wird. Wird auch hier keine entsprechende Methodenimplementierung gefunden, geben *Objective-C*-Laufzeitumgebungen einem Objekt die Möglichkeit über eine *Forward Invocation* eine andere Methode anstatt der nicht gefundenen anzugeben, welche dann aufgerufen wird (für Details hierzu siehe [App09b], S. 18). Die hier beschriebene Suche nach einer geeigneten Methode zur Laufzeit nennt sich *Dynamic Method Resolution*. (vgl. [App09b], S. 10ff und [App11b], S. 44ff)

```
1 /* Excempt - Objective-C Code */
2 [receiver message]
3
4
5 -----
6
7
8 /* Excempt - equivalent C-Code after Objective-C-program-compilation */
9 objc_msgSend(receiver, selector)
```

Tabelle 3.34.: Die an das Objekt gesandte Nachricht wird vom Compiler in einen Aufruf einer speziellen *C*-Funktion `objc_msgSend` übersetzt. Das Argument `receiver` stellt einen Zeiger auf das Empfängerobjekt dar und wird dazu verwendet, der aufgerufenen Methode die Datenstruktur des Empfängerobjekts zugänglich zu machen. Der Parameter `selector` stellt eine Zeichenkette und die eigentliche Nachricht dar, welche zur Laufzeit ausgewertet wird – d.h. den Namen der aufzurufenden Methode.

```
1 objc_msgSend(receiver, selector, arg1, arg2, ...)
```

Tabelle 3.35.: Zusätzlich in der Nachricht angegebene Parameter werden auch an die Funktion `objc_msgSend` übergeben. Sie werden bei der Bestimmung der aufzurufenden Methodenimplementierung zur Laufzeit hinzugezogen.

Das Vorgehen zur Bindung einer Methode zur Laufzeit nennt sich *Dynamic Method Binding* und unterscheidet sich grundlegend von der Vorgehensweise in Sprachen wie *C++* oder *Java*. In diesen Sprachen werden die Methodenaufrufe immer zur Compilezeit an konkrete Methodenimplementierungen gebunden. In *Objective-C* geschieht dies erst zur Laufzeit. Das aus *SmallTalk* stammende Konzept bietet den Vorteil, dass Methodenimplementierungen während der Laufzeit nachgeladen werden können und somit nicht im ursprünglichen Quellcode vorkommen müssen. Zudem lassen sich bereits existierender Klassen, auch solchen deren Quellcode nicht vorliegt, mit wenig Mühe um weitere Funktionen erweitern, indem auf Basis des *Dynamic Method Bindings* eine Kombination der Konzepte *Dynamic Loading* (Details u.a. in [App07]) und *Objective-C*-Kategorien angewendet wird (siehe Unterkapitel 3.2.5).

Nach Abschluss eines Methodenaufrufs, also intern nach Beendigung der *C*-Repräsentation der aufgerufenen Methodenimplementierung, wird der Rückgabewert an die Stelle zurückgegeben, an der der Methodenaufruf, anhand einer Nachrichtensendung, stattfand.

Wird eine Nachricht an ein statisch typisiertes Objekt gesendet – der Typ entspricht einem Klassenbezeichner (siehe Unterkapitel 3.2.3.3) – kann der Compiler zur Compilezeit überprüfen, ob dieses Empfängerobjekt eine entsprechende Methodenimplementierung besitzt. Ist dies nicht der Fall meldet der Compiler einen Fehler.

Wird eine Nachricht hingegen an ein dynamisch typisiertes Objekt gesendet – ein Objekt vom Typ `id`, `Class` oder `Protocol` – liegen dem Compiler zur Compilezeit nicht genügend Informationen über das Empfängerobjekt vor um eine konkrete Aussage darüber zu treffen, ob eine entsprechende Methodenimplementierung existiert (siehe Unterkapitel 3.2.3.3). Fehlt eine Methodenimplementierung zur Compilezeit, kann der Compiler keinen Fehler melden. Beim Aufruf einer vermeintlich existierenden Methode zur Laufzeit, greift die Laufzeitumgebung über die Funktion `objc_msgSend` auf die `isa`-Variable des dynamisch typisierten Objekts zu und bestimmt die konkrete Klasse hinter dem Empfängerobjekt. Anhand der *Dispatch Table* dieser Klasse wird ermittelt, ob eine entsprechende Methodenimplementierung gegeben ist. Falls nicht, wird ein Laufzeitfehler gemeldet und die Programmausführung abgebrochen. Falls ja, wird diese Methode aufgerufen. Anzumerken ist, dass einer Klasse zur Laufzeit über Kategorien (siehe Unterkapitel 3.2.5) oder *Dynamic Loading* weitere Methoden zugeführt werden können, welche zusätzlich in der *Dispatch Table* der Klasse eingetragen werden und somit zusätzlich zum Klassenumfang zur Compilezeit zur Verfügung stehen.

Um aus einer Methodenimplementierung eine andere Methode des gleichen Objekts aufzurufen, muss eine Nachricht an das Empfängerobjekt `self` gesendet werden. `Self` stellt eine Instanzvariable eines Objekts dar welche auf das Objekt selbst verweist[9]. Sie wird in der *C*-Datenstruktur

[9]Im Vergleich zum Schlüsselwort `this` aus *Java* erlaubt *Objective-C* eine Neuzuweisung an `self`. Die Herkunft eines

neben der Instanzvariable `isa` für jedes Objekt in *Objective-C* gespeichert. Beim Aufruf einer Methode auf `self`, wird bei der Suche nach einer geeigneten Methodenimplementierung in der Vererbungshierarchie auf der Ebene angefangen, auf der sich die aufrufende Methode befindet. Soll die aktuelle Ebene übersprungen werden und direkt in der Oberklasse gesucht werden, kommt das Schlüsselwort `super` zum Einsatz. `Super` stellt im Gegensatz zu `self` nur ein *Flag* dar und wird nicht als Instanzvariable gespeichert.

3.2.4.2. Deklaration, Definition sowie Syntax von Methoden und Nachrichten

Das Festlegen des Verhaltens einer Klasse in *Objective-C* erfolgt meistens in zwei Schritten:

1. Bei der Deklaration einer Methode wird die Signatur, die Sichtbarkeit und der Rückgabetyp angegeben. Methodendeklarationen können im Klasseninterface erfolgen, aber auch in Kategorien oder Protokollen (siehe Unterkapitel 3.2.5.1 bzw. 3.2.6). Da Klasseninterfaces/ Kategorien/ Protokolle in weitere Klassen eingebunden werden können, bilden die darin beschriebenen Methoden und Variablen die Schnittstelle der sie implementierenden Klassendefinition.
2. Die Implementierung der eigentlichen Funktionalität einer Methode erfolgt in einer Methodendefinition. Die Definition von Methoden erfolgt ausschließlich in der Klassenimplementierung. Methodenimplementierungen werden an Unterklassen vererbt, wo sie bezüglich ihres Verhaltens überschrieben werden können. Der Zugriff auf die ursprüngliche Verhaltensweise aus der Superklasse ist mit dem Schlüsselwort `super` möglich (siehe Unterkapitel 3.2.4.1).

Syntaktisch unterscheiden sich Methodendeklarationen und -definitionen in *Objective-C* nur durch den, bei Methodenimplementierungen auf die Signatur folgenden, Anweisungsblock in geschweiften Klammern, statt des bei Methodendeklarationen abschließenden Semikolons. Die Unterschiede zur Beschreibung von Prozeduren in *Objective-C* im Vergleich zu Sprachen aus der C-Familie fallen hingegen prägnanter aus. Die hierzu benötigte Syntax in *Objective-C* ist aus *SmallTalk* entliehen.

Bis auf die Diskrepanz am Ende einer Methodendeklaration und -definition, setzt sich die Beschreibung des Verhaltens eines Objekts in *Objective-C* aus folgenden Teilen zusammen:

1. Zuerst wird mit einem Pluszeichen '+' oder Minuszeichen '-' angegeben ob es sich bei der Methode um eine **Klassenmethode** respektive eine **Instanzmethode** handelt. Im Vergleich zu fehlenden Klassenvariablen, welche über statische Variablen emuliert werden müssen (siehe Unterkapitel 3.2.3.2), bietet *Objective-C* also ein Konzept für Klassenmethoden. Anstatt an ein spezifisches Instanzobjekt, sind Klassenmethoden an ein Klassenobjekt gebunden (Unterscheidung siehe Unterkapitel 3.2.2.4). Klassenmethoden lassen sich jederzeit und überall aufrufen, indem eine entsprechende Nachricht an das Klassenobjekt gesendet wird. Klassenmethoden haben keinen Zugriff auf die Instanzvariablen irgendeines Instanzobjekts derselben Klasse. Sie werden genutzt um Instanz-übergreifende Operationen durchzuführen.

---

solch „schizophrenen" Objekts lässt sich aber stets über den `isa`–*Pointer* bestimmen, welcher als „*is-a*"-Beziehung die exakte Klasse des Objekts angibt (siehe Unterkapitel 3.2.3.3).

2. Es folgt die optionale Angabe eines Rückgabetyps der Methode in runden Klammern. Der Rückgabetyp ist in *Objective-C* wie in vielen anderen Sprachen kein Bestandteil der Methodensignatur. Folglich überschreibt eine Methodenimplementierung einer Klasse, eine Methodenimplementierung in einer Oberklasse dieser Klasse auch dann, wenn diese sich nur bezüglich ihres Rückgabetyps aber nicht in Hinblick auf ihre Signatur unterscheiden. Als Typ für die Rückgabe ist jeder für Variablen in *Objective-C* gültiger Typ zulässig (siehe Unterkapitel 3.2.3.3). Wird kein Rückgabetyp angegeben wird der dynamische Datentyp `id` implizit angenommen.
3. Als letzte Komponente der Anweisung, vor dem Semikolon oder Implementierungsblock, folgt der Methodenname. Hier zeigt sich der größte syntaktische Unterschied zwischen Sprachen wie *C*, *C++* oder *Java* im Vergleich zu Objective-C oder *SmallTalk*. Der Name besteht je nach Falllagerung aus einem oder mehreren Teilen:

   a) Handelt es sich bei der Methode um eine Methode welche **keine Parameter** benötigt, besteht der Name der Methode nur aus **einem Teil**. Zum Beispiel lautet der vollständige Name der im Beispiel 3.36 deklarierten Methode einfach `aClassMethod`.

```
1 + (void) aClassMethod;
```

Tabelle 3.36.: Deklaration einer Methode ohne Parameter (Auszug aus dem Einführungsbeispiel 3.1.1).

   b) Erwartet die Methode hingegen **einen Parameter**, so setzt sich der Methodenname aus **zwei Teilen** zusammen. In *Objective-C* werden Methodenparameter, im Vergleich zu Funktionsparametern in *C* oder Methodenparametern in *C++*, nicht in runden Klammern hinter dem Methodennamen geschrieben. Stattdessen wechseln sich Parameter und Methodenname ab. Für jeden Parameter erlaubt *Objective-C* die Angabe eines Parameterbezeichners. Dieser Bezeichner erlaubt es dem Benutzer einer Methode ausführlichere Informationen, zur Rolle und Verwendung eines Parameters innerhalb einer Methode, anzugeben, als es die Sprachen *C* oder *C++* erlauben. Auch an den Stellen, an denen die Methode aufgerufen wird ist dank Parameterbezeichner klar welche Parameter erwartet werden und welche Rolle sie beim Methodenaufruf einnehmen. Parameterbezeichner sind Teil des Methodennamens.

   Ein Parameterbezeichner besteht mindestens aus einem Doppelpunkt. Davor kann eine beliebige, den Parameter beschreibende und für Menschen schlüssige, Zeichenkette angegeben werden. Hinter den Doppelpunkt werden, bei der Deklaration/ Definition einer Methode, der Parametertyp und der **Parametername** angegeben. Diese sind nicht Teil des Methodennamens. Obwohl ein Parameterbezeichner bis auf den Doppelpunkt keine weiteren alphanumerischen Zeichen enthalten muss, ist aus Gründen der Lesbarkeit davon abzuraten. Denn schließlich soll das Konzept der Parameterbezeichner eine zuverlässigere Auskunft über die Funktionsweise einer Methode ermöglichen.

   Der Parametertyp wird hinter dem Doppelpunkt des Methodenbezeichners in runden Klammern angegeben. Als Datentyp steht jeder Typ der auch für Variablen in *Objective-*

C gültig ist bereit (siehe Unterkapitel 3.2.3.1). Der Name des Parameters in der Methodendefinition dient als Zugriffspunkt auf den Parameterwert in der Methodenimplementierung. Es ist anzumerken, dass sich der Parametername für ein und denselben Parameter bei der Methodendeklaration und Methodendefinition voneinander unterscheiden darf. Da der Parametername kein Teil des Methodennamens ist, ergibt eine Divergenz in der Namensgebung der Parameter in der Methodendeklaration und -definition zu keinem Zeitpunkt während der Entwicklung und zur Laufzeit eines Programms einen Fehler.

An diesen Konventionen ausgerichtet, lautet der vollständige Name der Methode aus dem Beispiel 3.37, folglich `aMethodDeclaredInProtocolXWithOneParam:`. Der Doppelpunkt gehört zur Syntax der Sprache und darf nicht weggelassen werden.

```
1 -(void) aMethodDeclaredInProtocolXWithOneParam:(id)someObjectParam;
```

Tabelle 3.37.: Deklaration einer Methode mit einem Parameter (Auszug aus dem Einführungsbeispiel 3.1.1).

c) Erhält eine Methode **mehrere Parameter,** wechseln sich Parameterbezeichner und Parameter ab. Eine Aneinanderreihung aller Parameterbezeichner bildet den vollständigen Methodennamen. Für die Methode in Beispiel 3.38 ergibt sich der vollständige Name `anInstanceMethodWithAParam:andAFloatParam:`.

```
1 -(float) anInstanceMethodWithAParam:(ClassB *)classBInstance andAFloatParam:(float)
      someFloatParam;
```

Tabelle 3.38.: Deklaration einer Methode mit mehreren Parametern (Auszug aus dem Einführungsbeispiel 3.1.1).

Auch der Aufruf einer Objektmethode anhand einer Nachricht folgt dem vorgelegten, syntaktischen Schema der Methodendeklaration/ -definition. Die Syntax einer Nachricht besteht aus zwei Teilen, welche von einem Paar eckigen Klammern umgeben sind: (vgl. [App11b], S.12f)

1. Der öffnenden Klammer folgend wird das Empfängerobjekt, dessen Methode aufgerufen werden soll, angegeben.
2. Vor der schließenden Klammer steht der Methodenname, an dieser Stelle oft Methodenselektor genannt (siehe Unterkapitel 3.2.4.1). Dieser fällt je nach Fall anders aus:

   a) Erwartet die aufzurufende Methode **keine Parameter**, besteht der Methodenselektor nur aus **einem Namensteil**. Beispielsweise würde ein Methodenaufruf der Klassenmethode aus dem Beispiel 3.36 auf dem Empfängerobjekt `aClassAObject` – ein Klassenobjekt – die Schreibweise aus dem Codelisting 3.39 übernehmen. Der Methodenselektor ist in diesem Fall `aClassMethod`.

```
1 [aClassObject aClassMethod];
```

Tabelle 3.39.: Aufruf einer Methode welche keine Parameter besitzt.

b) Erwartet die aufzurufende Methode **einen Parameter**, so folgt der Parameterwert am Ende des Methodenselektors hinter einem Doppelpunkt. Im Vergleich zur Methodendeklaration/ -definition wird bei einer Nachricht auf die Angabe des Parametertyps verzichtet. Die Nachricht im Codelisting 3.40 ist ein Beispiel für den Aufruf der im Beispiel 3.37 deklarierten Methode, aus Beispiel auf einem Empfängerobjekt `someClassAInstance`. Der vollständige Methodenselektor lautet `aMethodDeclaredInProtocolXWithOneParam:`. Der Doppelpunkt gehört zur Sprachsyntax und darf nicht weggelassen werden.

```
1 [someClassAInstance aMethodDeclaredInProtocolXWithOneParam:someObject]
```

Tabelle 3.40.: Aufruf einer Methode welche einen Parameter besitzt.

c) Erwartet die aufzurufende Methode **mehrere Parameter**, so wechseln sich Parameterbezeichner und Parameterwerte in einer Nachricht ab. Die Verkettung aller Parameterbezeichner liefert in solchen Fällen den vollständigen Methodenselektor. Die Parameterwerte werden nicht in den Methodennamen mit einbezogen. Beim Aufruf der Methode aus dem Beispiel 3.38 auf einem Empfängerobjekt `someClassAInstance` lautet der vollständige Methodenselektor in der Nachricht aus Codelisting 3.41 `anInstanceMethodWithAParam:andAFloatParam:`.

```
1 [someClassAInstance anInstanceMethodWithAParam:someClassBInstance andAFloatParam:
      someFloat]
```

Tabelle 3.41.: Aufruf einer Methode welche mehrere Parameter besitzt.

Anzumerken ist, dass die Sendung von Nachrichten beliebig verschachtelbar ist. D.h. im Aufruf einer Methode können ggf. vorhandene Parameter durch weitere Methodenaufrufe repräsentiert werden. Zur Laufzeit ergibt sich der Parameterwert aus dem Aufruf der hinterlegten Methode und der Rückgabe der dahinterliegenden Methodenimplementierung.

Abschließend sei erwähnt, dass neben dem Abruf einer Objektfunktionalität in Nachrichtenform mit eckigen Klammern, in ***Objective-C 2.0*** durch Einführung der *Dot*-Notation, eine weitere Möglichkeit besteht Methoden aufzurufen. Auf den Namen eines Empfängerobjekts folgt hinter einem Punkt (dem *Dot*-Operator) der Name der Instanzvariable dieses Objekts, auf die implizit über eine geeignete Akzessor-Methode zugegriffen werden soll. Die *Dot*-Notation wird oft in Verbindung mit *Properties* verwendet (siehe Unterkapitel 3.2.3.5).

3.2.5. Kategorien und Klassenerweiterungen

3.2.5.1. Kategorien

Eines der Hauptanliegen bei der Entwicklung von *Objective-C* war die einfache Handhabung großer Programme. Bereits aus der prozeduralen Programmierung hatte sich das Konzept der Unterteilung einer Anwendung in möglichst kleine Bausteine als förderlich für die Lesbarkeit und Wartbarkeit erwiesen. Durch die Unterteilung des Programmes in Klassen und Objekte gingen objektorientierte Sprachen darüber hinaus. Die Programmlogik wird in Klassen durch Methoden geprägt. Dieses Objektverhalten lässt sich sogar in Unterklassen durch Hinzufügen neuer Methoden erweitern.

Durch die Erweiterung einer Klasse in einer Unterklasse ergibt sich zwischen Typ-kompatiblen Objekten der Ober- und Unterklasse aber semantische Unterschiede. Soll die erweiterte Funktionalität genutzt werden, muss die Unterklasse genutzt werden, welche das entsprechend erweiterte Verhalten implementiert. Eine direkte Erweiterung der eigentlichen Klasse kann aber aufgrund des Umstands nicht möglich sein, dass man auf deren Quellcode keinen Zugriff hat, zum Beispiel weil sie im Rahmen eines Frameworks nur in bereits kompilierter Form vorliegt. Um dieses Manko zu adressieren, entleiht sich *Objective-C* dem aus *SmallTalk* bekannten Konzept der Kategorien und erweitert es. (vgl. [App11b], S. 73 und [Seb11], S. 91)

Kategorien sind Erweiterungen bestehender Klassen um weitere Methoden. Dabei können auch solche Klassen durch Kategorien erweitert werden, deren Quellcode nicht vorliegt oder die bereits kompiliert sind. Kategorien werden oft eingesetzt um einer existenten Klasse neues Verhalten in Form zusammengehöriger Methoden beizubringen. Sie ähneln in der Idee den Protokollen – ähnlich den *Java Interfaces* – unterscheiden sich aber in der Implementierung (siehe Unterkapitel 3.2.6).

Durch den Einsatz von Kategorien kann die Implementierung von Klassen auf mehrere Implementierungsdateien entfallen. Die aus den verschiedenen Implementierungsdateien gebündelte Verhaltensweise kann im Code auf Objekten dieses Klassentyps genutzt werden. Hierfür müssen neben der eigentlichen Klassendeklaration für jede erweiterte Funktionalität die genutzt werden soll, die entsprechende Kategorie-Datei mit importiert werden.

Das Erweiterungskonzept von Klassen durch Kategorien, ohne die Erstellung von Unterklassen, wird durch die Laufzeitumgebung und die dynamische Bindung der Sprache ermöglicht (siehe Unterkapitel 3.2.4.1). Erst zur Laufzeit erstellt die Laufzeitumgebung für jede Klasse die *Dispatch Table* (siehe ebd.). Dabei werden Methoden aus Klassenerweiterungen den nativen Methoden der Hauptklasse in der Tabelle hinzugefügt. Zur Laufzeit sind Methoden aus Klassenerweiterungen nicht von nativen Methoden der erweiterten Klasse zu unterscheiden. Sie haben vollen Zugriff auf die Instanzvariablen der erweiterten Klasse, auch auf private Variablen, und können alle implementierten Methoden der Klasse aufrufen.

Anzumerken ist, dass es bezüglich der Anzahl Kategorien die eine Klasse erweitern können keine Begrenzung gibt. Es muss lediglich darauf geachtet werden, dass die Namen der, eine gemeinsame Klasse erweiternden, Kategorien unterschiedlich sind und dass sich die Signaturen der, in den Kategorien deklarierten, Methoden unterscheiden. Sonst kann zur Laufzeit nicht zuverlässig

ermittelt werden welche Implementierung verwendet werden soll, was zu einem nichtdeterministischen Verhalten des Programms führt. Die Sprachspezifikation erlaubt aber die einmalige Neudeklaration einer Methode aus der Hauptklasse in einer, die Klasse erweiternden, Kategorie. Zur Laufzeit wird dann die Methodenimplementierung in der Kategorie genutzt. Dies erlaubt zum Beispiel die Korrektur fehlerhafter Methodenimplementierungen auch nach dem Compilieren.

Das Anlegen von Kategorien ähnelt dem bereits zum Anlegen von Klassen vorgestellten Verfahren stark (siehe Unterkapitel 3.2.2.1 und vgl. [App11b], S. 73ff). Zur Beschreibung einer Kategorie werden zwei Teile benötigt, eine Kategoriedeklaration und eine Kategoriedefinition:

Kategoriedeklaration:

Eine Kategoriedeklaration erfolgt in einer *.h*-Datei (vgl. Unterkapitel 3.2.1.2 und 3.2.2.1). Die Konvention für die Namensgebung dieser Datei schreibt vor, zuerst den Namen der zu erweiternden Klasse anzugeben, gefolgt von einem Pluszeichen '+' und abschließend gefolgt vom Namen der Erweiterung (siehe Bsp. 3.42).

```
1 ClassA+SomeCategory.h
2 ClassA+SomeCategory.m
3
4 ClassA+SomeOtherCategory.h
5 ClassA+SomeOtherCategory.m
```

Tabelle 3.42.: Zwei Beispiele für die Anwendung der Namenskonvention für Kategorien. Für ein Klasse `ClassA` werden die Kategorien `SomeCategory` und `SomeOtherCategory` angelegt. Um die erweiterte Funktionalität der Klasse nutzen zu können, müssen die *Header*-Dateien beider Kategorien in den Quellcode der Datei eingebunden werden, in dem die erweiterte Klassenfunktionalität genutzt werden soll.

Die Anweisung `@interface ClassName (CategoryName)<optionalProtocolList>` markiert den Anfang einer Kategoriedeklaration (siehe Beispiel 3.43). Im Vergleich zu einer Klassendeklaration entfällt bei einer Kategoriedeklaration die Angabe der Superklasse. Lediglich der Name der zu erweiternden Klasse wird in der Kategoriedeklaration wiederholt. Gefolgt vom Namen der Kategorie, in runden Klammern, und einer optionalen Liste der, in der Kategorie implementierter, Protokolle.

Für die Erstellung einer Kategorie ist die Einbindung der zu erweiternden Klasse in der Kategoriedeklaration mit den Präprozessordirektiven `#include` oder `#import` unabdingbar (siehe 3.2.1.3). Natürlich können auch weitere Klassen nach den in Unterkapitel 3.2.2.2 beschriebenen Regeln importiert werden.

Zwischen der Anweisung `@interface` und dem, die Kategoriedeklaration abschließenden, Schlüsselwort `@end` werden, die ursprüngliche Klasse erweiternde, Methoden und *Properties* nach bekannter Weise deklariert (siehe 3.2.4.2 und 3.2.3.5). In Kategorien können hingegen **keine** (neuen) **Instanzvariablen** deklariert werden. Das gilt insbesondere auch für die von *Properties* gekapselten

Variablen, es werden nur die Akzessor-Methoden erstellt (Details hierzu im Paragraphen „Kategoriedefinition" in diesem Unterkapitel).

Kategoriedefinition:

Die Definition einer Kategorie erfolgt den Konventionen entsprechend in einer *.m*-Datei (vgl. Unterkapitel 3.2.1.2 und 3.2.2.1). Der Name der Datei ist bis auf die Dateiendung identisch zum Namen der Kategoriedeklaration.

Die Kategoriedefinition beginnt mit der Zeile `@interface ClassName (CategoryName)` (siehe Beispiel 3.43). Auf das Schlüsselwort `@interface` folgt der Name der zu erweiternden Klasse, gefolgt vom Namen der sie erweiternden und hier implementierten Kategorie in runden Klammern.

```
1  /* Category-declaration "ClassB+Foo.h"*/
2  @interface ClassB (Foo)
3  -(void) someExtendingMethod;
4  @end
5
6  -----
7
8  /* Category-definition "ClassB+Foo.m" */
9  #import "ClassB+Foo.h"
10
11 @implementation ClassB (Foo)
12 -(void) someExtendingMethod{
13   // Method implementation
14 }
15 @end
16
17 -----
18
19 /* Random class-implementation using methods from the new category */
20 #import "SomeClass.h"
21 #import "ClassB.h";
22 #import "ClassB+Foo.h"
23
24 @implementation SomeClass
25 - (void) aMethodWithAClassBParam:(ClassB *)classBInstance{
26   [classBInstance someExtendingMethod]; // this is possible!
27 }
28 @end
```

Tabelle 3.43.: Beispiel für die Deklaration (Zeile 1-4) und Definition (Zeile 6-12) einer, die Klasse `ClassB` erweiternden, Kategorie `Foo` und anschließende Verwendung der erweiterten Funktionalität dieser Kategorie in einer Klasse `someClass` (Zeilen 14-24). Obwohl die Klasse `ClassB` die Methode `someExtendingMethod` nicht selbst implementiert, lässt sich die Methode zur Laufzeit dank der Funktionserweiterung durch die Kategorie auf einem Objekt vom Typ `ClassB` nutzen (Zeile 22).

Wie eine Klassendefinition ihre zugehörige Klassendeklaration importieren muss, muss eine Kategoriedefinition auch ihre zugehörige Kategoriedeklaration importieren (siehe Unterkapitel3.2.1.3). Um weitere Methoden und ggf. Instanzvariablen anderer Klassen und Kategorien zu nutzen, müssen diese ebenfalls in die Kategorie eingebunden werden (siehe Unterkapitel 3.2.2.2).

Zwischen den Anweisungen `@interface` und dem, die Kategoriedefinition abschließenden, Schlüsselwort `@end` werden die Methoden der Kategorie, nach denselben Regeln wie in Klassendefinitionen, implementiert (siehe Unterkapitel 3.2.2.1 und 3.2.4.2). Die Methoden einer Kategoriedefinition haben vollen Zugriff auf alle Instanzvariablen, *Properties* sowie Methoden der erweiterten Klasse und aller weiteren, dieselbe Klasse erweiternden, Kategorien. Zudem kann in den Methodenimplementierungen auf, als geeignet sichtbar gekennzeichnete, Instanzvariablen, *Properties* und Methoden importierter, fremder Klassen und fremder Kategorien zugegriffen werden.

Anzumerken ist, dass in Kategorien keine Instanzvariablen angelegt werden können. Dies gilt auch für die gekapselten Variablen in der Kategorie neu angelegter *Properties.* Zusätzlich zu dieser Beschränkung gilt für *Properties* in Kategorien, dass die Properties nicht anhand des Schlüsselworts `@synthesize` definiert werden können. Folglich kann der Compiler keine Akzessor-Methoden für Properties in Kategorien generieren. Stattdessen müssen die *Getter*- und *Setter*-Methoden manuell angelegt oder mit `@dynamic` angegeben und zur Laufzeit eingebunden werden (zur Bedeutung von `@synthesize`, `@dynamic` und der Anlegung von Akzessor-Methoden für *Properties* per Hand, siehe 3.2.3.5).

3.2.5.2. Klassenerweiterungen

Klassenerweiterungen sind eine spezielle Unterart von Kategorien für die einige Ausnahmen zu den allgemein gültigen Regeln für Kategorien gelten. Kategorien werden auf Englisch *Class Extensions* genannt und sind auch unter den Begriffen *informale Kategorie* oder *anonyme Kategorie* bekannt.

Im Vergleich zu normalen Kategorien wird mit einer Klassenerweiterung nicht die nach außen hin benutzbare Schnittstelle einer Klasse erweitert, sondern die konkrete Implementierung, also die private Funktionalität einer Klasse. Den Konventionen entsprechend sollte nur eine Klassenerweiterung pro Klasse benutzt werden, theoretisch sind aber eine beliebige Anzahl Klassenerweiterungen pro Klasse möglich.

Klassenerweiterungen stehen nicht in einer Klassendeklaration, sondern üblicherweise direkt in der *.m*-Datei der Klassenimplementierung (siehe Unterkapitel 3.2.1.2). Wird eine Klassenerweiterung in der Klassenimplementierung angegeben, so steht sie außerhalb und normalerweise vor dem, mit `@implementation` beginnenden und mit `@end` endenden, Implementierungsblock der Klasse (siehe Beispiel 3.44).

Optional kann die Klassenerweiterung auch in einer weiteren *.h*-Datei erfolgen. Diese Datei muss dann in der erweiterten Klasse importiert werden. Den Konventionen entsprechend entspricht der Name einer solchen Klassenerweiterungsdatei dem Namen der erweiterten Klasse, gefolgt von einem Unterstrich '_' und einem Klassenerweiterungsnamen. Der Name der Klassenerweiterung spielt nur im Dateinamen eine Rolle.

Eine Klassenerweiterung beginnt mit der Anweisung `@interface ClassName ()<ListOfProtocols >` und endet mit dem Schlüsselwort `@end` (siehe Beispiel 3.44). Der Name einer Klassenerweiterung

besteht immer aus einem leeren String, d.h. hinter dem Namen der zu erweiternden Klasse steht ein Inhaltlich leeres rundes Klammernpaar.

```
1  /* Implementation file "ClassA.m" */
2  #import "ClassA.h"
3  #import "ClassB.h"
4
5  // Class Extension
6  @interface ClassA ()
7  {
8    @private
9      float someFloat;
10 }
11 @property BOOL someBool;
12 @property(strong, readwrite) NSString *anotherString;
13 @end
14
15 @implementation ClassA
16 // ClassA method definitions
17 ...
18 @end
```

Tabelle 3.44.: Beispiel für eine Klassenerweiterung, wie sie im Einführungsbeispiel 3.1.1 vorkommt. Innerhalb der Klassendefinition von `ClassA` können alle Methodenimplementierungen auf die Akzessor-Methoden der durch *Properties* gekapselten Variablen `someBool` und `anotherString` zugreifen. Zudem ist der direkte Zugriff auf diese Variablen und die Instanzvariable `someFloat` via Pfeiloperator (siehe Unterkapitel 3.2.3.1) innerhalb der Klassenimplementierung möglich. Von außen ist der direkte Zugriff auf die Variable und der indirekte Zugriff über die Akzessor-Methoden hingegen nicht möglich – die Akzessor-Methoden im Rahmen einer Klassenerweiterung sind nur innerhalb dieser Klasse sichtbar.

Innerhalb einer Klassenerweiterung lassen sich weitere Methoden, *Properties* und sogar, im Vergleich zu üblichen Kategorien, Instanzvariablen deklarieren. Wie auch bei Klassendeklarationen /-definitionen möglich, kann eine Klassenerweiterung Instanzvariablen in einem Block, umgeben von geschweiften Klammern, enthalten (siehe Unterkapitel 3.2.2.1 und 3.2.3.1). Im Unterschied zu den Instanzvariablen von Properties in regulären Kategorien, werden die Instanzvariablen von Properties die in einer Klassenerweiterung deklariert werden, bei der Definition angelegt (siehe Unterkapitel 3.2.5.1).

Alle durch einer Klassenerweiterung und der dort angegeben Protokolle, deklarierten Methoden müssen in der Klassenimplementierung implementiert werden.

Am häufigsten werden Klassenerweiterungen dazu verwendet, *Properties*, welche in der Klassendeklaration in Folge des *Information Hidings* als `readonly` deklariert wurden, für den schreibenden Zugriff anhand des *Dot*-Operators innerhalb der Klassenimplementierung mit `readwrite` umzudeklarieren (siehe Unterkapitel 3.2.3.5 und vgl. [App11b], S. 74).

3.2.6. Protokolle

Kategorien dienen dazu, das Verhalten einer Klasse durch das Hinzufügen von Methodenimplementierungen zu erweitern. Dabei handelt es sich um eine 1:n-Beziehung zwischen einer zu erweiternden Klasse und einer beliebigen Anzahl Kategorien, welche die Funktion der Klasse erweitern (siehe Unterkapitel 3.2.5).

Protokolle, wie auch Kategorien, bündeln Methoden und sind ein Indikator für die Funktionalität der mit ihnen in Verbindung gebrachten Klasse. Im Vergleich zu Kategorien, gilt aber die Relation n:m. Protokolle repräsentieren eine gewisse Funktionalität, welche von beliebigen Klassen zur gleichen Zeit implementiert werden kann. Während Kategorien die Implementierung einer Klasse erweitern, erweitern Protokolle die Schnittstelle einer Klasse**.** Ähnlich einem *Interface* in *Java*, deklariert ein Protokoll in *Objective-C* eine Menge zusammengehöriger Methoden und erweitert eine, das Protokoll implementierende, Klasse um die im Protokoll deklarierte Verhaltensweise.

Protokolle ermöglichen eine über die Klassenhierarchie hinausgehende Gruppierung von Objekten. Während *Objective-C* pro Klasse nur eine Superklasse erlaubt, kann über eine beliebige Anzahl Protokolle angegeben werden, dass eine Klasse, über die Funktionalität der Klasse und Superklasse hinaus, auch das in den Protokollen deklarierte Verhalten aufweist. Objekte welche dieselben Protokolle implementieren und damit zum Ausdruck bringen, dass sich ihre Verhalten, zumindest in gewissen Punkten Abseits der Klassenhierarchie, überschneidet, lassen sich somit logisch in Verbindung bringen. (vgl. [App11b], S. 50)

Es gibt drei prominente Situationen in der sich der Gebrauch von Protokollen lohnen kann: (vgl. [App11b], S. 50)

1. Zur Deklaration von Methoden, welche andere Programmierer implementieren sollen, ggf. in noch nicht existenten Klassenimplementierungen. So kann zum Beispiel eine Klasse erstellt werden, welche sich auf noch nicht vorhandene aber bereits in einem Protokoll deklarierte Funktionalität einer anderen Klasse beruft. Das Protokoll wird in die aktuelle Klasse eingebunden, was den Compiler zu keiner Fehlermeldung veranlasst. Erst zur Laufzeit und bei Abruf der im Protokoll deklarierten Funktionalität muss diese auch wirklich erbracht werden. Falls dies nicht der Fall ist, kommt es zu einem Laufzeitfehler. Hierbei spielt das Prinzip des *Dynamic Loading* und der *Dynamic Method Resolution* eine zentrale Rolle (siehe Unterkapitel 3.2.4.1).
2. Zur Deklaration einer Objektschnittstelle, bei der Details zur Klassenimplementierung für Außenstehende im Verborgenen bleiben sollen. In diesem Kontext wird von *anonymen Objekten* gesprochen. Anzumerken ist allerdings, dass auch wenn ein Programmierer eines solchen anonymen Objekts die dahintersteckende Klasse nicht preisgibt, also über ein Protokoll die Implementierung verbirgt, die Klasse und ihre Details trotzdem zur Laufzeit über Introspektion (siehe Unterkapitel 3.2.3.3) ermittelt werden können.
3. Um Gemeinsamkeiten von Klassen zu vereinigen, welche hierarchisch nicht miteinander verwandt sind. *Abstrakte Klassen* reichen nicht immer um gemeinsame Funktionalitäten zu binden, zum Beispiel dann, wenn die Objekte, welche ein ähnliches Verhalten aufweisen, in unterschiedlichen Zweigen der Vererbungshierarchie sind. Dies ist vor allem dem Umstand zuzuschreiben, dass *Objective-C* keine Mehrfachvererbung erlaubt, aber mit dem Konzept der Protokolle teilweise konterkariert werden kann.

Objective-C unterscheidet zwei Arten von Protokollen: (vgl. [App11b], S. 54ff)

1. **Formale Protokolle** ermöglichen die Zusammenfassung von Methodendeklarationen in einer eigenständigen Datei, einer Protokolldeklaration. Eine solche Deklaration lässt sich von einer beliebigen Klasse über `#import` oder `#include` einbinden (siehe Beispiel 3.45). Die Implementierung der in formalen Protokollen deklarierten Methoden, muss die Klassendefinition erfüllen. In Zusammenhang mit Klassen, welche die Methoden eines formales Protokolls implementieren, wird davon gesprochen, dass die Klasse das Protokoll *unterstützt* bzw. *adoptiert* oder *implementiert* (engl. *adopt*).

```
/* A generic formal-protocol-declaration */
@protocol SomeProtocol <ListOfOtherProtocols>
// method- & property-declarations
@end


-----


/* Class using that protocol */
#import "SomeProtocol.h"

@interface SomeClassUsingSomeProtocol <SomeProtocol>
// regular class-definition
...
// implementation of methods & properties from the SomeProtocol-protocol
...
@end
```

Tabelle 3.45.: Deklaration und Implementierung eines formalen Protokolls.

Formale Protokolle werden von der Sprachspezifikation her direkt unterstützt, so dass der Compiler u.a. Typ-Überprüfungen für Objektvariablen vom Typ eines Protokollobjekts durchführen kann. Dabei eignet sich ein Protokollname aber nicht direkt als Typname für Objekte! Stattdessen wird der Typ `id` verwendet und der Protokollname dahinter in spitzen Klammern angegeben (siehe Beispiel 3.46). Diese Notation erinnert stark an *generische Typen* wie sie u.a. in *Java* (ab Version *6*) vorkommen.

```
id <SomeProtocol> someObjectVariable;
```

Tabelle 3.46.: Deklaration einer Instanzvariable vom Typ eines Protokollobjekts. Die Klasse, welche die Variable deklariert, muss dieses Protokoll importieren und ggf. implementieren. Somit lassen sich auch heterogene Listen von nicht-hierarchisch verwandten Objekten erstellen.

Formale Protokolle werden darüber hinaus von der Laufzeitumgebung unterstützt. So kann

zur Laufzeit via Introspektion abgefragt werden, ob ein Objekt ein bestimmtes Protokoll implementiert.

Formale Protokolle werden mit der Direktive **`@protocol`** in einer eigenen *Header*-Datei beliebigen Namens deklariert. Die Deklaration endet auf `@end`. Auf das Schlüsselwort `@protocol` folgt der Name des Protokolls. Abschließend folgt eine optionale, durch Kommata getrennte, Liste von Protokollen, welche in dieses Protokoll eingewoben werden. Somit kann das Protokoll um die Funktionalität weiterer Protokolle erweitert werden. Nutzt eine Klasse das hier definierte formale Protokoll, muss sie alle Methoden dieses Protokolls sowie die Methoden aller direkt und indirekt eingewobenen Protokolle implementieren. Neben einer Liste von Methodendeklarationen können Programme, welche für *Mac OS X* ab Version *10.6* oder *iOS* entwickelt werden, auch *Properties* deklarieren.

Vor der Einführung der Sprachversion *2.0* mussten in *Objective-C* **alle** Methoden eines Protokolls, in einer dieses Protokoll adoptierenden Klasse, implementiert werden. Wurde dem nicht nachgegangen, gab der Compiler einen Fehler aus. Mit *Objective-C 2.0* wurden die beiden Schlüsselwörter `@required` und `@optional` eingeführt (siehe Beispiel 3.47). Alle Methodendeklarationen entsprechen der angegebenen Regel, welche zwischen einem dieser beiden Schlüsselwörter und dem Ende des Protokolls oder dem nächsten Aufkommen dieser Schlüsselwörter stehen. Die Schlüsselwörter unterscheiden sich wie folgt:

- Mit **`@optional`** lassen sich Methodendeklarationen als optional markieren, d.h. eine, das Protokoll einbindende, Klasse muss die mit `@optional` markierten Methoden nicht zwingend implementieren.
- Mit **`@required`** werden hingegen diejenigen Methodendeklarationen markiert, die eine, das Protokoll implementierende, Klasse unbedingt definieren muss. Wird keines der beiden Schlüsselwörter angegeben, wird für alle Methodendeklarationen im Protokoll von `@required` ausgegangen.

```
1 @protocol ProtocolX
2  @optional
3   - someOptionalProtocolMethod;
4  @required
5   -(void) aMethodDeclaredInProtocolXWithOneParam:(id)someObjectParam;
6 @end
```

Tabelle 3.47.: Beispiel für ein formales Protokoll, welches eine optionale und eine nicht-optionale Methodendeklaration angibt. Die das Protokoll adoptierende Klasse **muss** die nicht-optionale Methode implementieren und **kann** die andere definieren.

Abschließend bleibt anzumerken, dass in formalen Protokollen **keine Methodenimplementierungen**, sondern **nur Methodendeklarationen** angegeben werden können. Sie versichern dem Aufrufer von Methoden einer Klasse nur, dass diese Klasse diesem Protokoll entspricht und demnach die aufzurufende Funktionalität zur Laufzeit zur Verfügung stehen müsste. Der Programmierer muss dafür Sorge tragen, dass geeignete Methodenimplementierungen spätestens zur Laufzeit bereitstehen.

2. **Informale Protokolle** stellen eigentlich eine spezielle Unterart von Kategorien dar (siehe Unterkapitel 3.2.5). Im Vergleich zu Kategorien, welche nur dazu dienen eine Klassenimplementierung um weitere Funktionalität zu erweitern, gruppieren informale Protokolle zusammengehöriges Klassenverhalten in Form einer Liste von Methodendeklarationen zusammen und bieten diese mehr als einer Klasse zur Implementierung an. Zudem unterscheiden sie sich von Kategorien dahingegen, dass neben einer Deklaration des Protokolls in einer *Header*-Datei keine Definition des Protokolls stattfindet. Stattdessen werden informale Protokolle von Klassen importiert, die die darin deklarierten Methoden neu deklarieren und schließlich in der Klassendefinition implementieren. (vgl. [App11b], S. 55f)

   Durch die Abstammung informaler Protokolle von dem Konzept der Kategorien, ähneln sich die Deklaration eines informalen *Protokolls* und die einer Kategorie. Ein informales Protokoll wird in einer beliebig benannten *Header*-Datei nach dem, in Codelisting 3.48 dargestellten, Schema erstellt (siehe Beispiel 3.49). Zwischen den Schlüsselwörtern `@interface` und dem durch `@end` markierten Ende des informalen Protokolls, dürfen Methoden- und *Property*-Deklarationen stehen, aber **keine Instanzvariablen deklariert werden**. Zudem werden wie bei Kategorien – Klassenerweiterungen ausgenommen – die durch *Properties* gekapselte Instanzvariable bei der *Property*-Definition nicht erstellt (siehe Unterkapitel 3.2.5). (vgl. [App11b], S. 55f)

```
1 @interface ClassNameToExtend (CategoryName) <optionalListOfFormalProtocols>
```

Tabelle 3.48.: Schema der Deklaration eines informalen Protokolls.

```
1 @interface SomeClass (InformalProtocolName)
2  // some method- & property-declarations
3 @end
```

Tabelle 3.49.: Optisch unterscheidet sich die Deklaration eines informalen Protokolls nicht von der einer Kategorie. Im Vergleich zu einer Kategorie gibt es aber keine Definition des informalen Protokolls!

Abschließend ist anzumerken, dass informale Protokolle, im Vergleich zu formalen Protokollen, durch ihre Abstammung vom Kategorie-Konzept, **nicht als *generischen Typ*** für Objektvariablen genutzt werden können. Es ist also keine Typ-Überprüfung durch den Compiler möglich. Zudem lässt sich zur Laufzeit nicht überprüfen, ob eine Klasse ein informales Protokoll implementiert. (vgl. [App11b], S. 55f)

Teil III.

Einführung in die Programmierung mit dem iOS-SDK

4. Entwicklung für iOS

4.1. Einleitung

Nachdem im letzten Kapitel die Sprache *Objective-C* eingeführt, und auf die Bedeutung des Compilers und der Laufzeitumgebung an diversen Stellen hingewiesen wurde, wird in diesem Kapitel auf den letzten Baustein zur Programmierung von Anwendungen für *iOS* eingegangen, das *Framework*.

Zuerst werden in Unterkapitel 4.1.1 die *Frameworks Cocoa* und *Cocoa Touch* sowie die Entwicklungsumgebung *Xcode* kurz eingeführt. Anschließend werden zum besseren Verständnis von *iOS*-Anwendungen in Unterkapitel 4.2 wichtige Grundkenntnisse bezüglich des Aufbaus, der Objektinstantiierung und des Speichermanagements vermittelt. Abschließend erfolgt in Unterkapitel 4.3 die schrittweise Einführung in die Grundfunktionalität des *iOS-SDK* anhand von Beispielen. Ratschläge zur Benutzung und Erleichterung bei der Verwendung der Entwicklungsumgebung *Xcode* werden an geeigneten Stellen im Fließtext erwähnt.

4.1.1. Inhalt und Aufbau des iOS-SDK

Das *iOS-SDK* nimmt eine zentrale Rolle bei der Entwicklung von Anwendungen für *Apple*-Mobilgeräte ein. Seit der ursprünglichen Einführung als *iPhone SDK* im Jahr 2008 (siehe Unterkapitel 2) wurde das *iOS-SDK* kontinuierlich durch *Apple* erweitert, um den Fortschritten im Bereich der mobilen Anwendungsentwicklung und den Möglichkeiten der Hardware gerecht zu werden. Die aktuelle mit *iOS 5.1.1* herausgegebene Version des ***iOS-SDK*** enthält u.a. folgende Komponenten: (vgl. [Seb11], S. 189f)

- die ***Cocoa Touch*** Anwendungsumgebung, mit einem auf *Objective-C* basierenden Framework und einer Laufzeitumgebung für *iOS*-Geräte,
- die Entwicklungsumgebung ***Xcode***, welche speziell an die Bedürfnisse der Anwendungsentwicklung mit *Objective-C* auf *Cocoa*-Basis zugeschnitten ist, und
- den ***iOS Simulator***, welcher während der Anwendungserstellung zum schnellen Testen der Programmfunktionalität auf dem Entwicklungsrechner dient[1].

Cocoa geht aus dem Betriebssystem *NeXTSTEP* hervor, das *Apple* 1996 übernahm. Die *Cocoa*-Umgebung ersetzt die, bis *Mac OS 9* geerbte und rein *C*-basierte, *Carbon*-Umgebung und stellt seit *Mac*

[1] Beim *iOS Simulator* handelt es sich an der Bezeichnung orientiert um einen **Simulator**. Der Anwendungscode wird beim Aufruf des Simulators für die *x86*-Plattform des Entwicklungsrechners kompiliert und auf dessen *x86-Hardware* ausgeführt. *iOS*-Geräte basieren hingegen auf *ARM*-Prozessoren. Bei der Kompilierung und Ausführung der Anwendung auf dem Mobilgerät wird entsprechender Code erzeugt.

OS X die grafische Basis des Betriebssystems dar. Aus der *Cocoa*-Umgebung ging die *Cocoa-Touch*-Umgebung für *Apple*-Mobilgeräte hervor, mit der sie sich viele Gemeinsamkeiten teilt. Unterschiede zwischen *Cocoa* und *Cocoa Touch* ergeben sich aber bezüglich der grundverschiedenen Zielplattformen, welche unterschiedliche konzeptionellen Rahmenbedingungen gerecht werden müssen. Bei der Entwicklung mit *Cocoa Touch* muss u.a. dafür Rechnung getragen werden, dass: (vgl. [Dav11], S. 5ff)

- nur eine Anwendung zur selben Zeit im Vordergrund aktiv sein kann,
- nur ein Fenster unter *iOS* sichtbar sein kann, während bei Desktop-Betriebssystemen wie *Mac OS X* mehrere Programmfenster gleichzeitig sichtbar sind,
- *iOS*-Geräte ihren Anwendungen nur begrenzte Zugriffsrechte gewähren; alle Apps genannten Mobilanwendungen laufen in einer *Sandbox* und haben nur auf diejenigen Daten Zugriff, die sich in ihrem dedizierten Speicher befinden,
- Anwendungen stets flüssig laufen müssen, also nur begrenzte Reaktionszeiten gewährt bekommen, bevor Überwachungsmechanismen die Anwendung terminieren,
- *iOS*-Geräte bis vor kurzem nur geringe Bildschirmauflösungen besaßen[2] und ihre Schirme ein völlig anderes Seitenverhältnis als Computermonitoren aufweisen, und
- die auf *ARM-SoC* basierenden *iOS*-Geräte im Vergleich zu *x86-Hardware* geringere Systemressourcen aufweisen. So ist es u.a. diesem Umstand zuzuschreiben, dass *iOS*-Geräte bisher keinen *Garbage Collector* aufweisen (siehe Unterkapitel 4.2.3).

Neben diesen Plattform-bedingten Unterschieden, sind sich *Cocoa* und *Cocoa Touch* bezüglich ihrer Integration in die Systemumgebung aber sehr ähnlich. Beide setzen hierarchisch auf dem Kernel, eines von *Apple* entwickelten *FreeBSD-Unix*-Derivats, auf und ermöglichen Anwendungsentwicklern durch hochintegrierte Abstraktionsschichten den einfachen Zugriff auf die Funktionalität der Zielgeräte. Die Schichten ergeben den folgenden Aufbau (siehe Bild 4.1): (vgl. [App10a], S. 13ff)

1. Auf der untersten Ebene steht das Betriebssystem, bei *Mac OS X* und den diversen *iOS*-Geräten repräsentiert durch das *Unix*-Derivat ***Darwin***[3], welches u.a. Treiber, den Zugriff auf das Dateisystem und die Netzwerkinfrastruktur bereitstellt.
2. Darüber setzen zwei in *ANSI-C* gehaltene Schichten auf, welche als Abstraktionsebene für den Zugriff durch Anwendungen auf *Apple*-eigene Techniken und den Kernel dienen, sowie unter *Mac OS X* und iOS gleich benannt sind:
   a) die ***Core Services*** stellen eine Hülle für Datentypen und Algorithmen dar, während
   b) die ***Application Services*** eine Abkapselung für multimediale Technologien bilden.

[2] Die ersten drei *iPhone-* und *iPod-Touch*-Versionen boten eine maximale Auflösung von 320x480 Pixel. Das *iPhone* der vierten und fünften Generation sowie der *iPod Touch* der vierten Generation bieten dank ihrer *Retina Displays* hingegen schon eine Auflösung von 640x960 Pixel. Die erste und zweite Generation des *iPad* boten eine Auflösung von 1024x768 Pixel. Das *iPad* der neusten Generation (inoffiziell *iPad 3* genannt) bietet dank seines *Retina Displays* eine Auflösung von 2048x1536 Pixel bei einer Schirmdiagonale von 9,7 Zoll. Anzumerken ist, dass der Formfaktor des Bildschirms von Geräten mit, und solchen ohne *Retina Display* identisch ist, trotz der unterschiedlichen Auflösungen. Bei Geräten mit höherer Auflösung werden daher nicht mehr Elemente angezeigt als bei anderen mit geringerer Auflösung.

[3] Bei *iOS*-Geräten wird statt von Darwin auch von *Core OS* gesprochen.

3. Auf der obersten Ebene steht das **Anwendungs-*Framework***. Unter *Mac OS X* kann das u.a. das auf *Objective-C* basierende Framework *Cocoa*, die prozedurale *Carbon*-Umgebung oder das *Java-SDK* sein. Unter *iOS* steht allein das auf *Objective-C* basierende *Cocoa-Touch*-Framework zur Verfügung. Durch den *C*-Unterbau von *Objective-C* (siehe Unterkapitel 3.2.1) können auf Basis des *Cocoa-Frameworks* entwickelte Anwendungen direkt auf alle Funktionen der darunterliegenden Schichten zugreifen.

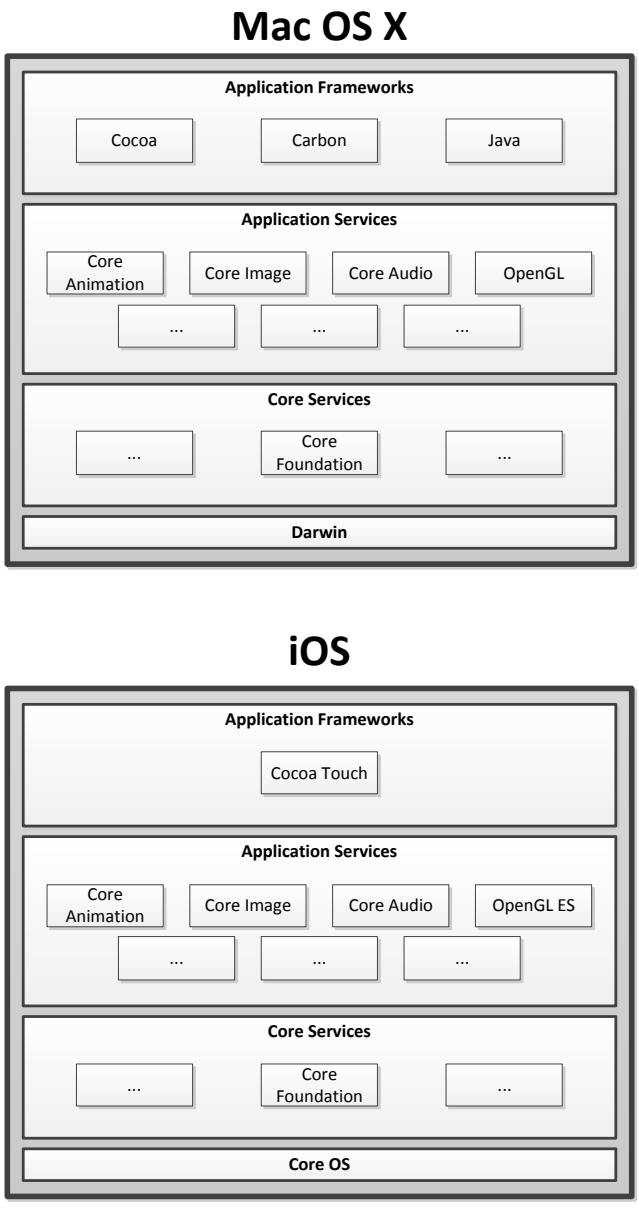

Abbildung 4.1.: Integration von *Cocoa* in *Mac OS X* und *Cocoa Touch* in *iOS* (vgl. Bild in [App10a], S. 13 und S. 15 sowie [Seb11], S. 189 und S. 209).

Zusätzlich zu den Gemeinsamkeiten bei der Integration in die Systemumgebung ähneln sich *Cocoa* und *Cocoa Touch* auch bezüglich ihres Aufbaus. Beide bestehen aus einer Vielzahl von Teil-*Frameworks*, jedes für einen spezifischen Einsatzzwecke ausgelegt. Die wichtigsten zwei Teil-Frameworks sind: (vgl. [Seb11], S. 191ff)

- das bei der *Mac-OS-X*-Entwicklung *Application Kit* (kurz ***AppKit***) und bei der *iOS*-Entwicklung ***UIKit*** (von *User Interface Kit*) genannte Teil-Framework, welches Klassen und Funktionalität beinhaltet, die im Zusammenhang mit grafischen Benutzungsschnittstellen stehen, und
- das in beiden Entwicklungsumgebungen gleich benannte ***Foundation Kit*** (auch *Foundation Framework* genannt), welches für die nicht-grafische Funktionalität zuständig ist. Es enthält u.a.:
  - das Wurzelobjekt *NSObject*, welches das Grundverhalten von *Cocoa*-Objekten festlegt,
  - die *Objective-C*-Grunddatentypen (siehe Unterkapitel 3.2.3.3),
  - die Funktionalität für die Speicherverwaltung (siehe Unterkapitel 4.2.3), und
  - die Basis für das *Notifications* genannte System, welches die Apple Umsetzung des *Listener*-Entwurfsmusters darstellt und Objekten die Möglichkeit bietet sich über Zustandsänderungen anderer Objekte informieren zu lassen.

Über die Funktionalität von *Cocoa* hinaus, bietet *Cocoa Touch* u.a. noch:

- das ***Core Location*** *Framework*, das zur Bestimmung der Position und Ausrichtung von *iOS*-Geräten mit passenden Computerchips dient,
- Zugriff auf weitere Sensordaten wie Beschleunigungswerte, und
- Zugriff auf die Geräte-Kamera(s) und die Fotobibliothek.

4.2. iOS-Grundlagen

4.2.1. Aufbau einer iOS-Anwendung

4.2.1.1. Anwendung des Model-View-Controller-Entwurfmusters in Cocoa

Die Struktur des *Cocoa*-Framework wurde durch *Apple* stark an das Konzept der Trennung der Belange und das damit oft in Verbindung gebrachte Entwurfsmuster *Model-View-Controller* (kurz *MVC*) orientiert. Auch Applikationen welche auf dem *Cocoa*-Framework aufbauen sind nach diesem Entwurfsmuster strukturiert; bedingt durch die enge Verwebung der Anwendungsklassen und -funktionalität mit den *Framework*-Klassen und der -funktionalität. Das *MVC*-Konzept sieht eine Trennung der Programmfunktionalität in drei unabhängige Kategorien vor:

1. Das ***Model*** beinhaltet die Klassen welche die Datenstruktur der Anwendung ausmachen und für die Speicherung der Daten zuständig sind. Klassen dieser Kategorie enthalten keine Funktionalität um Daten anzuzeigen. Bei Datenänderungen informiert das *Model* lediglich die *View*.

2. Unter der ***View***-Kategorie sind diejenigen Klassen zusammengefasst, welche die grafischen Elemente eines Programms ausmachen.
3. Die ***Controller***-Familie enthält Klassen zur Beschreibung der Anwendungslogik. Sie verbinden und vermitteln zwischen den Klassen aus dem *Model* und der *View* und kümmern sich um die Datenverarbeitung sowie die Auswertung von Anwender-/ Programmeingaben.

Die dieser Einordnung zugrundeliegende Idee der Entkopplung von Anwendungskomponenten fußt auf dem Leitmotiv des objektorientierten Paradigmas, die Wiederverwendung von Klassen, Objekten und Funktionalität sowie auf Code-Ebene von Anweisungen, zu promovieren. Das *MVC*-Entwurfsmuster bietet sich als perfekte Ergänzung zur dynamischen Natur von *Objective-C* sowie der Funktionalität der damit verbundenen Laufzeitumgebung an (siehe Unterkapitel 3.2.4.1). Dieser Strukturierung Rechnung tragend, gehören Objekte in *Cocoa* stets einer dieser drei Kategorien an und unterscheiden sich daher fundamental voneinander[4]. Z.Bsp. wird das Aussehen eines grafischen Elements in einer Klasse beschrieben, ohne dass diese Klasse Code für die Verarbeitung, zur Laufzeit durch dieses Element, anfallender Daten enthält. Die durch Einhaltung des Prinzips entstehenden generischen Elemente ermöglichen ihren leichten Austausch, die Wiederverwendung dieser Komponenten in anderen Situationen oder Programmen und eine bessere Wartbarkeit. (vgl. [Dav11], S. 46)

In *Cocoa* (*Touch*) ist das *MVC*-Konzept wie folgt umgesetzt: (vgl. [Dav11], S. 46)

1. Die Datenstrukturen im *Model* werden programmatisch durch den Programmierer beschrieben. In *Cocoa* (*Touch*) sind solche Klassen in den Sprachen *C*, *C++* oder Objective-C geschrieben. Eine Mischung der Sprachen ist möglich (siehe Unterkapitel 3.2.1).
2. Die Klassen, welche zur *View* gehören, werden grafisch im *Xcode Interface Builder* gestaltet. Optional kann ihre Erstellung programmatisch oder aus einer Mischung des grafischen und programmatischen Ansatzes erfolgen (siehe Unterkapitel 4.3.2). Das in *Cocoa* enthaltene Benachrichtigungssystem meldet Veränderungen des *Models* an die *View* (vgl. [Seb11], S. 196; für Details zu *Notifications* und dem *Notification Center* siehe Programmierhandbuch [App09a]).
3. Der *Controller*-Teil besteht aus Klassen die meist in *Objective-C* verfasst sind und von einer beliebigen Superklasse stammen, sogar von der Wurzelklasse *NSObject*. Oft sind diese Klassen aber Unterklassen spezialisierter *GUI*-Klassen aus dem *AppKit*- oder *UIKit*-Framework, wie zum Beispiel die Klassen *UIViewController* oder *UITableViewController* (siehe Unterkapitel 4.2.1.2, 4.3.1 und 4.3.2). Solche spezialisierten Klassen sind bereits mit viel situationsbedingt nützlicher Funktionalität ausgestattet, ermöglichen aber eine Anpassung durch den Programmierer an die eigenen Bedürfnisse. Das Konzept durch das ein *Controller*-Objekt auf (Benutzer-)Ereignisse in der *View* reagiert, wird über ein zentrales *Cocoa*-Konzept abgewickelt, die *Delegation* (vgl. [Seb11], S. 196).

   Bei der Anwendung des *Delegation*-Prinzips wird die Funktionalität eines Elements/ Objekts/ Klasse in eine zweite Klasse ausgelagert, das *Delegation*-Objekt (auch ***Delegate*** genannt).

[4] Eine solche Trennung im *Framework* entspricht zwar der Konvention des *MVC*, der Compiler fordert diese aber nicht. Es gibt durchaus Programmsituationen, wo Objekte die Funktionalität aus mehr als einer *MVC*-Kategorie enthalten. Der Code, der die grafischen Eigenschaften eines Knopfes beschreibt kann zum Beispiel um Code ergänzt werden, welcher vorgibt, was bei Benutzung des Knopfes passiert. Dieser Code/ diese Klasse ist aber dadurch weniger generisch und weniger dazu geeignet wiederverwendet zu werden.

Zur Programmlaufzeit wird der eigentlichen Klasse eine Referenz auf das *Delegation*-Objekt übergeben, wodurch die Methoden des *Delegates* aufgerufen werden können. Diese Vorgehensweise wird durch das *Dynamic Loading* und die *Dynamic Method Resolution* der *Objective-C*-Laufzeitumgebung ermöglicht (siehe Unterkapitel 3.2.4.1). Dieser Vorgang erlaubt zum Beispiel, dass einem grafischen Element der *View*-Kategorie zur Laufzeit ein *Delegate*-Objekt zugeführt wird, das die Programmlogik zur Verarbeitung von Daten und Ereignissen für dieses grafische Element enthält – also ein Element der *Controller*-Kategorie[5]. Dieses Konzept ermöglicht eine saubere Aufteilung des Programmverhaltens auf Klassen der verschiedenen *MVC*-Kategorien und eine einhergehende leichte Austauschbarkeit und Erweiterbarkeit.

4.2.1.2. Die Kernelemente einer iOS-Applikation

In jedem *iOS*-Projekt lassen sich bestimmte Grundbausteine in Form der folgenden Klassen und Objekte identifizieren: (siehe Abb. 4.2):

- Die Klasse **`main`**, genauer gesagt die darin enthaltene Methode `main`, stellt den Einstiegspunkt in jede *iOS*-Anwendung dar. Dieser Methode obliegt es bestimmte *Cocoa* Start-Objekte zu erstellen, welche für den Anwendungsablauf unabdingbar sind. So werden u.a. ein `UIApplication`- und ein `AppDelegate`-Objekt erstellt. Die Klasse ist in *ANSI-C* geschrieben.

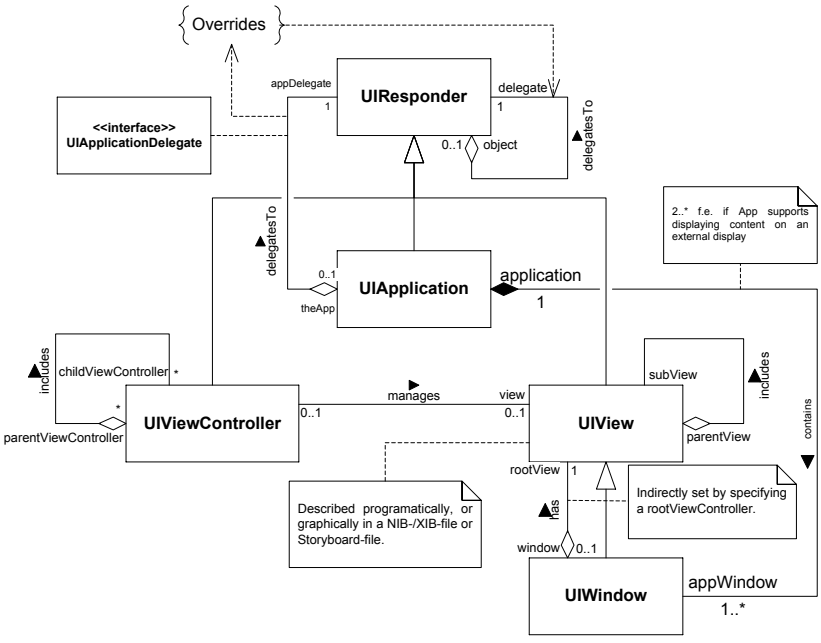

Abbildung 4.2.: Vom Autor erstelltes Klassendiagramm der Kernelemente und ihrer Beziehungen in einer *iOS*-Anwendung.

[5] Anmerkung: ein *Delegate* kann auch als Datenquelle dienen und somit neben der *Controller*-Kategorie und der *View*-Kategorie auch der *Model*-Kategorie angehören.

- Die Klasse **`UIApplication`** ist in *Objective-C* geschrieben und stellt den Eintrittspunkt in den objektorientierten Teil der Applikation dar. Die Klasse gilt als Grundpfeiler einer Anwendung und zeichnet sich als primärer Kontroll- und Koordinationspunkt der Anwendung verantwortlich. Für jede Anwendung existiert zur Laufzeit genau ein `UIApplication`-Instanzobjekt. Es besitzt ein auf sich registriertes `AppDelegate`-Objekt, welches das oberste *Delegation*-Objekt und damit das Haupt *Controller*-Objekt darstellt (siehe Unterkapitel 4.2.1.1). Zudem besitzt ein `UIApplication`-Objekt eine Referenz auf mindestens ein `UIWindow`-Objekt, welches den grafischen Inhalt der Anwendung enthält.
- Ein **`AppDelegate`**-Objekt stellt den obersten *Event-handler* einer Anwendung in der Hierarchie der Ereignisbehandlung dar und ist in jeder Anwendung einmalig. Jedes grafische Element der Anwendung kann ein *Delegation*-Objekt als seinen *Controller* bestimmen, der auf die Eingaben und Ereignisse dieses Elements reagiert. Die *GUI*-Elemente sind ineinander verschachtelt und somit indirekt auch ihre *Controller*. Reagiert ein bestimmter *Controller* nicht auf ein Ereignis, wird dieses zur Auswertung die Hierarchie hoch gereicht, bis es schließlich zur Auswertung beim `AppDelegate`-Objekt ankommt.

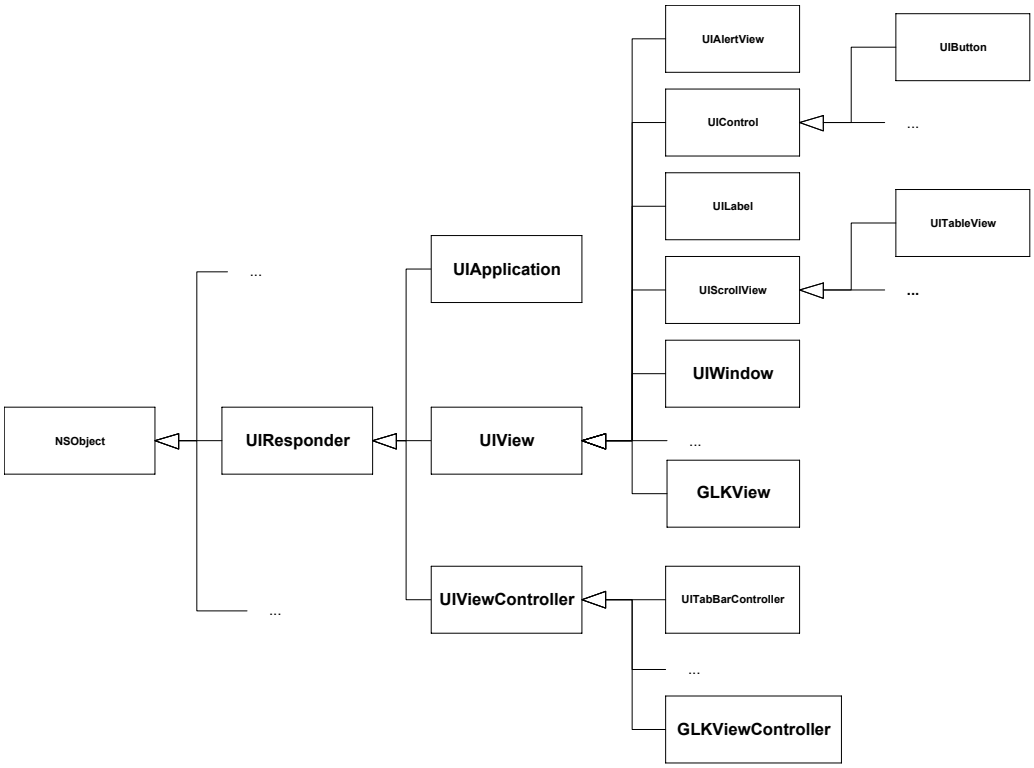

Abbildung 4.3.: Auszug der Ableitungshierarchie hervorzuhebender Klassen aus dem *UIKit-Framework*.

- Ein **`UIWindow`**-Objekt enthält den grafischen Inhalt einer Anwendung und stellt die Wurzel einer Hierarchie aus Anwendungssichten dar, welche durch `UIView`-Objekte repräsentiert

werden. Für jede Anwendung existiert typischerweise nur ein `UIWindow`-Instanzobjekt[6].

- Ein **`UIView`**-Objekt stellt eine rechteckige Fläche des Schirms dar, in der beliebige, grafische Anwendungselemente angezeigt werden können. Die Klasse `UIView` ist die Oberklasse vieler spezialisierter Unterklassen, die bestimmte Aufgabenbereiche abdecken. So sind zum Beispiel die Klasse `UITableView` auf die Darstellung von Tabellen und die Klasse `UIWebView` auf die Darstellung von Webseiten speziell ausgelegt. Auch grafische Anwendungselemente wie Knöpfe (`UIButton`-Objekte) sind der Klasse `UIView` untergeordnet (siehe Abb. 4.3).

  `UIView`-Objekte können ineinander verschachtelt werden und sich dabei partiell oder vollständig überlappen. Das unterste Element einer so entstehenden Hierarchie ist stets ein `UIWindow`-Objekt (siehe Abb. 4.4).

  Die Menge der `UIView`-Objekte, welche zu einem Zeitpunkt der Anwendung gleichzeitig am Schirm dargestellt werden, bilden eine Szene der Anwendung ab (siehe Unterkapitel 4.3.1).

  Jedem `UIView`-Objekt (*View*-Gruppe in *MVC*, siehe Unterkapitel 4.2.1.1) kann ein Delegation-Objekt (*Controller*-Gruppe in *MVC*, siehe ebd.) zugewiesen werden. Dieses enthält die Programmlogik um Ereignisse, welche am `UIView`-Objekt anfallen, zu verarbeiten.

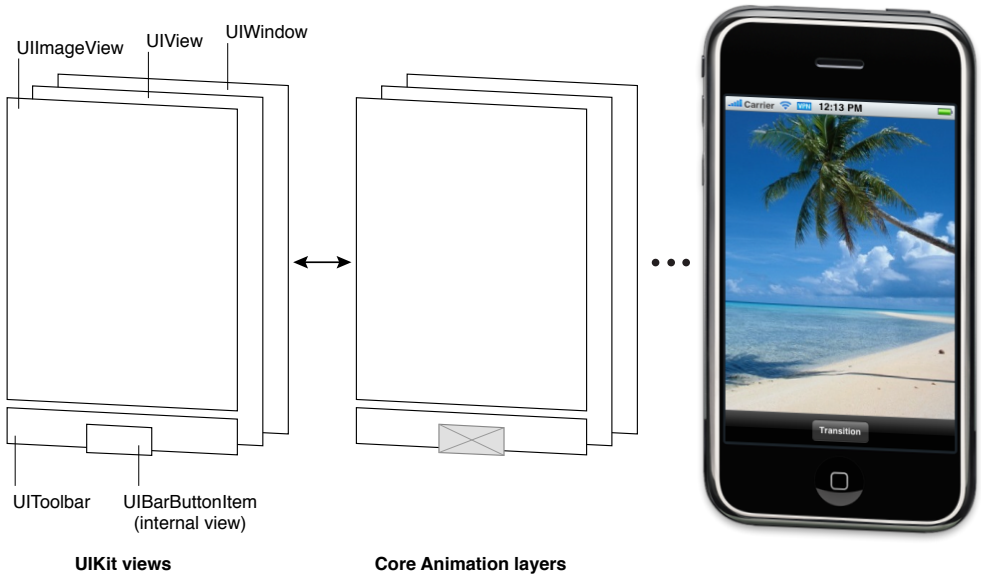

Abbildung 4.4.: Darstellung des hierarchischen Aufbaus einer grafischen Anwendung. (Bildquelle: [App11c], S. 11)

- Ein **`UIViewController`**-Objekt stellt ein *Delegation*-Objekt dar (siehe Unterkapitel 4.2.1.1), welches einem oder mehreren `UIView`-Objekten zugeteilt werden kann und für die Auswertung von Ereignissen zuständig ist, welche in diesem Sichten anfallen. Zudem kann ein

[6] Sind via *Apple AirPlay* weitere Schirme an einem *iOS*-Gerät angeschlossen, kann die Anwendung mehrere `UIWindow`-Objekte enthalten.

`UIViewController`-Objekt dafür verantwortlich sein, Aktualisierungen der mit dem Objekt gekoppelten Sicht durchzuführen, wenn Änderungen an Daten[7] auftreten, welche mit der Sicht verbunden sind (siehe ebd.).

`UIViewController`-Objekte können ineinander verschachtelt werden. Die dabei entstehende Hierarchie orientiert sich implizit an der Verschachtelung der `UIView`-Objekte, an die die `UIViewController`-Objekte gebunden sind. Das oberste `UIViewController`-Objekt der Hierarchie ist das `AppDelegate`-Instanzobjekt – jedes `UIViewController`-Objekte ist zumindest in diesem enthalten. Ist ein `UIViewController`-Objekt nicht dafür programmiert, auf ein bestimmtes Ereignis zu reagieren, wird das Ergebnis zur Auswertung entlang der Hierarchie hoch gereicht.

4.2.2. Objektinstantiierung in Cocoa

Im Unterschied zu anderen objektorientierten Sprachen gibt nicht die Sprachspezifikation von *Objective-C* vor „Wie" Klassen instantiiert werden, also anhand welcher Vorgänge und wie diese Vorgänge implementiert sind, sondern das ***Framework*** (siehe Unterkapitel 3.2.2.5 und vgl. [Seb11], S. 66). *Apple* schlägt zu diesem Zweck im *Cocoa-Framework* eine Konvention vor, welche in der Wurzelklasse **`NSObject`** realisiert wurde. Anwendungsentwickler können von dieser Referenzimplementierung gebraucht machen oder eine eigene Implementierung nutzen.

Die *Apple*-Konvention sieht eine zweigeteilte Objektinstantiierung vor:

1. In der **Allokationsphase** wird von der Laufzeitumgebung anhand der `NSObject`-Klassenmethode **`alloc`** für eine neue Objektinstanz Speicher angefordert. Der Speicher entstammt dem virtuellen Speicherbereich der Anwendung, welcher bei Programmstart für die Anwendung im physischen Speicher reserviert wird (siehe Unterkapitel 4.1.1). Die Menge allozierter Speicher orientiert sich an der Anzahl und den Typen der Objektvariablen. Die Methode gibt ein Instanzobjekt zurück, das in der aktuellen Form noch nicht nutzbar ist. Genauer gesagt wird ein Zeiger auf die Startadresse des Speicherbereichs zurückgeliefert, der für die Objektinstanz reserviert wurde. Die Methode `alloc` kann vom Programmierer überschrieben werden. (vgl. [App10a], S. 89)

   Außerdem erledigt die alloc-Methode noch folgende wichtige Aufgaben: (vgl. [Seb11], S. 66ff und [App10a], S. 89f)

   - Der reservierte Speicherbereich wird zur Sicherheit mit Nullen überschrieben, damit nicht fälschlicherweise auf Daten eines vorherigen Objekts zugegriffen wird.
   - Die `isa`-Instanzvariable des neu erstellten Objekts wird initialisiert. Sie stellt einen wichtigen Bestandteil des dynamischen Aspekts von *Objective-C* dar und stellt vereinfacht ausgedrückt eine Referenz auf die Ursprungsklasse des Objekts dar (siehe Unterkapitel 3.2.3.1).
   - Alle weiteren Instanzvariablen des Objekts werden initialisiert. Ganzzahlen werden unter anderem mit 0, Gleitkommazahlen mit 0.0 und Objekttypen mit `nil` initialisiert (siehe Unterkapitel 3.2.2.5).

[7] Die Daten stehen dabei in Klassen aus der *Model*-Gruppe im *MVC*-Konzept. Das `UIView`-Objekt entstammt der *View*-Gruppe und das `UIViewController`-Objekt der *Controller*-Gruppe. (siehe Unterkapitel 4.2.1.1)

- Außerdem wird ein, *Retain Count* genannter, Zähler initialisiert, welcher mitzählt, wie oft das Objekt während der Laufzeit im Programm referenziert wird. Da das neu erstellte Objekt bei seiner Allokation bereits von einem anderen Objekt referenziert wird, nämlich dem Erzeugerobjekt bzw. Sender der `alloc`-Nachricht (siehe Unterkapitel 3.2.4.1), wird der Zähler gleich um Eins inkrementiert.

  Das Unterkapitel 4.2.3 beschäftigt sich näher mit dieser Variable.

2. In der **Initialisierungsphase**, werden instanzbezogene Voreinstellungen für das Objekt getroffen. Hierfür stehen in `NSObject` zwei Methoden zur Verfügung, die in Unterklassen benutzt oder überschrieben werden können. Die Methoden unterscheiden sich dahingehend, ob ein Klassenobjekt oder ein Instanzobjekt erzeugt wird (siehe Unterkapitel 3.2.2.4).

   a) Die Klassenmethode **`initialize`** bietet sich für die Initialisierung eines Klassenobjekts an. Sie liefert ein initialisiertes und im Code regulär benutzbares Klassenobjekt zurück (im vgl. zur Methode `alloc`). Sie kann gefahrlos vom Programmierer in einer Unterklasse überschrieben werden.

   Zur Laufzeit wird diese Methode automatisch durch die Laufzeitumgebung aufgerufen. Genau dann, wenn ein Klassenobjekt für mehr als nur zur Erstellung von Instanzobjekten gebraucht wird, oder wenn Klassenmethoden aufgerufen werden sollen.

   Besitzt eine Klasse eine Superklasse – in Cocoa meist direkt oder indirekt die Klasse `NSObject` – und wird die `initialize`-Methode in der Unterklasse implementiert, wird die `initialize`-Methode der Superklasse zuerst aufgerufen. Überschreibt eine Unterklasse die `initialize`-Methode der Superklasse nicht, wird, wenn ein Klassenobjekt der Unterklasse benötigt wird, die `initialize`-Methode der Superklasse aufgerufen.

   Da stets nur ein Klassenobjekt pro Klasse existieren sollte und da im Fall, dass eine Unterklasse die `initialize`-Methode der Superklasse überschreibt, beide initialize-Methoden aufgerufen werden, muss der Anwendungsentwickler beim Überschreiben der Methode dafür sorgen, dass kein weiteres Klassenobjekt erstellt wird. Zur Vorbeugung schlägt *Apple* den in Codelisting 4.1 aufgeführten Aufbau der `initialize`-Methode vor.

```
1 +(void)initialize{
2   if(self == [ThisClass class]){
3     // Perform initialization here
4   }
5 }
```

Tabelle 4.1.: Vorschlag für den Aufbau der `initialize`-Methode gemäß *Apple*-Konvention.

In der zweiten Zeile der `initialize`-Methode wird durch Aufruf der Methode `class` ein Klassenobjekt erstellt und der Variable `self` zugewiesen (zur Rolle der Instanzvariable `self` siehe Unterkapitel 3.2.4.1). Wurde bereits ein Klassenobjekt erstellt, liefert

ein Aufruf der Methode `class` einen *Nullpointer* zurück, in Objective-C als `nil` repräsentiert (siehe Unterkapitel 3.2.2.5). Unter diesen Umständen wird die klassenbezogene Initialisierung nicht durchgeführt.

b) Zur Initialisierung von Instanzobjekten wird ähnlich verfahren, wie bei der Erstellung von Klassenobjekten. Die Klasse `NSObject` stellt zu diesem Zweck die Instanzmethode **`init`** zur Verfügung, welche sich für instanzbezogene Voreinstellungen anbietet. Sie liefert ein initialisiertes und im Code regulär benutzbares Klassenobjekt zurück (im vgl. zur Methode `alloc`). Damit die Methode in Kombination mit allen Klassen benutzbar ist, besitzt sie den generischen Rückgabetyp `id` (siehe Unterkapitel 3.2.3.3).

   Die Methode kann in beliebigen Unterklassen überschrieben werden. Zudem kann sie um weitere Methoden ergänzt werden, deren Namen nach Konvention mit `init` beginnen sollten und um beliebige Parameter ergänzt sein können. In Zusammenhang mit der `init`-Methode, oder erweiterte Methoden, zur Initialisierung von Instanzobjekten wird allgemein von Konstruktoren oder Konstruktor-Methoden gesprochen.

   Per Konvention sollten die Konstruktor-Methoden einer Klasse, die eine Oberklasse besitzt, einen Konstruktor der Oberklasse aufrufen, da jede Stufe der Ableitungshierarchie eigenständige, instanzbezogene Einstellungen am Instanzobjekt vornehmen kann. *Apple* schlägt das Schema in Codelisting 4.2 für Konstruktor-Methoden vor.

```
1 - (id) init{
2   if(self = [super init]){
3     // Custom initialization
4   }
5   return self;
6 }
```

Tabelle 4.2.: Vorschlag für den Aufbau einer Konstruktor-Methode gemäß *Apple*-Konvention.

In der zweiten Zeile der `init`-Methode wird zuerst die Initialisierungsmethode der Superklasse aufgerufen und der Rückgabewert der Instanzvariable `self` zugewiesen (zur Rolle der Instanzvariable `self` siehe Unterkapitel 3.2.4.1). Schlägt die Initialisierung aus irgendeinem Grund fehl, oder ist die Initialisierung eines Instanzobjekts gänzlich oder nur anhand einer bestimmten Initialisierungsmethode verboten, kann die Initialisierung auch `nil` zurückgeben (zur Bedeutung von `nil`, siehe Unterkapitel 3.2.2.5). Die *Apple*-Konvention sieht aus diesem Grund im Code einer jeden Konstruktor-Methode eine `if`-Anweisung vor, welche überprüft ob die Initialisierungsroutine der Superklasse erfolgreich war. Falls dies der Fall ist, können benutzerdefinierte Initialisierungen des Instanzobjekts erfolgen. Abschließend wird ein Zeiger auf das eben initialisierte Instanzobjekt zurückgegeben. War die Initialisierung nicht erfolgreich wird ein Nullpointer zurückgegeben.

Über Sinn und Unsinn einer, wie hier vorgestellten, Objekterzeugung in zwei Schritten wird häufig diskutiert. Auch über Apples Entscheidung den Vorgang der Initialisierung nicht in der Laufzeitumgebung oder Sprachspezifikation festzulegen, sondern ihn in die Klassendefinition zu verla-

gern, ist ein fortwährender Diskussionsthema. Ein Vorteil dieses Vorgehens ist, dass dem Programmierer die Möglichkeit gegeben wird, möglichst tiefgreifenden Einfluss auf die gesamten Vorgänge der Objektinitialisierung zu nehmen. (vgl. [Seb11], S. 66ff)

Um den Code kürzer und leserlicher zu gestalten, bietet *Apple* das Konzept spezieller Konstruktor-Methoden an, ***Convenience-Konstruktoren*** genannt. Solche Methoden abstrahieren den Vorgang der Objekterzeugung indem sie beide Erstellungsphasen gruppieren. Intern ruft ein *Convenience-Konstruktor* lediglich die Allozierungs- und die entsprechende Initialisierungsmethode auf, je nachdem ob es sich um ein Klassen- oder Instanzobjekt handelt. Auch für den Namen solcher *Convenience-Konstruktoren* schlägt *Apple* eine Konvention vor. Demnach beginnen *Convenience-Konstruktoren* namentlich mit dem Namen der Klasse in der sie implementiert sind. Als Beispiel sei der *Convenience-Konstruktor* `stringWithString:` der Klasse `NSString` erwähnt, welcher ein neues Objekt, das eine Zeichenkette repräsentiert, alloziert und initialisiert, indem die als Parameter angegebene Zeichenkette Zeichen für Zeichen kopiert wird.

4.2.3. Speicherverwaltung

Die Speicherverwaltung nimmt einen wichtigen Teil der Programmierung ein. Insbesondere bei der Entwicklung von Anwendungen mit aufwendigen, objektorientierten *Frameworks*, welche auf Mobilgeräten mit wenig Arbeitsspeicher laufen sollen. Die von *Apple* für die Programmierung mit *Cocoa* ausgelieferte Laufzeitumgebung bietet zwei Möglichkeiten um Sicherzustellen, dass Objekte, solange sie gebraucht werden erhalten bleiben, und zerstört werden, wenn sie nicht mehr benötigt werden um somit freien Speicher zu erhalten:

1. Anhand eines ***Garbage Collectors*** identifiziert die Laufzeitumgebung selbständig, welche Objekte nicht mehr gebraucht werden und zerstört sie, wenn freier Speicher benötigt wird. Um dies zu erreichen kann ein *Garbage Collector* zirkuläre Abhängigkeiten zwischen Objekten erkennen und auflösen. Zudem verwaltet er für jedes Objekt einen Zähler, welcher die Anzahl Referenzen auf ein Objekt festhält. Fällt der Zähler auf Null, kann das Objekt aus dem Speicher entfernt werden.

   Unter *iOS* ist diese Methode aber derzeit nicht realisiert, weil Algorithmen zur *Garbage Collection* viel Rechenaufwand benötigen und sich somit nicht für mobile Systeme eignen (siehe [App10a], S.81ff).

2. Eine weniger raffinierte, dafür Ressourcen-schonendere, Methode der Speicherverwaltung, delegiert die Aufgabe zur Auflösung zirkulärer Abhängigkeiten zwischen Objekten an den Anwendungsentwickler. Jedes Objekt im *Cocoa Touch Framework* besitzt einen ***Retain Count*** genannten Zähler, dessen Rolle zur Laufzeit es ist, die Besitzansprüche an dieses Objekt zu verfolgen. Bei der Objektinitialisierung und der Zuweisung eines Objekts an eine Variable wird der Zähler inkrementiert. Wird ein Objekt von einem anderen nicht mehr benötigt, wird der Zählerstand reduziert. Erreicht der Zähler Null, wird das Objekt zerstört.

   Bis zum Erscheinen der *iOS* Version *5* musste dieser ***Reference Counting*** genannte Prozess durch den Programmierer manuell durchgeführt werden. Bei Objektinitialisierungen und bei der Erstellung von Beziehungen zwischen Objekten mussten die Besitzansprüche mit dem

Schlüsselwort `retain` angegeben sowie mit `release` oder `autorelease` wieder freigegeben werden.

Mit *iOS 5* und der damit eingeführten neusten Version der *Apple-LLVM*-Laufzeitumgebung wird dem Programmierer diese lästige Arbeit, durch das neue Konzept des ***Automatic Reference Counting*** – kurz *ARC* genannt – abgenommen. Vom Entwickler wird nicht mehr verlangt an geeigneten Stellen seiner Anwendung `retain`- und `release`-/ `autorelease`-Anweisungen anzugeben. Stattdessen übernimmt der Compiler das automatische Einfügen von `retain`-Anweisungen bei Objekterzeugungen und Objektzuweisungen, sowie von `release`-Anweisungen bei der Auflösung von Besitzansprüchen.

Anzumerken ist aber, dass *ARC* nicht in der Lage ist, **z**yklische Abhängigkeiten zwischen Objekten aufzudecken und aufzulösen. Dafür muss der Programmierer weiterhin selber Sorge tragen. Hierfür muss er Abhängigkeitsschleifen zwischen Objekten manuell auflösen, durch zutreffende Angabe der Schlüsselwörter `strong` und `weak` als Variablen- bzw. *Property*-Attribute (siehe Unterkapitel 3.2.3.5, Abschnitt „*Property*-Attribute" und das dort beschriebene Beispiel 3.32).

Für weitere Details zum Thema *Reference Counting* und *ARC* siehe [LLV12] und [App12b].

4.3. Einführung in die Funktionalität des iOS-SDK

Im folgenden werden einige Funktionen des *Cocoa Touch Frameworks* und der Entwicklungsumgebung *Xcode* eingeführt, welche zur Erstellung der *iRoller2000*-Anwendung im Rahmen dieser Diplomarbeit benötigt wurden (siehe Kapitel 5). Die Vorstellung der Funktionalität erfolgt anhand einer Beispielanwendung, die schrittweise eingeführt und aufgebaut wird. Dem Leser wird durch detaillierte Beschreibungen des Vorgehens die Möglichkeit gegeben, den Entwicklungsprozess der Beispielanwendung und der *iRoller2000*-Applikation besser nachzuvollziehen. Außerdem kann das Beispielprogramm anhand der Anleitung selbst erstellt werden, was zu tieferen Kenntnissen der Materie führen sowie zu Experimenten verführen soll.

Die Beispielanwendung befindet sich als *Xcode-Projekt* auf dem Speichermedium, welches dieser Diplomarbeit beigefügt ist (siehe Appendix VI). Die Beschreibung der Funktionalität und die Beispielanwendung wurden dabei mit folgender Software realisiert:

- auf dem Produktivsystem läuft das Betriebssystem *Mac OS X 10.7.3*,
- als Entwicklungsumgebung wird *Xcode 4.3.2* verwendet,
- als *API* kommt das *iOS-SDK 5.1* zum Einsatz,
- für das Beispielprojekt wird mit aktiviertem *Automatic Reference Counting* entwickelt (siehe Unterkapitel 4.2.3), und
- als Betriebssystem auf den mobilen Test-Geräten kommt *iOS 5.0.1* zum Einsatz.

4.3.1. Erster Schritt: Die Basics

4.3.1.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- Erstellung eines einfachen *iOS*-Projekts,
- kurze Vorstellung der Entwicklungsumgebung *Xcode*,
- Vorstellung der Elemente des neu angelegten Projekts,
- Einfügen einfacher *GUI*-Elemente,
- Manipulation von Attributen der grafischen Elemente in der *IDE*, und
- Erweiterung der rein grafischen Elemente um selbst geschriebene Applikationslogik.

4.3.1.2. Implementierung

Um mit der Entwicklung der Beispielanwendung beginnen zu können, muss in *Xcode* über den Menüeintrag *File* > *New* > *Project* ein neues Projekt angelegt werden. Ist der Startbildschirm von *Xcode* aktiviert muss dieser Schritt alternativ über die Schaltfläche „*Create a new Xcode project*" erfolgen (siehe Abb. 4.5).

Abbildung 4.5.: Der *Xcode*-Willkommensbildschirm. Ein Klick auf die Schaltfläche „*Create a new Xcode project*" startet den Werdegang zum Anlegen eines neuen Projekts. Die im Bild leere, weiße Fläche auf der rechten Seite listet etwaige bereits mit *Xcode* bearbeitete Projekte auf.

Im Anschluss präsentiert sich ein Auswahlmenü in dem sich der Entwickler zwischen verschieden Projektvorlagen entscheiden kann (siehe Abb. 4.6). Die Kategorie „*iOS* > *Application*" bietet Schablonen für grafische Anwendungen auf Basis des *Cocoa Touch Frameworks*, welche neben den

nötigen Klassen auch Zielplattformbeschreibungen und Kompilerkonfigurationen mit sich bringen. Aus dieser Kategorie wird für die Beispielanwendung das *Template „Single View Application"* gewählt.

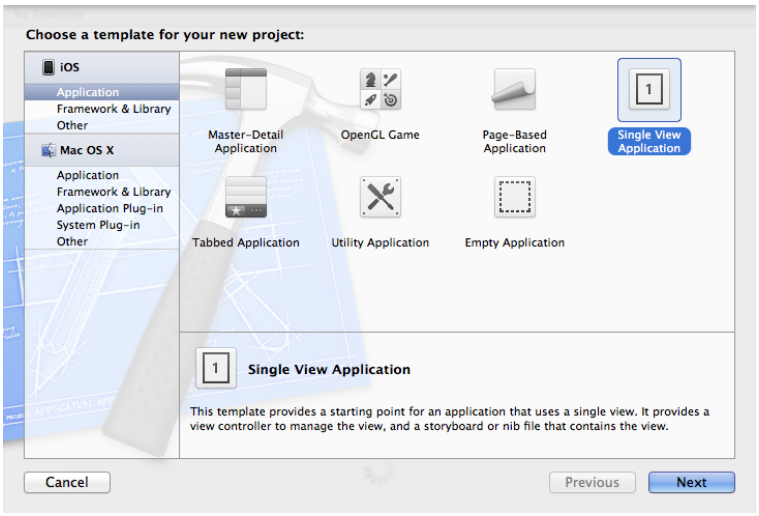

Abbildung 4.6.: Erster Schritt bei der Erstellung eines neuen Projekts: die Auswahl einer passenden Projektvorlage.

Nach einem Klick auf den Knopf *„Next"* können Einstellungen für das neue Projekt vorgenommen werden (siehe Abb. 4.7). Neben dem Anwendungs- und Firmen-/ Entwicklernamen kann ein Präfix festgelegt werden, das automatisch allen in diesem Projekt erstellten *Objective-C*-Klassen vorangestellt wird. Die Vergabe eines Präfix ist nicht zwingend erforderlich, kann aber helfen Konflikten mit Bezeichnern *Apple*-eigener Klassen oder Klassen aus fremden Projekten vorzubeugen. Zu den Entscheidungen die der Programmierer bei den Projekteinstellungen treffen muss, gehört auch ob seine Anwendung nur für *iPhones/ iPod Touchs*, nur für *iPad*s, oder für beide Gerätefamilien entwickelt wird. Dann folgen in den Einstellungen drei Kontrollkästchen, wovon das erste vorgibt ob ein *Storyboard* mit dem Projekt angelegt werden soll. Ein *Storyboard* stellt eine bequeme und einfache Möglichkeit dar, Anwendungen im, in *Xcode* eingebauten, *Interface Builder* grafisch zu erstellen (für Details zu Storyboards, siehe Unterkapitel 4.3.2). Das zweite Kontrollkästchen aktiviert die Nutzung des Speicherverwaltungskonzepts *Automatic Reference Counting* (kurz *ARC*), welche den Programmierer die Arbeit abnimmt, Objekte mit `retain-` und `release-` / `autorelease-Methoden` Speicher-technisch zu verwalten (siehe Unterkapitel 4.2.3). Das letzte Kästchen legt bei der Projekterzeugung Testklassen für Unittests an.

In diesem Dialog wird für die Beispielanwendung der Produktname *„iOS-TestApp"* und der Firmenbezeichner *„de.uni_koblenz"* vergeben. Als Zielplattform wird die *iPhone*-Gerätefamilie ausgesucht. Die Erstellung der Anwendung soll möglichst leicht und übersichtlich anhand des *Interface Builders* erfolgen, weshalb ein Haken bei *„Use Storyboards"* gesetzt wird. Um Arbeit und Fehlerquellen zu vermeiden wird *ARC* zugeschaltet. Auf Unittests und einen Klassenpräfix wird verzichtet.

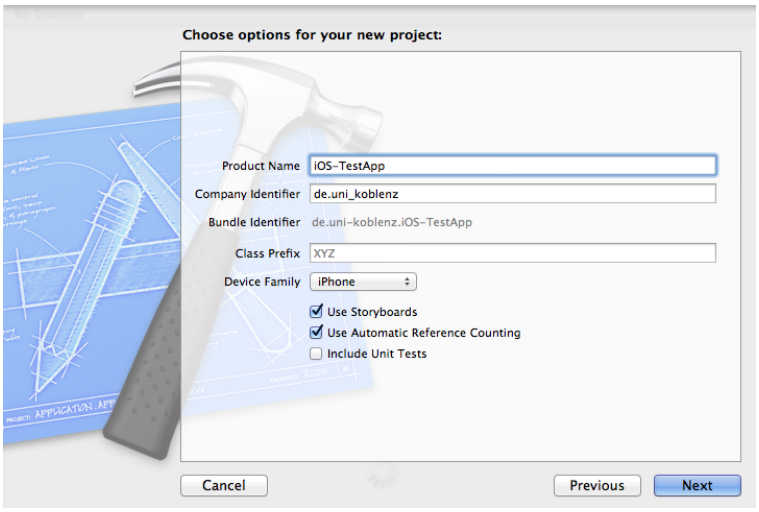

Abbildung 4.7.: Zweiter Schritt beim Anlegen eines neuen Projekts: die Festlegung genereller Projekteigenschaften.

Nach einem weiteren Klick auf die Schaltfläche „*Next*" wird der Entwickler aufgefordert einen Ordner anzugeben, in dem der Basisordner für das Projekt hinterlegt werden soll (siehe Abb. 4.8). Um den Konventionen zur Benennung des Namensraums eines Projekts gerecht zu werden, wird das Beispielprojekt in einem Toplevel-Ordner „*de*", in einem Unterordner „*uni_koblenz*" abgelegt. Somit ergibt sich für die Beispielanwendung „*iOS-TestApp*", nach Erstellung des Projekts, die Pfadstruktur „*de/uni_koblenz/iOS-TestApp*".

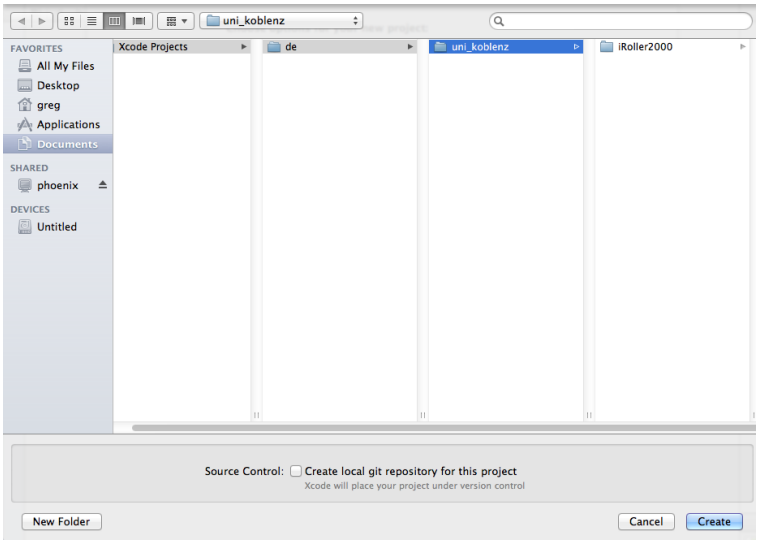

Abbildung 4.8.: Letzter Schritt der Projekterstellung: die Angabe des Speicherorts für die Anwendungsdaten.

Nach dem Drücken der Schaltfläche „*Create*" wird das Projekt angelegt. Das Hauptfenster der *Xcode*-Entwicklungsumgebung, ***Workspace window***[8] genannt, wird für das aktuelle Projekt geöffnet (siehe Abb. 4.9). Es ist in mehrere Bereiche unterteilt (siehe Abb. 4.10):

- Im oberen Teil des Fensters befindet sich die ***Toolbar***. Hier kann das Projekt mit einem Klick auf den linken runden Knopf gestartet werden. Im *Dropdown*-Menü daneben wird die Zielplattform für die Kompilierung ausgewählt – z.Bsp. den *iOS Simulator* oder ein angeschlossenes Gerät. Ganz rechts können Einstellungen am Aussehen des Arbeitsplatzfensters vorgenommen werden. Die dort ansässigen, ersten drei Buttons stellen die Anzahl der Ansichten in der Hauptansicht ein (Mittlerer Bereich unter der Toolbar). Wird z.Bsp. der Knopf *„Assistant Editor"* angeklickt, erscheint in der Hauptansicht ein zweiter Dateieditor. Dieser eignet sich gut zur gleichzeitigen Editierung einer Klassendefinition während in der zweiten Ansicht die Klassendeklaration oder eine andere Klasse, die für die aktuell modifizierte Klasse eine wichtige Rolle einnimmt, angezeigt wird. Die drei Knöpfe daneben schalten diverse Bereiche der Xcode Benutzungsoberfläche an oder aus, um Platz zu sparen oder es dem Entwickler zu erlauben sich auf das Nötigste zu konzentrieren. Ein Klick auf den „*Organizer*"-Knopf öffnet ein neues Fenster namens „*Organizer*". Der *Organizer* eignet sich u.a zur Anzeige der am System angeschlossenen Mobilgeräte und zum Studium von Dokumentationen zu bspw. Xcode oder dem *iOS-SDK* (siehe Abb. 4.11).

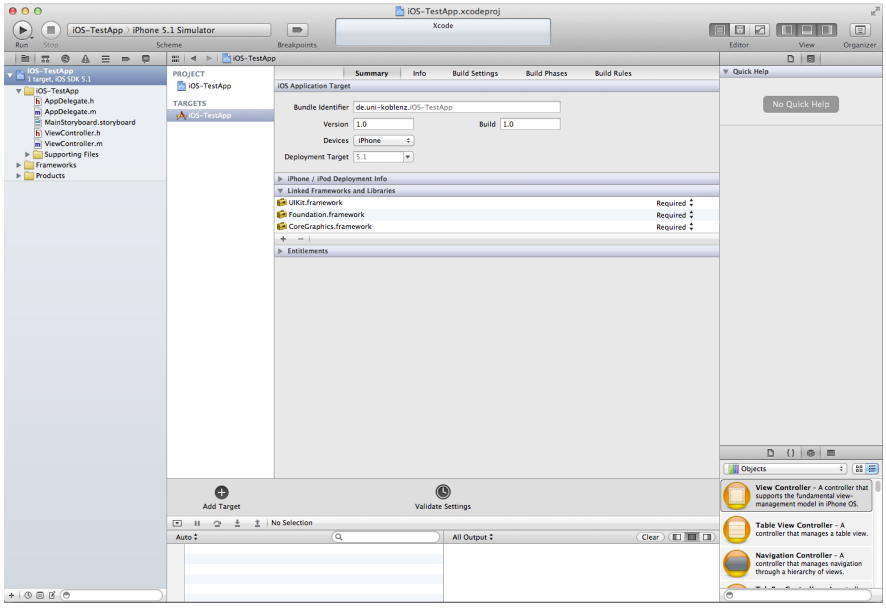

Abbildung 4.9.: Hauptansicht der *Xcode*-Entwicklungsumgebung nach der Erstellung eines Projekts.

[8] Da die *Xcode*-Oberfläche bisher nicht ins Deutsche übersetzt wurde, werden im Folgenden die englischen Bezeichnungen verwendet um Verwechslungen und Verwirrung auszuschließen.

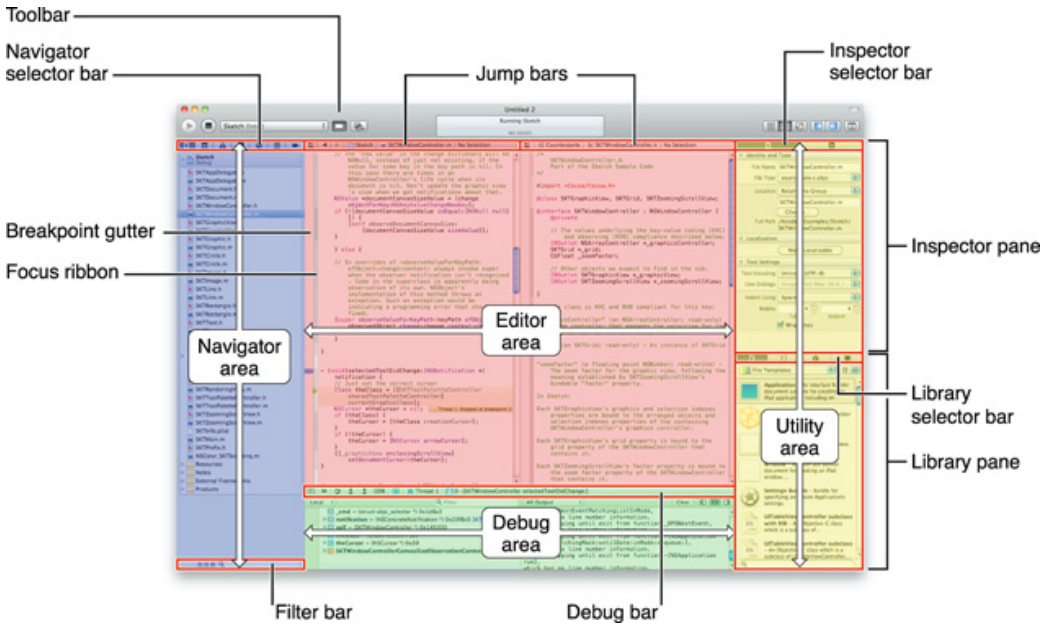

Abbildung 4.10.: Übersicht der verschiedenen Bereiche in dem *Xcode 4 Workspace window*. (Bildquelle: [App11d])

Abbildung 4.11.: Blick in die Dokumentation des *iOS-SDK 5.1* in der *„Documentation"*-Unteransicht des *Organizers*.

- Auf der linken Seite unter der *Toolbar* befindet sich der ***Navigator***. In einer kleinen Leiste am oberen Rand dieses Bereichs kann zwischen verschiedenen Unteransichten gewechselt werden. Die prominenteste Stelle nimmt der unter dem ersten Icon erreichbare ***Project navigator*** ein. Hier können alle Projektdateien angezeigt, aufgerufen, verschoben oder gelöscht und neue Dateien erstellt werden. (für Details zum *Project navigator* siehe Abschnitt nach der

hier aufgeführten Aufzählung)

- Die Hauptansicht in der Mitte unter der Toolbar bildet den Kern von *Xcode* und wird ***Editor pane*** genannt. Hier werden die Projektdateien in einem, dem Dateiformat entsprechend angepassten, Editor bearbeitet. Direkt nach Projekterstellung werden hier die allgemeinen Projekteinstellungen sowie u.a. die eingebundenen Frameworks angezeigt. Im späteren Verlauf lässt sich zu den Projekteinstellungen zurückkehren, indem das Wurzelobjekt des Projekts im *Project navigator* ausgewählt wird. Wird eine Datei mit Quellcode ausgewählt erscheint der Code-Editor im *Editor pane*. Bei der Auswahl einer Datei welche die Anwendung grafisch beschreibt – eine *Nib-/ Xib-* oder *Storyboard*-Datei – wird der seit Version *4* direkt in *Xcode* integrierte *Interface Builder* im *Editor pane* angezeigt.
- Den rechten Bereich des *Workspace window* belegt die ***Utility bar***, welche wiederum in zwei Unterbereiche aufgeteilt ist:
  - Im oberen Segment, dem ***Inspector pane***, können diverse Angaben wie die Identität, die (grafischen) Eigenschaften und die Objektbeziehungen der gerade editierten Datei oder des gerade selektierten *GUI*-Element anzeigt werden.
  - Im unteren Bereich der *Utility bar* befindet sich die ***Library Pane***, welche Vorlagen für Code-Bausteine und *GUI*-Elemente bereithält. Ein solches Template lässt sich einfach durch *Drag&Drop* in den *Editor pane* und damit in die Anwendung einfügen.
- Den Bereich am mittleren unteren Rand des *Workspace window*, bildet die ***Debug area***. Hier können zu *Debug*-Zwecken bei Erreichen eines *Breakpoints* die zum aktuellen Laufzeitpunkt bestehenden Objekte und ihre Eigenschaften eingesehen werden. Außerdem können hier, während der Laufzeit anfallende, benutzerprogrammierte *Console-* und *Debugger*-Ausgaben angezeigt werden.

Im *Project navigator* können folgende Elemente ausgemacht werden:

- Das oberste Element mit der Bezeichnung „*iOS-TestApp*" stellt das Wurzelobjekt des Projekts dar. Ein Klick darauf offenbart allgemeine Einstellmöglichkeiten zum Projekt im Editor. Diese Ansicht entspricht der Ausgangsansicht nach Erstellung eines neuen Projekts (siehe Abb. 4.9). Hier können u.a. neue *Targets* angelegt, eine feingranulare Konfigurationen des gesamten Build-Prozess vorgenommen sowie die eingebundenen Bibliotheken eingesehen und geändert werden.
- Darunter befinden sich drei Gruppen, welche durch Ordner-Symbole repräsentiert sind und jeweils einen grauen Pfeil, zum Auf- und Zuklappen der Ordnerstruktur, links neben ihrem Bezeichner, aufweisen. Neue Gruppen können über das mit der rechten Maustaste erreichbare Kontextmenü erstellt werden. Gruppen können umbenannt, verschoben oder gelöscht werden.

  Eine Gruppe fasst Dateien zu einer logischen Einheit in *Xcode* zusammen, um mehr Ordnung zu schaffen. Zu einer Gruppe wird kein Ordner im Projektpfad angelegt. Einer Gruppe angehörige Dateien werden physikalisch nicht von Dateien aus anderen Gruppen getrennt. Nach der Erstellung eines neuen Projekts sind üblicherweise drei Gruppen vorzufinden:

– Die erste Gruppe, welche als Bezeichner den Namen des Projekts – hier „*iOS-TestApp*" – trägt, gruppiert alle Quellcodedateien und grafischen Dateien zusammen, die der Programmierer bearbeiten kann. Zudem gibt es eine Untergruppe für zusätzliche Dateien wie u.a. *Icons* und Bilder.

– Die Gruppe namens *Frameworks* enthält alle in das Projekt eingebundene Bibliotheken. Einfluss auf die hier aufgeführten Elemente lässt sich über das Wurzelobjekt des *Project navigators*, in der damit verbundenen Ansicht der allgemeinen Projekteinstellungen, nehmen.

– Die „*Products*"-Gruppe enthält eine compilierte Fassung der Anwendung, für jedes, in den Projekteinstellungen angelegtes, *Target*.

Bei näherem Ansehen der Dateien der ersten Gruppe – in der Beispielanwendung die Gruppe namens „*iOS-TestApp*" – fallen die folgenden Dateitypen und Dateien auf:

- Auf *.h* und *.m* endende Dateien stellen Klassendeklarationen bzw. Klassendefinitionen dar (siehe Unterkapitel 3.2.1.2 und 3.2.2.1). Solche Dateien können in *Xcode*-Projekten für etliche, teils ganz unterschiedliche Zwecke eingesetzt werden. So beschreibt z.Bsp. die Klasse `AppDelegate` die oberste *Controller*-Instanz der Anwendung (siehe Unterkapitel 4.2.1). Die Klasse *ViewController* behandelt dagegen das *Controller*-Objekt welches die Programmlogik für die einzige Sicht der Anwendung enthält (siehe Unterkapitel 4.2.1). Weiterhin können Instanzen dieser Datentypen u.a. für die Programmierung von Netzwerkroutinen oder zur Datenhaltung dienen (siehe Unterkapitel 5.3.3).

  Beim Aufruf einer solchen Datei, wechselt der *Xcode Editor pane* in den Modus für die Code-Editierung (siehe Abb. 4.12).

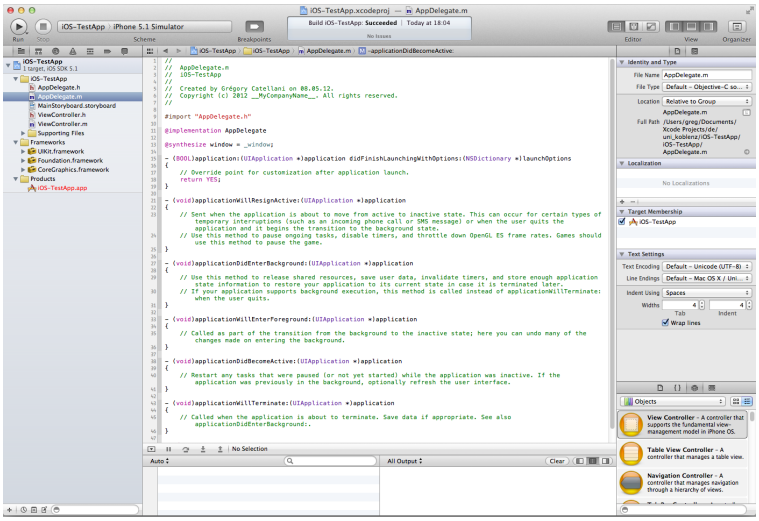

Abbildung 4.12.: Anzeige des Code-Editors im *Editor pane*, nachdem im *Project navigator* eine Code-Datei ausgewählt wurde.

- Dateien mit der Endung *.nib*, *.xib* und *.storyboard* beschreiben die Anwendung anhand grafischer Komponenten in Szenen. Eine **Szene** beschreibt den Inhalt der Anwendung zu einem

bestimmten Zeitpunkt der Laufzeit und setzt sich zumeist aus einem `UIViewController` und einer `UIView` zusammen – oder aus Unterklassen dieser Klassen.

Während in *.nib-* und *.xib*-Dateien eine einzige Anwendungsszene modelliert werden kann und Transitionen zwischen Szenen im Code beschrieben werden müssen, dienen die mit *iOS 5* eingeführten *Storyboards* dazu gleich mehrere Szenen einer Anwendung und die Abfolge in der sie dem Anwender präsentiert werden, auf einmal, in einer einzigen Datei grafisch zu beschreiben.

Bei Auswahl eines Ablegers dieser Dateitypen, z.Bsp. *MainStoryboard.storyboard*, zeigt der *Editor pane* den grafischen Editor *Interface Builder* an, der seit *Xcode 4* direkt in Xcode integriert ist (siehe Abb. 4.13).

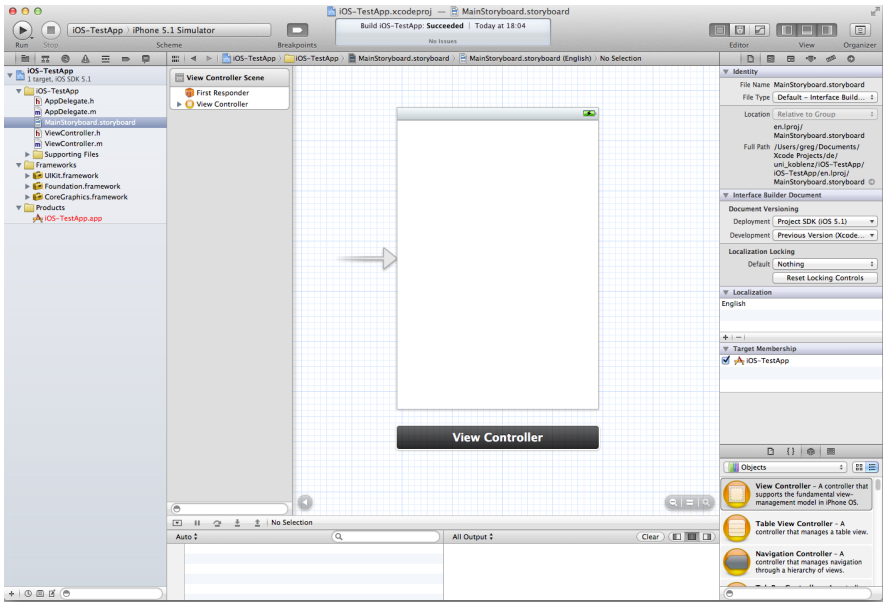

Abbildung 4.13.: Anzeige des *Interface Builder* im *Editor pane*, nachdem eine grafische Beschreibungsdatei im *Project navigator* markiert wurde.

Nachdem das Verfahren zur Erstellung eines neuen Projekts, die Grundbausteine von *Xcode* und die eines *Xcode*-Projekts vorgestellt wurden, werden erste grafische Elemente in die Anwendung eingeführt. Hierzu wird die Datei *MainStoryboard.storyboard* im *Interface Builder* geöffnet (siehe Abb. 4.13). In dieser Ansicht lassen sich folgende Elemente ausmachen:

- Die Leiste an der linken Seite beinhaltet für jede Anwendungsszene eine Gruppe der Elemente dieser Szene. Die Beispielanwendung weist aktuell nur eine Szene namens „*View Controller Scene*" auf (siehe Abb. 4.13).

  Für jede Szene existiert ein ***First Responder***. Hierbei handelt es sich um das grafische Element das den Fokus, direkt nachdem die Szene, in dem dieses Element enthalten ist, angezeigt wird, erhält. Konkret muss es sich beim Element, das den Fokus erhalten soll, um ein Objekt vom Typ `UIResponder` handeln (siehe Abb. 4.3).

Unter dem *First Responder* einer Szene befinden sich alle grafischen Elemente dieser Szene. Der kleine Pfeil neben „*View Controller*" weist darauf hin, dass weitere Elemente in diesem Element enthalten sind. So verfügt das „*View Controller*"-Objekt über ein eingebettetes *View*-Objekt das durch das „*View Controller*"-Objekt verwaltet wird – zu sehen wenn der Pfeil angeklickt wird. Wird ein Element der Liste angeklickt, wird es im Interface Builder markiert, was am bläulichen Rahmen um das Element ersichtlich ist.

Am unteren Rand der Leiste befindet sich ein Suchfeld, das dem schnellen Auffinden von Elementen und der besseren Orientierung in komplexen grafischen Oberflächen dient.

- Rechts neben der Leiste werden alle Elemente der Datei grafisch dargestellt. Die Beispielanwendung beinhaltet zu diesem Zeitpunkt nur eine Szene, sie wird durch das Objekt „*View Controller*" dargestellt. Dieses Objekt wird als rechteckige, umrahmte, weiße Fläche mit der aus *iOS* bekannten *Status bar* am oberen Ende, repräsentiert.

  Der leicht abgesetzte, schwarze Balken ist mit einer Szene verbunden, wird aber zur Laufzeit nicht angezeigt. Ein Klick auf einen solchen Balken markiert die komplette, an ihn gebundene, Szene im *Interface Builder*. Die im Balken hinterlegten Elemente sind die selben, wie sie in der linken Leiste des *Interface Builder* für diese Szene zu finden sind. Ein Klick auf das linke *Icon* wählt den First Responder für diese Szene, sofern festgelegt. Das rechte Symbol markiert die gesamte Szene. Eine so markierte Szene lässt sich mit der Maus im *Interface Builder* verschieben und neu anordnen.

  Der Pfeil, welcher in die linke Seite der „*View Controller*"-Szene mündet, legt fest, dass diese Szene nach dem Start der Anwendung als erste angezeigt wird. Pfeile dieser Art sind öfters in Storyboards anzutreffen und werden ***Segue*** genannt. Hierbei handelt es sich um ein an das *Storyboard*-Konzept gekoppeltes, mit *iOS 5* neu eingeführtes Element welches Transitionen zwischen Szenen der Anwendung im *Interface Builder* grafisch darstellt (hierzu mehr in Unterkapitel 4.3.2 und 4.3.3).

Als erstes werden drei Beschriftungen (in Cocoa `UILabel` genannt), zwei Textfelder (`UITextField`), ein Wippschalter (`UISwitch`) und ein Knopf (`UIButton`) in der „*View Controller*"-Szene angelegt. Diese grafischen Elemente können sowohl im Code oder im *Interface Builder* erstellt werden. Da die zweite Methode dem Programmierer etlichen Aufwand erspart und das *Layouting* einer Szene enorm erleichtert, wurde diese für die Erstellung der Beispielanwendung gewählt.

Das erste grafische Element, eine Beschriftung, wird unten rechts in der *Library pane* ausgesucht. Hierzu muss die „***Object Library***" in der Leiste am oberen Ende der *Library pane* ausgewählt werden – das vorletzte Icon in Form eines Würfels (siehe Abb. 4.14). Zum schnellen Auffinden eines bestimmten *GUI*-Elements steht das Suchfeld am unteren Rand zur Verfügung. Die Suche nach dem Begriff *Label* fördert das gewünschte Ergebnis. Per *Drag&Drop* wird das Element dann in die einzige aktuell vorhandene Szene einfügt (siehe Abb. 4.15).

Abbildung 4.14.: Die *Object Library* in der rechten unteren Ecke der *Workspace window* bietet dem Programmierer eine Auswahl aller grafischen Elemente im *Cocoa Touch Framework*. Durch *Drag&Drop* kann ein Element in den *Interface Builder* gezogen werden.

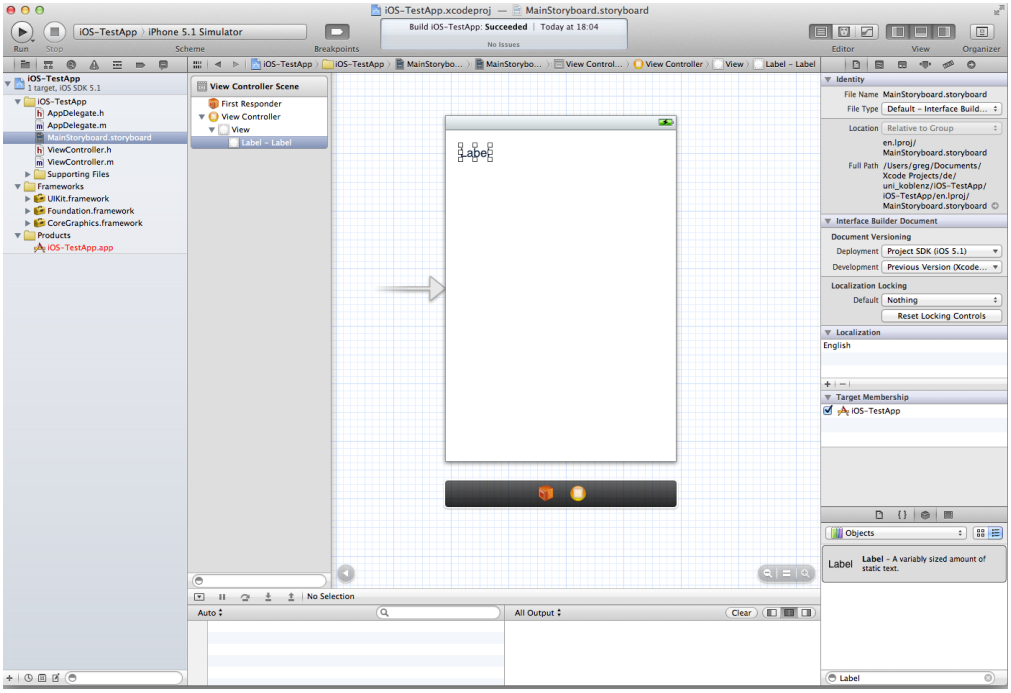

Abbildung 4.15.: Einfügen des ersten grafischen Elements.

Bei der Ausrichtung von grafischen Elementen helfen blau gestrichelte Linien wenn zwei oder mehr Elemente aneinander geführt werden. Dies kann beobachtet werden wenn das eben eingeführte *Label* in die obere linke Ecke der *View* gezogen wird. Dabei rastet das Element automatisch ab einer gewissen Entfernung zum Rand der *View* ein. Erst durch vehementes Bewegen in Richtung des Rands, lässt es sich näher daran platzieren. Dieses Verhalten geht auf die Förde-

rung der Einhaltung der, in den *Apple* „*iOS Human Interface Guidelines*" beschriebenen, Anforderungen an das Layout grafischer Oberflächen zurück (siehe [App12a]). Diese Richtlinien sollen *iOS*-Programme und ihr Bedienungskonzept einheitlich aussehen lassen und zu einer besseren Bedienbarkeit sowie einer damit einhergehenden, höheren Akzeptanz durch die Benutzer führen.

Nach der Positionierung des Elements soll dessen Textinhalt von „*Label*" in „*Text:*" geändert werden. Hierfür muss es doppelt mit der linken Maustaste angeklickt und dann der neue Bezeichner eingegeben werden (siehe Abb. 4.16).

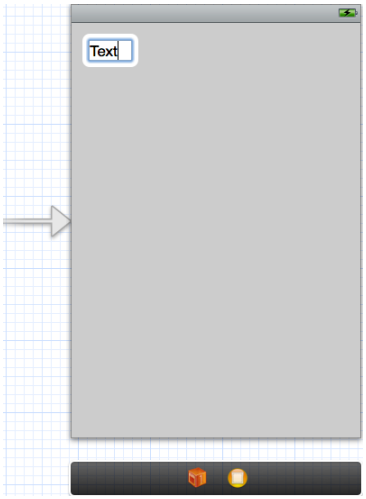

Abbildung 4.16.: Modifikation des Textinhalts eines *Labels*.

Als nächstes sollen unter dem ersten *Label* noch zwei weitere *Labels* eingefügt werden. Anstatt das gleiche Element erneut aus der *Object Library* in die Szene einfügen zu müssen, kann ein Element auch einfach dupliziert werden. Durch Anwendung einer *ALT-Drag&Drop*-Aktion[9] kann das erste *Label* unter Wahrung aller vom Programmierer vorgenommener Einstellungen kopiert werden. Im aktuellen Fall also inklusive der vorgenommenen Textänderung. Nachdem zwei weitere Labels angelegt und ihr Textinhalt in „*Number:*" bzw. „*Switch:*" geändert wurde, ergibt sich ein Bild wie in Abb. 4.17.

[9]Bei gedrückter *ALT*-Taste wird die linke Maustaste über dem zu kopierenden Element gedrückt. Bei Bewegung des Mauszeigers erscheint unter diesem eine Kopie des Elements.

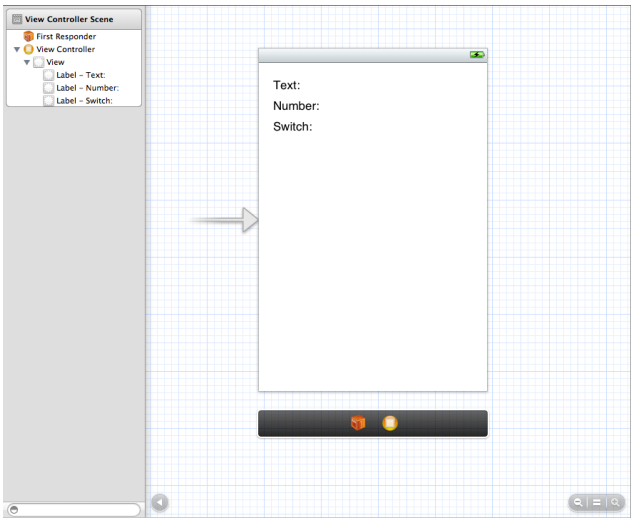

Abbildung 4.17.: Die Anwendung mit zwei weiteren *Labels*.

Für die aktive Interaktion des Benutzers mit der Beispielanwendung sollen zwei Textfelder hinzugefügt werden. Hierzu wird ein „*Text Field*" genanntes Element aus der *Object Library* auf der Höhe des *Labels* mit dem Textinhalt „*Text:*" und ein weiteres auf der Höhe des „*Number:*"-*Labels* platziert. Hierbei sei an die Möglichkeit erinnert, ein Element mit *ALT-Drag&Drop* zu duplizieren. Bei der Ausrichtung der Textfelder helfen erneut die blauen Linien die beim aneinander Annähern von Elementen eingeblendet werden.

Als nächstes sollen die Textfelder bezüglich ihrer grafischen Eigenschaften verändert werden. Um dem Anwender etwas mehr Platz für die Eingabe seines Textes zur Laufzeit zu geben, werden die Textfelder durch Anklicken und Ziehen an den Rändern vergrößert. Erweiterte Einstellungen können über den kontextsensitiven „*Attributes Inspector*" erfolgen, welcher sich zur Beeinflussung des Aussehens jedes beliebigen grafischen Elements einer Anwendung nutzen lässt. Dieser befindet sich auf der rechten Seite der *Workspace window*, im oberen Teil der *Utility bar* und verbirgt sich unter dem vierten Symbol von links in der oberen Leiste – das Symbol ähnelt einem Schieberegler (siehe Abb. 4.18). Je nach ausgewähltem Element stehen andere Einstellungsmöglichkeiten zur Verfügung. Jede aufgeführte Eigenschaft weist eine entsprechende *Objective-C-Property* (siehe Unterkapitel 3.2.3.5) in der Klasse auf, welche das Aussehen des *GUI*-Elements hinter den Kulissen repräsentiert. Dem Programmierer steht es frei Änderungen am Aussehen eines Elements über den *Attributes Inspector* oder im Code vorzunehmen – beide Methoden können auch kombiniert werden. Bspw. kann das allgemeine Aussehen einer Anwendung im *Interface Builder* festgelegt und im Code Anpassungen programmiert werden, welche zur Laufzeit das Aussehen einzelner Elemente anpasst, etwa als Reaktion auf bestimmte Ereignisse.

Für die Beispielanwendung wird dem ersten Textfeld über den *Attributes Inspector* der Text „*Enter text here*" als Platzhalter hinzugefügt und die Schriftfarbe für Eingaben durch den Anwender auf Blau umgestellt. Die Eigenschaften des zweiten Textfeldes bleiben vorerst unverändert.

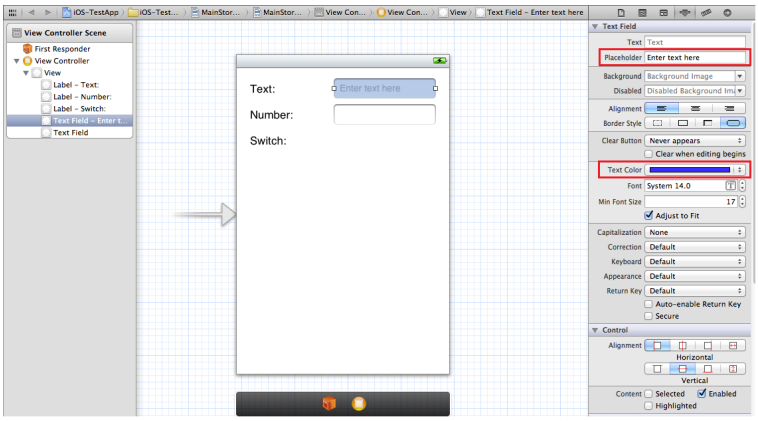

Abbildung 4.18.: Erweiterte Einstellungen der grafischen Merkmale eines *GUI*-Elements können im *Attributes Inspector* erfolgen.

Als nächstes *GUI*-Element wird ein Schalter eingefügt, der dem Anwender das Umschalten zwischen den Werten *On* und *Off* erlaubt und damit zur Abfrage von Wahrheitswerten prädestiniert ist. Zu finden ist das Element unter dem Namen „*Switch*" in der *Object Library*.

Als letztes soll ein Knopf eingefügt werden, welcher zur Laufzeit beim Drücken einen Dialog anzeigen wird. Das benötigte Element firmiert in der *Object Library* unter dem Namen „*Round Rect Button*". Nach dem Platzieren des Knopfes wird sein Text nach einem Doppelklick auf dem Element in „*Press me*" geändert. Das bisherige Ergebnis zeigt Abb. 4.19.

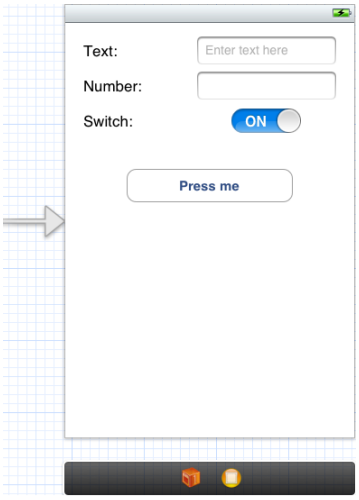

Abbildung 4.19.: Die aus grafischer Sicht komplette Beispielanwendung nach der ersten Lektion.

Mit dem zuletzt eingefügten Knopf wurden allen grafischen Elemente dieser Lektion eingeführt. Auch aus grafischer Sicht ist die erste Szene der Beispielanwendung komplett. Um sich das Ergebnis anzusehen wird in der *Xcode Toolbar* im Dropdown-Menü der Eintrag „*iOS Simulator*" ausge-

wählt und der Simulator mit einem Klick auf den „*Run*"-Knopf an der linken Seite gestartet. Das Ergebnis im Simulator ist Abb. 4.20 zu entnehmen.

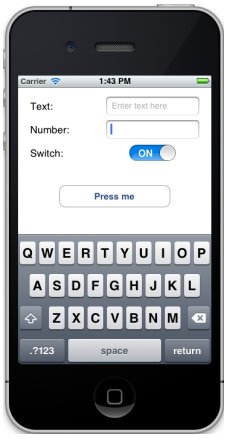

Abbildung 4.20.: Der *iOS Simulator* mit den bisher vorgestellten grafischen Elementen.

Wird das zweite Textfeld der Anwendung im Simulator ausgewählt, welches eigentlich durch den daneben platzierten „Number:"-Bezeichner zur Eingabe einer Zahl animiert, fällt bezüglich der aufgeklappten virtuellen Tastatur auf, dass diese nicht wie aus anderen Anwendungen gewohnt, die Eingabe auf numerische Zeichen begrenzt. Um das Verhalten zu korrigieren genügt es im *Properties Editor* unter der Eigenschaft „*Keyboard*" den Tastaturtyps auf „*Number Pad*" umzustellen. Das neue Ergebnis im Simulator stellt Abb. 4.21 dar.

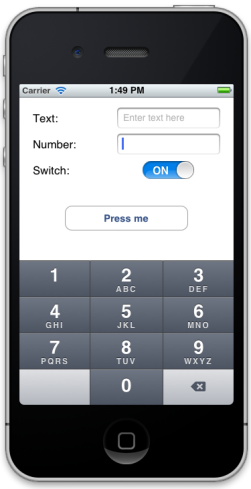

Abbildung 4.21.: Der Tastaturtyp „*Number Pad*" liefert das aus anderen Anwendungen für die Eingabe von numerischen Werten gewohnte Nummernfeld statt einer alphanumerischen Tastatur mit Sonderzeichen.

Alle bisher beschriebenen Vorgänge bezogen sich auf den rein grafischen Teil der Anwendung,

also auf Klassen/ Elemente der *View*-Kategorie im *Model-View-Controller*-Konzept (vgl. 4.2.1.1). Im nächsten Schritt wird der Beispielanwendung Programmlogik in Form eines *Controller*-Objekts und darin beschriebener Methoden hinzugefügt. Ein solches Objekt muss nicht zwangsläufig manuell erstellt werden, so erstellen manche Projektvorlagen wie die für die Beispielanwendung gewählte Vorlage „*Single View Application*" eins oder mehrere *solcher Controller*-Objekte beim Anlegen des Projekts. Das mit erstellte *Controller*-Objekt wird durch die Klasse `ViewController` in Form der Dateien `ViewController.h` und `ViewController.m` beschrieben.

Klickt ein Benutzer den einzigen in der Anwendung vorhandenen Knopf soll ihm ein Hinweisdialog mit den Inhalten und Werten der zwei Textfeldern und des Schalters präsentiert werden. Um das zu realisieren muss aus dem Code heraus auf die betroffenen Elemente zugegriffen werden können. Dazu bietet *Xcode* das Konzept der ***IBOutlets*** an – *Interface Builder Outlet*. *IBOutlets* lassen sich auf mehrere Art und Weisen erstellen. Eine einfache Möglichkeit ist es die „*MainStoryboard.storyboard*"-Datei der Beispielanwendung im *Interface Builder* zu öffnen[10]. Anschließend wird in der *Xcode Toolbar* der Assistenzeditor aktiviert – das zweite Symbol der ersten Gruppierung auf der rechten *Toolbar*-Seite. Der Assistenzeditor zeigt den Code des *Controllers* der einzigen im Beispielprogramm enthaltenen Szene an, die Deklaration der `ViewController`-Klasse (siehe Abb. 4.22).

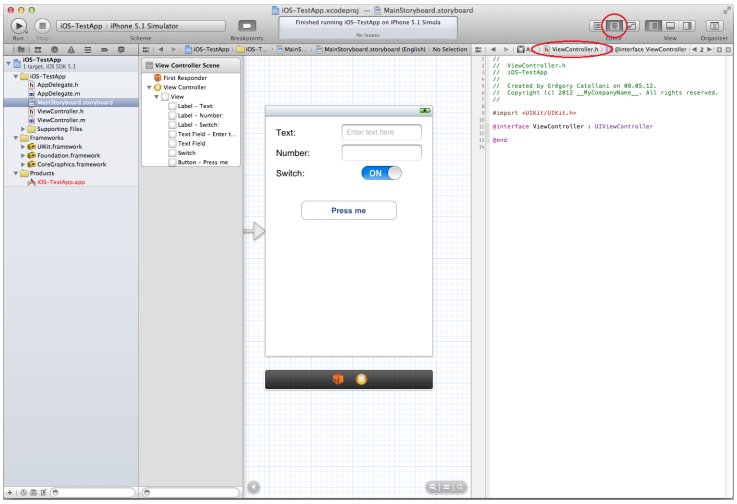

Abbildung 4.22.: Vorbereitungen zum einfachen Erstellung von *IBOutlets*. Neben der grafischen Datei *MainStoryboard* welche auf der linken Seite im *Interface Builder* angezeigt wird, wird der Assistenzeditor dargestellt. In der Beispielanwendung, welche auf der „*Single View Application*"-Vorlage beruht, wird hier die Deklaration der Klasse `ViewController` angezeigt.

Um ein *IBOutlet* zu erstellen wird das erste Textfeld markiert und der Mauszeiger innerhalb des Textfeldes positioniert. Durch Drücken sowie Festhalten der Steuerungstaste (engl. *CTRL*) bei gleichzeitig gedrückter linker Maustaste erscheint beim Bewegen der Maus eine blaue Linie, welche sich vom Textfeld aus erstreckt. Wird diese Linie in den Codeeditor geführt, in den Bereich zwischen

[10]Dieses Verfahren funktioniert auch bei beliebig anderen grafischen Dateien, auch in anderen *Xcode*-Projekten.

den Anweisungen `@interface` und `@end`, erscheint der Schriftzug „*Insert Outlet, Action, or Outlet Collection*" (siehe Abb. 4.23). Lässt der Programmierer an dieser Stelle die Maustaste los, erscheint ein Dialog in dem zuerst die Verbindungsart zwischen grafischem Element und Codestelle angegeben werden kann (siehe Abb. 4.24). Um auf das Textfeld und dessen Inhalt im Code zugreifen zu können, wird eine Verbindung vom Typ „*Outlet*" benötigt. Als Namen wird „*aTextField*" eingegeben. Die restlichen Felder des Dialogs geben den Typ des grafischen Elements, also dessen Klasse, sowie die Art wie die Referenz auf das Objekt gespeichert werden soll, an. Für die Beispielanwendung werden nur der Verbindungstyp „*Type*" und der Name im Feld „*Name*" geändert
.

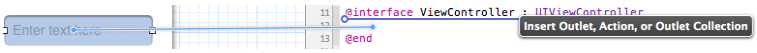

Abbildung 4.23.: Durch ein *CTRL-Drag&Drop* kann zu einem grafischen Element aus dem *Interface Builder* heraus leicht eine Verbindung im Code hergestellt werden.

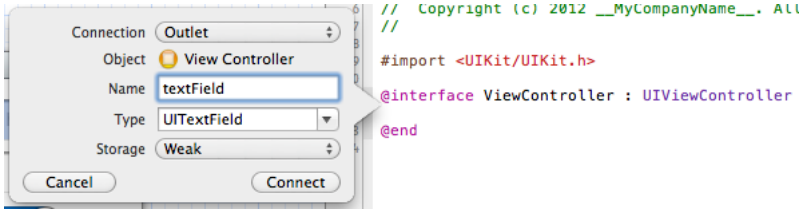

Abbildung 4.24.: Dialog zur Erstellung eines neuen *IBOutlet*.

Nach einem Klick auf „*Connect*" wird in der Deklaration der Klasse `ViewController` eine neue *Property* (siehe Abb. 3.2.3.5) mit dem im *Outlet*-Erstellungsdialog angegebenen Namen `aTextField` erstellt (siehe Abb. 4.25). Das Schlüsselwort *IBOutlet* hinter den *Property*-Attributen und vor dem Typ der *Property* dient *Xcode* als Hinweis dafür, dass diese *Property* mit einem grafischen Element verbunden ist. Zur Programmlaufzeit spielt dieses Schlüsselwort keine Bedeutung und wird daher bei der Kompilierung entfernt.

Abbildung 4.25.: Eine neu erstellte *Property*, welche eine Referenz auf ein im *Interface Builder* definiertes Textfeld darstellt.

Zusätzlich zum *Outlet* für das erste Textfeld werden für das zweite Textfeld und den Schalter auf die selbe Art zwei weitere *Outlets* namens „*aNumberField*" und „*aSwitch*" erstellt (siehe Abb. 4.26).

```
1  //
2  //  ViewController.h
3  //  iOS-TestApp
4  //
5  //  Created by Grégory Catellani on 08.05.12.
6  //  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7  //
8
9  #import <UIKit/UIKit.h>
10
11 @interface ViewController : UIViewController
12 @property (weak, nonatomic) IBOutlet UITextField *aTextField;
13 @property (weak, nonatomic) IBOutlet UITextField *aNumberField;
14 @property (weak, nonatomic) IBOutlet UISwitch *aSwitch;
15
16 @end
17
```

Abbildung 4.26.: Vollständige Liste aller *IBOutlets* welche für den Zweck der Beispielanwendung benötigt werden.

Die Verbindung die ein Outlet zwischen einer *Property* und einem grafischen Element erstellt, lässt sich in der Leiste wo die Zeilennummern im Code-Editor angezeigt werden, erkennen. Ein leerer, runder Kreis weist darauf hin, dass ein *Property* mit dem Schlüsselwort *IBOutlet* existiert, ohne dass eine tatsächliche Verbindung zwischen dieser *Property* und einem grafischen Element der Anwendung existiert. Erst ein, wie ausgefüllter Kreis deutet auf eine funktionierende Verbindung hin (siehe Abb. 4.26).

Eine weitere Stelle um *IBOutlets* für beliebige *GUI*-Elemente zu erstellen, modifizieren und löschen ist der „***Connections inspector***" welcher über die *Utility bar* erreichbar ist – das letzte Symbol in der Leiste das einen nach rechts weisenden Pfeil zeigt (siehe Abb. 4.27). Hier kann ein neues *IBOutlet* in der Kategorie „*Referencing Outlets*" erstellt werden, indem der leere runde Kreis rechts neben dem Schriftzug „*New Referencing Outlet*" angeklickt und bei gedrückter linker Maustaste die nun erscheinende Linie an eine geeignete Stelle im Codeeditor gezogen wird. Es erscheint der bereits bekannte Dialog zu den Eingenschaften des anzulegenden *Outlets* (siehe Abb. 4.23 und 4.24).

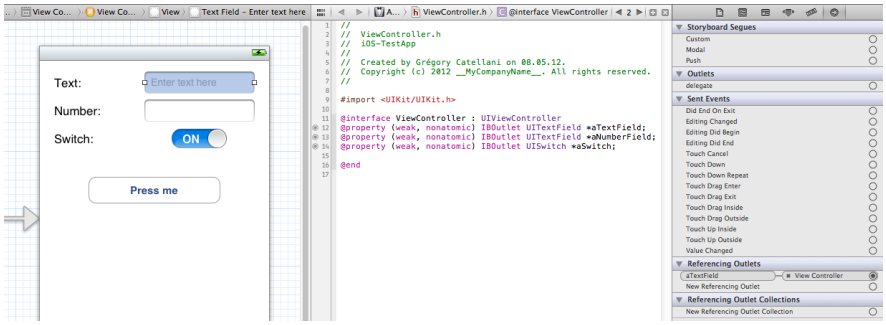

Abbildung 4.27.: Ein Blick in den *Connections Inspector* – rechte Bildseite.

Als nächstes wird eine Methode im Code erstellt werden, welche ausgeführt wird, wenn der Anwender zur Laufzeit den mit „*Press me*"-Knopf drückt. Zur Erleichterung dieses Vorgangs kann wie bei der Erstellung von *IBOutlets*, der Knopf angeklickt und durch *Ctrl-Drag&Drop* in den Code der `ViewController.h`-Datei eine für diesen Zweck ***IBAction*** genannte Verbindung zwischen dem *GUI*-Element und dem Code angelegt werden. Im darauf angezeigten Dialog wird als Ver-

bindungstyp *Action* ausgewählt und als Name „*buttonClicked*" angegeben (siehe Abb. 4.28). Dieser gibt den Namen der Methode vor, die nach dem Klick auf „*Connect*" erstellt wird.

Die Dialog-Eigenschaft *Type* gibt den Typ des *GUI*-Elements (bzw. dessen Klasse) an, welches die Methode auslöst. Die Angabe des Typs wird vom Compiler dazu verwendet zu überprüfen ob ein Objekt das die hier erstellte Methode aufruft auch wirklich von diesem Typ ist – in der Beispielanwendung also, ob es sich beim Methodenaufrufer um ein Objekt der Klasse `UIButton` handelt. Damit generell jedes beliebige Element und Objekt die Methode aufrufen kann wird als Standardwert der dynamische Datentyp `id` angegeben (siehe Unterkapitel 3.2.3.3).

Die Dialog-Eigenschaft *Event* bietet eine Auswahl von Ereignissen bei deren Auftreten zur Laufzeit die hier erstellte Methode aufgerufen werden soll. Zum Beispiel wird eine *Action* mit Ereignistyp „*Touch Up Inside*" ausgelöst, wenn der Anwender den Knopf zur Laufzeit drückt und ihn wieder loslässt während sein Finger sich innerhalb der Abgrenzung des Knopfs befindet. Diese Ereignisart wird von vielen Entwicklern bevorzugt gewählt, weil dem Anwender hierdurch eine Möglichkeit gegeben wird, eine vielleicht aus Versehen getätigte Markierung ohne Konsequenzen abbrechen zu können – dazu muss der Finger außerhalb der Begrenzung des *GUI*-Elements geführt werden.

Die letzte Option im offenen Dialog bestimmt welche Parameter an die zu erstellende Methode übergeben werden sollen. Bei Auswahl des Arguments *Sender* wird der Methode eine Referenz auf das *GUI*-Element zugeführt, das für die Ausführung der *Action* verantwortlich ist. Bei Auswahl des Parameters *Event* wird der Methode angegeben, welches Ereignis bzw. welcher Ereignistyp die Ausführung der *Action* erwirkte. Die letzten beiden Argument sind besonders dann nützlich, wenn eine Methode von mehr als einem *GUI*-Element und/ oder zu mehr als einem Ereignis das auf einem *GUI*-Element ausgeführt wird, aufgerufen wird. Da die Methode im Fall der Beispielanwendung nur durch die Betätigung eines grafischen Elements und durch nur durch die Ereignisart „*Touch Up Inside*" ausgelöst werden soll, wird als Auswahl für die Eigenschaft *Arguments* der Eintrag *None* gewählt.

IBActions lassen sich wie *IBOutlets* zusätzlich zur gerade beschriebenen Methode im *Connections Inspector* anlegen, modifizieren und löschen.

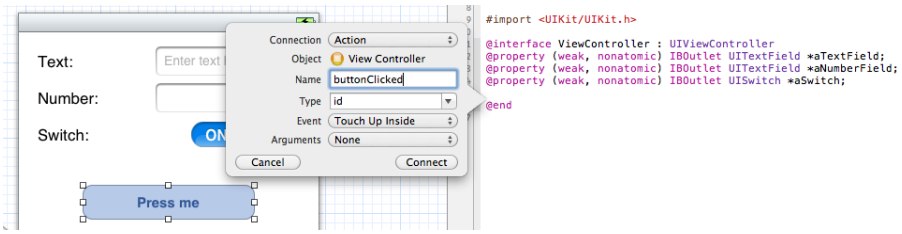

Abbildung 4.28.: Erstellung einer *IBAction*. Einer Verbindung zwischen einem Element der Benutzungsoberfläche und einer Methode, die aufgerufen wird, wenn ein vorgegebenes Ereignis für dieses grafische Komponente eintritt.

Als Ergebnis des Vorgangs zum Anlegen der *IBAction* wird in der Deklaration der `ViewController`-Klasse eine Methodendeklaration `buttonClicked` mit dem speziellen Rückgabetyp *IBAction* erstellt (siehe Abb. 4.29). Hierbei handelt es sich lediglich um ein Alias das für den Datentyp `void`

steht. Es wird benutzt um *Xcode* als Hinweis dafür zu dienen, dass diese Methode durch ein Ereignis in der Benutzungsoberfläche ausgelöst wird.

```
1  //
2  //  ViewController.h
3  //  iOS-TestApp
4  //
5  //  Created by Grégory Catellani on 08.05.12.
6  //  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7  //
8
9  #import <UIKit/UIKit.h>
10
11 @interface ViewController : UIViewController
12 @property (weak, nonatomic) IBOutlet UITextField *aTextField;
13 @property (weak, nonatomic) IBOutlet UITextField *aNumberField;
14 @property (weak, nonatomic) IBOutlet UISwitch *aSwitch;
15
16 - (IBAction)buttonClicked;
17 @end
18
```

Abbildung 4.29.: Die von *Xcode* angelegte Methodendeklaration, deren Implementierung ausgelöst wird wenn der Benutzer den *„Press me"*-Knopf in der Oberfläche drückt.

Zusätzlich zur Methodendeklaration in der Deklaration der `ViewController`-Klasse erstellt *Xcode* beim Anlegen einer *IBAction* auch eine leere Methodenimplementierung in der Klassendefinition. Um sich diese schnell anzeigen zu lassen kann der Eintrag *„ViewController.h"* in der kleinen Leiste (*Jump Bar*) über dem Code-Editor angeklickt und der Eintrag *„ViewController.m"* ausgewählt werden (siehe Abb. 4.30 und Abb. 4.31).

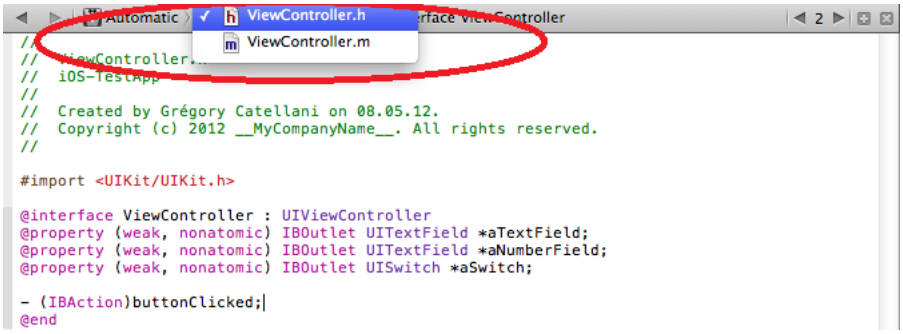

Abbildung 4.30.: Wechsel der Ansicht aus der Deklaration in die Definition der `ViewController`-Klasse über die *Xcode Jump Bar.*

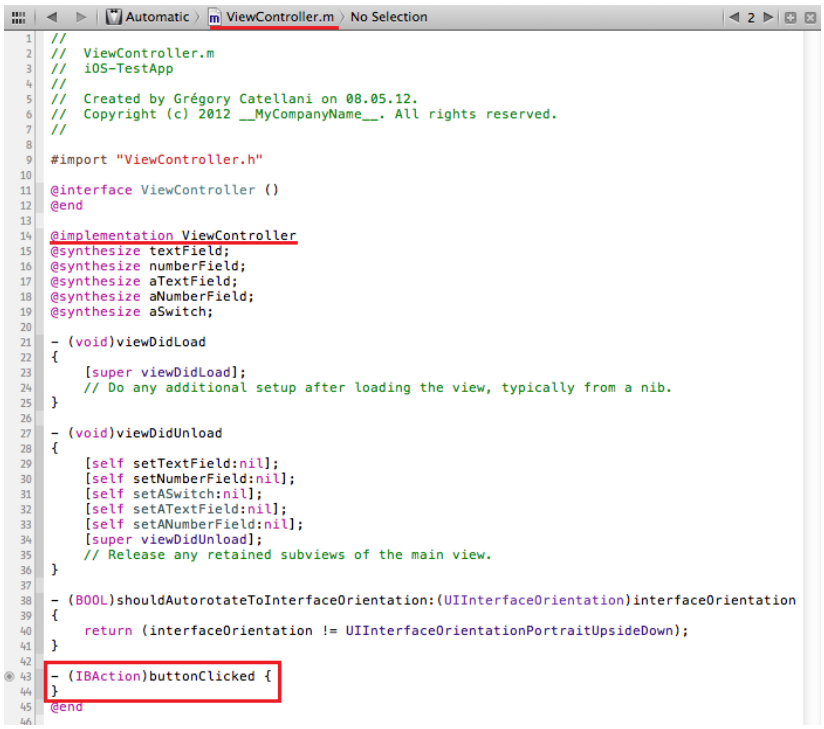

Abbildung 4.31.: Leere Implementierung der Methode `buttonClicked` welche in Folge einer *IBAction* aufgerufen wird, die ausgeführt wird wenn der Nutzer zur Laufzeit den „*Press me*"-Knopf drückt.

In der neuen Methode sollen die Inhalte der Textfelder sowie des Schalters zuerst zur leichteren Referenzierung in Variablen zwischengespeichert und dann in einem Dialog ausgegeben werden (siehe Code 4.3). In der zweiten Zeile wird anhand des Schlüsselworts `self` die, mit der hier deklarierten Property `aTextField` verbundene, Akzessor-Methode und dann anschließend, auf dem Rückgabeobjekt, die *Getter*-Methode der *Property text* aufgerufen (siehe Unterkapitel 3.2.3.5 und 3.2.4.2). Hierdurch wird der Inhalt des ersten Textfeldes zum Zeitpunkt zu dem der „Press me"-Knopfe gedrückt wird in der Variable `textFieldContent` gespeichert.

In der dritten Zeile wird der Inhalt des zweiten Textfelds als Zahlenwert gespeichert. Die vierte Zeile speichert den Zustand des Schalters (`UISwitch`) als numerischen Wert. Dabei gibt die *Property on* eines Schalters, die durch die Getter-Methode `isOn` abgefragt wird, einen Wert vom Typ `BOOL` zurück (siehe Unterkapitel 3.2.3.3).

In der fünften Zeile wird eine Zeichenkette[11] erstellt die Informationen über die Inhalte der Textfelder und den Wert des Kippschalters beinhaltet. Hierzu wird der Konstruktor `initWithFormat:` (siehe Unterkapitel 4.2.2) der Klasse `NSString` verwendet. Dieser Konstruktor erwartet eine formatierte Zeichenkette, ähnlich der `printf`-Methode in *C*. Darin sind alle Schlüsselwörter erlaubt die in *C* in Zusammenhang mit der `printf`-Methode erlaubt sind. Zusätzlich kann in *Objective-C* der Platzhalter `%@` angegeben werden, welcher für ein beliebiges *Objective-C*-Objekt steht (vgl.

[11] Eine Zeichenkette wird in *Objective-C* in Hochkommata angegeben und durch ein `@`-Zeichen angeführt.

[App09c], S. 17ff).

In den letzten beiden Zeilen des Methodenrumpfs wird ein neuer Hinweisdialog, ein `UIAlertView`-Objekt, erstellt und dem Nutzer zur Laufzeit angezeigt wenn er den „*Press me*"-Knopf drückt.

```
1 - (IBAction) buttonClicked {
2   NSString *textFieldContent = self.aTextField.text;
3   int numberFieldValue = self.aNumberField.text.integerValue;
4   int switchValue = self.aSwitch.isOn;
5   NSString *alertMsg = [[NSString alloc] initWithFormat:@"Text: %@, Number:%d,
        Switch-value:%d", textFieldContent, numberFieldValue, switchValue];
6   UIAlertView *anAlert = [[UIAlertView alloc] initWithTitle:@"Title" message:
        alertMsg delegate:nil cancelButtonTitle:@"Okay" otherButtonTitles:nil];
7   [anAlert show];
8 }
```

Tabelle 4.3.: Die folgende Methodenimplementierung zeigt dem Benutzer zur Laufzeit einen Hinweisdialog mit den Werten der Textfelder und des Schalters, wenn er den „*Press me*"-Knopf drückt.

Bei der Auswahl von Methoden welche sich auf einem bestimmten Objekt aufrufen lassen, hilft die Autovervollständigung. Sie lässt sich durch Drücken der Tasten Steuerung&Leertaste aufrufen (siehe Abb. 4.32).

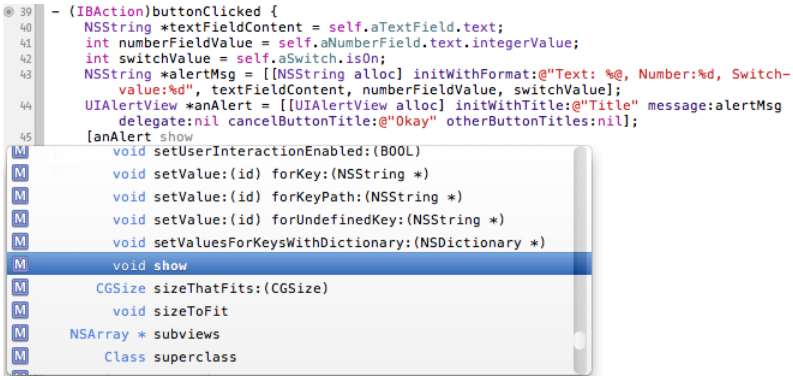

Abbildung 4.32.: Nutzung der Autovervollständigung welche sich mit der Tastenkombination Steuerung&Leertaste aufrufen lässt.

Wird der Mauszeiger über einen Klassenzeichner im Code positioniert und werden anschließend die *ALT*-Taste und die linke Maustaste getätigt, blendet *Xcode* passende Informationen zum Klassenbezeichner ein (siehe Abb. 4.33). Noch detailliertere Informationen zu einem Objekt können angezeigt werden, wenn bei gedrückter *ALT*-Taste ein doppelter Linksklick auf einen Klassenbezeichner ausgeführt wird. Hierzu öffnet *Xcode* den *Organizer* in der Dokumentationsansicht und zeigt alle Ergebnisse an, die eine Recherche des Namens des Klassenbezeichners in allen heruntergeladenen *API*-Dokumentationen ergibt (siehe Abb. 4.34).

Abbildung 4.33.: Aufruf einer kurzen, abstrakten Hilfeeinblendung zum markierten Objekt.

Abbildung 4.34.: Detaillierte Hilfe zu einem Objekt im *Organizer*.

Nachdem die Methode implementiert ist, ist die Beispielanwendung dieser ersten Lektion sowohl grafisch wie programmatisch vollständig. Um den Funktionsumfang zu testen, wird der *iOS Simulator* mit einem Klick auf „*Run*" in der *Xcode Toolbar* gestartet. Wird der Text „Test" ins erste und die Zahl „123" in das zweite Textfeld eingegeben, der Kippschalter auf „On" gestellt und der „Press me"-Knopf gedrückt erscheint der selbst programmierte Hinweisdialog (siehe Abb. 4.35)

Abbildung 4.35.: Das Ergebnis der ersten Lektion im *iOS Simulator*.

4.3.2. Zweiter Schritt: Tab Bar Controller

4.3.2.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- die Erweiterung des *Storyboards* aus dem letzten Unterkapitel um einen *Tab Bar Controller*,
- die Einführung des *Identity Inspector*, und
- die programmatische Erstellung des grafischen Inhalts einer weiteren Anwendungsszene.

4.3.2.2. Implementierung

Die geringe Bildfläche aktueller *iOS*-Geräte stellt andere Ansprüche an die Gestaltung von Bedienungsoberflächen als sie aus der Anwendungsprogrammierung für Betriebssysteme auf Laptops und Desktop-Rechner bekannt sind. Daher muss der Programmierer bei der Identifikation der Daten die dem Anwender zu präsentieren sind sowie bei deren sinnvollen Gruppierung und Verteilung auf der geringen Schirmfläche Sorgfalt tragen. Reicht die Fläche trotz Einsparungen nicht, müssen Konzepte elaboriert werden um weitere Informationen leicht zugänglich zu machen und die Möglichkeit zum Szenenwechsel, zur Darstellung weiterer Inhalte, möglichst platzschonend zu gestalten.

Zu diesem Zweck bietet *Apple* im *iOS-SDK* einige Ansätze zur leichten, übersichtlichen und platzsparenden Navigation durch Anwendungen an. Neben Tabellen und dem damit eng verbundenen Konzept der *Navigation Controller* (siehe Unterkapitel 4.3.3) hält das *SDK* auch das *Tab Bar Controller* Konzept bereit. ***Tab Bar Controller*** verbinden mehrere unabhängige Szenen, die zur Darstellung zumeist lose gekoppelter oder gänzlich voneinander unabhängiger Daten genutzt werden. Der Wechsel zwischen den Szenen erfolgt dabei über eine kleine schwarze Toolbar am unteren Rand der Anwendung, die ***Tab Bar***.

Als Ausgangssituation für die Einführung der hier vorzustellenden Funktionalität wird die in Unterkapitel 4.3.1 erstellte Beispielanwendung verwendet und ergänzt[12]. Um einen *Tab Bar Controller* in das Projekt einzufügen wird die Datei *MainStoryboard.storyboard* im *Xcode Interface Builder* geöffnet. Anschließend wird ein „*Tab Bar Controller*"-Objekt aus der *Object Library* per *Drag&Drop* in das *Storyboard* eingefügt . Im *Interface Builder* sind jetzt drei neue Objekte neben dem, bereits aus dem letzten Kapitel bekannten, „*View Controller*"-Objekt zu sehen: ein *Tab Bar Controller* und zwei neue Szenen (siehe Abb. 4.36).

[12] Der Stand der Anwendung nach Unterkapitel 4.3.1 befindet sich in Form eines *Xcode*-Projekts auf dem, dieser Ausarbeitung beiliegenden, Datenträger (siehe Appendix VI).

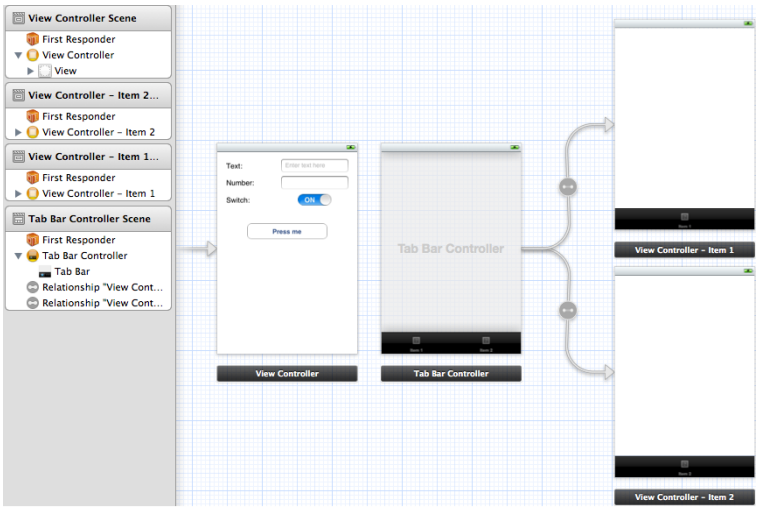

Abbildung 4.36.: Ansicht der *MainStoryboard.storyboard*-Datei nach Einfügen eines *Tab Bar Controller*.

Die transparente Fläche innerhalb des *„Tab Bar Controller"*-Objekts weist auf den Umstand hin, dass der *Controller* nicht direkt eine *View* enthält. Dieser Umstand spiegelt sich auch in der *„Interface Builder"*-Leiste auf der linken Seite wieder. Deutlich macht dies auch ein Klick auf den Pfeil neben dem *„Tab Bar Controller"*-Objekt der *„Tab Bar Controller Scene"* genannten Szene in der *„Interface Builder"*-Leiste. Der *Tab Bar Controller* enthält nur eine *Tab Bar*.

Die *Tab Bar* wird durch eine schwarze Fläche am unteren Rand der Szene repräsentiert. Sie enthält einzelne, ***Tab Bar Items*** (`UITabBarItem`) genannte, Elemente, die den Aufruf einer anderen Szene bewirken wenn sie zur Laufzeit angeklickt werden.

Bei der Erstellung eines *Tab Bar Controllers* über die *Object Library* werden zwei Szenen miterstellt, zu denen der Nutzer über das entsprechende *Tab Bar Item* wechseln kann. Bis auf ein Ebenbild der *Tool Bar* enthalten sie zum jetzigen Zeitpunkt keine weiteren grafischen Elemente.

Die Verbindung zwischen den neuen Szenen und dem *Tab Bar Controller*, welcher die Logik zum Wechsel zwischen den Szenen enthält, wird durch Pfeile zwischen den Elementen verdeutlicht. Bei diesen Verbindungselementen handelt es sich um *Segue*-Objekte. Ein solches Objekt wurde bereits im vorherigen Kapitel eingesetzt um die Startszene der Beispielanwendung festzulegen (siehe Unterkapitel 4.3.1). Der Name ***Segue*** entstammt dem Italienischen, bedeutet soviel wie Transition und wird auf Englisch *„seg-way"* ausgesprochen (vgl. [Dav11], S. 368). *Segues* werden in *Storyboards* eingesetzt um Übergänge zwischen verschiedenen Szenen der Anwendung grafisch im *Interface Builder* zu verdeutlichen und programmatisch festzulegen (siehe Unterkapitel 4.3.3). *Segues* eignen sich auch um Daten zwischen verbundenen Szenen auszutauschen (siehe Unterkapitel 4.3.4).

Würde die Anwendung jetzt kompiliert und ausgeführt werden, würde weiterhin nur die Szene aus dem vorherigen Kapitel angezeigt werden (siehe Unterkapitel 4.3.1). Damit diese Szene aber in Kombination mit dem neu eingeführten *Tab Bar Controller* aufgeführt werden kann, wird eine der neuen Szenen, welche bei der Anlegung des *Tab Bar Controller* miterstellt wurde, durch die Szene aus dem vorherigen Kapitel ersetzt. Hierzu wird die obere der beiden neuen Szene, mit dem

Tool Bar Bezeichner „*Item 1*", durch einen Klick auf den schwarzen Balken unter der Szene markiert – die komplette Szene und der schwarze Balken werden blau umrahmt. Nach Betätigung der Entfernen-Taste, wird die Szene aus dem letzten Kapitel an die Stelle der gerade gelöschten Szene gerückt. Anschließend wird der *Tab Bar Controller* markiert. Durch Drücken der linken Maustaste während sich der Mauszeiger innerhalb der Fläche des *Tab Bar Controller* befindet und gleichzeitiges betätigen der Steuerung-Taste wird eine blaue Verbindungslinie sichtbar, wie sie bereits aus dem letzten Kapitel für die Erstellung von *IBOutlets* und *IBActions* bekannt ist. Mit der Linie wird der Tab Bar Controller und die Szene aus dem vorherigen Kapitel verbunden. Ist die Verbindung hergestellt erscheint ein Auswahldialog in dem die *Segue*-Art festgelegt werden kann (siehe Abb. 4.37).

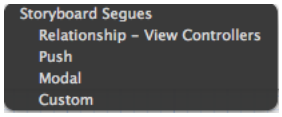

Abbildung 4.37.: Auswahlmenü für eine Verbindung zwischen einem Tab Bar Controller und einem View Controller.

Durch die Auswahl der Option „*Relationship - View Controllers*" wird dem *Tab Bar Controller* aufgetragen einen neuen Eintrag, in der durch ihn verwalteten *Tab Bar,* und geeignete Codelogik, zum Wechsel in die neue, nunmehr mit ihm verbundene Szene, anzulegen. Als grafische Rückmeldung auf die erfolgreiche Erstellung der Verbindung erscheint ein neues *Segue*-Objekt zwischen dem *Tab Bar Controller* und der Szene. Zudem erhält die *Tab Bar* einen neuen „*Item*" genannten Eintrag (siehe Abb. 4.38).

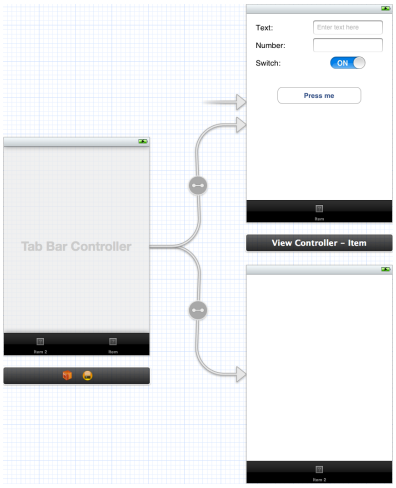

Abbildung 4.38.: Ersetzung einer Szene die bei der Erstellung des „Tool Bar Controller"-Objekts automatisch mit erstellt wurde, durch die Szene aus dem letzten Unterkapitel.

Damit der *Tab Bar Controller* und die damit verbundene Funktionalität, zum Wechseln der Szene, zur Laufzeit sicht- und benutzbar ist, muss das *Segue*, in Form des kleinen Pfeils welches aus

dem Nichts zu kommen scheint und in der linken Seite der alten Szene mündet, mit der linken Maustaste an das „*Tab Bar Controller*"-Objekt geheftet werden (siehe Abb. 4.39).

Zuletzt sollen zwei Änderungen kosmetischer Natur durchgeführt werden. Durch die Erste soll dafür Sorge getragen werden, dass die Szene aus dem letzten Kapitel über den ersten Eintrag der *Tab Bar* erreichbar ist statt über den Zweiten – dies ist dem Umstand zuzusprechen, dass beim Anlegen einer neuen Verbindung zwischen einem *Tab Bar Controller* und einer noch nicht mit diesem verbundenen Szene, ein neues *Tab Bar Item* an letzter Stelle in der *Tab Bar* angelegt wird. Diese Änderung lässt sich durch Beeinflussung der Anordnung der *Tab Bar Items* im *Tab Bar Controller* per *Drag&Drop* leicht bewerkstelligen (siehe Abb. 4.39).

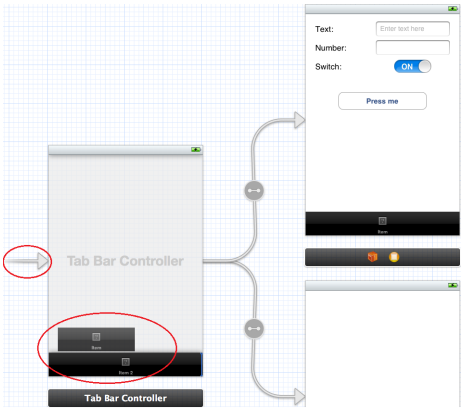

Abbildung 4.39.: Manuelle Festlegung des Tab Bar Controllers als Startszene. Anschließende Neuanordnung der *Tab Bar Items*.

Die zweite kosmetische Änderung betrifft die Titel der *Tab Bar Items* in der *Tab Bar*. Hierzu werden die Bezeichner in der *Tool Bar* der jeweiligen Szenen doppelt mit der linken Maustaste angeklickt und ein neuer Titel eingegeben. Für die Szene aus dem letzten Kapitel wird „*Lesson 1*" und für die neue, untere Szene „*Lesson* 2" als Titel angegeben (siehe Abb. 4.40).

Abbildung 4.40.: Umbenennung eines *Tab Bar Items* nach einem Doppelklick auf dessen Bezeichner.

Wird die Anwendung jetzt kompiliert erscheint die *Tab Bar* mit den umsortierten und umbenannten Einträgen am unteren Rand. Im oberen Bereich wird die Szene angezeigt, welche mit dem ersten *Tab Bar Item* des *Tab Bar Controller* in Verbindung steht – in der Beispielanwendung die im letzten Kapitel erstellte Szene mit den Bezeichnern, Textfeldern und dem Knopf. Ein Klick auf den zweiten Eintrag offenbart eine bislang leere zweite Szene (siehe Abb. 4.41).

Abbildung 4.41.: Ein *Tool Bar Controller* in Aktion.

Der grafische Inhalt der zweiten Szene soll programmatisch erstellt werden. Hierzu müssen entsprechende Anweisungen in einem, diese Szene verwaltenden, „*View Controller*"-Objekt eingefügt werden. Für jedes grafische Element das in *Xcode* angelegt wird, wird durch *Xcode* automatisch ein *Controller*-Objekt angelegt. Der Typ der Klasse welche die Programmlogik des *Controller*-Objekts beinhaltet, orientiert sich dabei am Typ des grafischen Elements das über die *Library pane* angelegt wurde. Um die Klassentypangabe eines *Controller*-Objekts einzusehen und zu ändern wird ein grafisches Element markiert und der „***Identity inspector***" aufgerufen – das dritte Icon im oberen Bereich der *Utility bar*. Als Klassentyp des *Controllers*, der die Szene aus dem letzten Kapitel verwaltet, wird in der Kategorie „*Custom Class*" hinter dem Bezeichner *Class* der Eintrag `UIViewController` angezeigt (siehe Abb. 4.42) – der Eintrag wird angezeigt wenn die vollständige Szene über den schwarzen Balken markiert wurde.

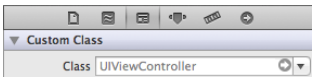

Abbildung 4.42.: Der *Identity Inspector* gibt Aufschluss über die Klasse, welche die Programmlogik eines grafischen Elements enthält.

Eine Klasse oder Unterklasse vom Typ `UIViewController` entstammt dem *iOS-SDK*. Sie stellt

Programmlogik bereit, welche an das grafische Element mit dem der Controller dieses Klassentyps verbunden ist, gebunden ist. Der Inhalt dieser Klassen kann aber nur schlecht durch den Programmierer um weitere Programmlogik ergänzt werden. Da der Inhalt der zweiten Szene programmatisch erstellt werden soll, muss dementsprechend eine andere Klasse für dessen *Controller* festgelegt werden.

Da keine geeignete Klasse existiert wird zuerst eine neue Klasse angelegt. Dies kann durch einen rechten Mausklick auf die „*iOS-TestApp*"-Gruppe im *Project navigator* und die anschließende Auswahl des Eintrags „*New File...*" erfolgen. Da ein *Controller*-Objekt Programmlogik in *Objective-C* enthält, wird die Vorlage „*Objective-C class*", in der Unterkategorie „*Cocoa Touch*" der Kategorie „*iOS*" im sich öffnenden Dialog ausgesucht (siehe Abb. 4.43, linkes Teilbild). Nach einem Klick auf „*Next*" wird als Name der Klasse „*ViewController2*" eingegeben, als Oberklasse `UIViewController` ausgesucht und die Haken in den darunterliegenden Feldern entfernt (siehe Abb. 4.43, rechtes Teilbild). Abschließend wird der Speicherort für die Klassendeklaration und -definition festgelegt.

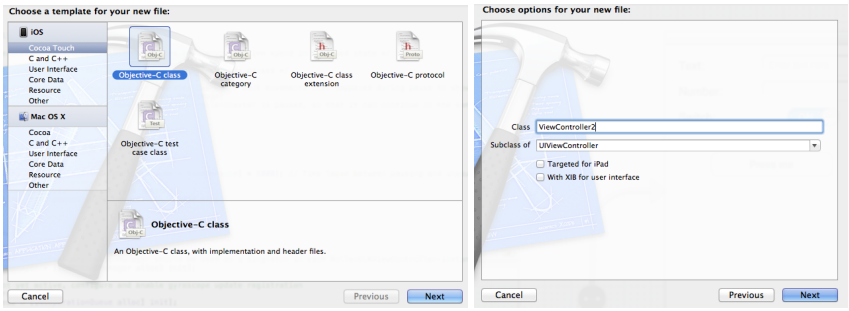

Abbildung 4.43.: Erstellung einer neuen Klasse, welche die Programmlogik und die Beschreibung der grafischen Elemente der zweiten Szene enthalten wird.

Im Anschluss an die Erstellung der Klasse wird die leere Szene im Interface Builder erneut markiert, der I*dentity Inspector* aufgerufen und hinter *Class* der Eintrag „*ViewController2*" ausgewählt. Damit ist die Verbindung zwischen dem „*View Controller*"-Objekt dieser Szene und der neuen *Objective-C*-Klasse „*ViewController2*" hergestellt.

Als nächstes wird Code für die Anzeige eines einfachen *Labels* in dieser Szene geschrieben. Zu diesem Zweck wird im *Project navigator* die Implementierung der Klasse `ViewController2` aufgerufen – die Datei `ViewController2.m`. Da es sich bei jedem grafischen Element um eine Unterklasse der Klasse `UIView` handelt und grafische Elemente ineinander verschachtelt am Bildschirm angezeigt werden (siehe Unterkapitel 4.2.1 und Abb. 4.4), muss zuerst festgestellt werden wo das neu zu erstellende *Label*, ein Objekt vom Typ `UILabel`, in der Sichthierarchie eingefügt werden muss damit es zur Laufzeit angezeigt wird. Da die Szene bis auf dieses *Label* keine weiteren Elemente enthalten wird und das *Label* somit auch nicht in ein weiteres Element verschaltet werden kann, wird es direkt in das unterste *Container*-Element der Szene eingebettet, die *View*. Auf die *View* einer Szene kann im Code einer Klasse oder Unterklasse vom Typ `UIViewController` über eine *Property* namens **`view`** zugegriffen werden.

Anschließend muss die Stelle im Code ausfindig gemacht werden an der die Anweisungen stehen müssen, welche Einfluss auf den Inhalt der *View* nehmen werden. Die Oberklasse `UIViewControl-`

ler der selbst erstellten Klasse ViewController2 stellt u.a. zu diesem Zweck spezielle Methoden bereit. Diese Methoden ermöglichen es einem Anwendungsentwickler detaillierten Einfluss, auf jeden Zeitpunkt in der eine Szene am Bildschirm dargestellt wird, zu nehmen. Hierzu gehören unter anderem die folgenden Methoden: (vgl. [App12c])

- Die Methode **viewDidLoad** wird aufgerufen, wenn das UIViewController-Instanzobjekt das sie enthält, in den Speicher geladen wird. Dieser Vorgang kann auch lange bevor die, durch dieses Objekt beschriebene, Szene dargestellt wird, stattfinden. Sie eignet sich u.a. zur Beschreibung grafischer Elementen deren Aussehen nach der Kompilierung nicht mehr sonderlich variiert und zum Aufruf von Funktionalität welche hinter den Kulissen abläuft, wie das Laden von Daten von der Festplatte.
- Die Methode **viewWillAppear:** wird aufgerufen, kurz bevor die, in dieser Klasse beschriebene, Szene auf dem Schirm angezeigt wird. Sie eignet sich u.a. um Änderungen am Aussehen der Szene durchzuführen, kurz bevor sie angezeigt wird – z.Bsp. als Reaktion auf Ereignisse die in anderen Szenen eintraten wie die Modifikation von Datensätzen – oder es kann eine möglicherweise geschlossene Netzwerkverbindungen wieder aufgebaut werden.
- Die Methode **viewDidAppear:** wird aufgerufen wenn die, in dieser Klasse beschriebene, Szene am Schirm erscheint. Sie eignet sich sich u.a. um pausierte *OpenGL-ES*-Animationen wieder aufzunehmen.
- Die Methode **viewWillDissapear:** wird aufgerufen, kurz bevor die, in dieser Klasse beschriebene, Szene vom Bildschirm verschwindet. Sie eignet sich u.a. um Modifikationen an Datensätzen auf der Festplatte zu speichern.
- Die Methode **viewDidDissapear:** wird aufgerufen, nachdem die, in dieser Klasse beschriebene, Szene vom Bildschirm verschwunden ist. Sie eignet sich u.a. dazu eine Netzwerkverbindung zu trennen, welche in der Methode viewWillDissapear: dazu benutzt wurde Daten auf einer Netzwerkfestplatte wie der *Apple iCloud* zu speichern.
- Die Methode **viewWillUnload** wird aufgerufen, kurz bevor die, in dieser Klasse beschriebene, Szene aus dem Speicher entfernt wird. Außerdem kann sie aufgerufen werden, wenn der freie Speicher des Mobilgeräts zu Neige geht. Sie eignet sich u.a. um grafische Elemente der Sicht freizugeben, die nirgends wo anders in der Anwendung mehr benötigt werden.
- Die Methode **viewDidUnload** wird aufgerufen nachdem die, in dieser Klasse beschriebene, Szene aus dem Speicher entfernt wurde. Sie eignet sich u.a. um letzte Aufräumarbeiten durchzuführen bevor das UIViewController-Instanzobjekt, das die Szene verwaltet, aus dem Speicher entfernt wird.

Für den wenig komplexen Zweck ein einfaches Label anzuzeigen, genügt die Ergänzung der bereits in der Implementierung der ViewController2-Klasse enthaltenen Methode viewDidLoad (siehe Codelisting 4.4). In der dritten Zeile der Methodenimplementierung wird ein neues UILabel-Objekt an den Bildschirmkoordinaten (10, 10) mit einer Breite von 100 und einer Höhe von 50 Pixeln erstellt. Dann wird über die text-*Property* der Inhalt des Bezeichners festgelegt. Abschließend wird das neu erstellte Objekt in die es umgebende Sicht eingebettet.

```
1 -(void)viewDidLoad
2 {
3   [super viewDidLoad];
4    // Do any additional setup after loading the view.
5   UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(10, 10, 100, 50)];
6   label.text = @"Lesson 2";
7   [self.view addSubview:label];
8 }
```

Tabelle 4.4.: Ergänzung der Methodenimplementierung von `viewDidLoad` zur Anzeige eines einfachen Bezeichners in der zweiten Szene der Beispielanwendung.

Nach Ausführung der Applikation im *iOS Simulator* und anschließendem Umschalten in die zweite Anwendungsszene präsentiert sich der eben erstellte Schriftzug *„Lesson 2"* (siehe Abb. 4.44).

Abbildung 4.44.: Das Ergebnis der zweiten Lektion im *iOS Simulator*.

4.3.3. Dritter Schritt: Navigation Controller & Table View

4.3.3.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- Erstellung eines *Navigation Controller*,
- Einrichten eines *Table View Controllers* und einer dazugehörigen *Table View*,
- Gestaltung dynamischer *Table View Cells*,
- Erstellung einer Detailansicht zur Editierung von Tabelleneinträgen,
- Benutzung von *Segues* zur Übertragung von Daten aus Tabelleneinträgen in die Detailsicht,
- Programmatisches Befüllen der Tabelle.

4.3.3.2. Implementierung

Da die im Rahmen dieser Lektion durchgeführten Arbeiten aufwendiger sind, werden sie in Einzelschritten erklärt.

Arbeiten auf der grafischer Ebene der Anwendung in der Datei MainStoryboard.storyboard:

Als Ausgangsprojekt für dieses Unterkapitel wird das in den letzten beiden Unterkapiteln benutzte Projekt verwendet (siehe Unterkapitel 4.3.1 und 4.3.2). Nachdem im letzten Unterkapiteln ein *Tab Bar Controller*, zur Verteilung unabhängiger Programmdaten auf mehrere Szene, eingeführt wurde, soll in diesem Kapitel ein Konzept zur Repräsentation miteinander eng gekoppelter, hierarchisch strukturierter Informationen eingeführt werden, der *Navigation Controller*. Ein *Navigation Controller* ermöglicht die Navigation durch eine Hierarchie von Programmszenen, indem es einen Stapel „*View Controller*"-Objekte verwaltet. Wird in einer, von einem *Navigation Controller* verwalteten, Szene eine Transition zu einer weiteren, vom selben *Navigation Controller* verwalteten, Szene durchgeführt, wird das „*View Controller*"-Objekt das die neue Szene verwaltet auf den Stapel des *Navigation Controllers* gelegt – die neue Szene wird am Schirm angezeigt. Für die Navigation zurück, also aus einer aufgerufenen Szene zurück zu der davor, steht die *Navigation Bar* am oberen Rand des Bildschirms zur Verfügung.

Häufig werden *Navigation Controller* in Kombination mit Tabellen verwendet da das Stapel-Konzept eines *Navigation Controllers* sich prima zur Repräsentation von Tabellen und darin enthaltener Informationen eignet. So kann die Ansicht der Tabellenzellen als Ausgangsansicht (engl. *Root View*) des *Navigation Controllers* definiert werden. Durch geeignete Maßnahmen, kann ein Klick auf einen Tabelleneintrag eine Detailansicht, mit den Daten der markierten Zelle, aufrufen. Diese Detailansicht wird auf den Stapel des *Navigation Controllers* gelegt und verdrängt die Repräsentation der Tabelle. Die Rückkehr zur Tabellenansicht aus der Detailansicht ist über einen entsprechenden Knopf in der *Navigation Bar* möglich – intern wird die Detailansicht dabei vom Stapel des *Navigation Controllers* entfernt und die darunterliegende Szene wird angezeigt.

Durch die häufige Verwendung eines *Navigation Controllers* in Kombination mit einer *Table View*, wird beim Einfügen eines *Navigation Controllers* per *Drag&Drop* aus der *Object library* in ein Projekt eine Tabellenansicht miterstellt (siehe Abb. 4.45). Die blaue Leiste am oberen Bildschirmrand stellt die *Navigation Bar* dar. Für jede Szene welche der Verwaltung des zugehörigen *Navigation Controller* unterliegt, kann die *Navigation Bar* einen mittig platzierten, optionalen Bezeichner enthalten. Zudem wird beim Abtauchen in der Sichthierarchie ein Zurück-Knopf auf der linken Seite der *Navigation Bar* angeboten, welcher aber vom Programmierer überschrieben werden kann. Außerdem kann die *Navigation Bar* weitere vom Programmierer hier platzierte Elemente enthalten, z.Bsp. ein Knopf welcher wenn betätigt die in der aktuellen Szene getätigten Eingaben abspeichert.

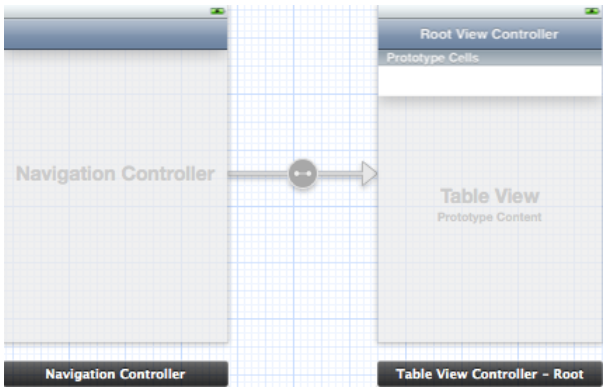

Abbildung 4.45.: Wird ein *Navigation Controller* aus der *Object library* eingefügt, erstellt *Xcode* zugleich einen *Table View Controller*.

Damit der neu eingefügte *Navigation Controller* und die mit erstellte Tabelle über die *Tab Bar* der Beispielanwendung aus erreichbar ist, wird er über ein Segue mit dem *Tab Bar Controller* verbunden. Hierzu muss der *Tab Bar Controller* markiert und per *Ctrl-Drag&Drop* mit dem *Navigation Controller* verbunden werden – im erscheinenden Dialog wird der Eintrag „*Relationshop - View Controllers*" ausgewählt. Anschließend erscheint die *Tab Bar* im unteren Teil der „*Navigation Controller*"-Szene. Der Name des neuen Eintrags in der *Tab Bar* wird durch Doppelklick auf den Schriftzug „*Item*" im *Navigation Controller* in „*Lesson 3+4*" umgeändert.

Als nächstes wird die Tabellenansicht editiert, welche mit „*Root View Controller*" betitelt ist. Dieser Schriftzug kann durch Doppelklick und nachfolgender Eingabe verändert werden – in der Beispielanwendung wird auch an dieser Stelle der Bezeichner „*Lesson 3+4*" gewählt. Wird die Anwendung zum jetzigen Zeitpunkt ausgeführt und in die neue Szene gewechselt, ist bis auf die *Navigation Bar* mit dem neuen Schriftzug „*Lesson 3+4*" noch nichts zu sehen.

Um dies zu ändern muss die Tabelle mit Daten gefüllt und das Tabellenaussehen bestimmt werden. Das Befüllen der Tabelle kann auf zwei Weisen erfolgen:

1. Mit *iOS 5* wurden **statische Tabellenfelder** eingeführt (engl. ***Static cells***). Hierbei handelt es sich um Zellen die vollständig im *Interface Builder* bezüglich ihres Aussehens, ihrer Stelle in der Tabelle und ihres Inhalts beschrieben werden. Statische Zellen werden u.a. dann verwendet, wenn der Inhalt einer Tabelle bereits vor der Laufzeit feststeht und sich während der Programmausführung nicht mehr ändert.

2. Vor *iOS 5* und heute immer noch gebräuchlich sind **dynamische Tabellenfelder**, die durch Zellenprototypen im *Interface Builder* bezüglich ihres Aussehens modelliert und anschließend im Code programmatisch mit Inhalt versehen werden. Dynamische Zellen werden u.a. dann verwendet wenn der Tabelleninhalt erst während der Laufzeit feststeht und sich dieser ändern kann – z.Bsp. dann wenn eine Tabelle Daten repräsentiert, welche bei Programmausführung von der Festplatte eingelesen werden, die zur Laufzeit geändert werden können.

Ziel der Beispielanwendung wird es sein, das Laden von Daten von der Festplatte beim Wechsel in die hier beschrieben Szene, die Modifikation der Daten zur Laufzeit und das abschließende Speichern der Daten bei Verlassen dieser Szene, zu ermöglichen. Für diese Zwecke werden dynamische Tabellenfelder benötigt.

Um Prototypen für dynamische Tabellenfelder zu definieren, kann auf die, bereits mit dem *Navigation Controller* erstellten, Felder im *Table View Controller* zurückgegriffen werden. Bei der Erstellung eines neuen *Table View Controllers*, wird standardmäßig ein leerer Prototyp für eine dynamische Zelle angelegt. Zusätzliche können angelegt werden indem die Fläche mit dem Schriftzug „*Table View Prototype Content*" im *Interface Builder* angeklickt, der *Attributes inspector* aufgerufen und die Anzahl hinter „*Prototype Cells*" erhöht wird. Für die Beispielanwendung wird die Anzahl auf zwei gesteigert. Ein Tabellenfeld soll dazu dienen neue Tabelleneinträge anzulegen, die restlichen Zellen stellen die angelegten Daten dar.

Über die Option ***Style*** kann das generelle Aussehen der Tabelle festgelegt werden. Das *iOS-SDK* stellt zwei zur Verfügung, welche sich programmatisch beliebig erweitern lassen: (siehe Abb. 4.46)

1. Durch Auswahl des Stils ***Plain*** wird eine uniforme Liste erstellt, bei der alle Tabellenfelder direkt oder, nur durch eine Überschrift getrennt, in Gruppen (engl. ***sections***) untereinander aufgeführt sind.
2. Eine Tabelle im ***Grouped-style*** stellt ihre Daten hingegen in visuell voneinander getrennten Gruppen von Zellen dar.

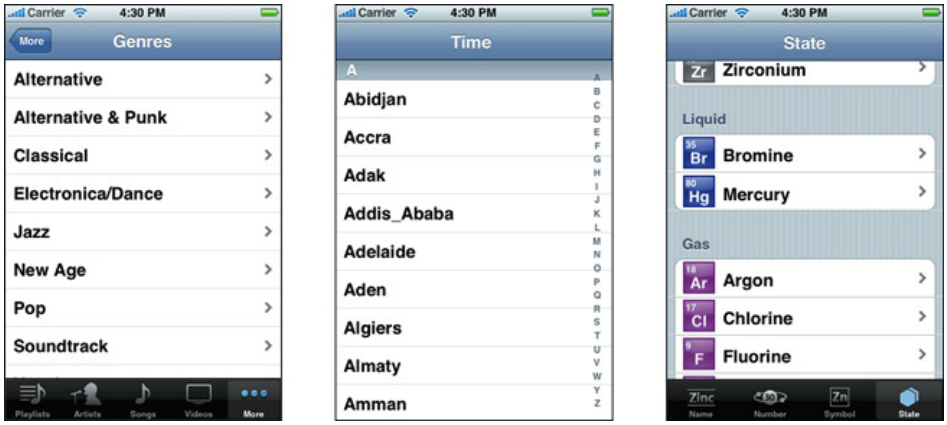

Abbildung 4.46.: Links: *Plain-style* ohne Gruppen. Mitte: *Plain-style* mit Gruppen. Rechts: *Grouped-style* mit mehreren Gruppen. Bildquelle: [App11a], S. 7.

Die hier programmierte Beispielanwendung soll zwei visuell voneinander getrennte Gruppen ent-

halten – daher wird als Tabellenstil *Grouped* festgelegt. Die erste Gruppe wird ein einzelnes Tabellenfeld besitzen, welche einen neuen Eintrag in der Tabelle anlegt. Die zweite Gruppe stellt alle Tabelleneinträge dar, welche über das Feld in der ersten Gruppe angelegten wurden.

Nachdem das generelle Aussehen der Tabelle festgelegt wurde, werden die zwei Zellprototypen entworfen. Zellen lassen sich bezüglich ihres Aussehens und Inhalts, beliebig und sehr detailliert, entweder direkt im *Interface Builder* oder in einer eigenen Klasse im Code, beschreiben. So können Zeilen neben Schriftzügen und Bilder eine nahezu unerschöpfliche Auswahl an grafischen Elemente enthalten.

Für die Beispielanwendung reicht die Bestimmung des Zellenaussehens über den *Interface Builder* und die Wahl eines einfachen Zellenaussehens, das nur Text darstellen kann, völlig aus. Dafür wird die erste Zelle markiert, worauf im *Attributes inspector* diverse Einstellungsoptionen verfügbar werden. Die Option „*Style*" legt das allgemeine Aussehen der Zelle fest. Es stehen folgende Stile zur Auswahl: (siehe Abb. 4.47)

1. Die Auswahl des Stils ***Custom*** erlaubt dem Programmierer das Aussehen eines Tabellenfeldes durch das Einfügen beliebiger grafischer Elemente detailliert zu modellieren.
2. Der Stil ***Basic*** bietet ein weitaus schlichteres Aussehen. Hier kann lediglich ein einzelner Schriftzug in der Zelle angegeben werden.
3. Der Stil ***Right Detail*** bietet zusätzlich zum Schriftzug, aus dem *Basic*-Stil, noch Platz für einen weiteren Bezeichner, der sich zur Angabe zusätzlicher Informationen zum Inhalt der Zelle eignet.
4. Der Stil ***Left Detail*** bietet die selben Elemente wie der Stil *Right Detail*. Sie sind nur anders in der Zelle angeordnet und unterscheiden sich geringfügig bezüglich ihres Aussehens.
5. Die Auswahl des Stils ***Subtitle*** legt zwei Bezeichner untereinander an und suggeriert damit eine Zelle mit mehreren Reihen.

Anzumerken ist, dass der Programmierer die Position und das Aussehen des oder der Bezeichner die mit den vier letzten Stilen angelegt werden, nach Belieben ändern kann. Die vorgegebenen Stile dienen nur als Schablone und bilden oft benötigte Standardrepräsentationen ab.

Abbildung 4.47.: Vier von *Apple* vorgegebene Zellenstile. In absteigender Reihenfolge: *Basic*, *Right Detail*, *Left Detail* und *Subtitle*.

Der erste Zellprototyp der Beispielanwendung erhält den *Basic*-Stil. Der darauf im Tabellenfeld erscheinende Bezeichner „*Title*" wird durch Doppelklick in „*Add table view entry*" umbenannt. Die-

ser Bezeichner erscheint später zwar nicht bei der Programmausführung, hilft aber bei der Bestimmung welcher Zellprototyp welchen selbst entworfenen Typ Zelle im Interface Builder repräsentiert. Anschließend wird für diese Zelle im *Attributes inspector* unter dem Eintrag *Identifier* der Bezeichner „*AddCell*" angegeben. Dieser *Identifier*, auch *Reuse Identifier* genannt, eignet sich dazu im Code Zellen dieses Typs zu erstellen.

Für das Aussehen der zweiten Zellprototyps wird der Stil *Subtitle* ausgewählt und der *Identifier* „*NormalCell*" vergeben. Im oberen Schriftzug der Zelle wird „*Text*" und im „*Number*" angegeben (siehe Abb. 4.48).

Abbildung 4.48.: Tabellenansicht im *Grouped*-Stil samt zwei selbst gestalteter Zellprototypen.

Als nächstes wird eine neue Anwendungsszene angelegt, welche dazu dienen soll den Inhalt neuer und bereits erstellter Tabelleneinträgen zu bearbeiten – die Detailansicht. Hierzu wird ein „*View Controller*"-Objekt aus der *Object library* neben dem *Table View Controller* positioniert. Dann werden beide Zellenprototypen nacheinander markiert und per *Ctrl-Drag&Drop* jeweils mit der neuen Szene verbunden – im sich öffnenden Dialog wird die *Push*-Option ausgewählt. Dies hat die Erstellung zwei neuer *Segues* zur Folge, welche im später Verlauf dieses Unterkapitels dazu verwendet werden die Daten einer markierten Zelle an die neue Szene zu übergeben (siehe Abb. 4.49).

Zur besseren Beschreibung der Aufgabe des neuen „*View Controller*"-Objekts wird nach einem Doppelklick in die Mitte dessen *Navigation Bar* der Bezeichner „*Detail View*" vergeben. Anschließend werden aus der *Object library* je zwei *Labels* und zwei Textfelder in den *View Controller* eingefügt. Die *Labels* erhalten den Textinhalt „*Text:*" beziehungsweise „*Number:*" (siehe Abb.). Für das zweite Textfeld wird im *Attributes inspector* der Tastaturtyp „*Number Pad*" festgelegt, in Anlehnung an den Bezeichner des zweiten *Labels*.

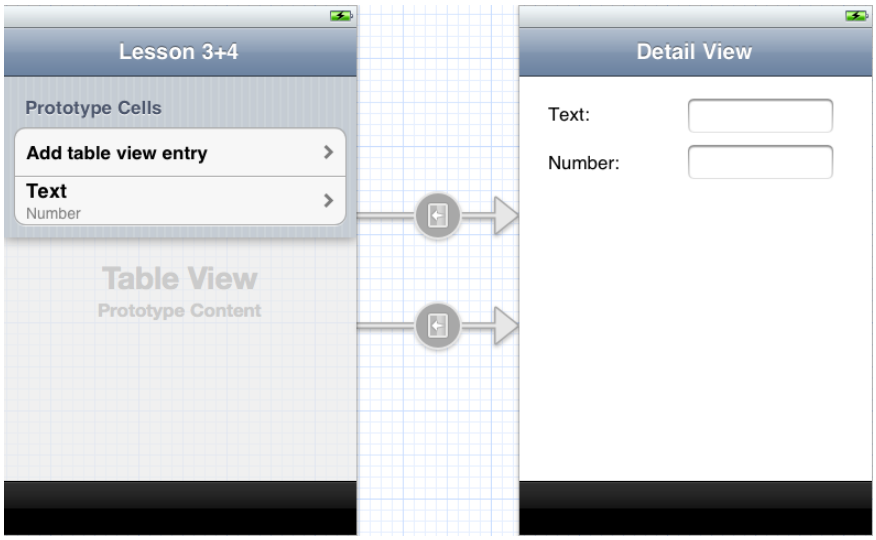

Abbildung 4.49.: Aktueller Stand der Anwendung im *Interface Builder*. Links die Tabellenansicht und rechts die „*Detailansicht*" genannte Szene.

Vorbereitungen für die Erstellung der Programmlogik:

Als nächstes wird die Arbeit hinter die grafischen Kulissen verlagert. Als Vorbereitung dafür sind die folgenden Schritte notwendig:

1. Zuerst wird eine neue *Objective-C*-Klasse namens **`TestTableViewController`** als Unterklasse der Klasse `UITableViewController` angelegt. Im zweiten Schritt des „*File Creation*"-Dialogs werden die Optionen bezüglich des *iPads* als Zielplattform und zur Erstellung einer *XIB*-Datei nicht angekreuzt. Diese Klasse enthält später die Programmlogik zum Laden der Tabelleninhalte, zur Erstellung der Tabellenfelder nach den vordefinierten Zellprototypen und zur Bearbeitung von Ereignissen welche zur Laufzeit in der Tabellenansicht auftreten.
2. Als nächstes wird der „*Table View Controller*"-Objekt in der *MainStoryboard*-Datei markiert als *Controller*-Klasse des Objekts im *Identity Inspector* die neu erstellte Klasse `TestTableViewController` angegeben. Diese Verbindung gibt an, dass die Programmlogik für das „*Table View Controller*"-Objekt in der Klasse `TestTableViewController` implementiert wird.
3. Anschließend wird eine weitere *Objective-C*-Klasse namens **`DetailViewController`** als Unterklasse von `UIViewController` erstellt. Im zweiten Schritt des „*File Creation*"-Dialogs werden die Optionen bezüglich des *iPads* als Zielplattform und zur Erstellung einer *XIB*-Datei nicht angekreuzt. Diese Klasse wird die Programmlogik zum Darstellen und Zurückgeben der (modifizierten) Daten aus einer, in der Tabelle markierten, Zelle enthalten.
4. Im *Storyboard* wird das „*View Controller*"-Objekt aus der „*Detail View*"-Szene markiert und im *Identity Inspector* die Klasse `DetailViewController` angegeben.
5. Abschließend wird eine letzte *Objective-C* Klasse namens **`SomeData`** als Unterklasse von `NSObject` erstellt. Im zweiten Schritt des „*File Creation*"-Dialogs werden die Optionen be-

züglich des *iPads* als Zielplattform und zur Erstellung einer *XIB*-Datei nicht angekreuzt. Jede Zelle der Tabelle erhält ihre Daten zur Laufzeit aus einem Instanzobjekt dieser Klasse. Vorrangig wird sie aber im nächsten Unterkapitel 4.3.4 dazu verwendet vorzuführen, wie komplexe Datenstrukturen auf der Festplatte gespeichert und geladen werden können.

Modifikation der Klasse SomeData:

Nachdem die oben aufgeführten Vorbereitungen durchgeführt wurden, wird eine Datenstruktur und ein Konstruktor in der Klasse `SomeData` implementiert. Hierzu werden im Quellcode der Datei `SomeData.h` folgende Ergänzungen durchgeführt: (siehe Codelisting 4.5)

1. Als erstes wird eine neue *Property* namens **`aString`** erstellt, welche eine Zeichenkette vom Typ `NSString` kapselt (siehe Unterkapitel 3.2.3.5).
2. Anschließend wird eine weitere *Property* erstellt, welche eine Ganzzahl namens **`anInt`** vom Typ `NSInteger`[13] kapselt.
3. Abschließend wird ein Konstruktor namens **`initWithString:andInt:`** deklariert (siehe Unterkapitel 4.2.2).

```
1 @interface SomeData : NSObject
2 @property (nonatomic, copy) NSString *aString;
3 @property (nonatomic) NSInteger anInt;
4
5 -(id) initWithString:(NSString *)theString andInt:(NSInteger)theInt;
6 @end
```

Tabelle 4.5.: Auszug aus der Deklaration der Klasse `SomeClass`.

Als nächstes werden in der Klassenimplementierung `SomeData.m` folgende Ergänzungen durchgeführt: (siehe Codelisting 4.6)

1. Zuerst werden die *Properties* `aString` und `anInt` synthetisiert.
2. Darauf folgt die Implementierung der `initWithString:andInt:`-Methode. In diesem Konstruktor wird getreu dem in Unterkapitel 4.2.2 beschriebenen Schema erst die `init`-Methode der Superklasse aufgerufen. Lief der Aufruf dieser Methode erfolgreich durch werden die Werte der zwei Methodenparameter den Instanzvariablen `aString` und `anInt` zugewiesen.

[13] Beim Bezeichner `NSInteger` handelt es sich um einen primitiven Datentypen. Je nachdem ob es sich bei der Zielplattform des Programms um ein 32- oder 64-Bit System handelt, repräsentiert `NSInteger` einen 32-bit oder 64-bit Integer.

```
@implementation SomeData
@synthesize aString, anInt;
-(id) initWithString:(NSString *)theString andInt:(NSInteger)theInt{
  if(self = [super init]){
    self.aString = theString;
    self.anInt = theInt;
  }
  return self;
}
@end
```

Tabelle 4.6.: Auszug aus der Implementierung der Klasse `SomeClass`.

Modifikation der Klasse TestTableViewController:

Als nächstes wird die Programmlogik, zur Anzeige und Manipulation der Tabelleneinträge, in der Klasse `TestTableViewController` implementiert. Dazu werden die folgenden Ergänzungen in der Klassenimplementierung **`TestTableViewController.m`** vorgenommen[14]:

1. Als erstes wird die *Header*-Datei der Klasse `SomeData` importiert, da im späteren Verlauf auf Implementierungsdetails dieser Klasse zurückgegriffen wird (siehe Codelisting 4.7).

```
#import "TestTableViewController.h"
#import "SomeData.h"
```

Tabelle 4.7.: Importanweisungen am Anfang der Klassenimplementierung.

2. In der Klassenerweiterung (siehe Unterkapitel 3.2.5.2) wird eine neue *Property* angelegt (siehe Codelisting 4.8). Sie kapselt ein Instanzobjekt namens **`dataArray`** vom Typ `NSMutableArray`[15]. Dieses Array wird später mit `SomeData`-Instanzobjekten gefüllt und bildet die Quelle der Tabelleneinträge.

[14] Da eine komplette Abbildung der Klassenimplementierung unübersichtlich wäre, werden die im folgenden durchgeführten Änderungen durch Codeauszüge aus der Datei begleitet. Der vollständigen Quellcode aller Projektdateien befindet sich auf dem Datenträger, welcher dieser Ausarbeitung beiliegt (siehe Appendix VI).

[15] Ein `NSMutableArray`-Instanzobjekt unterscheidet sich dahingegen von einem `NSArray`-Instanzobjekt, dass es Änderungen des Inhalts des repräsentierten Arrays erlaubt. Ein *Objective-C*-Array unterscheidet sich von Arrays aus anderen Sprachen dahingegen, dass es sich dabei um einen heterogene Datencontainer handelt, in dem Objekte beliebiger Klassen in beliebiger Kombination gespeichert werden können. Die Grundvoraussetzung dafür ist, dass die in einem Array gespeicherten Objekte vom dynamischen Typ `id` sind – was in *Objective-C* für jeden Objekttyp zutrifft (siehe Unterkapitel 3.2.3.3).

```
@interface TestTableViewController ()
@property (nonatomic, strong) NSMutableArray *dataArray;
@end
```

Tabelle 4.8.: Deklaration der *Property* `dataArray` in der Klassenerweiterung.

3. In Anschluss an die Deklaration der neuen *Property* wird diese in der Klassenimplementierung mit dem Schlüsselwort **`@synthesize`** definiert (siehe Codelisting 4.9).

```
@implementation TestTableViewController
@synthesize dataArray;
...
@end
```

Tabelle 4.9.: Definition der *Property* `dataArray` in der Klassenimplementierung.

4. In der Methode **`viewDidLoad`** (siehe Unterkapitel 4.3.2) wird das gerade definierte `dataArray` mit Beispieldaten gefüllt (siehe Codelisting 4.10).

   Anmerkung: In diesem Kapitel werden die Daten beim Start der Anwendung fest vorgegeben! Im nächsten Kapitel 4.3.4 wird gezeigt wie sich Daten von der Festplatte laden und darauf ablegen lassen.

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // Uncomment the following line to preserve selection between presentations.
  // self.clearsSelectionOnViewWillAppear = YES;
  // Uncomment the following line to display an Edit button in the navigation bar
      for this view controller.
  // self.navigationItem.rightBarButtonItem = self.editButtonItem;

  SomeData *data1 = [[SomeData alloc] initWithString:@"Data1" andInt:1];
  SomeData *data2 = [[SomeData alloc] initWithString:@"Data2" andInt:2];
  self.dataArray = [[NSMutableArray alloc] initWithObjects:data1, data2, nil];
}
```

Tabelle 4.10.: Erstellung von Beispieldaten für die Anzeige in der Tabelle.

5. Um die Speicherbalance zu wahren, wird das Objekt das der Instanzvariable `dataArray` zugewiesen wurde in der Methode **`viewDidUnload`** (siehe Unterkapitel 4.3.2) wieder freigegeben (siehe Codelisting 4.11).

```
1 - (void)viewDidUnload
2 {
3   [super viewDidUnload];
4   // Release any retained subviews of the main view.
5   // e.g. self.myOutlet = nil;
6   self.dataArray = nil;
7 }
```

Tabelle 4.11.: Freigabe des Instanzobjekts das `dataArray` zugewiesen wurde.

6. Anschließend wird die Methode **`numberOfSectionsInTableView:`** modifiziert (siehe Codelisting 4.12). Sie gibt die Anzahl der Gruppen (siehe Abb. 4.46) in einer Tabelle an – in der Beispielanwendung sind es zwei Gruppen.

```
1 - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
2 {
3   // Return the number of sections.
4   return 2;
5 }
```

Tabelle 4.12.: Die Beispielanwendung besitzt zwei Gruppen (engl. *sections*).

7. Die erste Gruppe enthält nur ein Tabellenfeld, das zum Anlegen neuer Tabelleneinträge gedacht ist. Die zweite Gruppe enthält so viele Einträge wie in der Instanzvariable `dataArray` gespeichert sind. Diesen Umständen wird in der Methode **`tableView:numberOfRowsInSection:`** Rechnung getragen (siehe Codelisting 4.13). In der fünften Zeile wird das Schlüsselwort `self` in Kombination mit der *Dot*-Notation verwendet um die Akzessor-Methode der *Property* `dataArray` zu verwenden (siehe Unterkapitel 3.2.3.5).

```
1 - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)
      section
2 {
3   // Return the number of rows in the section.
4   if(section == 0) return 1;
5   else return self.dataArray.count;
6 }
```

Tabelle 4.13.: Rückgabe der Anzahl Tabellenfelder pro Tabellengruppe.

8. Als nächstes werden in der Methode **`tableView:cellForRowAtIndexPath:`** die Tabellenfelder erstellt (siehe Codelisting 4.14).

   Abhängig von der Tabellengruppe soll entweder eine spezielle Zelle für die Erstellung neuer Tabelleneinträge oder die bereits angelegten Tabelleneinträge angezeigt werden. Den Grundstein für die Differenzierung dieser unterschiedlichen Zelltypen wurde im früheren Verlauf

dieses Unterkapitels im *MainStoryboard* festgelegt. Dort wurden zwei Tabellenfeldtypen unter den Identifikationsbezeichnern *AddCell* und *NormalCell* gestaltet.

Um der Gruppe entsprechende Tabellenfelder anlegen zu können, wird der Methodenparameter **`indexPath`** abgefragt. Ein `NSIndexPath`-Instanzobjekt stellt eine Datenstruktur dar, welche zwei Properties namens `section` und `row` besitzt. Wählt ein Anwender zu Laufzeit eine Zelle aus, enthält **`section`** den Index der Gruppe in der sich diese Zelle befindet und **`row`** den Index der markierten Zelle in dieser Gruppe.

Die erste Gruppe der Beispielapplikation erhält eine Zelle vom Typ *AddCell* mit dem Schriftzug *„Add table-entry"*. Für die zweite Gruppe wird für jeden Eintrag in `dataArray` eine Zelle, die sich beim Aussehen an dem Zellprototypen *NormalCell* orientiert, erstellt.

```
1 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(
      NSIndexPath *)indexPath
2 {
3   static NSString *CellIdentifier;
4   UITableViewCell *cell;
5   if(indexPath.section == 0){
6     CellIdentifier = @"AddCell";
7     cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
8     cell.textLabel.text = @"Add table-entry";
9   } else {
10    CellIdentifier = @"NormalCell";
11    cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
12    SomeData *tmpData = [self.dataArray objectAtIndex:indexPath.row];
13    cell.textLabel.text = tmpData.aString;
14    cell.detailTextLabel.text = [NSString stringWithFormat:@"%d", tmpData.anInt];
15  }
16  return cell;
17 }
```

Tabelle 4.14.: Erzeugung der Tabellenfelder.

9. Abschließend wird eine neue Methode namens **`prepareForSegue:sender:`** erstellt (siehe Codelisting 4.15). Eine so benannte Methode wird automatisch von der Laufzeitumgebung aufgerufen bevor eine Transition über ein *Segue* (siehe Unterkapitel 4.3.2) zwischen zwei, in einer gemeinsamen *Storyboard*-Datei enthaltenen, Szenen stattfindet. Über den Parameter `segue` und die hierin enthaltene *Property* `destinationViewController` lässt sich das Ziel der Transition im Code bestimmen.

   Die Methode wird in dieser Klasse dazu verwendet einen Datensatz, der mit einer vom Anwender selektierten Tabellenzelle assoziiert ist, an das Ziel der Transition zu übermitteln – hier ein Instanzobjekt vom Typ `DetailViewController`. Die Methode ruft dabei eine weitere Methode im Zielobjekt auf und übergibt ihr die Daten. Um die externe Methode aufrufen zu können, muss eine von zwei Konditionen erfüllt sein:

   a) Die Schnittstelle der fremden Klasse – hier die Klasse `DetailViewController` – wird statisch über eine *#import*-Anweisung in die aktuellen Klasse eingebunden (siehe Unterkapitel 3.2.1.3 und 3.2.2.2).

b) Statt die fremde Klasse statisch einzubinden und eventuell so problematische Abhängigkeitsverhältnisse hervorzurufen kann sich das dynamische Verhalten von *Objective-C* und der für *iOS*-Programme genutzten Laufzeitumgebung zu Nutze gemacht werden. Dank Introspektion kann zur Laufzeit ermittelt werden, ob ein Objekt eine bestimmte Methode implementiert. Durch ein *Key-Value-Coding* genanntes Konzept, das jede Klasse auf Basis der `NSObject`-Klasse implementiert, wird bei Aufruf der Methode `setValue:forKey:` dank des Konzepts der dynamischen Methodenauflösung (siehe Unterkapitel 3.2.4.1) zur Laufzeit nach einer *Setter*-Methode mit dem Bezeichner des ersten Methodenparameters gesucht. Wird eine entsprechende Methode gefunden, wird sie aufgerufen und ihr der Wert des zweiten Parameters der `setValue:forKey:`-Methode übergeben. So kann eine Methode indirekt aufgerufen werden, auch dann wenn nicht sicher ist ob die fremde Klasse die Methode implementiert.

Um den dynamischen Aspekt der Sprache *Objective-C* zu veranschaulichen, wird in der Beispielanwendung die zweite Variante gewählt. Zuerst wird in der Methode (zur Laufzeit) geprüft ob das Zielobjekt der Transition die *Setter*-Methoden `setDelegate:` und `setSelection:` implementiert (siehe vierte und siebte Zeile im Codelisting).

Dann wird die aktuelle Klasse, bzw. ein Instanzobjekt der Klasse, als *Delegate* (siehe Unterkapitel 4.1.1) des Empfängerobjekts bestimmt – in diesem Fall ein Instanzobjekt der Klasse `DetailViewController`. Dies dient dazu, dass das `DetailViewController`-Objekt weiß wohin es das erhaltenen `SomeData`-Objekt nach der Bearbeitung durch den Anwender zurückgeben soll.

Im zweiten Anweisungsblock wird der `indexPath` der zur Laufzeit vom Anwender selektierten Zelle ermittelt. Diese Information wird später dazu benötigt, die vom `DetailViewController` zurückgelieferten Daten an der richtigen Stelle im `dataArray` zu speichern. Anschließend wird das zu der vom Anwender selektierten Zelle gehörige `SomeData`-Instanzobjekt ermittelt und zusätzlich zum `indexPath` in ein `NSDictionary`-Instanzobjekt gespeichert. Wurde die Zelle mit dem Schriftzug „*Add table-entry*" angeklickt, wird ein neues `SomeData`-Instanzobjekt erstellt und im `NSDictionary`-Objekt gespeichert (siehe neunte Zeile im Codelisting).

Abschließend wird das erstelle Informationsbündel an das Zielobjekt der *Segue*-Transition übermittelt.

```
1 -(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{
2   UIViewController *destination = segue.destinationViewController;
3
4   if([destination respondsToSelector:@selector(setDelegate:)]){
5     [destination setValue:self forKey:@"delegate"];
6   }
7   if([destination respondsToSelector:@selector(setSelection:)]){
8     NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
9     id object = (indexPath.section == 0) ? [[SomeData alloc] init] : [self.dataArray
          objectAtIndex:indexPath.row];
10    NSDictionary *selection = [NSDictionary dictionaryWithObjectsAndKeys:indexPath,
          @"indexPath", object, @"object", nil];
11    [destination setValue:selection forKey:@"selection"];
12  }
13 }
```

Tabelle 4.15.: Zusammentragen und -setzen der nötigen Informationen vor der Transition in die *„Detail View"*-Szene.

Modifikation der Klasse DetailViewController :

Als nächstes müssen Änderungen in der Klasse `DetailViewController` durchgeführt werden, damit die Daten der markierten Zelle angezeigt und modifiziert werden können. Folgende Änderungen sind hierzu in der Klassendeklaration **`DetailViewController.h`** erforderlich (siehe Codelisting 4.16):

1. Zuerst wird eine neue *Property* namens **`selection`** vom Typ `NSDictionary` erstellt. Bei einer Transition zu einem Instanzobjekt dieser Klasse, werden die Daten der zu editierenden Zelle in diese Variable kopiert.
2. Als nächstes wird eine weitere *Property* namens **`delegate`** vom generischen Datentyp `id` angelegt. Diese speichert eine Referenz auf das `TestTableViewController`-Instanzobjekt an die die zu editierenden Zelldaten später zurückgegeben werden sollen. Hierbei ist aus Gründen der Speicherverwaltung darauf zu achten, als *Property*-Attribut *weak* anzugeben (siehe Unterkapitel 4.2.3).
3. Als letztes werden jeweils ein *IBOutlet* für das obere und das untere Textfeld der *„Detail View"*-Szene angelegt[16]. Für das erste *IBOutlet* wird der *Property*-Bezeichner **`aTextField`** und für das zweite IBOutlet der *Property*-Beteichner **`aNumberField`** gewählt.

[16]Zur Erinnerung: ein *IBOutlet* wird am einfachsten angelegt indem im *Storyboard* das Textfeld angeklickt und eine Verbindung zum Code durch *Ctrl-Drag&Drop* erstellt wird (siehe Unterkapitel 4.3.1).

```
1 @interface DetailViewController : UIViewController
2 @property (nonatomic, copy) NSDictionary *selection;
3 @property (nonatomic, weak) id delegate;
4 @property (weak, nonatomic) IBOutlet UITextField *aTextField;
5 @property (weak, nonatomic) IBOutlet UITextField *aNumberField;
6 @end
```

Tabelle 4.16.: Auszug aus der Deklaration der Klasse `DetailViewController`.

Abschließend erfährt die Definition der `DetailViewController`-Klasse in der Datei **`DetailViewController.m`** die folgenden Modifikationen (siehe Codelisting 4.17):

1. Die Schnittstelle der Klasse `SomeData` wird importiert. Hierdurch wird der Zugriff auf die Datenstruktur und die Akzessor-Methoden der Klasse ermöglicht, deren Instanzen die Daten der Tabelleneinträge in der Tabellenansicht enthalten. Die Informationen zu einer Zelle, die zur Laufzeit an die Detailansicht übertragen werden, werden in den Textfeldern der Detailansicht angezeigt, wo sie zur Laufzeit vom Benutzer modifiziert werden können.
2. In der Klassenerweiterung wird eine weitere *Property* namens **`editedData`** vom Objekttyp `SomeData` angelegt. Sie speichert das dekodierte Datenobjekt, welches bei der *Segue*-Transition übermittelt wurde.
3. Die zwei zuvor deklarierten *Properties* werden in der Klassenimplementierung synthetisiert.
4. In der Methode **`viewDidLoad`** wird die `SomeData`-Objektinstanz welche bei der *Segue*-Transition, in einem `NSDictionary`-Objekt kodiert, übermittelt wurde extrahiert und der Variablen `editedData` zugewiesen. Anschließend wird der Inhalt der `SomeData`-Datenstruktur auf die zwei Textfelder verteilt. Zusätzlich wird am Ende der Methode dafür Sorge getragen, dass das Textfeld `aTextField`, zur Laufzeit sofort nach der Transition in die „*Detail View*"-Szene zur Editierung markiert wird.
5. In der Methode **viewDidUnload** werden alle zuvor erstellten *Properties* freigegeben.

```objc
#import "DetailViewController.h"
#import "SomeData.h"

@interface DetailViewController ()
@property (nonatomic, weak) SomeData *editedData;
@end

@implementation DetailViewController
@synthesize aTextField,
@synthesize aNumberField;
@synthesize editedData, selection, delegate;

...

- (void)viewDidLoad {
  [super viewDidLoad];
  // Do any additional setup after loading the view.
  self.editedData = [self.selection objectForKey:@"object"];
  self.aTextField.text = self.editedData.aString;
  self.aNumberField.text = [NSString stringWithFormat:@"%d", self.editedData.anInt];
  [self.aTextField becomeFirstResponder];
}

- (void)viewDidUnload {
  [self setATextField:nil];
  [self setANumberField:nil];
  [super viewDidUnload];
  // Release any retained subviews of the main view.
  self.editedData = nil;
  self.delegate = nil;
  self.selection = nil;
}

...

@end
```

Tabelle 4.17.: Implementierung der Klasse `DetailViewController`.

Ausführung der Beispielanwendung:

Wird die Anwendung ausgeführt und über das *Tab Bar Item* „Lesson 3+4" in die Tabellenansicht gewechselt, erscheint die Tabelle mit den zwei erstellten Gruppen (siehe Abb. 4.50, linkes Teilbild). Die oberste Gruppe enthält ein einzelnes Tabellenfeld, nach dessen Betätigung eine Transition in die *„Detail View"*-Szene erfolgt. In der neuen Sicht können Informationen in die beiden Datenfelder eingegeben werden. Über den Zurückknopf mit dem Titel *„Lesson 3+4"* geht es in die Tabellenansicht zurück. Wird eine der unteren zwei Zellen angeklickt werden in der *„Detail View"*-Szene die Daten aus dem, mit der markierten Zelle assoziierten, `SomeData`-Instanzobjekt in den Textfeldern angezeigt (siehe ebd., rechtes Teilbild).

Abbildung 4.50.: Darstellung der Tabellen- und Detailansicht im iOS Simulator.

Zum Abschluss des Kapitels ist anzumerken, dass Modifikationen der Daten oder das Anlegen neuer Tabelleneinträge nach einem Neustart der Anwendung und nach Verlassen der *„Detail View"*-Szene verloren gehen. Dieses Manko wird im nächsten Unterkapitel adressiert (siehe Unterkapitel 4.3.4).

4.3.4. Vierter Schritt: Datenpersistenz

4.3.4.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- Speichern von komplexen Datenstrukturen auf die Festplatte,
- Laden von komplexen Datenstrukturen von der Festplatte,
- Hinzufügen, Editieren und Löschen von Einträgen in *Table Views*.

4.3.4.2. Implementierung

In diesem Unterkapitel soll das Projekt aus dem letzten Unterkapitel 4.3.3 um Funktionalität erweitert werden, welche die Beispielanwendung dazu befähigt, sich, zur Laufzeit durchgeführte, Modifikationen der Tabelleneinträge permanent zu merken. Diese Datenpersistenz lässt sich durch ein Eigenes oder unter Berufung auf eines der folgenden Konzepte aus dem *iOS-SDK* realisieren: (vgl. [Dav11], S. 445ff)

- ***Property* Listen** sind einfach zu benutzen. Damit können aber keine Instanzobjekte eigens geschriebener Klassen abspeichern und laden.
- **Objektarchive** sind leicht zu implementieren und erlauben zusätzlich zu den Klassen welche mit *Property Listen* gespeichert werden können, auch das Speichern beliebiger, selbst erstellter Klassen.
- Mit ***SQLite3*** steht *iOS*-Programmen ein voll ausgeprägtes relationales Datenbanksystem zur Verfügung. Diese gibt einem Anwendungsentwickler reichlich Möglichkeiten, ist aber für den Einsatz in kleineren Programmen meist unnötig kompliziert und umfangreich.
- ***Core Data*** ist ein von *Apple* entwickeltes *Framework* für Datenpersistenz. Es spezifiziert eine eigene Modellierungssprache mit der Datenstrukturen und Beziehungen zwischen Daten abgebildet werden können. Die Erstellung solcher Beschreibungen kann direkt in *Xcode* erfolgen. Außerdem beinhaltet das *Framework* Funktionen zur Verwaltung von Datenstrukturen zur Laufzeit. Für einfache Projekte mit flachen und relativ einfachen Datenstrukturen ist dieses Konzept meist unnötig kompliziert und umfangreich. Sie bietet aber eine solide und effiziente Alternative zu allen anderen Methoden.

Für die Beispielanwendung wird die Variante mit Objektarchiven gewählt, da sich dieses Konzept leicht implementieren lässt und Objekte eigens geschriebener Klassen abgespeichert werden können – so z.Bsp. `SomeData`-Instanzobjekte. Die Klassen müssen hierzu lediglich serialisiert werden, so dass ihre Instanzobjekte als Bitstrom u.a. auf die Festplatte gespeichert, von der Festplatte gelesen oder über das Netzwerk transferiert werden können. Eine Serialisierung einer Klasse wird erreicht, indem die Klasse dem Protokoll **`NSCoding`** aus dem *iOS-SDK* genügt. Dieses Protokoll gibt zwei Methoden vor, die sich dazu eignen Daten zu kodieren und zu dekodieren.

Im folgenden werden die Änderungen und Ergänzungen an den Projektdateien beschrieben, die nötig sind um das `NSCoding`-Protokoll in der Klasse `SomeData` zu realisieren und um `SomeData`-Instanzobjekte von der Festplatte zu laden sowie darauf zu speichern.

Modifikation der Klasse SomeData:

Die Klasse `SomeData` wird erweitert um das `NSCoding`-Protokoll zu unterstützen. Hierfür sind die folgenden Ergänzungen in der Deklaration der Klasse `SomeData.h` nötig (siehe Codelisting 4.18):

1. Zur Erleichterung weiterer Schritte und damit Tippfehler später keine unerwünschtes Verhalten hervorrufen, werden drei Konstanten mit der *C*-Präprozessordirektive `#define` erstellt (siehe Unterkapitel 3.2.1.3). Die Konstanten spielen eine Rolle bei der Kodierung und Dekodierung von `SomeData`-Instanzobjekten und der darin enthaltenen Daten.
2. Abschließend wird die Klassendeklaration um die Unterstützung des `NSCoding`-Protokolls erweitert (siehe Unterkapitel 3.2.6).

```
1  #import <Foundation/Foundation.h>
2
3  #define _dataKey            @"DataObject"
4  #define _aStringKey         @"AString"
5  #define _anIntKey           @"AnInt"
6
7  @interface SomeData : NSObject <NSCoding>
8  @property (nonatomic, copy) NSString *aString;
9  @property (nonatomic) NSInteger anInt;
10
11 -(id) initWithString:(NSString *)theString andInt:(NSInteger)theInt;
12 @end
```

Tabelle 4.18.: Die für die Serialisierung angepasste `SomeData`-Klassendeklaration.

Anschließend werden die folgenden Ergänzungen in der Definition der Klasse `SomeData.m` durchgeführt (siehe Codelisting 4.19):

1. In *Xcode* können Methoden durch angeben der Präprozessordirektive **`#pragma mark`** gruppiert werden. Die Angabe dieser Anweisung erfolgt innerhalb der Klassendefinition – zwischen den Schlüsselwörtern `@implementation` und `@end` – aber außerhalb von Methodenimplementierungen. Auf das Schlüsselwort folgt der Name der Gruppe, im Fall der Beispielanwendung „*NSCoding*". *Xcode* fasst die beiden in den nächsten Unterpunkten implementierten Methoden, dann unter diesem Namen in der ***Jump Bar*** zusammen (siehe Abb. 4.51). Ein Bindestrich im Gruppennamen bewirkt eine Trennlinie an der Stelle in der *Jump Bar*.

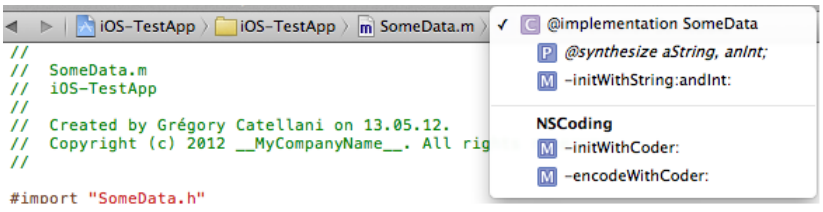

Abbildung 4.51.: *Xcode Jump Bar* mit der durch `#pragma mark` erstellten „*NSCoding*"-Methodengruppierung für die `SomeData`-Klassenimplementierung.

2. Als nächstes wird die Konstruktor-Methode **`initWithCoder:`** implementiert, die vom Protokoll `NSCoding` vorgegeben wird. Der Aufbau orientiert sich dabei an dem in Unterkapitel 4.2.2 beschrieben Schema.

   Der einzige Parameter der Methode liefert ein Dekodier-Objekt. Dieses besitzt eine Schlüssel-kodierte Version eines `SomeData`-Instanzobjekts (siehe Paragraph „Modifikation der Klasse `TestTableViewController`" in diesem Unterkapitel). Darin enthalten sind die Informationen für die Datenstruktur des Instanzobjekts. Anhand der Schlüssel `AString` und `AnInt` (siehe Deklaration der Klasse) werden die Werte ausgelesen.

3. Abschließend wird die Methode **`encodeWithCoder:`** implementiert, die vom Protokoll `NSCoding` vorgegeben wird. Sie erhält ein Kodierobjekt, das eine eine Schlüssel-kodierte Version des Instanzobjekts erstellt und sich dazu eignet dessen Datenstruktur auf einem Medium abzulegen.

```
1  #pragma mark - NSCoding
2
3  -(id)initWithCoder:(NSCoder *)aDecoder{
4    if(self = [super init]){
5      self.aString = [aDecoder decodeObjectForKey:_aStringKey];
6      self.anInt = [aDecoder decodeIntegerForKey:_anIntKey];
7    }
8    return self;
9  }
10
11 -(void)encodeWithCoder:(NSCoder *)aCoder{
12   [aCoder encodeObject:self.aString forKey:_aStringKey];
13   [aCoder encodeInteger:self.anInt forKey:_anIntKey];
14 }
```

Tabelle 4.19.: Die für die Serialisierung angepasste `SomeData`-Klassenimplementierung.

Modifikation der Klasse TestTableViewController:

Die Klasse `TestTableViewController` welche die Tabelle aus der „*Table View*"-Szene verwaltet wird im folgenden erweitert um die Einträge der Tabelle aus `SomeData`-Instanzobjekten, die

von der Festplatte geladen werden, zu gewinnen und nach Modifikation wieder auf der Festplatte abzuspeichern. Dazu werden die folgenden Änderungen an der Implementierungsdatei `TestTableViewController.m` durchgeführt (siehe Codelistings 4.20 und 4.21):

1. Am Ende der Implementierung wird eine neue Methodengruppe mit der Anweisung „`#pragma mark - Data save and load`" erstellt, die Methoden für den Zugriff auf, auf der Festplatte abgelegte, `SomeData`-Instanzobjekte, in *Xcode* zusammenfasst.

2. Die neue Methode **`dataFilePath`** wird sowohl von den neu zu erstellenden Methoden zum Laden und Speichern von Daten auf der Festplatte benutzt. Sie wird benötigt um die Datei, in der die `SomeData`-Instanzobjekte auf der Festplatte abgelegt werden, in der Ordnerstruktur der Anwendung zu ermitteln.

   Eine wichtige Rolle beim Abspeichern und Laden von Daten, in jeder Anwendung, spielt der *Documents*-Ordner (vgl. [Dav11], S. 446). Jede Anwendung besitzt genau einen so genannten Ordner indem sie Daten ablegen darf. Ein Zugriff auf den Dokumentenordner anderer Programme ist nicht gestattet (siehe Unterkapitel 4.1.1).

3. Die Methode **`loadDataFromDisk`** greift mit einem `NSKeyedUnarchiver`-Instanzobjekt auf die Datei im *Documents*-Ordner zu und liest sie in Rohfassung in einen Puffer ein. Dann wird in anhand des Schlüssels „*DataObject*" (siehe `SomeData`-Klassendeklaration) nach in der Datei enthaltenen `SomeData`-Instanzobjekten gesucht. Sie sind unter diesem Schlüssel in der Datei kodiert abgelegt. Gefundene `SomeData`-Instanzobjekte werden extrahiert und in ein **`dataArray`** genanntes Array gespeichert. Sollten keine Daten auf der Platte gespeichert gewesen sein, wird das Array neu initialisiert.

4. Als nächstes wird die Methode `saveDataToDisk` implementiert. Diese benutzt ein `NSKeyedArchiver`-Objekt, das jedes `SomeData`-Instanzobjekt, und die damit verbundene Datenstruktur, jeweils unter dem Schlüssel „*DataObject*" (siehe `SomeData`-Klassendeklaration) in eine *XML*-Datei auf der Platte ablegt. Beim Enkodieren eines `SomeData`-Instanzobjekts wird die `encodeWithCoder:`-Methode des Objekts aufgerufen.

   Als Resultat enthält die *Data.plist*-Datei für jedes *SomeData*-Objekt einen Schlüssel „*DataObject*" und darunter zwei Unterschlüssel „`AString`" und „`AnInt`", welche die Werte der Objektdatenstruktur enthalten.

```objc
#pragma mark - Data save and load

- (NSString *) dataFilePath{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
      NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  return [documentsDirectory stringByAppendingPathComponent:@"Data.plist"];
}

- (void) loadDataFromDisk{
  NSData *data = [[NSData alloc] initWithContentsOfFile:[self dataFilePath]];
  NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] initForReadingWithData:
      data];

  self.dataArray = [unarchiver decodeObjectForKey:_dataKey];
  [unarchiver finishDecoding];

  if(self.dataArray == nil) self.dataArray = [[NSMutableArray alloc] init];
}

- (void) saveDataToDisk{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] initForWritingWithMutableData
      :data];
  [archiver setOutputFormat:NSPropertyListXMLFormat_v1_0];

  [archiver encodeObject:self.dataArray forKey:_dataKey];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}
```

Tabelle 4.20.: Methoden zum Laden und Speichern von *SomeData*-Instanzobjekten.

5. Abschließend wird die Methode **`viewDidLoad`** angepasst. Hier entfällt die im letzten Unterkapitel eingeführte statische Erstellung von *SomeData*-Instanzobjekte (vgl. Codelisting 4.10). Stattdessen wird die Methode `loadDataFromDisk` aufgerufen um mögliche *SomeData*-Instanzobjekte von der Festplatte zu laden.

```
1 - (void)viewDidLoad
2 {
3   [super viewDidLoad];
4
5   // Uncomment the following line to preserve selection between presentations.
6   // self.clearsSelectionOnViewWillAppear = YES;
7
8   // Uncomment the following line to display an Edit button in the navigation bar
        for this view controller.
9   // self.navigationItem.rightBarButtonItem = self.editButtonItem;
10
11  //    SomeData *data1 = [[SomeData alloc] initWithString:@"Data1" andInt:1];
12  //    SomeData *data2 = [[SomeData alloc] initWithString:@"Data2" andInt:2];
13  //    self.dataArray = [[NSMutableArray alloc] initWithObjects:data1, data2, nil];
14  [self loadDataFromDisk];
15 }
```

Tabelle 4.21.: Angepasste `viewDidLoad`-Methode.

Als nächstes sollen `SomeData`-Instanzobjekte abgespeichert werden können. Hierfür sind folgende Änderungen in der Datei `TestTableViewController.m` nötig (siehe Codelisting 4.22, 4.23 und 4.24):

1. Zuerst wird die Methode **`tableView:canEditRowAtIndexPath:`** implementiert. Sie steht in jeder Unterklasse der Klasse `UITableViewController` zur Verfügung, ist standardmäßig aber auskommentiert. Sie gestattet die Umsortierung und die Löschung von Zellen, was für jedes Tabellenfeld separat eingestellt werden kann.

   Bis auf das Tabellenfeld mit dem Bezeichner *„Add table-entry"* aus der ersten Tabellengruppe (engl. *section*) kann und darf jede Zelle gelöscht und umsortiert werden.

2. Die Methode **`tableView:commitEditingSytle:forRowAtIndexPath:`** beschreibt was bei der Bearbeitung eines Tabellenfeldes passiert. Dabei kann zwischen verschiedenen Bearbeitungsszenarien unterschieden werden.

   In der Beispielanwendung dürfen einzelne Tabellenzellen gelöscht werden. Ist dies für eine Zelle zur Laufzeit der Fall, wird sie aus dem `dataArray`-Array und der Tabellenansicht entfernt. Danach wird die Änderung auf der Festplatte gespeichert.

3. Die Methode **`tableView:moveRowAtIndexPath:toIndexPath:`** enthält die Anweisungen die ausgeführt werden, wenn der Anwender eine Zelle umsortiert. Nach einer Umsortierung in der Tabelle, werden die betroffenen Einträge im `dataArray`-Array entsprechend umsortiert.

4. Die Methode **`tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:`** wird zwar von der Klasse `UITableViewController` angeboten, ist aber nicht standardmäßig, wenngleich auskommentiert, in jeder Unterklasse enthalten[17]. Sie

[17] Tippt der Entwickler die Zeichenkette *„tableView"* ein und betätigt die Tasten Steuerung und Leertaste gemeinsam, kann die Methode leicht anhand der *Xcode*-Eingabevervollständigung angelegt werden.

wird dazu verwendet zu überprüfen ob eine bestimmte Zelle bei der Umsortierung an eine bestimmte Stelle in der Tabelle verschoben werden darf.

Die Beispielanwendung enthält zwei Gruppen. In der oberen steht eine spezielle Zelle, die dem Hinzufügen neuer Tabelleneinträge dient. In diese Gruppe soll kein Tabellenfeld der unteren Gruppe verschoben werden. Versucht der Anwender es trotzdem indem er eine Zelle aus der unteren Gruppe hochzieht und loslässt, springt die Zelle wieder an ihre Ausgangsposition zurück.

5. Als nächstes wird die Methode **`tableView:canMoveRowAtIndexPath:`** implementiert. Darin kann für jede Zelle oder Gruppe gesondert eingestellt werden ob sie verschoben werden kann. In der Beispielanwendung darf die Zelle zum Erstellen neuer Einträge nicht verschoben werden, die anderen Tabellenfelder aber schon.

```
// Override to support conditional editing of the table view.
- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)
    indexPath
{
  // Return NO if you do not want the specified item to be editable.
  if(indexPath.section == 0) return NO;
  else return YES;
}

// Override to support editing the table view.
- (void)tableView:(UITableView *)tableView commitEditingStyle:(
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)
    indexPath
{
  if (editingStyle == UITableViewCellEditingStyleDelete) {
    // Delete the row from the data source
    [self.dataArray removeObjectAtIndex:indexPath.row];
    [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
        withRowAnimation:UITableViewRowAnimationFade];
    [self saveDataToDisk];
  }
}

// Override to support rearranging the table view.
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath *)
    fromIndexPath toIndexPath:(NSIndexPath *)toIndexPath
{
  [self.dataArray exchangeObjectAtIndex:[fromIndexPath row] withObjectAtIndex:[
      toIndexPath row]];
  [self saveDataToDisk];
}

-(NSIndexPath *)tableView:(UITableView *)tableView
    targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath
    toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath
{
  if(sourceIndexPath.section != proposedDestinationIndexPath.section) return
      sourceIndexPath;
  else return proposedDestinationIndexPath;
}

// Override to support conditional rearranging of the table view.
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)
    indexPath
{
  // Return NO if you do not want the item to be re-orderable.
  if(indexPath.section == 0) return NO;
  else return YES;
}
```

Tabelle 4.22.: Detaillierte Beschreibung des Verhaltens und Möglichkeiten zur Editierung der Tabelle.

6. Nachdem die vorherigen Methoden, die das Bearbeitungsverhalten der Tabelle detailliert festlegen, implementiert wurden, wird die **`viewDidLoad`**-Methode ein letztes Mal bearbeitet.

   Die zwei von *Apple* dem `UITableViewController`-Template beigefügten, aber standardmäßig auskommentierten, Anweisungen werden aus auskommentiert. Die erste dient dazu, dass wenn ein Anwender eine Zelle zum Editieren anklickt und nach der Editierung in der „*Detail View*"-Szene zur Tabellenansicht zurückkehrt, diese markierte Zelle unmarkiert wird. Bliebe diese Zeile auskommentiert wäre das Tabellenfeld permanent markiert bis eine andere Zelle angeklickt wird.

   Die zweite Anweisung, fügt der *Navigation Bar* (siehe Unterkapitel 4.3.3) einen *Edit*-Knopf in der rechten Ecke hinzu. Dieser ist bereits mit Programmlogik belegt, die dazu dient die Tabelle in einen Editier-Modus und aus diesem heraus zu schalten[18].

```
1 - (void)viewDidLoad
2 {
3   [super viewDidLoad];
4
5   // Uncomment the following line to preserve selection between presentations.
6   self.clearsSelectionOnViewWillAppear = YES;
7
8   // Uncomment the following line to display an Edit button in the navigation bar
        for this view controller.
9   self.navigationItem.rightBarButtonItem = self.editButtonItem;
10
11  //    SomeData *data1 = [[SomeData alloc] initWithString:@"Data1" andInt:1];
12  //    SomeData *data2 = [[SomeData alloc] initWithString:@"Data2" andInt:2];
13  //    self.dataArray = [[NSMutableArray alloc] initWithObjects:data1, data2, nil];
14  [self loadDataFromDisk];
15 }
```

Tabelle 4.23.: Finale Version der `viewDidLoad`-Methode der Klasse `TestTableViewController`.

7. Markiert ein Benutzer zur Laufzeit ein Tabellenfeld in der Tabellenansicht wird die „*Detail View*"-Szene aufgerufen (siehe Unterkapitel 4.3.3). Über die `prepareForSegue:sender:`-Methode wird dieser Szene das, zur markierten Zelle passende, `SomeData`-Instanzobjekt übergeben, wo die Daten editiert werden können (siehe Unterkapitel 4.3.3, Paragraph „Modifikation der Klasse `TestTableViewController`", neunter Schritt). Nach Modifikation der Daten und anschließendem Zurückkehren aus der Detailansicht in die Tabellenansicht soll das modifizierte Datenobjekt zurück übermittelt werden.

   Als Wiedereintrittspunkt der modifizierten Daten in die Tabellenansicht dient eine neue Methode namens **`setEditedSelection:`**. Ihr einziger Parameter enthält die Daten, welche in

[18] Der *Edit*-Knopf kann alternativ auch vom Programmierer angelegt und mit eigens geschriebener Programmlogik versehen werden.

der „*Detail View*"-Szene modifiziert wurden. Die Modifikationen werden in das `dataArray`-Array und dann anschließend auf die Festplatte geschrieben. Wurde ein neuer Tabelleneintrag über die „*Add table-entry*"-Zelle in der ersten Tabellengruppe, wird ein neues `SomeData`-Instanzobjekt erstellt und im `dataArray`-Array sowie auf der Festplatte gespeichert.

Anmerkung: bei dieser Methode handelt es sich nicht um eine von *Apple* vorgegebene Methode, da *Apple* für *Segues* nur Methoden für den unidirektionalen Datentransfer vorsieht. Ein Manko, das ggf. in späteren Revisionen des *iOS-SDK* behoben wird.

```
1 -(void)setEditedSelection:(NSDictionary *)dict{
2   NSIndexPath *editedIndexPath = [dict objectForKey:@"indexPath"];
3   SomeData *editedData = [dict objectForKey:@"object"];
4   if(editedIndexPath.section == 0){
5     [self.dataArray addObject:editedData];
6     [self.tableView reloadData];
7   }else{
8     [self.dataArray replaceObjectAtIndex:editedIndexPath.row withObject:editedData];
9     [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:editedIndexPath]
          withRowAnimation:UITableViewRowAnimationAutomatic];
10  }
11  [self saveDataToDisk];
12 }
```

Tabelle 4.24.: Beim Verlassen der „*Detail View*"-Szene werden die modifizierten Daten an diese Methode zurückgeschickt, verarbeitet und schließlich auf der Festplatte gespeichert.

Modifikation der Klasse DetailViewController:

Um die Ergänzung der Klasse `TestTableViewController` um die Methode `setEditedSelection:` in der „*Detail View*"-Szene zu nutzen wird die Methode **`viewWillDissapear:`** eingeführt (siehe Unterkapitel 4.3.2) und um Anweisungen ergänzt die, die in der Detailansicht modifizierten Daten an die Tabellenansicht zurückzuliefern (siehe Codelisting 4.25).

Die Methode wird aufgerufen, kurz bevor die aktuell sichtbare Szene durch eine Andere verdrängt wird. Die modifizierten Daten werden beim Betätigen des Zurück-Knopfes an die `setEditedSelection:`-Methode geschickt. Das korrekte Empfängerobjekt wurde beim Aufruf des *Segues* in der `delegate`-Variable abgelegt (siehe Unterkapitel 4.3.3, Paragraph „Modifikation der Klasse `TestTableViewController`", neunter Schritt).

```
- (void) viewWillDisappear:(BOOL)animated
{
  [super viewWillDisappear:animated];

  if([self.delegate respondsToSelector:@selector(setEditedSelection:)]){
    [self.aTextField endEditing:YES];
    [self.aNumberField endEditing:YES];
    self.editedData.aString = self.aTextField.text;
    self.editedData.anInt = [self.aNumberField.text integerValue];

    NSIndexPath *indexPath = [self.selection objectForKey:@"indexPath"];
    NSDictionary *editedSelection = [NSDictionary dictionaryWithObjectsAndKeys:
        indexPath, @"indexPath", self.editedData, @"object", nil];
    [self.delegate setValue:editedSelection forKey:@"editedSelection"];
  }
}
```

Tabelle 4.25.: Beim Verlassen der *„Detail View"*-Szene werden die modifizierten Daten an die Tabellenansicht zurückgeschickt.

Ausführung der Beispielanwendung:

Wird die Anwendung ausgeführt und über das *Tab Bar Item „Lesson 3+4"* in die Tabellenansicht gewechselt, zeigen sich zwei Gruppen von Tabellenfeldern (siehe Abb. 4.52, linkes Teilbild). Die untere Gruppe ist bei der ersten Ausführung leer. Über die Zelle *„Add table-entry"* kann ein neuer Tabelleneintrag erstellt werden (siehe ebd., mittleres Teilbild). Beim Verlassen der *„Detail View"*-Szene erscheint der neue Eintrag in der Tabelle. Über den *Edit*-Knopf in der *Navigation Bar* können Tabelleneinträge neu angeordnet oder gelöscht werden (siehe ebd., rechtes Teilbild). Die Daten einzelner Zellen lassen sich durch Anklicken in der Detailansicht ändern.

Wird die Anwendung verlassen, oder in die *„Lesson 1"*- oder *„Lesson 2"*-Szene gewechselt und die Tabelle in irgendeiner Weise modifiziert, werden die Daten auf der Festplatte abgespeichert. Bei Rückkehr in die Tabellenansicht *„Lesson 3+4"* sind alle Einträge weiterhin vorhanden.

Abbildung 4.52.: Das Ergebnis der vierten Lektion im *iOS Simulator*.

4.3.5. Fünfter Schritt: Gestenerkennung

4.3.5.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- Architektur und Terminologie der Gestenerkennung,
- Erkennung von *Touches* und *Taps*,
- Erkennung eingebauter *iOS*-Gesten.

Da die hier beschriebenen Konzepte bezüglich der Theorie umfangreicher als die Themen der letzten Unterkapitel sind, wird ihr ein eigenes Unterkapitel gewidmet, bevor eine Beispielimplementierung erstellt wird.

4.3.5.2. Theorie

Nachdem der Benutzer die in den vorherigen Unterkapiteln vorgestellte Beispielanwendung bezüglich der Benutzungsoberfläche nur anhand der grafischen Elemente bedienen konnte wird in diesem Unterkapitel auf die Gestenerkennung eingegangen. Die Bildschirme der Apple-Mobilgeräte bieten durch die Möglichkeit mehrere unabhängige Berührungen gleichzeitig zu erkennen viel Interaktionspotential zwischen Anwender und der grafischer Benutzungsoberfläche. Die Möglichkeiten gehen weit über die Funktionalität reiner grafischen Benutzungsoberfläche die sich per Maus und Tastatur oder rudimentär per Stifteingabe bedienen lassen hinaus.

Im Zusammenhang mit der Erkennung von Berührungen und Gesten wird im *Cocoa Touch Framework* zwischen folgender Terminologie unterscheiden: (vgl. [Dav11], S. 604ff)

- Ein ***Touch*** bezieht sich auf eine Fingerberührung des Bildschirms, das Streichen mit einem Finger über den Schirm oder das Abheben des Fingers vom Display. Die Anzahl registrierter Touches entspricht dabei der Anzahl Finger die gleichzeitig mit dem Gerät interagieren.
- Ein ***Tap*** tritt auf wenn ein Finger nur kurz den Schirm berührt und die Schirmfläche sofort wieder verlässt. *iOS* bietet eingebaute Mechanismen, welche die Anzahl aufeinanderfolgender Taps registrieren. Nach Ablauf einer intern festgelegten Frist in der kein weiterer *Tap* registriert wird, wird ein Zähler, der die *Taps* verfolgt, zurückgesetzt.
- Eine Geste (engl. ***gesture***) beschreibt eine Ereignissequenz die den Zeitraum zwischen einer Berührung des Schirms (*Touch*) durch einen oder mehrere Finger und dem Moment in dem der letzte Finger den Schirm verlässt, umspannt. Die Geste gilt erst als beendet wenn **alle** Finger den Bildschirm verlassen. Eine Geste kann unterbrochen werden, wenn während ihrer Ausführung ein Systemereignis eintritt – z.Bsp. der Eingang eines Telefonanrufs.
- Ein ***Gesture Recognizer*** stellt ein spezielles Objekt dar, das die Interaktion des Benutzers mit der Benutzungsoberfläche kontinuierlich überwacht. Ein *Gesture Recognizer* kann auf die Erkennung bestimmter Gesten programmiert werden:
  - auf solche die bereits in *iOS* eingebaut sind, wie die Wischgesten, und/ oder

– auf programmatisch, durch den Anwendungsentwickler in Klassen, beschriebene Gesten.

Während viele *GUI*-Elemente bereits Programmlogik enthalten, um auf bestimmte *Touch*-Ereignisse zu reagieren – z.Bsp. ein `UIButton` der auf ein „*Touch Up Inside*" Ereignis reagiert – kann ein Gesture Recognizer vom Programmierer für jedes beliebige Element, auch für ganze Sichten, auf die Erkennung bestimmter Gesten angesetzt werden. Als Faustregel gilt, dass wenn eine Geste mehr als nur das berührte Objekt betrifft, die Programmlogik für die Bearbeitung dieses Ereignisses im *Controller*-Objekt der Sicht oder Szene implementiert wird, die dieses Objekt umfasst.

- Ein *Touch*-Ereignis (engl. ***event***) tritt auf wenn der Nutzer den Schirm berührt. Für jede Geste wird ein Ereignis-Objekt erstellt, das Informationen über den oder die *Touches*, die sich während der Geste ereigneten, aufweist. Bei der Auswertung des Ereignisses kommt das Modell der ***Responder Chain*** zu tragen.
- ***Responder*** und die ***Responder Chain***: für die Auswertung von Ereignissen – u.a. von *Touch*-Ereignissen – ist ein *Responder*-Objekt zuständig. Hierbei handelt es sich um ein Objekt einer beliebigen Subklasse der Klasse **`UIResponder`** – darunter fallen u.a. die Klasse `UIView` und dessen Subklassen wie `UIButton` aber auch die Klasse `UIViewController` und dessen Subklassen wie `UITableViewController` (siehe Abb. 4.3).

  Wird ein Ereignis ausgelöst wird dieses zuerst an den ***First Responder***, das Objekt mit dem der Nutzergerade interagiert, zur Auswertung gereicht. Der *First Responder* stellt den Anfang bzw. die **erste Ebene der *Responder Chain*** dar. Kann der aktuelle *Responder* nichts mit einem *Event* anfangen wird es die *Responder Chain* hochgereicht.

  Der Aufbau der Responder Chain orientiert sich an der Hierarchie der Sichten (vgl. hierzu 4.2.1.2). Nach dem *First Responder* wird ggf. dessen *View Controller* mit der Auswertung des Ereignisses beauftragt. Danach die dem *First Responder* übergeordnete Sicht und dann ggf. dessen *View Controller*. Das geht so lange die Hierarchie nach oben weiter, bis das Instanzobjekt welches das Anwendungsfenster repräsentiert mit der Auswertung des Ereignisses beauftragt wird – ein Instanzobjekt der Klasse `UIWindow`. Ist auch dieses nicht für das Ereignis zuständig wird es an das `UIApplication`-Instanzobjekt weitergeleitet und schlussendlich an das `AppDelegate`-Instanzobjekt. Reagiert keines dieser Objekte auf ein Ereignis, wird es verworfen.

  Wertet ein Objekt ein Ereignis erfolgreich aus wird es normalerweise nicht weiter in der *Responder Chain* hochgereicht. Ist nach Meinung des Programmierers die Auswertung eines Ereignisses durch einen *Responder* nicht vollständig möglich, muss es, nach einer teilweisen Abarbeitung, manuell weitergegeben werden. Den übergeordneten *Responder* kann der Programmierer über die Property `nextResponder` eines beliebigen `UIResponder`-Objekts bestimmen.

  Ein *Responder* wird über ein *Touch*-Ereignis durch eine von vier Methoden informiert:

  1. Die Methode **`touchesBegan:withEvent:`** wird aufgerufen wenn eine Geste anfängt.
  2. Die Methode **`touchesMoved:withEvent:`** wird aufgerufen wenn ein oder mehrere Finger sich auf dem Schirm bzw. einem bestimmten Schirmbereich oder Element bewegen. Diese Methode wird während einer Geste mehrfach aufgerufen.

3. Die Methode **`touchesEnded:withEvent:`** wird aufgerufen wenn alle Finger den Schirm verlassen. Die aktuelle Geste wurde dann erfolgreich beendet.
4. Die Methode **`touchesCancelled:withEvent:`** wird aufgerufen wenn ein Systemereignis ein *Touch*-Ereignis unterbricht. Zum Beispiel wenn ein Telefonanruf oder eine Warnung zu einem niedrigen Hauptspeicherstand eingeht. Wird diese Methode durch ein solches Systemereignis aufgerufen, wird die Methode `touchesEnded:withEvent:` nicht aufgerufen.

In ihren Standardimplementierungen beinhalten die vier soeben vorgestellten Methoden keine Programmlogik. In manchen grafischen Elemente hat *Apple* die Methoden aber bereits mit geeigneter Programmlogik ausgestattet. Werden diese Methoden durch einen Anwendungsentwickler in einer Klasse eingesetzt, muss er die Programmlogik selber erstellen. Dabei helfen ihm zwei Parameter bei der Auswertung der Geste:

1. Der Parameter **`touches`** beschreibt einen Satz `UITouch`-Ereignisse, welche die Schirmberührungen durch jeweils einen Finger repräsentieren. Sind mehrere Finger in der Ausführung einer Geste involviert, enthält dieser Parameter mehrere Elemente.

   Aus den `UITouch`-Elementen dieses Parameters kann mit der Variable `tapCount` auf die Anzahl *Taps*, die ein Finger ausführt, geschlossen werden.

   Darüber hinaus lässt sich u.a. auch die Position jedes einzelnen Fingers, der an der Ausführung einer Geste beteiligt ist, ermitteln.
2. Der Parameter **`event`** stellt das Instanzobjekt dar zu dem die *Touches*-Ereignisse gehören.

4.3.5.3. Implementierung

Zum Zweck der Vorführung wird die in den letzten Unterkapiteln erstellte Beispielanwendung ergänzt. Zuerst wird in der Datei `MainStoryboard.storyboard` ein neuer *View Controller* eingefügt. In diese neue Szene werden anschließend acht *Labels* platziert (siehe Abb. 4.53):

- Vier `UILabels` werden auf der linken Seite der Szene untereinander platziert. Sie enthalten die Texte „*Detected Event:*", „*# of Touches:*", „*# of Taps:*" bzw. „*Swipe Detected:*".
- Auf der rechten Seite der Szene werden vier weitere *Labels* platziert. Jeweils auf der Höhe eines der vorigen vier *Labels*. Sie erhalten keinen Inhalt. Da sie später längere Zeichenketten enthalten werden, sollten sie zudem etwas breiter gestaltet werden.

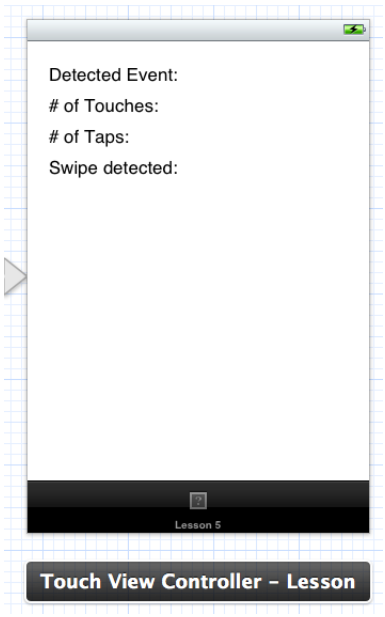

Abbildung 4.53.: Die neue Szene im *Storyboard*.

Als nächstes wird eine neue *Objective-C*-Klasse namens `TouchViewController` mit der Oberklasse `UIViewController` erstellt. Dabei werden die unteren Felder auf der zweiten Seite des Erstellungsdialogs nicht angekreuzt.

Anschließend wird diese neu erstellte Klasse im *Storyboard* dem neu eingeführten *View Controller* über den *Identity Inspector* zugewiesen.

In der Klassenerweiterung (siehe Unterkapitel 3.2.5.2) der Klassenimplementierung `TouchView-Controller.m` wird für jedes der vier Labels, auf der rechten Seite in der Szene, ein *IBOutlet* erstellt.

Danach werden die folgenden fünf Methoden an das Ende der Datei unter eine Methodengruppierung namens „*Touch event handling*" eingefügt: (siehe Codelisting 4.26)

1. Die erste Methode namens **`updateLabelsFromTouches:`** wird von den vier, oben vorgestellten, *Touch*-Methoden aufgerufen um die Inhalte der Labels mit der Anzahl der detektierter *Touches* und *Taps* zu füllen.
2. Als nächstes werden die **vier** oben beschriebenen ***Touch*-Methoden** implementiert. Sie aktualisieren das oberste Label mit dem aktuell eingetretenen *Event*-Typ und rufen anschließend die `updateLabelsFromTouches:`-Methode auf.

```
#pragma mark - Touch event handling

- (void) updateLabelsFromTouches:(NSSet *)touches{
  touchLabel.text = [NSString stringWithFormat:@"%d", [touches count]];
  tapLabel.text = [NSString stringWithFormat:@"%d", [[touches anyObject] tapCount]];
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
  eventLabel.text = @"Touches began";
  [self updateLabelsFromTouches:touches];
}

- (void) touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
  eventLabel.text = @"Drag detected";
  [self updateLabelsFromTouches:touches];
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
  eventLabel.text = @"Touches ended";
  [self updateLabelsFromTouches:touches];
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event{
  eventLabel.text = @"Touches cancelled";
  [self updateLabelsFromTouches:touches];
}
```

Tabelle 4.26.: Implementierung der *Touch*-Methoden.

Wird die Anwendung jetzt ausgeführt[19] und wird in die neue Sicht gewechselt, kann der Anwender sich die Anzahl *Touches* und *Taps* anzeigen, die er durch Gesten bewirkt[20]. Auffallend ist, dass die Anzahl *Taps* stimmt, aber die Anzahl *Touches* auf maximal eins begrenzt ist. Um dieses Manko zu Beheben muss das *Storyboard* erneut aufgerufen werden. Im *Interface Builder* muss anschließend das *View*-Objekt der neuen Szene markiert werden – das ist die Sicht innerhalb des *Controllers*, also die (aktuell) weiße Fläche um die Labels herum. Abschließend muss im *Attributes Inspector* die Option ***Multiple Touch*** aktiviert werden.

Als nächstes wird die automatische Erkennung einer, bereits im *iOS-SDK* vordefinierten, Geste implementiert . Hierzu wird in der Methode **`viewDidLoad`** ein neues „*Gesture Recognizer*"-Objekt erstellt, für das festlegt wird, dass es nur auf horizontale Wischgesten von links nach rechts und rechts nach links reagieren soll (siehe Codelisting 4.27). Bei Erkennung einer solchen Geste wird eine neu implementierte Methode **`reactToHorizontalSwipe`** aufgerufen, die das unterste, rechte *Label* in der Szene aktualisiert. Nach anderthalb Sekunden wird der Inhalt dieses Labels wieder gelöscht, ansonsten würde das Label zur Ausführungszeit ständig einen unveränderten Inhalt anzeigen, nachdem eine Wischgeste erkannt wurde.

[19] Obwohl die in diesem Unterkapitel vorgestellten Berührungen alle im *iOS Simulator* durchgeführt werden können ist anzumerken, dass nicht alle Gesten im iOS Simulator erfolgreich ausgeführt und/ oder erkannt werden können. Das Testen sollte daher auf einem *Apple*-Mobilgerät stattfinden.

[20] Im *iOS Simulator* können Zweifinger-Gesten durch drücken der *ALT*-Taste simuliert werden.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  // Do any additional setup after loading the view.
  UISwipeGestureRecognizer *horizontalSwipeRecognizer = [[UISwipeGestureRecognizer
      alloc] initWithTarget:self action:@selector(reactToHorizontalSwipe)];
  horizontalSwipeRecognizer.direction = UISwipeGestureRecognizerDirectionLeft |
      UISwipeGestureRecognizerDirectionRight;
  [self.view addGestureRecognizer:horizontalSwipeRecognizer];
}

...

-(void) reactToHorizontalSwipe{
  swipeLabel.text = @"Horizontal Swipe Detected";
  [self performSelector:@selector(eraseSwipeLabelContent) withObject:nil afterDelay
      :1.5];
}

-(void) eraseSwipeLabelContent{
  swipeLabel.text = @"";
}
```

Tabelle 4.27.: Implementierung eines neuen „*Gesture Recognizer*"-Objekts und der damit verbundenen Methoden.

Wird die Anwendung erneut ausgeführt wird bei Erkennung einer Wischgeste das untere *Label* korrekt aktualisiert und nach anderthalb Sekunden wieder geleert (siehe Abb. 4.54). Zudem werden die Anzahl tatsächlich, registrierter *Touches* jetzt korrekt angezeigt.

Abbildung 4.54.: Das Ergebnis der fünften Lektion im *iOS* Simulator.

4.3.6. Sechster Schritt: Gyroskop & Beschleunigungssensor

4.3.6.1. Übersicht

In diesem Unterkapitel werden die folgende Themen behandelt:

- Benutzung des Beschleunigungssensors,
- Benutzung des Gyroskops.

4.3.6.2. Theorie

Apple-Mobilgeräte verfügen neben den Sensoren zur Erkennung von Bildschirmberührungen noch etliche weitere Sensoren. Für die Erkennung von Bewegungen des Geräts und die damit u.a. verbundene automatische Rotation des Bildschirminhalts oder die Nutzung als Spielsteuerung, stehen folgende Sensoren zur Verfügung:

- Der eingebaute Beschleunigungssensor misst Beschleunigungen auf drei Achsen (siehe Abb. 4.55). Die Maßeinheit für gemessenen Werte sind G-Kräfte.

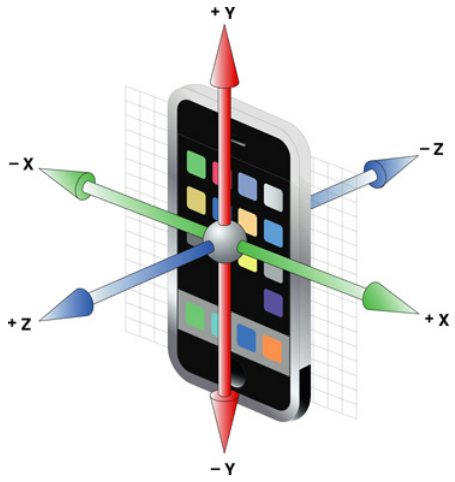

Abbildung 4.55.: Ausrichtung der Geräteachsen. Bildquelle: [App10b], S. 5

- Neuere Geräte[21] beinhalten zusätzlich ein Drei-Achsen-Gyroskop, das die Rotation des Geräts um die eigenen Achsen in Radianten pro Sekunde misst. Die Werte sind nicht absolut, sondern relativ zum letzten Zeitpunkt an dem der Wert abgefragt wurden. Die Werte werden kumuliert und bei Abfrage auf Null zurückgesetzt. Das Vorzeichen der gemessenen Werte folgt der Rechte-Hand-Regel: wird die rechte Hand um eine der drei Achsen so gelegt, dass der Daumen in die positive Richtung der Achse zeigt, entspricht eine Rotation mit positivem Vorzeichen auf dieser Achse einer Drehung in Richtung der vier, die Achse umgreifenden Finger.

[21] Ab dem *iPhone 4*, dem *iPad 2* und dem *iPod Touch 4*.

Dem Programmierer wird mit dem *Core Motion Framework* eine Zugriffsschicht auf die gemessenen Werte des Beschleunigungssensors und des Gyroskops an die Hand gegeben. Als Haupteinstiegspunkt gilt die Klasse `CMMotionManager`. Ein Instanzobjekt dieser Klasse kann in zwei Verhaltensmodi eingestellt werden:

1. In einem Modus liefert das `CMMotionManager`-Instanzobjekt kontinuierlich Daten an die Anwendung. Bei jedem Update, welches der Anwendung neue Werte zuführt, können spezielle, vom Programmierer vorgegebene, Anweisungen ausgeführt werden. Statt bei einem Update eine Methode in einem *Delegate*-Objekt aufzurufen, wie dies an vielen anderen Stellen im *iOS-SDK* Gang und Gäbe ist, wird hier das, seit *iOS 4* neu eingeführte, Konzept der **Blöcke**, für die Ausführung von Anweisungen, benutzt.

   Bei Blöcken handelt es sich um **anonyme Methoden** welche keinen Namen aufweisen, stattdessen werden sie durch ein '^'-Symbol repräsentiert. Hinter diesem Symbol werden in runden Klammern optionale Parameter in einer Komma-separierten Liste angegeben – der Aufbau der Methodensignatur orientiert sich an dem aus *C*-Programmen. Blöcke erlauben den Code welcher die Funktion aufruft und die Implementierung dieser Funktion zusammenhängend an einer Stelle anzugeben – dies entspricht der Einhaltung des **Lokalitätsprinzips** (vgl. [Ebe07]). Bei Verwendung eines *Delegates* würden zusammengehörige Anweisungen auf mehrere Dateien verteilt werden, obwohl sie ein gemeinsames Ziel verfolgen. Blöcke können innerhalb beliebiger *Objective-C*-Methoden angegeben werden.

2. Im anderen Modus werden die Werte nicht automatisch `CMMotionManager`-Objekt an die Anwendung geliefert. Stattdessen fragt der Programmierer sie manuell ab.

In beiden Modi werden die Werte zwischen zwei Abrufen im `CMMotionManager`-Objekt kumuliert. Bei Abruf werden sie zurückgesetzt.

Die Sensorwerte werden in zwei Datenstrukturen in einem `CMMotionManager`-Objekt gespeichert und sind über folgende *Properties* zugänglich:

1. Die Struktur `CMAccelerometerData` enthält die kontinuierlich aktualisierten Beschleunigungssensorwerte jeder der drei Achsen, während
2. die `CMGyroData`-Strutkur die kontinuierlich aktualisierten Werte des Gyroskops auf allen drei Achsen enthält.

4.3.6.3. Implementierung

Zur Veranschaulichung der Theorie soll die in den letzten Unterkapiteln erstellte Beispielanwendung ergänzt werden. Eine neue Szene soll die kontinuierlich gemessenen Beschleunigungs- und Rotationswerte des Geräts während der Laufzeit anzeigen. Außerdem wird das Block-Konzept vorgestellt, indem ein `CMMotionManager`-Objekt so eingestellt wird, dass es bestimmte Anweisungen automatisch ausführt wenn die Werte aktualisiert werden.

Zuerst wird im *MainStoryboard* ein neuer *View Controller* eingefügt und mit dem *Tab Bar Controller* verbunden. Der neuen Szene werden zwei, die ganze Breite und jeweils die halbe Höhe des Bildschirms, umspannende *Labels* eingefügt (siehe Abb. 4.56, linkes Teilbild). Diese sollen zur Laufzeit

die gemessenen Sensorwerte auf allen Achsen anzeigen. Weil diese Angaben mehrere Zeilen umspannen werden, wird für beide *Labels* individuell im *Attributes Inspector* der Parameter ***Lines* auf** den Wert **Null** gesetzt (siehe ebd., rechtes Teilbild). Er gibt die maximale Anzahl an Zeilen an, auf welchen ein *Label* seinen Inhalt verteilen kann. Wird der Wert auf Null gesetzt, wird die Zeilenanzahl variabel an den Text angepasst. Obwohl die Schriftgröße bei Erreichen der Höhenbegrenzung des Labels automatisch verringert wird, muss der Programmierer für eine angemessene Höhe sorgen, wenn der vollständige Schriftzug angezeigt werden soll. Zusätzlich wird für die Labels im *Attributes Inspector* unter *Alignment* eingestellt, dass sie ihre Inhalte zentriert anzeigen.

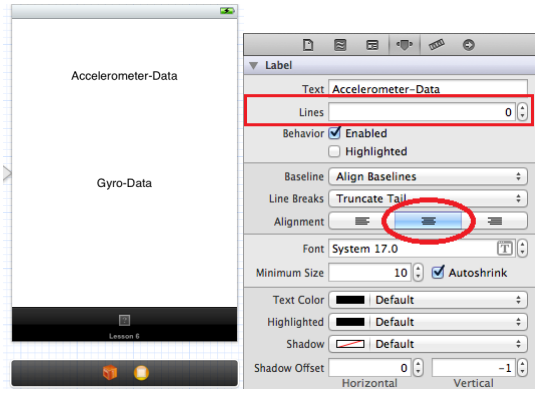

Abbildung 4.56.: Links: die neue Szene im *Storyboard*. Rechts: Einstellung der *Label*-Attribute im *Identity Inspector*.

Um den Zugriff auf die Sensorwerte im Projekt zu ermöglichen, muss das *Core Motion Framework* in das Projekt eingebunden werden. Hierzu wird das Wurzelobjekt im *Project Explorer* markiert und unter „*Target* > *Build Phases* > *Link Binary With Libraries*" über das Plus-Symbol das *Framework* ***CoreMotion.framework*** hinzugefügt (siehe Abb. 4.57).

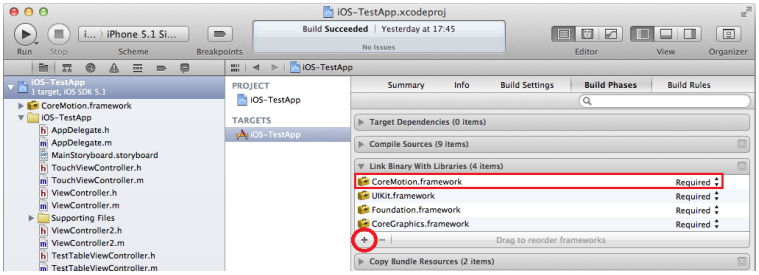

Abbildung 4.57.: Hinzufügen des *Core Motion Frameworks* in das Projekt

Als nächstes wird eine neue Klasse namens `SensorViewController`, eine Unterklasse von `UIViewController` erstellt – die zwei unteren Haken im zweiten Schritt im Erstellungsdialog werden nicht ausgewählt. Dann wird *View Controller* der neu erstellten Szene mit der gerade erstellten Klasse verbunden.

In der Deklaration der Klasse `SensorViewController` wird der Zugriff auf das *Core Motion Framework* mit der Importanweisung `#import <CoreMotion/CoreMotion.h>` sichergestellt. Dann wird in der Klassenerweiterung der Klassenimplementierung für jedes der beiden Labels in der neuen Szene jeweils ein *IBOutlet* erstellt. Zuletzt wird die Methode `viewDidLoad` um folgende Anweisungen ergänzt: (siehe Codelisting 4.28)

- Zuerst wird ein neues Instanzobjekt der Klasse `CMMotionManager` erstellt.
- Dann wird ein **neues** `NSOperationQueue`-Instanzobjekt erstellt, das die Ereignisse enthalten wird, welche ausgelöst werden wenn die gemessenen Werte des Beschleunigungssensors und des Gyroskops aktualisiert werden. Jede Anwendung besitzt grundsätzlich eine Hauptausführungsschleife in die die Ereignisse auch einfließen könnten. *Apple* rät allerdings ausdrücklich davon ab diese für Sensorereignisse zu nutzen, da falls sie vollläuft nicht mehr garantiert werden kann, dass kritischen Systemereignisse zuverlässig abgearbeitet werden können. Benötigt das System für die Abarbeitung von Systemereignissen länger als durch eine interne Karenz erlaubt, werden die Anwendungen die für diesen Umstand verantwortlich sind nachdrücklich terminiert – es drohen Datenverluste und Inkonsistenzen.
- Als nächstes wird durch eine geeignete Anweisung geprüft ob die Ausführungsplattform einen Beschleunigungssensor besitzt. Ist dies nicht der Fall wird ein entsprechender Hinweis im Label für die Beschleunigungswerte angegeben. Falls ein Beschleunigungssensor zur Verfügung steht wird dieser so eingestellt, dass die Daten jede Zehntelsekunde aktualisiert werden.

  Dann wird das `CMMotionManager`-Instanzobjekt angehalten bei Werteaktualisierungen automatisch die, im folgenden Block angegebenen, Anweisungen auszuführen. Darunter zählt der Zugriff auf die Werte jeder der drei überwachten Geräteachsen und die Aktualisierung des entsprechenden Labels in der Szene. Weil der *Motion Manager* die Ereignisse in einer anderen als der Hauptausführungsschleife abarbeitet und nach *Apple* der Zugriff auf grafische Elemente zur Thread-Sicherheit stets aus der Hauptausführungsschleife erfolgen soll, wird die Aktualisierung des *Label*-Inhalts im Haupt-Thread ausgeführt.
- Abschließend wird das selbe Prozedere wie für den Zugriff auf die Beschleunigungswerte für den Zugriff auf die Werte des Gyroskops angewendet.

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  // Do any additional setup after loading the view.
  CMMotionManager *motionManager = [[CMMotionManager alloc] init];
  NSOperationQueue *sensorEventQueue = [[NSOperationQueue alloc] init];

  if(!motionManager.isAccelerometerAvailable){
    self.acceleratorLabel.text = @"No accelerometer available!";
  } else {
    motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
    [motionManager startAccelerometerUpdatesToQueue:sensorEventQueue withHandler:^(
        CMAccelerometerData *data, NSError *error){
        NSString *accLabelText;
        if(!error) accLabelText = [NSString stringWithFormat:@"Accelerometer:\nx:
            %+.2f\ny: %+.2f\nz: %+.2f", data.acceleration.x, data.acceleration.y,
            data.acceleration.z];
        else {
          [motionManager stopAccelerometerUpdates];
          accLabelText = @"Accelerometer error!";
        }
        [acceleratorLabel performSelectorOnMainThread:@selector(setText:) withObject
            :accLabelText waitUntilDone:NO];
    }];
  }

    if(!motionManager.isGyroAvailable) self.gyroLabel.text = @"No gyroscope
        available!";
  else {
    motionManager.gyroUpdateInterval = 1.0 / 10.0;
    [motionManager startGyroUpdatesToQueue:sensorEventQueue withHandler:^(CMGyroData
         *data, NSError *error){
        NSString *gyroLabelText;
        if(!error) gyroLabelText = [NSString stringWithFormat:@"Gyroscope:\nx: %+.2f
            \ny: %+.2f\nz: %+.2f", data.rotationRate.x, data.rotationRate.y, data.
            rotationRate.z];
        else {
          [motionManager stopGyroUpdates];
          gyroLabelText = @"Gyroscope error!";
        }
      [gyroLabel performSelectorOnMainThread:@selector(setText:) withObject:
          gyroLabelText waitUntilDone:NO];
    }];
  }
}
```

Tabelle 4.28.: Zugriff auf die Werte des Beschleunigungssensors und des Gyroskops.

Nach den oben beschriebenen Modifikationen werden die *Labels* zur Laufzeit, kontinuierlich mit aktuellen Sensorwerten aktualisiert (siehe Abb. 4.58, linkes Teilbild). **Anzumerken ist** allerdings, dass die hier beschriebenen Funktionalitäten **nicht im *iOS Simulator*** getestet werden können, da dieser die nötigen Sensoren nicht besitzt und geeignete Sensorwerte (bisher) nicht simulieren kann

(siehe ebd., rechtes Teilbild).

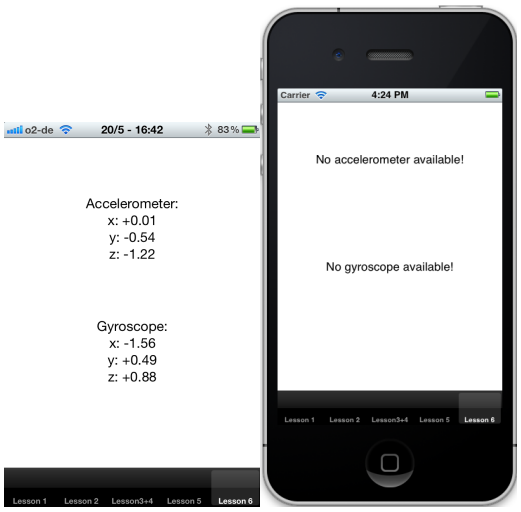

Abbildung 4.58.: Ergebnis der sechsten Lektion: links auf einem *iPhone 4* und rechts im *iOS Simulator*.

Teil IV.

Entwicklung einer OpenGL-ES Netzwerkanwendung für iOS

5. Vorstellung der iRoller2000-Applikation

5.1. Einleitung

In den vorigen Kapiteln wurden die Komponenten und Grundlagen zur Erstellung von Anwendungen für *iOS* vorgestellt. Die Basis bildet die objektorientierten Sprache *Objective-C* samt des dazugehörigen Compilers und der dazugehörigen Laufzeitumgebung (siehe Unterkapitel 3). Darauf aufsetzend steht Anwendungsentwicklern das *iOS-SDK Framework* zur Verfügung, das auf die Programmierung von *iOS*-Anwendungen zugeschnitten ist (siehe Unterkapitel 4).

Anhand dieser Komponenten wurde, im Rahmen dieser Diplomarbeit, eine grafische, netzwerkfähige Anwendung für ein *Head-mounted Display* auf Basis von *iOS*-Geräten entwickelt. In diesem Kapitel werden die **Funktion des Systems** und die **Anforderungen** an die Anwendung erläutert (siehe Unterkapitel 5.2). Anschließend werden die Bestandteile des Programms und das selbst entwickelte Netzwerkprotokoll, durch Zuhilfenahme von Diagrammen und Standaufnahmen der Anwendung, eingehend vorgestellt (siehe Unterkapitel 5.3).

5.2. Anforderungen

5.2.1. Funktion des Systems

Ziel des Projekts wird eine *iOS*-Anwendung sein, die auf mehreren *Apple*-Mobilgeräten zugleich läuft (**Client**) und deren Ausgabe durch ein weiteres Mobilgerät synchronisiert wird (**Server**). Dem Benutzer soll die Möglichkeit gegeben werden die Anwendung im Client- oder Server-Modus in Betrieb zu nehmen. Wenn zur Laufzeit alle Clients mit dem Server verbunden sind, soll eine virtuelle Achterbahnfahrt angezeigt werden. Zur besseren Orientierung des Nutzers in seiner Umgebung während des Tragens des *Head-mounted Display* (kurz *HMD*) soll, vor dem Start der Achterbahndarstellung und in einem Pause-Modus, das Livebild der Geräterückkamera angezeigt werden.

Der Server soll Kommandos zum Starten der Anwendung, zum Synchronisieren der Ausgabe, zum Pausieren, zum Stoppen und zur Vermittlung von Bewegungsinformationen an entsprechenden Geräten per Funkverbindung (typischerweise nach dem Standard *IEEE 802.11)* übertragen. Die Serveranwendung soll die Auswahl einer beliebigen, auf den Geräten abgelegten Achterbahnstrecke erlauben. Darüber hinaus soll das Erstellen einer beliebigen Anzahl, logischer Gerätegruppierung möglich sein, die sich an der physischen Gruppierung der Mobilgeräte in einem *HMD* orientieren. Je ein Gerät jeder logischen Gruppe wird die Bewegungen über das integrierte Gyroskop erfassen. Diese Informationen sollen, sofern ein Zweitgerät in der Gruppe enthalten ist, an

den Server übermittelt werden, der sie entsprechend weiterreichen soll. Die registrierten Bewegungen sollen der Anpassung des Betrachtungswinkels in der virtuellen Achterbahn dienen.

Fällt der Server während des Betriebs aus und sind Clients verbunden, so sollen diese versuchen die Verbindung erneut aufzubauen. Zugleich sollen sie das Livebild der Gerätekamera anzeigen. Klinkt sich ein Client aus irgend einem Grund, während er mit einem Server verbunden ist, aus, soll der Server die Darstellung der Achterbahn unterbrechen und dies den Clients mitteilen.

5.2.2. Randbedingungen

Die Anwendung soll für beliebige *iOS*-Geräte mit oder ohne *Retina Display* entwickelt werden. Bei der Erstellung der grafischen Oberfläche soll den größeren Abmaßen der *iPad*-Displays Rechnung getragen werden. Zudem soll darauf Rücksicht genommen werden, dass manche Apple-Mobilgeräte keine Kamera und/ oder kein Gyroskop besitzen. Für die Entwicklung soll die aktuell neuste Revision *5* des *iOS-SDK* verwendet werden um u.a. das Speichermanagement zu vereinfachen (siehe Unterkapitel 4.2.3).

Als Kernanwendung soll die Achterbahnsimulation *RollerCoaster2000* in der Version *2005* verwendet, welche das *Framework OpenGL ES* nutzt. Für dem Einsatz auf *iOS*-Geräten sollen einige Änderungen am Code des Achterbahnsimulators vorgenommen werden.

5.2.3. Versionen

Die Anwendungsentwicklung erfolgt anhand der folgenden Werkzeuge:

- Programmiersprachen: *Objective-C 2.0* und *C99*
- Compiler und Laufzeitumgebung: *Apple LLVM 3.1*
- *SDK*: *iOS-SDK 5.0.1*
- Entwicklungsumgebung: *Xcode 4.3.2*
- Betriebssystem: *Mac OS X 10.7*
- Testplattformen:
  - *iOS Simulator* mit *iOS 5.0.1*
  - *iPhone 3GS* mit *iOS 5.0.1*
  - *iPhone 4* mit *iOS 5.0.1*
  - *iPad 2* (*WiFi*) mit *iOS 5.0.1*
- *OpenGL-ES*-Anwendung: *RollerCoaster 2005* in einer durch den Autor angepassten und ergänzten Version für *iOS*

5.2.4. Anforderungsliste

Die folgenden Anforderungen wurden durch Gespräche mit dem Betreuer und durch das Durchspielen von Anwendungsszenarien erhoben. Die im Rahmen dieser Arbeit erstellte Anwendung *iRoller2000* genügt **allen**, im folgenden beschriebenen Anforderungen.

Die Reihenfolge der Anforderungen ist grob am Programmablauf orientiert.

5.2.4.1. Funktionale Anforderungen

1. Die Anwendung muss eine auf *Apple*-Mobilgeräten lauffähige Version des *RollerCoaster2000*-Projekts aufweisen.
2. Die Anwendung muss zwischen Client- und Serverbetrieb unterscheiden.
3. Dem Benutzer sollte die Möglichkeit gegeben werden vor dem Anwendungsstart den Client- oder Serverbetrieb festzulegen.
4. Der Benutzer sollte den Port für den Servertrieb einstellen können.
5. Der Benutzer sollte die IP-Adresse des Servers, mit dem sich der Client verbinden wird, einstellen können.
6. Der Benutzer sollte den Port des Servers, mit dem sich der Client verbinden wird, einstellen können.
7. Die Serveranwendung muss beim Start alle im *Documents*-Ordner (siehe Unterkapitel 4.3.4) der Anwendung enthaltene Achterbahnstreckenbeschreibungen laden.
8. Hat der Benutzer in der Serveranwendung logische Gerätegruppierungen angelegt, müssen diese beim Anwendungsstart von der Festplatte geladen werden.
9. Beendet der Benutzer die Serveranwendung, müssen angelegte Gerätegruppierungen auf der Festplatte gespeichert werden.
10. Sind in der Serveranwendung keine Gerätegruppierungen angelegt, muss dem Anwender ein entsprechender Hinweis angezeigt werden.
11. Ist in der Serveranwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, muss die Anzahl erwarteter Clients angezeigt werden.
12. Direkt nach dem Start der Client-Anwendung muss der Versuch gestartet werden eine Verbindung zum Server herzustellen.
13. Besitzt das Gerät auf dem die Client-Anwendung läuft eine Kamera, muss dem Benutzer beim Start ein Livebild der Kamera angezeigt werden.
14. Besitzt das Gerät auf dem die Client-Anwendung läuft **keine** Kamera, muss dem Benutzer eine schwarzes, leeres Bild angezeigt werden.
15. Beim Verbindungsversuch eines Clients zum Server, muss der Benutzer der Client-Anwendung über den Verbindungsstatus informiert werden.

16. Scheitert die Verbindung eines Clients zu einem nicht erreichbaren Server, muss ein neuer Verbindungsversuch alle drei Sekunden stattfinden.
17. Scheitert die Verbindung eines Clients zu einem nicht erreichbaren Server, muss der Benutzer darüber informiert werden.
18. Verbindet sich ein Client mit dem Server, muss eine ***HELLO*-Nachricht** an den Server gesendet werden.
19. Eine *HELLO*-Nachricht enthält die *IPv4*-Adresse des, sich zum Server verbindenden, Clients.
20. Ist in der Server-Anwendung keine Gerätegruppierung angelegt und verbindet sich ein Client, muss dem Client eine ***NOT-WELCOME*-Nachricht** zugestellt werden.
21. Ist in der Server-Anwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, muss bei der Verbindung eines Clients anhand dessen *IPv4*-Adresse geprüft werden ob eine Gerätegruppierung einen Client mit dieser IP-Adresse enthält.
22. Ist in der Serveranwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt und verbindet sich eines **autorisierter** Client, muss dem Client eine ***WELCOME*-Nachricht** zugestellt werden.
23. Eine *WELCOME*-Nachricht muss die Information enthalten ob das Gyroskop des Empfängers nach dem Start der Achterbahnsimulation aktiviert werden soll.
24. Eine *WELCOME*-Nachricht muss die Information über die horizontalen Verschiebung der Ausrichtung während der Achterbahnsimulation mitteilen.
25. Eine *WELCOME*-Nachricht muss die Information enthalten, ob der Empfänger der Nachricht alleine in einer Gerätegruppierung ist.
26. Eine *NOT-WELCOME*-Nachricht muss zur Folge haben, dass der Client erst nach fünf Sekunden versucht sich erneut mit dem Server zu verbinden – dies soll, falls erwünscht, dem Server-Betreiber die Zeit geben die IP-Adresse des Clients in den Anwendungseigenschaften einzugeben.
27. Eine *NOT-WELCOME*-Nachricht muss zur Folge haben, dass der Client den Benutzer darüber informiert dass dieser Client nicht auf dem Server autorisiert ist.
28. Ist in der Server-Anwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, verbindet sich eines autorisierter Client und sind damit nicht alle erwarteten Clients mit dem Server verbunden, muss die Anzahl verbundener Clients angezeigt werden.
29. Ist in der Server-Anwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, verbindet sich eines autorisierter Client und sind damit nicht alle erwarteten Clients mit dem Server verbunden, muss die Anzahl weiterhin erwarteter Clients angezeigt werden.
30. Ist in der Server-Anwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, verbindet sich eines autorisierter Client und sind damit alle erwarteten Clients mit dem Server verbunden, muss dem Anwender ein entsprechender Hinweis angezeigt werden.

31. Ist in der Serveranwendung keine Gerätegruppierung angelegt oder sind nicht alle Clients mit dem Server verbunden, muss der S*tart*-Knopf (siehe Nicht-funktionale Anforderungen) deaktiviert sein.
32. Ist in der Serveranwendung keine Gerätegruppierung angelegt oder sind nicht alle Clients mit dem Server verbunden, muss der *Pause*-Knopf (siehe Nicht-funktionale Anforderungen) deaktiviert sein.
33. Ist in der Serveranwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, verbindet sich eines autorisierter Client und sind damit alle erwarteten Clients mit dem Server verbunden, muss der *Start*-Knopf freigegeben/ aktiviert werden.
34. Ist in der Serveranwendung mindestens eine Gerätegruppierung mit mindestens einem Client angelegt, verbindet sich eines autorisierter Client und sind damit alle erwarteten Clients mit dem Server verbunden, muss der *Pause*-Knopf weiterhin deaktiviert bleiben.
35. Wird der aktivierte *Start*-Knopf gedrückt, muss der Server ein ***START*-Kommando** an alle verbundenen, autorisierten Clients schicken.
36. Eine *START*-Nachricht muss die Information enthalten welche Achterbahnstrecke dargestellt wird.
37. Sendet ein Server eine *START*-Nachricht, muss der obere Bereich der Anwendung, wo bisher Informationen zu den verbundenen Clients standen, durch eine Ansicht der Achterbahnsimulation ersetzt werden.
38. Nach dem Senden einer *START*-Nachricht und nachdem der obere Bereich der Server-Anwendung durch eine Ansicht für die Achterbahnsimulation ersetzt wurde, muss der Server seinen *OpenGL-ES*-Puffer initialisieren.
39. Nach dem Initialisieren seines *OpenGL-ES*-Puffers muss der Server die Achterbahnsimulation vor dem ersten Anzeigedurchlauf pausieren.
40. Erhält ein Client eine *START*-Nachricht muss das Bild der Gerätekamera durch eine Ansicht der Achterbahnsimulation ersetzt werden.
41. Erhält ein Client eine *START*-Nachricht und besitzt die Client-Anwendung in ihrem *Documents*-Ordner (siehe Unterkapitel 4.3.4) eine Beschreibungsdatei zur angegebenen Achterbahnstrecke muss diese eingelesen werden.
42. Erhält ein Client eine *START*-Nachricht und besitzt die Client-Anwendung in ihrem *Documents*-Ordner **keine** Beschreibungsdatei zur angegebenen Achterbahnstrecke muss eine fest im Quellcode kodierte Standardstrecke geladen werden.
43. Nach dem Laden der Achterbahnstreckeninformationen muss der Client seinen *OpenGL-ES*-Puffer initialisieren.
44. Nach der Initialisierung seines *OpenGL-ES*-Puffers und vor dem ersten Anzeigedurchlauf der Achterbahnsimulation muss die Achterbahnsimulation pausiert werden.
45. Nach der Initialisierung seines OpenGL-ES-Puffers und nach dem Pausieren der Achterbahnsimulation vor dem ersten Anzeigedurchlauf, muss der Client eine ***READY*-Nachricht** an den Server schicken – dies wird benötigt um die Ausgabe der Geräte synchron zu halten, da verschiedene Gerätemodelle unterschiedlich lang zur Initialisierung der Puffer brauchen.

46. Erhält ein Server eine *READY*-Nachricht eines autorisierten Clients und haben sich alle Clients entsprechend gemeldet, muss der Server eine ***GO*-Nachricht** an alle Clients senden.
47. Sendet ein Server eine *GO*-Nachricht, muss der Server die Darstellung der Fahrt in der Achterbahnsimulation wieder aufnehmen.
48. Erhält ein Client eine *GO*-Nachricht vom Server, muss er die Darstellung der Fahrt in der Achterbahnsimulation wieder aufnehmen.
49. Besitzt das Gerät auf dem die Client-Anwendung läuft ein Gyroskop und wurde in der *WELCOME*-Nachricht angegeben dass dieses aktiviert werden soll, müssen Gerätebewegungen anhand des Gyroskops erfasst werden.
50. Besitzt das Gerät auf dem die Client-Anwendung läuft ein Gyroskop, wurde in der *WELCOME*-Nachricht angegeben dass dieses aktiviert werden soll und ist dieses Gerät alleine in einer Gerätegruppierung, darf das Gerät registrierte Bewegungsinformation nicht an den Server senden.
51. Besitzt das Gerät auf dem die Client-Anwendung läuft ein Gyroskop, wurde in der *WELCOME*-Nachricht angegeben dass dieses aktiviert werden soll und ist dieses Gerät alleine in einer Gruppe, muss das Gerät die Bewegungsinformation direkt auswerten und die Ausrichtung in der virtuellen Achterbahnansicht anpassen.
52. Besitzt das Gerät auf dem die Client-Anwendung läuft ein Gyroskop, wurde in der *WELCOME*-Nachricht angegeben dass dieses aktiviert werden soll und ist dieses Gerät mit einem weiteren Mobilgerät in einer Gerätegruppierung, muss das Gerät die Bewegungsinformation via ***CLIENT-MOVE*-Nachrichten** an den Server senden.
53. Besitzt das Gerät auf dem die Client-Anwendung läuft ein Gyroskop, wurde in der *WELCOME*-Nachricht angegeben dass dieses aktiviert werden soll, ist dieses Gerät mit einem weiteren Mobilgerät in einer Gerätegruppierung und wurde die Bewegungsinformation an den Server gesendet, darf der Client die Ausrichtung in der virtuellen Achterbahnansicht **nicht** anpassen.
54. Eine *CLIENT-MOVE*-Nachricht muss die *IPv4*-Adresse des sendenden Clients enthalten.
55. Eine *CLIENT-MOVE*-Nachricht muss die relative Rollbewegung, seit der letzten Sendung einer *CLIENT-MOVE*-Nachricht, um die Y-Achse des Geräts in Radianten/Sek. enthalten.
56. Eine *CLIENT-MOVE*-Nachricht muss die relative Rollbewegung, seit dem letzten senden einer *CLIENT-MOVE*-Nachricht, um die X-Achse des Geräts in Radianten/Sek. enthalten.
57. Erhält der Server Bewegungsinformationen von einem Client, muss er die Bewegungsdaten, neu verpackt in einer ***SERVER-MOVE*-Nachricht**, an diesen Client zurücksenden – dies ist nötig um die Ausrichtungen zusammengehöriger Clients einer Gerätegruppierung synchron zu halten.
58. Erhält der Server Bewegungsinformationen von einem Client, muss er die Bewegungsdaten, neu verpackt in einer *SERVER-MOVE*-Nachricht, an den zweiten Client der Gruppe senden.
59. Eine *SERVER-MOVE*-Nachricht muss die Parameter zu den Rollbewegungen aus der *CLIENT-MOVE*-Nachricht übernehmen/ enthalten.

60. Erhält ein Client Bewegungsinformationen vom Server in Form einer *SERVER-MOVE*-Nachricht, muss er diese auswerten und die Ausrichtung in der virtuellen Achterbahnansicht anpassen.
61. Besitzt das Gerät auf dem die Server-Anwendung läuft ein Gyroskop, **kann** die virtuelle Achterbahnansicht auf dem Server an die Geräteausrichtung angepasst werden.
62. Der Benutzer der Server-Anwendung muss die Möglichkeit haben den Ablauf der Achterbahn jederzeit durch das drücken des *Pause*-Knopfes zu pausieren.
63. Der Benutzer der Server-Anwendung muss die Möglichkeit haben den Ablauf der Achterbahn jederzeit durch Berührung der Fläche in der die Achterbahnsimulation angezeigt wird, zu pausieren.
64. Löst der Benutzer der Server-Anwendung die *Pause*-Funktion aus, muss eine ***PAUSE*-Nachricht** an alle Clients gesendet werden.
65. Löst der Benutzer der Server-Anwendung die *Pause*-Funktion aus, muss die virtuelle Achterbahnfahrt in der Sever-Anwendung unterbrochen werden.
66. Erhält ein Client eine *PAUSE*-Nachricht, muss er seine virtuelle Achterbahnfahrt pausieren.
67. Erhält ein Client eine *PAUSE*-Nachricht und wurde die virtuelle Achterbahnfahrt pausiert, muss die Ansicht in die Kameraansicht wechseln, wo das Livebild der Gerätekamera angezeigt wird.
68. Wurde die Simulation pausiert und beendet der Server-Anwender die Pause indem er den *Pause*-Knopf erneut bedient, muss der Server erneut eine *PAUSE*-Nachricht an alle Clients schicken.
69. Wurde die Simulation pausiert und beendet der Server-Anwender die Pause indem er den *Pause*-Knopf erneut bedient, muss der Server die obere Ansicht, in der bisher Informationen zum Server-Status standen, durch die Darstellung der Achterbahn ersetzen.
70. Wurde die Pause beendet und zeigt der Server die Ansicht der Achterbahn wieder an, muss die Achterbahnfahrt wieder aufgenommen werden.
71. Wurde die Pause durch Empfang einer PAUSE-Nachricht beendet, muss ein Client die Kameraansicht durch die Achterbahnansicht ersetzen.
72. Wurde die Pause durch Empfang einer PAUSE-Nachricht beendet und wurde die Kameraansicht durch die Achterbahnansicht ersetzt, muss die Achterbahnfahrt wieder aufgenommen werden.
73. Wird der *Trackchang*e-Knopf in der Server-Anwendung gedrückt, muss allen verbundenen Clients eine ***STOP*-Nachricht** geschickt werden.
74. Wird der *Trackchang*e-Knopf in der Server-Anwendung gedrückt, muss die Server-Funktionalität auf neue Verbindungen zu reagieren gestoppt werden.
75. Wird die Trackchange-Ansicht angezeigt soll die aktuell markierte Achterbahnstrecke dargestellt werden.
76. Wird der *Config*-Knopf (siehe Nicht-funktionale Anforderungen) in der Server-Anwendung gedrückt, muss allen verbundenen Clients eine *STOP*-Nachricht geschickt werden.

77. Wird der *Config*-Knopf in der Server-Anwendung gedrückt, muss die Funktionalität auf neue Verbindungen zu reagieren gestoppt werden.
78. Wird die Serveranwendung geschlossen – der Benutzer drückt die *Home*-Taste – während autorisierte Clients verbunden sind, müssen diese anhand einer *STOP*-Nachricht darüber informiert werden, dass der Server die Funktionalität einstellt.
79. Eine *STOP*-Nachricht muss den Grund für die Unterbrechung enthalten.
80. Erhält ein Client eine *STOP*-Nachricht und wird die Achterbahnsimulation angezeigt, muss diese beendet werden.
81. Erhält ein Client eine *STOP*-Nachricht, lief die Achterbahnsimulation und wurde diese soeben beendet, muss in die Kameraansicht gewechselt werden wo das Livebild der Gerätekamera angezeigt wird.
82. Erhält ein Client eine *STOP*-Nachricht, muss er den Benutzer über den Grund der Unterbrechung informiert werden.
83. Erhält ein Client eine *STOP*-Nachricht, muss er jede drei Sekunden versuchen sich erneut mit dem Server zu verbinden.
84. Wird die Client-Anwendung geschlossen – der Benutzer drückt die *Home*-Taste – während der Client mit einem Server verbunden ist, muss der Client eine ***BYE*-Nachricht** an den Server senden.
85. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients, muss die Verbindung zu diesem Client aufgelöst werden.
86. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients und sind weitere Clients mit dem Server verbunden, müssen diese durch eine *STOP*-Nachricht informiert werden.
87. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients, muss die Serverfunktionalität zurückgesetzt werden – alle Verbindungen werden aufgelöst, der Server wartet auf die Neuverbindung aller Clients.
88. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients und wird die Achterbahnsimulation in der Server-Anwendung angezeigt, muss diese unterbrochen werden.
89. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients und wurde die Achterbahnsimulation unterbrochen, muss die Ansicht in eine Informationsansicht gewechselt werden – die selbe die auch vor dem Start und beim Pausieren angezeigt wird.
90. Erhält die Server-Anwendung eine *BYE*-Nachricht eines autorisierten Clients, muss der Benutzer in der Informationsansicht darüber informiert werden, dass sich ein Client sich abgemeldet hat.

5.2.4.2. Nicht-funktionale Anforderungen

1. Die Anwendungsklassen müssen nach ihrer Zugehörigkeit zu der Gruppe ***Model***, ***View*** oder **Controller** (siehe Unterkapitel 4.2.1.1) in einer entsprechenden *Xcode*-Gruppe abgelegt wer-

den.

2. Die Anwendungsklassen müssen nach ihrer Zugehörigkeit zu einer grafischen Szene der Anwendung, in einer entsprechenden *Xcode*-Gruppe abgelegt werden.
3. Alle Texte in der Anwendung müssen in Englisch verfasst sein.
4. Die Server-Anwendung muss eine ***Start*-Taste** besitzen.
5. Die Server-Anwendung muss eine ***Pause*-Taste** besitzen.
6. Die Server-Anwendung muss eine **Taste zum Wechseln der Achterbahnstrecke** besitzen.
7. Die Server-Anwendung muss eine **Taste zur Konfiguration der Gerätegruppierungen** besitzen.
8. Als Standard-Port für die Server-Anwendung, wenn nicht anders vom Benutzer in den Einstellungen definiert, muss der Port *1337* verwendet werden.
9. Als Port für die Server-Anwendung, wenn vom Benutzer in den Einstellungen definiert, muss der in den Einstellungen festgelegte Port sein.
10. Als Standard-Port für die Verbindung eines Clients zu einem Server muss der Port *1337* genutzt werden, wenn nicht anders vom Benutzer in den Einstellungen festgelegt.
11. Als Port für die Verbindung eines Clients zu einem Server muss der Port genutzt werden, welchen der Benutzer in den Einstellungen vorgegeben hat.
12. Der Standard-Wert für die horizontale Verschiebung der Ansicht in der Achterbahn für den Client, welcher im *HMD* vor dem **linken Auge** sitzt, soll **0.005** sein – dies entspricht einer leicht nach links versetzten, horizontalen Ausrichtung über dem Gleis der virtuellen Achterbahn.
13. Der Standard-Wert für die horizontale Verschiebung der Ansicht in der Achterbahn für den Client, welcher im *HMD* vor dem **rechten Auge** sitzt, soll **-0.005** – sein – dies entspricht einer leicht nach rechts versetzten, horizontalen Ausrichtung über dem Gleis der virtuellen Achterbahn.
14. Die Programmlogik für die Darstellung der grafischen Anwendung soll von der Logik für die Netzwerkfunktionalität abgekoppelt sein.
15. Das Repertoire an Netzwerknachrichten soll leicht ergänzbar sein.

5.2.4.3. Qualitätsanforderungen

1. Die Synchronizität unter allen Clients die mit einem gemeinsamen Server verbunden sind, muss stets gegeben sein.
2. Nach beenden einer Pause der Achterbahnsimulation muss die Achterbahnsimulation an der selben Stelle wie vor der Pause weiterlaufen.
3. Das Programm muss fehlerfrei beendet werden können.

4. Wird eine neue Gerätegruppierung in der Server-Anwendung angelegt, muss vor dem Speichern überprüft werden ob der Gruppenname und die Eingabefelder für das erste Gruppenmitglied ausgefüllt sind.
5. Wird eine Gerätegruppierung in der Server-Anwendung bearbeitet, muss vor dem Speichern überprüft werden ob der Gruppenname und die Eingabefelder für das erste Gruppenmitglied ausgefüllt sind.
6. Beim Anlegen oder Modifizieren einer Gerätegruppierung muss bei Eingaben in die Textfelder, welche die IP-Adresse der Clients enthalten, sichergestellt werden, dass nur solche Zeichen eingegeben werden können, die in IPv4-Adressen vorkommen.
7. Beim Anlegen oder Modifizieren einer logischen Gruppe muss bei Eingaben in die Textfelder, welche das X-Offset für den jeweiligen Client enthalten, sichergestellt werden, dass nur numerische Zeichen, das Vorzeichen '-' und das Dezimalzeichen '.' eingegeben werden können.
8. Bricht der Anwender das Anlegen einer neuen Gruppe ab, soll ein Hinweis angezeigt werden ob der Anwender dies wirklich wünscht.
9. Bricht der Anwender das Anlegen einer neuen Gruppe ab und bestätigt den Hinweis dass er wirklich Abbrechen möchte, muss auf die Speicherung einer neuen Gruppe verzichtet werden.
10. Bricht der Anwender das Anlegen einer neuen Gruppe ab und bricht den Hinweis dass er die neue Gruppe verwerfen will ab, soll er weiterhin Informationen in der neuen Gerätegruppierung eingeben können.
11. Bricht der Anwender das Editieren einer Gruppe ab und wurden Änderungen vorgenommen, soll ein Hinweis angezeigt werden der den Anwender über den Verlust getätigter Modifikationen informiert.
12. Bricht der Anwender das Editieren einer Gruppe ab und quittiert den Hinweis dass er die Änderungen verwerfen will, muss auf die Speicherung der modifizierten Daten verzichtet werden.
13. Bricht der Anwender das Editieren einer Gruppe ab und bricht den Hinweis dass er die Änderungen verwerfen will ab, soll er weiterhin Änderungen an der Gerätegruppierung vornehmen können.
14. Der gesamte Quellcode und alle Kommentare müssen in Englisch vorliegen.
15. Schwierigkeiten bei der Programmierung müssen sinnvoll dokumentiert werden.
16. Die Anwendung darf keine Kompilierungsfehler enthalten.
17. Die Anwendung sollte auf Client- und Server-Seite ein möglichst Überraschungsarmes Design vorweisen.

5.2.4.4. Technische Anforderungen

1. Das *iOS-SDK* in Version *5.0.1* muss bei der Entwicklung verwendet werden.

2. Bei der Entwicklung müssen nur durch *Apple* öffentlich zugänglich gemachte Funktionen des *iOS-SDK* benutzt werden.
3. Die Anwendung – Server wie Client – muss auf dem *iPhone 3GS* mit *iOS 5.0.1* lauffähig sein.
4. Die Anwendung – Server wie Client – muss auf dem *iPhone 4* mit *iOS 5.0.1* lauffähig sein.
5. Die Anwendung – Server wie Client – muss auf dem *iPhone 4S* mit *iOS 5.0.1* lauffähig sein.
6. Die Anwendung – Server wie Client – muss auf dem *iPod Touch 4G* mit *iOS 5.0.1* lauffähig sein.
7. Die Anwendung – Server wie Client – muss auf dem *iPad 2* mit *iOS 5.0.1* lauffähig sein.
8. Die Anwendung – Server wie Client – muss im *iOS Simulator* für das *iOS-SDK* Version *5.0.1* lauffähig sein.
9. Die Anwendung muss auf Geräten mit Gyroskop lauffähig sein.
10. Die Anwendung muss auf Geräten ohne Gyroskop lauffähig sein.
11. Die Anwendung muss auf Geräten mit Kamera lauffähig sein.
12. Die Anwendung muss auf Geräten ohne Kamera lauffähig sein.

5.3. Durchführung

Bei der Erstellung und Gestaltung der Anwendung wurde zuerst die *RollerCoaster2000* Version *2005* an die Bedürfnisse zur Ausführung auf *Apple*-Mobilgeräten angepasst. Außerdem wurde die Unterstützung zum Einlesen von Achterbahnstreckenbeschreibungen implementiert, welche im ursprünglichen *RollerCoaster2000*-Projekt aber nicht in der Version *2005* enthalten ist.

Anschließend wurde die *iRoller2000*-Applikation um das angepasste *RollerCoaster-2005*-Projekt erstellt. Die Gruppierung der Quellcodedateien in *Xcode* orientiert sich am *Model-View-Controller*-Konzept (siehe Unterkapitel 4.2.1.1). Die Gruppe *Model* enthält die Beschreibung der im Programm verwendeten Datenstruktur (siehe Unterkapitel 5.3.1). Die Gruppe *View* umfasst die Klassen welche die Benutzungsoberfläche beschreiben (siehe Unterkapitel 5.3.2). Die *Controller*-Gruppe enthält die rein programmatische, hinter den Kulissen ablaufende, Netzwerkfunktionalität (siehe Unterkapitel 5.3.3).

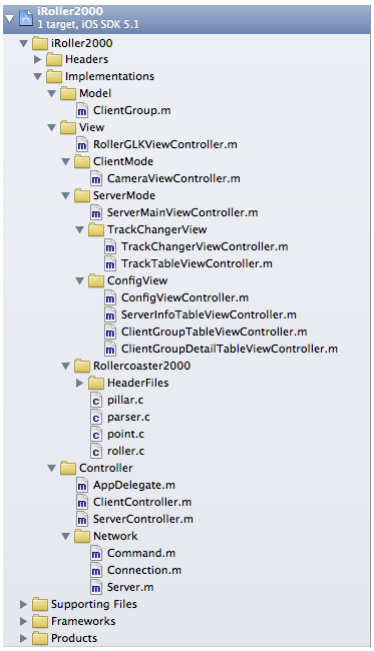

Abbildung 5.1.: 5.1Die Klassen der *iRoller2000*-Anwendung im *Xcode Project navigator*.

Die Anwendung verwendet bis auf an die eigenen Bedürfnisse angepasste Klassen aus dem *RollerCoaster2005*-Projekt und Netzwerkfunktionsvorführklassen von PETER BAKHYRYEV – beide Projekte genehmigen die Nutzung und Weitergabe ihres Codes – nur vom Autor verfasste Klassen. Die Programmlogik der eigens erstellten Klassen basiert auf den folgenden *Apple-Frameworks* (siehe Abb. 5.2):

- das ***AVFoundation*** *Framework* (*Audio/Video Foundation*) bietet Zugriff auf die Gerätekamera,
- das ***CFNetwork*** *Framework* (Core Foundation Network) wird für die Vernetzung von Client(s) und Server gebraucht,
- das ***CoreGraphics*** *Framework* ist für Operationen rund um die Verwendung von *OpenGL-ES* zuständig,
- das ***CoreMedia*** *Framework* dient der Verwaltung und Anzeige der Gerätekameraaufnahmen,
- das ***CoreMotion*** *Framework* wird den Zugriff auf das Gyroskop genutzt,
- das ***Foundation*** *Framework* bietet Basisklassen wie `NSObject` an,
- das ***GLKit*** *Framework* wird für Hilfsklassen und -funktionalität im *OpenGL-ES*-Bereich genutzt,
- das ***OpenGLES*** *Framework* (*Open Graphics Library for Embedded Systems*) dient der Darstellung der Achterbahnsimulation, und
- das ***UIKit*** *Framework* ist für die Basisfunktionalität rund um das *GUI* der Anwendung zuständig.

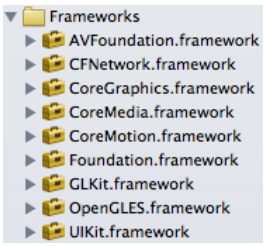

Abbildung 5.2.: Bei der Erstellung der *iRoller2000*-Applikation verwendete *Frameworks*.

Die Anwendung ist gemäß den Anforderungen in Client- und Server-Funktionalität unterteilt. Zur Bestimmung des Betriebsmodus kann der Anwender in den globalen Mobilgeräteeinstellungen unter dem Eintrag *iRoller2000* Änderungen vornehmen welche sich beim nächsten Programmstart auswirken (siehe Abb. 5.3). Gibt der Anwender eine IP-Adresse in das *Hostaddress*-Feld ein, startet die Anwendung im Client-Modus und verbindet sich mit dem Server mit der angegeben IP-Adresse. Ist dieses Feld beim Start der Anwendung leer, wird der Server-Modus aktiviert.

Abbildung 5.3.: Anwendungseinstellungen in der *Settings*-Applikation des Mobilgeräts.

Seit *iOS 4* laufen *iOS*-Applikationen nachdem sie in den Hintergrund treten, standardmäßig für eine gewisse Zeit weiter. Um Probleme bei der Synchronizität zwischen den Clients zu umgehen, wurde in den Projekteinstellungen festgelegt, dass die Anwendung beendet wird, wenn sie vom Anwender verlassen wird – z.Bsp. wenn dieser den *Home*-Taste am Mobilgerät betätigt. Zudem wurde vorgeschrieben, dass die Anwendung als Betriebssystem *iOS* ab Version *5* und einen *WLAN*-Adapter für die angebotene Funktionalität benötigt. (siehe Abb. 5.4)

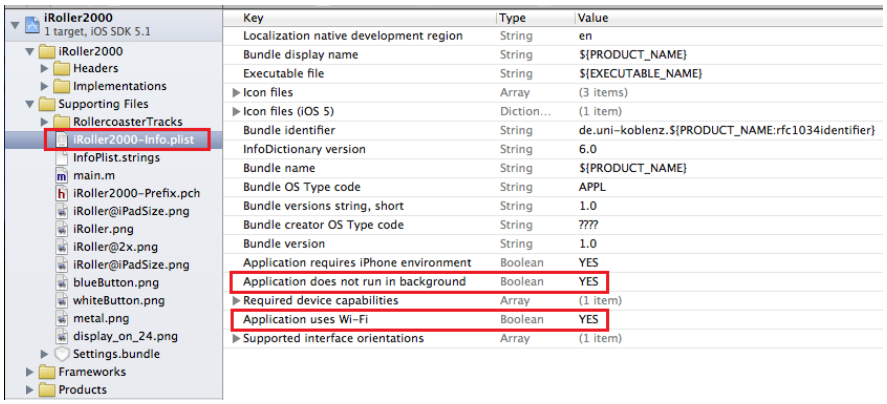

Abbildung 5.4.: Erweiterte Projekteinstellungen.

5.3.1. Model

Diese *Xcode*-Gruppe enthält eine einzige Klasse, welche die Datenstruktur einer `Clientgroup` sowie Funktionalität für die Datenpersistenz (siehe Unterkapitel 4.3.4) dieser Datensätze festlegt. Eine `Clientgroup` besteht aus einem oder zwei Mobilgeräten (Clients), welche in ein gemeinsames *HMD* eingesetzt werden.

Jede `Clientgroup` besitzt einen Namen (siehe Codelisting 5.1). Zudem wird für jedes Gerät einer Gruppierung eine IP-Adresse sowie die horizontale Verschiebung in der Achterbahnansicht gespeichert. Die horizontale Verschiebung in eine Richtung orientiert sich an der Distanz zwischen der Nase und Augenmitte des Betrachters. Diese Werte werden für die korrekte Anzeige eines stereoskopischen Bildes benötigt.

```
1 @interface ClientGroup : NSObject <NSCoding>
2 @property (nonatomic, copy) NSString *groupname;
3 @property (nonatomic, copy) NSString *client1Hostname;
4 @property (nonatomic) float client1XOffset;
5 @property (nonatomic, copy) NSString *client2Hostname;
6 @property (nonatomic) float client2XOffset;
7 @end
```

Tabelle 5.1.: Datenstruktur einer `Clientgroup`.

Beim Anwendungsstart im Server-Modus wird eine Liste aller vom Benutzer festgelegter *Clientgroups* von der Festplatte des Mobilgeräts geladen (siehe Unterkapitel 5.3.3). Eine neue Clientgroup kann in der Konfigurationsansicht angelegt und anschließend in einer Detailansicht mit Daten gefüllt werden (siehe Unterkapitel 5.3.2). Bereits angelegte *Clientgroups* lassen sich dort auch editieren, in ihrer Reihenfolge neu anordnen oder löschen. Vor Beendigung der Anwendung im Server-Modus, wird die Liste auf der Festplatte gespeichert.

5.3.2. View

Diese Kategorie enthält die Klassen welche das grafische Grundgerüst und die damit eng verbundene Logik der Anwendung beschreiben (siehe Abb. 5.5). Sie lassen sich aufgrund ihrer Zugehörigkeit zum Server- oder Client-Teil der Anwendung weiter unterteilen. Die einzige Ausnahme bildet die Klasse `RollerGLKViewController`, die sowohl bei der Anwendungsausführung im Server- wie auch Client-Modus genutzt wird. Zusätzlich gehören dieser Gruppierung alle Dateien des *RollerCoaster2000*-Projekts an.

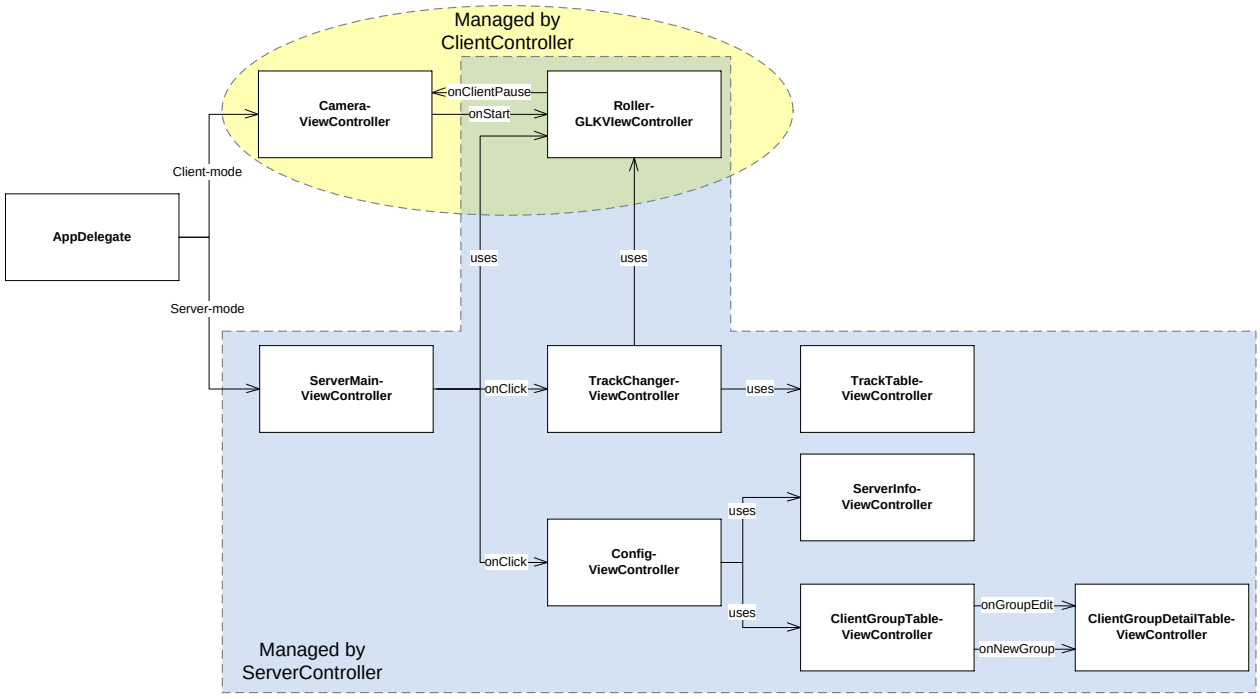

Abbildung 5.5.: Vereinfachte Übersicht aller Szenen der Anwendung.

5.3.2.1. Anwendungsszenen im Server-Modus

Start- und Hauptansicht:

Wird die Anwendung im Server-Modus gestartet, präsentiert sich dem Benutzer eine zweiteilige Ansicht, welche in der Klasse **`ServerMainViewController`** beschrieben wird (siehe Abb. 5.6). Der obere Bereich hält Informationen über den Serverstatus bereit. Der untere Bereich stellt dem Anwender die vier Hauptbedienelemente der Anwendung zur Verfügung:

- Der *Start*-Knopf führt zur Ausführung der virtuellen Achterbahnfahrt, die dann im oberen Teil der Hauptansicht angezeigt wird und die Informationsansicht ersetzt. Er kann erst betätigt werden, wenn sich alle Clients mit dem Server verbunden haben.

- Der *Pause*-Knopf führt zu einer Unterbrechung der Achtersimulation, deren Ansicht dann durch die vom Start bekannte Informationsansicht ersetzt wird. Ein erneutes betätigen der Taste während der Pause führt zur Wiederaufnahme der Achterbahnfahrt.
- Der *Trackchange*-Knopf führt den Anwender in eine neue Ansicht, in der er sich die auf dem Mobilgerät gespeicherten Achterbahnstrecken ansehen und eine für die Simulation aussuchen kann.
- Der *Config*-Knopf führt den Benutzer in eine neue Szene, in welcher er eine Liste der für die Aufnahme der Achterbahnsimulation erwarteten Mobilgeräte (Clients) verwalten kann.

Je nachdem ob die Applikation auf einem *iPhone* oder *iPad* abläuft, ändern sich die Positionen mancher Bedienelemente in der Anwendung. In der Hauptansicht werden die vier, eben vorgestellten Knöpfe auf dem *iPad* nebeneinander dargestellt.

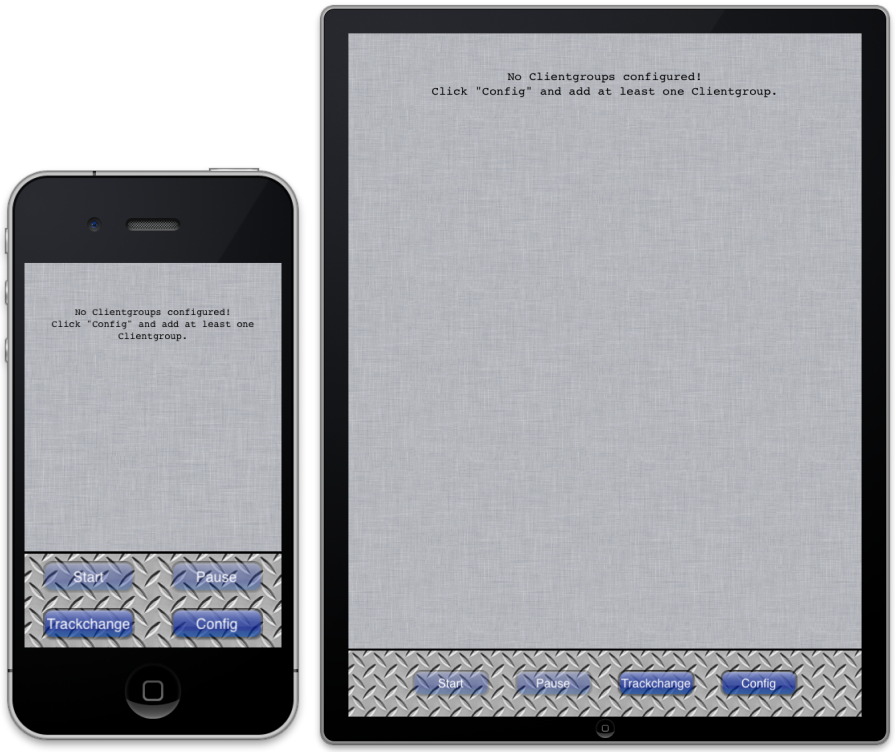

Abbildung 5.6.: Die Start- und Hauptansicht der Anwendung im Server-Modus. Links auf dem *iPhone* und rechts auf dem *iPad*.

Ansicht für die Auswahl der Achterbahnstrecke:

Wechselt der Anwender über den *Trackchange*-Knopf aus der Hauptansicht in die „*Track Selection*"-Szene (siehe Abb. 5.7) wird ihm eine Liste, auf der Festplatte des Mobilgeräts abgelegter, Achterbahnstrecken in einer Auswahlliste angezeigt. Im oberen Teil der Szene wird eine Strecken-

vorschau in der *RollerCoaster2000*-Engine angezeigt. Ein Klick auf den *Back*-Button markiert die zuletzt markierte Strecke für die nächste Achterbahnfahrt.

Die Szene wird in den Klassen `TrackChangerViewController` und `TrackTableViewController` beschrieben.

Abbildung 5.7.: Auswahl der Achterbahnstrecke.

Konfigurations- und Detailansicht:

Wechselt der Anwender über den *Config*-Knopf aus der Hauptansicht in die „*Configuration*"-Szene werden ihm Netzwerkinformationen zum Server sowie eine Liste aller vom Anwender angelegten *Clientgroups* angezeigt (siehe Abb. 5.8, linkes und rechtes Teilbild). Die Netzwerkinformationen dient der Vereinfachung bei der Eingabe der benötigten Verbindungsparameter auf den Client-Geräten.

Beim ersten Anwendungsstart ist die *Clientgroup*-Liste leer. Über das Tabellenfeld mit dem Bezeichner „*Add Group*" kann der Anwender die Liste um neue Einträge ergänzen. Ein Klick auf den *Edit*-Knopf in der *Navigation bar* (siehe Unterkapitel 4.3.3) ermöglicht die Neuanordnung und Löschung bereits angelegter *Clientgroups*.

Wird eine *Clientgroup* – eine logische Gruppierung von Mobilgeräten die zusammen in einem *HMD* eingesetzt werden – in der Tabelle angeklickt oder eine Neue angelegt, können ihre Erstellung in der Detailansicht bearbeitet werden (siehe Unterkapitel 5.8, mittleres Teilbild). Dem Nutzer steht es frei eine *Clientgroup* nach Belieben zu benennen (siehe mittleres Teilbild in Abb. 5.8). Bevor sie gespeichert werden kann muss der Nutzer zusätzlich zum Gruppennamen zumindest für ein

Gerät des *HMD* eine IP-Adresse und eine horizontale Verschiebung der Achterbahnansicht angeben. Die Felder für die Eingabe der IP-Adressen erlauben nur solche Zeichen, die in IPv4-Adressen in *quad-dotted* Notation vorkommen. Die *X-Offset*-Felder erlauben die Eingabe numerischer Zeichen, des im angelsächsischen Raum verwendeten Dezimaltrennzeichens '.' sowie der Symbole '+ und '-' als Vorzeichen.

Die Konfigurationsansicht wird in den Klassen `ConfigViewController`, `ServerInfoTableViewController` und `ClientGroupTableViewController` beschrieben. Die Detailansicht wird in der Klasse `ClientGroupDetailTableViewController` beschrieben.

Abbildung 5.8.: Die Verwaltung der *Clientgroups* erfolgt in der Konfigurationsansicht direkt in der Server-Anwendung.

Zusammenfassung des grafischen Anwendungsablaufs im Server-Modus:

Bei Eintritt in das Programm wird die links in Abb. 5.9 abgebildete Start- und Hauptansicht angezeigt. Sind alle Clients verbunden und drückt der Anwender die *Start*-Taste wird die Achterbahnsimulation angezeigt – oberes, linkes Teilbild. Ein Knopfdruck auf *Pause* stoppt die Achterbahn – oberes, rechtes Teilbild. Ein Klick auf *Unpause* führt die Simulation fort. Klickt der Anwender in der Hauptansicht auf den *Trackchange*-Button kann er die vorrätigen Achterbahnstrecken ansehen und eine für die Simulation aussuchen – mittleres Teilbild. Drückt der Benutzer den *Config*-Knopf gelangt er in eine Ansicht welche allgemeine Serverinformationen und eine editierbare Liste von *Clientgroups* bereitstellt – unteres, linkes Teilbild. Fügt der Anwender der Liste einen neuen Eintrag hinzu oder markiert er einen vorhandenen zur Editierung gelangt er in eine Detailansicht – unteres, rechtes Teilbild.

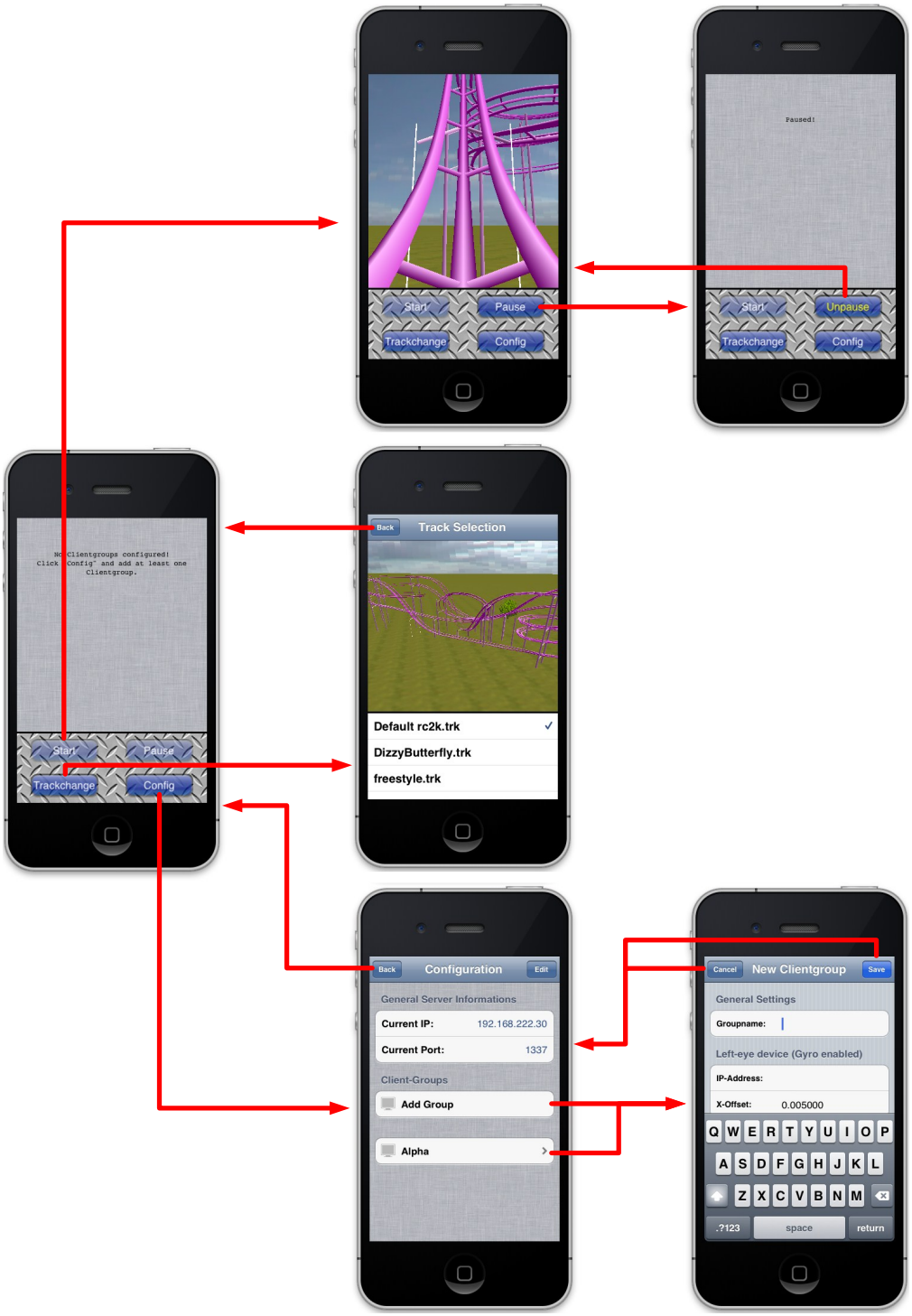

Abbildung 5.9.: Übersicht der Anwendung im Server-Modus.

5.3.2.2. Anwendungsszenen im Client-Modus

Kamera- und Statusansicht:

Wurde in den Programmeinstellungen vom Anwender eine IP-Adresse in das *Hostaddress*-Feld eingegeben startet die Anwendung im Client-Modus. Direkt nach Anwendungsstart werden neben einem Livebild der Gerätekamera auch Informationen über den aktuellen Verbindungszustand zum Server angezeigt (siehe Abb. 5.10). Der Client versucht sich ohne Zutun des Nutzers mit dem Server zu verbinden (siehe ebd., linkes Teilbild). Schlägt dies fehl wird der Anwender über den Grund informiert und ein neuer Verbindungsversuch wird gestartet (siehe ebd., mittleres Teilbild). War der Verbindungsversuch erfolgreich erscheinen die Verbindungsparameter die vom Server an den Client übermittelt wurden (siehe ebd., rechtes Teilbild). Besitzt die Plattform auf der die Anwendung ausgeführt keine Kamera, erscheint der Hintergrund der Szene schwarz (siehe ebd., linkes Teilbild).

Diese Szene wird in der Klasse `CameraViewController` beschrieben.

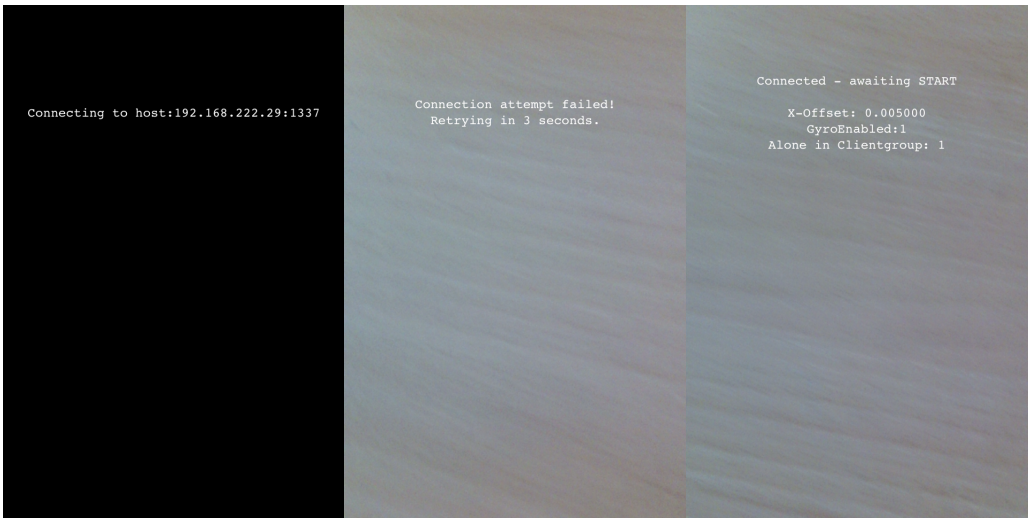

Abbildung 5.10.: Kameraansicht im Client-Modus nach dem Start der Anwendung. Links: Verbindungsaufbau nach dem Start. Mitte: Fehlgeschlagener Verbindungsversuch. Rechts: Erfolgreiche Verbindung.

Achterbahnansicht:

Startet der Anwender die Achterbahnsimulation durch Klick der *Start*-Taste auf dem Server, wechselt der Client in die Achterbahnansicht (siehe Abb. 5.11) – beschrieben in der Klasse `RollerGLK-ViewController`. Gemäß der in den Servereinstellungen für den Client hinterlegten horizontalen Verschiebung wird der Aufnahmewinkel der Achterbahnszene angepasst. Besitzt die Ausführungsplattform der Anwendung ein Gyroskop resultieren Bewegungen in Änderungen der Blickrichtung innerhalb der Achterbahnsimulation (siehe ebd., linkes und rechtes Teilbild). Falls kein

Gyroskop zur Verfügung steht, verweilt die Ausrichtung in der Achterbahn starr nach vorne gerichtet (siehe ebd., mittleres Teilbild).

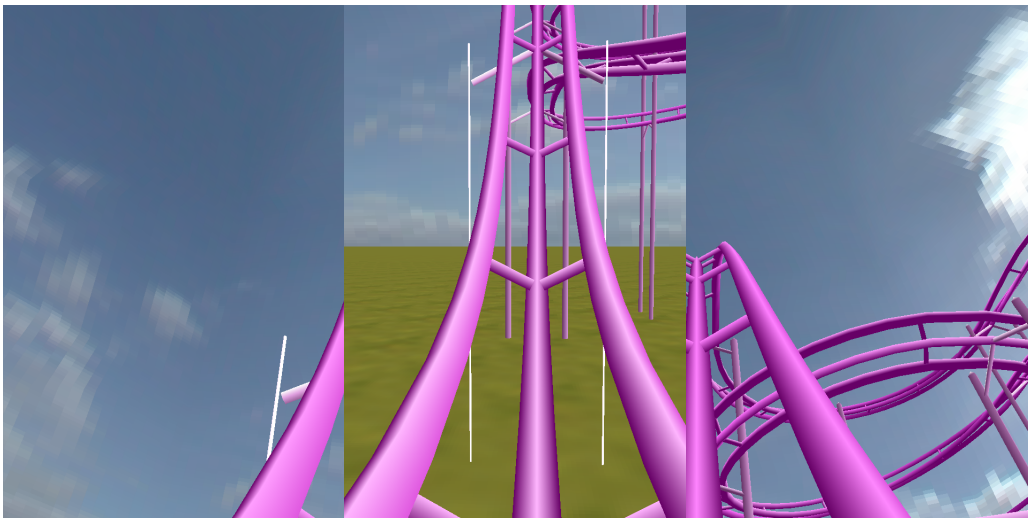

Abbildung 5.11.: Blick nach Links, Geradeaus und Rechts in der Achterbahnsimulation.

Sendet der Server eine *PAUSE*- oder *STOP*-Nachricht wird die Achterbahnsimulation pausiert und die Client-Anwendung wechselt zurück in die Kameraansicht. Dort wird der Benutzer über den Grund der Unterbrechung informiert (siehe Abb. 5.12).

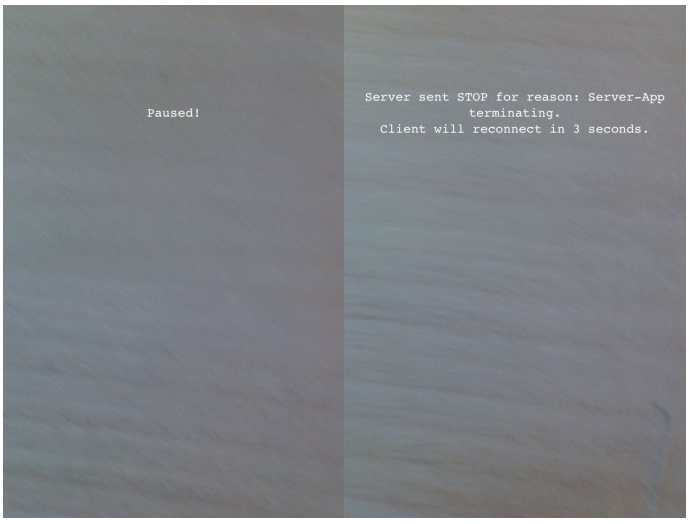

Abbildung 5.12.: Sendet der Server das Kommando zum Pausieren oder zum Stoppen wird wieder in die Kameraansicht gewechselt.

Sendet der Server eine weitere *PAUSE*-Nachricht nachdem die Achterbahnfahrt pausiert wurde, wird die Kameraansicht erneut durch die Achterbahnansicht ersetzt. Die Achterbahn fährt an der

Stelle fort wo sie angehalten wurde.

Sendete der Server eine *STOP*-Nachricht versucht der Server sich nach Ablauf einer Karenz erneut zu verbinden.

Grafische Zusammenfassung der Anwendung im Client-Modus:

Kurz nach dem Start der Anwendung im Client-Modus versucht der Client sich mit dem Server zu verbinden – linkes Teilbild in Abb. 5.13. Dabei wird im Hintergrund ein Livebild der Gerätekamera angezeigt, was dem Träger des *HMD* die Orientierung in seiner Umgebung und/ oder die Bedienung des Servers erleichtert.

Schlägt die Verbindung zum Server fehl, wird der Nutzer darüber informiert – zweites Teilbild von links. Ist die Verbindung erfolgreich werden dem *HMD*-Träger Informationen über die Verbindung zum Server und die Rolle des Clients angezeigt – drittes Teilbild von links. Startet der Server die Achterbahnfahrt wechselt der Client in die Achterbahnansicht – viertes Teilbild von links. Verfügt das Ausführungsgerät über ein Gyroskop oder erhält es vom Server Bewegungsinformationen kann der Blickwinkel in der Achterbahnsimulation verändert werden – siehe Teilbild über und unter dem vierten Teilbild.

Sendet der Server eine *PAUSE*-Nachricht wechselt der Client zurück in die Kameraansicht und informiert den Nutzer über den Status – fünftes Teilbild von links. Bei erneutem Empfang einer *PAUSE*-Nachricht wird die Achterbahnfahrt wiederaufgenommen.

Sendet der Server zu einem beliebigen Zeitpunkt eine *STOP*-Nachricht, weil z.Bsp. die Server-Anwendung beendet wird, so wird der Benutzer darüber informiert und der Client versucht die Verbindung neu aufzubauen – drei Teilbilder ganz rechts.

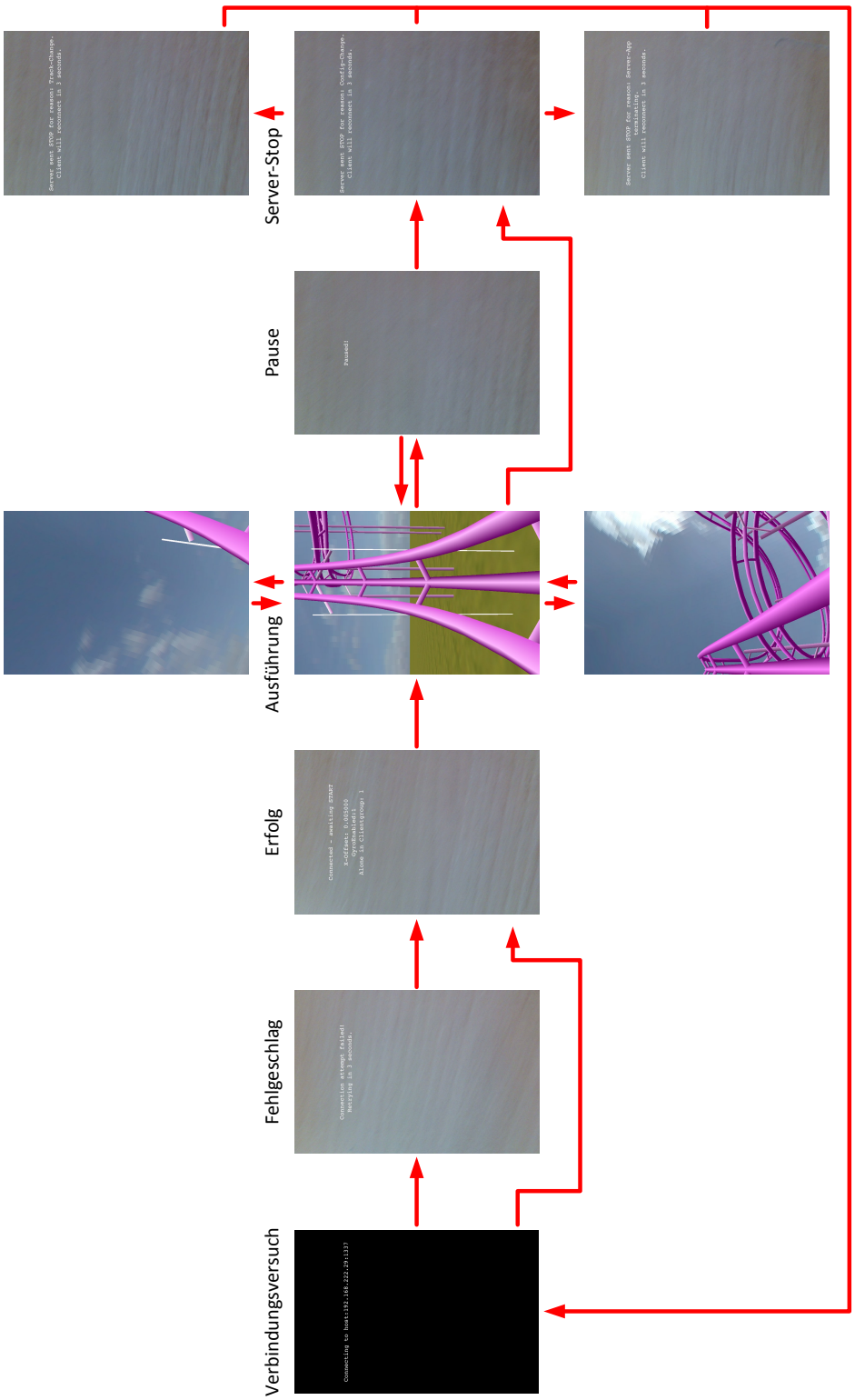

Abbildung 5.13.: Übersicht der Anwendung im Client-Modus.

5.3.3. Controller

Die *Controller*-Gruppierung enthält die im Hintergrund agierenden Anwendungsprozesse. Dazu gehören die Klasse `Command` (siehe Unterkapitel 5.3.3.1) und `AppDelegate` (siehe Unterkapitel 5.3.3.2) sowie die Programmlogik für den Server- (siehe Unterkapitel 5.3.3.3) und den Client-Modus (siehe Unterkapitel 5.3.3.4).

5.3.3.1. Command – Syntax der Netzwerkkommunikation

Die `Command`-Klasse legt die Nachrichtentypen in einer Aufzählung fest, die zwischen Client und Server ausgetauscht werden können (siehe Abb. 5.14). Zusätzlich gibt ein Array die Anzahl Parameter an, die jedem Nachrichtentypen zugeordnet sind (siehe Codelisting 5.2). Anhand dieser Aufzählung und der Festlegung der Parameteranzahl sortieren Server und Client nicht definierte oder missgebildete Nachrichten aus.

Die Bedeutungen der einzelnen Nachrichtentypen werden im Kontext der Beschreibung der Server- (siehe Unterkapitel 5.3.3.3) und Client-seitigen Abläufe (siehe Unterkapitel 5.3.3.4) erklärt.

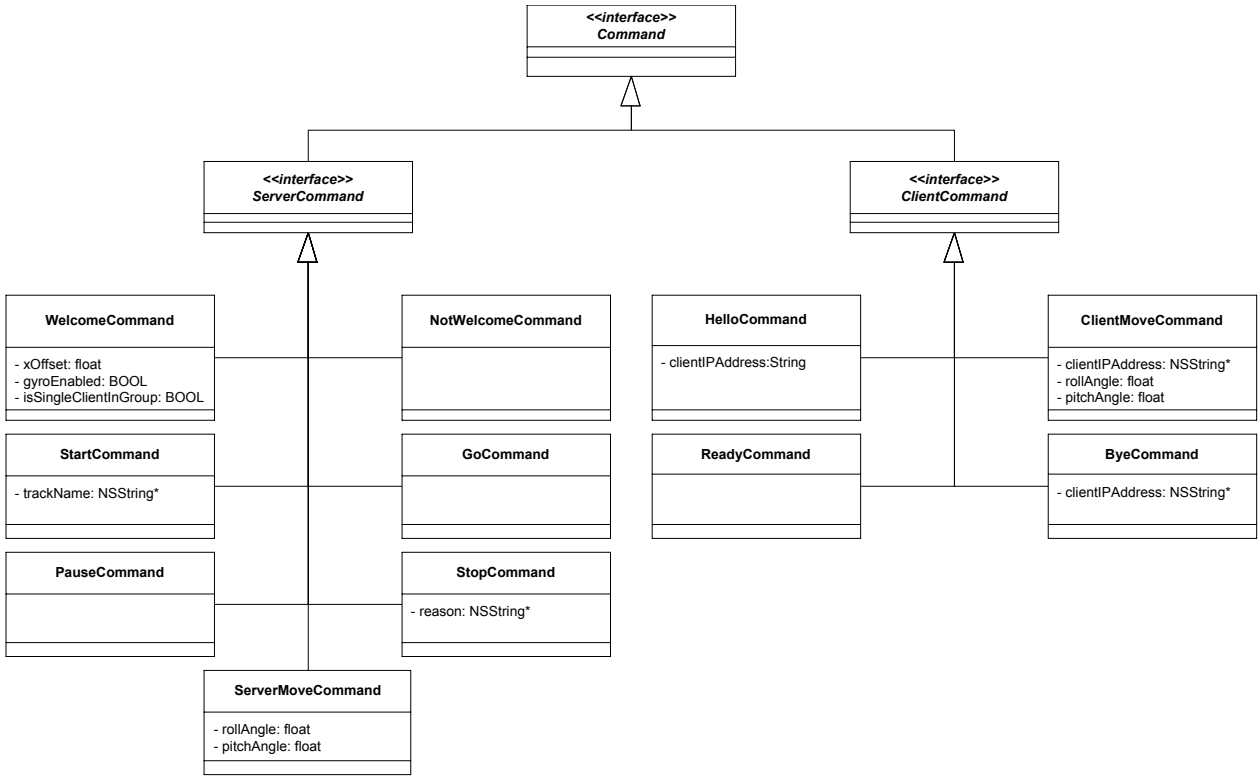

Abbildung 5.14.: Übersichtsdiagramm der Server- und Client-Nachrichtentypen.

```
const typedef enum{
  HELO, // Hello: sent by client on connection to server
  CMOV, // ClientMove: sent by client informing it of gyro-updates
  RDY,  // Ready: sent by client when OpenGL-ES buffers are initialized
  BYE,  // Bye: sent by client when the client-app is terminated
  WELC, // Welcome: sent by server to greet authorized clients
  NOTW, // NotWelcome: rejection sent by server to unauthorized clients
  STRT, // Start: sent by server to inform clients to start the coaster
  GO,   // Go: sent by server after client-buffer-sync was performed
  PAUS, // Pause: sent by server to pause/ unpause roller-execution
  STOP, // Stop: sent by server f.e. on server-app shutdown or config-changes
  SMOV  // ServerMove: sent by server to inform clients in a group of gyro-updates
} iROLLER_COMMAND;

static const short iROLLER_COMMAND_ARGUMENT_LENGTHS[11] = {
  1, //  0 - Hello :: "HELO" Client_IP-Address(=NSString*)
  3, //  1 - ClientMove :: "CMOV" Client_IP-Address(=NSString*) rollAngle(=float)
     pitchAngle(=float)
  0, //  2 - Ready :: "RDY"
  1, //  3 - Bye :: "BYE" Client_IP-Address(=NSString*)
  3, //  4 - Welcome ::= "WELC" xOffset(=float) gyroEnabled(=BOOL)
     isSingleClientInGroup(=BOOL)
  0, //  5 - NotWelcome ::= "NOTW"
  1, //  6 - Start :: "STRT" Trackname(=NSString*)
  0, //  7 - Go :: "GO"
  0, //  8 - Pause :: "PAUS"
  1, //  9 - Stop :: "STOP" reason(=NSString *)
  2  // 10 - ServerMove :: "SMOV" rollAngle(=float) pitchAngle(=float)
};
```

Tabelle 5.2.: In der Anwendung festgelegte Nachrichten und ihre Parameter(-anzahl).

5.3.3.2. App-Delegate – der Einstiegspunkt in die Anwendung

Den eigentlichen Einstieg in die *iRoller2000*-App, aus Sicht des Anwendungsprogrammierers, findet in der `application:didFinishLaunchingWithOptions:`-Methode der Klasse `AppDelegate` (siehe Unterkapitel 4.2.1.2). Hier wird die obere *Status bar* des Mobilgeräts ausgeblendet – wo u.a. die Uhrzeit angezeigt wird. Dann wird festgelegt, dass sich das Mobilgerät zur Anwendungslaufzeit nicht sperrt – was insbesondere dann für den Benutzer störend ist wenn die Applikation im Client-Modus läuft, das Gerät in einem *HMD* angebracht ist und der Benutzer das Gerät dadurch nur mit Mühe entsperren kann. Anschließend wird ermittelt ob die Ausführungsplattform ein *iPad* ist, in welchem Fall einige Vorkehrungen bei der Erstellung der grafischen Oberfläche getroffen werden. Abschließend werden die Programmeinstellungen, die der Benutzer in der *Settings*-App festlegen kann, ausgelesen und daran entschieden ob der Server- oder Client-Modus aktiviert wird.

Die `AppDelegate`-Klasse bildet zudem den Ausstiegspunkt der Anwendung. In der Methode `applicationWillTerminate:` wird geprüft ob die Anwendung im Server- oder Client-Modus lief. Danach werden die Server-Funktionalität heruntergefahren – die Clients erhalten eine *STOP*-

Nachricht (siehe Unterkapitel 5.3.3.3). Abschließend werden die Cliengroups auf der Festplatte abgespeichert. War der Client-Modus aktiv wird die Client-Funktionalität abgeschaltet bevor die Anwendung schließt – der Client sendet eine *BYE*-Nachricht an den Server (siehe Unterkapitel 5.3.3.4).

5.3.3.3. Server-Funktionalität

Die Server-Funktionalität wird in der Hauptklasse `ServerController` und den Nebenklassen `Server` und `Connection` festgelegt. Es wird zwischen folgenden Abläufen unterschieden:

Startabläufe:

Nach dem Start der Anwendung im Server-Modus werden rudimentäre Einstellungen vorgenommen und die Hauptansicht `ServerMainViewController` (siehe Unterkapitel 5.3.2.1) angezeigt (siehe Abb. 5.15, *init*-Aktivität). Anschließend werden die *Clientgroups* von der Festplatte geladen und die Anzahl erwarteter Clients in der Hauptansicht der Anwendung angezeigt (siehe ebd., *startServer*-Aktivität). Abschließend wird ein *Listen*-Port am Netzwerk-Interface geöffnet um auf Verbindungen von Clients zu warten.

Betätigt der Benutzer in der Hauptansicht den *Trackchange*- oder *Config*-Knopf wird der Serverprozess beendet. Kehrt er aus diesen Szenen (siehe Unterkapitel 5.3.2.1) zurück werden die *Clientgroups* neu eingelesen und der Serverprozess erneut gestartet.

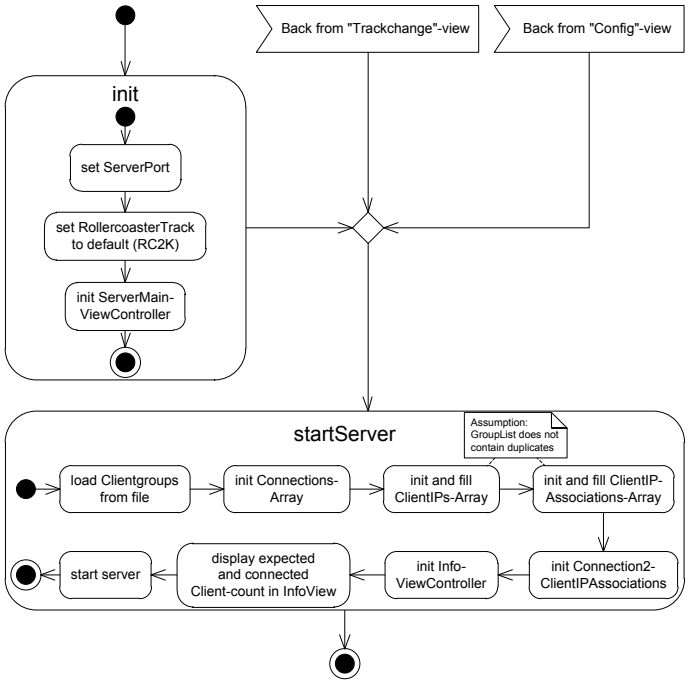

Abbildung 5.15.: Abläufe beim Starten der Server-Funktionalität.

Stopabläufe:

Beendet der Anwender den Server – z.Bsp. durch Drücken der *Home*-Taste am Mobilgerät – wird kurz vor der Anwendungsterminierung eine *STOP*-Nachricht an den Client geschickt (siehe Abb. 5.16). Dann trennt der Server alle Verbindungen zu Clients, führt Speicheraufräumarbeiten durch und terminiert.

Schlägt der Aufbau einer Verbindungsgrundlage für Clients fehl – z.Bsp. wenn der zu verwendete Port durch eine andere Anwendung auf dem Gerät blockiert ist – werden die oben genannten Schritte mit der Ausnahme der *App*-Terminierung ausgeführt. In diesem Fall wird nach wenigen Sekunden erneut versucht die Serverfunktionalität zu starten.

Drückt der Anwender zu einem beliebigen Zeitpunkt der Laufzeit den *Trackchange*- oder *Config*-Knopf (siehe Unterkapitel 5.3.2.1) wird die Serverfunktionalität ebenfalls gestoppt. Den verbundenen Clients wird per *STOP*-Nachricht mitgeteilt, dass der Server Änderungen der Achterbahnstrecke oder Konfiguration erfährt. Die Serverfunktionalität wird erst dann wieder gestartet (siehe Paragraph „Startabläufe" in diesem Unterkapitel), wenn der Benutzer die Achterbahnauswahls- oder Konfigurationsansicht verlässt.

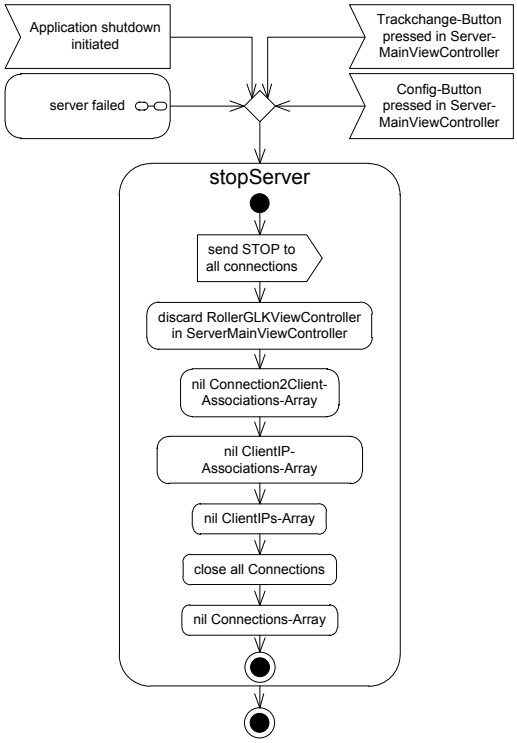

Abbildung 5.16.: Abläufe beim Stoppen der Server-Funktionalität.

Bedienungs- und Kommandoabläufe:

Hello (Empfang): Ist der Server initialisiert und verbindet sich ein Client, sendet der Client eine *HELO*-Nachricht (*Hello*) an den Server (siehe Abb. 5.17). Eine *HELO*-Nachricht enthält die IP-Adresse des Clients. Anhand dieser überprüft der Server ob der Client dazu bemächtigt ist an der Achterbahnsimulation teilzunehmen – welche Clients dem Server beitreten dürfen wird in der Konfigurationsansicht anhand der *ClientGroup*-Liste verwaltet (siehe Unterkapitel 5.3.2.1).

Ist ein Client nicht autorisiert sich einer Achterbahnfahrt anzuschließen, wird ihm eine *NOTW*-Nachricht (*Not Welcome*) zugeschickt und daraufhin die TCP-Verbindung zum Client geschlossen. Dem Client steht es frei sich erneut zu verbinden.

Ist der Client an der Teilnahme willkommen, wird ihm eine *WELC*-Nachricht (*Welcome*) zugeschickt. Nach erfolgreichen Verbindung eines autorisierten Clients wird der Informationstext in der Hauptansicht des Servers aktualisiert. Sind alle Clients verbunden wird der *Start*-Knopf freigegeben über den die Achterbahnsimulation gestartet werden kann.

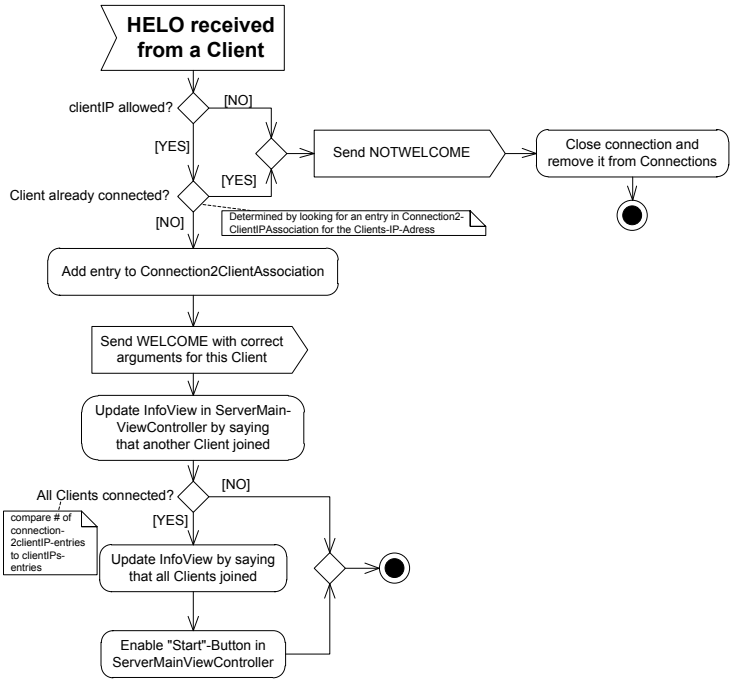

Abbildung 5.17.: Abläufe des Servers beim Empfang einer *HELO*-Nachricht von einem Client.

Welcome (Versandt): Verbindet sich ein autorisierter Client mit dem Server wird ihm eine *WELC*-Nachricht (*Welcome*) geschickt. Diese enthält drei Parameter zur allgemeinen Konfiguration des Clients während der Achterbahnsimulation. Der erste Parameter gibt die horizontale Verschiebung für den Aufnahmewinkel des Clients in der Achterbahnsimulation an. Der Zweite gibt an ob der Client sein Gyroskop einschalten soll. Der letzte Parameter gibt an ob ein Client alleine in einer *Clientgroup* ist, das ist z.Bsp. dann der Fall wenn nur ein Mobilgerät in ein *HMD* eingesetzt wird.

Der letzte Parameter ergibt sich aus der Anzahl Clients in einer *Clientgroup*, die anderen zwei legt der Server-Administrator in der Detailansicht für jeden Client individuell fest, nachdem er eine *Clientgroup* in der Konfigurationsansicht anlegt oder editiert (siehe Unterkapitel 5.3.2.1).

Not Welcome (Versandt): Verbindet sich ein nicht autorisierter Client wird ihm eine *NOTW*-Nachricht (*Not Welcome*) zugeschickt. Dem Client liegt es frei sich erneut zu verbinden, eventuell handelt es sich um ein Missverständnis das der Server-Administrator durch eine Korrektur der Server-Einstellungen beheben kann.

Start (Versandt): Nachdem sich alle Clients verbunden haben, kann die Achterbahnsimulation durch einen Klick auf den *Start*-Knopf in der Hauptansicht gestartet werden (siehe Unterkapitel 5.3.2.1). Direkt nach dem Klick wird allen Clients eine *STRT*-Nachricht (*Start*) zukommen gelassen (siehe Abb. 5.18), die angibt welche Achterbahnstrecke während der simulierten Fahrt darzustellen ist. Zusätzlich ersetzt der Server die Informationsansicht im oberen Teil seiner Hauptansicht durch eine eigene Repräsentation der Achterbahnsimulation. Nach dem Start wird der *Start*-Knopf deaktiviert und der *Pause*-Knopf kurz darauf aktiviert.

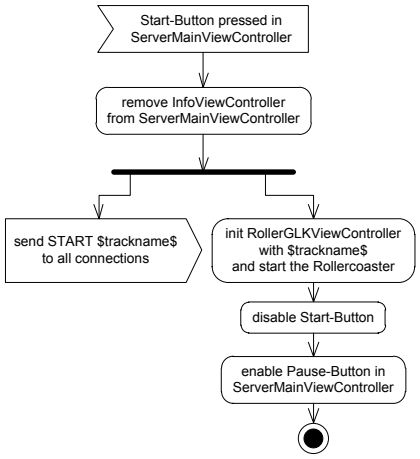

Abbildung 5.18.: Abläufe des Servers nach einem Klick auf den *Start*-Knopf.

Rdy (Empfang): Nachdem ein Client eine *STRT*-Nachricht (*Start*) erhält lädt er die in der Nachricht angegebene Achterbahnstreckenbeschreibung. Liegt diese Strecke nicht vor, wird eine Standard-Strecke geladen. Dann initialisiert der Client seine *OpenGL-ES*-Puffer. Da es zu zeitliche Abweichungen bei der Initialisierung der *OpenGL-ES*-Puffer zwischen verschiedenen Apple-Mobilgerätemodellen kommt, wird die Darstellung der Achterbahn direkt nach der erfolgreichen Initialisierung der Puffer gestoppt[1]. Dann sendet der Client eine *RDY*-Nachricht (*Ready*) an den Server um ihn zu informieren. Erst wenn jeder Client dem Server bestätigt hat, dass er seine Puffer fertig

[1] Ein *iPhone 3GS* braucht z.Bsp. wesentlich weniger Zeit um die Puffer zu initialisieren als ein *iPhone 4*.

initialisiert hat, kann die Achterbahnsimulation wirklich loslegen. Dann sendet der Server allen Clients eine *GO*-Nachricht.

Go (Versandt): Nachdem alle Clients via *RDY*-Nachricht (*Ready*) gemeldet haben, dass sie ihre Puffer für die bevorstehende Achterbahnfahrt initialisiert haben, sendet der Server eine *GO*-Nachricht an alle Clients. Sie veranlasst die Clients, die Achterbahn zu zeichnen. Nach Versandt der Nachricht nimmt auch der Server die Zeichnung der Achterbahn auf.

Client Move (Empfang): Falls ein Client nicht alleine in einer *Clientgroup* ist, das Mobilgerät über ein Gyroskop verfügt und der Server dem Client in der *STRT*-Nachricht (*Start*) mitteilt dieses zu aktivieren, dann sendet dieser Client die gemessenen Bewegungsinformationen in regelmäßigem Absätnden inf Form von *CMOV*-Nachrichten (*Client Move*) an den Server. Details zur Einstellung des Gyroskops finden sich in der Klasse `RollerGLKViewController` in der Methode `enableGyro`.

Ist ein Client hingegen alleine in einer Gerätegruppe, verfügt er über ein Gyroskop und soll dieses laut *STRT*-Nachricht vom Server aktivieren, so wertet er die Bewegungsinformationen aus und ändert seinen Blickwinkel in der Achterbahnsimulation direkt. In diesem Fall werden keine *CMOV*-Nachrichten geschickt, was zu einer Reduzierung der über das Netzwerk übertragenen Pakete führt.

Server Move (Versandt): Der Server sendet die in Form von *CMOV*-Nachrichten (*Client Move*) empfangenen Bewegungsinformationen neu verpackt als *SMOV*-Nachricht (*Server Move*) an beiden Clients der Gruppe aus der die Information ursprünglich kamen (siehe Abb. 5.19). Würde nur dem zweiten Client aus der Gruppe die Information geschickt werden und der andere Client, der sein Gyroskop auswertet, die Bewegungen direkt umsetzen, dann könnte dies unter Umständen zu einer Asynchronität zwischen den Geräten eines *HMD* führen.

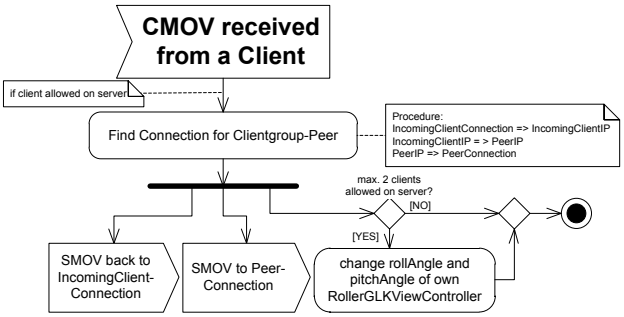

Abbildung 5.19.: Server-Abläufe beim Empfang einer *CMOV*-Nachricht.

Pause (Versandt): Betätigt der Serveradministrator den *Pause*-Knopf oder berührt er die Fläche der Hauptansicht in welcher die Achterbahnsimulation angezeigt wird, pausiert der Server die

Achterbahnfahrt (siehe Abb. 5.20). Dazu sendet er eine *PAUS*-Nachricht (*Pause*) an alle beteiligten Clients und unterbricht auch die lokale Achterbahnsimulation. An die Stelle der Achterbahnsimulation in der Server-Hauptansicht tritt wieder die Informationsansicht, die den Administrator über den aktuellen Serverzustand informiert.

Ist die Achterbahn gestoppt, wird der Schriftzug des *Pause*-Knopfes in „*Unpause*" geändert. Zusätzlich werden alle Knöpfe für eine kurze Zeit deaktiviert. Dies soll verhindern, dass die Achterbahnsimulation schnell hintereinander gestoppt und wieder gestartet wird. Ein solches Vorgehen führt nämlich zu erheblichen Problemen auf den Mobilgeräten, da diese ihre Puffer beim Wechsel zwischen Achterbahn- und Kameraansicht und zurück nicht schnell genug leeren und wieder befüllen können. Es droht eine Asynchronität der Bildausgabe zwischen den Clients. Nach der Karenzzeit werden alle Knöpfe bis auf den *Start*-Knopf wieder aktiviert. Wird der nunmehr „*Unpause*" beschriftete Knopf gedrückt, wird den Clients erneut eine *PAUS*-Nachricht zugeschickt und die Achterbahnsimulation wird sowohl auf den Clients wie auf dem Server wieder aufgenommen.

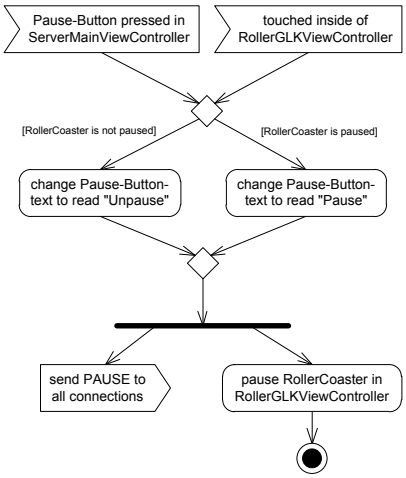

Abbildung 5.20.: Abläufe des Servers nach Aktivierung des *Pause*-Modus.

Bye (Empfang): Wird die *iRoller2000*-Anwendung auf einem beliebigen am Server verbundenen Client beendet, sendet dieser vor Programmterminierung eine *BYE*-Nachricht. Infolge der Nachricht unterbricht der Server seine Funktionalität und startet sich neu (siehe Abb. 5.21 und Paragraph „Startabläufe" in diesem Unterkapitel). Dabei wird allen beteiligten Clients eine *STOP*-Nachricht geschickt, welche angibt dass ein Client den Server verließ. Wurde die Achterbahnsimulation angezeigt, wird diese unterbrochen.

Haben sich alle Clients wieder verbunden und wird die Achterbahnsimulation erneut gestartet, fängt die Achterbahnfahrt wieder am Anfang der Strecke an.

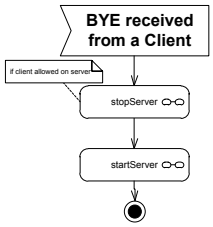

Abbildung 5.21.: Server-Abläufe beim Empfang einer *BYE*-Nachricht.

Stop (Versdandt): Wird die Server-Applikation verlassen, verabschiedet sich ein Client mit einer *BYE*-Nachricht vom Server oder wird eine Änderung an der Achterbahnstrecke/ Konfiguration vorgenommen so wird allen verbundenen Clients eine *STOP*-Nachricht mit einer Begründung geschickt. Befanden sich Client und Server in der Achterbahnansicht wird diese unterbrochen.

5.3.3.4. Client-Funktionalität

Startabläufe:

Nach dem Start der Anwendung im Client-Modus – das *Hostaddress*-Feld wurde in den Geräteeinstellungen ausgefüllt – werden neben grundsätzlichen Initialisierungen auch die Kamera des Mobilgeräts initialisiert und das Livebild angezeigt (siehe Abb. 5.22). Der Client versucht sich dann mit dem Server zu verbinden. Der aktuelle Verbindungszustand wird dem Anwender in einem *Overlay* in der Kameraansicht angezeigt. Schlägt die Verbindung fehl, versucht der Client sich alle drei Sekunden erneut zu verbinden. Konnte die TCP-Verbindung erfolgreich aufgebaut werden sendet der Client dem Server eine *HELO*-Nachricht (*Hello*) mit seiner IP-Adresse.

Erhält ein Client zu einem beliebigen Zeitpunkt während der Ausführung eine *STOP*- oder *NOTW*-Nachricht (*Not Welcome*) wechselt der Client zurück in die Kameraansicht – sofern er diese nicht bereits anzeigt – und versucht sich erneut mit dem Server zu verbinden.

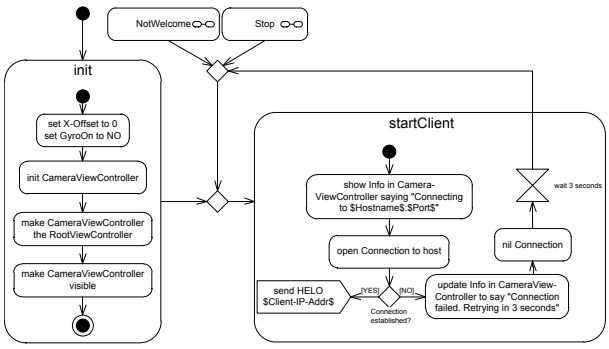

Abbildung 5.22.: Abläufe beim Start der Client-Funktionalität.

Stopabläufe:

Beendet der Benutzer die Anwendung auf einem Client – z.Bsp. durch Drücken der *Home*-Taste am Mobilgerät – sendet der Client eine *BYE*-Nachricht an den Server (siehe Abb. 5.23). Anschließend wird die TCP-Verbindung zum Server getrennt und Aufräumarbeiten durchgeführt.

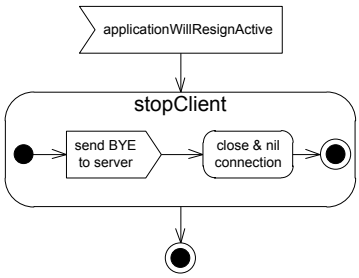

Abbildung 5.23.: Abläufe beim Stoppen der Client-Funktionalität.

Kommandoabläufe:

Hello (Versandt): Nachdem ein Client erfolgreich eine TCP-Verbindung zum Server aufgebaut hat, sendet er diesem eine *HELO*-Nachricht (*Hello*). Die Nachricht enthält die IP-Adresse des Clients und dient dem Server zur Überprüfung ob der Client autorisiert ist an der Achterbahnsimulation teilzunehmen. Ist dies der Fall erhält er eine *WELC*- (*Welcome*), ansonsten eine *NOTW*-Nachricht (*Not Welcome*), zurück.

Welcome (Empfang): Hat ein Client eine *HELO*-Nachricht (*Hello*) an einen Server geschickt und ist der Client autorisiert sich mit dem Server zu verbinden, erhält der Client vom Server eine *WELC*-Nachricht zurück (siehe Abb. 5.24). Eine *WELC*-Nachricht gibt die horizontale Verschiebung der Kameraposition in der Achterbahnsimulation für den Client vor. Zusätzlich enthält sie die Anweisung ob das Gyroskop des Clients aktiviert werden soll oder nicht. Außerdem enthält sie die Information ob der Client alleine in einer Clientgroup ist oder nicht. Dem Anwender werden diese Informationen in der Kameraansicht präsentiert.

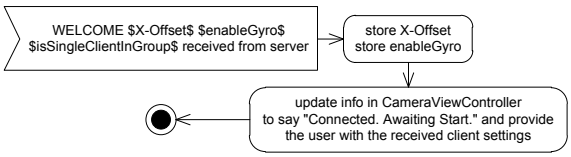

Abbildung 5.24.: Client-Abläufe beim Empfang einer *WELCOME*-Nachricht.

Not Welcome (Empfang): Ist ein Client nicht auf einem Server willkommen, wird ihm eine *NOTW*-Nachricht (*Not Welcome*) zugestellt. Der Anwender wird in der Kameraansicht entsprechend informiert (siehe Abb. 5.25). Anschließend versucht der Client sich alle drei Sekunden erneut mit dem Server zu verbinden. Eventuell liegt ein Missverständnis oder Konfigurationsfehler vor, den der Administrator in der Zwischenzeit in der Konfigurationsansicht der Server-Anwendung behebt.

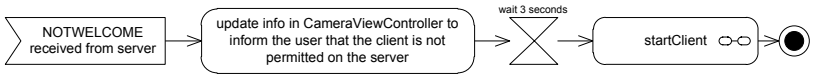

Abbildung 5.25.: Client-Abläufe beim Erhalt einer *NOTW*-Nachricht.

Start (Empfang): Nachdem alle Clients erfolgreich am Server angemeldet sind, wird der *Start*-Knopf in der Hauptansicht der Server-Anwendung freigegeben. Drückt der Administrator diesen, startet er die Achterbahnsimulation. Hierbei sendet der Server allen Clients eine *STRT*-Nachricht (*Start*), welche als einzigen Parameter den Namen der bei der Achterbahnfahrt anzuzeigende Strecke enthält (siehe Abb. 5.26).

Die Clients tauschen die Kameraansicht gegen die Achterbahnansicht (siehe Unterkapitel 5.3.2.2), laden die Streckenbeschreibung und initialisieren ihre *OpenGL-ES*-Puffer. Da die verschiedenen *iOS*-Geräte für diesen Vorgang unterschiedlich lang brauchen und damit sich hierdurch keine Asynchronität bei der Darstellung der Achterbahn zwischen den Clients ergibt, wird die Achterbahnsimulation kurz vor der Anzeige pausiert. Im Anschluss wird eine *RDY*-Nachricht an den Server gesendet.

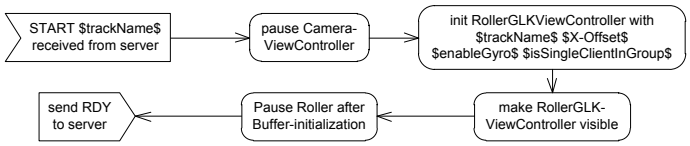

Abbildung 5.26.: Client-Abläufe beim Erhalt einer *STRT*-Nachricht.

Ready (Versandt): Bevor die Achterbahnsimulation endgültig starten kann, muss der Server von jedem Client eine *RDY*-Nachricht (*Ready*) erhalten. Diese bezeugt, dass ein Client seinen *OpenGL-ES*-Puffer fertig initialisiert hat.

Go (Empfang): Sobald sich alle Clients beim Server via *RDY*-Nachrichten (*Ready*) gemeldet haben, dass sie ihre *OpenGL-ES*-Puffer initialisiert haben, schickt der Server jedem Client eine *GO*-Nachricht. Erst mit dem Empfang dieser Nachricht wird die Achterbahnstrecke auf allen beteiligten Geräten angezeigt.

Client Move (Versandt): Wurde einem Client vom Server in der *STRT*-Nachricht (*Start*) mitgeteilt, dass er sein Gyroskop aktivieren soll und ist der Client nicht allein in einer Clientgroup, so sendet er im regelmäßigem Intervall *CMOV*-Nachrichten (*Client Move*) mit Bewegungsinformationen an den Server (siehe Abb. 5.27). Ist der Client alleine in seiner Gruppe resultiert eine Lageveränderung des Geräts direkt in einer Änderung des Betrachtungswinkels in der Achterbahn auf diesem Gerät. In diesem Fall werden keine *CMOV*-Nachrichten an den Server geschickt um Bandbreite zu sparen.

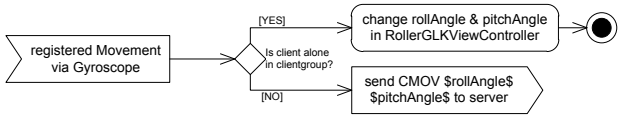

Abbildung 5.27.: Client-Abläufe nach einem Update der Bewegungswerte die vom Gyroskop des Geräts gemessen werden.

Server Move (Empfang): Erhält ein Server per *CMOV*-Nachrichten (*Client Move*) Bewegungsinformationen von einem Client sendet er diese neu verpackt als *SMOV*-Nachrichten (*Server Move*) an beide Clients aus der *Clientgroup* aus der die *CMOV*-Nachricht stammt (siehe Abb. 5.28). Empfängt ein Client eine *SMOV*-Nachricht verändert er den Sichtwinkel in der Achterbahnsimulation.

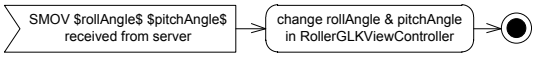

Abbildung 5.28.: Client-Abläufe beim Empfang einer *SMOV*-Nachricht.

Pause (Empfang): Empfängt ein Client eine *PAUS*-Nachricht (*Pause*) und zeigt er die Achterbahnansicht an (siehe Unterkapitel 5.3.2.2), wird die Achterbahnfahrt pausiert und die Client-Ansicht wechselt zurück zur Anzeige des Kamerabilds (siehe Abb. 5.29). Der Benutzer wird über den Grund der Unterbrechung informiert.

Befindet sich der Client zum Zeitpunkt des Empfangs einer *PAUS*-Nachricht in der Kameraansicht, wird die Achterbahnsimulation an der Stelle an der sie pausiert wurde, wieder aufgenommen.

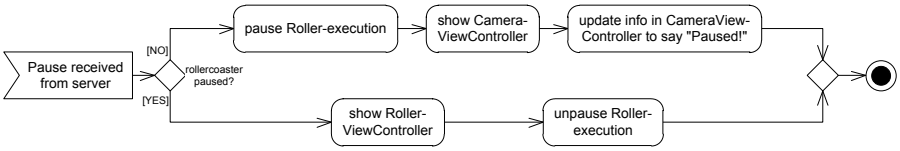

Abbildung 5.29.: Client-Abläufe beim Empfang einer *PAUS*-Nachricht.

Bye (Versandt): Wird die Client-Anwendung terminiert – z.Bsp. weil der Nutzer die *Home*-Taste gedrückt hat – und ist der Client mit einem Server verbunden, wird eine *BYE*-Nachricht an den Server gesendet. Anschließend wird die Anwendung beendet.

Stop (Empfang): Empfängt ein Client eine *STOP*-Nachricht wird die Achterbahnfahrt, sofern sie gerade lief, gestoppt und die Kameraansicht angezeigt (siehe Abb. 5.30). Darin wird der Anwender über den Grund der Unterbrechung informiert. Daraufhin trennt der Client die TCP-Verbindung zum Server und unternimmt nach drei Sekunden einen neuen Versuch sich mit dem Server zu verbinden.

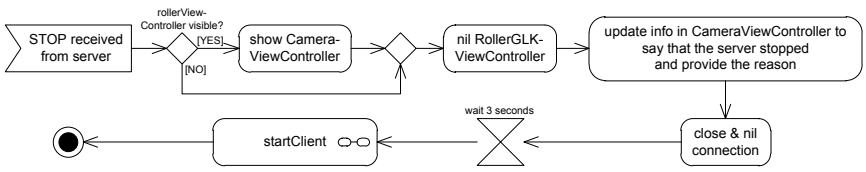

Abbildung 5.30.: Client-Abläufe beim Empfang einer *STOP*-Nachricht.

5.3.3.5. Kommunikationsabläufe zwischen Server und Client

Nachdem die Server- und Client-Funktionalität sowie die unterschiedlichen Netzwerknachrichten in den letzten Unterkapiteln im Detail behandelt wurden, werden hier mehrere mögliche Kommunikationsszenarien zwischen Client und Server in in kurzer, zusammenhängender Form dargestellt und erklärt (siehe Abb. 5.31).

Das linke Kommunikationsprotokoll zeigt einen Ablauf bei dem sich ein autorisierter Client während der Achterbahnsimulation mit einer *BYE*-Nachricht vom Server verabschiedet. Hiernach stoppt der Server via *STOP*-Nachricht die restlichen mit ihm verbundenen Clients und startet die Server-Funktionalität neu.

Im zweiten Ablauf wird die Server-Anwendung durch den Administrator beendet, woraufhin der Versandt einer *STOP*-Nachricht an alle betroffenen Clients erfolgt. Die Client-Anwendung stoppt die Achterbahnsimulation, zeigt die Kameraansicht an und versucht sich alle drei Sekunden neu mit dem Server zu verbinden.

Im letzten Ablaufdiagramm wird der Verbindungsversuch eines nicht-autorisierten Clients durch eine *NOTW*-Nachricht (*Not Welcome*) vom Server unterbunden. Der Client schließt die TCP-Verbindung, informiert den Benutzer und versucht sich alle drei Sekunden erneut zu verbinden.

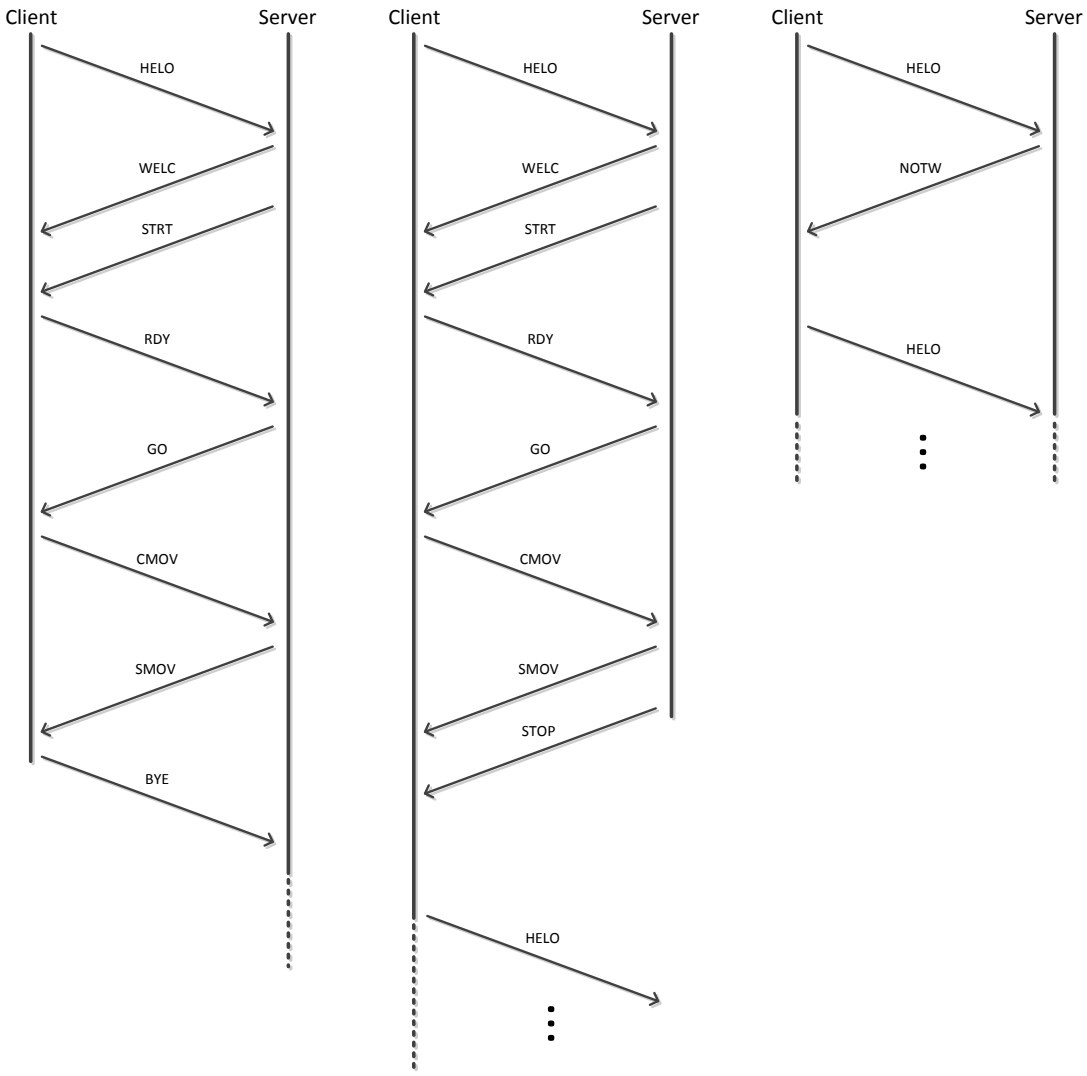

Abbildung 5.31.: Mögliche Kommunikationsabläufe zwischen Client und Server.

Teil V.

Fazit

6. Zusammenfassung

6.1. Résumé

Bei heutigen Entwicklungen eines *Head-mounted Displays* (kurz *HMD*) steht nicht mehr nur die reine Darstellung von Inhalten im Vordergrund. Um den gesteigerten Anforderungen im Bereich des *wearable computings* gerecht zu werden und den Menschen besser im Alltag zu unterstützen ist eine Vielzahl zusätzlicher Peripherie und Sensoren nötig. Aber auch die Ansprüche an die Darstellungsmöglichkeiten von Inhalten steigen durch die Fortschritte im Bereich der Displaytechnik weiter an. Unlängst dominieren Begriffe wie *Full-HD-* und *4K*-Auflösungen sowie die stereoskopische Darstellung von Inhalten die Medienlandschaft. Um den gesteigerten Ansprüchen der Konsumenten gerecht zu werden, muss eine *HMD* sowohl in ihrer Kernkompetenz als auch darüber hinaus überzeugen können.

Zu diesem Zweck wird am *Institut für integrierte Naturwissenschaften der Universität Koblenz-Landau* an einem *HMD* auf Basis der hochauflösenden *Retina Displays* in *Apple*-Mobilgeräten geforscht. Neben der hohen Auflösung stellt vor allem der geringe Preis beim gleichzeitig gebotenen, hohen Umfang an Sensoren und rechnerischen Kapazitäten, einen Anreiz für eine diesbezügliche Lösung dar. Werden mehrere dieser Geräte zu einem *HMD* verbunden können sie mit entsprechender Software nicht nur einfaches Bildmaterial wiedergeben, sondern u.a. auch stereoskopische Bildausgaben produzieren und die Bewegungen des Benutzers verfolgen.

Um die Machbarkeit eines solchen *HMD* zu zeigen, stand die Erstellung einer Anwendung für *iOS* im Vordergrund dieser Arbeit. Sie sollte die Vorzüge der *Apple*-Mobilgeräte für die Darstellung stereoskopischer Inhalte sowie die sinnvolle Nutzung weiterer Gerätesensoren zur Erweiterung des *HMD*-Erlebnisses aufweisen. Im Vorfeld der Konzeption und Entwicklung einer für diese Zwecke geeigneten Applikation wurden die von *Apple* aufeinander abgestimmten Konzepte zur Entwicklung von *iOS*-Anwendungen erklärt (siehe Kapitel 2). Die eng gekoppelten Grundsteine des Ansatzes bilden die dynamische, objektorientierte Sprache *Objective-C*, der um das objektorientierte Paradigma erweiterte Compiler sowie die auf die Bedürfnisse dynamischer Sprachen entwickelte Laufzeitumgebung (siehe Teil II). Darauf aufgebaut sind die, für den Einsatz auf Mobilgeräten, hoch optimierten Frameworks welche in der Summe das *Cocoa Touch Framework* und das *iOS-SDK* bilden und die Entwicklungsumgebung *Xcode* (siehe Teil III).

Anhand der gewonnen Erkenntnisse zur Entwicklung von Applikationen mit dem *iOS-SDK* sowie den darunterliegenden Sprachkonzepten wurden Anforderungen an eine Software für den Einsatz in einem *HMD* auf Basis der *Retina Displays* erhoben und eine spezifikationskonforme Anwendung entwickelt (siehe Teil IV). Die Einführungskapitel sowie die im Rahmen der Arbeit erstellte Aufgabe bilden die Grundlage für weitere Konzepte und Einsatzmöglichkeiten einer *HMD* auf *iOS*-Basis (siehe Unterkapitel 6.2).

6.2. Weiterführende Projekte

Die im Rahmen dieser Arbeit erstellte Anwendung im Speziellen und das Konzept einer *3D*-stereoskopischen *HMD* auf *iOS*-Basis im Allgemeinen, lässt sich auf mehreren Ebenen erweitern. Das grafische und funktionelle Grundgerüst der *iRoller2000*-Anwendung kann in weiterverwendeter Form dafür genutzt werden die Auswahl und Ausführung verschiedener *HMD*-Anwendungen zu ermöglichen. Auf der technischen Seite könnten in der aktuellen Software sowie in Weiter- oder Neuentwicklungen zusätzliche Gerätesensoren wie etwa der *GPS*- oder *Audio*-Chip genutzt werden. Die aktuelle Steuerung der Anwendung könnte durch ein neues Bedienungskonzept besser an die Bedürfnisse für den Einsatz in einem *HMD* angepasst werden. Hierzu könnte z.Bsp. die in neuen Modellen der *Apple*-Mobilgeräte eingebaute Spracherkennung und -steuerung benutzt werden, deren Funktionalität aber zum aktuellen Zeitpunkt von *Apple* in einer nicht-öffentlichen Bibliothek geführt wird.

6.3. Vergleichbare Projekte

Konzepte für den Aufbau eines *HMD* auf Basis der *Retina Displays* sind selten. Eins der wenigen Projekte, entwickelt an der *University of Southern California* geht nicht über einen rudimentären, experimentellen Zustand hinaus, der zudem erhebliche Schwächen bei der Verteilung der Bewegungsdaten aufweist[1].

Andere *HMD*-Modelle nutzten die Mobilgeräte hingegen nicht direkt, sondern verbinden sich per *Apple-Dock-Connector*. Die Treiber solcher Lösungen eignen sich aber oft nur zur Anzeige des Gerätebildinhalts und bieten keine Stereoskopie oder weiterführende Funktionalität[2].

HMD-Lösungen mit vergleichbaren Umfang, welche nicht auf dem *Retina Display* basieren sind dagegen häufiger anzutreffen. Die meisten fristen aber nur ein Nischendasein, weil sie entweder zu geringe Auflösungen[3] oder wenig Anwendungsmöglichkeiten bieten. Zumeist schreckt der hohe Preis besser ausgestatteter Geräte die Anwender ab[4].

In Zukunft könnten die weitere Verkleinerung aktueller Techniken und bessere Bedienkonzepte dazu führen, dass *HMDs* der Durchbruch in den Alltag gelingt. Ähnlich der wachsenden Verbreitung aktueller Smartphones und Tablet-Computer.

[1] `http://people.ict.usc.edu/~suma/papers/olson-vr2011poster.pdf`, Stand: 29.05.2012
[2] `http://www.vuzix.com/consumer/products_av310w.html`, Stand: 29.05.2012
[3] `http://cinemizer.zeiss.com/cinemizer-oled/de_de/cinemizer-oled.html`, Stand: 29.05.2012
[4] `http://www.epson.com/cgi-bin/Store/jsp/Moverio/Home.do`, Stand: 29.05.2012

Teil VI.

Appendix

Inhalt des Datenträgers

- Quellcode der *OpenGL*-Version des *Rollercoaster2000*-Projekts.
- Quellcode der *OpenGL-ES*-Version *RollerCoaster2000*-Projekts (Version *2005*).
- Quellcode der im Rahmen dieser Diplomarbeit entwickelten Applikation *iRoller2000*.
- Quellcode der im Rahmen des *iOS-SDK*-Kapitels (siehe Teil III) entwickelten Beispielanwendung:
  - je ein Archiv mit dem Quellcode der Beispielanwendung nach jeder Lektion.
- Handbücher von *Apple* welche bei der Erstellung der *iRoller2000*-Applikation benutzt wurden und/ oder in der Ausarbeitung referenziert wurden.
- Ausarbeitung der Diplomarbeit im *PDF*-Format.
- Quellcode der Ausarbeitung als *LyX*-Datei (erstellt mit *LyX*[5] Version *2.03* unter Verwendung von *MikTex*[6] Version *2.9*), inklusive:
  - aller in der Ausarbeitung enthaltenen Bilder,
  - aller für die Ausarbeitung erstellten Binärdateien, u.a. *UML*-Diagramme, und
  - dem Literaturverzeichnis als *Bib*-Datei (verwaltet mit *JabRef*[7] 2.7 unter Verwendung der *Java*-Laufzeitumgebung[8] Version *1.7.0_03*).

[5] `http://www.lyx.org/`
[6] `http://www.miktex.org`
[7] `http://www.jabref.sourceforge.net`
[8] `http://www.oracle.com/technetwork/java/index.html`

Literaturverzeichnis

[Ale] ALEX ALLAIN: *The C Preprocessor*. http://www.cprogramming.com/tutorial/cpreprocessor.html, Abruf: 7. April 2012

[App07] APPLE INC. (Hrsg.): *Code Loading Programming Topics*. Apple Inc., August 2007. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/LoadingCode/LoadingCode.pdf

[App09a] APPLE INC. (Hrsg.): *Notification Programming Topics*. Apple Inc., August 2009. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Notifications/Notifications.pdf

[App09b] APPLE INC. (Hrsg.): *Objective-C Runtime Programming Guide*. Apple Inc., Oktober 2009. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf

[App09c] APPLE INC. (Hrsg.): *String Programming Guide*. Apple Inc., Oktober 2009. https://developer.apple.com/library/mac/documentation/cocoa/Conceptual/Strings/Strings.pdf

[App10a] APPLE INC. (Hrsg.): *Cocoa Fundamentals Guide*. Apple Inc., Dezember 2010. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf

[App10b] APPLE INC. (Hrsg.): *UIAcceleration Class Reference*. Apple Inc., April 2010. http://developer.apple.com/library/ios/documentation/UIKit/Reference/UIAcceleration_Class/UIAcceleration_Class.pdf

[App11a] APPLE INC. (Hrsg.): *Table View Programming Guide for iOS*. Apple Inc., Januar 2011. http://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/TableView_iPhone/TableView_iPhone.pdf

[App11b] APPLE INC. (Hrsg.): *The Objective-C Programming Language*. Apple Inc., Oktober 2011. https://developer.apple.com/library/mac/documentation/cocoa/conceptual/objectivec/objc.pdf, Abruf: 03.04.2012

[App11c] APPLE INC. (Hrsg.): *View Programming Guide for iOS*. Apple Inc., März 2011. http://developer.apple.com/library/ios/DOCUMENTATION/WindowsViews/Conceptual/ViewPG_iPhoneOS/ViewPG_iPhoneOS.pdf

[App11d] APPLE INC. (Hrsg.): *Xcode 4 User Guide*. Apple Inc., Oktober 2011. https://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/Introduction/Introduction.html

[App12a] APPLE INC. (Hrsg.): *iOS Human Interface Guidelines*. Apple Inc., März 2012. http://developer.apple.com/library/ios/DOCUMENTATION/UserExperience/Conceptual/MobileHIG/MobileHIG.pdf

[App12b] APPLE INC. (Hrsg.): *Transitioning to ARC Release Notes*. Apple Inc., Februar 2012. http://developer.apple.com/library/ios/releasenotes/ObjectiveC/RN-TransitioningToARC/RN-TransitioningToARC.pdf

[App12c] APPLE INC. (Hrsg.): *UIViewController Class Reference*. Apple Inc., Februar 2012. http://developer.apple.com/library/ios/documentation/uikit/reference/UIViewController_Class/UIViewController_Class.pdf

[Ben12] BENZ, Benjamin: ARM, aber sexy - Was Smartphones und Tablets so schnell macht. In: *c't - Magazin für Computertechnik* Ausgabe 6 (27. Februar 2012)

[Dav11] DAVE MARK, JACK NUTTING, JEFF LAMARCHE: *Beginning iOS 5 Development - Exploring the iOS SDK*. 4. Edition. Apress, 2011

[Ebe07] EBERT, Prof. D.: *Strukturbezogene Prinzipien*. https://www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Teaching/WS0708/Programming/softwareEngineering003.png. Version: 2007

[LLV12] LLVM OVERSIGHT GROUP: *Automatic Reference Counting*. http://clang.llvm.org/docs/AutomaticReferenceCounting.html. Version: 2012, Abruf: 7. Mai 2012

[Mar09] MARK DALRYMPLE, SCOTT KNASTER: *Learn Objective-C on the Mac*. First Edition. Apress, 2009 http://books.google.de/books?id=zhGJnXQNvmsC&pg=PA119&lpg=PA119&dq=%23pragma+objective+c&source=bl&ots=-E0obEpwkg&sig=e7fx5i-DNS-vOiLwoOkw-4d5JzQ&hl=en&sa=X&ei=zU9_T-rxNcP1sgazpqGfBA&ved=0CDEQ6AEwBQ#v=onepage&q=%23pragma%20objective%20c&f=false

[Pro09] PROF. DR. ARND POETZSCH-HEFFTER: *Konzepte Objektorientierter Programmierung - Mit einer Einführung in Java*. 2. Auflage. Springer-Verlag, Berlin, 2009

[Seb11] SEBASTIAN MEYERS, TORBEN WICHERS: *Objective-C 2.0 - Programmierung für Mac OS X und iPhone*. 2. Auflage. Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg, 2011

[Som07] SOMMERVILLE, Ian: *Software Engineering*. 8. Auflage. Pearson Studium, 2007