TECHNISCHE UNIVERSITÄT DRESDEN

Technische Universität Dresden Fakultät Informatik, Institut für Systemarchitektur, Professur für Rechnernetze

Belegarbeit

KONZEPTE FÜR MEHRMANDANTENFÄHIGKEIT VON KOMPLEXEN iOS-APPS

Sebastian Hagedorn
Matr.-Nr.: 3387648

Betreut durch:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
und:
Dr.-Ing. Thomas Springer
Dipl.-Inf. Matthias Neubert

Eingereicht am 30.11.2012

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Professur Rechnernetze

AUFGABENSTELLUNG FÜR DIE BELEGARBEIT

Thema: **Konzepte für Mehrmandantenfähigkeit von komplexen iOS-Apps**

Name, Vorname	Hagedorn, Sebastian	Studiengang	Diplom Medieninformatik
Matrikel-Nr.	3387648	Projekt/Schwerpunkt	MUC
Betreuer	Dr. Thomas Springer	Externer Betreuer	Matthias Neubert (SALT)
Beginn am	1. Juni 2012	Einzureichen am	30. November 2012

ZIELSTELLUNG

Große Softwaresysteme wie ERP-Anwendungen sind kostenintensiv und aufwändig, sodass sie selten spezifisch für einen Kunden entwickelt und implementiert werden. Stattdessen sollte eine Codebasis so konfigurierbar sein, dass sie für viele Kunden verwendbar ist - Konzepte wie Modularisierung sind in der Java-Welt weit verbreitet und werden durch Technologien wie OSGi unterstützt. Mit dem Einzug von umfangreichen Business-Anwendungen auf mobile Geräte wie dem iPad geht der Wunsch einher, auch dort Modularisierung zum Erlangen der Mehrmandantenfähigkeit einzusetzen. Während bei Java-basierten mobilen Plattformen viele altbekannte Konzepte und Entwicklerwerkzeuge übernommen werden können, sind Apples Entwicklungsumgebung Xcode und die Programmiersprache Objective-C von diesen Entwicklungen weitestgehend abgeschnitten.

In der Belegarbeit soll untersucht werden, wie man dennoch eine größtmögliche Trennung einzelner Anwendungsbereiche in iOS-Projekten erreichen kann. Denkbar wäre beispielsweise eine Kapselung von zusammengehörigen Klassen zu Frameworks oder Subprojekten. Ziel ist es, kundenspezifische Anpassungen von Grundfunktionalitäten, Bugfixes und Updates zu isolieren. Die Arbeit erfolgt im Rahmen einer Werkstudententätigkeit bei SALT Solutions GmbH, wo eine iPad-App als Ergänzung zur hauseigenen Warehousing-Software entwickelt wurde. Die Mehrmandantenfähigkeit ist ein aktuelles Problem der monolithischen App, sodass der praktische Teil in der Restrukturierung dieser Anwendung liegen wird. Die dabei gewonnenen Erkenntnisse sollen direkt in die Belegarbeit einfließen und anschließend bewertet werden.

SCHWERPUNKTE

- Recherche gängiger Praktiken zur Modularisierung und Mehrmandantenfähigkeit auf anderen Plattformen
- Sammlung und Evaluation von möglichen Ansätzen für Xcode-/iOS-Projekte
- Anwendung des vielversprechendsten Ansatzes auf die iPad-App von SALT Solutions GmbH
- Analyse und Bewertung der Lösung/Umsetzung

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill (betreuender Hochschullehrer)

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 30.11.2012

INHALTSVERZEICHNIS

1 EINLEITUNG

1.1 MOTIVATION

Komplexe Softwaresysteme wie ERP[1]-Anwendungen sind extrem kostenintensiv und aufwändig in der Realisierung. Deshalb ist es nicht nur wünschenswert, sondern auch notwendig, dass eine Anwendung an **viele Kunden** verkauft werden kann, um die Kosten auf eine breite Basis zu verteilen. Dennoch dürfen kundenspezifische Konfigurationen und Anpassungen nicht ausgeschlossen werden. Eine Anwendung, die als Gesamtpaket sowohl einen hohen Wiederverwendungs- als auch Anpassungsgrad hat, wird im Rahmen dieser Arbeit als mehrmandantenfähig bezeichnet. Eine genauere Definition des Begriffs erfolgt in Kapitel 2, Grundlagen.

Um die beiden Ziele Wiederverwendbarkeit und Konfigurierbarkeit zu vereinbaren, haben sich Konzepte wie das der **Modularisierung** bewährt. Modularisierung hilft zudem, Entwicklungszweige zusammenzuführen, die über einen längeren Zeitraum unabhängig voneinander weiterentwickelt worden sind. Während das Zusammenführen solcher Zweige auf Projektebene ab einem bestimmten Punkt zu komplex wird, erlaubt eine modularisierte Anwendung, auch das Zusammenführen auf Modulebene durchzuführen und somit die Komplexität stark einzugrenzen.

Um eine Anwendung modulbasiert zu entwickeln, ist es von entscheidender Bedeutung, dass unterstützende Technologien und Entwicklerwerkzeuge zur Verfügung stehen. Da ein Großteil der bestehenden Business-Anwendungen auf Java basiert, ist für Java-basierte Systeme eine breite Auswahl solcher Technologien und Werkzeuge verfügbar.

Zunehmend werden bestehende ERP-Systeme um komplexe, **mobile Clients** erweitert, die ebenfalls Mehrmandantenfähigkeit unterstützen und dabei auf bewährte Konzepte, Technologien und Werkzeuge zurückgreifen sollen. Relativ unproblematisch ist dies bei Java-basierten Plattformen wie Android, während sich Apple bei der **iOS-Plattform** auf Objective-C als Hauptentwicklungssprache festgelegt hat und somit nicht von der breiten Werkzeug- und Technologieauswahl profitieren kann. Mit einem weltweiten Marktanteil von fast 70 % bei Tablet-Geräten ist das auf

[1] Enterprise Resource Planning

iOS basierende Apple iPad jedoch eine besonders relevante Zielplattform für mobile ERP-Clients [isu12], zumal das iPad im Bereich der Geräteaktivierungen im Business-Umfeld Anfang des Jahres einen Anteil von über 97 % erreichen konnte [goo12].

1.2 ZIELSTELLUNG

In dieser Belegarbeit wird untersucht, wie man auf der iOS-Plattform Mehrmandantenfähigkeit erlangen kann. Im Vordergrund steht dabei die größtmögliche **Entkopplung einzelner Anwendungsbereiche**, um kundenspezifische Anpassungen von Grundfunktionalitäten, Bugfixes und Updates zu isolieren. Auch die Skalierbarkeit der Anwendungsentwicklung und des Deployments für viele Mandanten ist ein wichtiges Kriterium, auf das in Abschnitt 2.1.1 näher eingegangen wird.

Die Arbeit wird im Rahmen einer Werkstudententätigkeit bei der SALT Solutions GmbH angefertigt, wo eine iPad-App als Ergänzung zur hauseigenen Warehousing-Software entwickelt wurde. Die App wird im Folgenden mit dem Namen *alexa MRS* bezeichnet. Die Mehrmandantenfähigkeit ist ein aktuelles Problem der monolithischen App, weshalb Konzepte zur Restrukturierung erforderlich sind. Diese Konzepte sollen im Rahmen dieser Arbeit entwickelt und evaluiert werden. Eine vollständige Anwendung aller Konzepte auf die alexa-MRS-App würde den Rahmen dieser Belegarbeit sprengen, daher werden ausgewählte Maßnahmen an einem Prototyp implementiert und evaluiert. Die Erkenntnisse aus der Entwicklungsarbeit an der alexa-MRS-App bilden dennoch eine wichtige Grundlage zur Konzipierung und Bewertung der prototypischen Arbeit.

1.3 RAHMENBEDINGUNGEN

Damit die Forschungsergebnisse in der Praxis anwendbar sind, müssen einige Rahmenbedingungen beachtet werden. Zwar ist ein Vertrieb von iOS-Apps außerhalb des Apple App Stores[2] im Rahmen des iOS Developer Enterprise Programs[3] grundsätzlich möglich, jedoch ist vielen Entwicklern (einschließlich der SALT Solutions GmbH) die **App-Store-Kompatibilität** wichtig. Dies hat zur Folge, dass die Apple App Store Review Guidelines [app12a] beachtet werden müssen, die u. a. das Nachladen von Code und eigene dynamische Bibliotheken ausschließen. Konzepte zum Erlangen der Mehrmandantenfähigkeit, die nicht App-Store-kompatibel sind, werden deutlich gekennzeichnet und sollten nur nach gründlicher Diskussion der Konsequenzen angewandt werden.

Die erarbeiteten Lösungen sollten mit Apples kostenlos bereitgestellter Entwicklungsumgebung **Xcode**[4] kompatibel sein. Durch die enge Kopplung von Xcode und anderen wichtigen Entwicklungswerkzeugen wie dem Simulator oder Instruments ist ein Verzicht auf Xcode mit starken

[2] `http://www.apple.com/itunes/`
[3] `https://developer.apple.com/programs/ios/enterprise/`
[4] `https://developer.apple.com/xcode/`

Nachteilen für die Entwickler verbunden. Zudem müssen die vorgeschlagenen Lösungen auf unmodifizierten Endgeräten[5] lauffähig sein. Als Basis für die praktische Arbeit dienen Xcode in der Version 4.5 sowie iOS 6.

1.4 GLIEDERUNG

Die vorliegende Arbeit umfasst sieben Kapitel. In Kapitel 2 werden die Grundlagen der Problemdomäne beschrieben, insbesondere werden die Begriffe der Mehrmandantenfähigkeit und der Modularisierung definiert. Im zweiten Teil des Kapitels werden bestehende Technologien zur Unterstützung von Mehrmandantenfähigkeit zusammengefasst. Das 3. Kapitel gibt eine Einführung in die in dieser Arbeit im Vordergrund stehende iOS-Plattform. Neben der Programmiersprache Objective-C werden auch die Entwicklungsumgebung Xcode sowie die Vertriebsmöglichkeiten und -limitierungen untersucht. In Kapitel 4 werden die iOS-spezifischen Gegebenheiten genutzt, um Teilkonzepte für mehrmandantenfähige Anwendungen zu erarbeiten. Der Großteil dieses Kapitels widmet sich dabei modularen Ansätzen. Die Teilkonzepte werden in Kapitel 5 zu einem Gesamtkonzept zusammengesetzt, das von einer prototypischen Implementierung begleitet wird. Sowohl die Teilkonzepte als auch das Gesamtkonzept werden anschließend in Kapitel 6 evaluiert, bevor die Arbeit in Kapitel 7 mit einer Zusammenfassung der wesentlichen Erkenntnisse und einem Ausblick abschließt.

[5] Für die Modifizierung von Endgeräten wird häufig der Begriff „Jailbreak“ verwendet

2 GRUNDLAGEN

Dieses Kapitel stellt die begrifflichen und konzeptionellen Grundlagen für die nachfolgenden Kapitel dar. Abschnitt 2.1 definiert Begriffe, die in dieser Arbeit von tragender Bedeutung sind. Abschnitt 2.2 fasst die wichtigsten Konzepte und Technologien zusammen, die auf anderen Plattformen die Mehrmandantenfähigkeit von Anwendungen ermöglichen.

2.1 BEGRIFFSEINFÜHRUNG

2.1.1 Mehrmandantenfähigkeit

In der Literatur und Forschung wird der Begriff der **Mehrmandantenfähigkeit** (engl.: Multitenancy) meist nur im Kontext von Serveranwendungen benutzt, wobei selten klare Definitionen aufgestellt werden. Im Rahmen von Software-as-a-Service-Plattformen (SaaS) bedeutet Mehrmandantenfähigkeit häufig, dass mehrere Mandanten (Kunden) einer Serveranwendung die gleiche Laufzeitinstanz nutzen, dabei aber völlig isoliert von anderen Mandanten arbeiten können [BZP+10, GSH+07]. Man kann dieses Szenario mit einem virtuellen Einmandantenbetrieb vergleichen, d. h. der Mandant kann die Anwendung so benutzen als liefe sie auf dedizierter Hardware. Da sich tatsächlich aber viele Mandanten sowohl die Hardware als auch die Laufzeitinstanz teilen, sind v. a. im Sicherheitsbereich besondere Maßnahmen erforderlich, um die Daten verschiedener Mandanten voneinander zu isolieren. Die Anwendungskomplexität wächst zusätzlich durch Konfigurationsmöglichkeiten, die sowohl die grafische Oberfläche (GUI) als auch die funktionale Ebene (z. B. mandantenspezifische Berechnungsvorschriften) betreffen können. Der Fokus der Mehrmandantenfähigkeit liegt bei diesen SaaS-Anwendungen klar auf der Kostenersparnis zur Laufzeit, da sich mehrere Mandanten die Kosten der Hardware-Ressourcen teilen können. Die Entwicklungs- und Wartungskosten der Software steigen dagegen aufgrund der gewachsenen Komplexität.

Verglichen mit [BZP+10] und [GSH+07] wird der Begriff der Mehrmandantenfähigkeit in [MUTL09] vielschichtiger betrachtet. Als generelle Ziele von Mehrmandantenfähigkeit werden Konfigurierbarkeit, Skalierbarkeit und Sicherheit (insbesondere im Hinblick auf die Datenisolation) definiert. Zur Klassifizierung werden verschiedene Modelle eingeführt, die als Levels bezeichnet werden. Dabei entspricht das höchste Level, Level 4 („Single Configurable Instance"), dem Ansatz, der zuvor in [BZP+10] und [GSH+07] beschrieben wurde – mehrere Mandanten teilen sich eine konfigurierbare Laufzeitinstanz („Shared App"). Interessanter für die vorliegende Arbeit ist Level 2 („Multiple Configurable Instances"), das von einer konfigurierbaren Codebasis ausgeht, zur Laufzeit aber getrennte Anwendungen für jeden Mandanten bereitstellt. Da in dieser Arbeit die client-seitige Mehrmandantenfähigkeit untersucht werden soll, ist dieses Modell besser anwendbar, weil jeder Mandant seine eigene (iOS-)Hardware besitzt – die Hardwareskalierbarkeit spielt demnach keine Rolle. In einem Vergleich zwischen Level-2- und Level-4-Anwendungen wurde gezeigt, dass Level-4-Anwendungen lediglich bei der in dieser Arbeit irrelevanten Hardwareskalierbarkeit Vorteile haben, während Level-2-Anwendungen Vorteile bei der über die Konfiguration hinausgehenden Anpassung, Isolation und Anwendungsentwicklung haben [MUTL09].

Von den genannten generellen Zielen – Konfigurierbarkeit, Skalierbarkeit, Sicherheit – soll in dieser Arbeit besonders die Konfigurierbarkeit der Codebasis im Vordergrund stehen. Die **Skalierbarkeit** bezieht sich im Folgenden nicht auf das Laufzeitverhalten, sondern auf die Entwicklung, das Deployment und die Wartung der Anwendung für potenziell sehr viele Mandanten. Die mehrmandantenspezifische **Sicherheit** spielt nur eine untergeordnete Rolle, da die Anwendungen auf dedizierter Hardware laufen und auf ihre eigene lokale Datenbank zugreifen. Die Trennung der Mandanten geschieht idealerweise bereits zur Compile-Zeit, alternativ ist aber auch eine generische Anwendung denkbar, die nach einer Authentifizierung die korrekte Konfiguration lädt. Solch eine generische Anwendung könnte dann einfach über den Apple App Store vertrieben werden. Für den Datenaustausch mit serverseitigen Datenbanken können herkömmliche Sicherheitsmechanismen wie Authentifizierung und Verschlüsselung verwendet werden.

Oft wird unter **Konfigurierbarkeit** lediglich die mandantenspezifische GUI-Anpassung verstanden, beispielsweise ein einheitliches Farbschema und angepasste Texte. Im Rahmen dieser Belegarbeit wird die Konfigurierbarkeit in drei Teilgebiete unterteilt: Die GUI, das Datenmodell und die Programmlogik. Um v. a. Programmlogik mandantenspezifisch anzupassen, ohne die Wiederverwendbarkeit gemeinsam genutzter Klassen einzuschränken, bietet sich eine modulare Architektur an, die auch von Apple für Business-Anwendungen empfohlen wird [Rah12]. Module können dann entweder zur Laufzeit durch mandantenspezifische Einstellungen konfiguriert werden oder bereits beim Build-Prozess in der mandantenspezifischen Konfiguration fest eingebaut werden. Die Modularisierung wird im nächsten Abschnitt genauer thematisiert.

Mehrmandantenfähigkeit sollte nicht mit einer Mehrnutzerfähigkeit verwechselt werden, die in der Regel durch eine Nutzeranmeldung realisiert wird. Zwar kann eine Anwendung auch in geringem Maße an einzelne Nutzer angepasst werden, die Kontrolle über die Art der Anpassung obliegt dabei aber dem Betreiber der Software, also dem Mandanten. Pro Mandant werden oft mehrere Nutzer unterstützt, die sich dann beim Start der Anwendung authentisieren müssen.

Zusammenfassend lässt sich sagen, dass Mehrmandantenfähigkeit Kosten und Aufwand sparen soll, wenn eine Basisanwendung für viele Mandanten zur Verfügung gestellt werden kann, die

jeweils eigene Konfigurationen der Software benötigen. Die Kostenersparnis wird durch eine möglichst große Schnittmenge des Codes aller Mandanten und einfache Anpassungsmechanismen erreicht, sodass die Entwicklungskosten der Basisanwendung auf alle Mandanten verteilt werden können. Auch sollten die Wartungskosten (pro Mandant) mit steigender Mandantenanzahl geringer werden. Mehrmandantenfähigkeit selbst ist dabei keine anwendbare Technologie, sondern sollte vielmehr als Motivation für das Anwendungsdesign verstanden werden.

2.1.2 Module und Abhängigkeiten

Im Gegensatz zum Begriff der Mehrmandantenfähigkeit gibt es zahlreiche präzise Definitionen von **Softwaremodulen**, die weitestgehend übereinstimmen. Knoernschild hat in [Kno12, Kapitel I.1 Module Defined] folgende Definition aufgestellt:

> „*A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers.*"

Konsumenten sind andere Anwendungsbereiche oder Module, die Funktionen eines Moduls aufrufen. „Deployable" legt nahe, dass Module einfach ausgetauscht, entfernt oder hinzugefügt werden können, ohne die Lauffähigkeit der Gesamtanwendung zu beeinträchtigen. Das Deployment kann zur Compile-Zeit oder, wie z. B. bei OSGi[6], dynamisch zur Laufzeit geschehen.

Module können wiederum selbst Konsumenten sein, d. h. auf die Schnittstellen anderer Module zugreifen. Diese Beziehung nennt man **Abhängigkeit** (engl.: Dependency), und alle Abhängigkeiten eines Moduls sollten explizit festgelegt sein [See09]. Ein Modul kann nur dann genutzt werden, wenn alle Abhängigkeiten aufgelöst worden sind, d. h. alle Module, zu denen eine Abhängigkeitsbeziehung besteht, sind ebenfalls eingebunden. Um Module leicht zu verwalten (sie „manageable" zu machen), sollte die Schnittstelle eines Moduls möglichst klein und keine Aggregation der Schnittstellen aller Modulklassen sein. Beim Design gilt es, Gruppen von Klassen zu finden, die untereinander enge Beziehungen haben, aber nur wenige Beziehungen zu Klassen aus anderen Gruppen haben.

Um die Schwierigkeit dieser Aufgabe zu verstehen, hilft es, den Begriff des **physischen Designs** zu betrachten. Der Unterschied zwischen physischem und logischem Design wurde sowohl in [CN91, Kapitel 1 Classes and Inheritance] als auch in [Lak05, Kapitel 1 Einleitung] und dem darauf aufbauenden Werk [Kno12, Kapitel I.5.4 Modular Tension] erklärt. Beim Design von Softwaresystemen wird oft nur dem logischen Klassendesign ausreichend Beachtung geschenkt. Das logische Design umfasst Entitäten (meist Klassen) und deren logische Beziehungen untereinander, z. B. Vererbungshierarchien, Benutzungs- und Referenzierungsbeziehungen. Die logischen Abhängigkeiten sind direkt aus dem Klassendiagramm ablesbar – Entitäten mit ausgehenden Beziehungen sind von anderen Entitäten abhängig, Entitäten mit eingehenden Beziehungen sind für andere Entitäten notwendig. Eine Entität funktioniert nicht ohne alle anderen Entitäten, von denen sie abhängig ist.

[6] `http://www.osgi.org/`

Bildet man Module auf dieser Basis, kann es zu schwerwiegenden Fehleinschätzungen kommen. Sind z. B. zwei Klassen verschiedener Module in der gleichen Datei implementiert, wird die Datei zur Kompilierung beider Module benötigt und es besteht eine physische Abhängigkeit zwischen den Modulen, die aus dem Klassendiagramm nicht ersichtlich ist. In [Lak05] wird daher zwischen dem logischen und dem physischen Interface unterschieden – das logische Interface ist aus den Beziehungen im Klassendiagramm ersichtlich, das physische Interface wird durch `#import`- bzw. `#include`-Anweisungen bestimmt. Oft wird das physische Interface vom Programmierer unbeabsichtigt über Modulgrenzen hinaus erweitert, beispielsweise durch globale Funktionen, global genutzte Enumerations und Konstanten. Die kleinsten physischen Entitäten sind Quellcodedateien, wobei in Sprachen wie C und Objective-C die Header- und Implementierungsdateien jeweils zu einer physischen Entität zusammengefasst werden. Solch eine physische Entität wird in [Lak05, Kapitel II.3 Components] **Komponente** (engl.: Component) genannt. Die kompilierte Form dieser physischen Entität wird in [CN91, Kapitel 3 Classes and Inheritance] als **Software-IC** bezeichnet. Komponenten sollten die Grundlage für das logische und physische Design sein, um ungewollte Abhängigkeiten auszuschließen [Lak05, Kapitel 0 Einleitung].

Zur Bestimmung der physischen Abhängigkeiten kann ein Graph genutzt werden, der auf Basis der `#import`- bzw. `#include`-Anweisungen automatisch generiert werden kann [Lak05, Kapitel II.3 Compontents]. Zudem haben logische Abhängigkeiten stets physische Abhängigkeiten zur Folge – das Fehlen von logischen Abhängigkeiten ist eine notwendige, aber keine hinreichende Bedingung für das Fehlen von physischen Abhängigkeiten. Insbesondere zyklische Abhängigkeiten zwischen Modulen sollten unbedingt vermieden werden, da die Module ansonsten stets zusammen benutzt werden müssen und somit vielmehr ein großes Modul bilden.

Ein wichtiges Designkriterium von Modulen ist deren **Granularität**. Dabei muss man zwischen hoher Wiederverwendbarkeit kleiner Module und einfacher Verwendbarkeit großer Module abwägen [Kno12, Kapitel I.5.1 The Use/Reuse Paradox]. Um eine sehr spezifische Aufgabe zu erfüllen, kann man also ein großes, spezifisches Modul bereitstellen, das im Idealfall nur eine definierte Schnittstelle zur Ausführung dieser Aufgabe bereitstellt. Die Benutzung ist einfach, dafür lässt sich das Modul nur in wenigen Situationen verwenden. Untergliedert man die Aufgabe in Teilmodule, erhöht sich deren Wiederverwendbarkeit, aber zur Benutzung ist zusätzliches Wissen nötig, z. B. die Reihenfolge der Benutzung aller Module und deren spezifischer Schnittstellen.

Modularisierung ist ein wichtiges Konzept, um Teile einer Anwendung voneinander zu entkoppeln. Damit dies nicht nur auf logischer Ebene geschieht, sondern auch beim Deployment einer Anwendung vorteilhaft genutzt werden kann, müssen die physischen Abhängigkeiten möglichst gering, keinesfalls zyklisch und klar definiert sein. Physisch voneinander getrennte Module können die Link- und Kompilierungsdauer senken, ausgetauscht und wiederverwendet werden, und erhöhen die Wartbarkeit der Gesamtanwendung [CN91, Kapitel 1 System Building].

2.2 VERFÜGBARE TECHNOLOGIEN UND WERKZEUGE

Um generelle Konzepte wie Mehrmandantenfähigkeit und speziell Modularisierung umzusetzen, ist Unterstützung durch konkrete Technologien und Werkzeuge wünschenswert. Besonders für Java-basierte Software gibt es zahlreiche Technologien und Entwicklungswerkzeuge, die Modularisierung forcieren und die Entwicklung modularisierter Anwendungen dadurch erleichtern. Eine Auswahl dieser soll in den folgenden Abschnitten vorgestellt werden, strukturiert nach ihrem Platz im Entwicklungszyklus.

2.2.1 Konzeption und Refactoring

In diesem Abschnitt wird das Design der Anwendungsarchitektur diskutiert und wie Werkzeuge diesen Prozess unterstützen können. Die Aufgabe ist dabei entweder, eine Anwendung von Grund auf zu entwickeln, oder eine bestehende Anwendung umzustrukturieren.

Die Grundlage für das logische Design sind häufig **Entwurfsmuster** (engl.: Design Patterns), die sich in Worten oder Diagrammen beschreiben lassen. Eine umfassende Sammlung von Patterns, die speziell für modulare Systeme gedacht sind, kann in [Kno12] eingesehen werden. Zwar sind alle Beispiele des Buchs auf Java bzw. OSGi bezogen, die Ideen lassen sich jedoch in den meisten Fällen ohne Weiteres auf andere Programmiersprachen und Technologien übertragen. Neben solch allgemeingültigen Mustern können auch technologie- und frameworkspezifische Konventionen und Muster in das Anwendungsdesign einfließen, sodass u. U. eine Abwägung zwischen allgemeingültigen Mustern und technologiespezifischen Mustern stattfinden muss.

Schwieriger ist jedoch meistens die Frage, wie die Durchsetzung von Mustern und Designentscheidungen im Laufe der Softwareentwicklung überwacht werden kann. Unter dem Paradigma **Round-Trip Engineering** wird versucht, das ursprüngliche Architekturmodell und die tatsächliche Implementierung zu synchronisieren [HLR08]. Ein grundlegender Gedanke ist dabei die automatische Codegenerierung aus dem Modell heraus. Für Java-Anwendungen gibt es zahlreiche Werkzeuge, die Round-Trip Engineering in verschiedenen Ausprägungen unterstützen. Die kostenlose Software *ArgoUML*[7] erlaubt sowohl das Erzeugen von UML-Diagrammen aus Java-Quell- und -Bytecode („Reverse Engineering"), als auch die Generierung von Codegerüsten aus UML-Diagrammen („Forward Engineering"). Für Letzteres werden mehrere Zielprogrammiersprachen unterstützt. *Object-AID*[8] ist ein Plugin für die v. a. für Java-Entwicklung beliebte Entwicklungsumgebung Eclipse[9], das UML-Diagramme aus dem aktuellen Java-Quellcode erzeugt. Auf diese Weise lassen sich unerwünschte Auswirkungen einer Codeänderung auf das logische Design sofort erkennen [BC02]. Kommerzielle Werkzeuge wie *Visual Paradigm for UML* (VP-UML) werben mit umfangreicher Unterstützung von Round-Trip Engineering, wobei VP-UML in diesem Bereich derzeit Java, .NET und C++ unterstützt[10].

[7] `http://argouml.tigris.org/`
[8] `http://www.objectaid.com/`
[9] `http://www.eclipse.org/`
[10] `http://www.visual-paradigm.com/product/vpuml/provides/roundtripcodeengine.jsp`

Die Erzeugung von UML-Diagrammen aus vorhandenem Quellcode ist v. a. für das Refactoring bestehender Anwendungen interessant. Ausgehend von einer monolithischen Anwendung kann man auf diese Weise versuchen, zusammengehörige Klassen zu finden und stückweise in Modulen zu isolieren. In [DYM+08] wurde ein Versuch unternommen, diese zeitaufwändige Aufgabe zu automatisieren, indem ein Algorithmus entwickelt wurde, der sog. Klassencluster (also potenzielle Module) automatisch findet. Der Algorithmus baut auf einem **Abhängigkeitsgraphen** auf, der sowohl aus Java-Quell- als auch -Bytecode generiert werden kann. Dies kann je nach Differenz zwischen logischem und physischem Design der Anwendung (s. Abschnitt 2.1.2) zu unterschiedlichen Ergebnissen führen. Der zugrundeliegende Algorithmus selbst ist dabei programmiersprachenunabhängig, die Implementierung allerdings an Eclipse und Java gebunden.

Zur Modularisierung von Java-Anwendungen ohne spezielle Zusatztechnologien wird in [Kno12, Kapitel 0 Introduction] das Java Archive (JAR/`.jar`) als physische Einheit zur Repräsentation eines Moduls vorgeschlagen. Um sicherzustellen, dass logisch unabhängige JARs auch physisch unabhängig sind (und somit getrennt benutzt werden können), kann z. B. das kostenlose Werkzeug *JarAnalyzer*[11] genutzt werden.

Alternativ lässt sich die physische Unabhängigkeit auch schon während des Build-Vorgangs überprüfen. Die Voraussetzung dafür ist das in [Lak05, Kapitel II.5 Levelization] beschriebene **Levelize-Pattern**, das Zyklen im Abhängigkeitsgraphen verbietet. Dadurch ist jeder mögliche Weg im Graph endlich. Alle Module ohne Abhängigkeiten werden als Module des Levels 0 definiert, das Level jedes anderen Moduls ist die minimale Pfadlänge von einem Level-0-Modul zu dem betreffenden Modul. Das maximale Level ist die Länge des längsten Pfads im Abhängigkeitsgraph [Lak05, Kapitel II.5 Levelization]. Darauf aufbauend wird in [Kno12, Kapitel II.12 Levelize Build] das **Levelize Build Pattern** eingeführt. Dieses besagt, dass alle Module eines Levels `i` fertig kompiliert sein müssen, bevor die Module des Levels `(i+1)` kompiliert werden können. Somit wird ausgeschlossen, dass ungewollte physische Abhängigkeiten von Modulen eines Levels `i` zu Modulen eines höheren Levels bestehen. Ein klares Verständnis über die vorhandenen Levels lässt sich z. B. durch die Verwendung von JarAnalyzer erlangen.

Wenn eine modulare Anwendung von Grund auf programmiert wird, muss nicht nur das logische Design stimmig sein, sondern auch sichergestellt werden, dass das geplante Design durchgehend und langfristig umgesetzt wird. Die Werkzeuge, die dazu zur Verfügung stehen, können oft gleichzeitig zum Refactoring bestehender, nichtmodularer Anwendungen genutzt werden. In diesem Abschnitt wurden beispielhaft einige dieser Werkzeuge (vorwiegend für Java-Anwendungen) im Zusammenspiel mit gängigen Entwurfsmustern und -zielen vorgestellt.

[11] `http://www.kirkk.com/main/Main/JarAnalyzer/` – eine separate Version für Microsoft .NET-Assemblies ist ebenfalls kostenfrei verfügbar

2.2.2 Modulverwaltung

Ist eine Anwendung einmal in Module unterteilt worden, ist einer der Vorteile, dass die Module verteilt entwickelt, kompiliert und getestet werden können. Allerdings müssen vor dem Kompilieren bestehende Abhängigkeiten aufgelöst werden, auch wenn die Abhängigkeiten zu örtlich entfernt entwickelten Modulen bestehen. In der Praxis sind solche entfernt entwickelten Module oft externe Bibliotheken, die spätestens zur Compile-Zeit benötigt werden.

In einem naiven Ansatz würden alle Abhängigkeiten aufgelöst werden, indem der benötigte Code manuell in das aktuell zu kompilierende Projekt kopiert wird. Gegen diesen Ansatz sprechen mehrere Faktoren: Der Workflow ist umständlich und kostet den Entwickler unnötige Arbeitszeit. Regeln für den Build-Prozess lassen sich nur global für alle Module spezifizieren. Bei einer großen Anzahl von Modulen stellt sich die Frage, auf welchem Weg die benötigten Module (und deren mögliche Abhängigkeiten, sog. transitive Abhängigkeiten) gefunden und beschafft werden können. Des Weiteren fehlt eine klare Definition, welcher Entwicklungsstand der eingebundenen Module benötigt wird und wie Änderungen daran synchronisiert werden.

Das Build-Tool **Apache Maven**[12] löst diese Probleme für Java-Projekte auf eine für den Entwickler sehr komfortable Weise. Die Grundlage für die **Abhängigkeitsverwaltung** stellt eine spezielle XML-Datei dar, die pro Modul existiert. Diese Datei trägt den Namen Project Object Model, kurz POM, und beschreibt u. a. das Modul selbst sowie dessen Abhängigkeiten. Eine Abhängigkeit wird durch die sog. Coordinate identifiziert, die aus einem Namen, einem eindeutigen Identifikator und einer Versionsnummer besteht. Geladen werden die Abhängigkeiten entweder aus einem zentralen, öffentlich lesbaren Repository (bietet sich für öffentliche Bibliotheken an) oder aus einem lokalen Repository (für eigene, private Bibliotheken). Dieser Prozess verläuft für den Nutzer transparent, vorausgesetzt, die referenzierten Abhängigkeiten sind in binärer Form vorhanden und verfügen ebenfalls über eine sie beschreibende POM-Datei. Auch die transitiven Abhängigkeiten werden von Maven automatisch aufgelöst.

Üblich ist zudem eine hierarchische Zerlegung eines Gesamtprojekts in lokale Module. Dabei fungiert das Wurzelprojekt nur als Hülle einer Menge von Modulen, die in der POM-Datei unter dem `modules`-Tag aufgelistet werden. Etwaige Abhängigkeiten zwischen diesen Maven-Modulen werden nicht im Wurzelprojekt, sondern in den POM-Dateien der Module selbst angegeben und bestimmen implizit die Reihenfolge beim Build-Prozess. Die unter `modules` angegebenen Module referenzieren jedoch im Gegensatz zu den Abhängigkeiten keine Binärdateien sondern Projekte, die im Rahmen des Build-Prozesses noch kompiliert werden müssen.

Maven unterstützt neben der Verwaltung von Abhängigkeiten auch den **Build-Prozess** selbst. Die Verzeichnisstruktur des Projekts ist dabei durch Konventionen festgelegt[13] und typische Build-Abläufe werden in sog. Lifecycles gekapselt. Ein Lifecycle besteht aus einer Reihe von Maven-Befehlen („Build-Phasen"), wobei jeder Befehl durch ein gleichnamiges Maven-Plugin implementiert wird. Der `default`-Lifecycle von Maven enthält u. a. die Build-Phasen `compile`, `test` und `install`. Der Befehl `compile` kompiliert das Projekt, `test` führt Unit-Tests durch und `install` macht das Modul im lokalen Maven-Repository für andere Projekte verfügbar. Es ist möglich,

[12] `http://maven.apache.org/`
[13] Bei Bedarf lässt sich auch eine eigene Struktur im POM definieren

Abhängigkeiten nur für bestimmte Build-Phasen zu definieren. Ein typisches Beispiel in Java ist die JUnit-Bibliothek[14], die nur für die Test-Phase benötigt wird und in der finalen Binärdatei nicht enthalten sein muss.

Zwar ist Maven ursprünglich für Java-Projekte entwickelt worden, durch Plugins können jedoch auch andere Programmiersprachen unterstützt werden. Neben Plugins für C/C++[15] und .NET[16] gibt es auch Plugins zur Unterstützung von Objective-C, von denen eins in Abschnitt 4.3.5 untersucht wird. Maven kann in zahlreiche IDEs, darunter Eclipse, IntelliJ IDEA[17] und NetBeans[18], integriert werden.

Im Vergleich zu anderen Build-Werkzeugen wie *GNU Make*[19] oder *Apache Ant*[20] ist Maven einfacher zu handhaben, weil viele Details des Build-Prozesses hinter Standardkonfigurationen und Konventionen verborgen bleiben. Dies führt insbesondere bei großen Projekten zu übersichtlicheren Build-Anweisungen, allerdings erschwert es auch die Konfigurierbarkeit ohne gründliche Einarbeitung. Voraussetzung für die Nachvollziehbarkeit des Build-Vorgangs ist das allgemeine Verständnis des Maven-Konzepts. Darüber hinaus ist die Abhängigkeitsauflösung mit öffentlichen und privaten Repositories sowie lokalen Caches von Modulen ein Alleinstellungsmerkmal von Maven.

Eine andere Möglichkeit, verteilte Abhängigkeiten aufzulösen, bieten **Source-Control-Management-Werkzeuge** (SCM) wie Git[21]. Dabei fällt das Verifizieren von Abhängigkeiten und das Bauen von Software nicht in den Aufgabenbereich von SCM-Werkzeugen, sondern nur die Beschaffung von Modulen und deren Versionsverwaltung. Dafür sind diese Werkzeuge komplett unabhängig von den verwendeten Technologien, Projektstrukturen und Programmiersprachen. In Git gibt es den Befehl `git submodule`, der es erlaubt, Teile eines Projekts in getrennte Repositories auszulagern. Eine genauere Untersuchung von Git Submodules erfolgt in Abschnitt 4.3.4. Mit SCM-Werkzeugen lässt sich auch die zweidimensionale Konfiguration von Modulen gut abbilden: Die Versionsnummer identifiziert den allgemeinen Entwicklungsstand, während mandantenspezifische Änderungen und Anpassungen in Branches (dt: Abzweigungen) ausgelagert werden. Im Gegensatz zu sonst typischen Szenarien werden dabei nie die Abweichungen der Branches vom Hauptentwicklungszweig übernommen, sondern allgemeine Weiterentwicklungen von den einzelnen Branches per Merge übernommen.

Die beiden vorgestellten Ansätze, Maven und SCM, lassen sich auch kombinieren. Dazu muss das SCM-Repository in der POM-Datei referenziert sein. Maven abstrahiert die verwendete SCM-Technologie[22] und lädt den in der POM-Datei durch die Versionsnummer spezifizierten Stand aus dem SCM-Repository. Zum Auffinden der korrekten Version werden Tags verwendet. Sofern sich das Objective-C-Plugin für Maven als praktikabler Ansatz herausstellt, wird diese Methode näher untersucht werden.

[14] `http://www.junit.org/`
[15] `http://java.freehep.org/freehep-nar-plugin/`
[16] `http://incubator.apache.org/npanday/`
[17] `http://www.jetbrains.com/idea/`
[18] `http://netbeans.org/`
[19] `http://www.gnu.org/software/make/`
[20] `http://ant.apache.org/`
[21] `http://git-scm.com/`
[22] Neben Git werden auch SVN, CVS uvm. unterstützt: `http://maven.apache.org/scm/scms-overview.html`

Die einfache Verwaltung von Modulen und deren Beziehungen untereinander ist wichtig, um Module einerseits getrennt entwickeln zu können, sie aber andererseits beim Build-Prozess wieder zusammenführen zu können. Während Build-Tools mit Abhängigkeitsverwaltung programmiersprachenspezifisch sind, fällt diese Einschränkung bei SCM-Systemen weg, weshalb diese Technologien ohne Weiteres auch für die iOS-Entwicklung genutzt werden können.

2.2.3 Laufzeitunterstützung

In den vorangegangenen Abschnitten wurden Module als Bausteine eines Softwareprojekts in der Entwicklungsphase gesehen, die entweder binär oder in Form von Quellcode vorliegen. Bevor die Anwendung gebaut wird, muss festgelegt werden, welche Module berücksichtigt werden sollen. Abhängigkeiten der eingebundenen Module werden dabei entweder explizit angegeben oder vom Build-System automatisch erkannt und aufgelöst.

Um von der Modularisierung auch zur Laufzeit profitieren zu können, ist eine dafür ausgelegte Laufzeitumgebung erforderlich. Die Hauptaufgabe einer solchen Laufzeitumgebung besteht darin, Module während der Laufzeit zu laden oder zu entladen, und dabei die Abhängigkeiten korrekt aufzulösen. Daraus ergibt sich eine Reihe von Vorteilen: Module können einzeln und ohne Unterbrechung des Gesamtprogramms aktualisiert werden, um entweder neue Funktionen oder Bugfixes einzubringen. Aktuell nicht benötigte Module können zur Freigabe von Speicher entladen werden. Das Konzept erlaubt einen Mandantenwechsel „on the fly", indem neu benötigte Module nachgeladen und nicht mehr benötigte Module entladen werden, bzw. mandantenspezifische Modulversionen ausgetauscht werden. Durch eine geeignete Architektur können auch mehrere gleichzeitig laufende Anwendungen auf gemeinsam genutzte Module zugreifen, die nur einmal pro Laufzeitumgebung geladen werden müssen und somit ebenfalls den Speicherverbrauch reduzieren.

Für Java-basierte Systeme ist mit der **OSGi-Plattform** eine weitverbreitete Technologie verfügbar. Die OSGi-Spezifikation beschreibt einerseits die Laufzeitplattform („Service Platform") und andererseits ein Programmiermodell zur Entwicklung der Module [RAR07]. Es gibt mehrere stabile und zertifizierte Implementierungen der OSGi-Spezifikation, eine vollständige Liste ist auf der Webseite[23] der OSGi Alliance einsehbar.

Die Laufzeitunterstützung von Modulen ist besonders erstrebenswert, wenn das Deployment der gesamten Anwendung aufwändig oder problembehaftet ist. Dies ist beispielsweise dann der Fall, wenn ein ferngesteuertes Deployment nicht möglich ist oder hohe Anforderungen an die Verfügbarkeit der Anwendung bestehen, sodass ein kurzzeitiger, installationsbedingter Ausfall nicht akzeptabel ist [Kno12, Kapitel III.13.3 Digesting OSGi].

Vorgreifend soll hier erwähnt werden, dass die Laufzeitunterstützung von Modulen unter iOS nicht ohne Weiteres möglich ist – die Gründe dafür werden in den Abschnitten 3.3 (iOS-Runtime-Umgebung) und 3.4 (Apple Guidelines) genauer erläutert. Diese Tatsache ist für die Mehrmandantenfähigkeit von ERP-Clients auf Basis von iOS aber keine wesentliche Einschränkung: Das

[23] `http://www.osgi.org/Certification/Certified/`

Deployment von Apps ist entweder aus der Ferne oder direkt vom Endgerät aus möglich (s. Abschnitt 3.5, Alternative Vertriebsmodelle) und die Installationszeit im Bereich von Sekunden oder wenigen Minuten schränkt die Benutzbarkeit von ERP-Clients in keinster Weise ein. Auch die Unterstützung mehrerer Mandanten einer Laufzeitinstanz ist in dieser Arbeit nicht von Bedeutung, wie in Abschnitt 2.1.1 (Mehrmandantenfähigkeit) beschrieben.

2.3 ZUSAMMENFASSUNG

In diesem Grundlagenkapitel wurden die Begriffe der Mehrmandantenfähigkeit und Modularisierung eingeführt und für diese Arbeit definiert. Das Ziel von Mehrmandantenfähigkeit ist die Kostensenkung durch einfache Wartbarkeit und Konfigurierbarkeit einer Anwendung für viele Kunden. Modularisierung kann dazu beitragen, indem einzelne Anwendungsbereiche voneinander entkoppelt werden. Zur Umsetzung wurden in Abschnitt 2.2 aktuell verfügbare Technologien, Werkzeuge und Entwurfsmuster vorgestellt, hauptsächlich für die Entwicklung mit Java.

3 DIE iOS-PLATTFORM

Inwiefern die im vorangestellten Kapitel vorgestellten Technologien und Werkzeuge auch für die Entwicklung mit Xcode und Objective-C nutzbar sind, soll im Hauptteil dieser Arbeit untersucht werden. Zuvor wird in diesem Kapitel eine Einführung in die auf der iOS-Plattform verwendete Programmiersprache Objective-C gegeben, anschließend sollen die Möglichkeiten der von Apple bereitgestellten Entwicklungsumgebung Xcode erörtert werden. Im Anschluss werden drei weitere Bereiche des iOS-Ökosystems kurz vorgestellt: Die Runtime-Umgebung, die Apple Review Guidelines und die verschiedenen Vertriebsmodelle für iOS-Apps.

3.1 OBJECTIVE-C

Ziel dieses Abschnitts ist es nicht, eine vollständige Einführung in **Objective-C** zu geben. Vielmehr sollen Unterschiede zu gängigen Programmiersprachen wie Java hervorgehoben und Eigenschaften von Objective-C herausgestellt werden, die für die Mehrmandantenfähigkeit von Anwendungen von Vor- oder Nachteil sein können. Zum Erlernen von Objective-C und den von Apple bereitgestellten Frameworks Cocoa und Cocoa Touch sind einerseits zahlreiche Bücher erhältlich, andererseits bietet die Apple-Dokumentation[24] umfangreiche und kostenlose Lernmaterialien.

3.1.1 Geschichte

Verglichen mit der Programmiersprache Java, die Anfang der 1990er Jahre entwickelt wurde[25], ist Objective-C eine relativ alte Sprache. 1983 veröffentlichten Cox und Novobilski die erste Auflage des Buchs „Object-Oriented Programming – An Evolutionary Approach“ [CN91], in dem Objective-C als Syntaxergänzung zu C vorgestellt wird, um **objektorientierte Programmierung**

[24] `https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/`
[25] `http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html`

zu ermöglichen. Inspiriert wurden sowohl die Syntax als auch das Konzept der objektorientierten Zusatzschicht von Smalltalk-80[26].

Das Buch widmet sich nicht nur der neuen Syntax, sondern auch Implementierungsdetails. Im Vordergrund steht dabei das dynamische Binden von Nachrichten an Methodenimplementierungen zur Laufzeit, was die lose Kopplung eines Gesamtsystems vereinfacht. Durch dieses und andere Sprachelemente wird eine modulare Struktur von Objective-C-Anwendungen eigentlich begünstigt [CN91, Kapitel 1 System Building; 3 Classes and Inheritance]. Dass die Auswahl an umfangreichen Werkzeugen und anwendungsübergreifenden Modularisierungstechnologien dennoch sehr überschaubar ist, kann wohl auf die über lange Zeit sehr geringe Verbreitung der Sprache zurückgeführt werden. Im TIOBE Programming Community Index[27] blieb Objective-C bis Mitte 2008 unter 0,25 %. Seit der Einführung des iPhone SDKs im März 2008[28], das es Entwicklern erstmals erlaubte, eigene Apps zu programmieren und für Endnutzer anzubieten, gab es einen steilen, bis heute ungebrochenen Anstieg im TIOBE Index auf derzeit 10,38 %. Zwar wird Objective-C auch heute kaum außerhalb von Mac OS X und iOS verwendet, die enorme Verbreitung von iOS-Geräten und -Entwicklern hat Objective-C aber inzwischen zum dritten Rang hinter Java und C im TIOBE Index verholfen (Stand: November 2012). Dabei verzeichnet Objective-C derzeit weiterhin das höchste Wachstum aller Programmiersprachen in den Top 20 und wurde 2011 aufgrund des größten absoluten Wachstums in einem Jahr in die TIOBE Hall of Fame der Programmiersprachen aufgenommen [tio12].

Während die grundlegende Syntax unverändert geblieben ist, hat Apple die Sprache in den vergangenen Jahren entscheidend weiterentwickelt. Dazu gehören ein automatisiertes Speichermanagement (ARC[29]) und vereinfachte Accessor-Methoden (Properties[30]), getragen durch einen neuen Compiler (LLVM[31]), der laufend aktualisiert wird [Bea12]. Die in dieser Belegarbeit thematisierten Frameworks und Werkzeuge sind großteils nur unter iOS bzw. Mac OS X lauffähig. Die Programmiersprache selbst ist plattformunabhängig, mit GNUstep[32] ist auch eine Portierung des Basis-Frameworks Cocoa für alternative Plattformen verfügbar.

3.1.2 Messaging

In Objective-C werden Methoden wie in anderen objektorientierten Sprachen in Klassen implementiert. Der Aufruf unterscheidet sich jedoch nicht nur syntaktisch, sondern v. a. konzeptionell. Anstatt Methoden auf Klassen oder Objekten (Klasseninstanzen) direkt aufzurufen, werden **Nachrichten** (engl.: Messages) an Zielobjekte[33] gesendet. Das Binden einer Nachricht an eine Methodenimplementierung erfolgt ausschließlich zur Laufzeit. Die Nachricht enthält dabei neben einem sog. **Selector** einen Pointer auf den Empfänger sowie potenziell weitere Argumente. Da Nach-

[26] `http://www.smalltalk.org/`

[27] `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm` – der Index versucht die Beliebtheit von Programmiersprachen anhand von Suchmaschinenergebnissen anzugeben

[28] `http://www.apple.com/pr/library/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta.html`

[29] `http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/`

[30] `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocProperties.html` – standardisierte Getter und Setter

[31] `https://developer.apple.com/technologies/tools/whats-new.html#llvm-compiler`

[32] `http://www.gnustep.org/`

[33] Auch Klassen sind in Objective-C Objekte

richten selbst normale Objective-C-Objekte sind, können sie zur Laufzeit erstellt und manipuliert werden. Dieser Umstand bleibt bei Benutzung der **Standardnachrichtensyntax** oft verborgen. Eine einfache Nachricht wird folgendermaßen gesendet:

```
1 [receiver aMessage:firstArg withSecondArg:secArg];
```

Auf diese Weise wird implizit und für den Programmierer transparent ein Nachrichtenobjekt erstellt, mit dem Empfängerobjekt `receiver`, dem Selector `aMessage:withSecondArg:` und den beiden Argumenten `firstArg` und `secArg`. Greift man in diesen Prozess nicht programmatisch ein, wird zur Laufzeit in der Methodentabelle des Empfängerobjekts nach einer Methoden gesucht, die dem Namen des Selectors entspricht. Wird eine Methode gefunden, wird sie samt aller in der Nachricht enthaltenen Argumente aufgerufen – andernfalls gibt es eine Laufzeit-Exception. Per Konvention sollte man die oben beschriebene Syntax nur benutzen, um bereits zur Compile-Zeit feststehende, in einem Interface beschriebene Methoden per Nachricht aufzurufen. Wird dies eingehalten, gleicht das Senden einer Nachricht dem Aufruf einer gleichnamigen Methode, wie z. B. bei einem Methodenaufruf in Java. Andernfalls erzeugt der Compiler eine Warnung, einen Fehler gibt es dennoch erst zur Laufzeit. Diese Konvention ermöglicht es, Typsicherheit und feste Schnittstellen wie in Sprachen mit statischer Bindung zu nutzen, um robusten Code zu schreiben.

Möchte man hingegen die Möglichkeiten der **dynamischen Bindung** bewusst nutzen, kann man entweder eigene Nachrichtenobjekte vom Typ `NSInvocation`[34] erstellen und absenden oder auf vorgefertigte Sprachkonstrukte zurückgreifen. Apple hat in der Basisklasse des Cocoa- und Cocoa-Touch-Frameworks, `NSObject`, eine Methode namens `performSelector:`[35] definiert und implementiert. Sie kapselt das Anlegen eines Nachrichtenobjekts, wobei der Selector-Name, der als Parameter übergeben wird, zur Compile-Zeit nicht auf seine Gültigkeit überprüft wird. Selector-Namen lassen sich zur Laufzeit aus Strings und damit beispielsweise auch aus Nutzereingaben generieren, sodass eine solche Prüfung auch nicht vollständig möglich wäre.

Nutzt man die Möglichkeiten der dynamischen Bindung gezielt aus, lässt sich eine größtmögliche physische Entkopplung einzelner Klassen und damit auch Module erreichen. Da der Compiler keine Möglichkeit hat, die Nachrichten zu verifizieren, sind auch keine Schnittstelleninformationen anderer Klassen notwendig. Bei konsequenter Nutzung fallen Klassen-Imports weg und Klassen können völlig unabhängig voneinander kompiliert werden. Selbstverständlich müssen die Zielobjekte zur Laufzeit auf die Nachrichten reagieren, sei es durch eine geeignete Methodenimplementierung oder andere Maßnahmen, die unter 3.1.6 genauer beschrieben werden. Der Vorteil der physischen Unabhängigkeit geht also zu Lasten der Robustheit, da viele Fehler erst zur Laufzeit erkannt werden können. Um zumindest einigermaßen sicher Nachrichten verschicken zu können, lässt sich zur Laufzeit durch Introspektion herausfinden, ob ein Zielobjekt eine Nachricht verarbeiten kann. Typischerweise wird dies auf folgende Weise implementiert:

```
1 if([receiver respondsToSelector:@selector(myMessage)]) {
2    [receiver performSelector:@selector(myMessage)];
3 }
```

[34] Voraussetzung dafür sowie für die meisten weiteren Beispiele ist das Cocoa (Touch) Framework für Mac OS X bzw. iOS von Apple

[35] Die Methode existiert in verschiedenen Abwandlungen, u. a. mit Parametern und für Hintergrundoperationen

Dies wiederum macht den Code sehr unübersichtlich. Üblich ist es daher, weitestgehend auf die Standardnachrichtensyntax zurückzugreifen, die Warnungen durch den Compiler erzeugen kann. Nur an Stellen, an denen die lose Kopplung essenziell ist, sollte man bewusst eigene Nachrichtenobjekte erzeugen, die vom Compiler nicht überprüft werden. Obwohl also das Binden prinzipiell erst zur Laufzeit geschieht, wurde in Objective-C durch eine Konvention die Möglichkeit geschaffen, feste Schnittstellen zu simulieren und nur bei Bedarf auf lose Kopplung samt deren Risiken zu setzen. Dennoch bedeutet das bloße Vorhandensein der dynamischen Bindung, dass es kein Werkzeug geben kann, das alle Laufzeitabhängigkeiten von Klassen grafisch darstellt, da Methodenaufrufe und Nachrichten lediglich per Konvention zusammengehalten werden. Der tatsächliche Kontrollfluss ist aus dem statischen Code nicht in jedem Fall ablesbar.

3.1.3 Schnittstellenbeschreibung

Deklaration und Implementierung einer Klasse erfolgen in Objective-C typischerweise in zwei getrennten Dateien: Die Deklaration in der **Header-Datei** (`.h`), die Implementierung in der **Implementierungsdatei** (`.m`). Für die Implementierung ist eine vorherige Deklaration notwendig, sodass die Header-Datei stets in der Implementierungsdatei der gleichen Klasse importiert wird – daraus ergibt sich, wie in Abschnitt 2.1.2 beschrieben, dass sich kleinstmögliche physische Komponenten in der Regel aus zwei Dateien zusammensetzen. Die Klassendeklaration ist die üblichste, aber nicht die einzige Möglichkeit, eine Schnittstelle zu beschreiben.

Im Unterschied zu Java geschieht die Schnittstellenbeschreibung einer Klasse in Objective-C explizit in der Header-Datei. Ein „Interface" (dt.: Schnittstelle) in Java beschreibt hingegen eine allgemeine Schnittstelle ohne Zugehörigkeit zu einer bestimmten Klasse. Diese universellen Schnittstellenbeschreibungen heißen in Objective-C **Protocol** und können verpflichtende und optionale Methoden für Klassen, die das jeweilige Protocol implementieren, definieren. Zu den aus Java bekannten abstrakten Klassen, eine Mischung aus implementierten und nicht-implementierten Schnittstellen, gibt es in Objective-C kein Äquivalent.

Ein für diese Forschungsarbeit besonders relevanter Unterschied zu Java ist das Fehlen von Packages. Damit geht einher, dass man die **Sichtbarkeit** von Schnittstellen nicht auf Paketebene begrenzen kann – eine Schnittstelle ist entweder überall sichtbar („public") oder nur für eine Klasse und deren Unterklassen („private"[36]). Für die Modularisierung ist dies ein Problem, da es keine triviale Möglichkeit gibt, Sichtbarkeiten von Schnittstellen auf bestimmte Bereiche (Module) einer Anwendung zu beschränken. In Abschnitt 4.3.3 wird ein Ansatz vorgestellt, der dieses Problem mithilfe von Unterprojekten zu beheben versucht.

Eine weitere Folge der fehlenden Paketstrukturen ist das Fehlen von hierarchischen **Namensräumen**. Im Gegensatz zur Sichtbarkeitsproblematik hat sich hier eine simple Namenskonvention als Lösung etabliert, indem für jedes Projekt (insbesondere für Frameworks) ein eigener Präfix für alle Klassen gewählt wird. Namenskonflikte treten dadurch nur selten auf, dafür sind die `#import`-Anweisungen im flachen Namensraum oft übersichtlicher.

[36] Um Methoden nicht öffentlich sichtbar zu machen, sollten sie in der Implementierungsdatei deklariert werden

3.1.4 Key-Value Coding

Das Key-Value-Coding-Prinzip (KVC) wird durch das dynamische Binden von Nachrichten an Methoden ermöglicht. Im einfachsten Fall erlaubt es, Accessor-Methoden nicht durch Selectoren, sondern String-Schlüssel (engl.: Keys) zu adressieren. Die beiden Codeblöcke sind im Normalfall semantisch identisch:

```
// normale Nachrichtensyntax
NSNumber *i = @23;
[person setAge:i]; // Aufruf des Standard-Setters
i = [anotherPerson age]; // Aufruf des Standard-Getters
```

```
// Verwendung von KVC
NSNumber *i = @23;
[person setValue:i forKey:@"age"];
i = [anotherPerson valueForKey:@"age"];
```

Ein Vorteil der Benutzung von KVC ist die **lose Kopplung**, da die Keys keinerlei Überprüfung durch den Compiler unterliegen. Zudem lassen sich die Keys einfach zur Laufzeit generieren, z. B. durch Auslesen aus einer Konfigurationsdatei – in Abschnitt 4.4 (Modularisierung und Konfiguration des Datenmodells) wird darauf näher eingegangen.

Besonders interessant wird KVC bei geschachtelten Zugriffen auf Objekt-Properties (statt Keys werden dann Key-Paths verwendet, also durch Punkte konkatenierte Keys) und bei der Arbeit mit Objekten vom Typ `id`[37], die keinerlei Schnittstelleninformationen bereitstellen. Auch hier läuft man Gefahr, zur Laufzeit einen Fehler zu erzeugen, wenn das Zielobjekt den Key nicht verarbeiten kann. Es muss also zwischen einfachem Code und Laufzeitrobustheit abgewogen werden.

In der Implementierung der Klasse des Zielobjekts ist es möglich, die Werte (engl.: Values) zu validieren, bevor sie durch `setValue:forKey:` gesetzt werden. Wird durch `valueForKey:` auf ein Collection-Objekt (z. B. vom Typ NSArray oder NSDictionary) zugegriffen, lassen sich auch einfache, vordefinierte **Funktionen** in den Key-Path einbinden. Folgender Codeausschnitt sucht aus einem Array `peopleArray` von Personen-Objekten die jüngste Person heraus, unter der Annahme, dass jedes Objekt des Arrays eine `age`-Property besitzt:

```
id youngestPerson = [self valueForKeyPath:@"peopleArray.@min.age"];
```

Keys und Key-Paths müssen nicht zwingend auf Properties zugreifen und es gibt zahlreiche weitere Anwendungsmöglichkeiten, die in der Apple KVC-Dokumentation[38] nachgelesen werden können.

[37] `id`-Objekte haben keine Klassenzugehörigkeit – der Typ besagt lediglich, dass es sich um ein Objekt handelt, das Nachrichten empfangen kann

[38] `http://developer.apple.com/library/ios/documentation/cocoa/conceptual/KeyValueCoding/Articles/KeyValueCoding.html`

3.1.5 Datenhaltung mit Core Data

Core Data ist kein Sprachelement von Objective-C, sondern ein Framework zur **persistenten Speicherung von Objekten**. Dabei wird die zugrundeliegende Datenspeicherung transparent gehalten, derzeit sind Implementierungen für SQLite[39], XML-Dateien und ein Binärformat auswählbar.

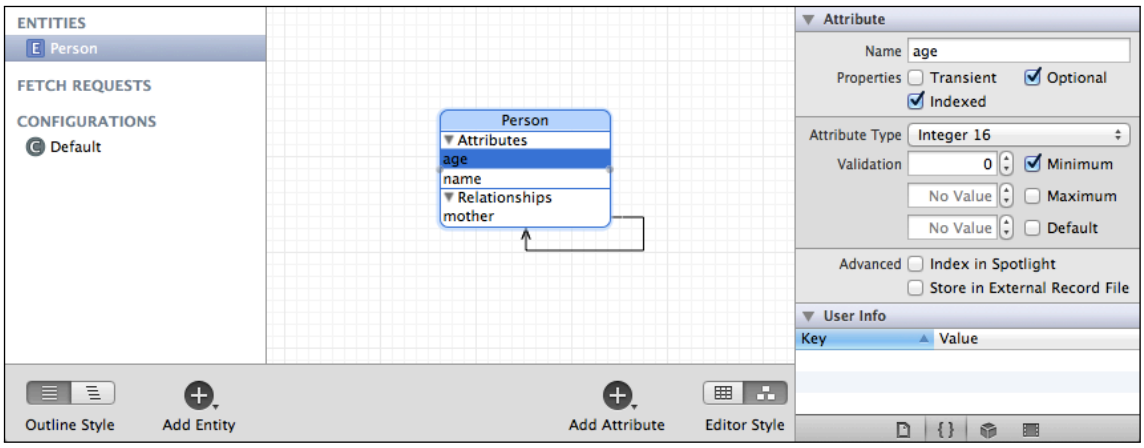


Abbildung 3.1: Core-Data-Modell im Xcode-Editor

Das Datenmodell kann in Xcode mithilfe einer in Abbildung 3.1 dargestellten grafischen Oberfläche (GUI) erstellt und verwaltet werden. Core-Data-Abfragen werden nicht mit SQL, sondern durch Objekte vom Typ `NSFetchRequest` formuliert. Dabei können neben der Ergebnisentität und Prädikaten zum Filtern und Sortieren weitere Abfragedetails angegeben werden. Core Data verlangt nicht, dass zu jeder im Datenmodell definierten Entität eine Objective-C-Klasse existiert. In diesem Fall sind in der Rückgabemenge einer Abfrage generische Objekte vom Typ `NSManagedObject` enthalten. Um schließlich auf die Attribute der Entitäten zugreifen zu können, wird das im vorangegangenen Abschnitt beschriebene **KVC-Prinzip** verwendet:

```
1  // Ausgabe der Namen aller persistent gespeicherten Personen
2
3  // Abfrage soll Personen-Entitäten liefern
4  NSFetchRequest *req = [[NSFetchRequest alloc] initWithEntityName:@"Person"];
5  NSArray *results = [self.managedObjectContext executeFetchRequest:req
6                                                               error:nil];
7  // iteriere über generische Objekte
8  for (NSManagedObject *person in results) {
9     NSLog(@"Name: %@", [person valueForKey:@"name"]); // Zugriff via KVC
10 }
```

Alternativ können Entitäten mit Klassen und Attribute mit Properties verbunden werden, sodass ein typsicherer Zugriff im Code gewährleistet werden kann.

[39] `http://www.sqlite.org/`

Die Code-Gerüste der Entitätsklassen lassen sich mit Xcode und Core Data automatisch erzeugen, jedoch wird keine Synchronisierung von Änderungen am Datenmodell oder im Code unterstützt. Eine tiefgreifende Einführung in Core Data ist in der Apple-Dokumentation[40] einsehbar.

3.1.6 Categories und Proxies

Objective-C stellt mit den sog. **Categories** ein Sprachkonstrukt zur Verfügung, zu dem es in vielen anderen Programmiersprachen kein Äquivalent gibt. Mit Categories lassen sich, ähnlich wie mit Unterklassen, Methoden bestehender Klassen **überschreiben** oder neue Methoden **hinzufügen**. Im Unterschied zu Unterklassen geschieht dies bei Categories aber auf der gleichen Vererbungshierarchieebene – man ändert bestehende Klassen anstatt neue hinzuzufügen. Damit einher geht, dass man keine `super`-Implementierung der zu ändernden Methode und Klasse aufrufen kann, da man eben diese überschreibt. In untenstehendem Beispiel wird die bestehende Klassenmethode `greenColor` von `UIColor` überschrieben:

```
1 // Name der Category:   MyOwnGreen
2 // (der Header wurde aus Platzgründen weggelassen)
3 @implementation UIColor (MyOwnGreen)
4
5 // greenColor von UIColor wird überschrieben
6 + (UIColor *)greenColor {
7    // return [super greenColor]; // nicht möglich!
8    return [UIColor colorWithRed:0.1f green:1.0f blue:0.1f alpha:0.5f];
9 }
```

Da Categories zur Laufzeit eingebunden werden, können wie in dem angegebenen Beispiel auch Methoden von Framework-Klassen überschrieben werden, zu denen der Quelltext nicht vorliegt. Im Beispielfall wird die Zusammensetzung der Farbe Grün im gesamten Geltungsbereich der Category verändert, sodass auch Framework-GUI betroffen sein kann. Aufgrund der Gefahr, transparente Seiteneffekte der Originalmethode zu überschreiben, rät Apple vom Überschreiben bereits vorhandener Methoden ab und lässt den Compiler eine Warnung erzeugen. Zwar suggeriert Xcodes Code-Vervollständigung, dass der Geltungsbereich durch den `#import` der Category bestimmt wird, allerdings gelten Categories durch die Einbindung zur Laufzeit prinzipiell global, sofern sie Teil einer Anwendung sind.

Objekte vom Typ `NSProxy`, im Folgenden **Proxies** genannt, erlauben in erster Linie das spätmöglichste Initialisieren (engl.: Lazy initialisation) von Objekten, die einen hohen Ressourcenbedarf haben. Solange die eigentlichen Objekte nicht unbedingt geladen werden müssen, kann ein Proxy-Objekt die eingehenden Nachrichten verwalten und über deren Verarbeitung oder Weiterleitung bestimmen. Verschiedene Drittanbieter-Frameworks[41] nutzen Proxies zur Realisierung von **aspektorientierter Programmierung** (AOP) unter iOS. Aspekte sind Funktionalitäten, die über die ganze Anwendung verteilt benutzt werden (engl.: Cross-cutting concerns), aber unabhängig

[40] `http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/CoreData/cdProgrammingGuide.html`

[41] z. B. Molnar: `https://github.com/moszi/AOP-in-Objective-C/`

von der eigentlichen Anwendungslogik sind [KLM+97]. Häufig genannte Beispiele sind Logging, Transaktionalität oder Sicherheitsmechanismen wie Authentifizierung, aber auch mandantenspezifische Anpassungen können u. U. als Aspekte implementiert werden. Sinn und Zweck von AOP ist es, Aspekte zentral zu implementieren und zentral zu bestimmen, an welchen Punkten Aspekte eingebunden werden sollen, sodass der normale Anwendungscode vollkommen unverändert bleibt.

Während die erwähnten AOP-Frameworks unter iOS zwar eine relativ zentrale Implementierung von Aspekten zulassen, bleibt der eigentliche Anwendungscode nicht unverändert, da anstatt der ursprünglichen Objekte spezielle Proxy-Objekte genutzt werden müssen. Dieses Problem lässt sich wiederum mit Categories beheben, indem man die Initialisierungsmethoden der Originalobjekte in Categories überschreibt und dort die entsprechenden Proxy-Objekte anlegt und zurückgibt[42]:

```
@implementation UIColor (Proxy)

- (id)init {
   // super greift in diesem Fall nicht auf UIColor,
   // sondern auf die Oberklasse NSObject zu
   self = [super init];
   AOPProxy *proxy;
   if (self) {
        // das Proxy-Objekt sorgt für die Einbindung der Aspekte
       proxy = [[AOPProxy alloc] initWithInstance:self];
   }
   return proxy;
}

@end
```

Es bleibt festzuhalten, dass Categories und Proxies neben vielen anderen Möglichkeiten auch eine einfache Variante aspektorientierter Programmierung erlauben. Andererseits können beide Sprachkonstrukte zu schwer lesbarem und schwer zu debuggendem Code führen, weil Methodenimplementierungen überschrieben (Categories) oder übergangen (Proxies) werden können, ohne dass sich in der implementierenden Klasse ein Hinweis darauf befindet. Die technische Grundlage für Categories und Proxies ist die dynamische Bindung in Objective-C, die unter 3.1.2 vorgestellt wurde.

[42] Das angegebene Konzept ist stark vereinfacht und lässt einige Probleme außer Acht – es dient lediglich als grobes Konzept und Denkanstoß

3.2 XCODE

Die seit 2003 von Apple angebotene integrierte Entwicklungsumgebung (IDE) Xcode[43] hat mit der Einführung des iPhones stark an Bedeutung und Funktionsumfang zugenommen. Seit Juni 2010 ist **Xcode 4** verfügbar[44], das eine komplett überarbeitete Nutzeroberfläche bietet, in die zuvor einzeln angebotene Werkzeuge integriert worden sind. Die wichtigsten dieser Werkzeuge werden in Abschnitt 3.2.1 kurz vorgestellt. Im darauffolgenden Abschnitt 3.2.2 wird die allgemeine Struktur eines iOS-Xcode-Projekts grob umrissen und der Build-Prozess beschrieben. Inwiefern dieser konfigurierbar ist, wird abschließend in Abschnitt 3.2.3 erklärt.

3.2.1 Funktionsumfang

Wie die meisten IDEs unterstützt Xcode das Schreiben, Kompilieren, Ausführen und Testen bzw. Debuggen von **Code**. Auch SCM-Werkzeuge[45] sind standardmäßig integriert. Damit mobile Anwendungen auch ohne Endgerät testbar sind, enthält Xcode einen **Simulator**, der verschiedene Geräte- und iOS-Versionen sowie Gesten und Sensoren simulieren kann. Um Anwendungen nicht nur auf ihre Korrektheit zu überprüfen, sondern auch **Performance**-Kriterien wie Speicher- und CPU-Verbrauch zu analysieren, ist die Werkzeug-Sammlung Instruments im Xcode-Paket enthalten.

Eine Besonderheit ist die Vielzahl der verschiedenen Editoren, die in Xcode enthalten sind. Je nach zu editierender Datei wählt Xcode den passenden Editor. Während **Interface Builder** in früheren Xcode-Versionen ein eigenständiges Werkzeug war, ist es nun der Standard-Editor für GUI-Dateien (XIBs/`.xib`), die sich durch Drag & Drop und Konfigurierungsoptionen erstellen lassen. Seit Xcode 4.2 lassen sich nicht nur einzelne Bildschirminhalte in GUI-Dateien auslagern, sondern mithilfe von Storyboard-Dateien (`.storyboard`) auch die Verbindungen zwischen diesen. Sowohl XIB- als auch Storyboard-Dateien basieren auf XML und können effektiv von SCM-Werkzeugen verwaltet werden. Der Storyboard-Editor ist beispielhaft in Abbildung 3.2 dargestellt. Ein weiteres Beispiel für einen speziellen Editor ist der in Abschnitt 3.1.5 vorgestellte Core-Data-Editor.

Leider verfügt Xcode nur über eingeschränkte **Refactoring**-Möglichkeiten. Während die projektweite Umstellung auf neue Objective-C-Versionen und damit einhergehende Features gut unterstützt wird, fehlen wichtige allgemeine Refactoring-Möglichkeiten wie das globale Umbenennen von Methoden mit geänderten Parametern und die Analyse von Abhängigkeiten zwischen Klassen und Dateien. Auch das automatische Bereinigen nicht genutzter `#import`-Anweisungen wird nicht unterstützt. Da Xcode von Drittanbietern nicht ohne Weiteres erweiterbar ist[46], besteht eine hohe Abhängigkeit zwischen Xcode-Nutzern und Apple. Aufgrund der hohen Aktualisierungsraten von Xcode und wichtigen Alleinstellungsmerkmalen wie Storyboards und dem Core-Data-Editor sind alternative IDEs rar, zumal Xcode kostenlos verfügbar ist und direkt nach der Installation fertig konfiguriert ist.

[43] `https://developer.apple.com/xcode/`
[44] Die derzeit aktuelle Version ist 4.5.2, Stand November 2012
[45] Git und SVN
[46] Die Plugin-API ist undokumentiert und kann jederzeit von Apple geändert werden – ein Beispiel-Plugin ist hier verfügbar: `https://github.com/sap-production/xcode-ide-maven-integration/`

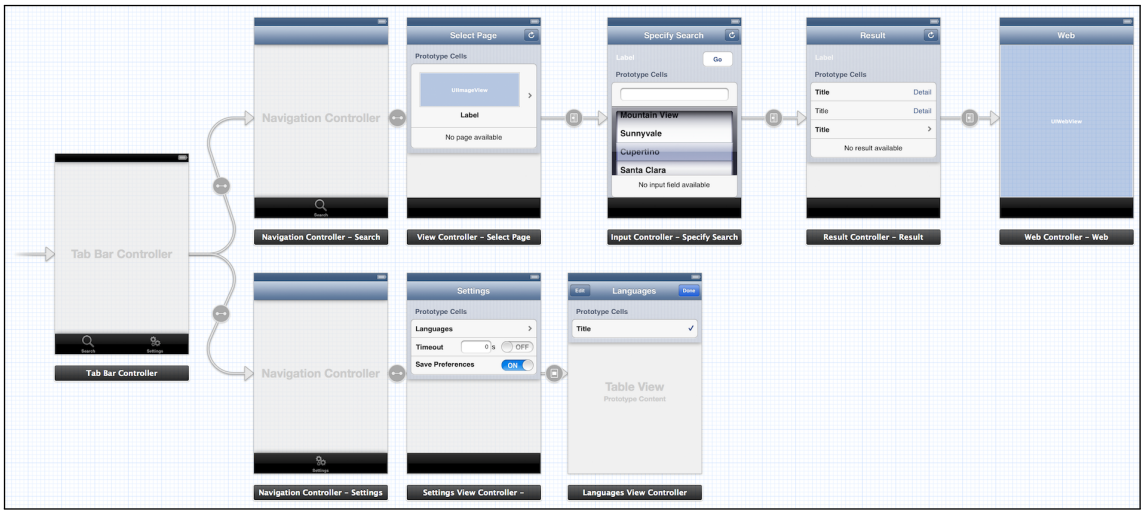


Abbildung 3.2: Storyboard im Xcode-Editor

3.2.2 Projektstruktur und Build-Prozess

Jedes Xcode-Projekt verfügt neben dem Quellcode und den Ressourcen über eine XML-basierte **Projektdatei** (`.xcodeproj`)[47]. In der Projektdatei sind alle zum Projekt gehörigen Dateien referenziert. Nutzt man die standardmäßigen Xcode-Gruppen zur hierarchischen Strukturierung, wird dabei vollständig von der Lage der Dateien im Dateisystem abstrahiert. Dies hat auch zur Folge, dass der Namensraum flach ist und zur eindeutigen Referenzierung alle Dateinamen disjunkt sein müssen. Alternativ lassen sich auch Ordner in Xcode-Projekte einpflegen, die die Lage einer Datei im Dateisystem reflektieren und auch mit dem Dateisystem synchronisiert werden. Allerdings müssen dann alle Header und Ressourcen mit einem relativen Pfad adressiert werden.

Jedes Projekt sollte mindestens ein **Target** besitzen. Während das Projekt selbst nicht gebaut werden kann, enthalten Targets alle nötigen Informationen, um das Projekt (oder Teile davon) zu kompilieren und zu verlinken. Die kompilierte Form eines Targets wird **Produkt** genannt und kann u. a. eine fertige App oder eine statische Bibliothek sein. Alle Targets werden in der Projektdatei gespeichert und können ihre eigenen Build-Parameter besitzen – das Zusammenspiel von Target- und Projekteinstellungen wird unter Abschnitt 3.2.3 genauer betrachtet.

Ein Target kann für verschiedene Zwecke gebaut werden, u. a. zum Testen, Analysieren oder Archivieren, analog zu den unter 2.2.2 eingeführten **Lifecycles** von Maven. In Xcode sind sechs solcher Lifecycles[48] fest eingebaut:

- Build – Einfaches Bauen von Targets
- Run – Bauen und Ausführen im Simulator oder auf einem Gerät
- Test – Ausführen von Unit-Tests

[47] Genau genommen handelt es sich dabei um ein Projektpaket, das neben den angegebenen Informationen auch nutzerspezifische Einstellungen speichert – `https://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/AboutBundles/AboutBundles.html`

[48] Diese Build-Zwecke haben in Xcode keinen speziellen Namen, daher wird in dieser Arbeit auf den Maven-Begriff zurückgegriffen

- Profile – Bauen und Ausführen mit Instruments
- Analyze – Bauen und statische Codeanalyse
- Archive – Bauen für den Vertrieb, z. B. im App Store

Jeder Lifecycle hat andere, projektweite Optionen, die sich nicht auf den Build-Vorgang, sondern in erster Linie auf die Ausführungsart des gebauten Produkts beziehen. In Abbildung 3.3 sind die Optionen des Profile-Lifecycles dargestellt. Enthält ein Projekt mehrere Targets, lässt sich bestimmen, welche Targets bei welchem Lifecycle gebaut werden sollen. Die Optionen aller sechs Lifecycles werden in einem **Schema** (engl.: Scheme) abgespeichert. Bei Bedarf können mehrere Schemata angelegt werden, um die Optionen für alle Lifecycles schnell und konsistent zu ändern.

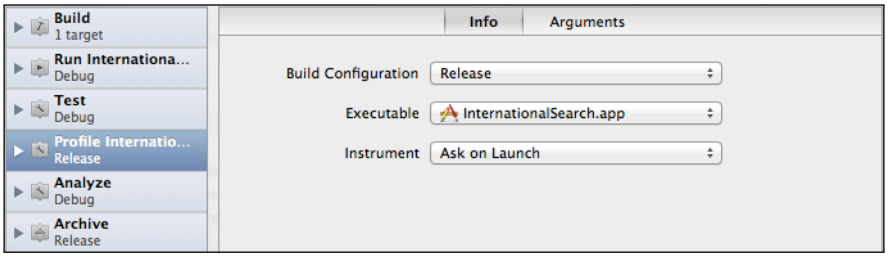


Abbildung 3.3: Optionen des Profile-Lifecycles

Der eigentliche Build-Vorgang wird in sog. **Build-Phasen** (engl.: Build Phases) unterteilt. Ein typisches iOS-Projekt hat dabei vier Phasen:

- Target Dependencies – Auflösen von Abhängigkeiten
- Compile Sources – Quelldateien kompilieren
- Link Binary with Libraries – Statische und dynamische Bibliotheken verlinken
- Copy Bundle Resources – Ressourcen werden in das gebaute Produkt eingefügt, um sie dort referenzieren zu können

Die sog. **Build-Regeln** (engl.: Build Rules) legen fest, welcher Compiler für welchen Dateityp zuständig ist. In den meisten Fällen können sowohl die Build-Phasen als auch die Build-Regeln unverändert bleiben, lediglich die **Build-Einstellungen** (engl.: Build Settings) werden angepasst. Die Build-Einstellungen sind Compiler- und Linker-Parameter und können alle Build-Phasen beeinflussen. Die Anpassungsmöglichkeiten werden im nächsten Abschnitt beschrieben.

3.2.3 Konfigurationsmöglichkeiten

Alle zuvor genannten Abschnitte des Build-Prozesses lassen sich innerhalb von Xcode weiter konfigurieren. Lifecycles können über vorgegebene Optionen angepasst werden, zusätzlich können Skripte zur Ausführung vor und nach jedem Lifecycle eingefügt werden. Auch zwischen den

Build-Phasen können Skripte einfach per GUI als weitere Phasen hinzugefügt werden, die zeitliche Anordnung aller Phasen ist einstellbar. Für Dateitypen, die Xcode nicht bekannt sind, lassen sich eigene Build-Regeln festlegen. In der Praxis am relevantesten sind allerdings die **Build-Einstellungen**.

Diese Einstellungen zum Bauen einer Anwendung werden auf mindestens zwei Stufen verteilt. Build-Einstellungen des Projekts (Stufe 1) gelten für alle Targets, sofern sie dort nicht überschrieben werden (Stufe 2). Zusätzlich kann eine Konfigurationsdatei im Projekt angegeben werden, die genau zwischen diesen beiden Stufen liegt. Build-Einstellungen können wiederum für verschiedene Build-Konfigurationen definiert werden, standardmäßig werden von Xcode die Build-Konfigurationen Debug und Release angelegt. Wie die letztlich gültige Einstellung ermittelt wird, stellt Xcode wie in Abbildung 3.4 gezeigt grafisch dar.

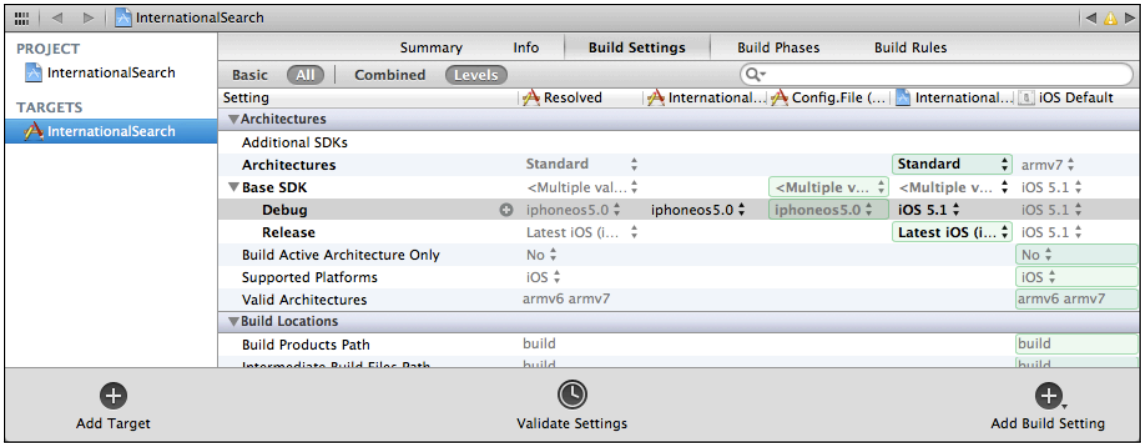


Abbildung 3.4: Vererbung der Einstellungen von rechts nach links

Im Projekt ( ) ist für die Debug-Konfiguration der Einstellung „Base SDK“ der Wert `iOS 5.1` gesetzt. Die Konfigurationsdatei (Config.File) überschreibt diesen Wert mit `iphoneos5.0`, der letztlich gültig ist (gekennzeichnet durch eine grüne Box), da er von dem Target (International...) nicht verändert wird. Sollte entlang dieser Hierarchie kein Wert angegeben sein, gilt die iOS-Default-Spalte ( ) am rechten Rand. Compiler-Flags können darüber hinaus pro Datei gesetzt und überschrieben werden.

Während Objective-C-Quellcode auch unabhängig von Xcode kompiliert werden kann, ist das iOS SDK und der Archivierungsprozess für App-Store-Anwendungen eng an Xcode gekoppelt. Der Xcode Build-Prozess kann allerdings auch über die **Kommandozeile** angestoßen werden, was es externen Werkzeugen wie Maven ermöglichen kann, iOS-Anwendungen zu bauen. Eine Xcode-Installation ist in diesem Fall dennoch unerlässlich.

Xcode selbst unterstützt auch Unterprojekte, deren Produkte von anderen Projekten referenziert werden können. Damit lässt sich das in Abschnitt 2.2.1 (Konzeption und Refactoring) eingeführte „Levelize Build“-Pattern umsetzen. Dies wird genauer in Abschnitt 4.3.3 (Workspaces und Subprojekte) behandelt werden.

3.3 iOS-RUNTIME-UMGEBUNG

Auf die iOS-Runtime-Umgebung haben Anwendungsentwickler keinerlei Einfluss, daher soll sie an dieser Stelle nur kurz beschrieben werden. Das iOS-Betriebssystem basiert auf Darwin[49], also dem selben **UNIX-System**, das auch Mac OS X zugrunde liegt [HC11, Kapitel 2 Mac OS X and iOS]. Im Gegensatz zu Mac OS X ist es iOS-Nutzern und deren Anwendungen allerdings nicht möglich, per Kommandozeile oder Systemfunktionen direkt mit dem Betriebssystem zu interagieren. Stattdessen laufen alle Anwendungen in einer vom Betriebssystem verwalteten Sandbox, die nicht nur den Zugriff auf anwendungsfremde Dateien, sondern auch direkte Systemaufrufe (engl.: System Calls) unterbindet. Alle Zugriffe auf Hardwarekomponenten (Sensoren, Peripheriegeräte,...) und Systemressourcen (Dateisystem, Threads,...) können nur über öffentliche Schnittstellen der im iOS SDK enthaltenen Frameworks erfolgen. Systemnahe Frameworks sind in der Regel in C geschrieben, während die Frameworks der höheren Schichten, insbesondere Cocoa Touch, über Objective-C-Schnittstellen verfügen.

Die Hauptaufgabe der iOS-Runtime[50] ist das dynamische Binden von Objective-C-Nachrichten an Methoden sowie das dynamische Laden von Klassen. Neben Objective-C (und damit automatisch C) unterstützt die Runtime-Umgebung C++ und Objective-C++[51].

Ohne das iOS-Betriebssystem zu modifizieren, lässt sich auch die Runtime-Umgebung nicht modifizieren. Dies lässt sich je nach Sichtweise als Einschränkung oder Sicherheitsmechanismus interpretieren. Eine Konsequenz dessen ist, dass alternative Programmiersprachen wie Java oder gar Laufzeitsysteme wie OSGi unter iOS nicht lauffähig sind. Der eingeschränkte Zugriff auf das Dateisystem sowie das Sandbox-Prinzip verhindern zudem die Nutzung eigener dynamischer Bibliotheken[52].

3.4 APPLE GUIDELINES

Bevor eine App im Apple App Store vertrieben werden kann, wird die Anwendung von Apple getestet und muss dabei einer Reihe von **Anforderungen** genügen. Um diesen Prozess möglichst transparent zu gestalten, hat Apple die **App Store Review Guidelines** im Internet veröffentlicht [app12a]. Wird gegen eine der Regeln verstoßen, wird die App abgewiesen und der Grund dem Entwickler per Email mitgeteilt. Kann dieser den Ablehnungsgrund beheben, kann er die App erneut zur Prüfung einreichen. Die im Folgenden benutzten Nummerierungen beziehen sich auf das Inhaltsverzeichnis der Review Guidelines.

Die Guidelines enthalten viele selbstverständliche Anforderungen – Apps dürfen nicht abstürzen (2.1), keine auffälligen Bugs enthalten (2.2), Beschreibung und tatsächliche Funktionalität müssen konsistent sein (2.3), es dürfen keine privaten APIs benutzt werden (2.5) und die Sandbox-

[49] `http://darwinbuild.macosforge.org/`

[50] `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html`

[51] `http://cocoadev.com/wiki/ObjectiveCPlusPlus`

[52] Ausgenommen sind die von Apple bereitgestellten und dem Betriebssystem bekannten dynamischen Framework-Libraries

Restriktion darf nicht umgangen werden (2.6). Allerdings enthält das Dokument auch Anforderungen, die weniger offensichtlich sind. Die für Business-Apps und speziell modularisierte Apps kritischsten Punkte sind Folgende:

- 2.7 „*Apps that download code in any way or form will be rejected*" – Dies impliziert, dass die Anwendung zum Zeitpunkt der Auslieferung alle Module enthalten muss, da Module nicht nachgeladen werden dürfen
- 2.20 „*Developers 'spamming' the App Store with many versions of similar Apps will be removed from the iOS Developer Program*" – Fraglich ist, ob dieser Punkt mandantenspezifische Versionen umfasst
- 11.1 „*Apps that unlock or enable additional features or functionality with mechanisms other than the App Store will be rejected*" – Dieser Punkt schließt aus, dass Module serverseitig aktiviert werden dürfen; allerdings ist eine Aktivierung per In-App Purchase[53] denkbar
- 11.11 „*In general, the more expensive your App, the more thoroughly we will review it*" – Aufgrund der hohen Entwicklungskosten von Business-Apps muss man mit einer strengen Auslegung der Richtlinien rechnen

Beachtet man alle Regeln, dürfen weder zahlreiche mandantenspezifische Versionen der App angeboten werden, noch darf eine Grundanwendung, die alle Module enthält, serverseitig konfiguriert werden, um ausgewählte Module zu aktivieren. Ein weiteres Problem stellen die **Human Interface Guidelines** (HIG) [app12b] dar, deren Beachtung nach 10.1 der Review Guidelines verpflichtend ist. In Kapitel 4 der HIG wird geraten, dass eine Anwendung einen konzentrierten Fokus haben soll. Übersteigt die Anwendung einen bestimmten Funktionsumfang, sollte die Zerteilung in mehrere kleine Apps erwogen werden. Dieser Ansatz ist für komplexe Business-Anwendungen generell schwer umsetzbar und von den aktuellen Kunden der SALT Solutions GmbH unerwünscht.

Es zeigt sich, dass die Einreichung einer mehrmandantenfähigen Business-App in den Apple App Store zumindest problembehaftet ist. Im folgenden Abschnitt werden daher alternative Vertriebsmethoden beschrieben.

3.5 ALTERNATIVE VERTRIEBSMODELLE

Erst seit September 2012 ist Apples **B2B**-Vertriebsmodell[54] in Deutschland verfügbar. Die Vertriebsart ähnelt dabei der des öffentlichen App Stores. Der Entwickler muss für das kostenpflichtige Apple Developer Program[55] ($ 99/Jahr) eingeschrieben sein, um für den Verkauf bestimmte, signierte Anwendungen zu bauen und über iTunes Connect[56] bei Apple einzureichen. Damit die

[53] `https://developer.apple.com/appstore/in-app-purchase/index.html`
[54] `https://developer.apple.com/news/?id=09042012a`
[55] `https://developer.apple.com/programs/ios/`
[56] `https://itunesconnect.apple.com/`

Anwendung nicht öffentlich sichtbar wird, muss lediglich die Option „Custom B2B App" in iTunes Connect ausgewählt werden. Anschließend können vom Entwickler Kunden, die über ihre Apple ID[57] identifiziert werden, in eine Liste eingetragen werden. Ausschließlich diese Kunden können die App im App Store finden. Wichtig ist, dass die Anwendung zuvor genau wie im öffentlichen App Store von Apple geprüft wird. In [Rah12] empfiehlt Apple jedoch ausdrücklich, für jeden Kunden eine einzelne, auf dessen Wünsche und Workflows zugeschnittene App einzureichen. Daher darf davon ausgegangen werden, dass insbesondere die Punkte 2.20 (Einreichung ähnlicher Apps) und 10.1 (HIG) der unter Abschnitt 3.4 beschriebenen Guidelines weniger streng ausgelegt werden.

Um B2B-Apps kaufen zu können, muss ein Kunde zuvor eine Mitgliedschaft im Apple **Volume Purchase Program**[58] (VPP) abgeschlossen haben. Diese Mitgliedschaft ist kostenlos, allerdings nur Unternehmen zugänglich, die über eine Dun-&-Bradstreet-Registrierungsnummer (DUNS[59]) verfügen. Über das VPP kann der Kunde nicht nur alle öffentlichen Apps in beliebigen Mengen kaufen, sondern auch B2B-Apps, bei denen seine Apple ID in iTunes Connect vom Entwickler hinterlegt worden ist. Beim Kauf einer App wird im Gegensatz zum normalen App Store keine ausführbare Anwendung, sondern eine Liste von Gutscheincodes heruntergeladen. Die **Installation** der Anwendung kann auf vier verschiedene Weisen durchgeführt werden [Rah12]:

- Manuell/Eingabe – Der Nutzer des Endgeräts gibt einen Gutscheincode im App Store ein und kann die Anwendung laden
- Manuell/URL – Pro Gutschein wird eine URL erzeugt, die dem Nutzer per Email, SMS oder Webseite (z. B. Unternehmensportal im Intranet) mitgeteilt wird; der Aufruf der URL auf dem Endgerät entwertet den Gutscheincode und installiert die App
- Mobile Device Management (MDM)[60] – servergesteuerte Konfiguration von iOS-Geräten; erfordert eigene Serverkomponente
- Apple Configurator[61] – Die kostenlose Mac-Anwendung erlaubt es, per USB angeschlossene iOS-Geräte einheitlich zu konfigurieren und unterstützt die Installation von Anwendungen durch Gutscheincodes

Als Alternative steht Unternehmen das iOS Developer **Enterprise Program**[62] für $ 299 pro Jahr zur Verfügung. Grundvoraussetzung ist auch hier eine vorhandene DUNS. Im Gegensatz zum B2B-Ansatz ist das Programm darauf ausgerichtet, dass die Apps innerhalb des Unternehmens entwickelt werden, in dem sie auch eingesetzt werden. Technisch wird dies mithilfe von Zertifikaten gelöst: Jedes unternehmenseigene Gerät benötigt ein Enterprise Distribution Provisioning Profile, und alle unternehmenseigenen Apps werden mit einem Enterprise Distribution Certificate signiert. Die Profile können auf beliebigen Geräten installiert werden[63], sind aber nur so

[57] `http://www.apple.com/support/appleid/` – Apple IDs sind Nutzerkonten bei Apple, die in der Regel aus einer Email/Passwort-Kombination bestehen

[58] `https://developer.apple.com/programs/volume/`

[59] `http://www.dnb.com/`

[60] `http://images.apple.com/iphone/business/docs/iOS_MDM_Mar12.pdf`

[61] `http://itunes.apple.com/us/app/apple-configurator/id434433123/`

[62] `https://developer.apple.com/programs/ios/enterprise/`

[63] Auch außerhalb des Unternehmens, was aber bei Bekanntwerden den Ausschluss aus dem Enterprise Program zur Folge hat

lange gültig wie das Zertifikat. Da die Gültigkeit des Zertifikats zeitlich beschränkt ist, muss das Enterprise Program jährlich verlängert werden, damit installierte Apps weiterhin funktionieren.

Problematisch ist, dass das Zertifikat an die Apple ID des Enterprise Programs gebunden ist. Damit eine Anwendung außerhalb eines Unternehmens entwickelt und vertriebsfertig gebaut werden kann, müssen eigentlich vertrauliche Informationen[64] an das Entwicklerunternehmen weitergegeben werden. Dieses Vorgehen ist in der Praxis durchaus gängig, widerspricht aber der eigentlichen Intention des Enterprise Programs.

Diese Differenz spiegelt sich auch im fehlenden Abrechnungsmodell wieder. Während der Kunde beim B2B-Modell eine Lizenz pro Endgerät erwerben muss, bekommt er beim Enterprise Program eine universal ausführbare App. Oft als Vorteil gesehen wird dagegen die Tatsache, dass diese Apps zu keinem Zeitpunkt von Apple geprüft oder freigegeben werden müssen. Die Installation kann wie beim B2B-Modell auf vielfältige Art und Weise durchgeführt werden. Wird die Anwendung auf einem im Unternehmensnetz zugänglichen Webserver abgelegt, reicht eine URL zur automatischen Installation. Alternativ kann die Installation servergesteuert per MDM oder kabelgebunden per Apple Configurator erfolgen.

Ausgehend von der Intention des Programms ist für ein Unternehmen, das ähnliche Apps an mehrere Mandanten vertreibt, das B2B/VPP-Programm das Passendere. Pro mandantenspezifischer App kann ein Kundenkreis durch Apple IDs genau definiert werden, das Einreichen einer App ist identisch zu dem Prozess im App Store, und die Abrechnung erfolgt pro installierter App. Möchte man mit dem Mandanten einen Festpreis für die Entwicklung inklusive einer beliebigen Anzahl Lizenzen vereinbaren, kann die Anwendung kostenlos angeboten werden. Für das eingangs erwähnte Szenario sollte das Enterprise Program nur dann als Alternative in Erwägung gezogen werden, wenn die App aus funktionalen Gründen den Review Guidelines nicht genügen kann oder das VPP-Programm im Land des Mandanten nicht verfügbar ist[65].

3.6 ZUSAMMENFASSUNG

Sowohl bei iOS als auch bei Mac OS X setzt Apple mit Objective-C auf eine **Programmiersprache**, die zwar vergleichsweise alt, aber außerhalb der genannten Plattformen kaum verbreitet ist. Im ersten Teil dieses Kapitels wurden Besonderheiten der Sprache vorgestellt. Eine zentrale Rolle spielt die dynamische Bindung, die generische und flexible Programmiermodelle wie Key-Value Coding und Proxies zulässt. Andererseits kann die bewusste Umgehung der Typsicherheit zu instabilen Anwendungen führen, da Fehler erst zur Laufzeit entdeckt werden können. Problematisch für modulorientiertes Entwickeln ist das Fehlen von Packages, sodass die Sichtbarkeit von Schnittstellen nicht auf bestimmte Anwendungsbereiche beschränkt werden kann.

Als Hauptentwicklungswerkzeug dient die von Apple bereitgestellte **IDE** Xcode, die sich mittlerweile in der vierten Generation befindet. Positiv hervorzuheben sind v. a. die einfache Gestaltung der GUI mit Interface Builder oder dem Storyboard-Editor sowie Projekt-Templates und Standardkonfigurationen, die ohne weiteres Zutun ein lauffähiges App-Grundgerüst bereitstellen. Der

[64] Das Zertifikat selbst sowie der private Schlüssel
[65] Derzeit ist das VPP in 10 Ländern verfügbar: `http://enroll.vpp.itunes.apple.com/`

Build-Prozess kann durch zahlreiche Optionen oder eigene Skripte angepasst werden. Problematisch ist, dass Xcode nicht von Drittanbietern erweitert werden kann und Entwickler somit stark von Apple abhängen. Alternative IDEs sind zwar vorhanden, aber nicht weit verbreitet und eher als Ergänzung denn als Ersatz von Xcode anzusehen.

Wenig Spielraum bietet die unter Abschnitt 3.3 vorgestellte **Runtime-Umgebung**, die auf unmodifizierten Endgeräten nur C-basierte Programmiersprachen unterstützt. Das Sandbox-Prinzip ist zwar aus Sicherheitsaspekten positiv zu bewerten, verhindert jedoch anwendungsübergreifende Laufzeitdienste und dynamische Bibliotheken.

Im letzten Teil des Kapitels wurden Alternativen zum App Store als **Vertriebskanal** vorgestellt, weil die unter Abschnitt 3.4 vorgestellten Bedingungen zur Aufnahme in den App Store für Business-Anwendungen teilweise zu restriktiv sind. Einige Aufnahmebedingungen werden beim B2B-Vertriebsmodell aufgeweicht, während das Enterprise Program keinerlei Überprüfung seitens Apple vorsieht. Es ist allerdings kostenintensiver und die Intention des Vertriebsmodells weicht oft vom tatsächlichen Einsatzszenario ab.

4 MEHRMANDANTENFÄHIGKEIT MIT iOS

Im folgenden Kapitel werden konkrete Konzepte zum Erlangen von Mehrmandantenfähigkeit erläutert und kurz bewertet. An dieser Stelle wird noch kein ganzheitliches Konzept eingeführt, sondern vielmehr eine Palette verschiedener, kombinierbarer Ansätze zusammengestellt, die sich in ihrer Komplexität und Flexibilität stark unterscheiden.

In den Abschnitten 4.1 und 4.2 werden einfach zu implementierende, aber in ihrer Funktionalität limitierte Möglichkeiten zur mandantenspezifischen Anpassung einer App untersucht. Den Hauptteil dieses Kapitels füllt Abschnitt 4.3 aus, in dem modulbasierte Ansätze diskutiert werden. Anschließend wird der Konfigurierbarkeit des Datenmodells ein eigener Abschnitt (4.4) gewidmet, bevor das Kapitel mit einer Zusammenfassung abschließt. Die in diesem Kapitel beschriebenen Lösungsmöglichkeiten werden anschließend in Kapitel 5 zu einem Gesamtkonzept für einen Prototyp zusammengesetzt.

4.1 TARGET-BASIERTE ANSÄTZE

Targets bilden in Xcode die Grundlage, um aus einer gemeinsamen Codebasis verschiedene Produkte (Apps) zu bauen. Damit sich die Produkte unterscheiden, also mandantenspezifisches Verhalten implementieren, kann die Unterscheidung entweder im Programmcode vorgenommen werden oder über Konfigurationsdateien gesteuert werden. Beide Varianten werden in den folgenen Abschnitten diskutiert.

4.1.1 Programmatisch

Zu jedem iOS-Target gehört neben den unter Abschnitt 3.2.2 vorgestellten Build-Parametern eine das Produkt beschreibende Konfigurationsdatei, die Key-Value-Paare enthält. Der Name der im Folgenden kurz **Info-PLIST** genannten Datei setzt sich standardmäßig aus dem Target-Namen, dem Suffix `-Info` und der Dateiendung `.plist`[66] zusammen[67]. Wird dem Projekt ein neues iOS-Target hinzugefügt, legt Xcode automatisch eine neue Info-PLIST für dieses Target an. Da sich die Informationen der Datei im Anwendungscode leicht auslesen lassen, kann man sie in einem naiven Ansatz direkt für mandantenspezifischen Code benutzen. Im folgenden Codeblock wird davon ausgegangen, dass im Projekt drei Targets namens *T1*, *T2* und *T3* existieren. Diese Bezeichnungen werden auch in späteren Beispielen als fiktive Mandantennamen dienen. Darüber hinaus soll davon ausgegangen werden, dass pro Mandant eine Bilddatei für das jeweilige Firmenlogo im Projekt enthalten ist.

```
1  // Auslesen der Info-PLIST
2  NSDictionary *info = [[NSBundle mainBundle] infoDictionary];
3  // Produktnamen erfragen
4  NSString *appName = [info objectForKey:@"CFBundleDisplayName"];
5
6  // Code abhängig vom Produktnamen ausführen
7  UIImage *logo;
8  if ([appName isEqualToString:@"T1"]) {
9      logo = [UIImage imageNamed:@"logo_t1"];
10 } else if ([appName isEqualToString:@"T2"]) {
11     logo = [UIImage imageNamed:@"logo_t2"];
12 } else if ([appName isEqualToString:@"T3"]) {
13     logo = [UIImage imageNamed:@"logo_t3"];
14 }
```

Dieser Ansatz birgt sehr offensichtliche Probleme hinsichtlich seiner Skalierbarkeit. Geht man von einer großen Anzahl von Mandanten aus, entsteht ein unüberschaubares `if-else`-Statement mit viel redundantem Code, zudem sind String-basierte Konditionen fehleranfällig. Werden solche Codeblöcke an vielen Stellen verwendet, erhöht sich nicht nur die Fehleranfälligkeit, sondern auch die Größe des Produkts, da der gesamte Code in allen Targets enthalten ist. Für letztgenanntes Problem lässt sich Abhilfe schaffen, indem man die Unterscheidung mit **Präprozessor-Makros** durchführt – ungenutzte Anweisungen werden dann schon vor der Kompilierung verworfen. Voraussetzung ist, dass in den Build Settings der Targets unter „Preprocessor Macros“ eine Zuweisung der Art `TENANT_X=1` eingetragen ist. Der Anwendungscode sähe dann folgendermaßen aus:

```
1  UIImage *logo;
2  #ifdef TENANT_ONE
3      logo = [UIImage imageNamed:@"logo_t1"];
4  #endif
```

[66] Die Dateiendung steht für „Property List“

[67] In den Build Settings kann unter „Packaging“ auch eine anders benannte Datei referenziert werden

```
#ifdef TENANT_TWO
    logo = [UIImage imageNamed:@"logo_t2"];
#endif
#ifdef TENANT_THREE
    logo = [UIImage imageNamed:@"logo_t3"];
#endif
```

Neben dem geringeren binären Footprint fällt die Problematik des String-basierten Vergleichs weg. Dem steht entgegen, dass Präprozessor-Anweisungen vom Compiler nicht geprüft werden können und daher trotzdem ein großes Fehlerpotenzial bieten. Auch die Les- und Wartbarkeit des Codes ist bei einer größeren Anzahl an Mandanten ungenügend, sodass beide vorgestellten Ansätze in der Praxis nicht verwendet werden sollten.

4.1.2 Konfigurativ

Anstatt Target-Informationen explizit im Code abzufragen, können Anpassungen auch direkt durch die Einstellungen des Targets unternommen werden. Ein gutes Beispiel dafür sind die vorgegebenen Konfigurationsmöglichkeiten der Info-PLIST. Für bestimmte Grafiken, darunter das App-Icon und ein Ladebildschirm, muss nur der Dateiname hinterlegt werden. Dieser wird dann vom Framework über einen vordefinierten Key in der Info-PLIST gefunden und das Bild schließlich geladen.

Grundsätzlich kann man einer von zwei Herangehensweisen folgen: Entweder sind Ressourcen verschiedener Mandanten am **Dateinamen** zu unterscheiden (beispielsweise durch Prä- oder Suffixe), oder ausschließlich durch ihre **Target-Zugehörigkeit**. Die beiden Ansätze sind in den Abbildungen 4.1 und 4.2 skizziert.

Im erstgenannten Fall (Abbildung 4.1) lauten die Dateinamen der Mandantengrafiken beispielhaft `logo_t1.png`, `logo_t2.png` und `logo_t3.png`. Anstatt die Dateinamen im Code zu benutzen, werden sie unter einem einheitlichen Key (im untenstehenden Code-Beispiel `logo`) in der Info-PLIST abgelegt und können durch diese zusätzliche Schicht auf transparente Weise zur Laufzeit gefunden werden.

```
NSDictionary *info = [[NSBundle mainBundle] infoDictionary];
NSString *imgName = [info objectForKey:@"logo"];
UIImage *logo = [UIImage imageNamed:imgName];
```

Durch feste Konventionen für die Prä- oder Suffixe kann vermieden werden, dass für jede Ressource ein Eintrag in der Info-PLIST (oder einer anderen Konfigurationsdatei) vorhanden sein muss. Die Generierung des Namens sollte dann in eine Service-Klasse ausgelagert werden. Von entscheidender Bedeutung ist, dass alle Dateinamen disjunkt sind und es somit möglich (aber nicht notwendig) ist, die Ressourcen aller Mandanten in einem Target einzubinden. Das in Abbildung 4.1 dargestellte Target *T1* enthält die Anweisung, alle Versionen der Ressouce ins Produkt *T1.app* zu kopieren. Dies ermöglicht die Auslieferung einer App, die nach dem Build-Vorgang einen Mandantenwechsel erlaubt.

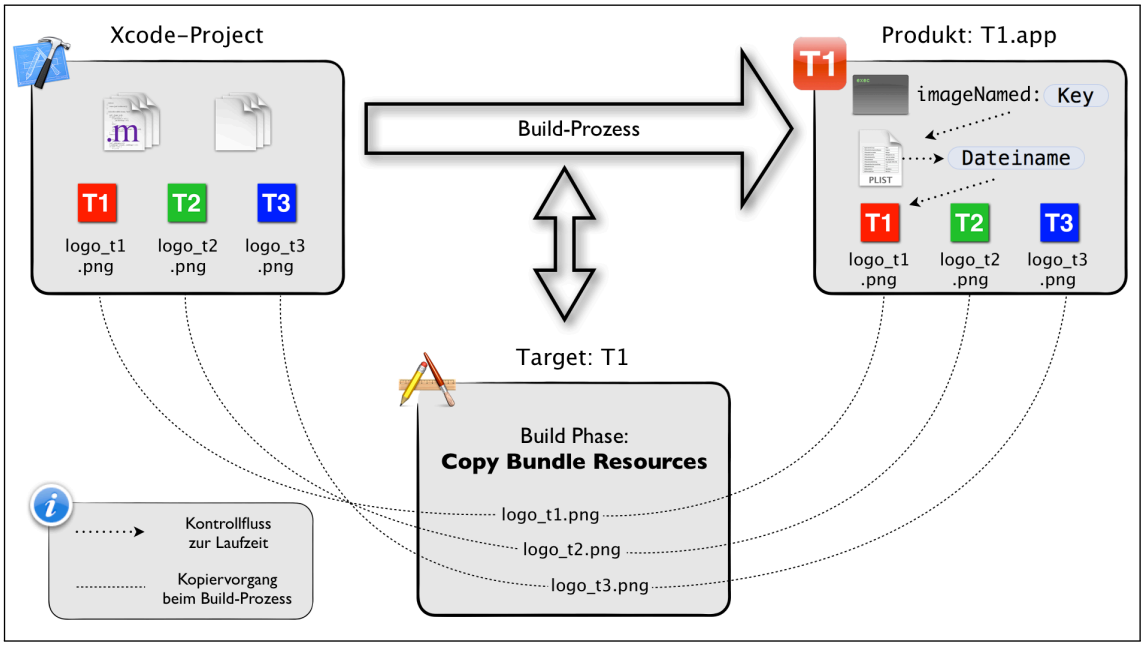


Abbildung 4.1: Unterscheidung der Ressourcen mittels Dateinamen

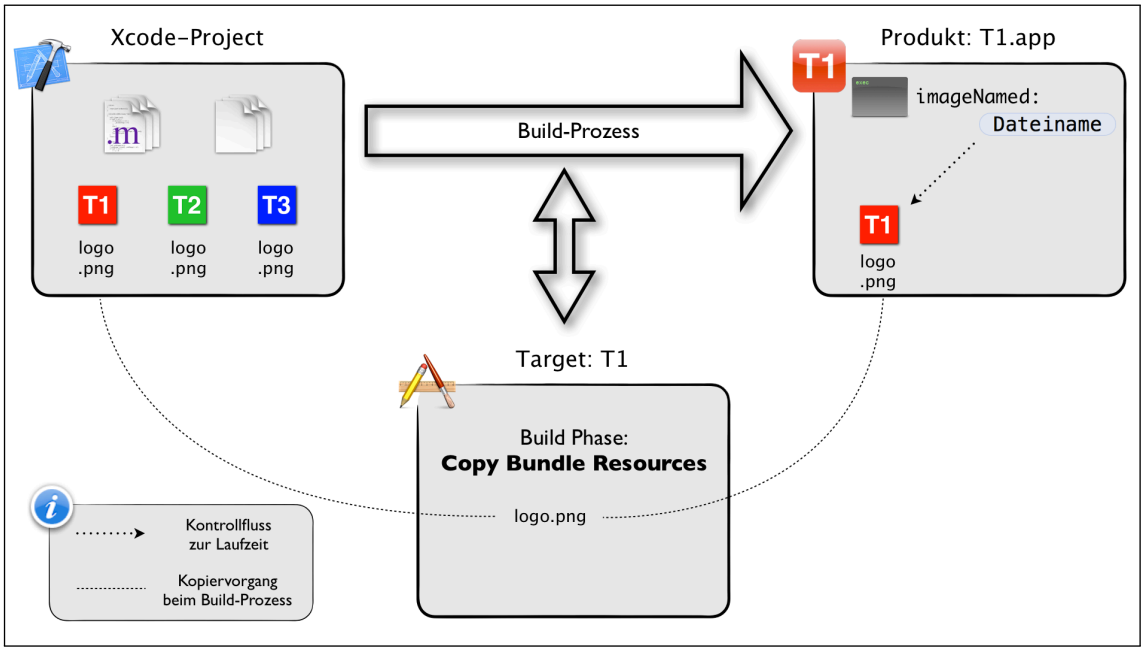


Abbildung 4.2: Unterscheidung der Ressourcen durch Target-Zugehörigkeit

Ausgeschlossen ist dies, wenn drei Dateien namens `logo.png` im Projekt vorhanden sind, wie in Abbildung 4.2 gezeigt. Aus Sicht des Dateisystems auf dem Entwicklercomputer ist der Fall unproblematisch, da die Dateien in verschiedenen Ordnern abgelegt sein können und die Lage im Dateisystem durch die Verwendung von Xcode-Gruppen abstrahiert wird. Dies hat aber, wie in Abschnitt 3.2.2 thematisiert, zur Folge, dass pro Target nur eine der drei Dateien eingebunden werden kann. Die Verwendung ist nun noch einfacher:

```
1 UIImage *img = [UIImage imageNamed:@"logo"];
```

Die Auswahl der mandantenspezifischen Version der Ressource geschieht im Rahmen des Build-Vorgangs in der letzten Build-Phase, „Copy Bundle Resources". Hier muss darauf geachtet werden, in jedem Target nur eine Version jeder Ressource hinzuzufügen, da mehrere Versionen im flachen Namensraum des App-Archivs nicht mehr unterscheidbar wären. Aus Abbildung 4.2 ist ersichtlich, dass das Target *T1* lediglich die für das mandantenspezifische Produkt *T1.app* relevante Ressource referenziert[68].

Theoretisch ist die Anpassung einer Anwendung durch Konfigurationsdateien nicht auf Ressourcen beschränkt. Durch die dynamische Bindung von Objective-C wären auf ähnliche Weise mandantenspezifische Versionen einer Methode denkbar. Dies soll anhand eines weiteren Codebeispiels kurz gezeigt werden:

```
1 // Mandantenspezifische Info-PLIST wird geladen
2 NSDictionary *info = [[NSBundle mainBundle] infoDictionary];
3 // Pro Mandant kann der eingetragene Selector-Name verschieden sein
4 NSString *selectorName = [info objectForKey:@"someMethod"];
5 // Erzeugung des Selectors
6 SEL selector = NSSelectorFromString(selectorName);
7 // Senden einer Nachricht
8 [self performSelector:selector];
```

Aufgrund der in Abschnitt 3.1.2 beschriebenen Risiken, insbesondere der fehlenden Compiler-Prüfung, muss von der Verwendung dieses Ansatzes allerdings abgeraten werden. Stattdessen sollte auf die in Abschnitt 4.3 vorgestellten modulbasierten Ansätze zurückgegriffen werden.

4.1.3 Fazit

Während die naive, programmatische Anpassung mithilfe von Target-Informationen aufgrund ihrer schlechten Skalier- und Wartbarkeit nicht empfohlen werden kann, sind **konfigurative Ansätze** für die **Ressourcenauswahl** eine einfache und im Code transparente Möglichkeit, mandantenspezifische Anpassungen vorzunehmen. Werden in einem Produkt alle Versionen einer Ressource mitgeliefert, ist ein Mandantenwechsel nach dem Vertrieb der App theoretisch denkbar. Allerdings muss diese Möglichkeit teuer bezahlt werden, da die Größe der Anwendung mit steigender Mandantenanzahl linear wächst[69]. Problematisch ist auch, dass alle in einer fertig

[68] Dazu wird der vollständige Pfad im Dateisystem genutzt

[69] Diese Einschätzung beruht auf der Erkenntnis, dass der Footprint einer App im Wesentlichen durch die Größe der eingebundenen Ressourcen bestimmt wird

gebauten App eingebundenen Ressourcen im Dateisystem ohne Weiteres auffindbar sind[70]. In der Regel ist es aber unerwünscht, dass ein Mandant die Ressourcen eines anderen Mandanten einsehen kann, sei es in der App selbst oder im Dateisystem. Daher ist zu empfehlen, dass die App entweder nur mandantenspezifische Ressourcen enthält, oder alle Ressourcen zur Laufzeit von einem Server geladen werden, bei dem sich der Mandant zuvor authentisieren muss.

Die vorgestellten Ansätze allein bieten keine praktikable Möglichkeit, komplexe mandantenspezifische Anpassungen am Quellcode einer App vorzunehmen. Sie sind daher nur als Ergänzung zu den weiteren Ansätzen dieses Kapitels anzusehen.

4.2 STRINGS UND LAYOUT

Die GUI einer mehrmandantenfähigen App muss über grafische Ressourcen hinaus anpassbar sein, insbesondere **lokalisierte Strings** spielen dabei eine entscheidende Rolle. Genau wie Quellcodedateien haben String-Ressourcen eine zweidimensionale Ausprägung – während beim Quellcode zwischen der SCM-Revision und dem Mandanten (Branch) unterschieden wird, sind Sprache und Mandantenversion die beiden wesentlichen Dimensionen von String-Ressourcen[71]. Dabei haben praktische Untersuchungen an der alexa-MRS-App ergeben, dass nur wenige Strings tatsächlich mandantenspezifisch sind. Es bietet sich also an, zur leichten und konsistenten Pflege eine Basis-String-Ressourcen-Datei zu verwalten. Bei Bedarf können ausgewählte Schlüssel in mandantenspezifischen Ressourcen-Dateien überschrieben werden. Dieses Szenario kann programmatisch leicht realisiert werden. Allerdings hat dies einen erhöhten Rechenaufwand zur Laufzeit zur Folge, da String-Schlüssel in mehreren Dateien gesucht werden müssen. Zudem hat das von Xcode nativ verwendete `.strings`-Format[72] für String-Ressourcen-Dateien zwei entscheidende Nachteile:

- SCM-Werkzeuge wie Git können die Dateien nicht mergen, sondern behandeln sie wie binäre Dateien[73]
- Oftmals möchten die Endkunden ihre Strings selbst per Excel-Spreadsheet verwalten

Um beide Nachteile zu beseitigen, können die Strings im CSV-Format[74] abgespeichert werden. Das textbasierte Format wird von der standardmäßigen Git-Installation beim Merge-Vorgang unterstützt[75] und lässt sich mit Spreadsheet-Programmen wie Microsoft Excel editieren. Im Rahmen der Werkstudententätigkeit bei der SALT Solutions GmbH wurde ein Werkzeug namens *stringsbuilder* entwickelt, welches die CSV-Dateien bei jedem Build-Vorgang automatisch zu einer

[70] Die im App Store angebotenen `.ipa`-Dateien sind einfache ZIP-Archive

[71] Auch String-Ressourcen können eine SCM-Revision besitzen, diese spielt aber im Vergleich zu Quellcodedateien eine untergeordnete Rolle

[72] `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/LoadingResources/Strings/Strings.html`

[73] Der Grund dafür ist nicht das Dateiformat selbst, sondern die Kodierung im UTF-16-Format

[74] `http://tools.ietf.org/pdf/rfc4180.pdf` – Comma-Separated Values

[75] Vorausgesetzt, die CSV-Dateien werden nicht UTF-16-kodiert – Microsoft Excel exportiert CSV im Western (Mac OS Roman)-Format

`.strings`-Ressource zusammenfasst. Das Zusammenfügen von allgemeingültigen und mandantenspezifischen Strings geschieht also bereits zur Compile-Zeit und kostet keine Performance zur Laufzeit. Welche CSV-Dateien beachtet werden sollen, wird in einer Target-spezifischen PLIST-Datei definiert. Darüber hinaus beherrscht stringsbuilder das Zusammenfügen von über Module verteilten CSV-Dateien[76] sowie die Spezifikation der Ausgangs- und Zielkodierung. Das Werkzeug ist inklusive einer Beispielanwendung auf Bitbucket[77] veröffentlicht worden.

Lokalisierte und mandantenspezifische Strings stellen die GUI vor eine weitere Herausforderung, da sie keine statischen Längen besitzen. Um dieses Problem zu lösen, ist seit iOS 6 **Auto Layout** verfügbar. Auto Layout selbst ermöglicht keine Mehrmandantenfähigkeit, aber begünstigt die Erstellung einer generischen GUI, die sich an mandantenspezifische Ausprägungen von Daten anpassen kann. Die Grundlage von Auto Layout sind umfangreiche Bedingungen (engl.: **Constraints**), die zwischen allen GUI-Elementen gesetzt werden können. Mithilfe von Constraints lassen sich komplexe, relative Beziehungen zwischen GUI-Elementen definieren. Beispielsweise kann formuliert werden, dass zwei Buttons immer die gleiche Breite haben sollen. Dabei soll der größere von beiden die Breite beider Buttons bestimmen, sofern die Buttons zusammen nicht größer als ein bestimmter Rahmen sind. Somit kann sich die GUI verschiedenen String-Längen anpassen, ohne gegen wohldefinierte, höher priorisierte Bedingungen zu verstoßen. Constraints können entweder in Interface Builder, im Storyboard-Editor oder im Quellcode gesetzt werden.

Um **XIB- oder Storyboard-Dateien** zu lokalisieren, gibt es 3 Möglichkeiten:

- Strings werden im Code gesetzt
- Strings sind direkt in die GUI-Dateien eingebettet und es existieren Kopien dieser Dateien für jede unterstützte Sprache (standardmäßige Lokalisierung in Xcode)
- Jede GUI-Datei bezieht ihre Strings aus einer separaten, lokalisierten `.strings`-Datei (seit Xcode 4.5 unter dem Namen „Base Internationalization“ möglich)

Der zweite Ansatz ist dabei der ungünstigste, weil die GUI-Dateien untereinander nicht synchronisiert werden – bei `i` verschiedenen Sprachen müssen GUI-Änderungen manuell an `i` verschiedenen Dateien durchgeführt werden. Verschärft wird diese Problematik bei `j` mandantenspezifischen Stringmengen, die zu `(i*j)` Kopien einer GUI-Datei führen. Weniger problematisch ist der letztgenannte Ansatz (Base Internationalization), bei dem es zwar `i` verschiedene `.strings`-Dateien, aber nur eine GUI-Datei gibt. Leider unterstützt dieser Ansatz lediglich die Sprach-, aber nicht die Mandantendimension. Die Kompatibilität mit stringsbuilder wird gebrochen, weil die GUI-Dateien nicht auf eine globale, sondern eine jeweils eigene `.strings`-Datei zugreifen. Daher ist es für mehrmandantenfähige Apps am einfachsten, alle Strings im Code zu setzen und in den GUI-Dateien nur Platzhalter zu benutzen. Auf diese Art lassen sich alle Strings eines Projekts konsistent mit CSV-Dateien und stringsbuilder verwalten und benutzen.

Eine große Hilfe für die Durchsetzung eines mandantenspezifischen Corporate Designs sind die seit iOS 5 verfügbaren **Appearance-Proxies**. Sie erlauben es, Standard-GUI-Elemente zentral

[76] Module werden in diesem Zusammenhang nur als getrennte Ordner im Dateisystem gesehen
[77] `https://bitbucket.org/shagedorn/stringsbuilder/`

zu konfigurieren, ohne dazu Unterklassen bilden zu müssen. Der untenstehende Codeabschnitt ändert die Färbung aller im Projekt instanziierten `UINavigationBar`-Objekte:

```
1 [[UINavigationBar appearance] setTintColor:[UIColor blueColor]];
```

Die im `UIAppearance`-Protokoll[78] definierte `appearance`-Klassenmethode gibt ein Proxy-Objekt zurück. Alle Änderungen am Proxy-Objekt werden von allen zukünftigen Instanzen der Klasse übernommen. Damit lässt sich ein Farbschema leicht anwendungsweit durchsetzen, ohne dass alle Anwendungsbereiche mandantenspezifischen Code enthalten. Viele `UIView`-Unterklassen des GUI-Frameworks UIKit implementieren das `UIAppearance`-Protokoll, das auch für eigene GUI-Klassen verwendet werden kann.

Zusammen mit Abschnitt 4.1 wurden bislang Mehrmandantenkonzepte vorgestellt, die vornehmlich die **Programmoberfläche** und die Ressourcen betreffen. Wie eingangs in Abschnitt 2.1.1 erwähnt, sollen neben der Oberfläche aber auch der Code selbst sowie das Datenmodell konfigurierbar sein. Diesen beiden Problemen widmen sich die Abschnitte 4.3 respektive 4.4.

4.3 MODULBASIERTE ANSÄTZE

Analog zu den Abschnitten 2.2.1 und 2.2.2 im Grundlagen-Kapitel sollen im Folgenden einzelne Schritte hin zu einer modularen Architektur beschrieben werden. Im Unterschied zu Kapitel 2 sind alle folgenden Werkzeuge und Technologien entweder iOS-spezifisch oder zumindest im Kontext der iOS-App-Entwicklung anwendbar. Die Refactoring-Abschnitte 4.3.1 und 4.3.2 unterliegen der Annahme, dass eine bestehende Anwendung auf ihre modulare Architektur hin untersucht wird bzw. eine solche eingeführt werden soll. Die weiteren Unterabschnitte beschreiben Möglichkeiten, eine vorhandene modulare Architektur technisch zu forcieren. Dazu werden Xcode Subprojekte, das SCM-Werkzeug Git sowie Werkzeuge zur Dependency-Verwaltung untersucht.

4.3.1 Analyse und Refactoring durch Werkzeuge

Grundlegend für das Finden von Modulen ist die Analyse bestehender Abhängigkeiten zwischen Klassen und Gruppen von Klassen. Die von Apple bereitgestellten Entwicklungswerkzeuge bieten diese Funktionalität in keiner Form an, weshalb im Rahmen dieser Arbeit Werkzeuge von Dritten auf ihre Tauglichkeit zur Modulfindung untersucht worden sind.

Ein einfaches Werkzeug ist das auf Github[79] veröffentlichte, quelloffene Python-Skript **objc_dep**[80]. Anhand der `#import`-Anweisungen innerhalb eines Xcode-Projekts erstellt das Skript einen physischen Abhängigkeitsgraphen im `.dot`-Dateiformat, dem Komponenten auf Dateiebene (Header- und Implementierungsdateien zusammengefasst) zugrunde liegen. Diese Art des Reverse Engineerings legt ungewollte Abhängigkeiten und andere Designfehler bei kleineren Projekten offen.

[78] `http://developer.apple.com/library/ios/#documentation/uikit/reference/UIAppearance_Protocol/`
[79] `https://github.com/` – Hosting von Git-Repositories
[80] `https://github.com/nst/objc_dep/`

Das Werkzeug stößt allerdings bei großen Projekten schnell an seine Grenzen, weil es einen unüberschaubaren, zusammenhängenden Graph ausgibt. In Anlage A.1 befindet sich ein mit objc_dep generierter Graph für ein kleines iOS-Projekt mit nur 40 Klassen. Zum Vergleich wurde der Graph für alexa MRS mit knapp 1500 Klassen generiert, das Ergebnis des über mehrere Stunden andauernden Renderings[81] ist in Anhang A.2 einzusehen und illustriert die Untauglichkeit des Werkzeugs bei großen Projekten. Hilfreich ist, dass zyklische Abhängigkeiten blau eingefärbt werden und dadurch auch in unübersichtlichen Graphen schnell erkennbar werden.

Einen größeren Funktionsumfang bietet das ebenfalls kostenlose Werkzeug **Doxygen**[82], das hauptsächlich zur Generierung von Code-Dokumentation gedacht ist. Darüber hinaus unterstützt es die Erstellung verschiedener Graphen und Diagramme im `.dot`-Dateiformat. Im Gegensatz zu objc_dep erstellt Doxygen keine projektweiten Graphen, sondern stellt die transitiven Beziehungen einer Ausgangsklasse dar. Für Basisklassen entstehen dadurch relativ große Graphen, während die Graphen von Klassen am unteren Ende der Vererbungshierarchie oft nur die Klasse selbst abbilden. Durch den Fokus auf eine Ausgangsklasse lassen sich zwei Typen von Abhängigkeitsgraphen generieren. „Include dependency"-Graphen bilden die Klassen ab, von denen die fokussierte Klasse abhängig ist. „Included by dependency"-Graphen zeigen diejenigen Klassen an, die selbst von der fokussierten Klasse abhängen. Durch die feine Granularität der Diagramme erleichtert Doxygen die genaue Analyse kleiner Programmteile, aber eine Bewertung der Gesamtarchitektur ist mit den generierte Diagrammen nicht möglich. Insbesondere für große Projekte entstehen zudem unüberschaubar viele Diagramme. Verstärkt wird dieses Problem durch die Tatsache, dass Doxygen für die Diagramme sowohl die Header- als auch die Implementierungsdateien verwendet und sich dadurch viele Diagramme doppeln. Einige Objective-C-Sprachkonstrukte (z. B. Categories) werden von Doxygen nicht unterstützt und führen zu fehlerhaften physischen Graphen.

Ein verwandtes Problem sind ungenutzte `#import`-Anweisungen, die nicht nur Graphen unnötig aufblähen, sondern auch reale physische Abhängigkeiten zur Folge haben. Leider ist die aus Eclipse und anderen Java-IDEs bekannte Funktion „Organize Imports" in Xcode nicht vorhanden. Die alternative, kostenpflichtige IDE **AppCode**[83] verspricht verbesserte Refactoring-Möglichkeiten, darunter eine Funktion namens „Optimize Imports". In einem Test[84] konnte jedoch nicht festgestellt werden, dass ungenutzte Imports entfernt werden.

Eine grobe Übersicht über die Klassenstruktur eines Xcode-Projekts lässt sich mit **OmniGraffle**[85] erlangen. Im Gegensatz zu objc_dep und Doxygen werden dafür nicht die Abhängigkeiten, sondern die Vererbungshierarchien analysiert. Dies macht einzelne Gruppen von Klassen zwar deutlich übersichtlicher, allerdings fehlen auch logische und physische Abhängigkeiten, sodass das Werkzeug für die Modulfindung nicht geeignet ist. Positiv anzumerken ist, dass OmniGraffle Xcode-Gruppen erkennen und anzeigen kann, wodurch auch sehr große Projekte relativ überschaubar bleiben.

[81] Das Rendering der `.dot`-Dateien wurde mit GraphViz durchgeführt: `http://www.graphviz.org/`
[82] `http://www.stack.nl/~dimitri/doxygen/`
[83] `http://www.jetbrains.com/objc/`
[84] Dabei wurde eine Trial-Version von AppCode 1.6 genutzt
[85] `http://www.omnigroup.com/products/omnigraffle/`

Abgesehen von AppCode erzeugen alle bisher genannten Werkzeuge rein grafische Ergebnisse. Eine inhaltliche Filterung der Ergebnisse ist ebensowenig möglich wie die Übertragung von Änderungen an den Diagrammen auf den Ausgangscode. Die bereits in Kapitel 2 erwähnte kommerzielle Software Visual Paradigm for UML (**VP-UML**) ermöglicht einfaches Forward Engineering, also die Code-Erzeugung aus Diagrammen für Objective-C. Da allerdings die entgegengesetzte Richtung sowie vollumfassendes Round-Trip Engineering nur für andere Programmiersprachen unterstützt werden, ist auch diese Software nur bedingt geeignet, modulare Architekturen zu erkennen oder einzuführen.

Die bestehenden Werkzeuge sind nur sehr eingeschränkt nutzbar. Während die kommerziellen Werkzeuge wichtige Funktionen vermissen lassen, scheitern die kostenlosen Lösungen v. a. an der Unübersichtlichkeit der generierten Graphen bei großen Projekten. Es bleibt festzuhalten, dass man sich bei iOS-Projekten nicht auf Werkzeuge verlassen kann, sondern eigene Dokumentationen und Diagramme sorgfältig pflegen sollte. Um auch große Projekte leicht überschauen zu können, ist es hilfreich, gängige Entwurfsmuster und Konventionen zu beachten. Die wichtigsten werden im folgenden Abschnitt vorgestellt.

4.3.2 Architektur und Design Patterns

Zwei wesentliche Entwurfsmuster, die sich im gesamten Cocoa-Touch-Framework wiederfinden, sind das Delegation- und das Model-View-Controller-Pattern (MVC). Beide Muster unterstützen die lose Kopplung von Klassen. Während Delegation nur die konkrete Interaktion zwischen zwei Klassen beschreibt, kann das MVC-Pattern auf die Gesamtarchitektur einer Anwendung bezogen werden.

Nach dem **MVC**-Prinzip kann jede Klasse einer Anwendung in eine von drei Kategorien eingeordnet werden – Model, View, oder Controller.

Die stärkste Trennung findet zwischen Model- und View-Klassen statt, die komplett unabhängig voneinander implementiert werden und sich nie gegenseitig importieren dürfen. Dies hat zur Folge, dass beide Klassentypen einen hohen Wiederverwendungsgrad haben. Der Controller wird oft als Klebstoff (engl.: Glue) zwischen Model- und View-Klassen bezeichnet: Er verwaltet das Modell, bereitet Daten zur Anzeige auf, fügt die Daten in die richtigen Views ein und reagiert auf Nutzeraktionen, die von den Views über fest definierte Schnittstellen an den Controller weitergeleitet werden. Controller-Klassen müssen in der Regel spezifisch für eine Anwendung geschrieben werden. In Abbildung 4.3 sind die drei MVC-Klassentypen und ihre typischen Interaktionsmuster grafisch dargestellt. Bei mobilen Anwendungen hat sich der Begriff des „View-Controllers“ für die Einheit aus einem Controller und der von ihm verwalteten View-Hierarchie etabliert. In der Regel füllt ein ViewController den gesamten Bildschirm aus, wofür in diesem Kontext auch die englische Bezeichnung als „Screen“ üblich ist. ViewController sind oft die primären Baueinheiten, aus denen eine iOS-Anwendung zusammengesetzt wird.

Das **Delegation**-Pattern hilft, die Kopplung zwischen Views und deren Controllern möglichst lose zu gestalten und eine hohe Wiederverwendbarkeit der Views zu erreichen. Im UIKit-Framework ist es nicht üblich, Unterklassen von Views zu bilden, um anwendungsspezifische Daten darzustel-

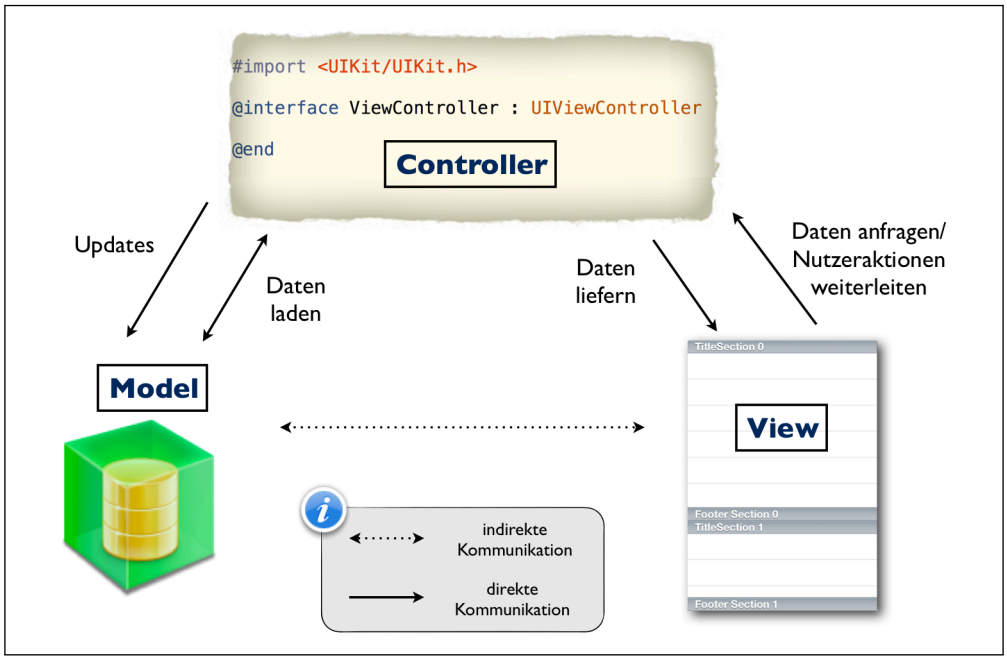


Abbildung 4.3: Model-View-Controller-Pattern

len. Stattdessen erfragt ein View-Objekt die Daten und Details zu deren Darstellung zur Laufzeit bei seinem Controller-Objekt, dem sogenannten Delegate[86]. Der View delegiert somit die Beschaffung der konkreten Daten an ein anderes Objekt. Um zu verhindern, dass das View-Objekt zur Kommunikation die genaue Schnittstelle der Controller-Klasse kennen und daher importieren muss, definiert das View-Objekt selbst ein Protokoll für sein Delegate[87]. Dies entspricht dem in [Kno12, Kapitel II.11 Separate Abstractions] festgehaltenen „Separate Abstractions"-Grundprinzip, welches besagt, dass Schnittstellen nahe den Klassen definiert werden sollen, die sie benutzen, und weit entfernt von den Klassen, die die Schnittstelle implementieren. In dem vom View definierten Protokoll werden alle Nachrichten spezifiziert, die das Controller-Objekt bzw. Delegate beantworten können muss. Demnach muss das View-Objekt die Controller-Klasse nicht kennen, sondern es genügt die Tatsache, dass der Controller das Delegate-Protokoll implementiert. Es besteht eine einseitige Abhängigkeit des Controllers vom View, aber der View wird in seiner Wiederverwendbarkeit nicht eingeschränkt. Das Delegation-Prinzip wird besonders häufig, aber nicht ausschließlich zwischen View- und Controller-Klassen verwendet. Lose Kopplung lässt sich u. a. auch mit dem Notification/Observer-Pattern und dem Target/Action-Pattern realisieren, die beide in [BY10] näher beschrieben werden.

Die konsequente Anwendung des MVC-Patterns erleichtert sowohl die vertikale als auch die horizontale Unterteilung einer Anwendung. Bei der **vertikalen Teilung** werden Klassen zusammengefasst, die zu einem bestimmten Objekt der Business-Domäne gehören [Kno12, Kapitel II.8 Module Reuse]. Typische Beispiele für Business-Domänen-Objekte einer ERP-Anwendung sind Artikel, Kunden und Aufträge. Alle zu einem solchen Domänen-Objekt gehörenden ViewController bilden die Grundlage für ein Modul. Die weiteren Modulklassen werden faktisch durch die Imports der Controller bestimmt und können aus diesen abgelesen werden, wobei die Komplexität dieser

[86] Das Delegate-Objekt muss nicht zwangsweise ein Controller sein, aber dies ist die gängigste Praxis
[87] Dies geschieht in der gleichen Datei, in der auch das Interface der View-Klasse definiert wird

Aufgabe v. a. durch die Transitivität der Imports entsteht. Zur Unterstützung kann hierbei das in Abschnitt 4.3.1 vorgestellte Werkzeug Doxygen dienen, das die transitiven Abhängigkeiten einer Klasse gut darstellen kann. Oft werden Klassen auch von ViewControllern verschiedener Module benutzt – in diesem Fall muss zwischen zwei Lösungsmöglichkeiten abgewogen werden:

- Die importierte Klasse wird in die Infrastruktur ausgelagert, auf die alle betroffenen Module zugreifen können
- Die importierte Klasse wird einem Modul zugeordnet, und alle anderen betroffenen Module haben eine Abhängigkeit zu diesem Modul

Die Entscheidung muss im Einzelfall erfolgen und ist abhängig davon, ob die betroffene Klasse ungeachtet ihrer Referenzierung in den Controller-Klassen einem Modul eindeutig zugeordnet werden kann.

Innerhalb dieser vertikalen Module bietet sich eine **horizontale Schichtenarchitektur** an, die durch das MVC-Pattern geprägt ist. Demnach gibt es initial drei Schichten: Model, View und Controller. Viele iOS-Apps greifen ausschließlich auf View-Klassen aus dem Framework zurück, sodass in diesen Fällen die View-Schicht leer bzw. nicht vorhanden ist. Bei komplexen Anwendungen kann die Model-Schicht in mehrere kleine Schichten unterteilt werden. Die Controller sollten stets die oberste Schicht bilden, da sie Klassen aus den anderen Schichten verwalten, wie aus Abbildung 4.3 ersichtlich ist. Während das MVC-Pattern die Anordnung der vertikalen und horizontalen Schichten bestimmt, kann das Delegation-Pattern genutzt werden, um Verbindungen zwischen Schichten möglichst lose zu gestalten und bidirektionale Abhängigkeiten zu verhindern.

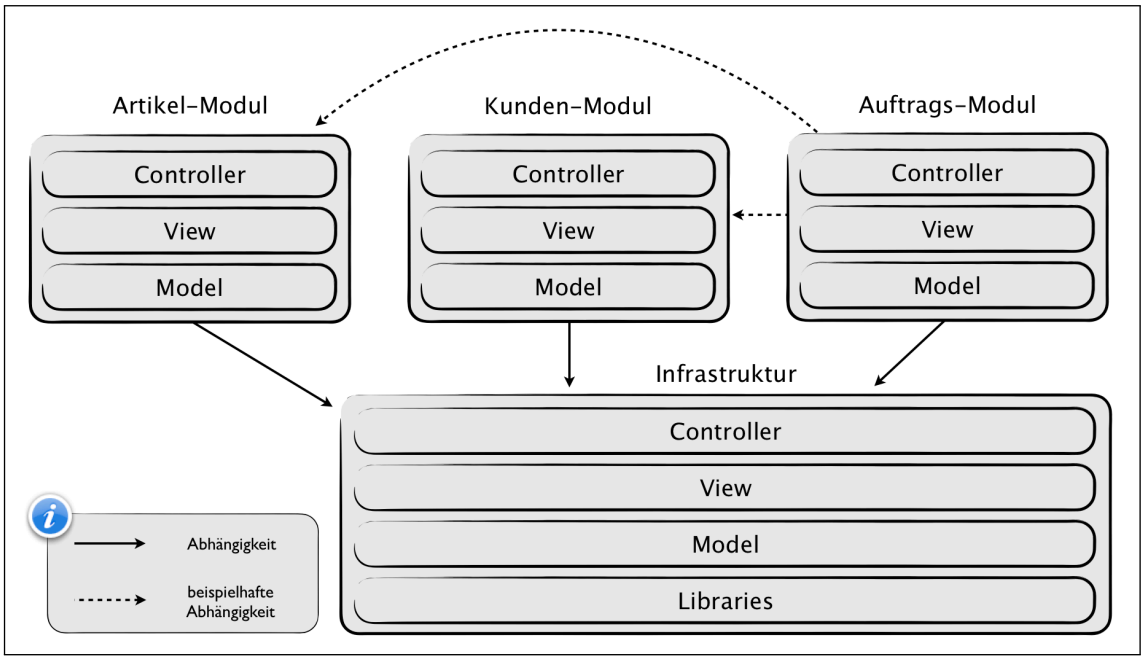


Abbildung 4.4: Modulare Beispielarchitektur

Neben den domäne-spezifischen Modulen bestehen die meisten Anwendungen aus einem Kern, der Module verwalten und deren Basis-ViewController aufrufen kann. Zu diesem Kern, im Folgen-

den auch Infrastruktur-Modul genannt, gehören auch Klassen und Bibliotheken, die modulübergreifend genutzt werden. Entscheidend ist, dass das Infrastruktur-Modul keine Abhängigkeiten zu den Domäne-Modulen besitzt. Um diese Module einheitlich verwalten und ansprechen zu können, sollte das Infrastruktur-Modul entsprechende Basisklassen und Protokolle bereitstellen. Ein **grobes Architekturmodell**, das die Empfehlungen dieses Abschnitts aufgreift, ist in Abbildung 4.4 dargestellt. Darin eingezeichnet sind beispielhafte Abhängigkeiten zwischen den vertikalen Modulen. Das Artikel- und das Kunden-Modul können unabhängig von anderen Modulen benutzt werden, sofern das Infrastruktur-Modul vorhanden ist. Das Auftrags-Modul setzt dagegen voraus, dass das Artikel- und das Kunden-Modul verfügbar sind.

Am einfachsten lässt sich eine modulare Architektur mit Dateisystemordnern und entsprechenden Xcode-Gruppen abbilden. Dies ist allerdings nur eine logische Einordnung und keine tatsächliche Modularisierung mit klar definierten und geprüften Abhängigkeiten. Der Verzicht auf Xcode-Gruppen und die Verwendung von synchronisierten Ordnern in Xcode käme diesem Ziel näher, da bei jeder `#import`-Anweisung der komplette Pfad im Dateisystem und damit indirekt auch das Modul angegeben werden müsste. Modulweite Abhängigkeiten lassen sich allerdings nicht klar definieren und die Methode verlangt weiterhin Programmierdisziplin, wenngleich diese deutlich unterstützt wird, weil die referenzierten Module aus den Imports direkt ablesbar sind. Die Entkopplung von Dateisystem und Xcode-Gruppen und die resultierenden einfachen, weil flachen Imports sind jedoch ein beliebtes Feature bei Entwicklern, das nach Möglichkeit erhalten bleiben sollte. Im folgenden Abschnitt werden Xcode-Subprojekte thematisiert, die modulare Projekte mit Xcode-Gruppen erlauben.

4.3.3 Workspaces und Subprojekte

Es gibt in Objective-C keine Möglichkeit, Quellcodebereiche eines Projekts in Paketen oder ähnlichen Strukturen voneinander zu trennen, sodass Sichtbarkeiten eingeschränkt und Abhängigkeiten explizit definiert werden können. Lediglich binäre Bibliotheken können beim Link-Vorgang eingebunden werden. Auch bei der Verwendung von Xcode Subprojekten und Workspaces wird letztlich auf **statische, kompilierte Bibliotheken** zugegriffen. Anstelle einer Binärdatei zeigt Xcode bei Subprojekten jedoch das vollständige Teilprojekt samt Quellcode an, obwohl beim Link-Vorgang nur das kompilierte Produkt verwendet wird. Falls Code-Änderungen stattgefunden haben, werden referenzierte Sub- oder Workspace-Projekte neu kompiliert und deren aktualisierte Produkte beim Linken verwendet.

Prinzipiell kann jedes Xcode-Projekt als **Subprojekt** verwendet und in beliebig vielen **Workspaces** eingebunden werden. Bei beiden Varianten wird das eingebundene Projekt lediglich referenziert, bleibt aber ein eigenständiges Projekt ohne Wissen über seine Referenzierung. Projekte, die Subprojekte referenzieren, werden im Folgenden **Superprojekte** genannt. Wie es der Name impliziert, besteht eine Hierarchieordnung zwischen Super- und Subprojekten: Superprojekte wissen von ihren Subprojekten[88] und können auf diese zugreifen, während Subprojekte mangels Wissen von ihren Superprojekten keinerlei Zugriff auf deren Code und Ressourcen besitzen. Projekte in einem Workspace verhalten sich wie mehrere Subprojekte einer Hierarchieebene. Sie besitzen

[88] Die Pfade zu den Subprojekten sind in der `.xcodeproject`-Datei enthalten

kein direktes Wissen voneinander, können aber die Produkte anderer Workspace-Projekte nutzen. Beim Build-Vorgang eines Projekts erkennt Xcode veraltete, referenzierte Produkte eines Workspaces und baut diese neu. Die Reihenfolge der zu bauenden Produkte legt Xcode automatisch fest[89] und setzt voraus, dass keine zyklischen Abhängigkeiten bestehen.

Für Subprojekte und deren primäre Targets wird das Xcode-Template **Cocoa Touch Static Library** verwendet. Static Libraries (dt.: Statische Bibliotheken) enthalten ausschließlich kompilierten Code in Form einer `.a`-Datei. Dies ist problematisch bei Modulen, die neben Code auch Ressourcen wie Bilder oder XIB-Dateien enthalten. Für solche Module muss im Subprojekt ein zweites Target vom Typ **Bundle** angelegt werden. Damit sowohl Ressourcen als auch Code im Superprojekt nutzbar sind, muss das Static-Library-Produkt in der „Link Binary with Libraries"-Build-Phase und das Bundle in der „Copy Bundle Resources"-Phase eingebunden werden. Um sicherzustellen, dass das Bundle-Target auch gebaut und gegebenenfalls aktualisiert wird, sollte es unter „Target Dependencies" des Static-Library-Targets selbst oder des Superprojekt-Targets aufgeführt sein. Für jedes Subprojekt und damit jedes Modul werden die Ressourcen in getrennten Bundles gehalten. Viele APIs, die den Zugriff auf Ressourcen erleichtern, durchsuchen standardmäßig nur das sogenannte „Main Bundle", das lediglich die Ressourcen des Superprojekts enthält. Der folgende Code-Abschnitt zeigt, wie auf die Bundle-Ressourcen eines Moduls zugegriffen werden kann:

```
1 // URL des Modulbundles herausfinden
2 NSURL *bundleURL = [[NSBundle mainBundle] URLForResource:@"ArticleBundle"
3                                              withExtension:@"bundle"];
4 // Bundle laden
5 NSBundle *bundle = [NSBundle bundleWithURL:bundleURL];
6 // Ressource in Modulbundle finden
7 NSString *imgPath = [bundle pathForResource:@"logo_t1" ofType:@"png"];
8 // Ressource laden
9 UIImage *img = [[UIImage alloc] initWithContentsOfFile:imgPath];
```

Als Alternative zu statischen Bibliotheken war in früheren Xcode-Versionen das **Framework**-Template verfügbar. Im Gegensatz zu statischen Bibliotheken können Frameworks Ressourcen enthalten und wären dadurch besser geeignet für Module, die nicht ausschließlich aus Code bestehen. Das **iOS-Universal-Framework**-Projekt[90] stellt ähnliche Templates für die aktuelle Xcode-Version zur Verfügung, die jedoch entweder Compiler-Warnungen mit sich bringen oder Änderungen an der Xcode-Installation erfordern. Darüber hinaus sind die sogenannten „Embedded Frameworks" (Frameworks mit Ressourcen) nicht vollständig kompatibel mit Subprojekten, weshalb sie im weiteren Verlauf dieser Arbeit nicht in Betracht gezogen werden.

Für zweistufige Hierarchien, wie sie in dem unter Abschnitt 4.3.2 beschriebenen Architekturmodell zwischen Infrastruktur und den Modulen vorherrschen, erfüllen Subprojekte die wesentlichen Anforderungen: Die Abhängigkeiten zwischen Modulen sind aus den Einträgen der „Link Binary with Libraries"-Phase klar erkennbar und grenzen die Sichtbarkeit von Schnittstellen effektiv ein. Jede Static Library stellt nur diejenigen Header-Dateien zur Verfügung, die in der „Copy Headers"-Phase explizit exportiert werden. Über das „Public Headers Folder Path"-Attribut in den Build

[89] Im Build-Schema eines Targets kann die Reihenfolge auch explizit definiert werden
[90] `https://github.com/kstenerud/iOS-Universal-Framework/`

Settings kann zudem festgelegt werden, ob flache Imports auch modulübergreifend möglich sind oder die Modulzugehörigkeit aus den Imports ablesbar ist. Dies ist im folgenden Code-Abschnitt beispielhaft dargestellt:

```
/* Annahme: Article.h liegt im Modul 'Article'
   und wird in der 'Copy Headers'-Phase exportiert
*/
// Public Headers Folder Path ist leer: Flacher Import
#import <Article.h>
// Public Headers Folder Path: 'Article': Hierarchischer Import
#import <Article/Article.h>
```

Subprojekte können ihrerseits Subprojekte (Sub-Subprojekte) besitzen und gegen deren Produkte linken. Beim Build-Vorgang wird zuerst das Sub-Subprojekt, später das Subprojekt und zuletzt das Superprojekt gebaut. Für das Superprojekt ist die **mehrstufige Hierarchie** allerdings transparent, da zum Zeitpunkt seines Build-Vorgangs nur noch die statische Bibliothek des Subprojekt-Targets vorliegt. Imports der Art `#import <Module/Submodule/Class.h>` sind also nicht ohne Weiteres möglich. Muss das Superprojekt auf Klassen des Sub-Subprojekts zugreifen, ohne explizit gegen dessen Produkt zu linken, muss das Subprojekt die Header des Sub-Subprojekts öffentlich bereitstellen.

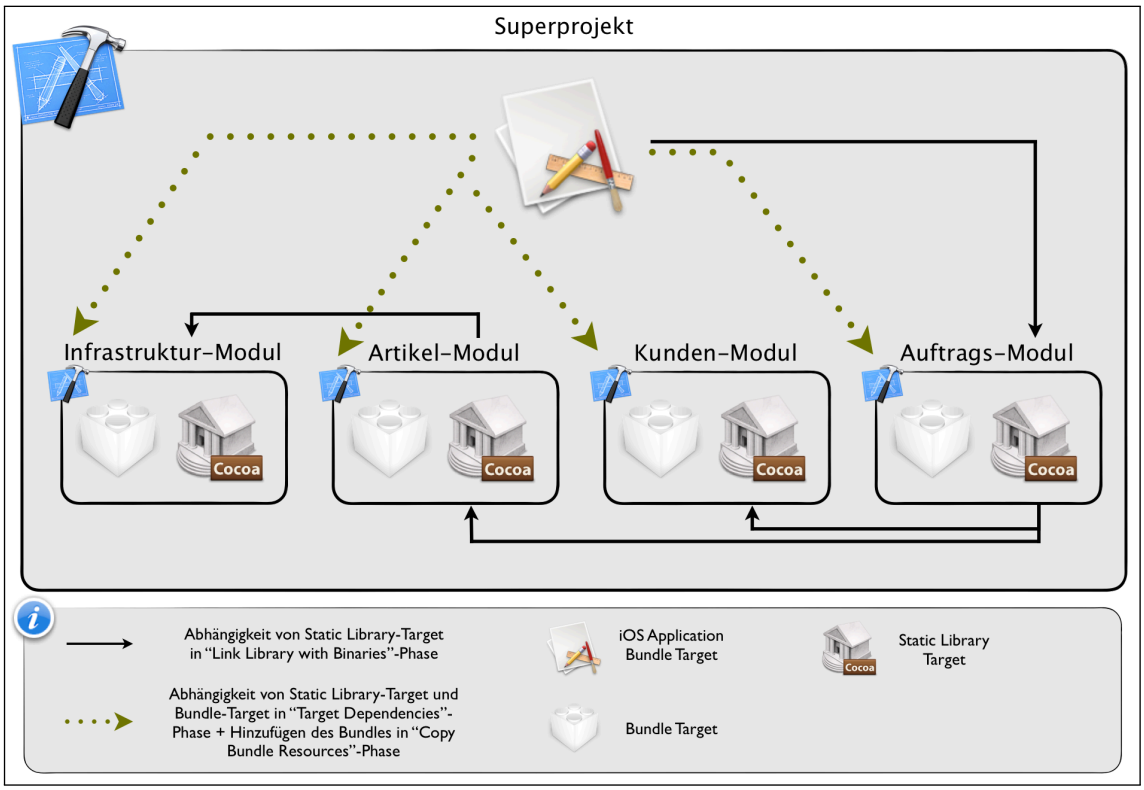


Abbildung 4.5: Abhängigkeiten zwischen Super- und Subprojekten

In Abbildung 4.5 ist dargestellt, wie sich die unter Abschnitt 4.3.2 vorgestellte **Beispielarchitektur mit Xcode Subprojekten** für iOS-Apps umsetzen lässt. Alle Module sind als Subprojekte umgesetzt, die jeweils Targets für eine statische Bibliothek und ein Ressourcen-Bundle besitzen.

Das Superprojekt dient als Hülle der Module und enthält ein iOS-Application-Target, das letztlich die lauffähige Anwendung baut. Alle Module setzen auf dem Infrastruktur-Modul auf (vgl. Abbildung 4.4) und müssten daher theoretisch dessen statische Bibliothek beim Link-Vorgang einbinden. Wie bereits in Abbildung 4.4 beispielhaft dargestellt, wird davon ausgegangen, dass das Auftrags-Modul zusätzliche Abhängigkeiten zum Artikel- und Kunden-Modul besitzt. Aus diesem Grund muss das Auftrags-Modul gegen diese beiden statischen Bibliotheken linken. Damit alle Module kompiliert und in der fertigen Anwendung verlinkt werden, müssen für das iOS-Application-Target theoretisch folgende Build-Konfigurationen gewählt werden:

1. Alle statischen Bibliotheken und alle Bundles müssen in der „Target Dependencies"-Phase referenziert sein
2. Alle statischen Bibliotheken müssen in der „Link Binary with Libraries"-Phase referenziert sein
3. Alle Bundles müssen in der „Copy Bundle Resources"-Phase referenziert sein

Entgegen der zweiten Bedingung sind in Abbildung 4.5 auf drei der vier Module richtigerweise keine „Link Binary with Libraries"-Referenzen vom Application-Target eingezeichnet. Der Grund ist, dass die statischen Bibliotheken bereits beim Link-Vorgang der Auftrags-Bibliothek eingebunden werden und der Linker im Fall einer doppelten Verlinkung mit dem Verweis auf Duplikatsymbole abbrechen würde. Aus dem gleichen Grund besitzt das Auftrags-Modul keine direkte Referenz auf die Infrastruktur, da diese bereits vom Artikel-Modul referenziert wird. Problematisch ist die Situation des Kunden-Moduls, dessen Abhängigkeit zum Infrastruktur-Modul in keiner Weise erkennbar ist, weil Target-Dependencies zwischen Subprojekten der gleichen Ebene nicht definierbar sind. Eine Abhängigkeit in der „Link Binary"-Phase führt beim Link-Vorgang des Auftrags-Moduls zum Fehler, wenn die Infrastruktur sowohl vom Artikel- als auch vom Kunden-Modul eingebunden wird. Die Build-Einstellungen müssen also den anderen eingebundenen Modulen und deren transitiven Abhängigkeiten angepasst werden[91].

Die einzig mögliche **Build-Reihenfolge** mit der dargestellten Konfiguration sieht folgendermaßen aus:

1. Infrastruktur: Bundle, Static Library
2. Artikel: Bundle, Static Library
3. Kunden: Bundle, Static Library
4. Auftrag: Bundle, Static Library
5. Superprojekt: iOS Application Bundle

Aus logischer Sicht ist die Reihenfolge der Artikel- und Kunden-Module austauschbar. Im umgekehrten Fall müsste aber statt des Artikel-Moduls das Kunden-Modul die Infrastruktur einbinden.

[91] In der Standardkonfiguration erkennt und ignoriert der Linker doppelt eingebundene Bibliotheken. Dann werden allerdings auch Klassen ignoriert, die nur dynamisch geladen werden, was im weiteren Verlauf dieses Abschnitts diskutiert und durch das Setzen des `-ObjC`-Linker-Flags ermöglicht wird.

Es ist von großer Bedeutung, dass das Infrastruktur-Modul vollkommen unabhängig von den weiteren Modulen existiert. Dies bedeutet insbesondere, dass es keine der Modulklassen importieren darf. Dennoch sollen die Module zur Laufzeit von der Infrastruktur geladen und aufgerufen werden. Möglich wird dies durch das **dynamische Klassenladen** von Objective-C. Die Information darüber, welche Module geladen werden sollen, wird in einer PLIST-Datei gehalten. Der Inhalt der PLIST sollte mit den tatsächlich beim Link-Vorgang eingebundenen Modulen übereinstimmen, weil ansonsten entweder ungenutzte Module die Anwendungsgröße erhöhen oder Module zur Laufzeit nicht gefunden werden können. Zum dynamischen Laden von Klassen eines Moduls kann folgender Code-Ausschnitt genutzt werden:

```
1  // PLIST auslesen
2  NSArray *plist = [NSArray arrayWithContentsOfFile:plistPath];
3
4  for (NSString *moduleControllerName in plist) {
5      // Klasse anhand ihres Namens laden
6      Class moduleBaseController = NSClassFromString(moduleControllerName);
7      if (moduleBaseController) {
8          // Laden erfolgreich
9          id instance = [[moduleBaseController alloc] init];
10         // Referenz speichern
11         [self.loadedModules addObject:instance];
12     } else {
13         // Klasse konnte nicht geladen werden
14     }
15 }
```

Um zusätzliches Wissen über die geladenen Klassen zu erlangen, kann das Infrastruktur-Modul Protokolle bereitstellen, die die geladenen Klassen implementieren müssen. Damit verhindert wird, dass beim Link-Vorgang scheinbar ungenutzte Klassen übergangen werden, sollte in allen Code-Targets das `-ObjC`-Flag unter „Other Linker Flags“ der Build Settings eingetragen werden.

Die Kombination aus Subprojekten, Static Libraries und deren Referenzierung in der „Link Binary with Libraries“-Phase hat viele Vorteile, die noch einmal kurz zusammengefasst werden sollen. Falls referenzierte Bibliotheken fehlen, werden die entsprechenden Targets automatisch gebaut, sofern sie in Subprojekten oder dem gleichen Workspace liegen. Zyklen sind zwar definierbar, führen aber spätestens nach einem Clean-Befehl zum Scheitern des Link-Vorgangs[92]. Somit wird das in [Kno12, Kapitel II.12 Levelize Build] empfohlene und in Abschnitt 2.2.1 dieser Arbeit beschriebene „Levelize Build“-Pattern bei der Verwendung von Subprojekten zwangsweise durchgesetzt. Der beschriebene Architekturvorschlag ist nicht trivial in der Umsetzung, führt aber zu einer klaren Abgrenzung der Module und zentral einsehbaren Abhängigkeiten. Xcode Subprojekte werden daher im in Kapitel 5 vorgestellten Gesamtkonzept eine zentrale Rolle spielen.

[92] Ohne den Clean-Befehl kann auf Bibliotheken vorheriger Builds zugegriffen werden

4.3.4 Source Control Management (Git)

Um große Softwareprojekte zu entwickeln und verschiedene Versionen zu verwalten, sind SCM-Werkzeuge unerlässlich. In dieser Arbeit wird im Folgenden nur **Git** näher betrachtet werden. Im Vergleich zu anderen, großteils zentralen SCM-Werkzeugen hat Git folgende Vorteile:

- Integration in Xcode[93]
- Hoher Verbreitungsgrad [ecl12]; große Auswahl an grafischen Oberflächen und kostenlosen Remote-Repositories[94]
- Lokale und verteilte Repositories ermöglichen uneingeschränkte Offline-Arbeit sowie verteilte Hierarchien von Repositories
- Schnelligkeit (u. a. bedingt durch lokale Repositories)
- Einfache Branches- und Tags-Erstellung; einfaches Mergen; lokale Branches und Tags
- Dateien können ohne Verlust der Versionshistorie im Repository verschoben werden

Einige der genannten Vorteile gelten auch für andere SCM-Werkzeuge wie SVN oder Mercurial, aber keins dieser Werkzeuge kann alle Vorteile vereinen. Ein Vergleich zwischen zentralen und dezentralen Systemen bzw. SVN und Git im Speziellen kann in [Cam10] gefunden werden.

Im einfachsten Fall wird ein Projekt in seiner Gesamtheit in **einem Repository** gehalten. Für die in Abschnitt 4.3.3 vorgestellte Projektstruktur würde dies bedeuten, dass sich sowohl das Superprojekt als auch seine Subprojekte in einem Repository befinden. Mit Git ist es nicht möglich, Teile eines Repositories auszuchecken. Dies impliziert, dass man Module nur unter Verlust der Versionshistorie im Dateisystem kopieren könnte, um sie in mehreren Projekten zu nutzen. Eine versionskontrollierte Modulentwicklung wäre nur schwer möglich.

Eine bessere Lösung ist es, jedes Modul in einem **eigenen Repository** zu halten. Die verteilte Modulentwicklung ist in diesem Fall problemlos möglich. Für Projekte, die auf andere Module zurückgreifen müssen, gelten allerdings zusätzliche Bedingungen. Repositories sollten nie im Dateisystem geschachtelt werden. Dies hätte zur Folge, dass Unterprojekte sowohl in ihrem eigenen als auch im Repository des Superprojekts versioniert werden würden, was den Speicheraufwand erhöhen und u. U. zu Inkonsistenzen führen würde. Das zweite Problem ist der fehlende Zusammenhang zwischen den Projekten. Wird ein Superprojekt zum ersten Mal aus dem Repository ausgecheckt, muss der Entwickler die im Xcode-Projekt referenzierten Module selbst identifizieren, deren Repositories finden, jedes Repository auschecken und darauf achten, dass die relativen Pfade zwischen Super- und Subprojekten richtig gesetzt sind. Darüber hinaus muss dem Entwickler bekannt sein, welcher Commit benötigt wird, da die aktuelle Version des Superprojekts nicht zwangsweise mit allen Versionen und Branches der Module kompatibel sein muss.

[93] `http://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/085-Save_and_Revert_Changes_to_Projects/manage_project_changes.html`

[94] z. B. Bitbucket: `https://bitbucket.org/`

Git Submodules folgen einem ähnlichen Prinzip wie Xcode Subprojekte und sind daher gut mit diesen vereinbar. Für jedes Modul wird weiterhin ein eigenes Repository verwaltet. Diese Repositories können in beliebigen anderen Repositories als sogenannte Submodules eingebunden werden. Anstelle des kompletten Repositories wird dabei nur ein einzelner, durch seinen Hash-Wert genau definierter Commit („Snapshot") im Repository des Superprojekts referenziert. Der Benutzer des Submodules erhält dadurch die volle Kontroller darüber, welcher Stand ausgecheckt werden soll. Wird das Repository des Superprojekts erstmalig ausgecheckt, werden die Submodules selbst nicht geladen, sondern nur deren Repository-Informationen. Um die referenzierten Snapshots aller Submodules zu laden, sind initial zwei Git-Befehle nötig: `git submodule init` und `git submodule update`. Sind die Submodules einmal initialisiert, genügt der Update-Befehl, um den aktuell im Superprojekt referenzierten Snapshot zu laden. Im Gegensatz zum zuvor diskutierten Ansatz ohne Submodules liegen die Snapshots der Unterprojekte im Dateisystem zwangsläufig innerhalb des Superprojekts. Git erkennt allerdings, dass es sich dabei nur um Snapshots aus anderen Repositories handelt und berücksichtigt keine der Dateien für Commits im Superprojekt. Um Änderungen an Submodules durchzuführen und im Repository des Superprojekts sichtbar zu machen, sind mehrere Schritte nötig:

1. Vor Änderungen im Submodule: Lokalen Branch innerhalb des Submodules erstellen
2. Änderungen an den Dateien des Snapshots durchführen
3. Änderungen auf lokalem Branch committen
4. Bei verteilten Repositories: Lokalen Branch auf Remote-Repository pushen
5. Superprojekt lokal committen: Der Hash-Wert des Commits auf dem lokalen Branch im Submodule wird nun im Superprojekt referenziert
6. Superprojekt pushen

Bei der Arbeit mit Remote Repositories ist es wichtig, dass zuerst die Änderungen am Submodule im Remote Repository verfügbar sind bevor der Commit des Superprojekts verteilt sichtbar ist. Andernfalls existiert im Superprojekt eine Referenz auf einen Snapshot, der nur lokal verfügbar ist. Xcode unterstützt Git Submodules leider nicht. Lediglich Änderungen am Hauptrepository können über die Xcode-GUI verwaltet werden, die Verwaltung der Submodules muss über die Kommandozeile oder andere GUI-Werkzeuge vorgenommen werden. Problematisch ist darüber hinaus die Versionierung der **Xcode-Projektdatei**, deren Merge-Vorgang zwar theoretisch möglich, aber oft problembehaftet ist. Das Mergen der Projektdatei ist v. a. dann kritisch, wenn Dateien zum Projekt hinzugefügt und andere entfernt worden sind. Bei einer von SALT Solutions verwendeten Lösung ignoriert Git den gesamten Inhalt des `.xcodeproj`-Bundles. Die darin enthaltene `project.pbxproj`-Datei, die u. a. die Target-Konfigurationen speichert, wird in einem separaten, versionskontrollierten Ordner gehalten. Diese Projektdatei enthält jedoch keine Referenzen zu den Dateien des Projekts und durch die separate Speicherung wird sie bei der Arbeit am Projekt auch nicht von Xcode aktualisiert. Beim erstmaligen Checkout muss der Entwickler daher folgenden Schritten folgen:

1. Die „project.pbxproj“-Datei aus einem bekannten Ordner in das `.xcodeproj`-Bundle kopieren
2. Projekt in Xcode öffnen
3. Alle Projektverzeichnisse und Dateien dem Projekt in Xcode hinzufügen

Somit baut jeder Entwickler die Projektdatei lokal neu auf, allerdings werden die wichtigen Target-Informationen von Git verwaltet. Werden diese lokal verändert, muss die `.pbxproj`-Datei entsprechend aus dem Bundle zurückkopiert werden. Insbesondere bei der Verwendung von Subprojekten erhöht sich der Verwaltungsaufwand der Projektdateien, dafür entfallen viele Schwierigkeiten, die beim Mergen der Projektdateien regelmäßig entstehen. Kurz gesagt werden leicht reproduzierbare, aber beim Merge-Vorgang fehleranfällige Informationen von Git ignoriert, um die essenziellen Informationen problemlos mit Git verwalten zu können.

4.3.5 Dependency-Verwaltung

Bei der Verwendung von Xcode Subprojekten und Workspaces muss der Entwickler selbst für die Beschaffung und Einbindung der Abhängigkeiten sorgen. Mithilfe von Git Submodules kann zwar der Quellcode der Module einfach geladen und versioniert werden, es fehlt aber eine explizite Angabe darüber, welche Version überhaupt benötigt wird. Werkzeuge zur Dependency-Verwaltung setzen an diesem Punkt an. Sie benötigen lediglich die Namen der Abhängigkeiten sowie deren Version, um die Abhängigkeiten zu finden, zu beschaffen und entweder in die Anwendung zu integrieren oder die Anwendung samt ihrer Abhängigkeiten zu bauen.

Das in Ruby geschriebene Werkzeug **CocoaPods**[95] ist für die Verwaltung von Open-Source-Bibliotheken von Drittanbietern gedacht. Jede zur Einbindung geeignete Bibliothek muss über eine „Pod Spec“ genannte Beschreibungsdatei verfügen. Zu den wichtigsten Informationen gehören der Name, eine Versionsnummer, das SCM-Repository und ein zur Versionsnummer passender Tag sowie der Pfad zu den Quellcodedateien. Jede CocoaPods-Bibliothek kann zudem eigene Abhängigkeiten angeben, die rekursiv aufgelöst werden.

Die `.podspec`-Datei muss nun in einem zentralen Repository hochgeladen werden, damit sie von CocoaPods gefunden werden kann. CocoaPods stellt dafür selbst ein Github-Repository für öffentliche Bibliotheken bereit. Alternativ kann CocoaPods so konfiguriert werden, dass Pod Specs in privaten Repositories oder lokalen Verzeichnissen gesucht werden. In jedem Fall wird in diesen Repositories nur die `.podspec`-Datei gespeichert, der Quellcode der Bibliothek wird anschließend aus dem in der `.podspec`-Datei referenzierten SCM-Repository gezogen.

Projekte, die CocoaPods-Bibliotheken nutzen, geben ihre Abhängigkeiten im sogenannten „Podfile“ an. Im einfachsten Fall enthält die Datei nur die Namen der Bibliotheken und die Zielplattform. Die Angabe einer Version ist optional, ebenso die Target-spezifische Angabe von Abhängigkeiten. CocoaPods lädt die Quellcodedateien aller Abhängigkeiten aus deren SCM-Repositories, baut aus diesen in einem separaten Projekt eine statische Bibliothek und bindet diese Bibliothek per Xcode

[95] `https://github.com/CocoaPods/CocoaPods/`

Workspace an das eigene Projekt. Das Werkzeug unterstützt dabei Bibliotheken mit und ohne ARC, dem automatischen Speichermanagement von Objective-C bzw. dem LLVM-Compiler. Öffnet man den gemeinsamen Workspace, lassen sich sowohl die Quellcodedateien des eigenen Projekts als auch die der CocoaPods-Bibliotheken in Xcode editieren.

Das Werkzeug funktioniert gut für seinen geplanten Einsatzzweck, die Einbindung von quelloffenen Drittanbieter-Bibliotheken. Für eigene Module, die ständig weiterentwickelt werden sollen, ist CocoaPods jedoch aus mehreren Gründen ungeeignet. CocoaPods lädt zwar die Abhängigkeiten aus SCM-Repositories, kopiert die Quellcodedateien und Ressourcen dann aber standardmäßig unter Verlust der Versionierung und Ordnerstruktur in ein eigenes Verzeichnis. Änderungen an den Bibliotheken werden dadurch nicht mit deren Original-Repositories synchronisiert, sondern im Repository des eigenen Projekts verwaltet. Mit dem Befehl `pod install --no-clean` lässt sich die Verbindung zum Original-Repository sowie die Ordnerstruktur erhalten, in diesem Fall sollte allerdings der gesamte „Pods"-Ordner von Git ignoriert werden, um eine doppelte Versionierung zu vermeiden. Eigene Workspaces oder Subprojekte für jede Pods-Bibliothek sind nicht möglich, was hierarchische Imports nach Modulen unterbindet. Auch die Entwicklung der Module wird erschwert, weil CocoaPods pro Bibliothek nur eine flache Xcode-Gruppe für alle Quellcodedateien anlegt. Bibliotheken können in ihren Pod Specs Ressourcen referenzieren, die per Skript in die fertige Anwendung kopiert werden, aber in Xcode nicht sichtbar sind. Für die Ressourcen gilt zudem die strenge Auflage, dass deren Namen global eindeutig sein müssen. Build-Konfigurationen lassen sich nur für die Gesamtheit aller Pods und nicht für einzelne Bibliotheken anpassen. Denkbar ist der Einsatz von CocoaPods zur **Verwaltung der Drittanbieter-Bibliotheken aller Module**. In diesem Fall sollte ein Skript die Podfiles der Module auswerten und zu einem Projekt-Podfile zusammensetzen, das nur die Abhängigkeiten der aktuell eingebundenen Module enthält. Die Verwendung von CocoaPods steht nicht in Konflikt zur Verwendung von Subprojekten im Hauptprojekt.

Bei SAP wurde ein **Xcode Maven Plugin**[96] entwickelt, das die Abhängigkeitsauflösung und den Build-Vorgang von Xcode-Projekten automatisieren soll. Das Maven-Plugin ist in erster Linie dafür gedacht, Xcode-Projekte in die Gesamtinfrastruktur eines Unternehmens einzubinden. Im Rahmen dieser Belegarbeit wurde die Möglichkeit der Abhängigkeitsauflösung näher betrachtet, die nur einen kleinen Teil des Funktionsumfangs des Plugins bzw. Mavens selbst ausmacht. Wie bei Java-Projekten besitzen Maven-Projekte zur Selbstbeschreibung eine POM-Datei. Anhand der Informationen aus der POM-Datei werden Abhängigkeiten in lokalen oder zentralen Maven-Repositories gefunden und geladen. Damit eine Abhängigkeit aufgelöst werden kann, muss sie jedoch zuvor in binärer Form im Repository abgelegt worden sein. Das Kompilieren von Abhängigkeiten bei Bedarf wird ebensowenig unterstützt wie die Einbindung von Quellcodebibliotheken. Das Plugin garantiert die physische Präsenz der Abhängigkeit unter einem vordefinierten Pfad, die Verlinkung in der „Link Binary with Libraries"-Phase muss allerdings vom Entwickler in Xcode vorgenommen werden. Nicht zuletzt aufgrund des hohen Einrichtungsaufwands und vieler Verzeichnisstrukturkonventionen wird das Xcode Maven Plugin im weiteren Verlauf der Arbeit keine Rolle spielen. Stattdessen soll das Zusammenspiel von Xcode Subprojekten und Git Submodules näher untersucht werden. Nach Möglichkeit sollte CocoaPods zur Auflösung von Drittanbieter-Bibliotheken genutzt werden.

[96] `http://sap-production.github.com/xcode-maven-plugin/site/index.html`

4.4 MODULARISIERUNG UND KONFIGURATION DES DATENMODELLS

Anwendungen, die Core Data zur Datenhaltung verwenden, besitzen standardmäßig ein Datenmodell, das verschiedene Versionen beinhalten kann. Die Versionen eines Datenmodells sind in einer `.xcdatamodeld`-Bundle-Datei abgelegt. Die im Datenmodell enthaltenen Entitäten entstammen in der Regel der Business-Domäne und lassen sich den Modulen der Anwendung logisch zuordnen. Es stellt sich daher die Frage, inwiefern das Datenmodell tatsächlich auf die Module aufgeteilt werden kann. Technisch gesehen können viele kleine Datenmodelle für die Module erstellt werden. Dabei gehen jedoch jegliche Beziehungen zwischen Entitäten verschiedener Module verloren. Die Nachmodellierung in Code wäre aufwändig, fehleranfällig und langsam, weil Anfragen an mehrere getrennte Datenbanken erfolgen müssten. Aus den genannten Gründen sollte das Datenmodell daher die Entitäten aller Module enthalten und im Infrastruktur-Modul platziert werden.

Wie in Abschnitt 3.1.5 erwähnt, lässt sich im Core-Data-Editor ein Mapping zwischen Entitäten und Modellklassen definieren, um im Code typsicher auf Attribute der Entitäten zugreifen zu können. Auf der Ebene der Modellklassen kann nun eine saubere Trennung nach Modulen erfolgen, weil das **Datenmodell** ohne Probleme kompiliert und zur Laufzeit geladen werden kann, auch wenn nicht alle referenzierten Klassen tatsächlich im Produkt verlinkt worden sind. Das Vorhandensein des gesamten Datenmodells fällt nicht ins Gewicht[97], da nur diejenigen Entitäten über Ausprägungen in der Datenbank verfügen, deren Module in der Anwendung präsent sind. Ein Zugriff auf Klassen oder Entitäten von nicht eingebundenen Modulen muss prinzipiell ausgeschlossen werden.

Bei mehrmandantenfähigen Anwendungen können auch die Datenmodelle der Mandanten voneinander abweichen. Die Grundannahme für die folgenden Überlegungen ist, dass der Großteil des Datenmodells übereinstimmt. Andernfalls muss für jeden Mandant ein eigenes Datenmodell erstellt werden. Um ein Datenmodell möglichst generisch zu gestalten, können pro Entität neben den Attributen, die voraussichtlich allen Mandanten gemein sind, zusätzliche **Reserveattribute** angelegt werden. Es reicht aus, damit den aktuellen Bedarf an Abweichungen abzudecken, da ergänzende Änderungen am Datenmodell ohne Schwierigkeiten möglich sind. Attribute benötigen stets einen Typ, sodass zumindest numerische und String-Reserveattribute angelegt werden sollten. Alternativ kann das Binärformat als generischer Typ verwendet werden.

Der Zugriff auf die Reserveattribute sollte in den Modellklassen gekapselt werden, sodass außerhalb des Datenmodells kein Unterschied zwischen Basis- und Reserveattributen erkennbar ist. Das **Mapping** zwischen mandantenspezifischen Properties und den Reserveattributen kann entweder direkt im Code oder über Konfigurationsdateien erfolgen. Über den SCM-Branch lassen sich beide Arten von Dateien leicht austauschen, wobei die Codedateien in jedem Fall angepasst werden müssen, um eine mandantenspezifische Schnittstelle bereitzustellen. Konfigurationsdateien haben den Vorteil, dass sie an mehreren Stellen verwendet und leicht ausgetauscht werden können, zudem sind sie programmiersprachenunabhängig. In Abbildung 4.6 wird das Mapping

[97] Eine Version des mit knapp 40 Entitäten relativ umfangreichen Datenmodells von alexa MRS benötigt kompiliert etwa 70 KB an Festplattenspeicher

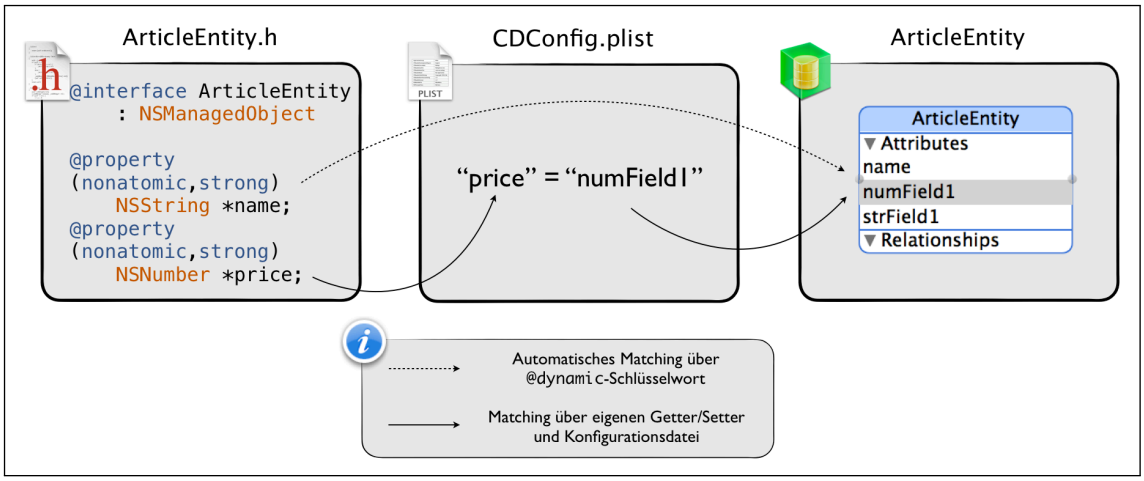


Abbildung 4.6: Mapping von Properties und Core-Data-Attributen

mit einer Konfigurationsdatei für die Beispiel-Entität ArticleEntity deutlich. Das `name`-Attribut ist sowohl in der Modellklasse als auch in der Core-Data-Entität vorhanden, sodass kein spezielles Mapping erforderlich ist. Es genügt die Deklaration der Name-Property und eine `@dynamic`-Anweisung im Implementierungsteil, um die Property mit dem Attribut zu verbinden. Das `price`-Attribut hingegen ist eine mandantenspezifische Ergänzung, die im Datenmodell so nicht vorgesehen ist. Deshalb wird die Property auf einen eigenen Setter umgeleitet, in dem per KVC auf ein Reserveattribut der Entität zugegriffen wird. Der Schlüssel für den Zugriff wird aus der Konfigurationsdatei ermittelt. Der Unterschied ist aus der Header-Datei nicht ersichtlich und somit für benutzende Klassen transparent. Im Folgenden ist ein Ausschnitt der ArticleEntity.m-Datei zu sehen, der den lesenden Zugriff auf das Reservefeld implementiert. Der Ansatz kann auch dazu genutzt werden, Basis-Attribute umzubenennen, falls nötig[98].

```
1  // Zugriff auf gleichnamiges Attribut
2  @dynamic name;
3
4  // Zugriff auf Attribut über Key, der
5  // in Konfigurationsdatei abgelegt ist
6  - (NSNumber *)price {
7      NSString *attributeKey = [self.configFile objectForKey:@"price"];
8      return [self valueForKey:attributeKey];
9  }
```

Reservefelder sollten nur in Ausnahmefällen genutzt werden, da mit jedem generischen Feld die Vorteile einer relationalen Datenbank aufgeweicht werden. Allerdings kann die Verwaltung vieler Datenmodelle schnell unübersichtlich werden, weshalb eine Abwägung zwischen dem Mehraufwand eines eigenen Datenmodells und der Abschwächung der Datenbankvorteile stattfinden muss.

[98] Einfacher ist es hingegen, diese Umbenennung beim Daten-Import bzw. -Export vom Server vorzunehmen, um die Modellklassen unverändert zu lassen

4.5 ZUSAMMENFASSUNG

In diesem Kapitel wurden verschiedene Techniken untersucht, die die Implementierung von mehrmandantenfähigen Anwendungen erleichtern können. Die Abschnitte 4.1 und 4.2 konzentrieren sich dabei auf die Anwendungs-**Ressourcen** und die Gestaltung der **Programmoberfläche**. In vielen Fällen lassen sich die mandantenspezifischen Informationen in Konfigurationsdateien auslagern, sodass der Programmcode von mandantenspezifischen Anpassungen unberührt bleibt.

Sollen mandantenspezifische **Funktionalitäten** eingebunden werden, dienen **Module** dazu, die Reichweite von Änderungen einzugrenzen. Die Untergliederung in Module erlaubt zudem das unabhängige Entwickeln einzelner Anwendungsteile und begünstigt die Wiederverwendbarkeit von Code. Auch wenn Objective-C nativ keine Modulstrukturen anbietet, können mithilfe von Xcode Subprojekten modularisierte Anwendungen gebaut werden. Für die Verwaltung von Modulen bieten sich SCM-Werkzeuge an, speziell Git Submodules können dafür genutzt werden. Die automatische Abhängigkeitsverwaltung ist dagegen nur in Grenzen möglich. Als größtes Problem erweist sich jedoch die Umstrukturierung bestehender Anwendungen, weil nur wenige Werkzeuge die Analyse vorhandener Abhängigkeiten unterstützen. Aus diesem Grund hilft derzeit nur die manuelle Erstellung von Diagrammen, die die Abhängigkeiten von Klassen und Modulen dokumentieren. Um die Gesamtübersicht über eine Anwendung zu erleichtern, sollten insbesondere diejenigen Design Patterns beachtet werden, die von der Plattform und deren Frameworks nahegelegt werden.

5 KONZEPT UND PROTOTYPISCHE IMPLEMENTIERUNG

Im folgenden Kapitel wird ein **Gesamtkonzept** zur Unterstützung von Mehrmandantenfähigkeit auf der iOS-Plattform vorgestellt. Das Konzept wird schritthaltend von einer **prototypischen Implementierung** begleitet und erläutert. Als Ausgangslage wird ein Projekt verwendet, das einer typischen, nicht-modularen iOS-App entspricht. Diese App wird in Abschnitt 5.1 ausführlich vorgestellt. Im darauffolgenden Abschnitt 5.2 werden Anforderungen aufgelistet, die das Ausgangsprojekt nicht ohne Weiteres erfüllen kann und folglich als Leitfaden für die Weiterentwicklung dienen. In Abschnitt 5.3 soll versucht werden, jeder Anforderung mit einer geeigneten Maßnahme zu begegnen, sodass die Anwendung am Ende dieses Kapitels idealerweise den gesamten Anforderungskatalog erfüllen kann. Alle getroffenen Maßnahmen werden anschließend in Abschnitt 5.4 zusammengefasst. Eine Evaluation des Konzepts findet schließlich in Kapitel 6 statt.

5.1 AUSGANGSLAGE

Als Ausgangspunkt dient eine prototypische ERP-Anwendung, die aus drei **Anwendungsteilen** besteht, welche später in Module umgewandelt werden sollen. Die drei Anwendungsteile sind eine Artikelübersicht, eine Kundenübersicht und eine Auftragsübersicht. Alle Anwendungsteile verfügen über eigene Modellklassen, zu denen es entsprechende Entitäten im Core-Data-Modell gibt. Um die Anwendung mit Testdaten zu befüllen, werden diese im PLIST-Format von einem öffentlichen Webserver[99] bereitgestellt und bei Bedarf geladen[100]. Die Anwendung wurde für den fiktiven Mandanten *T1* in einem Xcode-Projekt ohne Subprojekte oder sonstige externe Abhängigkeiten implementiert und in einem Git-Repository verwaltet. Ein Snapshot des *T1-ERP-App* benannten Ausgangsprojekts befindet sich in Anhang B.1.

[99] `https://bitbucket.org/shagedorn/public-resources-repository/`
[100] Die Testdaten liegen zudem lokal im Ressourcen-Ordner der Infrastruktur, werden im Projekt aber nicht genutzt

5.1.1 Struktur der Anwendung

Um die spätere Modularisierung zu erleichtern, wurden bereits einige Vorkehrungen getroffen. Die Klassen und Ressourcen jedes Anwendungsteils liegen in getrennten **Verzeichnissen** des Dateisystems. Zusätzlich existiert ein Infrastruktur-Ordner, in dem gemeinsam genutzte Klassen, Ressourcen und Protokolle abgelegt worden sind. Auch die von allen Anwendungsteilen genutzte Netzwerk-Bibliothek *AFNetworking*[101] liegt in einem separaten Ordner.

Das **MVC-Pattern** gibt die Struktur der Anwendung vor. Es existiert ein ViewController für den Startbildschirm („StartScreenViewController"), aus dem die ViewController der anderen Anwendungsteile gestartet werden können. Aus Abbildung 5.1 sind die einzelnen Bereiche der Anwendung ersichtlich. Die Anwendungsteile wurden weitestgehend unabhängig voneinander implementiert, einige übergreifende Klassen-Imports wurden allerdings zu Demonstrationszwecken eingeführt. Wie in den vorherigen Beispielen der Abschnitte 4.3.2 und 4.3.3 besteht eine gewollte Abhängigkeit der Auftragsübersicht von der Artikel- und Kundenübersicht. Diese Abhängigkeit besteht sowohl im Datenmodell als auch in den Klassen der Auftragsübersicht.

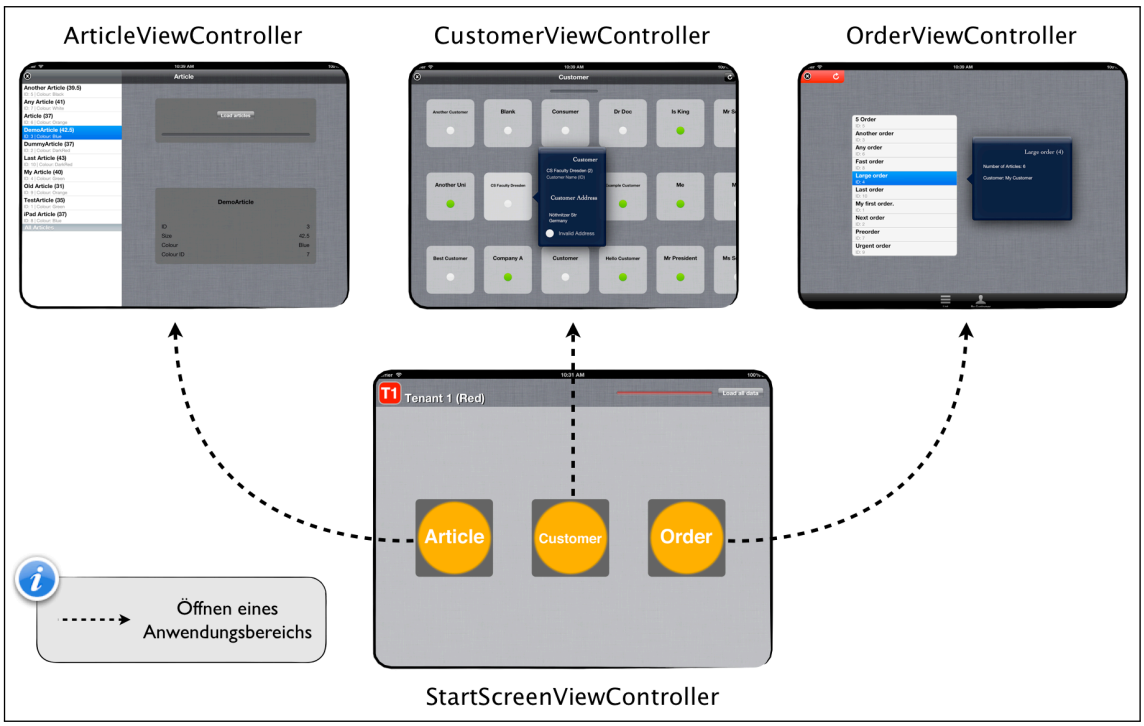


Abbildung 5.1: Struktur der Beispielanwendung

Auf die Verwendung von Storyboards wurde verzichtet, um die GUI-Dateien (XIBs) auf die einzelnen Ordner aufteilen zu können. Storyboards hingegen umfassen die GUI für mehrere ViewController (typischerweise die GUI der gesamten App) und beeinträchtigen dadurch die Flexibilität. Die sichtbaren Strings aller Anwendungsbereiche[102] werden in einer lokalisierbaren Strings-Datei verwaltet. Jeder String-Schlüssel verfügt über einen Modulpräfix, um Namenskollisionen zu vermeiden. Zur automatischen Anpassung der GUI wurde stellenweise Auto Layout eingesetzt.

[101] `https://github.com/AFNetworking/AFNetworking/`
[102] Von der Lokalisierung der Grafik- und Server-Ressourcen wurde in diesem Prototyp abgesehen

5.1.2 Das Module-Protokoll

Die Kopplung zwischen der Infrastruktur und den Anwendungsbereichen soll möglichst lose gestaltet werden. Der StartScreenViewController interagiert nie mit den konkreten Schnittstellen der Teilbereichs-Controller, sondern lediglich mit dem in der Infrastruktur definierten **Module-Protokoll**[103]. Jeder Teilbereich muss eine Klasse bereitstellen, die das Module-Protokoll implementiert – im Fall der T1-ERP-App sind das jeweils die primären ViewController der Bereiche. Derzeit importiert der StartScreenViewController allerdings noch all diese ViewController, um sie anfangs zu instanziieren. Auf diese Imports soll in späteren Versionen der Anwendung verzichtet werden.

Das Module-Protokoll deklariert sechs Properties und Methoden:

- `@property CloseModuleBlock closeModuleBlock`: Das Modul kann diesen Codeblock[104] nutzen, um zum StartScreenViewController zurückzukehren, der den Block beim Starten setzt. Dadurch kann das Modul z. B. als modaler Controller oder als Teil einer Navigations-Hierarchie geladen werden, ohne davon explizit in Kenntnis gesetzt zu werden.
- `- (UIImage*) modulePreviewImage`: Das Bild wird auf dem Startbildschirm zur Repräsentation des Moduls angezeigt.
- `- (NSString*) moduleDescription`: Ein kurzer Name des Moduls.
- `- (UIViewController*) moduleLaunchController`: Der zurückgegebene Controller wird vom Startbildschirm beim Öffnen des Moduls geladen. Falls der Bereichs-Controller selbst das Module-Protokoll implementiert, wird an dieser Stelle `self` zurückgegeben. Die Methode ermöglicht aber auch die Einbettung anderer ContainerViewController[105]. In der Beispielanwendung wird vom Auftrags-Modul ein TabViewController angelegt und zurückgegeben.
- `- (NSArray*) moduleDependencies`: Der StartScreenViewController kann sich über die Abhängigkeiten eines Moduls informieren. Diese Information wird insbesondere für die Reihenfolge beim Datenladen benötigt.
- `- (void) moduleLoadData:(NSManagedObjectContext*)context usingBlockForProgress:(UpdateBlock)updateBlock andCompletion:(CompletionBlock)completionBlock`: Diese Methode löst das Datenladen eines Moduls aus. Damit der Aufrufer über den Fortschritt benachrichtigt wird, werden zwei Codeblöcke übergeben, die vom Modul beim Ladefortschritt und der Beendigung aufgerufen werden sollten.

Das Protokoll ermöglicht eine hohe Abstraktion von den konkreten Klassen der Anwendungsbereiche und stellt die Grundlage für eine modulare Architektur da, die in den folgenden Schritten eingeführt werden soll.

[103] Um spätere Umbenennungen zu vermeiden, wurde an einigen Stellen bereits die Modul-Terminologie eingeführt – Module sind dabei nur die eingangs vorgestellten Anwendungsbereiche

[104] `http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/Blocks/`

[105] `http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/CreatingCustomContainerViewControllers/CreatingCustomContainerViewControllers.html`

5.1.3 Zwischenfazit

Die Ausgangsanwendung entspricht von ihrer inneren Struktur bereits weitestgehend einer modularen Anwendung. Um die Vorteile der Modularisierung in Hinblick auf die Mehrmandantenfähigkeit auszuschöpfen, bedarf es aber auch einer faktischen Entkopplung der Anwendungsteile, sowohl in der Entwicklungsumgebung Xcode als auch im SCM-Repository.

5.2 ANFORDERUNGSANALYSE

Für die Anforderungsanalyse wurde ein **Beispielszenario** entwickelt, das möglichst vielfältige Anforderungen an die Anpassung der Anwendung für mehrere Mandanten beinhaltet. Die für Mandant *T1* entwickelte ERP-Anwendung soll im Folgenden auch den Mandanten *T2* und *T3* zur Verfügung gestellt werden. T2 und T3 haben dabei folgende Anpassungswünsche:

AF.1 T2 und T3 möchten ihr eigenes Farbschema und eigene Grafiken auf die App anwenden

AF.2 T2 benötigt nur die Kundenübersicht

AF.3 T2 möchte die Anwendung auch auf Deutsch nutzen und Strings der englischen Lokalisierung ändern

AF.4 T3 hält die Auftragsinformationen nicht im PLIST-, sondern im JSON-Format; zudem umfassen Aufträge bei T3 noch ein zusätzliches Feld für das Datum

AF.5 Nachdem alle Anpassungen durchgeführt worden sind, wurden die Infrastruktur und die Artikelübersicht weiterentwickelt – die Veränderungen sollen allen Mandanten zur Verfügung gestellt werden

AF.6 Auch das Auftrags-Modul erhält generelle Anpassungen, die allen Mandanten zur Verfügung stehen sollen

Für die überschaubare Beispielanwendung könnten alle Änderungen relativ einfach mit SCM-Branches und Codeanpassungen vorgenommen werden. Damit dies aber auch für größere Projekte und eine deutlich höhere Anzahl an Mandanten umsetzbar ist, sollen die in Kapitel 4 vorgestellten Ansätze zur Umsetzung verwendet werden. Der Fokus gilt dabei dem Entwickler, der mit möglichst wenigen und einfachen Schritten die mandantenspezifische Version wechseln und Änderungen synchronisieren können soll. Alle Beispieländerungswünsche stehen stellvertretend für eine regelmäßige und fortlaufende Entwicklung der allgemeinen App sowie der mandantenspezifischen Ableger.

5.3 MASSNAHMEN UND IMPLEMENTIERUNG

Die in der Anforderungsanalyse beschriebenen Änderungswünsche werden im Folgenden nacheinander implementiert und die wesentlichen Schritte dokumentiert. Am Ende jedes Abschnitts

werden die Git-Tags angegeben, mit denen der entsprechende Stand aus dem in Anlage B.2 enthaltenen Projekt geladen werden kann. Der in Anlage B.1 enthaltene Snapshot des Ausgangsprojekts entspricht dem Tag `T1-ERP-APP`.

5.3.1 SCM-Branches und Targets

> *„AF.1: T2 und T3 möchten ihr eigenes Farbschema und eigene Grafiken auf die App anwenden"*

Um überhaupt verschiedene Anwendungen bauen zu können, ohne vor jedem Mandantenwechsel grundlegende Informationen wie den App-Namen neu anzugeben, muss für jede Mandantenversion ein **Target** angelegt werden. Gleichzeitig werden im **Git-Repository** vier neue Branches angelegt: `t1`, `t2`, `t3` und `general_dev`. Die Targets und deren spezifische Info-PLISTs und Build-Schemes[106] sind nur im dazugehörigen Branch sichtbar, um Vermischungen und Verwechslungen auszuschließen. Auf diese Weise kann eine mandantenspezifische Anwendung nur gebaut werden, wenn zuvor der richtige Branch ausgecheckt worden ist. Alle mandantenspezifischen Änderungen und Informationen sollten nur in den jeweiligen Branches eingefügt werden, während generelle Anpassungen wie Bugfixes und Weiterentwicklungen im `general_dev`-Branch umgesetzt werden.

In einem ersten Schritt wurden nun sämtliche mandantenspezifischen Informationen und Ressourcen aus dem `general_dev`-Branch entfernt[107]. Um für rein optische Anpassungen keinen Code ändern zu müssen, wurden Farb- und Konfigurationsinformationen in eine **PLIST-Ressource** („ColourScheme.plist") ausgelagert. Eine Singleton-Klasse namens „GUIStyle" liest diese PLIST beim Start der Anwendung aus und setzt über **Appearance-Proxies** die darin enthaltenen Farbwerte für bestimmte GUI-Elemente global. Im vorliegenden Prototyp werden auf diese Weise die Färbungen der Ladebalken und der Navigationsleisten gesetzt. Die in der PLIST definierten Attribute werden an keiner anderen Stelle im Programmcode gesetzt.

Die Änderungen auf dem Entwicklungsbranch müssen von allen Mandantenbranches übernommen werden, sofern diese bereits angelegt wurden. Zur farblichen Anpassung der Anwendung muss lediglich im jeweiligen Branch die „ColourScheme"-PLIST editiert werden. Die Grafiken wurden im Dateisystem ausgetauscht und die Dateinamen beibehalten, sodass in Xcode keinerlei Anpassungen stattfinden müssen. Insgesamt wurden im ersten Schritt die Target-Informationen (Produkt- und Mandantenname), das Logo, die Ladegrafik, die Farbinformationen und die Build-Schemes in den einzelnen Branches angepasst. Für einen Mandantenwechsel genügt ein Checkout des Mandantenbranches. Es wird empfohlen, das Xcode-Projekt vor dem Checkout zu schließen und anschließend einen Clean-Vorgang durchzuführen.

Tags: `DEV-ERP-APP-TARGET`, `T1-ERP-APP-TARGET`, `T2-ERP-APP-TARGET`, `T3-ERP-APP-TARGET`

[106] Damit Build-Schemes von SCM-Werkzeugen verwaltet werden können, muss die „Shared"-Option aktiviert werden
[107] Dieser Schritt ist nur nötig, wenn eine mandantenspezifische Anwendung als Ausgangspunkt genutzt wird

5.3.2 Modularisierung

„AF.2: T2 benötigt nur die Kundenübersicht"

Die oben genannte Anforderung ist programmatisch leicht zu bewerkstelligen, weil die Kundenübersicht zumindest aus logischer Sicht keine Abhängigkeiten zum Artikel- oder Auftragsbereich besitzt und somit ohne Weiteres einzeln verwendet werden könnte. Aus dem aktuellen Projekt ist jedoch keinesfalls ersichtlich, dass keine Abhängigkeiten von der Kundenübersicht ausgehen. Im Zweifel könnte der Code der anderen Bereiche weiterhin Teil der Anwendung bleiben, lediglich das Laden der ViewController würde deaktiviert werden. Bei einer größeren Anzahl an Modulen würde dies die Gesamtgröße der App unnötig in die Höhe treiben. Deshalb soll die Anwendung, auch in Vorbereitung auf die nächsten Schritte, in diesem Abschnitt **modularisiert** werden. Zuerst soll kurz analysiert werden, welche Abhängigkeiten aktuell zwischen den Anwendungsbereichen bestehen. Vom logischen Design nicht vorgesehene Abhängigkeiten (vgl. Abbildung 4.4) sollen im Zuge der Analyse beseitigt werden. Im darauffolgenden Schritt wird die Anwendung mit Hilfe von Xcode Subprojekten aufgeteilt. Die strukturellen Veränderungen sollen auf dem Entwicklungsbranch durchgeführt und von den Mandantenbranches übernommen werden.

Abhängigkeits-Analyse

Wie in Abschnitt 4.3.1 beschrieben, wird die Abhängigkeitsanalyse von keinem Werkzeug umfassend unterstützt. Im Fall der prototypischen Beispielanwendung hilft die überschaubare Klassenanzahl und das im Voraus bekannte Ziel der Modularisierung, Strukturfehler mit geringem Aufwand zu finden. In sehr großen Projekten kann es u. U. einfacher sein, zuerst die Zerlegung in Subprojekte durchzuführen und die unvorhergesehenen Abhängigkeiten anhand von Compiler- und Linkermeldungen aufzulösen.

Mit dem in Kapitel 4 vorgestellten Werkzeug objc_dep wurde ein **Abhängigkeitsgraph** erstellt, der in gerenderter Form in Anhang A.3 enthalten ist. Leider sind die Verzeichnisse, also die Modulkandidaten, nicht im Graph enthalten. Dennoch lassen sich einige wichtige Erkenntnisse ableiten:

1. Der StartScreenViewController besitzt nur Abhängigkeiten zu den Basis-ViewControllern der einzelnen Bereiche und zu den Protokollen der Infrastruktur

Die Abhängigkeiten zu den Basis-ViewControllern sollen im Zuge der Modularisierung beseitigt werden, sind aber momentan nachvollziehbar und notwendig.

2. Die Model-Klassen besitzen keine Abhängigkeiten zu Controller- oder View-Klassen
3. Die einzige eigene View-Klasse („CustomerCollectionViewCell") besitzt eine Abhängigkeit zu einer Model-Klasse

Während die zweite Erkenntnis für eine stringente Einhaltung des **MVC-Patterns** spricht, erweckt die dritte Beobachtung einen gegenteiligen Eindruck. Die model-spezifische View-Klasse wurde jedoch bewusst eingeführt, um den Controller möglichst frei von trivialem Initialisierungscode zu halten und den vom UIKit-Framework gegebenen Mechanismus zum Recycling von instanziierten CollectionViewCell-Objekten auszunutzen. Eine Wiederverwendung des Codes außerhalb des Kundenbereichs ist aufgrund der Einfachheit der Klasse nicht erstrebenswert. Da die Abhängigkeit die Grenzen des Kundenbereichs nicht überschreitet, hat sie auf die geplante Modularisierung keine Auswirkungen.

4. Es bestehen zyklische Abhängigkeiten zwischen den Model-Klassen

Die zyklischen Abhängigkeiten resultieren aus einer Empfehlung seitens Apple, Relationen im Core-Data-Datenmodell stets mit einer **inversen Relation** zu versehen. Fehlende inverse Relationen führen zu Compiler-Warnungen in Xcode, die es zu vermeiden gilt. Die gegenseitigen Abhängigkeiten der Entitäten sollten sich jedoch nicht auf die Model-Klassen übertragen, da sonst zwischen sämtlichen Modulen **zyklische Abhängigkeiten** bestehen. Toleriert werden können ausgehende Abhängigkeiten der Auftrags-Klasse, weil der Auftragsbereich gewollte Abhängigkeiten zu den anderen Bereichen besitzt. Auch modulinterne zyklische Abhängigkeiten sind zu vernachlässigen. Die Abhängigkeiten der „Article"- und „Customer"-Klassen von der „Order"-Klasse müssen allerdings beseitigt werden. Im Core-Data-Modell sind die betroffenen Relationen bereits optional, sodass die Relationen auf `nil` gesetzt werden dürfen.

Als Lösungsmöglichkeit bieten sich Categories für die Artikel- und Kunden-Klassen an. Beide Klassen verfügen über eine `orders`-Property vom Typ `NSSet`, und die von Core Data benötigten Zugriffsmethoden, die hier am Beispiel der Artikel-Klasse angegeben werden:

```
1 @interface Article (CoreDataGeneratedAccessors)
2 - (void)addOrdersObject:(Order *)value;
3 - (void)removeOrdersObject:(Order *)value;
4 - (void)addOrders:(NSSet *)values;
5 - (void)removeOrders:(NSSet *)values;
6 @end
```

Die `orders`-Relation wird nur indirekt über ihre inverse Relation in der Auftrags-Klasse gesetzt. Theoretisch kann die Property samt ihrer Zugriffsmethoden also entweder entfernt werden oder an Stelle des `Order`-Typs wird in den Signaturen der Zugriffsmethoden der generische Typ `id` verwendet. In beiden Fällen kann der Import der Auftrags-Klasse entfallen[108]. Im zweiten Fall besteht jedoch das Problem, dass die `orders`-Property weiterhin sichtbar (wenngleich nie gesetzt) ist, auch wenn die Auftragsübersicht nicht Teil der Anwendung ist.

Die sauberste Lösung ist es daher, die oben gezeigten Zugriffsmethoden, die Deklaration der `orders`-Property und die zugehörige `dynamic`-Anweisung in eine Category in einer separaten Datei auszulagern. Diese Datei ist im Fall der Artikel-Klasse nicht Bestandteil des Artikel-, sondern des Auftrags-Moduls. Somit ist die Schnittstelle nur verfügbar, wenn die Auftragsübersicht eingebunden ist. Dieser Ansatz wurde auf dem Entwicklungsbranch implementiert und von allen

[108] Auch eine Forward-Declaration mittels `class Order` ist denkbar

Mandantenbranches übernommen. Der Erfolg der Änderungen ist aus dem aktualisierten Graph in Anlage A.4 ersichtlich.

Tags: `DEV-ERP-APP-ACYCLIC-MODEL`, `T1-ERP-APP-ACYCLIC-MODEL`, `T2-ERP-APP-ACYCLIC-MODEL`, `T3-ERP-APP-ACYCLIC-MODEL`

5. Vier Klassen bzw. Protokolle werden besonders häufig und von Klassen aller Bereiche verwendet: „CoreDataAppDelegate", „CoreDataHelperClass", „Module", „ServerEntity"

Die **bereichsübergreifende Benutzung** der genannten Klassen und Protokolle legt eine Einordnung im Infrastrukturbereich nahe. Die tatsächliche Platzierung ist allerdings nicht aus dem Graph, sondern nur aus Xcode bzw. dem Dateisystem ersichtlich. Dabei zeigt sich, dass die „CoreDataHelperClass" fälschlicherweise im Artikel-Bereich platziert wurde. Dies spiegelt ein gängiges Problem wieder, dass Klassen oft dort platziert werden, wo sie zuerst benötigt werden. Später werden diese Klassen auch in anderen Bereichen genutzt, aber die Lage der Klassen wird nicht an die neue Situation angepasst. Durch flache Imports fällt dies in monolithischen Projekten weder beim Entwickeln noch beim Bauen der Anwendung auf. Problematisch ist in erster Linie das Auffinden solcher falsch platzierten Klassen, die anschließend nur verschoben werden müssen. Dies erfordert Anpassungen im Dateisystem und in der Xcode-Projektdatei.

Tags: `DEV-ERP-APP-INFRASTRUCTURE`, `T1-ERP-APP-INFRASTRUCTURE`, `T2-ERP-APP-INFRASTRUCTURE`, `T3-ERP-APP-INFRASTRUCTURE`

Xcode Subprojekte

Das Anlegen und Einbinden von Subprojekten wurde in Abschnitt 4.3.3 bereits ausführlich diskutiert. Das Anwenden des Konzepts auf den vorliegenden Prototyp hat sich dennoch als **Herausforderung** erwiesen.

Zuerst wurden die Ordner der Anwendungsbereiche aus dem Quellcodeordner des Gesamtprojekts in das Wurzelverzeichnis des Git-Repositories verschoben. Im Zuge dessen wurde die *AFNetworking*-Bibliothek in einen Unterordner der Infrastruktur verschoben. In der Projektdatei wurden die Gruppen aller Anwendungsbereiche entfernt. Für jeden der vier Bereiche (Infrastructure, Article, Customer und Order) wurde innerhalb der jeweiligen Ordner ein **neues Xcode-Projekt** erstellt, in dem die vorherige Gruppenstruktur nachgebildet wurde. Alle Projekte besitzen ein Static-Library- und eine Bundle-Target für den Code und die Ressourcen. Die Projekte wurden anschließend dem Ausgangsprojekt als Subprojekte hinzugefügt, die Projekt- und Build-Einstellungen wurden analog zu dem in Abschnitt 4.3.3 vorgeschlagenen Konzept umgesetzt.

Weil die Infrastruktur die Basis-ViewController aller anderen Module importiert und diese wiederum Klassen und Protokolle der Infrastruktur benutzen, ließe sich die Anwendung auch abgesehen von den unten aufgeführten Problemen noch nicht bauen. Um die Abhängigkeitszyklen zu entfernen, werden die Klassen der Module vom StartScreenViewController **dynamisch geladen**. Die Namen der zu ladenden Klassen werden aus einer PLIST-Ressource des Superprojekts („ActiveModules.plist") ausgelesen. Nach dem erfolgreichen Laden jeder darin aufgeführten Klasse

wird geprüft, ob die Klassen das Module-Protokoll (s. Abschnitt 5.1.2) vollständig implementieren. Anschließend wird ermittelt, ob auch deren Abhängigkeiten geladen werden konnten. Module mit fehlenden Abhängigkeiten werden wieder entladen[109]. Jedes Modul gibt seine Abhängigkeiten durch den Rückgabewert der im Module-Protokoll deklarierten `moduleDependencies`-Methode selbst an, wobei die Quelle der Information in jedem Modul eine Konfigurationsdatei („ModuleDescription.plist") ist, die neben den Abhängigkeiten weitere Beschreibungen des Moduls enthält.

Die darüber hinausgehenden **Code-Anpassungen** waren vergleichsweise trivial, können aber in größeren Projekten einen erheblichen Mehraufwand bedeuten. Damit Abhängigkeiten von anderen Modulen klar erkennbar sind, wurden hierarchische Imports verwendet, d. h. es mussten zahlreiche `#import`-Anweisungen aktualisiert werden. Beim Laden von Ressourcen muss stets das Ressourcen-Bundle angegeben werden. Zur Erleichterung des Ressourcenzugriffs wurde die „BundleResourceHelperClass" der Infrastruktur hinzugefügt. Im vorkompilierten Header des Superprojekts definierte Makros mussten in eine Header-Datei der Infrastruktur verschoben werden, um in der gesamten App zur Verfügung zu stehen. Auch die Imports der dynamisch gebundenen Frameworks mussten für alle Module separat angepasst werden.

Zahlreiche unvorhergesehene **Probleme** haben die Modularisierung zu einer komplexen Aufgabe werden lassen. Die „Copy Headers"-Phase verschiebt die Header-Dateien in ein Verzeichnis, das standardmäßig nicht zum Header-Suchpfad von Xcode-Projekten gehört. Als Alternative bleibt das Hinzufügen einer allgemeinen „Copy Files"-Phase oder die Anpassung des „Public Headers Folder Paths" der Module, indem der gewünschte relative Pfad um den Präfix `include/` erweitert wird[110]. Für die erfolgreiche Ausführung eines Build-Vorgangs genügt es, dynamisch verlinkte Frameworks einmal zu referenzieren, sei es in den Modulen selbst oder gesammelt im Superprojekt. Dies erzeugt jedoch Warnungen und Fehler beim Editieren der Quellcodedateien und die Autovervollständigung setzt aus, sodass die dynamischen Frameworks mehrfach eingebunden werden müssen. Für die iOS-Plattform stehen keine Templates zur Erstellung von Bundle-Targets zur Verfügung, sodass Mac-OS-X-Templates verwendet werden müssen. Neben der Anpassung von offensichtlich falschen Build-Einstellungen wie der Prozessorarchitektur und dem verwendeten SDK sollten auch Mac-spezifische Frameworks aus der Link-Phase und dem vorkompilierten Header entfernt werden. Damit PNG- und JPEG-Ressourcen nicht ins TIFF-Format konvertiert werden, muss zudem das `COMBINE_HIDPI_IMAGES`-Flag[111] entfernt werden.

Die beschriebenen Anpassungen ermöglichen ein Bauen der Anwendung für den Simulator und Testgeräte, zum Archivieren nutzt Xcode jedoch andere Verzeichnisstrukturen für die temporären Build-Artefakte. Damit die Header-Dateien der Subprojekte auch für **Archive-Builds** gefunden werden können, muss in allen Code-Targets das `ALWAYS_SEARCH_USER_PATHS`-Flag gesetzt werden und `$(PROJECT_TEMP_DIR)/../UninstalledProducts/include` zu den „User Header Search Paths" hinzugefügt werden. Damit die gebauten App-Archive auch in den App Store eingereicht werden können, dürfen sie letztlich nur ein Produkt enthalten. Aus diesem Grund müssen die `SKIP_INSTALL`-Flags aller Static-Library- und Bundle-Targets gesetzt sein[112].

[109] Das Entladen startet die Abhängigkeitsüberprüfung erneut, da es möglich ist, dass nun anderen Modulen eine Abhängigkeit fehlt

[110] Um die Benutzbarkeit der Module so einfach wie möglich zu gestalten, soll der Header-Suchpfad des Superprojekts unverändert bleiben

[111] Symbolischer Alias: „Combine High Resolution Artwork"

[112] Standardmäßig ist dies nur bei den Static Libraries der Fall

Die Verschiebung der Netzwerk-Bibliothek in die Infrastruktur ist als Übergangslösung vertretbar, weil die Bibliothek derzeit von allen Modulen benutzt wird und über die Infrastruktur gut erreichbar ist. Es fehlt allerdings eine klare Trennung zwischen der eigens implementierten Infrastruktur und dem Code von Drittanbietern, weshalb später CocoaPods zur Verwaltung von **Open-Source-Bibliotheken** zum Einsatz kommen soll. Störend ist momentan v. a., dass sämtliche Header der Bibliothek manuell zur „Copy Headers"-Phase der Infrastruktur hinzugefügt werden müssen. In realen Anwendungen kommen meist zahlreiche Drittanbieter-Bibliotheken zum Einsatz, was den Aufwand entsprechend erhöht.

Der **Merge-Vorgang** der Projektdatei des Superprojekts hat sich als problembehaftet erwiesen, weshalb die daran durchgeführten Änderungen in den Mandantenbranches wiederholt werden mussten. Da die komplexeren Änderungen jedoch in den Projektdateien der Unterprojekte stattgefunden haben, war der Merge-Vorgang insgesamt weitestgehend unproblematisch.

Die Umsetzung des Mandantenwunschs, dem **Ausbau der Artikel- und Auftrags-Module**, war nach der umfassenden Restrukturierung leicht möglich. Für einen rein visuellen Effekt genügt das Entfernen der Modul-Basisklassen aus der „ActiveModules"-PLIST im Branch des Mandanten. Damit Code und Ressourcen der beiden ausgebauten Module die Größe der Anwendung nicht beeinträchtigen, müssen die Referenzen der Static-Library- und Bundle-Targets aus dem Superprojekt entfernt werden. Weil die Infrastruktur bislang vom Artikel-Modul eingebunden wurde, muss sie nach dem Ausbau dieses Moduls nun vom Kunden-Modul verlinkt werden. Dieses Problem wurde bereits in Abschnitt 4.3.3 thematisiert.

Die Modularisierung hat bestätigt, dass die vorherige Analyse und Unterteilung der Anwendung in die verschiedenen Teilbereiche gründlich durchgeführt worden ist und keine ungewollten Abhängigkeiten bestanden haben. Auch nach der Modularisierung kann mit einem einfachen Checkout der SCM-Branches die Mandantenversion gewechselt werden und die Konfiguration der Anwendung ist ohne Codeänderungen möglich. Bis zu diesem Punkt befinden sich alle mandantenspezifischen Informationen in PLIST-Ressourcen, Grafiken und den Projektdateien. Der Preis dafür war eine aufwändige Restrukturierung, die bei größeren Projekten möglicherweise weitere unvorhergesehene Probleme mit sich bringt. Zudem nimmt der Build-Prozess mehr Zeit in Anspruch und für das Inkrafttreten von Code-Änderungen ist mitunter ein Clean-Vorgang nötig.

Tags: `DEV-ERP-APP-MODULES`, `T1-ERP-APP-MODULES`, `T2-ERP-APP-CUSTOMER-ONLY`, `T3-ERP-APP-MODULES`

5.3.3 Git Submodules und stringsbuilder

> „*AF.3: T2 möchte die Anwendung auch auf Deutsch nutzen und Strings der englischen Lokalisierung ändern*"

Alle mandantenspezifischen Anpassungen wurden bislang im Superprojekt durchgeführt. Einzig die Projektdatei des Kunden-Moduls musste im `t2`-Branch angepasst werden, um gegen das Infrastruktur-Modul zu linken, wenn das Artikel-Modul nicht eingebunden ist. Im folgenden Schritt soll stringsbuilder zur einfachen Verwaltung von lokalisierten Strings eingesetzt werden. Dazu ist

es erforderlich, dass die derzeit noch zentral in der „Localizable.strings"-Datei abgelegten String-Ressourcen auf die Module aufgeteilt werden. Um die Änderungen für T2 umzusetzen, müssen demnach Ressourcen-Dateien der Module verändert werden.

Zur schärferen Trennung der Module wurden die Ordner aller Subprojekte aus dem Hauptrepository entfernt, deren Inhalte in einzelne Repositories ausgelagert und diese neuen Repositories anschließend durch die Verwendung von **Git Submodules** wieder zum Hauptrepository hinzugefügt. Nach der Durchführung dieser Schritte ist im Dateisystem keine Veränderung erkennbar[113], aber jedes Modul verfügt nun über ein vollkommen unabhängiges Git-Repository. In jedem Modul-Repository wurde ein `general_dev`-Branch angelegt, nur im Kunden-Modul wurde für die veränderte Projektdatei zusätzlich der `t2`-Branch erstellt. Das Einbinden der Git Submodules konnte im Hauptrepository leicht von den einzelnen Branches übernommen werden, weil in fast allen Mandantenversionen der aktuellste Commit der `general_dev`-Branches der Module referenziert wird. Einzig im `t2`-Branch des Hauptrepositories musste in einem weiteren Schritt der `t2`-Branch des Kunden-Moduls ausgecheckt und im Superprojekt referenziert werden, damit das Projekt wieder erfolgreich gebaut werden kann. Die kostenlose Software **SourceTree**[114] hat sich für die Verwaltung der Git-Repositories als sehr hilfreich erwiesen, da SourceTree nicht nur das Anlegen von Git Submodules unterstützt, sondern diese auch visualisiert und beim Checkout des Gesamtprojekts aktualisiert. Für die Einführung der Git Submodules waren in Xcode keine Änderungen notwendig, weil sich die Dateisystemstrukturen nicht verändert haben. Ein Mandantenwechsel erfordert einen Checkout des Branches im Hauptrepository und ein Update der Submodules.

Tags: `DEV-ERP-APP-SUBMODULES`, `T1-ERP-APP-SUBMODULES`, `T2-ERP-APP-SUBMODULES`, `T3-ERP-APP-SUBMODULES`

Hinweis: Die Repositories der Submodules sind unter Anlage B.3 zusammengefasst. Allerdings referenziert das Hauptprojekt öffentliche Remotes dieser Repositories, um zu verhindern, dass die lokalen Repositories in vorbestimmten Verzeichnissen platziert werden müssen.

Im Folgenden wurde **stringsbuilder** in das Projekt integriert. Das kompilierte Tool befindet sich im Hauptrepository unter `Tools/stringsbuilder/` und wird bei jedem Build-Vorgang durch eine „Run Script"-Phase automatisch gestartet. Jedes Modul verfügt über eine „Strings.csv"-Datei, die die benötigten Strings aller verfügbaren Sprachen enthält. Im Superprojekt wurde die „BaseStrings.plist"-Datei hinzugefügt, die nur diejenigen CSV-Dateien referenzieren sollte, die in jedem Fall eingebunden werden müssen. Im Fall dieses Prototyps ist an dieser Stelle die CSV-Datei des Infrastruktur-Moduls referenziert.

Welche CSV-Dateien zusätzlich, also nach Wunsch der Mandanten, beachtet werden sollen, wird in der „TenantSpecificStrings.plist"-Datei definiert. Sinnvollerweise sollten genau die CSV-Dateien eingebunden werden, deren Module ebenfalls Teil der Anwendung sind – eine Optimierung wäre in diesem Fall ein Skript zur Synchronisierung zwischen der „ActiveModules"- und der „TenantSpecificStrings"-PLIST. Stringsbuilder liest zuerst die „BaseStrings"- und anschließend die „TenantSpecificStrings"-Datei, parst und verarbeitet alle darin referenzierten CSV-Dateien und verwendet als Output-Datei die „Localizable.strings"-Datei, die bislang statisch die Strings der

[113] Abgesehen von versteckten Git-Dateien
[114] `http://www.sourcetreeapp.com/`

gesamten Anwendung beinhaltet hat. Durch die Verwendung von stringsbuilder passt sich ihr Inhalt nun bei jedem Build-Vorgang an, sodass nur die tatsächlich benötigten Strings enthalten sind. Weil der Inhalt ständig automatisch aktualisiert wird, kann die Datei im Repository zur Ignore-Liste hinzugefügt werden oder ohne Rücksicht auf Merge-Konflikte überschrieben werden. Damit sich die Strings-Datei in einem lokalisierten Ordner (z. B. „en.lproj“) befindet, sollte sie vor der Verwendung von stringsbuilder in Xcode lokalisiert werden[115]. Am Dateizugriff zur Laufzeit und damit am Quellcode sind keine Änderungen nötig.

Um die Anforderung AF.3 zu erfüllen, wurde im Infrastruktur-Modul der Branch `t2` angelegt und der gleichnamige Branch im Kunden-Modul ausgecheckt, sofern nicht bereits geschehen. In der „Strings.csv“-Datei der Infrastruktur wurde eine neue Spalte für die **deutsche Übersetzung** hinzugefügt, der gleiche Vorgang wurde für die CSV-Datei des Kunden-Moduls wiederholt. Teil des Änderungswunschs ist es zudem, bestehende Strings zu überschreiben. Dazu wurde eine weitere CSV-Datei im Kunden-Modul angelegt, „MyStrings.csv“. Dieses Vorgehen hat den Vorteil, dass die „Strings.csv“-Datei bis auf die hinzugefügte Sprache unverändert bleibt und neue Sprachen später leicht zwischen Branches ausgetauscht werden können, ohne auch die mandantenspezifischen Strings zu übernehmen. In der „MyStrings.csv“-Datei genügt es nun diejenigen String-Keys samt Übersetzung anzugeben, die überschrieben werden sollen. Im Beispielprojekt wurden die Validierungsnachrichten geändert, die in den Detailansichten der Kunden ausgegeben werden. Damit die neuen Strings von stringsbuilder beachtet werden, muss die neue CSV-Datei der „TenantSpecificStrings“-PLIST hinzugefügt werden. Außerdem muss in Xcode im Superprojekt Deutsch als neue Sprache hinzugefügt werden.

Sobald die Änderungen der Submodules in die Remote-Repositories übertragen worden sind und im Hauptprojekt ein Commit erfolgt ist, genügt weiterhin ein Checkout der Mandantenbranches im Hauptprojekt, um zwischen den mandantenspezifischen Projektversionen zu wechseln. Je nach verwendetem Git-Tool ist möglicherweise ein manuelles Update der Submodules notwendig, SourceTree erledigt dies automatisch.

Tags: `DEV-ERP-APP-SB`, `T1-ERP-APP-SB`, `T2-ERP-APP-DE`, `T3-ERP-APP-SB`

5.3.4 CocoaPods und Reservefelder

> „*AF.4: T3 hält die Auftragsinformationen nicht im PLIST-, sondern im JSON-Format; zudem umfassen Aufträge bei T3 noch ein zusätzliches Feld für das Datum*“

Zur Erfüllung der Anforderung sind zwei getrennte Schritte notwendig. Um alle benötigten Open-Source-Bibliotheken automatisch zu verwalten, wurde zuerst CocoaPods in das Hauptprojekt integriert. Im darauffolgenden Schritt wurde das Datenmodell um ein Reservefeld erweitert, das durch Anpassungen im Auftrags-Modul von T3 für das Datum genutzt wird.

[115] Die Option zur Lokalisierung befindet sich im File Inspector

CocoaPods zur Verwaltung von Bibliotheken

Bislang war die einzige verwendete Open-Source-Bibliothek, *AFNetworking*, Teil der Infrastruktur. Damit die Bibliothek außerhalb der Infrastruktur nutzbar ist, mussten alle Header der Bibliothek in der „Copy Headers"-Phase exportiert werden. Für die folgenden Anpassungen des Auftrags-Moduls soll die *SBJson*-Bibliothek[116] eingesetzt werden. Die statische Platzierung der Bibliothek in der Infrastruktur würde bedeuten, dass auch die Infrastruktur mandantenspezifischen Code enthält, obwohl die Anpassung eigentlich nur das Auftrags-Modul betrifft. Die Platzierung im Auftrags-Modul hingegen hätte zur Folge, dass kein anderes Modul die gleiche Bibliothek nutzen kann, ohne eine Abhängigkeit zum Auftrags-Modul zu besitzen.

Zur Lösung des Problems wird **CocoaPods** eingesetzt, das in einer Textdatei („Podfile") definierte Abhängigkeiten („Pods") mit nur einem Kommandozeilenbefehl automatisch laden und in ein Projekt integrieren kann. Neben mandantenspezifischen Bibliotheken sollen auch alle weiteren Open-Source-Bibliotheken von CocoaPods verwaltet werden, sodass zuerst die Netzwerk-Bibliothek aus der Infrastruktur entfernt wurde. Weil CocoaPods die Verwendung von Workspaces erfordert und solche nicht als Unterprojekte hinzugefügt werden können, kann CocoaPods nur in Verbindung mit dem Hauptprojekt verwendet werden – das Podfile liegt dementsprechend im Wurzelverzeichnis des gesamten Projekts. Wünschenswert ist es aber, dass jedes Modul seine Abhängigkeiten selbst angeben kann. Dafür wurden die bereits vorhandenen „ModuleDescription"-PLIST-Dateien der Module genutzt. Das im Rahmen dieser Arbeit entwickelte *combinepods*-Tool[117] findet über die zentrale „ActiveModules"-PLIST die aktuell eingebundenen Module, durchsucht deren Modul-Beschreibungen nach Pods, und fügt alle gefundenen Pods zu einem globalen Podfile zusammen. Anschließend kann CocoaPods die Pod-Abhängigkeiten auflösen und zum Projekt hinzufügen. Diese Abfolge muss nach jedem Checkout des Projekts und nach jeder Änderung der modulintern benötigten Pods ausgeführt werden und wurde daher in ein Skript ausgelagert, das über das „RunPodsUpdate"-Target gestartet werden kann. Voraussetzung dafür ist, dass CocoaPods vorher lokal installiert wurde, u. U. muss anschließend der Pfad im Skript angepasst werden[118]. Das combinepods-Tool ist in kompilierter Form im Projekt enthalten und muss nicht separat installiert werden. Zum Erzwingen des erstmaligen Ladens der Pods wird der gesamte Pods-Ordner sowie das Podfile vom Git-Repository ignoriert. Ein Versuch, das „RunPodsUpdate"-Target bei jedem Build-Vorgang des Projekts automatisch auszuführen scheiterte daran, dass die Neuerstellung des Pods-Projekts ein erneutes Laden des Workspaces in Xcode erzwingt[119].

Auch diese Restrukturierungsmaßnahme erfordert einige **Code- und Konfigurationsanpassungen**. Um die von CocoaPods exportierten Header zu finden, wurden in den Modulen die Imports der entsprechenden Header um einen `../Pods`-Präfix erweitert. Eleganter wäre es, die Export-Pfade im Pods-Projekt anzupassen, aber diese Änderungen würden nicht dauerhaft bestehen, weil das Pods-Projekt bei jedem Durchlauf des Tools neu erzeugt und alle Änderungen verworfen werden. Zusätzlich muss die Build-Reihenfolge explizit vorgegeben werden, weil das Pods-Projekt zwar stets vor dem Superprojekt, aber möglicherweise nach den Modulen gebaut wird.

[116] `http://stig.github.com/json-framework/`
[117] `https://bitbucket.org/shagedorn/combinepods/`
[118] Skripte, die von Xcode gestartet werden, haben keinen Zugriff auf die üblichen `$PATH`-Angaben
[119] Die Meldung in Xcode, dass das Projekt verändert wurde, sollte mit „Revert" beantwortet werden

Im Build-Schema des Superprojekts wurde deshalb das Pods-Produkt hinzugefügt und an erster Stelle platziert.

Die Umstellung auf JSON im Auftrags-Modul war anschließend ohne Weiteres möglich. Dazu wurden folgende Schritte durchgeführt:

1. Wechsel auf den `t3`-Branch im Superprojekt und im Auftrags-Modul
2. Hinzufügen des „SBJson"-Eintrags in der „ModuleDescription"-PLIST des Auftrags-Moduls
3. Ausführen des „RunPodsUpdate"-Targets
4. Hochladen der Auftragsdatendatei im JSON-Format und Aktualisierung der Angaben in der „OrderServerInfo"-PLIST
5. Code-Anpassung: Die Serverantwort wird nicht mehr als PLIST behandelt, sondern die *SBJson*-Bibliothek wird zum Parsen verwendet

Die Änderungen müssen zuerst im Modul committet und in dessen Remote Repository geladen werden, bevor im Superprojekt ein Commit mit der aktualisierten Modul-Referenz stattfinden kann. Um einen Branch der Anwendung auszuchecken, müssen weiterhin die Submodules aktualisiert werden, zudem muss das „RunPodsUpdate"-Target ausgeführt und statt der Projektdatei die Workspacedatei geöffnet werden.

Tags: `DEV-ERP-APP-PODS`, `T1-ERP-APP-PODS`, `T2-ERP-APP-PODS`, `T3-ERP-APP-JSON`

Hinweis: Mit den Snapshots der angegebenen Tags sind keine Archive-Builds möglich. Dieses Problem wurde im nächsten Schritt behoben, indem die Pods-Konfigurationsdatei in allen Modulen referenziert wird. Zudem wird von allen Modul-Targets die Build-Einstellung `PODS_ROOT` mit `../Pods` überschrieben, sodass die Imports im Folgenden ohne Präfix auskommen.

Core Data Reservefelder

Die geradlinige Umsetzung der Anforderung AF.4 würde eine Erweiterung des Datenmodells im `t3`-Branch vorsehen. Nach einer Vielzahl solcher Änderungen würden die Datenmodelle der verschiedenen Branches stark divergieren und Änderungen auf dem generellen Entwicklungszweig ließen sich nur mit Mühe von den Mandantenbranches übernehmen. Die Idee hinter den in Abschnitt 4.4 eingeführten **Reservefeldern** ist es, das allgemeine Datenmodell für alle Mandanten zu verwenden, solange in den einzelnen Branches nur geringfügige Änderungen stattfinden sollen.

Die Implementierung konnte ohne nennenswerte Probleme an der prototypischen Anwendung durchgeführt werden. Im generellen Entwicklungszweig wurde das `stringField1`-Attribut zur Auftrags-Entität hinzugefügt, ohne in der Model-Klasse ein entsprechendes Interface zu deklarieren. Lediglich im `t3`-Branch wurde die Auftrags-Klasse um eine `orderDate`-Property ergänzt, deren Getter und Setter per KVC auf das `stringField1`-Attribut zugreifen. Das Feld kann auf

gleiche Weise in anderen Branches mit anderen Properties verbunden werden. Um die Funktionsfähigkeit des Ansatzes zu zeigen, wurden Datums-Einträge zur JSON-Auftragsdatei auf dem Server hinzugefügt. Diese Einträge werden beim Datenladen übernommen und in der Detailansicht der Aufträge unterhalb des Kundennamens angezeigt.

Tag: `T3-ERP-APP-DATE`

5.3.5 Weiterentwicklung der Module

> *„AF.5: Nachdem alle Anpassungen durchgeführt worden sind, wurden die Infrastruktur und die Artikelübersicht weiterentwickelt – die Veränderungen sollen allen Mandanten zur Verfügung gestellt werden"*

Die weiteren Anforderungen können **ohne strukturelle Anpassungen** implementiert werden und sollen die Vor- und Nachteile der bisherigen Änderungen unterstreichen. Die in diesem Schritt durchgeführten Anpassungen betreffen die Programmoberfläche des Startbildschirms und der Artikelübersicht. In der Modulübersicht wurden die Rahmen um die einzelnen Modulbilder verändert, während die Titelleiste in der Artikelübersicht nun den aktuell ausgewählten Artikel reflektiert. Obwohl beide Änderungen in verschiedenen Projekten und Repositories durchgeführt werden, erlaubt die Einbindung der Module durch Xcode Subprojekte eine integrierte Sicht auf das Gesamtprojekt. Die Veränderungen an den Subprojekten sind lokal sofort wirksam, zum Testen genügt die Ausführung des Superprojekts. Die Anpassungen an den Subprojekten können jederzeit im jeweiligen Repository committet und auch an die Remotes übertragen werden, ohne jedoch im Hauptrepository[120] sichtbar zu werden. Dadurch können Code-Zwischenstände ohne Rücksicht auf das Gesamtprojekt in den Modulen getestet und ins Repository eingebracht werden, denn im Gesamtprojekt sind nicht die Branches, sondern feste Commits der Module referenziert. Änderungen in den Modulen müssen erst bewusst ins Superprojekt eingebracht und dort committet werden.

Das Artikel-Modul besitzt keine mandantenspezifischen Anpassungen, daher greifen alle Branches des Hauptrepositories auf den `general_dev`-Branch des Moduls zu. Um die Weiterentwicklung des Artikel-Moduls allen Mandanten zur Verfügung zu stellen, müssen alle Branches des Hauptrepositories nacheinander ausgecheckt werden. Nach dem Checkout eines Hauptbranches muss der neuste Stand des Artikel-Submodules ausgecheckt werden. Dieser Stand wird in Form des letzten Commits im Hauptrepository festgehalten und kann nun dort committet werden. Im Gegensatz zur Arbeit mit einem Repository für alle Module ist **kein Merge-Vorgang** erforderlich und die Moduländerungen können für sich committet werden, ohne das Gesamtprojekt zu beeinflussen. Dafür sind mehr Schritte notwendig, bis die Änderungen bei einem Checkout des Hauptrepositories sichtbar sind.

Im Infrastruktur-Modul existiert ein Mandantenbranch für T2, der die deutsche Lokalisierung enthält. Die Änderungen im Rahmen von AF.5 müssen modulintern vom `t2`-Branch übernommen werden. Ein Merge-Vorgang ist nur innerhalb des Modul-Repositories nötig, die Vorgehensweise zur Einbindung der Änderungen im Hauptrepository ist identisch zur Vorgehensweise beim

[120] Ausgenommen ist die lokale Working Copy

Artikel-Modul und verlangt keinen Merge-Vorgang. Wichtig ist, dass im `t2`-Branch des Hauptprojekts auch der `t2`-Branch des Infrastruktur-Moduls ausgecheckt wird. Das Git-Werkzeug SourceTree erleichtert die Arbeit mit Git Submodules, weil es versucht, die Commits der Submodules einem Branch zuzuordnen und dessen Namen anzeigt. Dennoch muss nach dem Checkout eines Submodules darauf geachtet werden, einen lokalen Branch anzulegen, um nicht auf einem Detached HEAD[121] zu arbeiten.

Tags: `DEV-ERP-APP-UPDATES`, `T1-ERP-APP-UPDATES`, `T2-ERP-APP-UPDATES`, `T3-ERP-APP-UPDATES`

> „*AF.6: Auch das Auftrags-Modul erhält generelle Anpassungen, die allen Mandanten zur Verfügung stehen sollen*"

In der Detailansicht der Aufträge wird durch die Anpassung nach AF.6 ein QR-Code des Auftrags angezeigt, der auf dem Gerät mithilfe der *iOS-QR-Code-Encoder*-Bibliothek[122] generiert wird. Die Nutzung von CocoaPods erleichtert diese Aufgabe ungemein, da die Einbindung der Bibliothek lediglich das Hinzufügen eines Eintrags in der „ModuleDescription"-PLIST erfordert. Zudem bleiben alle Projektdateien der Module unverändert, was einen **konfliktfreien Merge-Vorgang** innerhalb des Auftrags-Moduls begünstigt. Die Änderungen am Pods-Projekt sind für den Merge-Vorgang ohne Bedeutung, weil der Ordner dieses Projekts nicht unter Versionskontrolle steht und alle Änderungen lokal durch das „RunPodsUpdate"-Target reproduziert werden. Hinzu kommt, dass nur die Pods der tatsächlich eingebundenen Module geladen werden. Um einer möglichen späteren Benutzung der Artikel- und Auftragsmodule durch T2 nicht im Wege zu stehen, sind diese Subprojekte auch im `t2`-Branch vorhanden, es fehlt lediglich deren Einbindung beim Link-Vorgang sowie die Referenzierung in der „ActiveModules"-Konfigurationsdatei. Durch die Verwendung von combinepods wird die QR-Code-Bibliothek für T2 nicht von CocoaPods berücksichtigt, weil sie nur vom Auftrags-Modul verwendet wird und dieses Modul nicht eingebunden ist.

Nach der Umsetzung der Anforderungen wurden alle Klassen dokumentiert. Mithilfe des Werkzeugs *appledoc*[123] kann die **Dokumentation** entweder über die Kommandozeile mit dem Befehl `appledoc .`[124] oder über das „UpdateAppleDoc"-Target[125] generiert werden. Alle Parameter des appledoc-Werkzeugs sind in der „AppleDocSettings"-PLIST festgehalten, das Werkzeug muss separat installiert werden und die generierte Dokumentation wird von Git ignoriert. Nach der lokalen Generierung befindet sich eine HTML-Variante im `Documentation/html`-Verzeichnis, zudem wird die Dokumentation automatisch in Xcode integriert. Leider verzichtet appledoc auf die Angabe von Verzeichnispfaden, sodass die Modulzugehörigkeit von Klassen nur aus dem Dateisystem und Xcode, nicht aber aus der generierten Dokumentation ersichtlich ist. Die Dokumentation im HTML-Format ist in Anhang B.4 enthalten.

Tags: `DEV-ERP-APP-QR`, `T1-ERP-APP-QR`, `T2-ERP-APP-QR`, `T3-ERP-APP-QR`

[121] `http://sitaramc.github.com/gcs/index.html`
[122] `https://github.com/akopanev/iOS-QR-Code-Encoder/`
[123] `http://gentlebytes.com/appledoc/`
[124] Vorher muss mit dem `cd`-Befehl ins Wurzelverzeichnis des Projekts gewechselt werden
[125] Dies erfordert u. U. eine Anpassung des Pfads im Skript, sollte aber bei der Verwendung des Standard-Installationsskripts von appledoc sofort funktionieren

5.4 ZUSAMMENFASSUNG

Aus der Abfolge aller durchgeführten Anpassungsmaßnahmen lässt sich ein **Gesamtkonzept** für eine Modularisierung im Hinblick auf Mehrmandantenfähigkeit von komplexen iOS-Apps ableiten. Dabei haben sich fünf wesentliche Schritte herauskristallisiert:

1. Anlegen von **Targets** (Xcode) und **Branches** (Git) für jeden Mandanten

Targets und Branches ermöglichen den schnellen Wechsel zwischen Mandanten für den Entwickler. Um die Codebasis für oberflächliche Anpassungen unverändert zu lassen, sollten mandantenspezifische Informationen nach Möglichkeit in Konfigurationsdateien ausgelagert werden.

2. Durchführung einer **Abhängigkeitsanalyse**
3. Modularisierung durch **Xcode Subprojekte**

Schritt 2 ist nur notwendig, wenn eine bestehende Anwendung umstrukturiert werden soll. Andernfalls sollten Modulstrukturen bereits beim initialen Designentwurf berücksichtigt werden. Um logische Modulstrukturen umzusetzen, bieten Xcode Subprojekte eine gangbare, wenn auch aufwändige Möglichkeit an. Zur Interaktion und losen Kopplung der Module wurden in dieser Arbeit das Module-Protokoll und dynamisches Klassenladen genutzt. Auch nach diesem Schritt sollten mandantenspezifische Anpassungen idealerweise aus Konfigurationsdateien ausgelesen werden.

4. Auslagerung der Module mit **Git Submodules**

Mit getrennten Repositories der Module ist es möglich, für Teile einer Anwendung mandantenspezifische Branches anzulegen, während andere Bereiche unmodifiziert benutzt werden. Aus dem Hauptrepository ist dabei leicht ersichtlich, welche Module mandantenspezifisch angepasst worden sind. Änderungen an Modulen ohne Branches können ohne Merge-Vorgänge in alle mandantenspezifischen Versionen der Hauptanwendung übernommen werden.

5. Verwaltung externer Bibliotheken mit **CocoaPods**

Die manuelle Einbindung von quelloffenen Bibliotheken ist problembehaftet, weil verschiedene Mandantenbranches mitunter verschiedene Bibliotheken benötigen, die Einbindung dieser Bibliotheken aber zahlreiche Änderungen an den Projektdateien erzwingt. Änderungen an Projektdateien lösen häufig Merge-Konflikte aus, die oft nur durch eine wiederholte Durchführung der Änderungen auflösbar sind. Diese Komplexität wird von CocoaPods in ein separates Projekt ausgelagert, das automatisch generiert werden kann und somit nicht unter Versionskontrolle stehen muss.

In allen Schritten wurde auf die in Kapitel 4 vorgestellten **Teilkonzepte** zurückgegriffen. Neben den fünf genannten wesentlichen Schritten kamen weitere Teilkonzepte in der prototypischen Arbeit zum Einsatz. Die Werkzeuge stringsbuilder und combinepods führen Modulinfos zu globalen Dateien der Hauptanwendung zusammen, um sie zur Laufzeit einfacher nutzbar zu machen und ungenutzte Informationen auszusparen. Durch die Verwendung von Auto Layout passt sich die Programmoberfläche dynamischen Informationen an und Reservefelder erlauben die gemeinsame Nutzung eines Datenmodells von allen Mandanten, solange die spezifischen Anpassungswünsche nur einen geringen Umfang besitzen.

Jeder Schritt bedeutet einen erheblichen, allerdings meist einmaligen **Mehraufwand** im Vergleich zu einer herkömmlichen Projektstruktur. Weil die Entwicklungsumgebung Xcode nur eine geringe native Unterstützung für modulare Projekte bereitstellt, ist das Gesamtkonzept eine Kombination aus Skripten, Projekteinstellungen und einer speziellen Konfiguration des SCM-Repositories. Eine Evaluation des Konzepts findet in Kapitel 6 statt.

6 EVALUATION

In der Evaluation werden zunächst die in Abschnitt 5.4 zusammengefassten wesentliche Schritte für sich betrachtet. In erster Linie soll dabei jeweils der **Implementierungsaufwand** dem **Nutzen** für die Mehrmandantenfähigkeit gegenübergestellt werden. Sofern möglich, sollen auch allgemeine Vor- und Nachteile aus softwaretechnischer Sicht kurz benannt und mit statistischen Werten unterlegt werden. Die Evaluation der Abschnitte wird knapp gehalten, weil alle Schritte nicht nur beschrieben und implementiert, sondern im Rahmes des Prototyps und einem Beispielszenario mit mehreren Mandanten getestet worden ist. Konkrete Implementierungsprobleme wurden dabei bereits in Kapitel 5 benannt.

Abschließend wird in Abschnitt 6.2 die **Gesamtheit aller Schritte** evaluiert. Dabei soll u. a. betrachtet werden, wieviel Aufwand zur Erstellung einer neuen Variante einer modularisierten App nötig ist. Zudem soll erörtert werden, wie die Chancen einer Umsetzung des Konzepts auf eine reale Anwendung stehen.

6.1 EVALUATION DER TEILKONZEPTE

Anlegen von Targets (Xcode) und Branches (Git) für jeden Mandanten

Das Anlegen von Targets kann in wenigen Schritten direkt in Xcode erledigt werden. Targets enthalten alle grundlegenden Informationen einer Mandantenversion, von den benötigten Ressourcen bis hin zu den verlinkten Modulen. Die Verwendung von mehreren Targets bringt keinerlei Codeanpassungen oder Strukturänderungen mit sich.

Das Erstellen von Git-Branches gehört zu den Routineaufgaben der Softwareentwicklung und ist in wenigen Sekunden zu bewältigen. Branches ermöglichen es, mandantenspezifische Dateien auszutauschen, ohne dass dafür Änderungen in der Entwicklungsumgebung vorgenommen werden müssen. Auch das Mergen von Branches ist eine Routineaufgabe, allerdings sind Konflikte abhängig vom Dateityp unterschiedlich aufwändig aufzulösen.

Problematisch ist, dass die Target-Informationen im Normalfall direkt in den **Projektdateien** gespeichert werden. Durch die Entscheidung, die einzelnen Targets ausschließlich im zugehörigen Branch sichtbar zu machen, sind die Projektdateien aller Mandanten zwangsweise verschieden und allgemeine Änderungen an den Projektdateien führen regelmäßig zu konfliktbehafteten Merge-Vorgängen. Um dieses Problem zu umgehen, kann versucht werden, sämtliche Target-Informationen in zusätzliche Konfigurationsdateien auszulagern. Die Anwendung verfügt dann nur über ein einziges Target, das sich einer über den Git-Branch austauschbaren Konfigurationsdatei anpasst. Sowohl Targets als auch Branches können direkt in Xcode verwaltet werden.

Durchführung einer Abhängigkeitsanalyse

Die Analyse bestehender Abhängigkeiten hat sich im Laufe dieser Arbeit immer wieder als problematisch erwiesen. Mit objc_dep konnte ein Werkzeug gefunden werden, das anhand von `#import`-Anweisungen einen **Abhängigkeitsgraph** erstellt. Für den Prototyp dieser Arbeit konnten aus dem Graph wertvolle Erkenntnisse gewonnen werden, allerdings ist die Verwendbarkeit bei großen Projekten stark eingeschränkt, wie in Anlage A.2 gezeigt.

Im Zuge der Evaluation wurde im Nachhinein noch ein Graph der vollständig modularisierten Anwendung generiert. Der in Anlage A.5 enthaltene Graph wirkt besonders überschaubar, weil jegliche Verbindungen zwischen den Modulen fehlen. Als Grund konnte die Verwendung von Spitzklammern für die modulübergreifenden Imports ausgemacht werden, die bewusst zur Unterscheidung von modulinternen Imports genutzt worden sind. Darüber hinaus macht das Werkzeug keinen Unterschied zwischen den Klassennamen selbst und den vorangestellten Pfadangaben. Aus diesem Grund werden modulinterne Imports (z. B. `#import "ServerEntity"`) und modulübergreifende Imports (z. B. `#import <InfrastructureModule/ServerEntity.h>`) nicht der selben Klasse zugeordnet. Um einen realistischen Graph zu erzeugen, wurden die Imports des Prototyps entsprechend angepasst, das Ergebnis der anschließenden Graph-Generierung ist in Anlage A.6 zu finden. Die Modulstrukturen lassen sich auch hier nur mit Mühe herausarbeiten. Zur Verbesserung der gegenwärtigen Lage kann das Werkzeug so weiterentwickelt werden, dass es die Pfade der Imports oder die Ordner im Dateisystem für eine Gruppierung nach Modulen nutzt. Eine Integration in Xcode ist derzeit nicht möglich, wäre allerdings wünschenswert.

Modularisierung durch Xcode Subprojekte

Der mit Abstand **aufwändigste Schritt** war die Auslagerung von Anwendungsbereichen in Subprojekte. Stark vereinfacht wurde dadurch in erster Linie das Ein- und Ausbauen vorhandener Module. Zudem fallen spätestens beim Bauen der Anwendung unvorhergesehene, modulübergreifende Abhängigkeiten auf. Weil die Werkzeugunterstützung bei der theoretischen Analyse ungenügend ist, kommt diesem Schritt eine umso größere Bedeutung zu.

Um die Anwendung an die veränderte Struktur anzupassen, ist die **Codebasis**[126] in diesem Schritt von knapp 2600 auf über 2800 Zeilen angewachsen. Im Fall des vorliegenden Prototyps sind das

[126] Ohne Bibliotheken von Dritten

beinahe 9 % Zuwachs, der prozentuale Wert dürfte aber bei großen Anwendungen deutlich geringer ausfallen. Positiv zu bewerten ist, dass das Ausbauen eines Moduls wenig Arbeit bedeutet, aber dennoch zur Folge hat, dass die Anwendung spürbar kleiner wird, weil sämtlicher Code und alle Ressourcen des Moduls vollständig ausgebaut werden. Die Beispielanwendung für T2, die nur das Kunden-Modul enthält, ist beispielsweise nur 600 KB groß, während die Varianten mit allen Modulen über 800 KB groß sind. Die erzielte Differenz ist dabei direkt abhängig von der Größe der ausgebauten Module, insbesondere deren benötigter Ressourcen. Das App-Archiv der modularisierten Anwendung unterscheidet sich in seiner Dateigröße nur unwesentlich von der Ausgangsanwendung. Durch die klare Definition von Abhängigkeiten ist nach einem erfolgreichen Build-Vorgang garantiert, dass alle anderen Module fehlerfrei laufen.

Festzustellen bleibt, dass Xcode Subprojekte für eine derartige Modularisierung vermutlich nicht vorgesehen sind. Für diese Einschätzung sprechen die zahlreichen, manuell durchzuführenden Anpassungen von Compiler- und Linker-Flags, zusätzliche Bundle-Targets für die Ressourcen und daraus resultierende Code-Änderungen für den Ressourcenzugriff. Weil doppelte Verlinkungen statischer Bibliotheken verhindert werden müssen, sind mitunter nicht alle Abhängigkeiten aus den Linker-Einstellungen des Moduls selbst ersichtlich. Um eine saubere Trennung von Anwendungsbereichen zu erreichen, konnte allerdings auch keine alternative Technologie oder Struktur gefunden werden und die Unterstützung durch Xcode erlaubt eine integrierte Sicht auf die Gesamtanwendung samt aller Module. Um die genannten Vorteile nutzen zu können, muss der einmalige Mehraufwand in Kauf genommen werden. Die Umsetzung des nächsten Schritts, der Verwendung von Git Submodules, ist ohne Subprojekte nicht sinnvoll.

Auslagerung der Module mit Git Submodules

Durch die Verwendung von Git Submodules wird deutlich sichtbar, welche Module für welche Mandanten angepasst worden sind. Dies reduziert sowohl die Anzahl als auch die Komplexität der Merge-Vorgänge.

Im Folgenden soll der **Aufwand** zum Einbringen einer Modulanpassung in alle Mandantenbranches **mit und ohne Git Submodules** verglichen werden. Für den Fall ohne Git Submodules gilt die Annahme, dass neben dem generellen Entwicklungszweig noch drei Mandantenbranches vorhanden sind. Wurde eine Änderung im Entwicklungszweig durchgeführt, müssen alle Mandantenbranches nacheinander ausgecheckt werden. Nach dem Checkout werden die Änderungen aus dem Entwicklungszweig per Merge übernommen, es sind also genau drei Merge-Vorgänge notwendig. Dabei ist es unwesentlich, in welchem Bereich der Anwendung die Anpassung vorgenommen wurde.

Werden Git Submodules verwendet, ist der Aufwand abhängig davon, wieviele Branches das von den Anpassungen betroffene Submodule besitzt. Ist nur der generelle Entwicklungszweig vorhanden, ist im Modul kein Merge-Vorgang notwendig. Besitzt dagegen jeder Mandant seinen eigenen Branch des Moduls, sind wie bei der Arbeit ohne Git Submodules drei modulinterne Merge-Vorgänge notwendig, jeweils vom generellen Entwicklungszweig in den Mandantenbranch. Im Hauptrepository sind Merges in keinem Fall notwendig, es muss nur in jedem Branch die gewünschte Version des geänderten Moduls ausgecheckt werden. Die Anzahl der

Merge-Vorgänge ist also im schlechtesten Fall so hoch wie bei der Arbeit ohne Git Submodules, im besten Fall sind keine Merge-Vorgänge notwendig. Mit dem Checkout der Module im Hauptrepository kommt ein zusätzlicher Arbeitsschritt hinzu, der im Gegensatz zu Merge-Vorgängen jedoch keine Probleme mit sich bringt. Weil die Einrichtung von Git Submodules einfach ist und keine Codeänderungen nötig sind, ist die Verwendung im Allgemeinen zu empfehlen. Allerdings bietet Xcode keinerlei Unterstützung von Submodules an, sodass diese in der Kommandozeile oder einer separaten Anwendung verwaltet werden müssen.

Verwaltung externer Bibliotheken mit CocoaPods

Die Einbindung von CocoaPods in ein Projekt mit Unterprojekten war nicht trivial. Sind die einzelnen Schritte aber einmal bekannt, können diese in **kurzer Zeit** durchgeführt werden. Der Nutzen ist bereits bei der Verwendung weniger Bibliotheken hoch. So erübrigt sich die Frage, ob Bibliotheken der Infrastruktur oder dem benutzenden Modul angehören sollten. In Zusammenarbeit mit dem combinepods-Werkzeug bietet CocoaPods eine Art Zwischenlösung: Die Bibliotheken sind zentral gelagert und damit für alle Module erreichbar, die tatsächlich eingebundenen Bibliotheken werden dennoch direkt von der Gesamtheit aller aktiven Module bestimmt. Die Bibliotheken müssen weder im Repository noch im eigenen Xcode-Projekt verwaltet werden.

Komplizierter wird die Verwendung von CocoaPods, wenn die genutzten Bibliotheken an die eigene Anwendung angepasst werden müssen. Über eine Konfiguration des Werkzeugs kann zwar die Verbindung zum Original-Repository gehalten werden, doch dort fehlt im Allgemeinen ein Schreibrecht zum unmittelbaren Einbringen von Änderungen. Die Verwaltung der Änderungen im eigenen Repository schränkt das Werkzeug erheblich ein, weil diese Änderungen bei einem erneuten Durchlauf von CocoaPods überschrieben werden würden. Eine mögliche Lösung ist das Abzweigen (engl.: Fork)[127] des öffentlichen Repositories in ein Repository mit Schreibrecht. Für dieses Repository muss eine Bibliotheks-Spezifikation („Pod Spec“) erstellt und CocoaPods zugänglich gemacht werden. Der Aufwand muss hier im Einzelfall abgewägt werden. Bei unveränderten Bibliotheken ist die Verwendung von CocoaPods generell zu empfehlen, zumal sich das Werkzeug durch die Kommandozeilenschnittstelle gut in Xcode integrieren lässt.

6.2 EVALUATION DES GESAMTKONZEPTS

Die Gesamtheit aller durchgeführten Schritte hat die Umsetzung der in Abschnitt 5.2 gestellten **Anforderungen** vollumfassend ermöglicht. Zu evaluieren ist jedoch in erster Linie, inwieweit der einmalige Mehraufwand der Restrukturierungen durch eine häufige Wiederholung solcher und ähnlicher Anpassungswünsche aufgewogen werden kann. Für die einzelnen Teilschritte wurde dies im vorangegangenen Abschnitt bereits getan.

Um auf Basis der modularisierten Anwendung eine **neue Mandantenversion** zu erstellen, sind nur wenige Schritte nötig. Zuerst sollte ein neuer Branch im Hauptrepository angelegt werden. Als Ausgangsbranch kann dabei der generelle Entwicklungszweig genutzt werden – sollte

[127] `https://help.github.com/articles/fork-a-repo/`

die neue Version einer anderen Mandantenversion ähnlich sein, kann auch deren Branch als Ausgangspunkt genutzt werden. Im nächsten Schritt müssen die Target-Informationen ausgefüllt bzw. geändert werden und die grafischen Ressourcen im Dateisystem ausgetauscht werden. Schließlich können alle Konfigurationsdateien des „Tentant"-Verzeichnisses im Hauptprojekt mit den mandantenspezifischen Informationen befüllt werden. Wenn Änderungen an der „ActiveModules"-PLIST durchgeführt werden, müssen u. U. auch die Verlinkungen zwischen den Subprojekten angepasst werden. Von jedem Modul kann eine andere Version gewählt werden. Auch hier wird im Zweifel auf den generellen Entwicklungszweig zurückgegriffen, es kann aber auch eine mandantenspezifische Implementierung abgezweigt werden. Vor dem ersten Build-Vorgang muss das „RunPodsUpdate"-Target ausgeführt werden.

Sofern keine neuen Funktionen erforderlich sind, kann eine neue Mandantenversion erstellt werden, ohne dafür den Quellcode zu verändern. Die Grundlage dafür bieten Konfigurationsdateien, die teilweise von Werkzeugen wie stringsbuilder und combinepods genutzt werden, um bestimmte Ressourcen mandantenspezifisch zu generieren.

Um die Anwendung weiterzuentwickeln und die Abhängigkeiten zwischen den Modulen zu erkennen, ist allerdings eine **genaue Kenntnis des** hier vorgestellten **Konzepts** von Nöten. Bis auf den ersten Schritt, Targets und Branches, sind die Teilkonzepte in der alltäglichen iOS-Entwicklung wenig verbreitet und bedürfen einiger Einarbeitungszeit. Jedes der Teilkonzepte fügt der Struktur der Anwendung oder des Repositories Komplexität hinzu. Weil es für die Modularisierung von iOS-Apps kein standardisiertes Konzept gibt, kann es beim Zusammenspiel mit anderen Werkzeugen zu Konflikten kommen. Im Fall von CocoaPods konnten diese Konflikte gelöst werden, während die Kompatibilität zu dem in Abschnitt 4.3.3 vorgestellten Maven-Plugin nicht hergestellt werden konnte. Positiv ist hervorzuheben, dass drei der fünf Teilkonzepte direkt in Xcode integriert werden können. Die beiden verbliebenen Teilkonzepte greifen auf separate Werkzeuge zurück, die jedoch in keinerlei Widerspruch zur Nutzung mit Xcode stehen und die Arbeit in Xcode ergänzen.

Das Gesamtkonzept hilft, Module innerhalb einer Anwendung bzw. verschiedenen Versionen einer Anwendung wiederzuverwenden, anzupassen und auszutauschen. Im Gegensatz zu Modulen bewährter Java-Technologien wie OSGi sind Module der vorliegenden Arbeit jedoch keine völlig unabhängigen Software-Einheiten, sondern abhängig von der Struktur der Gesamtanwendung, eigenen Protokollen und Konventionen. Für die anwendungs- oder unternehmensübergreifende Wiederverwendung sind sie daher nur bedingt geeignet. Zwar sind Xcode Subprojekte an sich universell verwendbar, die Nutzung und Einbindung der kompilierten Produkte in anderen Anwendungsteilen unterliegt jedoch keiner standardisierten Richtlinie.

Die Umsetzung des Gesamtkonzepts ist auch an einer **komplexen Anwendung** wie der alexa-MRS-App der SALT Solutions GmbH prinzipiell denkbar, erfordert jedoch einen hohen Zeitaufwand. Zu erwarten ist, dass aufgrund der schlechten Werkzeugunterstützung besonders die Abhängigkeitsanalyse problematisch ist, weil die in dieser Arbeit gewonnenen Erkenntnisse in diesem Bereich nicht zwangsläufig auf andere Anwendungen übertragbar sind. Die übrigen Schritte sollten dagegen durch die Vorarbeit an dem vorliegenden Prototyp erleichtert werden und wenige unerwartete Probleme mit sich bringen.

7 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit ist untersucht worden, inwiefern allgemeine Konzepte und vorhandene Technologien für die Mehrmandantenfähigkeit von mobilen Anwendungen auf der iOS-Plattform verwendet werden können. Dabei wurden im 2. Kapitel zunächst bestehende Konzepte und Technologien zusammengetragen. Es hat sich herausgestellt, dass der Großteil davon an die Programmiersprache Java gebunden ist und daher für die Verwendung unter iOS nicht geeignet ist. Lediglich programmiersprachenunabhängige Werkzeuge wie Git haben eine Anwendbarkeit im Rahmen der iOS-App-Entwicklung erhoffen lassen. Als primäres Grundkonzept für mehrmandantenfähige Anwendungen wurde eine **modulare Architektur** ausgemacht.

Um die **Besonderheiten der iOS-Plattform** herauszuarbeiten, wurden im 3. Kapitel in erster Linie Objective-C-Sprachelemente beschrieben, die in anderen Programmiersprachen nicht vorhanden sind. Im Fokus stand dabei die Laufzeitdynamik der Sprache, die eine lose Kopplung von Anwendungsteilen vereinfacht. Im darauffolgenden Kapitel wurden mithilfe der Laufzeitdynamik, Teilprojekten und Werkzeugen von Dritten Lösungsansätze für typische Probleme mehrmandantenfähiger Anwendungen entwickelt. Schwierig war dies besonders im Bereich der Strukturanalyse, die in Vorbereitung auf die geplante Modularisierung notwendig ist. Während für Java-basierte Anwendungen umfangreiche, meist kommerzielle Anwendungen zum Refactoring und Round-Trip Engineering verfügbar sind, konnten für Objective-C-Anwendungen nur Skripte zur Generierung von einfachen, oft unübersichtlichen Graphen gefunden werden.

Die Ansätze wurden in Kapitel 5 zu einem **Gesamtkonzept** zusammengefügt. Um die Sinnhaftigkeit und Umsetzbarkeit zu demonstrieren, wurde gleichzeitig ein Prototyp entwickelt, der in mehreren Schritten von einer monolithischen zu einer modularen Anwendung umgebaut wurde. Neben Xcode Subprojekten hat sich dabei v. a. Git als unverzichtbare Technologie erwiesen, um verschiedene Versionen einer Anwendung zu verwalten und bei Bedarf zu synchronisieren. Im Rahmen der prototypischen Arbeit wurden zudem die beiden Werkzeuge stringsbuilder und combinepods entwickelt, die die Kompatibilität von vorhandenen Technologien und Werkzeugen mit

der neuen, modularen Projektstruktur herstellen. Sowohl das Projekt als auch die beiden Werkzeuge sind veröffentlicht worden[128].

Die Arbeit am **Prototyp** und die **Evaluation** in Kapitel 6 haben gezeigt, dass das Konzept die Entwicklung mehrmandantenfähiger Anwendungen tatsächlich begünstigt. An dieser Stelle soll noch ein **Ausblick** gegeben werden, durch welche Maßnahmen die Anwendung des Konzepts erleichtert werden kann. Da eine App nur dann modularisiert werden kann, wenn die gewollten und ungewollten Abhängigkeiten klar ersichtlich sind, wäre die Entwicklung geeigneter Werkzeuge wünschenswert. Mit überschaubarem Aufwand sollte es möglich sein, die bestehenden, einfachen Werkzeuge und Skripte so zu erweitern, dass die Ordnerstrukturen bei der grafischen Darstellung berücksichtigt werden. Langfristig sind jedoch Werkzeuge nötig, die über die rein grafische Darstellung hinausgehen und idealerweise in Xcode integriert sind.

Damit **Module** auch anwendungsübergreifend austauschbar sind, sollten für sie **einheitliche Strukturen** gelten. Module im Sinne von Xcode Subprojekten können dafür als Ausgangspunkt dienen. Um die zahlreichen Detailanpassungen der Linker-Einstellungen zu umgehen, kann ein Xcode Projekt-Template für Module erstellt werden. Die Auslieferung des Templates mit Xcode würde schnell für eine weite Verbreitung sorgen, was die Akzeptanz und Nützlichkeit solcher Module entsprechend fördern würde. Dies wäre in erster Linie ein Vorteil für modulare Entwicklung im Allgemeinen, weniger im Hinblick auf die Mehrmandantenfähigkeit.

Zur Überprüfung der **Praxistauglichkeit** des Konzepts sollte es auf eine reale, komplexe Anwendung angewandt werden. Um die Skalierbarkeit für sehr viele Mandanten zu gewährleisten, muss nach Möglichkeiten gesucht werden, Änderungen vollautomatisch in verschiedene Versionen der Anwendung einzubringen und diese Versionen automatisch zu bauen. Mit der Unterteilung der Anwendung in Subprojekte und Git Submodules ist dafür eine gute Basis vorhanden.

[128] `https://bitbucket.org/shagedorn/`

LITERATURVERZEICHNIS

[app12a] *App Store Review Guidelines*. Webseite. `https://developer.apple.com/appstore/resources/approval/guidelines.html`. Version: September 2012. – Letzter Abruf: 19. September 2012

[app12b] *iOS Human Interface Guidelines*. Webseite. `https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html`. Version: September 2012. – Letzter Abruf: 19. September 2012

[BC02] BAROWSKI, L.A. ; CROSS, II J.H.: Extraction and use of class dependency information for Java. In: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002. – ISSN 1095–1350, S. 309 – 315

[Bea12] BEARD, Patrick C.: *Modern Objective-C*. Video. `https://developer.apple.com/videos/wwdc/2012/`. Version: Juni 2012. – Letzter Abruf: 14. September 2012

[BY10] BUCK, Erik M. ; YACKTMAN, Donald A.: *Cocoa Design Patterns für Mac und iPhone*. 1st. mitp, 2010

[BZP+10] BEZEMER, C.-P. ; ZAIDMAN, A. ; PLATZBEECKER, B. ; HURKMANS, T. ; HART, A.: Enabling multi-tenancy: An industrial experience report. In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010. – ISSN 1063–6773, S. 1 –8

[Cam10] CAMPANU, C.: *Git Gains Advantage over Subversion*. Webseite. `http://news.softpedia.com/news/Git-Gaines-Advantage-Over-Subversion-136737.shtml`. Version: März 2010. – Letzter Abruf: 28. Oktober 2012

[CN91] COX, Brad J. ; NOVOBILSKI, Andrew: *Object-Oriented Programming – An Evolutionary Approach*. 2nd. Addison Wesley Publishing Company, 1991

[DYM+08] DIETRICH, Jens ; YAKOVLEV, Vyacheslav ; MCCARTIN, Catherine ; JENSON, Graham ; DUCHROW, Manfred: Cluster analysis of Java dependency graphs. In: *Proceedings of the 4th ACM symposium on Software visualization*. New York, NY, USA : ACM, 2008 (SoftVis '08). – ISBN 978–1–60558–112–5, 91–94

[ecl12] *Eclipse Community Survey 2012*. Webseite. `http://www.infoq.com/news/2012/06/eclipse-survey`. Version: Juni 2012. – Letzter Abruf: 28. Oktober 2012

[goo12] *Good Technology Device Activations Report | Q1 2012*. Webseite. `http://media.www1.good.com/documents/Good_Data_Q1_2012.pdf`. Version: April 2012. – Letzter Abruf: 14. September 2012

[GSH+07] GUO, Chang J. ; SUN, Wei ; HUANG, Ying ; WANG, Zhi H. ; GAO, Bo: A Framework for Native Multi-Tenancy Application Development and Management. In: *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, 2007, S. 551 –558

[HC11] HALVORSEN, Ole H. ; CLARKE, Douglas: *OS X and iOS Kernel Programming*. 1st. Berkely, CA, USA : Apress, 2011. – ISBN 1430235365, 9781430235361

[HLR08] HETTEL, Thomas ; LAWLEY, Michael J. ; RAYMOND, Kerry: Model Synchronisation: Definitions for Round-Trip Engineering. In: *ICMT2008 - International Conference on Model Transformation: Theory and Practice of Model Transformations*. Zürich, Schweiz : Springer, 2008, 31–45

[isu12] *Apple's Share of Media Tablet Market Hits More than One-Year High in Q2*. Webseite. `http://www.isuppli.com/Display-Materials-and-Systems/News/Pages/Apples-Share-of-Media-Tablet-Market-Hits-More-than-One-Year-High-in-Q2.aspx`. Version: August 2012. – Letzter Abruf: 14. September 2012

[KLM+97] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-Oriented Programming. In: *European Conference on Object-Oriented Programming (ECOOP)* (1997), Juni

[Kno12] KNOERNSCHILD, Kirk: *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Prentice Hall, 2012

[Lak05] LAKOS, John: *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., 2005

[MUTL09] MIETZNER, R. ; UNGER, T. ; TITZE, R. ; LEYMANN, F.: Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In: *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, 2009. – ISSN 1541–7719, S. 131 –140

[Rah12] RAHARDJA, Dave: *Building and Distributing Custom B2B Apps for iOS*. Video. `https://developer.apple.com/videos/wwdc/2012/`. Version: Juni 2012. – Letzter Abruf: 14. September 2012

[RAR07] RELLERMEYER, Jan S. ; ALONSO, Gustavo ; ROSCOE, Timothy: R-OSGi: distributed applications through software modularization. In: *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. New York, NY, USA : Springer-Verlag New York, Inc., 2007 (Middleware '07), 1–20

[See09] SEEBERGER, Heiko: *OSGi in kleinen Dosen – Bundles und Life Cycle*. Webseite. `http://it-republik.de/jaxenter/artikel/%20OSGi-in-kleinen-Dosen-%96-Bundles-und-Life-Cycle-2118.html`. Version: Februar 2009. – Letzter Abruf: 14. September 2012

[tio12] *TIOBE Software: Tiobe Index*. Webseite. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Version: November 2012. – Letzter Abruf: 22. November 2012

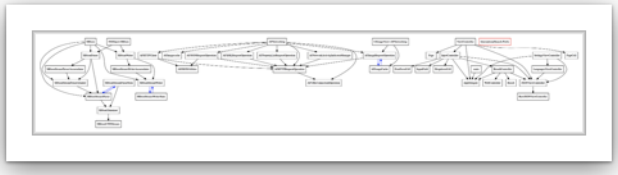
A DATEIEN

Die folgende Pfade beziehen sich auf die unter Anhang C eingereichte CD.

A.1 ABHÄNGIGKEITSGRAPH: KLEINES PROJEKT (40 KLASSEN)

Pfad: `Appendix/objc_dep_graph_int_search.pdf`

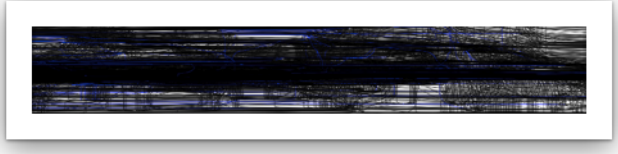
Vorschau:



A.2 ABHÄNGIGKEITSGRAPH: GROSSES PROJEKT (CA. 1500 KLASSEN)

Pfad: `Appendix/objc_dep_graph_alexa.pdf`

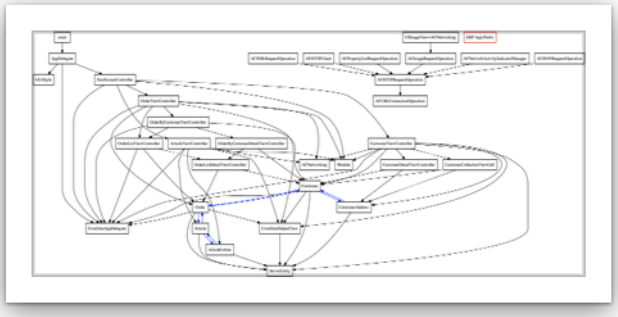
Vorschau:



A.3 ABHÄNGIGKEITSGRAPH: PROTOTYP (NICHT MODULARISIERT, CA. 30 KLASSEN)

Pfad: `Appendix/objc_dep_graph_prototype.pdf`

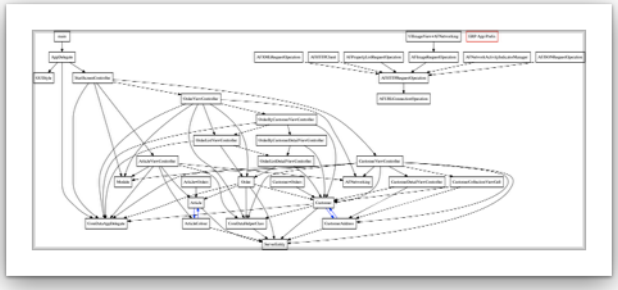
Vorschau:



A.4 ABHÄNGIGKEITSGRAPH: PROTOTYP (AZYKLISCHE MODEL-KLASSEN)

Pfad: `Appendix/objc_dep_graph_prototype_acyclic_model.pdf`

Vorschau:



A.5 ABHÄNGIGKEITSGRAPH: MODULAR ERP APP

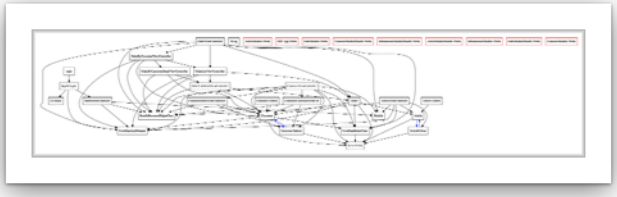
Pfad: `Appendix/objc_dep_graph_final.pdf`

Vorschau:

A.6 ABHÄNGIGKEITSGRAPH: MODULAR ERP APP (FLACHE IMPORTS)

Pfad: `Appendix/objc_dep_graph_final_refined.pdf`

Vorschau:

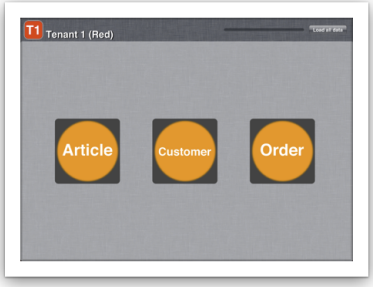


B PROTOTYP

B.1 AUSGANGSANWENDUNG (T1 ERP APP)

Pfad: `Prototype/T1-ERP-App/`
Öffentliches Repository: `https://bitbucket.org/shagedorn/t1-erp-app/`

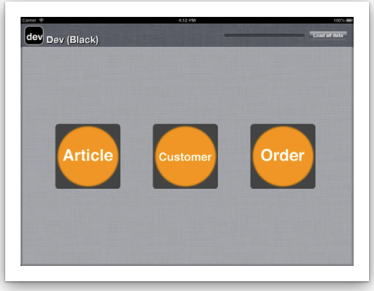
Vorschau:



B.2 MODULARISIERTE ANWENDUNG (MODULAR ERP APP)

Pfad: `Prototype/ERP-App/` (inkl. lokalem Git-Repository)
Öffentliches Repository: `https://bitbucket.org/shagedorn/modular-erp-app/`

Vorschau:



B.3 MODUL-REPOSITORIES

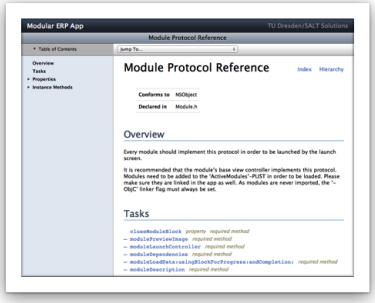
Pfad: `Prototype/Submodules/` (inkl. lokalem Git-Repository)
Öffentliche Repositories: `https://bitbucket.org/shagedorn/{ArticleModule, CustomerModule, InfrastructureModule, OrderModule}`

Vorschau:

B.4 API DOKUMENTATION

Pfad: `Prototype/Documentation/`

Vorschau:



C CD