TECHNISCHE UNIVERSITÄT DRESDEN

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

Diplomarbeit

KONZEPTE FÜR DIE ABWÄRTSKOMPATIBILITÄT VON iOS-APPS

Sebastian Hagedorn
Matr.-Nr.: 3387648

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

Eingereicht am 30.06.2013

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Thema: **Konzepte für die Abwärtskompatibilität von iOS-Apps**

Name, Vorname	Hagedorn, Sebastian	Studiengang	Medieninformatik
Matrikel-Nr.	3387648	Projekt/Schwerpunkt	Mobile
Betreuer	Dr. Thomas Springer	Externer Betreuer	-
Beginn am	1. Januar 2013	Einzureichen am	30. Juni 2013

ZIELSTELLUNG

Seit dem Erscheinen von iOS 2.0 im Jahr 2008 sind vier weitere Major-Versionen des mobilen Betriebssystems von Apple erschienen, dazu zahlreiche Zwischenversionen, die neben Fehlerbehebungen ebenfalls wichtige Neuerungen eingeführt haben. Aufgrund der hohen Entwicklungsgeschwindigkeit und starken Konkurrenz seitens Google (Android), aber auch der schnellen Adaptierung der Nutzer an neue Geräte- und Softwareversionen müssen Anwendungsentwickler neue Funktionen und APIs möglichst schnell in ihre Apps integrieren. Um Zugriff auf die neusten Schnittstellen zu erlangen, muss die aktuelle Version des von Apple bereitgestellten iOS SDKs verwendet werden. Durch eine Einstellung in der Entwicklungsumgebung Xcode lässt sich festlegen, bis zu welcher vorherigen iOS-Version eine Installation der Anwendung erlaubt wird. Zur Generierung von Warnungen und Fehlern in Xcode wird jedoch ausschließlich die Version des SDKs genutzt. Erlaubt der Entwickler die Installation auf älteren iOS-Versionen, muss er selbst dafür Sorge tragen, dass neue APIs und Symbole auf alten Versionen nicht referenziert werden. Übersieht der Entwickler ein solches Symbol, wird er während des Bauens der Anwendung nicht darauf hingewiesen, die Anwendung stürzt aber auf älteren Versionen ab. Fügt der Entwickler hingegen Code ein, um die Version zur Laufzeit abzufragen, entsteht viel redundanter Code, der von der eigentlichen Funktionalität der Anwendung ablenkt.

Ziel der Diplomarbeit ist die Entwicklung von Konzepten zur automatisierten Behandlung der Abwärtskompatibilität von iOS-Apps. Dies umfasst die Erkennung von APIs, die nicht auf allen unterstützten iOS-Versionen verfügbar sind, sowie die Überbrückung dieser Inkompatibilitäten. Die Mindestanforderung ist dabei, dass die Anwendung auf keiner der unterstützten Versionen abstürzt. Die Konzepte sollen zu einer einfach anwendbaren Gesamtlösung zusammengefasst und unter Einbeziehung von Entwicklern evaluiert werden.

SCHWERPUNKTE

- Recherche zu Konzepten für die Abwärtskompatibilität auf anderen Plattformen
- Sammlung möglicher Ansätze für Xcode-/iOS-Projekte
- Anwendung des vielversprechendsten Ansatzes auf einen Prototyp
- Analyse und Bewertung der Umsetzung (inkl. Test auf Geräten und Entwicklertests)

i. A. Braun

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, 30.06.2013

INHALTSVERZEICHNIS

1 EINLEITUNG

1.1 MOTIVATION

Vor sechs Jahren hat Apple mit dem iPhone nicht nur ein völlig neuartiges Gerät, sondern auch eine attraktive Plattform für Entwickler vorgestellt, wenngleich die Öffnung für Drittanbieter erst ein Jahr später mit iOS 2.0[1] vollzogen wurde. Seitdem sind 13 **Software-Updates** erschienen, die eine Aktualisierung des iOS SDKs und damit der Programmierschnittstellen (APIs) mit sich gebracht haben – dies entspricht einer durchschnittlichen Update-Rate von etwa viereinhalb Monaten[2]. Im Gegensatz zu Apples Hauptkonkurrent Google, der für das mobile Betriebssystem *Android* in einem vergleichbaren Zeitraum sogar 17 SDK-Updates ausgeliefert hat, können fast alle iOS-Nutzer von diesen schnellen und fortlaufenden Entwicklungen profitieren. Auf 10 von allen 16 jemals produzierten mobilen iOS-Endgeräten ist die aktuelle Version iOS 6.1 installierbar, selbst das vor vier Jahren vorgestellte iPhone 3GS lässt sich nach wie vor mit der neusten Software nutzen. Die **Nutzeradoption** schreitet bei Apple traditionell schnell voran, begünstigt durch die einfache und kostenlose Bereitstellung sämtlicher Updates. In [Sau12] ist belegt, dass sowohl iOS 4 als auch iOS 5 nur 16 Wochen nach Veröffentlichung auf über 60 % aller aktiven iOS-Geräte installiert waren. Verschiedene Quellen[3] legen nahe, dass es iOS 6 in einem vergleichbaren Zeitraum sogar auf 74 bis 80 % geschafft hat; innerhalb von acht Tagen waren mehr Geräte mit iOS 6 als mit iOS 5 in Benutzung [eth12]. Wenige Monate vor dem erwarteten Release von iOS 7 ist das aktuelle System bei einer Verbreitung von 93 % angelangt, wie Apple in der diesjährigen WWDC-Keynote bekanntgegeben hat [Coo13].

Die Endnutzer erwarten, dass nach einem Update des Betriebssystems auch die Apps innerhalb kürzester Zeit aktualisiert werden, um von neuen Funktionalitäten profitieren zu können. Apple erleichtert Entwicklern diese Aufgabe, indem das neue SDK bereits vor dem öffentlichen Release des zugehörigen Betriebssystems zur Verfügung gestellt wird[4].

[1] Ehemals *iPhone OS*

[2] Grundlage ist der Zeitraum von Juli 2008 bis Juni 2013

[3] Die Werte wurden dabei von [gam13], [Smi13] und [ora13] durch Analysetools in iOS-Apps erfasst

[4] Um Zugang zu Beta- und Vorabversionen zu erhalten, muss eine Mitgliedschaft im iOS Developer Program abgeschlossen werden: `https://developer.apple.com/programs/ios/`

Angesichts dieser Rahmenbedingungen scheint **Abwärtskompatibilität** auf der iOS-Plattform eher ein Randproblem zu sein. Führt man sich jedoch vor Augen, dass ingesamt über 500 Millionen iOS-Geräte verkauft worden sind [eng13], bedeuten selbst kleine prozentuale Anteile eine riesige Nutzerbasis. Besonders für soziale Apps und Messenger-Dienste ist es wichtig, nicht nur die meisten, sondern möglichst alle potenziellen Nutzer zu erreichen. Diese Erfahrung mussten im Januar 2013 auch die Entwickler des Nachrichtendienstes *WhatsApp*[5] machen, deren Ankündigung, das fünf Jahre alte iPhone 3G künftig nicht mehr zu unterstützen, ein großes Medienecho auslöste[6] [hei13].

Für iOS-Anwendungsentwickler stellt sich die Situation somit folgendermaßen dar: Der überwiegende Teil aller Nutzer verfügt über Geräte mit dem neusten Betriebssystem und erwartet dafür optimierte Anwendungen, während die Nutzer älterer Geräte zumindest die Grundfunktionalitäten einer App nutzen möchten. Der Fokus liegt also klar auf der neusten iOS-Version; Abwärtskompatibilität sollte möglichst einfach implementiert werden können und darf kein Hindernis für die Einbindung neuer Funktionalitäten darstellen.

Aus technischer Sicht wird die Unterstützung der neusten Geräte- und iOS-Generation durch regelmäßige Aktualisierungen der gesamten **Entwicklungswerkzeuge** seitens Apple sichergestellt. Die Verwendung der aktuellen Werkzeuge schließt ein, dass zum Bauen von Apps das aktuelle SDK verwendet wird[7]. Über eine Konfigurationseinstellung[8] in der Entwicklungsumgebung Xcode lässt sich bestimmen, ab welcher iOS-Version die Installation der App möglich ist – darüber hinaus bietet Xcode kaum Unterstützung, um die Anwendung auch tatsächlich für ältere Versionen kompatibel zu programmieren. Werden neu eingeführte APIs auf älteren Versionen des Betriebssystems aufgerufen, führt dies in den meisten Fällen zum Absturz der Anwendung. Die unbefriedigende derzeitige Situation ist, dass solche Abstürze erst durch Laufzeittests mit alten iOS-Versionen gefunden werden können. Wünschenswert wären daher Hinweise zur Compile-Zeit, welche APIs Probleme verursachen können, und wie diese Probleme behoben werden können.

1.2 ZIELSTELLUNG

In dieser Arbeit soll ein Konzept erarbeitet werden, das es Entwicklern erlaubt, ihren Anwendungscode so zu schreiben, als sei nur die **neuste Version des iOS SDKs** relevant. Durch diese grundlegende Entscheidung soll verhindert werden, dass neue Features aus Gründen der Abwärtskompatibilität zurückgehalten werden und dem Entwickler ein Wettbewerbsnachteil entsteht. Teil des Konzepts ist es, sicherzustellen, dass neu eingeführte Symbole[9] nur auf den unterstützenden iOS-Versionen referenziert werden, die Gesamtanwendung aber auch auf älteren Versionen stabil läuft. Dem Entwickler muss dabei eine Möglichkeit gegeben werden, das konkrete Verhalten im Falle eines Kompatibilitätsproblems zu steuern.

[5] `http://www.whatsapp.com/`

[6] Die zugrundeliegende Ursache dafür ist allerdings eine Entscheidung von Apple, die Prozessorarchitektur des iPhone 3G in der aktuellen Entwicklungsumgebung nicht mehr zu unterstützen

[7] Die Einbindung eines älteren SDKs in die aktuellen Werkzeuge ist in manchen Fällen möglich, aber kann nicht generell zugesichert werden

[8] „iOS Deployment Target“

[9] u. a. Klassen, Methoden und Funktionen

Auch Apple empfiehlt, bei der Anwendungsentwicklung nur bedingt Rücksicht auf ältere Versionen zu nehmen und neue Funktionalitäten auf solchen Systemen auszusparen, sofern sie die Stabilität oder Performance der Anwendung beeinträchtigen [TB11]. Dies ist jedoch derzeit keine triviale Aufgabe, da Xcode im Quellcodeeditor weder Such- noch Filteroptionen für **APIs** nach ihrer minimalen SDK-Version bietet. Zwar kann die Verfügbarkeitsinformation in den Header-Dateien und in der Dokumentation eingesehen werden, aber die manuelle Recherche birgt eine große Gefahr, einige der neuen APIs zu übersehen und dadurch Abstürze herbeizuführen.

Neben einem Konzept zur **Auflösung** von nicht abwärtskompatiblen APIs besteht ein wesentlicher Teil der Arbeit darin, Möglichkeiten zu untersuchen, diese APIs in einer existierenden Codebasis automatisch zu finden und durch **Warnungen** kenntlich zu machen. Um die Praxistauglichkeit der erarbeiteten Lösungen zu beweisen, soll das Konzept prototypisch umgesetzt werden. Die in Abschnitt 4.1 zusammengefassten Anforderungen an die Gesamtlösung stützen sich dabei auf eine initiale Befragung von Entwicklern, vorwiegend aus dem mobilen Bereich. Um das Konzept und dessen Umsetzung zu evaluieren, werden im Anschluss an die Implementierung **Entwicklertests** mit iOS-Programmierern durchgeführt.

1.3 RAHMENBEDINGUNGEN

Um möglichst viele Entwickler von dem erarbeiteten Konzept profitieren zu lassen und gleichzeitig den Rahmen dieser Diplomarbeit nicht zu überschreiten, müssen einige Rahmenbedingungen festgelegt werden. Es wird davon ausgegangen, dass die Entwickler eine **aktuelle Entwicklungsumgebung** nutzen und diese regelmäßig aktualisieren. Der Fokus liegt auf der Abwärtskompatibilität der gebauten iOS-Apps, nicht der Projektdateien im Hinblick auf die Version der Entwicklungsumgebung. Es wird die aktuelle Xcode-Version 4.6 verwendet werden, zusammen mit dem iOS 6.1 SDK. Allerdings sollen Konzept und Umsetzung auch mit kommenden Versionen der Entwicklungsumgebung und des SDKs kompatibel sein.

Die Abwärtskompatibilität findet ihre Grenzen in der von Xcode und dem Compiler unterstützten **iOS-Hardware**. Beispielsweise wurde mit der Veröffentlichung von Xcode 4.5 die Unterstützung der *armv6*-Prozessorarchitektur eingestellt[10], sodass keine Anwendungen mehr für das iPhone 3G und iOS 4.2.1 (oder darunter) gebaut werden können. Hardwarebeschränkungen wie diese werden in dieser Arbeit als untere Schranke für die Abwärtskompatibilität betrachtet. Ziel ist es nicht, die Anzahl der von Xcode unterstützten Versionen zu erhöhen, sondern die Stabilität der Apps für alle offiziell unterstützten Versionen zu maximieren.

Für den großflächigen Vertrieb von iOS-Apps gibt es zum Apple **App Store**[11] keine Alternative. Auch eine initiale Entwicklerbefragung lieferte das eindeutige Ergebnis, dass ein Konzept zur Abwärtskompatibilität zwingend mit dem App Store kompatibel sein muss. Dies bedeutet in erster Linie, dass eine kompilierte App auf allen Zielplattformen lauffähig sein muss, denn die Bereitstellung verschiedener Versionen einer App für verschiedene iOS-Systeme ist nicht möglich.

[10] `https://developer.apple.com/library/ios/releasenotes/DeveloperTools/RN-Xcode/#//apple_ref/doc/uid/TP40001051-SW174`

[11] `http://www.apple.com/iphone/from-the-app-store/`

Alle Implementierungen und praktischen Tests gehen von einer **unmodifizierten Laufzeitumgebung** aus[12]. Dementsprechend basieren alle anwendbaren Konzepte auf der Perspektive des Anwendungsentwicklers, wenngleich die Plattform- bzw. Frameworkperspektive in einzelnen Fällen aus Gründen der Vollständigkeit mit betrachtet wird.

Eine letzte Annahme für die Anwendbarkeit des Konzepts ist, dass bestimmte Grundfunktionalitäten einer App abwärtskompatibel sind. Setzen alle wesentlichen Teile einer App die neuste iOS-Version voraus, ist es sinnvoller, die Installation der App auf diese iOS-Version zu beschränken. Auch Spiele sind ein Sonderfall, weil deren Kompatibilität weniger von zur Verfügung stehenden APIs als vielmehr von der Hardwareperformance abhängig ist. Dieser Aspekt wird im weiteren Verlauf nicht genauer betrachtet werden.

1.4 GLIEDERUNG

Die Arbeit gliedert sich in sechs weitere Kapitel. In Kapitel 2 werden wichtige Begriffe eingeführt und bestehende Konzepte zur Abwärtskompatibilität auf anderen Plattformen untersucht. Das dritte Kapitel bildet den Einstieg in den iOS-spezifischen Teil dieser Arbeit, indem die iOS-Plattform zunächst überblicksweise vorgestellt wird und die für diese Arbeit zentralen Aspekte der Plattform im Detail betrachtet werden. Anschließend wird untersucht, welche der bekannten Ansätze aus Kapitel 2 für die iOS-Entwicklung anwendbar sind. Aus den anwendbaren Ansätzen werden die vielversprechendsten im 4. Kapitel aufgegriffen, umfassend diskutiert und zu einem Gesamtkonzept zur Abwärtskompatibilität von iOS-Apps zusammengefügt. Das Gesamtkonzept wurde in einem Prototyp umgesetzt, dessen Implementierungsentscheidungen und -details Gegenstand des 5. Kapitels sind. Kapitel 6 umfasst die Ergebnisse einer ausführlichen Evaluierung des Prototyps, die sich maßgeblich auf im Rahmen dieser Arbeit durchgeführte Entwicklertests stützen. Die Diplomarbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 7 ab.

[12] Für modifizierte Laufzeitumgebungen wird häufig der Begriff „Jailbreak“ verwendet

2 GRUNDLAGEN

Das folgende Kapitel teilt sich in begriffliche und technologische Grundlagen. Unter Abschnitt 2.1 wird der Begriff der *Abwärtskompatibilität* näher untersucht und genau definiert, was unter *kritischen APIs* zu verstehen ist. In Abschnitt 2.2 werden bestehende Technologien und Konzepte zur Überbrückung von Inkompatibilitäten vorgestellt. Betrachtet werden sowohl Ansätze aus der Wissenschaft als auch aus der Industrie.

2.1 BEGRIFFSEINFÜHRUNG

2.1.1 Arten der Kompatibilität

Obwohl der Begriff der **Abwärtskompatibilität** allgegenwärtig ist, wird er in der Fachliteratur nur selten für Software definiert. Die folgende Definition des Adjektivs „abwärtskompatibel" (engl.: „backwards-compatible") ist dem *Oxford Dictionary of English* [oxf10] entnommen:

„[Being] able to be used with an older piece of hardware or software without special adaptation or modification."

In der vorliegenden Arbeit steht das Verhältnis zwischen iOS-Apps und der iOS-Plattform[13] im Mittelpunkt. Eine iOS-App kann als abwärtskompatibel bezeichnet werden, wenn sie für die aktuelle Version der Plattform konzipiert und gebaut wurde, aber dennoch auf älteren Versionen der Plattform lauffähig ist. Alle älteren, aber noch relevanten Versionen einer Plattform werden als **Legacy-Systeme** bezeichnet [oxf10].

Im Detail wird üblicherweise zwischen *Quellcodekompatibilität* (engl.: „Source-Level Compatibility") und *Binärkompatibilität* (engl.: „Binary Compatibility") unterschieden [Lev12, red12, Red11,

[13] Die iOS-Plattform ist dabei als Begriff für die Menge aller von Apple bereitgestellten Frameworks und Bibliotheken anzusehen; physische Merkmale wie die Prozessorarchitektur werden in dieser Arbeit nur am Rande betrachtet, wie eingangs in Abschnitt 1.3 beschrieben

DW10, FCDR95]. Anwendungen, die nach einer Rekompilierung ohne Codeänderungen auf anderen Plattformen lauffähig sind, werden als **quellcodekompatibel** bezeichnet. Eine erfolgreiche Kompilierung setzt voraus, dass alle im Quellcode verwendeten Symbole[14] gefunden und aufgelöst werden können. Als Synonym für Quellcodekompatibilität wird häufig der Begriff der *API-Kompatibilität* verwendet [FCDR95].

Eine stärkere Bedingung ist die **Binärkompatibilität**, die garantiert, dass Anwendungen auch ohne eine Rekompilierung auf verschiedenen Plattformen lauffähig sind. Dafür wird zunächst die Quellcodekompatibilität vorausgesetzt [Lev12, Kapitel 13 The BSD Layer][15]. Grundlage für die Binärkompatibilität ist das *Application Binary Interface*, kurz ABI. Die binäre Schnittstelle wird von der API sowie vielen sprach- und architekturspezifischen Details bestimmt, darunter die Speichergröße von Datentypen, die Prozessorarchitektur, Konventionen für Systemaufrufe (engl.: „System Calls“) und Compiler-Spezifika [Red11, Kapitel 8 Versioning]. Binärkompatibilität ist besonders zwischen heterogenen Systemen wie den zahlreichen UNIX-Varianten ein Problem [Lev12], weniger zwischen inkrementellen Versionen eines Systems wie im Fall von iOS. Werden alle Komponenten (Anwendungen und Frameworks) mit dem gleichen Compiler[16] gebaut, lassen sich viele Konfliktfälle ausschließen. Dennoch können ABI-Kompatibilitätsprobleme in einigen Programmiersprachen[17] auch auf Quellcodeebene entstehen, wie das in [MS98] untersuchte *Fragile Base Class Problem*.

In [Red11] wird zusätzlich der Begriff der **funktionalen Kompatibilität** eingeführt. Sie garantiert über die Kompilier- oder Lauffähigkeit hinaus, dass eine Anwendung auf allen Plattformen das exakt gleiche Laufzeitverhalten aufweist. Dies muss nicht immer erstrebenswert sein. Während Performanceverbesserungen an einem Framework funktional kompatibel zu früheren Versionen sein sollten, brechen Bugfixes die funktionale Kompatibilität ganz bewusst. In anderen Quellen wird die funktionale Äquivalenz verschiedener API- und ABI-Versionen automatisch angenommen [red12].

2.1.2 Kritische APIs

Die in dieser Arbeit verwendete Definition einer **kritischen API** setzt folgende Randbedingungen voraus:

- Die Menge aller APIs ist klar definiert und bekannt[18]
- Es werden nur deklarierte und bekannte APIs genutzt (sogenannte „Public APIs“)
- Eine Plattformversion ist durch eine Versionsnummer eindeutig bestimmt
- Zu jeder API ist bekannt, auf welchen Versionen der Plattform sie verfügbar ist

[14] Je nach Programmiersprache sind das u. a. Funktionen, Methoden, Klassen und Definitionen

[15] Konzeptionell umfasst die Binärkompatibilität auch die Quellcodekompatibilität. In einigen Programmiersprachen und Sonderfällen kann dennoch der Fall eintreten, dass die Binärkompatibilität gewahrt wird, während die Quellcodekompatibilität gebrochen wird. Ein solches Beispiel befindet sich in [Red11, Kapitel 8.4.4 Binary Compatibility]

[16] Und den gleichen, die ABI beeinflussenden Compiler-Flags

[17] Insbesondere in kompilierten, objektorientierten Sprachen wie Java und C++

[18] z. B. durch die Header-Dateien eines SDKs

Die inkrementelle Entwicklung einer Plattform ist in Abbildung 2.1 am Beispiel von Mac OS X dargestellt. Mit jeder durch eine Versionsnummer gekennzeichneten Evolutionsstufe kann sich die **Menge der APIs** ändern, indem neue APIs hinzukommen oder alte APIs entfallen[19]. Im Kontext von Web Services werden häufig versionierte APIs verwendet, wie in [Ree11] beschrieben. Sofern versionierte APIs auf einer Plattform unterstützt werden[20], sollten semantisch verschiedene Versionen einer API als gänzlich verschiedene APIs behandelt werden, da man im Allgemeinen nicht davon ausgehen kann, dass sie austauschbar verwendbar sind. Das Hinzufügen einer neuen Version einer API entspricht also weitestgehend dem Entfernen der alten Version[21] und dem Hinzufügen der neuen Version. Umfasst die neue Version einer API ausschließlich syntaktische Änderungen, ist ein automatisches Mapping zwischen alter und neuer API denkbar – alte und neue APIs sind demnach austauschbar verwendbar, jedoch muss die Syntax des Aufrufs je nach verfügbarer Version angepasst werden.

In Abbildung 2.1 sind einige grundlegende Begriffe dargestellt. Das **Base SDK**[22] bestimmt die Plattformversion, gegen die eine Anwendung kompiliert wird. Da die Abwärtskompatibilität zu allen vorherigen Versionen nicht praktikabel ist, kann mit dem **Deployment Target**[23] eine untere Schranke festgelegt werden.

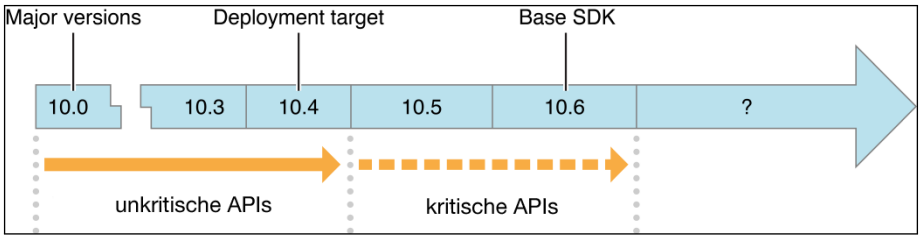


Abbildung 2.1: Kritische APIs am Beispiel von Mac OS X (Abbildung adaptiert aus [app10b])

Eine Anwendung mit den in Abbildung 2.1 verwendeten Parametern kann auf den Plattformen 10.4, 10.5 und 10.6 (sowie zukünftigen Versionen) installiert werden, auf älteren Systemen wird die Installation grundsätzlich verwehrt. APIs, die spätestens mit der Version des Deployment Targets (im Beispiel Version 10.4) eingeführt worden sind, werden als **unkritisch** bezeichnet, da sie in allen unterstützten Versionen verfügbar sind[24]. Alle später eingeführten oder geänderten APIs werden als **kritisch** bezeichnet, weil ihre Verfügbarkeit zwar vom Base SDK und damit vom Compiler angenommen wird, die Verfügbarkeit zur Laufzeit aber nicht garantiert werden kann.

Auf APIs, die in dieser Spanne entfernt worden sind, trifft die Problematik nicht zu, da solche APIs nicht im Base SDK enthalten sind und somit überhaupt nicht verwendet werden können.

[19] Bevor APIs gänzlich entfallen, sollten diese zuerst als veraltet (engl.: „deprecated“) markiert werden, um einen sauberen Übergang zu ermöglichen [Red11]
[20] In den Frameworks der iOS-Plattform besitzen einzelne APIs keine eigene Versionsnummer
[21] Sofern diese nicht weiterhin adressierbar ist
[22] Auch als „Build SDK“ oder nur „SDK“ bezeichnet
[23] Auch als „Minimum SDK“ bezeichnet
[24] Unter der Annahme, dass sie nicht zwischenzeitlich wieder entfernt worden sind

Formal kann eine kritische API folgendermaßen definiert werden:

$v(API)\ldots$ Version, mit der eine Schnittstelle eingeführt wurde
$v(DT)\ldots$ Version des Deployment Targets
$v(SDK)\ldots$ Version des Base SDKs

$$boolean\ isCritical = (v(API) > v(DT)) \wedge (v(API) \leq v(SDK))$$

2.2 BEKANNTE KONZEPTE ZUR UNTERSTÜTZUNG VON ABWÄRTSKOMPATIBILITÄT

Um fortwährende Kompatibilität auf einer Plattform sicherzustellen, müssen sowohl die Entwickler des Frameworks als auch die Entwickler der Anwendungen Maßnahmen ergreifen. Die wichtigsten dieser Maßnahmen werden in den folgenden Abschnitten benannt, geordnet nach ihrer Perspektive. Im Anbetracht des Themas dieser Arbeit soll dabei die Perspektive des Anwendungsentwicklers im Vordergrund stehen.

2.2.1 Die Frameworkperspektive

Einige wissenschaftliche Arbeiten setzen sich mit der Abwärtskompatibilität von Frameworks auseinander. Es muss beachtet werden, dass sich durch den Perspektivenwechsel vom Anwendungs- zum Frameworkentwickler auch die Bedeutung von Abwärtskompatibilität ändert – bislang wurde stets von der Anwendungsperspektive ausgegangen. Während eine abwärtskompatible Anwendung für ältere Frameworks (oder Laufzeitumgebungen im Allgemeinen) angepasst wird, muss ein aktuelles, abwärtskompatibles Framework für ältere Anwendungen nutzbar sein. Zur Unterscheidung der beiden Fälle werden im Zweifel die Begriffe *Anwendungsabwärtskompatibilität* und *Frameworkabwärtskompatibilität* genutzt. Auch Arbeiten über die Frameworkabwärtskompatibilität sind relevant, da sie generelle Mechanismen zur Steigerung der Kompatibilität zwischen Anwendungen und Frameworks verschiedener Versionen beschreiben.

In [SR07] und [SR09] werden auf dem Adapter-Pattern[25] aufbauende **Adaptierungsschichten** beschrieben, die entfallene oder geänderte APIs auf gültige APIs umleiten. Die Autoren gehen bei diesem Ansatz davon aus, dass die meisten Änderungen eines Frameworks durch Refactorings ausgelöst werden – die Funktionalität bleibt also gleich, nur die Aufrufe unterscheiden sich. Da die Adaptierung im Framework umgesetzt wird, muss der Änderungsverlauf der APIs auf der Frameworkseite bekannt sein. Dies impliziert, dass der Ansatz ausschließlich für Frameworkabwärtskompatibilität geeignet ist. Zum Erlangen von Anwendungsabwärtskompatibilität müsste eine solche Schicht auf der Anwendungsseite implementiert werden.

[25] Das Adapter-Pattern wurde in [GHJV94] eingeführt

Auch in allgemeinen Cross-Platform-Ansätzen werden **Entwurfsmuster** zur Überbrückung unterschiedlicher APIs genutzt, im Fall von [WK08] das Abstract-Factory-Pattern[26]. Wie bei den Ansätzen von [SR09, SR07] ist eine wesentliche Voraussetzung, dass die potenziellen Inkompatibilitäten im Vorfeld bekannt sind, um durch spezielle Designentscheidungen überbrückt zu werden. Unter diesen Bedingungen ist eine frameworkgestützte Anwendungsabwärtskompatibilität nur zu erreichen, wenn hinzukommende APIs bereits in früheren Versionen definiert, aber nicht implementiert worden sind. Die Anwendung könnte in diesen Fällen einen Rückruf (engl.: „Callback") erhalten, um die fehlende Implementierung auf Legacy-Systemen selbst zu überbrücken.

Nichtsdestotrotz können Frameworkentwickler Rahmenbedingungen schaffen, die die Entwicklung abwärtskompatibler Anwendungen erleichtern oder überhaupt ermöglichen. Um ein Framework langfristig erfolgreich zu gestalten, muss die API stabil und berechenbar sein [Red11]; Refactorings öffentlicher Schnittstellen sollten nach Möglichkeit ganz vermieden werden. In [Red11] sind Maßnahmen zusammengefasst, die das Design einer **stabilen Framework-API** auf Basis von C++ begünstigen. Das Buch umfasst die Definition eines API-Lebenszyklus sowie konkrete Implementierungsvorschläge für versionskompatible Schnittstellen. Das womöglich wichtigste Grundprinzip, das auch den zuvor genannten Entwurfsmustern zugrunde liegt, ist das der **Kapselung** (engl.: „Encapsulation"). Mit der Minimierung der nach außen sichtbaren Schnittstellen wird auch die Wahrscheinlichkeit von kompatibilitätsbrechenden Änderungen effektiv eingeschränkt. In C-basierten Sprachen bedeutet dies, nur die absolut notwendigen Schnittstellen durch die vom Framework exportierten Header sichtbar zu machen. Sollte die verwendete Programmiersprache selbst keine Möglichkeit der Introspektion bieten, können Makros und Konventionen helfen, die **Verfügbarkeit von Programmsymbolen** zur Compile- bzw. Laufzeit abzufragen. Apple setzt dies mithilfe von *Availability-Macros* um, deren Funktionsweise in Abschnitt 3.4.1 erläutert wird.

Große Plattformframeworks werden als dynamische Bibliotheken umgesetzt, um einerseits die Größe der Anwendungen zu minimieren und andererseits Änderungen wie Bugfixes allen Anwendungen zugänglich zu machen, ohne dass diese neu kompiliert oder verlinkt werden müssen [app12a, Dre11, FCDR95]. Damit dieser Schritt möglich ist, muss die **Binärkompatibilität** bei gleichbleibender API in jedem Fall gewahrt werden – dies liegt in der Verantwortung des Frameworkentwicklers. Während die Programmierschnittstelle in den meisten Sprachen zur Laufzeit abgefragt und im Zweifel auf der Anwendungsseite angepasst werden kann, wird die Binärschnittstelle zur Compile-Zeit festgelegt.

In [Dre11] werden ausführlich Konzepte zur Versionierung von ABIs auf verschiedenen Plattformen vorgestellt und diskutiert, einen Überblick der Thematik für C++-Anwendungen gibt auch [Red11]. Für eine Vielzahl von Programmiersprachen listet [FCDR95] auf, welche Quellcodeveränderungen zu ABI-Inkompatibilitäten führen. Für Java-Entwickler hat Oracle Richtlinien bereitgestellt, um die Binärkompatibilität von Java-Anwendungen sicherzustellen [ora05].

Im Idealfall wird vom Entwickler eines Frameworks explizit benannt, welche Art der Kompatibiliät zwischen verschiedenen Versionen eines Frameworks und den Anwendungen garantiert wird. Das Linux-Betriebssystem *Red Hat* definiert dazu verschiedene „Assurance Levels". Alle Frameworks und Bibliotheken, denen das Assurance Level 1 zugeordnet wurde, versprechen API- und ABI-Kompatibilität über 3 Major-Versionen hinweg; Level-2-Bibliotheken garantieren dies für alle

[26] Das Abstract-Factory-Pattern wurde in [GHJV94] eingeführt

Minor-Versionen innerhalb eines Major-Releases [red12]. Andere Plattformen, wie beispielsweise Android oder iOS, treffen keine solchen expliziten Vereinbarungen. Eine Übersicht der SDK-Level von Android-Releases zeigt, dass auch Subminor-Releases häufig API-Änderungen mit sich bringen – ob dabei nur neue APIs hinzukommen, ist nicht offensichtlich [andb]. API-Änderungen fanden auf iOS bislang nur in Major- und Minor-Releases statt [app13a], doch auch hier gibt es keine **festgeschriebenen Garantien**. Sowohl Android als auch iOS nutzen das Deprecation-Modell, sodass das Entfernen einer vorhandenen API für den Anwendungsentwickler vorhersehbar ist. Weder in der Android- noch in der iOS-Dokumentation werden Versprechen zur ABI-Kompatibilität gegeben. Allerdings ist aus der Praxis ersichtlich, dass Anwendungen über viele Plattformversionen hinweg lauffähig bleiben. Es darf in diesen Fällen davon ausgegangen werden, dass Google respektive Apple Anstrengungen unternehmen, die ABI-Kompatibilität für alle von den Entwicklungswerkzeugen unterstützten Plattformversionen zu erhalten.

2.2.2 Die Anwendungsperspektive

Im Folgenden wird davon ausgegangen, dass die zugrundeliegende Plattform ein hohes Maß an Versionskompatibilität für Anwendungen unterstützt, insbesondere die ABI-Kompatibilität wird vorausgesetzt. Folgen die Frameworkentwickler den im vorherigen Abschnitt diskutierten Richtlinien, können sich Anwendungsentwickler auf die Aspekte konzentrieren, die aus der Frameworkperspektive nicht oder nur schwer zu lösen sind. Dies beinhaltet insbesondere die **Detektion** und **Auflösung** kritischer APIs auf Legacy-Systemen. Im Gegenzug wird erwartet, dass Anwendungsentwickler nicht bewusst gegen die vom Frameworkentwickler getroffenen Maßnahmen vorgehen, z. B. durch die Verwendung privater Schnittstellen, die Benutzung von Schnittstellen entgegen ihrer Bestimmung laut Dokumentation, die Missachtung von Konventionen oder durch Hacks der Frameworks [TB11, Dre11].

Detektion kritischer APIs

Um die Verwendung kritischer APIs im Quellcode zuverlässig erkennen zu können, sollte dieser Prozess durch Werkzeuge unterstützt werden. Sowohl für die Detektions- als auch die Auflösungsphase dient die **Android**-Plattform aus verschiedenen Gründen als gutes Beispiel. Sie ist neben iOS die derzeit relevanteste mobile Plattform, jedoch im Vergleich zu iOS durch eine erheblich höhere Fragmentierung und Verbreitung älterer Versionen gekennzeichnet. Wie aus [and13b] hervorgeht, verwenden lediglich 4 % die aktuelle Android-Version 4.2, während mehr als ein Drittel aller Android-Nutzer die mehrere Jahre alte Version 2.3 nutzt. Durch das verschärfte Problem stehen Android-Entwicklern allerdings auch mehr Werkzeuge und Lösungsansätze zur Verfügung.

Um die Definition der kritischen API aus Abschnitt 2.1.2 anwenden zu können, müssen einige Android-spezifische Begriffe erläutert werden. Jede Android-Anwendung verfügt über eine XML-Datei namens `AndroidManifest.xml`[27]. In der Manifest-Datei können über das `<uses-sdk>`-Tag drei verschiedene Attribute definiert werden[28]:

[27] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

[28] Zur Steigerung der Lesbarkeit wurden Namensräume und andere syntaktische Details weggelassen – eine genaue Spezifikation kann in [andb] nachgeschlagen werden

- `minSdkVersion`: Entspricht dem *Deployment Target* und grenzt somit die Installationsbasis nach unten ein.

- `targetSdkVersion`: Entspricht konzeptionell dem *Base SDK*, allerdings kann über die Projekteinstellungen[29] eine andere SDK-Version für den tatsächlichen Build-Prozess bestimmt werden. Da es auf der Android-Plattform nicht nur reine API-Änderungen gegeben hat, sondern auch GUI- und Bedienkonzepte über Versionen hinweg angepasst worden sind, impliziert das Target-SDK auch das verwendete GUI-Konzept. In einigen Fällen nimmt die Laufzeitumgebung auf Basis dieser Einstellung automatische Adaptierungen vor[30].

- `maxSdkVersion`: Entspricht einer oberen Schranke für die Installation einer Anwendung – von der Benutzung wird im Allgemeinen abgeraten [andb].

Mithilfe der Manifest-Datei lässt sich also die Installationsbasis und somit die Gesamtmenge der kritischen APIs festlegen. Das **Lint-Tool**, das mit den Android Developer Tools (ADT) der Version 16 eingeführt wurde, beherrscht seit der 17. Version die sogenannten *API Checks*[31]. Im Gegensatz zum Java-Compiler, dem nur die Symbole des Base SDKs zur Verfügung stehen, kennt Lint die APIs aller SDK-Versionen und kann so gezielt nach der Verwendung kritischer APIs im Anwendungsquellcode suchen.

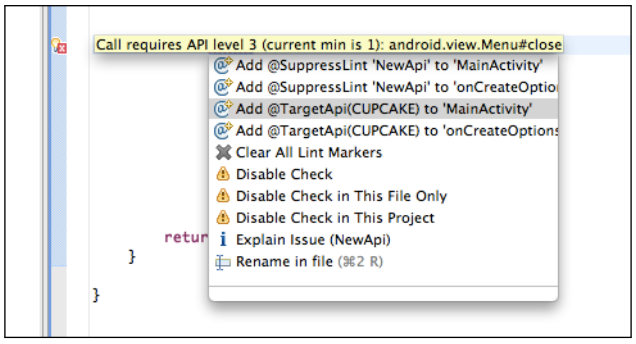


Abbildung 2.2: Kritische API in Lint/Eclipse

In Abbildung 2.2 ist die Darstellung einer Lint-Warnung in *Eclipse*[32] festgehalten. Im Beispielfall wurde das Deployment Target auf Version 1 gesetzt, die benutzte API verlangt aber mindestens Version 3. Für den Fall, dass der Programmierer bereits sichergestellt hat, dass die API nur auf kompatiblen Systemen aufgerufen wird, kann die Warnung mit einer `@TargetAPI`-Annotation[33] unterdrückt werden. Die API Checks werden von Google als „wichtigste aller Checks" bezeichnet und sind auch in die neue IDE *Android Studio*[34] integriert [ND13].

Die Android-**Dokumentation**[35] hält für jede Framework-API die Information bereit, mit welcher Version sie eingeführt wurde. Zusätzlich besteht die Möglichkeit, die verfügbaren APIs (Packages

[29] Die Einstellung ist in der `project.properties`-Datei unter dem Schlüssel `target` gespeichert
[30] Ein Beispiel dafür ist der Übergang von Hardware-Buttons zur Action Bar: `http://developer.android.com/guide/topics/ui/menus.html`
[31] `http://tools.android.com/recent/lintapicheck/`
[32] `http://www.eclipse.org/`
[33] `http://developer.android.com/reference/android/annotation/TargetApi.html`
[34] `http://developer.android.com/sdk/installing/studio.html`
[35] `http://developer.android.com/reference/packages.html`

und Klassen) nach ihrer mindestens benötigten Version zu **filtern**, wobei nur eine obere Grenze definierbar ist. Kritische APIs lassen sich somit nicht direkt anzeigen.

Für Entwickler, die die Evolution einer Plattform intensiv und fortlaufend verfolgen, können auch sogenannte **API Diffs** hilfreich sein. API Diffs sind das automatisch generierte Vergleichsergebnis der APIs von exakt zwei SDK-Versionen, typischerweise der aktuellen und der vorherigen Version. Es ist denkbar, sämtliche API Diffs automatisch auszuwerten und so eine Datenstruktur über die pro Version hinzugekommenen, also potenziell kritischen APIs zu erzeugen. Google veröffentlicht API Diffs mit jedem neuen Android-Release[36].

Auflösung kritischer APIs

Wie in der Einleitung dieser Arbeit erwähnt, ist die Nutzung kritischer APIs keineswegs unerwünscht. Allerdings muss sichergestellt werden, dass sämtliche kritische APIs ausschließlich auf kompatiblen Plattformen bzw. Plattformversionen aufgerufen werden. Maßnahmen zum Erreichen dieses Ziels werden unter dem Begriff der **Auflösung** zusammengefasst.

Erfolgt die Auflösung zur Entwicklungs- bzw. **Compile-Zeit**, entstehen verschiedene Apps für verschiedene Plattformversionen aus einer gemeinsamen Codebasis. Einen solchen Ansatz hat **BlackBerry**[37] für seine mobile Plattform gewählt[38]. Ausschlaggebend ist, dass die BlackBerry-Plattform selbst gar keine Anwendungsabwärtskompatibilität vorsieht:

„*Applications built using the BlackBerry JDE are forward-compatible with newer BlackBerry Java OS versions, but they are not backward-compatible with older versions.*" [bla11b]

Das Deployment Target entspricht in diesem Fall dem Base SDK, wodurch in der gebauten Anwendung keine kritischen APIs existieren können – APIs oberhalb des Deployment Targets sind nicht im Base SDK enthalten und führen zu Fehlern beim Kompilieren. Um im Quellcode dennoch neue APIs zu verwenden, sind in [bla11a] einige Maßnahmen beschrieben. All diesen Maßnahmen ist gemein, dass die Anwendung letztlich mit verschiedenen Base SDKs für verschiedene Plattformversionen gebaut werden muss. Eine der Maßnahmen verlässt sich auf die Tatsache, dass der Compiler ungenutzten (und potenziell inkompatiblen) Code selbstständig entfernt. Dieser Ansatz ist vor allem für versionsabhängige Callbacks des Frameworks anwendbar. Eine allgemeinere Maßnahme sieht die Verwendung von **Präprozessor-Makros** vor, die dafür sorgen, dass inkompatibler Code den Compiler gar nicht erst erreicht. Der Nachteil der Auflösung kritischer APIs beim oder vor dem Build-Vorgang ist neben oft unübersichtlichem und fehleranfälligem Code (speziell durch die Verwendung von Makros) die Tatsache, dass mehrere Apps zum Download angeboten werden müssen.

Flexibler ist die Auflösung zur **Laufzeit**, die auf verschiedene Weisen realisiert werden kann. In Abbildung 2.3 ist die typische Verwendung einer Framework-API dargestellt. Konzeptionell

[36] `http://developer.android.com/sdk/api_diff/17/changes/changes-summary.html`
[37] Ehemals *Research in Motion*
[38] Die folgenden Erkenntnisse beziehen sich auf die Entwicklung mit dem BlackBerry Java SDK: `https://developer.blackberry.com/java/`. Alternative Sprachen und Frameworks sind hier aufgezählt: `https://developer.blackberry.com/devzone/develop/platform_choice/index.html`

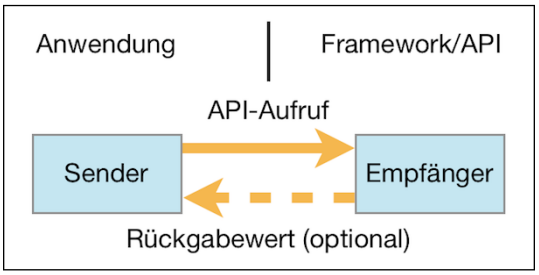


Abbildung 2.3: Sender und Empfänger beim API-Aufruf

kann die Auflösung auf Seiten des Senders oder des Empfängers stattfinden. Da Frameworks in der Regel in kompilierter Form vorliegen und nicht modifizierbar sind, ist eine Auflösung auf der Frameworkseite nicht trivial umsetzbar. Alternativ kann zwischen Sender und Empfänger eine Vermittlungsschicht eingeführt werden, die alle API-Aufrufe des Senders entgegennimmt und kritische APIs nur unter bestimmten Bedingungen an den eigentlichen Empfänger weiterleitet.

In Ermangelung wissenschaftlicher Arbeiten zum Thema Anwendungsabwärtskompatibilität wird im Folgenden erneut die Android-Plattform als praktisches Beispiel in den Vordergrund gerückt. Dabei soll zunächst die Senderseite betrachtet werden. Unter Android[39] ist es möglich, APIs aufzurufen, die außerhalb des Base SDKs liegen. Mittels **Reflection**[40] lässt sich zur Laufzeit herausfinden, welche Klassen und Methoden ein Framework zur Verfügung stellt, ohne dass diese Schnittstellen vom Compiler validiert werden. Somit lässt sich prüfen, ob eine API verfügbar ist – wenn dieser Test erfolgreich ist, kann sie verwendet werden. In Java-Code lässt sich Reflection folgendermaßen implementieren:

```
1 try {
2     // Versuch, Klasse zu laden
3     Class criticalClass = Class.forName("package.NameOfCriticalClass");
4     // Zugriff auf Klasse mittels Reflection
5     String descr = criticalClass.getField("FIELD_NAME").toString();
6 } catch (Throwable t) {
7     // API nicht verfügbar
8 }
```

Codebeispiel 2.1: Reflection in Java

Damit kann die Grundanwendung mit der niedrigsten zu unterstützenden SDK-Version (Deployment Target) gebaut werden. Der Ansatz ist **pessimistisch**, weil er vom kleinsten gemeinsamen Nenner aller Plattformen ausgeht und die Verwendung kritischer APIs eine absolute Ausnahmesituation darstellt. Bereits aus dem minimalen Codebeispiel ist ersichtlich, dass die häufige Nutzung von Reflection zu unübersichtlichem und typunsicherem Code führt, der zudem deutlich langsamer ist [Pow10, Mur11b]. Sowohl in [Pow10] als auch in [Mur11b, Kapitel 29 Handling Platform Changes] wird daher ein **optimistischer** Ansatz empfohlen, bei dem die Anwendung mit dem höchstmöglichen SDK gebaut und auf Reflection verzichtet wird – stattdessen sollten APIs

[39] Und mit Java im Allgemeinen

[40] Reflection ist verwandt mit Introspektion ([Sag98]) und wird häufig als stärkere Form der Laufzeitinspektion von Objekten und Klassen angesehen; nach dieser Sichtweise umfasst Reflection neben der Einsicht in Objekte/Klassen (Introspektion) auch deren Modifizierung [Eim05]

vor ihrer Verwendung auf ihre Verfügbarkeit hin geprüft werden. In diesem Fall stellt das Nichtvorhandensein einer kritischen API die Ausnahmesituation dar, die gesondert behandelt werden muss. Dies ist untenstehend in Java-Code für Android dargestellt.

```
1 if ( Build.VERSION.SDK_INT >= Build.VERSION_CODES.VERSION_NAME ) {
2     CriticalClass instance = new CriticalClass();
3 } else {
4     // Ausnahmebehandlung
5 }
```

Codebeispiel 2.2: Abfrage nach Plattformversion

Dieses Vorgehen wird in der Android-Dokumentation empfohlen [anda]. Allerdings muss im Vorfeld nicht nur bekannt sein, dass es sich um eine kritische API handelt, sondern auch, ab welcher Version sie verfügbar ist. In [Red11, Kapitel 8 Versioning] wird aus diesem Grund empfohlen, nicht nach Versionsnummern, sondern wie im folgenden Codebeispiel direkt nach Schnittstellen zu unterscheiden:

```
1 if ( ApiIsAvailable("package.CriticalClass") ) {
2     CriticalClass instance = new CriticalClass();
3 }
```

Codebeispiel 2.3: Abfrage nach Funktionalität

Die `ApiIsAvailable()`-Methode ist in Android nicht vorhanden, kann aber unter Nutzung von Reflection und Abwägung der dadurch entstehenden Performancenachteile leicht implementiert werden. Der Java Class-Loading-Mechanismus verhindert allerdings, dass die Codebeispiele 2.2 und 2.3 lauffähig sind. Beim Laden einer Klasse wird geprüft, ob alle referenzierten Symbole tatsächlich gefunden werden können. Auf Legacy-Systemen ist dies für `CriticalClass` nicht gegeben – obwohl sichergestellt ist, dass die Klasse auf diesen Systemen nicht benutzt werden würde, bricht der Class-Loader mit einem *VerifyError* ab. In [Pow10] ist im Detail beschrieben, wie diesem Problem mit Hilfe von Singletons und *Lazy Class Loading* begegnet werden kann. Das Problem ist programmiersprachenabhängig und soll daher an dieser Stelle nicht weiter diskutiert werden[41].

Wird der gesamte Anwendungscode von Versionsabfragen durchsetzt, ergeben sich daraus mehrere Nachteile. Hohe Redundanz und Streuung des Codes erschweren nachträgliche Änderungen, beispielsweise, wenn die Unterstützung einer bestimmten Version eingestellt wird. Die Gefahr, dass die Abfrage an einigen Stellen falsch oder garnicht implementiert wird nimmt zu. Nicht zuletzt leidet die Lesbarkeit des Anwendungscodes. Aus diesen Gründen werden in [Pow10] und [anda] **Abstraktionsschichten** eingeführt, die die Versionsabfragen kapseln. Der Anwendungscode muss so umgeschrieben werden, dass er die Schnittstellen der Zwischenschicht anstatt der Framework-API adressiert. Die Umsetzung basiert in beiden Fällen in erster Linie auf dem Abstract Factory Pattern – die Factory-Klasse instanziiert je nach Laufzeit-Version unterschiedliche Klassen, die über ein einheitliches Interface angesprochen werden können. Dieser Ansatz ist nicht Android- oder Java-spezifisch. In [SDH05] werden das Factory- und das Bridge-Pattern

[41] Insbesondere, da der Fokus dieser Arbeit auf Objective-C liegt und die Problematik in dieser Sprache nicht besteht

genutzt, um übersichtlichen und wiederverwendbaren Cross-Platform-Code in C++ zu schreiben, bei dem die Überbrückung unterschiedlicher APIs im Vordergrund steht.

Für häufig verwendete Android-APIs stellt Google mit den **Support Libraries**[42] selbst eine Zwischenschicht bereit, die potenziell kritische APIs nicht nur abfängt, sondern auch vollständig implementiert. Die Klassen der Support Libraries besitzen ein zu den Framework-Klassen identisches Interface, werden aber statisch in der Anwendung verlinkt und stehen unabhängig von der Laufzeitplattform zur Verfügung. Die Klassen der Support Libraries sind in der Regel nur durch die Zugehörigkeit zu einem anderen Paket von den Framework-Klassen zu unterscheiden. Im Gegensatz zu den zuvor diskutierten Abstraktionsschichten leiten die Android Support Libraries die Aufrufe nicht an die native Framework-API weiter, falls diese vorhanden ist. Stattdessen wird sowohl auf aktuellen als auch auf älteren Plattformen stets die Implementierung der Support Library genutzt. Nachteilig ist, dass Support Libraries durch ihre statische Verlinkung die Anwendungsgröße deutlich, und im Fall der Verfügbarkeit von APIs im nativen Framework unnötig vergrößern. Die Codeanpassungen sind dagegen minimal, schließlich muss nur die Paketzugehörigkeit beim Import der betroffenen Klassen geändert werden. Da Support Libraries das Framework nachbilden und offiziell vom Plattformentwickler bereitgestellt werden, sind sie konzeptionell eher auf der Frameworkseite einzuordnen, wenngleich sie beim Link-Vorgang ein fester Teil der Anwendung werden.

Für netzwerk- und serviceorientierte APIs hat Google einen weiteren Mechanismus entwickelt, der die Auswirkungen der hohen Versionsfragmentierung lindern soll. Bestimmte Schnittstellen wurden aus dem Plattform-Framework gestrichen und in eine Android-App ausgelagert. Die Anwendung trägt den Namen **Google Play Services**[43] und kann auf allen Geräten mit Android 2.2+ und dem Google Play Store[44] installiert werden. Die App läuft als Hintergrund-Service auf dem Gerät und fungiert als Erweiterung der dynamisch verlinkten Plattform-Frameworks, kann im Gegensatz zu diesen jedoch unabhängig vom Gerätehersteller und der aktuellen Plattformversion vom Nutzer auf den aktuellen Stand gebracht werden. Google Play Services ist eng an das Android-System gekoppelt und kann nicht von Entwicklern um eigene APIs erweitert werden. Derzeit werden APIs für Google Maps, ortsbezogene Dienste, Google+, Google Cloud Messaging und Google Play Game Services bereitgestellt [and13a]. Um auf die Google Play Services zuzugreifen, muss eine von Google bereitgestellte statische Bibliothek in die eigene Anwendung integriert werden. Die Bibliothek implementiert die Interaktion mit der Services-App und vereinfacht die Verfügbarkeitsabfrage der APIs. Grundsätzlich liegt die Verantwortung, die Verfügbarkeit zu prüfen, weiterhin beim Anwendungsentwickler. Problematisch ist zudem, dass Google Play Services im Emulator nicht zur Verfügung stehen und nur auf Geräten getestet werden können.

Eine Möglichkeit, Versionsabfragen komplett ohne Änderungen an der bestehenden Codebasis einzufügen, bietet **Aspektorientierte Programmierung** (AOP). Ein Aspekt ist eine Funktionalität oder Eigenschaft, die in verschiedenen und verteilten Bereichen einer Anwendung von Bedeutung ist und dabei die eigentliche Anwendungslogik nicht berührt – als Beispiele für die sogenannten **Cross-Cutting Concerns** werden häufig Logging oder Transaktionalität angeführt. Anstatt diese Cross-Cutting Concerns überall im Code verteilt zu implementieren oder explizit aufzurufen, kann

[42] `http://developer.android.com/tools/extras/support-library.html`
[43] Download: `https://play.google.com/store/apps/details?id=com.google.android.gms`
[44] `https://play.google.com/store/`

mittels AOP zentral definiert werden, welche Aspekte an welchen Stellen eingebunden werden sollen. Zur Definition wird oft das von AspectJ[45] eingeführte Join-Point-Modell verwendet, das folgende Begriffe umfasst [ecl09]:

- **Join Points** sind alle möglichen Quellcodepunkte, denen ein Aspekt zugeordnet werden kann. Die Granularität ist im allgemeinen AOP-Modell nicht festgelegt – je nach AOP-Technologie können Join Points Konstruktoren, beliebige Methoden oder Sprachkonstrukte wie Schleifen und konditionale Abfragen sein.
- **Pointcuts** selektieren einzelne Join Points, die genau definierte Bedingungen erfüllen, im einfachsten Fall eine Namenskonvention. Über reguläre Ausdrücke oder eine framework-spezifische Syntax lassen sich mit einem Pointcut z. B. alle Methoden auswählen, die mit `set` beginnen, d. h. die Menge aller Setter.
- **Advice** enthalten die Implementierung des Aspekts, z. B. den Logging-Code. Einem Advice ist ein Pointcut zugeordnet, der bestimmt, wann ein Advice ausgeführt wird. In den Advice ist spezifizierbar, ob der zusätzliche Code vor oder nach dem Join Point ausgeführt werden soll, oder diesen umschließen soll.

Zur Verdeutlichung der Funktionsweise soll das Codebeispiel 2.4[46] dienen.

```
1 // Definition eines Pointcuts 'set()': Ausführung eines Setters
2 pointcut set() : execution(* set*(*) );
3
4 // Definition eines Advice: Logging *vor* jedem 'set()'-Pointcut
5 before() : set() {
6   System.out.println(thisJoinPoint.getSignature());
7 }
```

Codebeispiel 2.4: Logging-Aspekt für alle Setter

Die Aufgabe der konkreten AOP-Technologie ist es nun, den Advice-Code vor jedem Aufruf eines Setters auszuführen. Dieser Prozess wird als **Weaving** bezeichnet, weil der Advice-Code in den normalen Anwendungscode eingewoben wird. Die Implementierung des Weavings hängt stark von den Möglichkeiten der Programmiersprache und der geforderten Kompatibilität mit den Standard-Entwicklungswerkzeugen und -Laufzeitumgebungen ab. AspectJ nutzt die kompilierten `.class`-Dateien als Grundlage für das Weaving, das zur Compile- oder Ladezeit durchgeführt werden kann [asp05]. Technisch gesehen wird dabei der Sender-Bytecode modifiziert, während der Java-Quellcode vom Sender unberührt bleibt. Auch der in [VBAM09] vorgestellte Ansatz mit dem Fokus des Weavings zur Laufzeit manipuliert den Java-Bytecode, wodurch Aspekte auch für kompilierte Framework-Klassen definierbar sind.

In dynamischeren Sprachen wie Smalltalk[47] kann AOP vollkommen ohne Compiler- oder Laufzeitunterstützung umgesetzt werden. Die Basis solcher Ansätze liegt in der laufzeitdynamischen

[45] AspectJ ist ein weitverbreitetes AOP-Framework für Java: `http://eclipse.org/aspectj/`
[46] Das Beispiel dient lediglich der Illustration des Konzepts und ist alleinstehend nicht lauffähig
[47] `http://www.smalltalk.org/`

Entkopplung von Nachrichten und Methodenimplementierungen. Um in Smalltalk eine Methode aufzurufen, wird vom Sender eine Nachricht an den Empfänger geschickt. Der Empfänger prüft mittels einer Tabelle, ob der Nachrichtenname einem Methodennamen (**Selektor**) entspricht. Jedem Selektor ist eine Methodenimplementierung zugeordnet – wird ein zur Nachricht passender Selektor gefunden, wird die in der Tabelle referenzierte Implementierung aufgerufen. Dieser Sachverhalt ist vereinfacht in Abbildung 2.4 dargestellt.

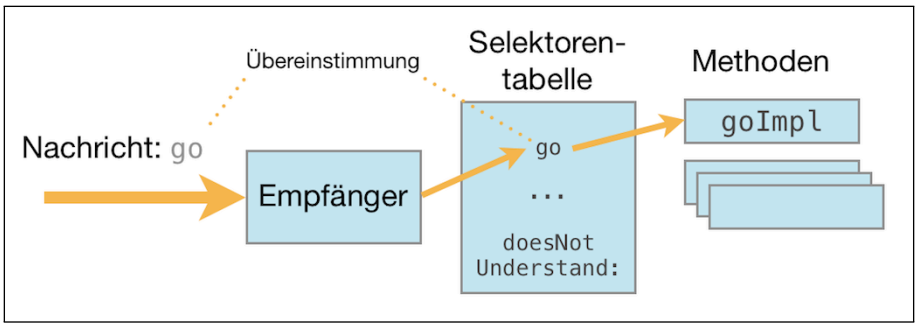


Abbildung 2.4: Erfolgreiches Senden einer Nachricht in Smalltalk

Wird kein dem Namen der Nachricht entsprechender Selektor gefunden, erhält das Empfängerobjekt von der Laufzeitumgebung eine `doesNotUnderstand:`-Nachricht[48] mit dem ursprünglichen Selektor als Argument. Dem Empfänger wird so die Möglichkeit gegegeben, dynamisch auf die ursprüngliche Nachricht zu reagieren. Diese Situation kann in Abbildung 2.5 nachvollzogen werden. Erst wenn für die `doesNotUnderstand:`-Nachricht keine Implementierung gefunden wird, kommt es zum Laufzeitfehler.

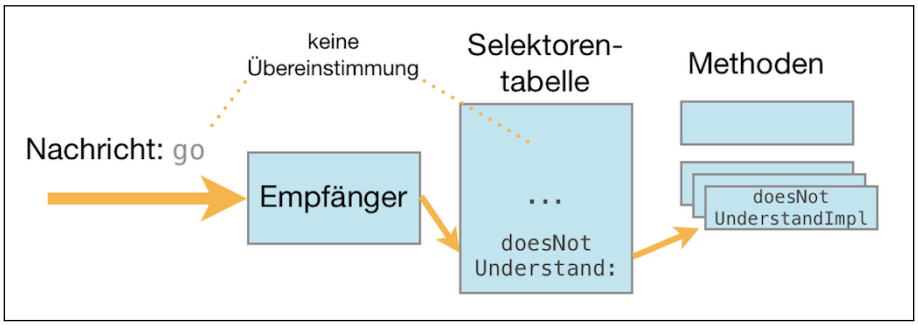


Abbildung 2.5: Fehlerfall beim Senden einer Nachricht in Smalltalk

Die Trennung von Nachrichten, Selektoren und Implementierungen ist wichtig, weil sich die Selektorentabelle durch die Programmiersprache zur Laufzeit modifizieren lässt. Auf diese Weise kann auch das Verhalten von Frameworkklassen verändert werden, ohne diese neu kompilieren zu müssen. In [BH90] wird dies genutzt, um die ursprüngliche Implementierung einer Methode in eigenen Code einzubetten, vereinfacht dargestellt in Abbildung 2.6. Dabei wird zuerst die originale Implementierung (`goImpl`) unter einem neuen Selektor (`originalGo`) erreichbar gemacht. Der ursprüngliche Selektor (`go`) wird hingegen auf eine eigene Methodenimplementierung (`customGoImpl`) umgeleitet, worin der Advice-Code platziert werden kann. Die Einbindung des Originalcodes (`goImpl`/Join Point) erfolgt über den Aufruf des neu hinzugefügten Selektors (`originalGo`). Der Ansatz ist in der mit Smalltalk eng verwandten Sprache Objective-C auch unter dem Begriff des **Method Swizzlings** bekannt [TB11].

[48] Doppelpunkte in Selektoren deuten auf einen Parameter hin

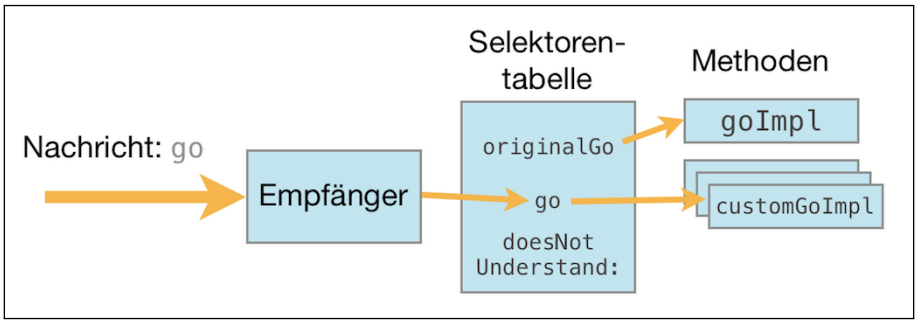


Abbildung 2.6: AOP in Smalltalk durch Vertauschung von Selektoren und Implementierungen

Eine in [BFJR98] vorgestellte und in [Hir03] genutzte Optimierung dieses Verfahrens sind sogenannte **Method Wrapper**. Auch dabei wird der ursprüngliche Selektor mit einer neuen Implementierung verbunden, der Aufruf der Originalcodes erfolgt aber ohne die Erzeugung eines weiteren Eintrags in der Selektorentabelle. Ein anderer Ansatz nutzt den `doesNotUnderstand:`-Mechanismus aus, um die Zuordnung von Nachrichten und Implementierungen vollständig überwachen zu können. Dazu müssen sogenannte **Encapsulators**[49] in die Klassenhierarchie eingefügt werden. Encapsulators enthalten keine Einträge in der Selektorentabelle[50], sodass alle Nachrichten an die `doesNotUnderstand:`-Implementierung weitergeleitet werden.

Alle genannten sowie weitere auf Smalltalk basierende Ansätze zur Umsetzung von AOP[51] wurden in [BFJR98] zusammengefasst und in [Duc99] evaluiert. All diesen Ansätzen ist gemein, dass sie im Gegensatz zu Compiler-orientierten Ansätzen nicht auf der Sender- sondern der **Empfängerseite** eingreifen.

Während der Recherche zu dieser Arbeit konnten keine Beispiele gefunden werden, die AOP direkt für die Umsetzung von **Abwärtskompatibilität** von Anwendungen nutzen. Jedoch wird das Programmiermodell häufig zum Erreichen verwandter Ziele eingesetzt. In [VBAM09] wird erklärt, wie sich Code und der Aufruf einer genau bestimmten Menge von APIs mittels AOP überwachen und analysieren lässt, ohne die Codestruktur und Wartbarkeit negativ zu beeinflussen. Ein in [GGKS07] beschriebenes Patent schützt die Verwendung von AOP, um Java-SE-Code nur durch Rekompilierung auf Java ME[52] lauffähig zu machen. Dabei werden alle Aufrufe von Methoden, die in Java ME nicht zur Verfügung stehen (vgl. kritische APIs), vom Compiler durch Aufrufe an andere Methoden umgeleitet. Eine Mischung aus dem Adapter-Pattern und AOP wird in [AMC+07] verwendet, um Cross-Platform-Spieleentwicklung zu betreiben. Dabei wird jede unterstützte Plattform als Aspekt umgesetzt: In den Advice können die plattformunabhängigen Basisklassen und deren Methoden um Plattformspezifika ergänzt werden.

Generell wird in zahlreichen Quellen, darunter [NAR08, Hir03, VBAM09, BFJR98], die Aussage getroffen, dass AOP sowohl für die Analyse als auch für die Modifizierung von Anwendungscode ein probates Werkzeug darstellt. Die Suche und Auflösung von kritischen APIs soll in dieser

[49] Der Begriff ist [Pas86] entnommen, wo auch das beschriebene Konzept erstmal ausführlich vorgestellt wurde. In anderen Quellen, wie [Duc99], ist stattdessen von *Minimal Objects* die Rede.

[50] Mit Ausnahme einer minimalen Anzahl von Selektoren zur Kompatibilität mit der Laufzeitumgebung

[51] Der Begriff der AOP wurde erst nach der Veröffentlichung der genannten Quellen geprägt, das zugrundeliegende Ziel ist jedoch das Gleiche

[52] Die Java Micro Edition (ME) enthält eine Untermenge der Bibliotheken der Java Standard Edition (SE) und wurde mit Blick auf mobile Plattformen und Embedded Systems spezifiziert: `http://www.oracle.com/technetwork/java/javame/`

Arbeit als Spezialfall davon betrachtet werden. Konzeptionell ist AOP bei der Auflösung kritischer APIs am ehesten als Zwischenschicht zwischen Sender und Empfänger einzustufen. Technisch umgesetzt wird es in manchen Technologien (u. a. AspectJ) auf der Sender- und in anderen auf der Empfängerseite (alle diskutierten Smalltalk-Ansätze).

2.3 ZUSAMMENFASSUNG

Zu Beginn dieses Kapitels wurden die für diese Arbeit wichtigsten Begriffe eingeführt und erklärt, darunter verschiedene Arten der Kompatibilität. Im weiteren Verlauf wird die Erhaltung der **Quellcodekompatibilität** im Vordergrund stehen, während die Binärkompatibilität zwischen Anwendungen und verschiedenen Plattformversionen als gegeben angesehen wird[53]. Der Weg zur Quellcodekompatibilität führt über die versionsspezifische Behandlung von bestimmten Schnittstellen, die unter dem Begriff der **kritischen APIs** zusammengefasst und genau definiert worden sind.

Sowohl Framework- als auch Anwendungsentwickler können zur Kompatibilität von Anwendungen beitragen. Diese Arbeit setzt auf einer für Dritte weitestgehend unveränderbaren Plattform (iOS) auf, weshalb die Perspektive des Frameworkentwicklers in erster Linie zum besseren Verständnis der Gesamtproblematik einbezogen wurde. Die Hauptaufgabe der Frameworkentwickler liegt in der Sicherstellung der ABI-Kompatibilität im Fall von kompatiblen APIs. Im Anschluss an die Frameworkperspektive wurde ein Vielzahl von Maßnahmen und Konzepten dargelegt, die Anwendungsentwickler implementieren können, um ihre Anwendungen abwärtskompatibel zu gestalten. Entwickler (oder stellvertretend deren Werkzeuge) müssen kritische APIs zuerst erkennen (Detektion), um sie anschließend auflösen zu können. Die Auflösung kann theoretisch schon zur Compile-Zeit erfolgen, was verschiedene Apps für verschiedene Plattformen zur Folge hat. Wird eine gemeinsame Binärdatei für alle Plattformen gefordert, müssen dynamische Auflösungstechniken angewandt werden. Im Folgenden sind die wichtigsten Werkzeuge, Technologien und Konzepte des Kapitels aufgelistet.

- **Detektion**
  - *Lint*: Werkzeug zur vollautomatischen Erkennung kritischer APIs in Android-Projekten
  - *API-Dokumentation*: Information über die mindestens benötigte Plattformversion für jede API; Filtermöglichkeiten
  - *API Diffs*: Ergänzung zur Dokumentation, indem der Änderungsverlauf in den Vordergrund gestellt wird
- **Auflösung**
  - *Compile-Zeit*
    - *Makros*: Präprozessor-Makros, um inkompatiblen Code vor dem Kompilieren zu entfernen

[53] Bei erfolgreicher Herstellung von Quellcodekompatibilität

- *Laufzeit*
  - *Reflection*: Nutzung von APIs, die nicht im Base SDK enthalten sind und nicht vom Compiler geprüft werden
  - *Konditionaler Code*: Nutzung einer API nur nach Abfrage einer Version oder einer Funktionalität
  - *Abstraktionsschicht*: Nutzung von Design Patterns, um verschiedene Implementierungen über ein einheitliches Interface ansprechen zu können
  - *Support Libraries*: Nachimplementierung und Imitierung von Framework-Klassen durch statisch verlinkbare Klassen
  - *Google Play Services*: Konzeptionell ähnlich zu dynamischen Bibliotheken, die sich gekapselt als App auf verschiedenen Plattformversionen (nach)installieren lassen
  - *AOP*: Vermeidung von verteiltem konditionalen Code durch die zentrale Definition von Aspekten; Implementierung programmiersprachenabhängig auf der Sender- oder Empfängerseite

Nicht betrachtet wurden bislang Technologien und Werkzeuge, die speziell für iOS oder Objective-C zur Verfügung stehen. In den folgenden Kapiteln soll daher ermittelt werden, welche der in diesem Kapitel beschriebenen und kategorisierten Technologien für das Gesamtkonzept auf der iOS-Plattform zur Verfügung stehen oder gegebenenfalls entwickelt werden müssen.

3 DIE iOS-PLATTFORM

Der Inhalt dieses Kapitels soll die Rahmenbedingungen verdeutlichen, unter denen sowohl das Konzept aus Kapitel 4 als auch die Implementierung aus Kapitel 5 bestehen müssen. In Abschnitt 3.1 wird eine allgemeine Einführung in die Programmiersprache und Entwicklungswerkzeuge der iOS-Plattform gegeben. Abschnitt 3.2 enthält eine Übersicht der iOS-Geräte- und Versions-Historie und diskutiert die wichtigsten und aus Entwicklersicht problematischsten Änderungen. Um spätere Designentscheidungen nachvollziehen zu können, werden einige ausgewählte Aspekte der iOS-Plattform in Abschnitt 3.3 im Detail beschrieben. Die im vorherigen Kapitel zusammengetragenen bekannten Konzepte zur Detektion und Auflösung kritischer APIs werden anschließend in Abschnitt 3.4 auf ihre Anwendbarkeit auf iOS-Apps geprüft. Ergänzend werden im gleichen Abschnitt die derzeitigen Best Practices zur Steigerung der Abwärtskompatibilität zusammengefasst.

3.1 PLATTFORMÜBERBLICK

Das iOS-Betriebssystem für mobile Geräte wurde 2007 unter dem Namen *iPhone OS* veröffentlicht und befindet sich inzwischen in Version 6. Es handelt sich um eine reduzierte Version von Apples Desktop-System *Mac OS X*[54], das wiederum auf **UNIX**[55] aufsetzt. Im Gegensatz zur Desktopvariante haben Entwickler und Anwendungen jedoch nur einen begrenzten Zugriff auf die Laufzeitumgebung und die unteren Systemschichten. So lassen sich native Apps ausschließlich über von Apple kontrollierte Vertriebskanäle installieren und Anwendungen können nur über die offiziell bereitgestellten Frameworks auf tiefere Systemschichten und Hardwarekomponenten zugreifen. Eine detaillierte Abhandlung des Betriebssystemkerns kann in [Lev12] nachgelesen werden.

Die iOS-App-Entwicklung setzt Grundwissen in drei Bereichen voraus: Der Programmiersprache Objective-C, den Frameworks des iOS SDKs sowie der Entwicklungsumgebung Xcode. Die ge-

[54] `http://www.apple.com/osx/`
[55] `http://www.unix.org/`

nannten Bereiche werden in den nächsten Abschnitten kurz vorgestellt, eine vollständige Einführung ist im Rahmen dieser Arbeit jedoch nicht vorgesehen. Eine detailliertere Zusammenfassung kann in [Hag12] nachgelesen werden, auch die offizielle Dokumentation von Apple enthält einen Technologieüberblick [app12c].

3.1.1 Objective-C

Objective-C ist eine Anfang der 1980er Jahre entwickelte, kompilierte, **objektorientierte Programmiersprache**. Eines der Hauptdesignziele war die vollständige Kompatibilität mit der etablierten Sprache C, wodurch jedes gültige C-Programm auch ein gültiges Objective-C-Programm darstellt. Konzeptionell orientiert sich Objective-C stark an der Sprache Smalltalk, die unter Abschnitt 2.2.2 bereits kurz vorgestellt wurde. Der wichtigste Unterschied zu anderen objektorientierten Sprachen wie Java oder C++ ist das von Smalltalk übernommene **Nachrichtenprinzip**. Anstatt Methoden direkt aufzurufen, wird einem Empfängerobjekt eine Nachricht geschickt. Der Empfänger versucht zunächst, in einer Tabelle eine dem Nachrichtennamen (**Selektor**) entsprechende Methodenimplementierung zu finden. Gelingt dies nicht, kann in einer generischen Methode eine laufzeitdynamische Behandlung der Nachricht erfolgen. Das Nachrichtenprinzip wurde in den Abbildungen 2.4 und 2.5 illustriert. Im Gegensatz zu Smalltalk heißt der Auffangselektor in Objective-C nicht `doesNotUnderstand:`, sondern `doesNotRecognizeSelector:`[56].

Nachrichten können auf vielfältige Weise gesendet werden. Die in Codebeispiel 3.1 gezeigte Nutzung der Standardnachrichtensyntax ähnelt dabei einem normalen Methodenaufruf aus anderen objektorientierten Sprachen.

```
1 // Dem Empfängerobjekt 'receiver' wird die Nachricht namens
2 // 'doSomething:' mit dem Objekt 'myArgument' als Argument geschickt
3 [receiver doSomething:myArgument];
```

Codebeispiel 3.1: Senden einer Nachricht in Objective-C

Die scheinbare Äquivalenz von Selektoren und Methoden kann vom Compiler genutzt werden, um Typprüfungen vorzunehmen. Sofern das Empfängerobjekt aus Beispiel 3.1 eine Methode mit dem Selektor `doSomething:` deklariert (und implementiert), kann von einer erfolgreichen Ausführung ausgegangen werden. Eine Beispieldeklaration befindet sich in Codebeispiel 3.2.

```
1 // Deklaration einer Methode mit dem Selektor 'doSomething:'
2 - (void) doSomething:(id)argumentName;
```

Codebeispiel 3.2: Deklaration einer Methode in Objective-C

Die vollständige Deklaration einer Methode wird **Signatur** genannt und enthält neben dem Selektor auch den Rückgabetyp (`void`), den Typ des Arguments (`id`), den Variablennamen des Arguments (`argumentName`) sowie die Information, ob es sich um eine Klassen- oder Instanzmethode handelt[57]. Zum Binden einer Nachricht an eine Methodenimplementierung wird zur Laufzeit je-

[56] In Objective-C wird eine ganze Reihe von Auffangselektoren durchlaufen, wobei `doesNotRecognizeSelector:` die letzte Stufe darstellt; ausführlich wird dies in Abschnitt 3.3.1 beschrieben

[57] Das im Beispiel verwendete Zeichen „`-`“ impliziert eine Instanzmethode, während „`+`“ für Klassenmethoden verwendet wird

doch ausschließlich der Selektor verwendet, Typinformationen werden ignoriert. Diese können dennoch vom Compiler verwendet werden, um auf mögliche Typkonflikte hinzuweisen.

Wird zu einem verwendeten Selektor keine passende Signatur deklariert, kann der Compiler daraus nicht zwingend schlussfolgern, dass die Nachricht gänzlich unbehandelt bleibt; beispielsweise kann die Nachricht in der `doesNotRecognizeSelector:`-Methode abgefangen werden. Um sich trotzdem auf vom Compiler garantierte **Typsicherheit** verlassen zu können, hat sich eine Konvention etabliert. Nutzt der Programmierer die in Codebeispiel 3.1 verwendete Standardnachrichtensyntax, drückt er zugleich die Intention aus, auf eine deklarierte Schnittstelle zugreifen zu wollen. Der Compiler prüft somit, ob die Schnittstelle zur Compile-Zeit vorhanden ist und generiert Warnungen bei fehlenden Deklarationen und Typkonflikten. Möchte der Programmierer diese Prüfungen bewusst umgehen, kann er eine alternative Syntax verwenden, die in Codebeispiel 3.3 dargestellt ist. Die Laufzeitsemantik ist identisch mit dem Codebeispiel 3.1.

```
1 // Versenden der Nachricht 'doSomething:' mit dem Argument 'myArgument'
2 [receiver performSelector:@selector(doSomething:) withObject:myArgument];
```

Codebeispiel 3.3: Senden einer Objective-C-Nachricht ohne Compiler-Prüfung

Durch die Verwendung dieser Syntax wird eine zuverlässige Compiler-Prüfung schon deshalb unmöglich, weil sich Selektoren auch zur Laufzeit aus Strings generieren lassen. Neben den gezeigten Varianten gibt es weitere Formen, Nachrichten zu verschicken, die aber an dieser Stelle nicht thematisiert werden sollen.

Die Nachrichtensyntax ist die offensichtlichste Ergänzung, die Objective-C seiner Grundsprache C hinzufügt. Fast alle weiteren Sprachergänzungen sind durch ein führendes `@`-Symbol gekennzeichnet, um Namenskonflikten aus dem Weg zu gehen. Die wichtigsten Syntaxergänzungen sollen kurz aufgezählt werden.

- `@class`: Forward-Deklaration einer Klasse
- `@interface`: Deklaration einer Klassenschnittstelle
- `@implementation`: Implementierung einer Klasse
- `@protocol`: Definition einer wiederverwendbaren Schnittstelle (vgl. `Interface` in Java)
- `@property`: Properties deklarieren Getter- und Setter-Methoden, u. a. für Instanzvariablen; der Zugriff auf Properties erfolgt mit einer vereinfachten Syntax oder der normalen Nachrichtensyntax

Alle Syntaxerweiterungen sind nicht nur mit C, sondern auch mit C++ kompatibel. Compiler wie GCC[58] und LLVM/Clang[59] unterstützen daher die Vermischung von C++- und Objective-C-Code unter der Bezeichnung Objective-C++.

[58] GNU Compiler Collection: `http://gcc.gnu.org/`
[59] `http://llvm.org/`

3.1.2 Das iOS SDK

Das iOS SDK umfasst ingesamt an die 50 **Frameworks** und über 130 weitere Bibliotheken[60]. Um einen besseren Überblick zu erhalten, wurden die Frameworks in vier Schichten unterteilt, die in Abbildung 3.1 dargestellt sind.

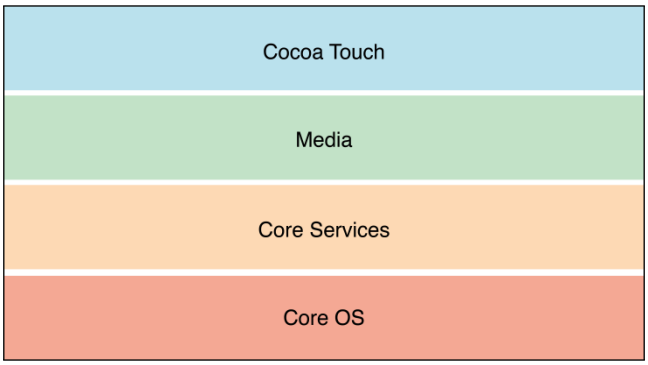


Abbildung 3.1: Schichten der iOS-Frameworks nach [app12c]

In [app12c] wird empfohlen, stets die höchstmögliche Schicht zu benutzen, da deren Frameworks Abstraktionen und somit Vereinfachungen der unteren Schichten darstellen. So verfügen sämtliche Frameworks der Cocoa-Touch-Schicht über objektorientierte Objective-C-Schnittstellen, während die Frameworks der unteren Schichten häufig nur C-Schnittstellen anbieten.

Die Entscheidung für oder gegen ein bestimmtes Framework hängt davon ab, welches Maß an Kontrolle über einen bestimmten Prozess notwendig ist. Beispielsweise lässt sich der Zugriff auf das Adressbuch sowohl über das *AddressBookUI*- als auch das *AddressBook*-Framework implementieren. Das erstgenannte Framework gehört der Cocoa-Touch-Schicht an und bietet eine Objective-C-Schnittstelle an, während die Alternative der Core-Services-Schicht angehört und nur über C-Schnittstellen angesprochen werden kann, dafür aber einen feingranularen Zugriff auf die Adressdatensätze zulässt. Ähnlich verhält es sich mit der Darstellung und Verarbeitung von grafischen Ressourcen: Das *UIKit*-Framework (Cocoa Touch) ermöglicht das Laden, Anzeigen und Animieren von Bildern über einfache Objective-C-Schnittstellen, für die erweiterte Bildmanipulation muss auf das *CoreGraphics*- oder *CoreImage*-Framework aus der Media-Schicht zurückgegriffen werden. Steht die Performance im Vordergrund, muss unter Umständen gar auf das *OpenGLES*-Framework[61] ausgewichen werden, das ebenfalls der Media-Schicht angehört.

Eine Sonderstellung besitzen die Frameworks *UIKit* und *Foundation*, die in jeder iOS-Anwendung verwendet werden. **UIKit** enthält neben sämtlichen charakteristischen GUI-Elementen auch die Klassen und Schnittstellen, die die Grundkommunikation zwischen der App und dem Betriebssystem ermöglichen. Dazu gehören Benachrichtigungen nach dem Start der App sowie Folgebenachrichtigungen, wenn die Anwendung in den Hintergrund rückt oder beendet werden soll. UIKit-APIs lassen sich durch den `UI`-Präfix[62] in Klassen- und Dateinamen leicht identifizieren.

[60] Im Gegensatz zu Bibliotheken enthalten Frameworks neben Code noch Header-Dateien und bei Bedarf zusätzliche Ressourcen

[61] `http://www.khronos.org/opengles/`

[62] Namensräume oder Paketstrukturen werden von Objective-C nicht unterstützt, weshalb Präfixe deren Zweck erfüllen müssen

Foundation stellt die wichtigsten Datentypen und -strukturen wie Strings, Arrays und Hashes sowie zahlreiche grundlegende Klassen bereit, die unabhängig von bestimmten Anwendungsdomänen von Bedeutung sind. Während UIKit speziell für mobile Geräte konzipiert und entwickelt wurde, ist die iOS-Variante des Foundation-Framework weitestgehend deckungsgleich mit dem Foundation-Framework des Mac OS X SDKs. Foundation-Klassen sind am einfachsten an dem `NS`-Präfix[63] erkennbar.

Alle Frameworks und Bibliotheken des iOS SDKs sind auf Apples mobilen Geräten vorinstalliert und werden **dynamisch verlinkt**[64]. Es ist weder möglich, die vorinstallierten Bibliotheken zu modifizieren, noch können eigene dynamische Bibliotheken installiert werden.

3.1.3 Xcode

Vordergründig für die Entwicklung von iOS- und Mac-Anwendungen stellt Apple seit vielen Jahren die integrierte Entwicklungsumgebung (engl.: Integrated Development Environment/**IDE**) Xcode kostenlos zur Verfügung[65]. Xcode unterstützt eine Vielzahl von Dateiformaten durch spezielle Editoren, von denen die wichtigsten im Folgenden kurz vorgestellt werden:

- **Quellcodeeditor**: Schreiben und Editieren von Quellcode inkl. Syntax-Highlighting, Autocomplete und Vorschau der API-Dokumentation
- **Projekteditor**: Konfiguration der Projekt- und Build-Einstellungen
- **Interface Builder**: Erstellen grafischer Oberflächen (XIB-Dateien) mit einem WYSIWYG-Editor[66]
- **Storyboard-Editor**: Konfiguration von Übergängen zwischen einzelnen Bildschirminhalten, aufbauend auf Interface Builder
- **Core-Data-Editor**: Konfiguration und Bearbeitung des (persistenten) Datenmodells
- **PLIST-Editor**: Editieren von PLIST-Dateien („Property Lists"), die auf den Apple-Plattformen einen Defacto-Standard für Konfigurationsdateien darstellen

Alle genannten Editoren dienen der Entwicklung von Anwendungen. Die Xcode-Installation enthält des Weiteren eine Vielzahl von Komponenten, die das Testen, Deployment und Debuggen dieser Anwendungen erleichtern. Über sogenannte **Schemes** lassen sich Laufzeitoptionen (Flags) und Testdaten (u. a. GPS-Routen und Nutzerdaten) speichern, um Testläufe reproduzierbar zu machen. Zur Durchführung von Laufzeittests kann eine iOS-App direkt aus Xcode gestartet und entweder auf einem angeschlossenen Gerät[67] oder dem mitgelieferten **Simulator** ausgeführt

[63] „NS" steht dabei für die Firma NeXTSTEP, die den Vorgänger des Foundation-Frameworks entwickelt hat und später von Apple aufgekauft wurde

[64] Dynamische Bibliotheken werden häufig auch als „Shared Libraries" bezeichnet

[65] Xcode kann über den Mac App Store oder das Developer-Portal bezogen werden: `https://developer.apple.com/xcode/index.php`

[66] engl.: „What you see is what you get" – zum Erstellen der Dateien wird eine GUI bereitgestellt, Programmierkenntnisse sind nicht erforderlich

[67] Gerätetests setzen eine Mitgliedschaft im iOS Developer Program voraus

werden. Sowohl bei Geräte- als auch Simulatortests können über den **Debugger** Objekte zur Laufzeit inspiziert werden. Die zur Xcode-Installation gehörende Anwendung **Instruments** erlaubt die Analyse der CPU- und Speicherauslastung einer Anwendung zur Laufzeit und weist bei der Benutzung vorgegebener Templates auf Speicherlecks und CPU-Engpässe hin. Die Oberfläche von Instruments während der Speicheranalyse einer App ist in Abbildung 3.2 dargestellt.

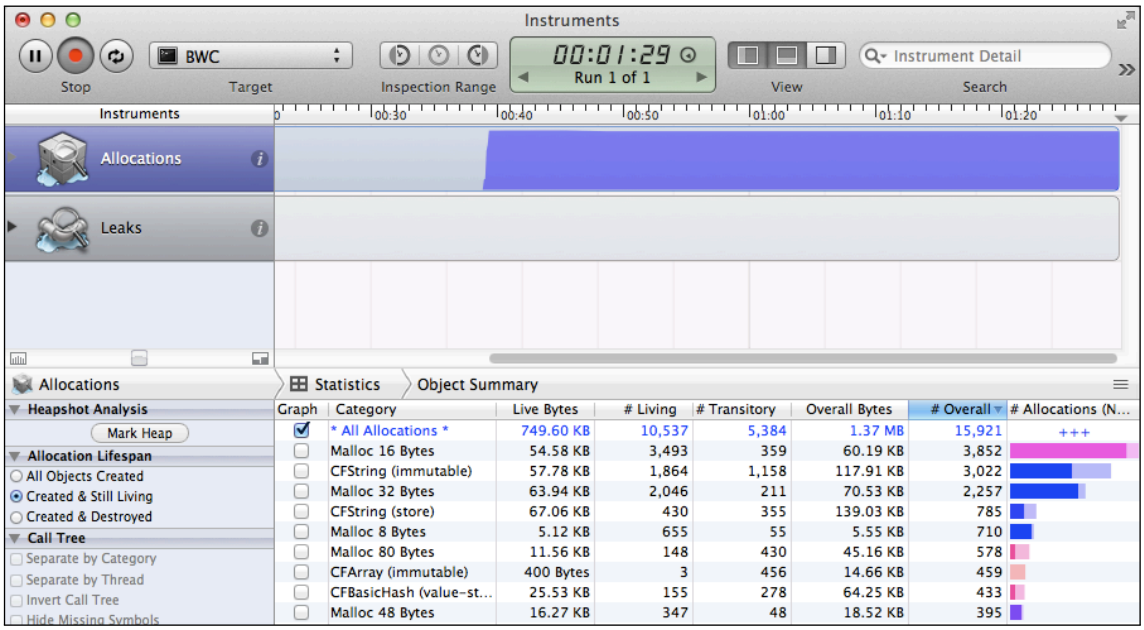


Abbildung 3.2: Speicher-Profiling mit Instruments

Neben den genannten Werkzeugen zur Untersuchung des Laufzeitverhaltens bietet Xcode auch eine statische Codeananalyse an, die auf dem Clang **Static Analyzer**[68] beruht. Der Static Analyzer durchwandert dabei jeden möglichen, zur Compile-Zeit ersichtlichen Weg im Programmcode und führt eine Reihe von Logiktests („Checks“[69]) aus.

Die Hauptansicht von Xcode dient der Verwaltung eines einzelnen Projekts. Im sogenannten **Organizer** (Abbildung 3.3) werden dagegen Informationen aufbereitet, die für alle oder viele Projekte relevant sind. Dazu zählen die Verwaltungsbereiche von Testgeräten, SCM-Repositories, Build-Caches und zum Vertrieb bestimmten Archiven sowie ein Browser für die Dokumentation.

Es besteht eine enge Kopplung zwischen Xcode und den **Apple-SDKs**. Sämtliche Frameworks sind ausschließlich als Teil der Xcode-Installation zu beziehen, wobei jede Xcode-Version nur die neusten Framework-Versionen enthält. Einige der Editoren, darunter Interface Builder und der Core-Data-Editor, arbeiten direkt mit den iOS- und Mac-Frameworks zusammen und sind dadurch für andere Projekte nicht nutzbar. Prinzipiell ist es möglich, ältere SDKs nachzuinstallieren – dies kann jedoch zu Kompatibilitätsproblemen mit den genannten Editoren führen und ist daher nicht empfehlenswert. Durch fortlaufende Weiterentwicklungen der SDKs und auch der Programmiersprache selbst motiviert Apple die Entwickler, neue Xcode- und Framework-Versionen schnell zu adaptieren und damit automatisch die alten Framework-Versionen zu ignorieren. Auf konkrete Beispiele der Programmiersprachenevolution wird später in Abschnitt 3.3.3 verwiesen.

[68] `http://clang-analyzer.llvm.org/`

[69] Eine Liste aller derzeit implementierten Checks ist online verfügbar:
`http://clang-analyzer.llvm.org/available_checks.html`

Abbildung 3.3: Xcode Organizer

Alternative IDEs für die iOS-Entwicklung sind rar, JetBrains **AppCode**[70] ist derzeit die einzige vollwertige Konkurrenz. Allerdings verlässt sich auch AppCode auf eine Xcode-Installation zum Bauen der Anwendungen und bietet keinen Ersatz für spezielle Editoren wie Interface Builder oder den Core-Data-Editor. AppCodes Fokus liegt auf Refactoring-Möglichkeiten, die Xcode nur in geringem Maße unterstützt.

Während die Programmiersprache Objective-C portabel ist, sind Xcode sowie sämtliche Apple-Frameworks ausschließlich unter Mac OS X[71] lauffähig. Mit GNUStep[72] existiert eine freie und plattformunabhängige Nachimplementierung des Cocoa-Frameworks[73], native iOS-Apps lassen sich aber nur unter Mac OS X bauen.

3.2 ÜBERSICHT DER VERSIONEN

Das iOS-Betriebssystem wurde zusammen mit dem *iPhone* der ersten Generation im Juni 2007 auf den Markt gebracht und kam wenig später im *iPod Touch* zum Einsatz. Seit seiner Markteinführung im April 2010 wird das *iPad* von iOS gestützt, auch die TV Set-Top Box *Apple TV* läuft seit der zweiten Generation (September 2010) unter iOS. Für die folgenden Betrachtungen werden zur besseren Übersicht lediglich **iPhone**- und **iPad**-Modelle betrachtet. Die iPod-Touch-Modelle haben zu keinem Zeitpunkt neue Technologien eingeführt, sondern orientierten sich zumeist an der letzten iPhone-Version. Für Apple TV gibt es derzeit kein SDK, sodass die Entwicklung von Anwendungen für Dritte nicht möglich ist. Daher spielt diese Gerätefamilie in dieser Arbeit keine Rolle. Apple hat bislang keinem anderen Hersteller eine Lizenz zur Nutzung des iOS-Betriebssystems

[70] `http://www.jetbrains.com/objc/`
[71] bzw. iOS, im Fall der iOS-Frameworks
[72] `http://www.gnustep.org/`
[73] Cocoa ist das Desktop-Gegenstück zu Cocoa Touch

		iOS Geräte										
		iPhone						iPad				
		Orig.	3G	3GS	4	4S	5	Orig.	2	3	4	Mini
Im Verkauf					✓	✓	✓		✓		✓	✓
Xcode-Support				✓	✓	✓	✓	✓	✓	✓	✓	✓
Architektur (arm...)		v6	v6	v7	v7	v7	v7s	v7	v7	v7	v7s	v7
iOS Version	**1.0** 1.1	✓										
	2.0 2.1 2.2	✓	✓									
	3.0 3.1 3.2	✓	✓	✓				✓				
	4.0 4.1 4.2 4.3		✓	✓	✓			✓	✓			
	5.0 5.1			✓	✓	✓		✓	✓	✓		
	6.0 6.1			✓	✓	✓	✓		✓	✓	✓	✓
∑	Major	3	3	4	3	2	1	3	3	2	1	1
	Minor	7	8	10	8	4	2	5	5	3	2	2

Tabelle 3.1: Versionsübersicht ausgewählter iOS-Geräte

auf anderen Geräten erteilt und eine Betrachtung der Firmenphilosophie und -geschichte lässt dies auch in Zukunft nicht erwarten [Isa11, Kapitel 25 Think Different: Jobs as iCEO]. Die Installation von iOS auf anderen Geräten durch den Endnutzer wird durch die iOS Lizenzbedingungen ausgeschlossen [app12b].

Um die Problematik der Abwärtskompatibilität und deren Ausmaß abschätzen zu können, sind in Tabelle 3.1 sämtliche bislang erschienenen iPhone- und iPad-Modelle mit ihrer initialen und maximalen iOS-Version aufgelistet, ergänzt um die Angabe der verwendeten Prozessor-Architektur, die derzeitige Unterstützung durch Xcode[74] und den Verkaufsstatus.

Deutlich erkennbar ist, dass die Software-Unterstützung die Verkaufsdauer der Geräte übersteigt und einige bereits eingestellte Geräte bis auf iOS 6 aktualisiert werden können. Beim Kauf eines neuen iOS-Geräts kann man im Durchschnitt mit **zwei Major-Updates** rechnen[75], bei den Minor-Updates gehen die Zahlen weiter auseinander. Besonders viele, bis heute anhaltende Updates hat das 2009 eingeführte **iPhone 3GS** erfahren, während das **iPad** der ersten Generation fast ein Jahr später eingeführt wurde, aber seit 2012 keine Aktualisierungen mehr erhält. Für neuere Modelle sind die statistischen Auswertungen weniger aussagekräftig, weil die Anzahl der zukünftig unterstützten Versionen offen ist. Es zeichnet sich jedoch ab, dass die Installationsbasis der

[74] Die Xcode-Unterstützung hängt direkt mit der Prozessorarchitektur zusammen; armv6-Anwendungen werden von Xcode nicht mehr unterstützt (Xcode 4.5 und neuer)

[75] Dabei wird davon ausgegangen, dass man das Gerät zum Verkaufsstart mit der minimalen initialen Version kauft

aktuellen Version zunimmt, was neben der Update-Fähigkeit alter Geräte auch der gewachsenen Modellpalette geschuldet ist.

Aus **Entwicklersicht** war die Einführung des iPads der kritischste Punkt der iOS-Versionshistorie. Initial wurde das iPad mit **iOS 3.2** ausgeliefert. Das Betriebssystem baut auf der vom iPhone bekannten Version 3.1 auf, stellt aber über das SDK einige Klassen zur Verfügung, die nur für die Verwendung auf Tablets vorgesehen sind. Mit der Veröffentlichung des iPhone 4 im Sommer 2010 erschien **iOS 4** auch für ältere iPhones und iPods, nicht jedoch für das iPad. Besonders problematisch war die resultierende Situation für sogenannte Universal Apps, die sowohl für die iPhone- als auch die iPad-Plattform entwickelt worden sind. Wird zur Entwicklung das iOS 4.0 SDK verwendet, muss die sichere Verwendung zahlreicher kritischer APIs auf dem iPad (mit iOS 3.2) berücksichtigt werden. Die Verwendung des iOS 3.2 SDKs würde dagegen auf iPhones und iPod Touches zahlreiche Features außer Acht lassen, die mit iOS 4 vorgestellt worden sind. Durch die Einführung von Multitasking in iOS 4 wurde zudem der Lebenszyklus der Apps erweitert, sodass nicht nur Unterschiede in den verfügbaren APIs, sondern auch im Laufzeitverhalten zwischen den Plattformen überbrückt werden mussten. Die Konsolidierung von iPad- und iPhone-Systemen wurde im Herbst 2010 mit iOS 4.2 vollzogen, eine Spaltung der SDKs ist seither nicht erneut eingetreten[76].

Neben den in Tabelle 3.1 gelisteten iOS-Versionen gibt es zahlreiche **Subminor-Versionen**, die Bugfixes enthalten. Von einigen iOS-Versionen gibt es verschiedene Build-Versionen für unterschiedliche Hardware, z. B. eine GSM[77]- und eine CDMA[78]-Variante des iPhones. Aus Entwicklersicht sind die verschiedenen Betriebssystem-Builds und Subminor-Versionen zu vernachlässigen, weil sie zu keinem Zeitpunkt die Framework-APIs beeinflusst haben – die Subminor- oder Build-Version spielt somit für die Anwendungsentwicklung keine Rolle.

Unabhängig von der iOS-Version erlaubt Xcode das Bauen von Anwendungen für eine Vielzahl von **Zielplattformen**. Lediglich eine inzwischen veraltete Prozessorarchitektur (armv6) verhindert die Installation neuer Anwendungen auf den ersten beiden iPhone-Generationen [xco12]. Durch die beschriebenen Probleme bei der Einführung des iPads und die Aufspaltung des SDKs zwischen iOS 3.2 und iOS 4.1 empfiehlt sich eigentlich iOS 4.2 als konservatives Deployment Target. Da jedoch lediglich das iPhone 3G, das aufgrund seiner Prozessorarchitektur keine Unterstützung mehr erfährt, nicht von iOS 4.2 auf 4.3 aktualisiert werden kann, wird iOS 4.3 gemeinhin als das niedrigste, sinnvolle Deployment Target angesehen. Seit Xcode 4.5 lässt sich kein Deployment Target unterhalb von iOS 4.3 mehr festlegen. Auch die Unterstützung von iOS 4.3 ist keine triviale Aufgabe, weil der iOS 4.3 Simulator auf dem aktuellen Betriebssystem Mac OS X 10.8 (*Mountain Lion*) nicht lauffähig ist. Stattdessen müssen Entwickler für Simulatortests die aktuelle Entwicklungsumgebung auf einem älteren Mac-System (10.7/*Lion*) installieren, um dort den Legacy-Simulator nutzen zu können. Es stellt sich daher die Frage, ob nicht iOS 5 als Deployment Target ausreicht, da alle iOS 4.3-Geräte (iPod Touches eingeschlossen) auch auf iOS 5 aktualisiert werden können. Dies spiegelt sich auch in Apples Statistik wieder, die für iOS 4 (und darunter) einen Nutzeranteil von etwa 1 % angibt [ios13].

[76] Es gibt dennoch gerätespezifische APIs, die jedoch erst zur Laufzeit Fehler verursachen, wenn sie auf nicht vorgesehenen Geräten verwendet werden

[77] `http://www.gsmworld.com/`

[78] `http://www.cdg.org/`

Auch ohne SDK-Fragmentierung gibt es von Zeit zu Zeit Probleme zwischen iOS-Versionen, die über API-Unterschiede hinaus gehen. Aktuelle Problem- und Sonderfälle werden später in Abschnitt 5.3 betrachtet. Allgemein lässt sich jedoch feststellen, dass die iOS-Plattform eine sehr homogene Entwicklung durchlaufen hat und die Geräte- und SDK-Fragmentierung minimal ist. In der bisherigen Geschichte gab es weder Paradigmenwechsel der UI-Gestaltung noch wurden der Lebenszyklus oder Grundbausteine und -patterns der Anwendungen grundlegend verändert. Die Unterstützung mehrerer Versionen ist daher erstrebenswert und mit vertretbarem Aufwand realisierbar.

3.3 AUSGEWÄHLTE ASPEKTE

Neben den bisher vermittelten Grundlagen der iOS-Plattform sollen in den folgenden Abschnitten einige Aspekte im Detail betrachtet werden. Allen ausgewählten Aspekten kommt in Hinblick auf die Abwärtskompatibilität und das in Kapitel 4 beschriebene Konzept eine tragende Rolle zu.

3.3.1 Laufzeitdynamik von Objective-C

In der Objective-C-Einführung in Abschnitt 3.1.1 wurde das Nachrichten- und Selektor-Prinzip erläutert, das der Sprache eine hohe Laufzeitdynamik gestattet. Im Folgenden sollen Möglichkeiten aufgezeigt werden, wie man auf dieser Grundlage bestehende Klassen und APIs zur Laufzeit beeinflussen kann.

Categories

Categories sind direkt in die Programmiersprache integriert und erlauben die **Laufzeitmodifikation** bestehender Klassen. In Codebeispiel 3.4 wird zunächst gezeigt, wie eine Category deklariert wird.

```
1  // Die Category 'MyObject' modifiziert die Basisklasse 'NSObject'
2  @interface NSObject (MyObject)
3
4  // *Hinzufügen* einer Methode
5  - (void) myMethod;
6
7  // *Überschreiben* einer Methode, die NSObject bereits implementiert
8  - (NSString*) description;
9
10 @end
```

Codebeispiel 3.4: Deklaration einer Category in Objective-C

Dabei wird die Basisklasse des Foundation-Frameworks, `NSObject`, durch eine Category namens `MyObject` ergänzt. Analog zur Deklaration einer Klasse erfolgt die Deklaration einer Category normalerweise in einer Header-Datei (`.h`-Datei). Die gezeigte Category deklariert zwei Methoden: `description`, die auch von `NSObject` deklariert wird[79], und `myMethod`, die ausschließlich von der Category deklariert wird. Die Implementierung erfolgt in einer separaten Datei (`.m`-Datei). Wird die Category beim Linkvorgang berücksichtigt und dadurch Teil einer Anwendung, wird sie im direkten Anschluss an die Basisklasse von der Laufzeitumgebung geladen[80]. Dabei werden die in der Category implementierten Methoden der Basisklasse hinzugefügt. Implementiert eine Category Methoden, die bereits in der Basisklasse vorhanden sind, werden diese beim Laden der Category überschrieben. Der Zustand der `NSObject`-Klasse nach dem Laden der Category aus Codebeispiel 3.4 ist in Abbildung 3.4 illustriert. In der Selektorentabelle[81] wurde der Eintrag des `description`-Selektors von der Category überschrieben und verweist auf die Implementierung der Category. Die Implementierung der Basisklasse ist zwar weiterhin existent, aber im abgebildeten Szenario nicht mehr erreichbar. Der `myMethod`-Selektor wurde hinzugefügt, alle weiteren bestehenden Selektoren und Methoden von `NSObject` bleiben unberührt. Der **Laufzeitzustand** der Basisklasse hängt also sowohl von der Klassenimplementierung als auch den verlinkten Categories ab, wobei die Implementierungen der Categories eine höhere Priorität genießen.

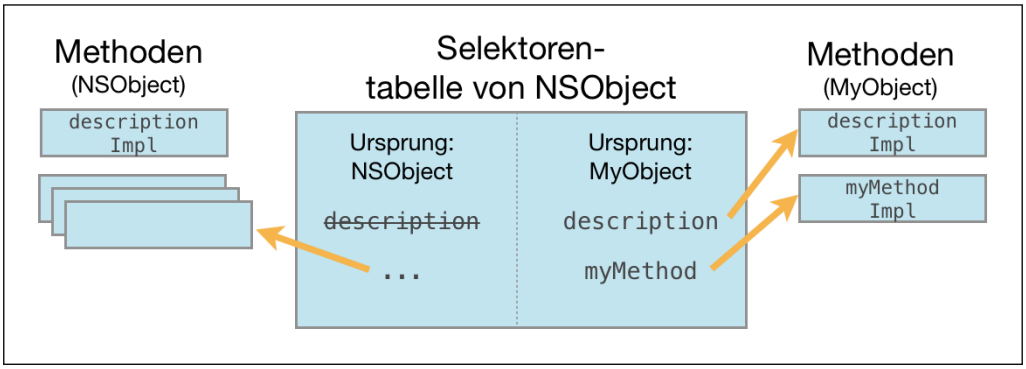


Abbildung 3.4: Auswirkung von Categories zur Laufzeit

Durch das Laden von Categories zur Laufzeit können Klassen modifiziert werden, deren Quellcode dem Anwendungsentwickler nicht vorliegt, beispielsweise Framework-Klassen. Allerdings führt das **Überschreiben** von Framework-Methoden schnell zu unerwünschten und unvorhersehbaren Seiteneffekten, zumal die Originalimplementierung nicht ohne Weiteres erreichbar ist. Deshalb wird von dieser Praxis abgeraten [TB11] – Xcode generiert in diesen Fällen Warnungen[82].

Die Benutzung von Categories erfolgt vollkommen **transparent**. Im Gegensatz zur Verwendung von Subklassen werden die Nachrichten direkt an die Basisklasse und deren Instanzen gesendet, weder im Quellcode noch am Laufzeitobjekt lässt sich das Vorhandensein einer Category

[79] API-Dokumentation: [app13c]

[80] Existieren mehrere Categories zu einer Basisklasse, ist die Ladereihenfolge nicht deterministisch

[81] Der Begriff der Selektorentabelle wurde hier von Smalltalk übernommen, um die im Grundlagen-Kapitel gezeigten Konzepte leichter transferieren zu können; die Selektorentabelle wird in Objective-C offiziell *Dispatch Table* genannt, erfüllt aber den gleichen Zweck; eine ausführliche Einführung in den Nachrichtenmechanismus von Objective-C wird in [app09] gegeben

[82] Dazu muss das `-Wobjc-protocol-method-implementation`-Compiler-Flag gesetzt sein, und die betroffene Methode muss direkt in der Klasse der Category (und nicht in einer Basisklasse) deklariert sein – andernfalls wird lediglich ein leicht zu übersehender Hinweis im Log ausgegeben

zum Zeitpunkt des Aufrufs erkennen[83]. Entgegen Behauptungen, wie sie etwa in [Jah12] getroffen werden, hängt der Geltungsbereich einer Category nicht von `#import`-Anweisungen ab[84]. Einmal geladen[85], gilt eine Category für alle Klassen und Instanzen der gesamten Anwendung, einschließlich aller Frameworks [TB11]. Daher sind Categories eine mächtiges, aber auch gefährliches Feature von Objective-C.

Falls es notwendig sein sollte, vor dem Laden einer Category Code auszuführen, kann dazu die `load`-Klassenmethode verwendet werden. Die Methode wird höchstens einmal und garantiert vor der Verwendung jeglicher anderer Methoden der Category aufgerufen. In `load` können beispielsweise Pointer auf Methoden gesichert werden, die beim Laden der Category überschrieben werden. Das Konzept des sogenannten „Method Swizzlings“ wurde bereits in Abschnitt 2.2.2 beschrieben, die technische Grundlage in Objective-C wird im folgenden Abschnitt skizziert.

Die Runtime-Bibliothek

Mit Categories lassen sich vorhandene Klassen auf einfache, vom Compiler überwachte Weise um Methoden erweitern. Ein limitierender Faktor ist dabei, dass sämtliche Instanzen der modifizierten Klasse betroffen sind. Problematisch ist auch, dass bereits vorhandene Methoden unwiederbringlich überschrieben werden und der Ladezeitpunkt einer Category nicht im Einflussbereich des Programmierers liegt. Alternativ lassen sich Klassen und deren Instanzen auf einer systemnahen Ebene durch **C-APIs** modifizieren. Diese APIs sind in einer Runtime-Bibliothek zusammengefasst, deren Nutzung den Import des `objc/runtime.h`-Headers voraussetzt. Die Bibliothek selbst wird in allen nativen Mac- und iOS-Anwendungen automatisch dynamisch verlinkt[86].

Die Hauptaufgabe der Bibliothek ist es, die Objektorientierung und Nachrichtenfunktionalität von Objective-C zu implementieren. Der Compiler übersetzt zunächst die Objective-C-Syntax in entsprechende Aufrufe an die Runtime-Bibliothek [app09]. So wird die in Codebeispiel 3.5 gezeigte Nachricht in einen **Funktionsaufruf** umgewandelt, der in Codebeispiel 3.6 dargestellt ist.

```
1 [receiver messageWithArg:(id)arg1 andArg:(id)arg2];
```

Codebeispiel 3.5: Senden einer Nachricht im Objective-C-Anwendungsquellcode

```
1 // 'receiver' -- Unveränderte Referenz auf Empfängerobjekt
2 // 'selector' -- Selektor der gesendeten Nachricht: 'messageWithArg:andArg:'
3 // 'arg1' / 'arg2' -- unveränderte Argumente der Nachricht
4 objc_msgSend(receiver, selector, arg1, arg2);
```

Codebeispiel 3.6: Senden einer Nachricht in C-Code (in Bezug auf Codebeispiel 3.5)

[83] Die in Abbildung 3.4 getroffene Aufteilung der Selektoren nach ihrem Ursprung ist rein konzeptionell und nicht tatsächlich einsehbar

[84] Leider suggeriert die Autocomplete-Funktion von Xcode fälschlicherweise einen gegenteiligen Eindruck

[85] Categories aus Bibliotheken werden nicht in allen Fällen verlinkt – dies kann jedoch durch das Setzen des `-ObjC`-Linkerflags forciert werden

[86] In dieser Arbeit wird hauptsächlich die sogenannte *Modern Runtime* in Zusammenhang mit Objective-C 2.0 betrachtet; unter iOS ist keine andere Runtime-Umgebung verfügbar

Die `objc_msgSend()`-Funktion implementiert die im nachfolgenden Unterabschnitt beschriebene Nachrichtenauflösung und gegebenenfalls -weiterleitung. Neben dieser Grundfunktion stellt die Runtime-Bibliothek zahlreiche Schnittstellen zur Verfügung, die das Inspizieren und Verändern bestehender, sowie das Hinzufügen neuer APIs zur Laufzeit gestatten. Einige davon sollen nun exemplarisch beschrieben werden.

- Introspektion
  - `class_copyIvarList(Class cls, unsigned int *outCount)`: Gibt die Menge aller Instanzvariablen einer Klasse zurück
  - `class_copyMethodList(Class cls, unsigned int *outCount)`: Gibt die Menge aller Methoden einer Klasse zurück
  - `class_getMethodImplementation(Class cls, SEL name)`: Gibt die Implementierung für einen bestimmten Selektor zurück, also den Eintrag in der Selektorentabelle
- Modifikation
  - `class_addMethod(Class cls, SEL name, IMP imp, const char *types)`: Fügt einer bestehenden Klasse eine Methodenimplementierung (`imp`) hinzu, die unter einem neuen Selektor (`name`) erreichbar gemacht wird
  - `class_setSuperclass(Class cls, Class newSuper)`: Verschiebt eine Klasse in der Vererbungshierarchie
  - `objc_allocateClassPair(Class superCls, const char *name, size_t size)`: Legt eine neue Klasse an
  - `objc_registerClassPair(Class cls)`: Registriert eine (neu angelegte) Klasse, damit diese anwendungsweit verwendet werden kann
  - `object_setClass(id object, Class cls)`: Setzt die Klasse einer einzelnen Instanz neu

Eine vollständige API-Dokumentation ist unter [app10a] verfügbar, doch schon die hier getroffene Auswahl verdeutlicht, dass nahezu alle Parameter einer Klasse zur Laufzeit manipuliert werden können. Insbesondere können Instanzvariablen und Methoden bestehender Klassen hinzugefügt, entfernt oder überschrieben werden.

Dynamisches Binden von Nachrichten

Das Binden einer Nachricht erfolgt in bis zu **fünf Stufen**. Zuerst versucht die Laufzeitumgebung, für den Selektor der Nachricht eine Methodenimplementierung in der Selektorentabelle zu finden. Schlägt dies fehl, kann programmatisch eingegriffen werden, etwa durch eine Weiterleitung der Nachricht an ein anderes Objekt. Die dazu benötigten Programmierschnittstellen und deren Aufrufreihenfolge ist in Abbildung 3.5 dargestellt, deklariert werden sie von `NSObject` und stehen somit allen Klassen und Objekten zur Verfügung[87].

[87] `NSObject` ist die Wurzel der Klassenhierarchie in Cocoa- und Cocoa-Touch-Apps

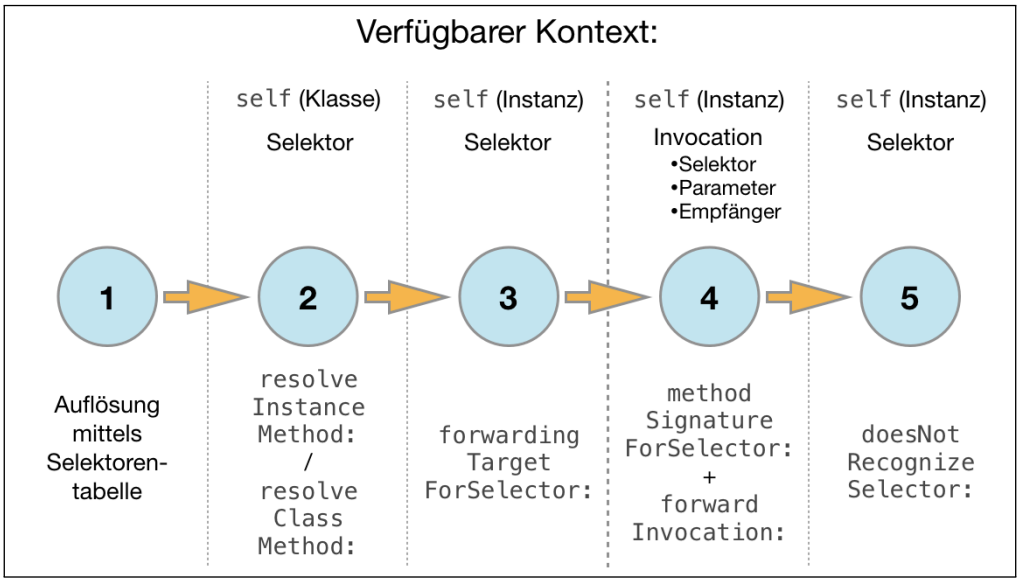


Abbildung 3.5: Stufen der Nachrichtenauflösung in Objective-C

Jeder Stufe der Nachrichtenauflösung stehen unterschiedliche **Kontextinformationen** zur Verfügung. Anfangs wird neben Informationen zum benachrichtigten Objekt lediglich der Selektor der aufzulösenden Nachricht bereitgestellt. In Stufe (4) werden sämtliche Nachrichteninformationen durch ein Objekt vom Typ `NSInvocation`[88] bereitgestellt. Die Erstellung dieses Objekts ist jedoch mit einem spürbaren Performancenachteil verbunden [app13c], weshalb eine Auflösung in den vorherigen Stufen erstrebenswert ist. Die einzelnen Stufen sollen nun im Detail beschrieben werden, wobei sich die Erkenntnisse auf die API-Dokumentation in [app13c] stützen.

(1) **Selektorentabelle**: In der Selektorentabelle wird eine zum Selektor der Nachricht passende Methodenimplementierung gesucht. Bevor zur nächsten Stufe übergegangen wird, werden die Selektorentabellen der Superklassen durchsucht. Aus Programmierersicht kann an dieser Stelle nicht eingegriffen werden.

(2) **Permanente Auflösung**: Die Schnittstellen dieser Stufe erlauben es, zur Laufzeit eine Implementierung für den in Frage stehenden Selektor nachzureichen. Wird von der Methode `YES`[89] zurückgegeben, wird die Auflösung als erfolgreich betrachtet und die ursprüngliche Nachricht wird erneut in Stufe (1) behandelt[90]. Der Rückgabewert `NO` leitet zur nächsten Stufe über und entspricht der Standardimplementierung von `NSObject`.

Zur Auflösung von Nachrichten an Klassen und Instanzen existieren separate Schnittstellen, die beide als Klassenmethoden deklariert sind. Dies hat zur Folge, dass der genaue Empfänger der Nachricht in dieser Stufe nicht bekannt ist, sondern nur dessen Klasse.

[88] API-Dokumentation: `https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSInvocation_Class/Reference/Reference.html`

[89] `YES` und `NO` werden in Objective-C als Konstanten für die Boolean-Werte `1` und `0` verwendet

[90] Scheitert die Auflösung dort erneut, wird eine Exception geworfen – es kommt nicht zum erneuten Durchlauf aller Stufen

3. **Unmittelbare Weiterleitung**: Wird von der Schnittstelle dieser Stufe ein anderes Objekt als `nil`[91] oder `self`[92] zurückgegeben, wird die Nachricht an dieses Objekt unverändert weitergeleitet. Eine Manipulation der Nachricht, zum Beispiel der Argumente, ist nicht möglich. Die Standardimplementierung gibt `nil` zurück und führt zur Behandlung der Nachricht in der nächsten Stufe.

4. **Mittelbare Weiterleitung**: Wie in der vorangegangenen Stufe kann die Nachricht an ein anderes Objekt weitergeleitet werden. Als Entscheidungsgrundlage steht die vollständige Nachricht samt ihrer Parameter zur Verfügung, wobei die Nachricht vor der Weiterleitung beliebig manipuliert werden kann. Damit das Nachrichtenobjekt erstellt werden kann, muss zunächst `methodSignatureForSelector:` als Hilfsmethode implementiert werden. Im Gegensatz zu den vorherigen Stufen ist die Auflösung in dieser Stufe verhältnismäßig teuer, was in Abbildung 3.5 durch eine verstärkte Trennlinie impliziert wird.

5. **Absturz**: Wird die letzte Stufe erreicht, ist ein Absturz unvermeidlich. Die Schnittstelle kann dazu genutzt werden, Crash Reports oder Debugging-Informationen zu sammeln. Entgegen der in [See12, Kapitel 2 Nachrichten und Objekte] vertretenen Ansicht kann ein Absturz auch durch Überschreiben der Standardimplementierung nicht verhindert werden. In der Dokumentation wird der Entwickler daher dazu verpflichtet, eine `NSInvalidArgumentException` zu werfen [app13c].

Mithilfe der Instanzmethode `respondsToSelector:` von `NSObject` lässt sich feststellen, ob ein Objekt auf eine Nachricht reagieren kann. Die Prüfung umfasst dabei lediglich die Stufen 1 und 2. Falls eine Klasse und deren Instanzen den Eindruck erwecken sollen, auch auf Nachrichten zu reagieren, die in späteren Stufen behandelt werden, muss zusätzlich `respondsToSelector:` überschrieben werden [app09].

Die dynamische Auflösung von Nachrichten wird im Allgemeinen als Ausnahmefall betrachtet. Wird eine Methode von einer Klasse deklariert, aber nicht statisch implementiert, generiert der Compiler eine Warnung. Im Fall von Properties lässt sich diese Warnung unterdrücken, indem der Property im Implementierungsteil das Schlüsselwort `@dynamic` vorangestellt wird. Der Programmierer muss dafür Sorge tragen, dass die Getter- und Setternachrichten tatsächlich zur Laufzeit behandelt werden.

3.3.2 Der Build-Prozess in Xcode

Der Build-Prozess in Xcode ist in **Build-Phasen** unterteilt. Nutzt man die vorgegebenen Projekt-Templates, werden für eine iOS-App vier Build-Phasen angelegt, die in Abbildung 3.6 dargestellt sind.

In Phase 1, **Target Dependencies**, können Abhängigkeiten explizit definiert werden. Befinden sich die Abhängigkeiten im selben Workspace wie die zu bauende App, werden die Abhängig-

[91] `nil` entspricht prinzipiell einem Nullpointer, wobei Nachrichten an `nil` keine Auswirkung haben, insbesondere keinen Fehler auslösen

[92] `self` ist ein Pointer auf die eigene Instanz, vergleichbar mit `this` in Java

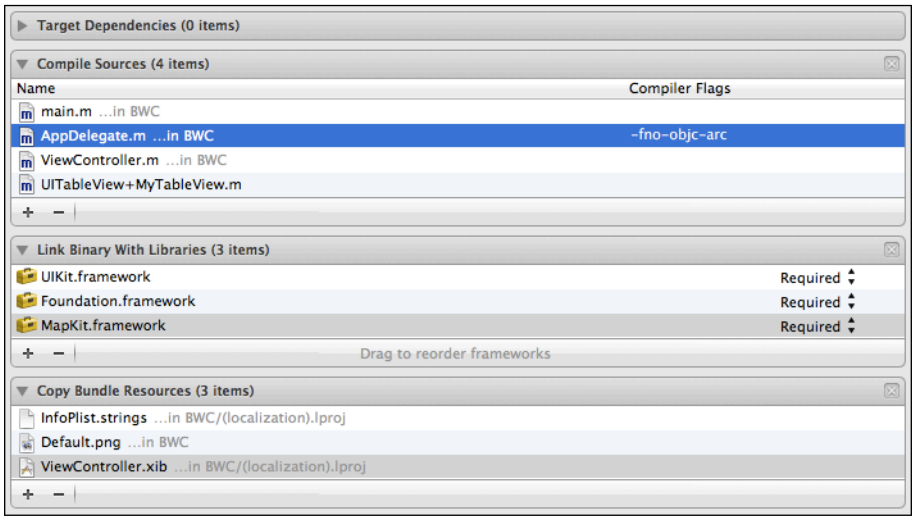


Abbildung 3.6: Build-Phasen in Xcode

keiten zuerst neu gebaut. Die Build-Reihenfolge unter den Abhängigkeiten wird von Xcode automatisch bestimmt und lässt keine zyklischen Abhängigkeiten zu. Abhängigkeiten zwischen Xcode-Projekten werden ausführlich in [Hag12] diskutiert.

Die **Compile-Sources**-Phase umfasst alle zu kompilierenden Quellcodedateien. Dabei können anwendungsweit definierte Compiler-Flags für einzelne Dateien überschrieben oder ergänzt werden. In Abbildung 3.6 ist zu erkennen, dass für die Datei `AppDelegate.m` das `-fno-objc-arc`-Flag gesetzt wurde, das die automatische Referenzzählung zur Speicherverwaltung (Automatic Reference Counting/*ARC*) deaktiviert.

Um Bibliotheken und Frameworks zu verlinken, dient Phase 3, **Link Binary with Libraries**. Ob die Bibliotheken statisch oder dynamisch verlinkt werden, hängt von der Art der Bibliothek ab. Da unter iOS keine eigenen dynamisch verlinkbaren Bibliotheken installierbar sind, sollten ausschließlich über das iOS SDK bereitgestellte dynamische Bibliotheken verlinkt werden. Zur Steigerung der Kompatibilität können Bibliotheken als *optional* gekennzeichnet werden. Die Folgen dieser Einstellung werden später in Abschnitt 3.4.2 geschildert.

Die letzte der vier Standardphasen, **Copy Bundle Resources**, kopiert Bilder, (lokalisierte) Strings-Dateien und andere Ressourcen in das ansonsten fertig gebaute App-Archiv. Für bestimmte Dateitypen lassen sich über die sogenannten *Build Rules* Programme festlegen, die diesen Kopiervorgang übernehmen und dabei manipulierend eingreifen können. Beispielsweise können Grafiken komprimiert und GUI-Dateien in ein Binärformat konvertiert werden.

Über die Build-Einstellungen (*Build Settings*) von Xcode lassen sich Parameter für die einzelnen Phasen festlegen, insbesondere Compiler- und Linker-Flags. Für die wichtigsten Einstellungen hält Xcode eine GUI mit symbolischen Namen anstelle der teils kryptischen Namen der Flags bereit. Nicht vorgegebene Compiler- und Linker-Flags lassen sich über die Einträge „Other C Flags"/„Other C++ Flags" bzw. „Other Linker Flags" setzen. Neben den Compileroptionen lässt sich auch der in der Xcode-Installation enthaltene Compiler selbst austauschen.

Die Reihenfolge der genannten Build-Phasen kann beliebig geändert werden, nicht benötigte Phasen können entfernt werden. Reicht die dadurch erlangte Flexibilität nicht aus, können auch eigene Skripte in den Build-Prozess eingebunden werden (**Skript-Build-Phasen**). Das Build-System *CMake*[93] nutzt diesen Umstand, um den Build-Prozess bestimmter Targets vollkommen auszulagern, wie aus Abbildung 3.7 ersichtlich ist. Lediglich die Auflösung der Abhängigkeiten wird Xcode überlassen, ein *Make*-Skript[94] ersetzt die übrigen Phasen.

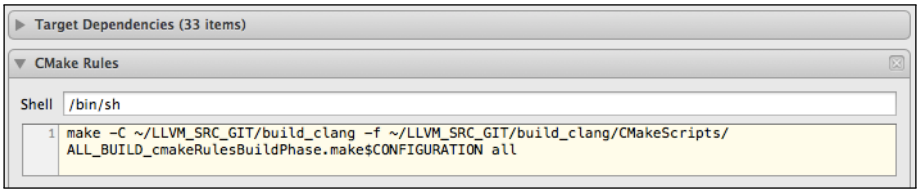


Abbildung 3.7: Build-Phasen eines von CMake generierten Xcode-Projekts

Umgekehrt lässt sich auch der Xcode-Buildvorgang über Skripte ansprechen und steuern, dazu kann der Befehl `xcodebuild`[95] verwendet werden. Alle Informationen zum Build-Vorgang und den dazugehörigen Phasen werden direkt in der Projektdatei (`.xcodeproj`-Datei[96]) gespeichert. Xcode bietet eine AppleScript-API[97] an, um Projektinformationen einfach auslesen zu können. Die API kann über die unter Mac OS X vorinstallierte Anwendung *AppleScript Editor* eingesehen werden, wie in Abbildung 3.8 gezeigt. Leider werden von Xcode 4 nicht alle dokumentierten AppleScript-APIs tatsächlich implementiert.

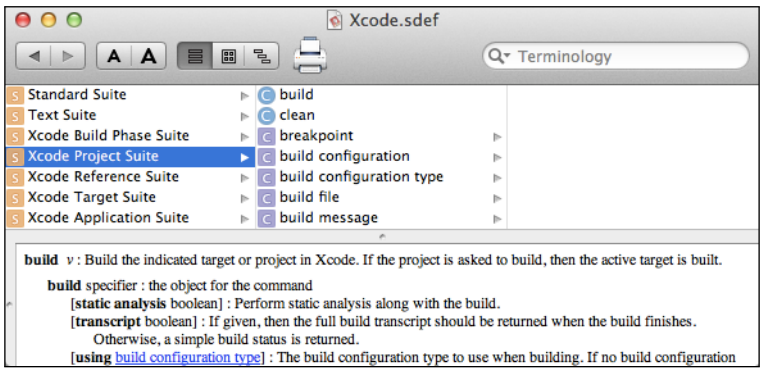


Abbildung 3.8: Die AppleScript-API von Xcode im AppleScript Editor

3.3.3 Die LLVM Compiler-Infrastruktur

Über viele Jahre hinweg war GNU GCC der Standardcompiler für C-basierte Sprachen auf der Mac-Plattform und daher auch die erste Wahl für das Kompilieren von iOS-Anwendungen. Der

[93] CMake erlaubt die Build-Konfiguration eines Projekts unabhängig von den verwendeten Build-Werkzeugen und Compilern; aus den Konfigurationsdateien generiert CMake ausführbare Build-Skripte oder IDE-Projektdateien: `http://www.cmake.org/`

[94] Make ist ein Build-Werkzeug, das den Kompilierungsvorgang mithilfe von Konfigurationsdateien („makefiles“) steuert und kontrolliert: `https://www.gnu.org/software/make/`

[95] Vor der Verwendung muss Xcode samt den *Command Line Tools* installiert werden: `https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/xcodebuild.1.html`

[96] Genau genommen handelt es sich dabei um ein Bundle, das mehrere Dateien enthält

[97] `https://developer.apple.com/library/mac/#documentation/applescript/conceptual/applescriptx/AppleScriptX.html`

Wandel hin zu einem moderneren Compiler wurde 2008 mit dem Release von Xcode 3.1 eingeläutet. Seit Xcode 4.2 wird die Codegenerierung durch GCC nicht mehr unterstützt[98], ab der kommenden Version wird Xcode vollständig auf LLVM setzen [xco13]. Um diesen schrittweisen Übergang von GCC zu LLVM besser nachzuvollziehen, sollen einige grundlegende Aspekte der Compiler-Architektur erklärt werden.

Die Komponenten eines Compilers können auf vielfache Weise kategorisiert werden. Die gröbste Unterteilung ist die in Front- und Backend, wie in Abbildung 3.9 dargestellt. Die Hauptaufgabe des **Frontends** ist das Parsen des Quellcodes, wobei in der Regel eine Baumstruktur (Abstract Syntax Tree/**AST**) des gesamten Programms erstellt wird. Auf der Basis des AST können programmiersprachenspezifische Analyse- und Optimierungsverfahren eingesetzt werden. Anschließend wird der AST in eine Zwischenrepräsentation (Intermediate Representation/**IR**) umgewandelt, die dem **Backend** als Eingabe dient. Aus der IR eines Programms wird letztlich der ausführbare Maschinencode generiert, wobei zuvor eine programmiersprachenunabhängige, häufig aggressive Optimierung durchlaufen wird.

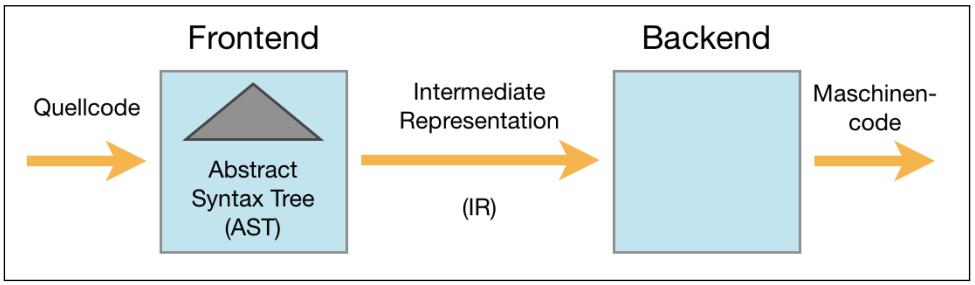


Abbildung 3.9: Frontend und Backend eines Compilers

Die Entwicklung von **GCC** wurde in den 1980er Jahren von Richard Stallman aus der Notwendigkeit, einen frei verfügbaren Compiler als Teil des GNU-Projekts ausliefern zu können, begonnen [Sta11]. Zahlreiche Entwickler[99] unterstützen die Weiterentwicklung des quelloffenen Compilers und machten GCC zum Defacto-Standard für C-basierte Programmiersprachen. Aus einer vollkommen anderen Motivation begann die **LLVM**-Entwicklung im Jahr 2000. Wie u. a. in [Lat06] beschrieben, ist LLVM selbst kein Compiler, sondern eine Infrastruktur, die **Module** und Bibliotheken zur Entwicklung von Compilern bereitstellt und grundsätzlich eher dem Backend zuzuordnen. Dabei sollen die Grundmodule unabhängig von Quellcodesprachen, Zielarchitekturen und Compilertypen[100] sein, um die Wiederverwendbarkeit zu maximieren. Die Unterstützung konkreter Sprachen und Plattformen kann durch separate Module realisiert werden. Neben der Unabhängigkeit von Quell- und Zielsprachen sind Effizienz und die Nutzung moderner Optimierungsalgorithmen wichtige Ziele der LLVM-Komponenten [Lat06].

In [Lat07] wird näher ausgeführt, wie sich aus den LLVM-Modulen ein fertiger Compiler konstruieren lässt. Existiert bereits ein Frontend zum Einlesen einer bestimmten Quellcodesprache, muss lediglich eine Transformation des frontendspezifischen AST in die LLVM-spezifische

[98] Diese und die folgenden Versionsangaben beziehen sich auf die in der Xcode- oder Betriebssysteminstallation enthaltenen Compiler; die Einbindung anderer Compiler ist prinzipiell möglich, jedoch kann die Kompatibilität nicht garantiert werden

[99] `http://gcc.gnu.org/onlinedocs/gcc/Contributors.html`

[100] Die LLVM-Module sollen sowohl für klassische, statische Compiler als auch für Just-in-Time-Compiler (JIT) und Interpreter von Nutzen sein

IR implementiert werden. Auf diese Weise ist der **llvm-gcc**-Compiler entstanden – für das altbewährte GCC-Frontend wurde ein Adapter geschrieben, der den GCC-AST in die IR von LLVM umwandelt, sodass die Codegenerierung von LLVM übernommen werden kann. Dieser, in Abbildung 3.10 illustrierte Ansatz erzielt eine maximale Kompatibilität zu GCC, da IDEs und andere Build-Systeme ausschließlich mit den Compiler-Frontends interagieren. Auch das LLVM-Backend wurde mit Rücksicht auf GCC-Kompatibilität entwickelt, sodass es möglich ist, binärkompatiblen Code zu generieren. Der llvm-gcc-Compiler wurde erstmalig mit Xcode 3.1 ausgeliefert.

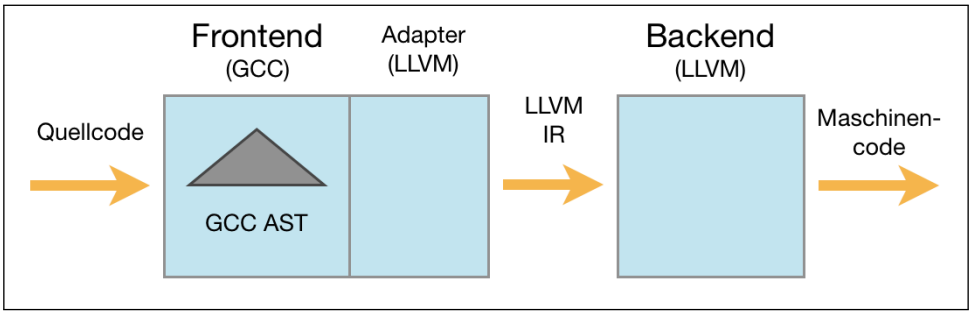


Abbildung 3.10: Architektur des llvm-gcc-Compilers

Unterdessen wurde im Rahmen des LLVM-Projekts auch an einem eigenen C-Frontend gearbeitet, das unter dem Namen **Clang**[101] veröffentlicht wurde. Genau wie das LLVM-Backend besteht Clang aus einer Reihe von Modulen, die minimale gegenseitige Abhängigkeiten besitzen und dadurch eine hohe Wiederverwendbarkeit, auch für andere Projekte, besitzen. Ein auf dem Clang-Frontend und dem LLVM-Backend basierender Compiler wurde mit Xcode 3.2 unter dem Namen *Apple LLVM Compiler* eingeführt[102], seit Xcode 4.2 ist dies der Standardcompiler in Xcode[103]. Zwar ist llvm-gcc bis einschließlich Xcode 4.6 verfügbar, einige von Apple initiierte Objective-C-Erweiterungen sind aber nur mit dem Clang-Frontend nutzbar [obj12].

Die Modularität des Frontends ist wichtig für eine **konsistente Codeanalyse** innerhalb einer IDE. Ist ein Compiler monolithisch aufgebaut und nur als Ganzes verwendbar, muss eine IDE eigene Parser implementieren, um Quellcode außerhalb des Kompilierungsvorgangs zu analysieren. So verfügt Xcode 3 über einen eigenen C-Parser, um Syntax-Highlighting und andere Grundfunktionen des Quellcodeeditors zu implementieren. Durch die Verwendung verschiedener Parser innerhalb einer IDE kann es zu widersprüchlichen und verwirrenden Warnungen und Fehlern kommen [Kre10]. Xcode 4 nutzt zur Quellcodeanalyse die selben Clang-Module, die auch der Clang-Compiler verwendet und erreicht damit eine konsistente Ausgabe von Warnungen und Fehlern.

LLVM und Clang sind **Open-Source**-Projekte, wobei Mitarbeiter namhafter Firmen wie ARM, Intel, Google und Apple den Großteil der Weiterentwicklung betreiben. Die liberale Lizenz[104] erlaubt es Apple, Clang direkt als Teil von Xcode auszuliefern, obwohl Xcode selbst nicht quelloffen ist. Der große und direkte Einfluss auf das Clang-Projekt ermöglicht es Apple zudem, Spracherweiterungen innerhalb kürzester Zeit einzuführen und einer breiten Anwenderbasis zur Verfügung zu

[101] `http://clang.llvm.org/`
[102] Der Clang-Compiler ist auch unabhängig von Xcode nutzbar und unterscheidet sich nur minimal von der mit Xcode ausgelieferten Version
[103] Gleichzeitig wurde der Support für den GCC-Compiler (nicht llvm-gcc) eingestellt
[104] The University of Illinois/NCSA Open Source License (NCSA): `http://opensource.org/licenses/UoI-NCSA.php`

stellen. Die wichtigsten, compilergestützten Neuerungen können in [obj12] eingesehen werden, am meisten Aufmerksamkeit erlangte die automatische Speicherverwaltung ARC[105].

3.3.4 ABI-Kompatibilität

Die ABI-Kompatibilität zwischen einem Framework und den darauf aufbauenden Anwendungen kann auf vielfältige Weise gebrochen werden. Am offensichtlichsten ist dies der Fall, wenn die Menge der APIs reduziert wird. In bereits kompilierten Anwendungen kommt es zu Laufzeitfehlern, wenn Funktionen nicht gefunden werden oder ganze Klassen fehlen[106]. Weit weniger offensichtlich ist das in [MS98] beschriebene **Fragile Base Class Problem**, das eigentlich ein Compilerproblem ist [FCDR95]. Es kann auftreten, wenn eine Anwendungsklasse von einer Frameworkklasse erbt und die Frameworkklasse (Basisklasse) in einer späteren Version erweitert wird. Hinsichtlich der API-Kompatibilität ist eine Erweiterung der Schnittstelle kein Problem. Nutzt der Compiler jedoch die Speichergröße der Basisklasse für Adressberechnungen in der Unterklasse, treten Fehler auf, wenn sich die Größe der Basisklasse ändert, ohne dass die Unterklasse neu kompiliert wird.

In [FCDR95] wurden konkrete Fälle des Fragile Base Class Problems für verschiedene Programmiersprachen auf dem Stand von 1995 betrachtet, darunter auch Objective-C. Die Grundlage aller ABI-Kompatibilitätsprobleme in Objective-C war die Anordnung der Instanzvariablen, woraus die Bezeichnung der **Fragile Ivars** entstanden ist. In Abbildung 3.11 ist die Ausgangslage grafisch dargestellt, beschrieben wird die Problematik u. a. in [Gal10].

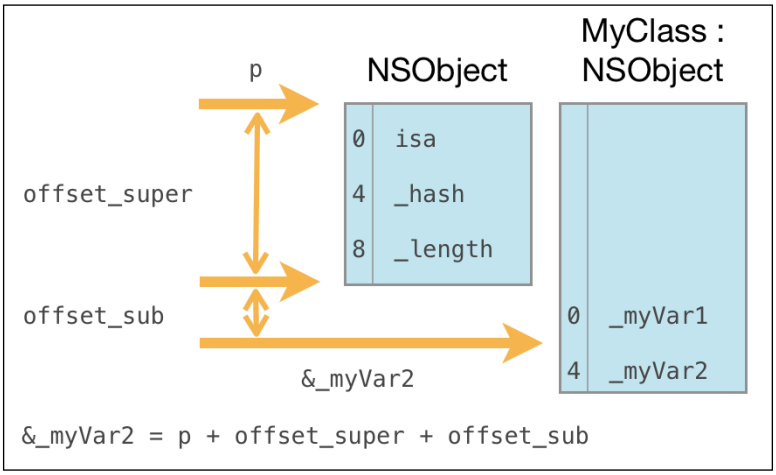


Abbildung 3.11: Berechnung der Adresse einer Instanzvariablen der Unterklasse

Die Instanzvariablen einer Unterklasse folgen im Speicher auf die Instanzvariablen der Basisklasse. Um die Adresse einer Instanzvariablen der Unterklasse zu berechnen, wird die Größe der Region aller Instanzvariablen der Basisklasse benötigt, in Abbildung 3.11 mit `offset_super` bezeichnet. Anschließend wird der relative Offset (`offset_sub`) der Variablen in der Unterklasse zum Offset der Basisklasse addiert. Um die absolute Adresse zu ermitteln, wird schließlich der

[105] `http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/`
[106] Davon abgesehen, und im Gegensatz zu den im Folgenden diskutierten Problemen, schlägt in diesem Fall auch eine Rekompilierung fehl, da die API-Kompatibilität gebrochen wurde

Pointer auf das Objekt ausgewertet. In früheren Versionen von Objective-C wurde `offset_super` zur Compile-Zeit der Unterklasse ermittelt und stellte somit eine Konstante dar. Wird das Variablenlayout der Basisklasse in einer späteren Frameworkversion erweitert, ergibt sich das in Abbildung 3.12 dargestellte Problem.

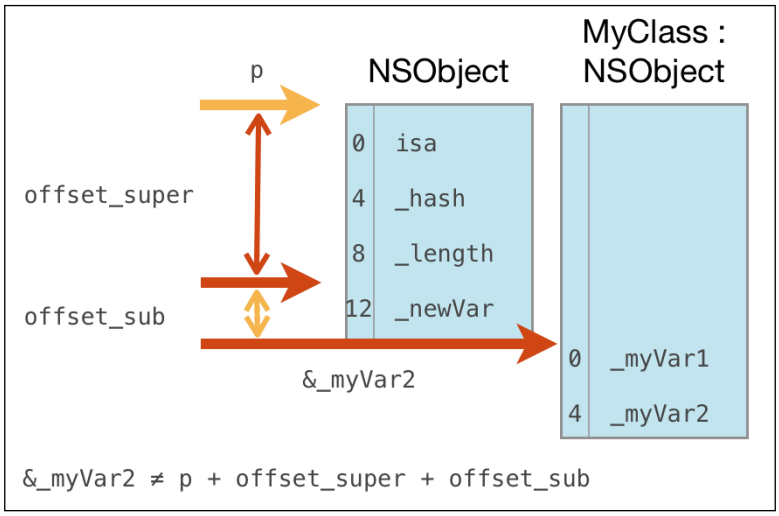


Abbildung 3.12: Berechnung der Adresse einer Instanzvariablen der Unterklasse nach Erweiterung der Basisklasse

Durch die Erweiterung der Basisklasse entspricht `offset_super` nicht mehr der tatsächlichen Größe der Variablenregion, wodurch auch die Folgeberechnung fehlerhaft ist, obwohl der relative Offset (`offset_sub`) korrekt ist.

Mit der Einführung der sogenannten *Modern Runtime* und *Objective-C 2.0* wurden **Non-Fragile Ivars** eingeführt. Die Runtime-Bibliothek stellt eine Funktion zur Verfügung, um den Variablenbereich einer Klasse zur Laufzeit und damit aktuell zu ermitteln. Der Compiler muss dafür Sorge tragen, dass diese Funktion anstelle der `offset_super`-Konstanten zur Adressberechnung verwendet wird. Für den Clang-Compiler lässt sich die ABI über das `-fobjc-abi-version`-Flag explizit steuern, wobei die non-fragile ABI dem Standardwert entspricht und die iOS-Plattform ausschließlich die moderne Runtime-Umgebung bereitstellt. Non-Fragile Ivars ermöglichen nicht nur dem Frameworkentwickler, das Variablenlayout gefahrenlos zu verändern, sondern erlauben auch den Anwendungsentwicklern, Instanzvariablen zur Laufzeit hinzuzufügen[107]. Insbesondere kann komplett darauf verzichtet werden, Variablen in den öffentlich sichtbaren Header-Dateien zu deklarieren, da der Compiler diese Information nicht benötigt.

Objective-C-Methoden sind von der Problematik des Fragile Base Class Problems grundsätzlich nicht betroffen, weil der Aufruf von Methoden über einen in Abschnitt 3.3.1 näher beschriebenen Mechanismus dynamisch zur Laufzeit erfolgt. Insgesamt lässt sich sagen, dass die hohe Laufzeitdynamik von Objective-C dazu führt, dass das Fragile Base Class Problem praktisch nicht (mehr) existent ist und die Herstellung der ABI-Kompatibilität verglichen mit anderen kompilierten Programmiersprachen unproblematisch ist.

[107] Dies ist nur möglich, bevor eine Klasse erstmals benutzt wird

3.3.5 Die App Store Rahmenbedingungen

Nutzer unmodifizierter iOS-Endgeräte können ihre Anwendungen nur über den Apple App Store beziehen. Im Umkehrschluss bedeutet dies, dass der App Store für Entwickler die einzige Möglichkeit darstellt, Anwendungen zu vertreiben. Um Apps in den App Store einreichen zu können, müssen Entwickler dem kostenpflichtigen iOS Developer Program beitreten. Jede eingereichte App wird vor der Freischaltung von Apple geprüft, wobei die zugrundeliegenden Kriterien in den App Store **Review Guidelines** [sto13] zusammengefasst sind.

Viele der Regeln bedürfen keiner Erklärung. So behält sich Apple vor, abstürzende ([sto13, Guideline 2.1]), fehlerhafte ([sto13, Guideline 2.2]) oder nicht der Beschreibung entsprechende ([sto13, Guideline 2.3]) Apps abzulehnen. Zudem dürfen Apps ausschließlich auf öffentliche, dokumentierte APIs zugreifen ([sto13, Guideline 2.5]). Fraglich ist, ob davon auch APIs betroffen sind, deren Funktionalität zwar dokumentiert ist, von deren Benutzung aber an anderer Stelle abgeraten wird. In dieser Arbeit sind drei APIs und Techniken von Interesse, deren Konformität mit den Review Guidelines geklärt werden muss. Am Ende eines jeden Unterpunkts wird angegeben, in welchen Kapiteln der Einsatz der API oder Technik erwogen wird.

Überschreiben von Methoden mittels Categories

Wie in Abschnitt 3.1.1 beschrieben, lassen sich Frameworkklassen mit Categories nicht nur erweitern, sondern auch manipulieren. Wenngleich von dieser Praxis von offizieller Seite abgeraten wird [Mas10, TB11, app12d], konnten keine Anzeichen oder Entwicklerberichte[108] gefunden werden, die einen Konflikt dieser Vorgehensweise mit den Review Guidelines bescheinigen. Ungeachtet dessen müssen die technischen Risiken dieses Ansatzes im Einzelfall gut abgewogen und begründet werden.

Relevanz: Abschnitt 5.2.3

Method Swizzling

Um bestehende Klassen mittels Categories zu manipulieren, aber dennoch die Originalimplementierung zu erhalten, kann Method Swizzling eingesetzt werden. Auch von der Anwendung dieser Technik, die bereits in Abschnitt 2.2.2 erläutert wurde, wird von offizieller Seite abgeraten [TB11], wobei Method Swizzling in der genannten Quelle in einem anderen Kontext betrachtet wird[109]. In Entwicklerforen kursieren Meldungen über die Ablehnungsankündigung seitens Apple aufgrund der Verwendung von Method Swizzling [ult12], allerdings wird dabei Bezug auf eine konkrete API genommen, die nicht mehr benutzt werden soll[110]. Demnach scheint der Ablehnungsgrund vielmehr die betroffene API als Method Swizzling an sich zu sein. Die quelloffene Netzwerk-Bibliothek AFNetworking[111], die in zahlreichen App-Store-Anwendungen benutzt wird,

[108] Eine Sammlung von Ablehnungsgründen wird z. B. unter `http://appreview.tumblr.com/` geführt

[109] Die Quelle beschreibt Method Swizzling als Möglichkeit, die Funktionalität von Frameworkklassen zu überschreiben, und nicht, die Funktionalität bei der Nutzung von Categories zu erhalten

[110] Durch die Einführung von ARC sollte `dealloc` nicht mehr aufgerufen werden, sofern ARC aktiviert ist

[111] `https://github.com/AFNetworking/AFNetworking/`

nutzt Method Swizzling und dient als Beweis, dass die Verwendung dieser Technik prinzipiell mit den Review Guidelines kompatibel ist [Tho13]. Zur Implementierung von Method Swizzling werden ausschließlich die öffentlichen, unter Abschnitt 3.3.1 beschriebenen APIs benötigt.

Relevanz: Abschnitt 2.2.2, Abschnitt 5.2.3

Runtime-Bibliothek: Manipulation der Superklasse

Die in der Runtime-Bibliothek zusammengefassten APIs sind mächtig, erfordern aber auch ein fundiertes Grundwissen über Objective-C und dessen Laufzeitumgebung. Für einen im späteren Teil dieser Arbeit diskutierten Implementierungsansatz ist die `class_setSuperclass()`-Funktion von Interesse. Obwohl es sich dabei um eine öffentliche API handelt, ist die App-Store-Kompatibilität ungewiss, da in der Dokumentation die eindeutige Anmerkung „You should not use this function" zu finden ist [app10a]. Ob dies einen direkten Einfluss auf den Review-Prozess hat ist fraglich, von den drei hier diskutierten Beispielfällen enthält dieser jedoch das deutlichste Indiz, von einer Verwendung abzusehen. Zudem ist die Funktion als `deprecated` markiert und steht in zukünftigen Versionen voraussichtlich nicht mehr zur Verfügung.

Relevanz: Abschnitt 4.5

Da keiner der Fälle direkt durch die Review Guidelines abgedeckt ist, kann über eine Entscheidung im Review-Prozess nur spekuliert werden. Aus den dargelegten Gründen sollten der Einsatz von Method Swizzling und das Überschreiben von Methoden mittels Categories keine negativen Auswirkungen auf den Review-Prozess haben, solange die Performance oder Stabilität der Anwendung nicht negativ beeinflusst wird. Absolute Sicherheit kann allerdings nur eine Reihe von Testläufen durch den Review-Prozess bringen.

Unabhängig vom Review-Prozess bringt der App Store die Einschränkung mit sich, pro Geräteklasse (iPad und iPhone/iPod) nur eine Version einer App gleichzeitig anbieten zu können. Nicht untertützt wird die Möglichkeit, verschiedene Apps für verschiedene Betriebssystemversionen anzubieten. Die Folge davon ist, dass ein gebautes App-Archiv sämtliche abzudeckenden Betriebssystemversionen unterstützen muss. Alte Versionen einer App können vom Nutzer lokal gespeichert und wiederhergestellt, aber nicht zu einem späteren Zeitpunkt erneut über den App Store bezogen werden.

3.4 ANWENDBARKEIT BESTEHENDER ANSÄTZE

Im Grundlagen-Kapitel wurden unter Abschnitt 2.2.2 verschiedene Methoden betrachtet, wie die Detektion und Auflösung kritischer APIs auf anderen Plattformen und in Forschungsarbeiten gehandhabt wird. Bestehende Ansätze für iOS und Objective-C wurden dabei bewusst ausgeklammert, um sie in den folgenden Abschnitten zu diskutieren. Für jeden in der Zusammenfassung aus Abschnitt 2.3 enthaltenen Ansatz soll geprüft werden, ob und unter welchen Voraussetzungen der Ansatz unter iOS bereits unterstützt wird, oder mit welchem Aufwand eine Umsetzung

denkbar wäre. Zur einfachen Nachvollziehbarkeit werden die Zusammenfassungen aus Kapitel 2 jeweils einleitend zitiert.

3.4.1 Detektion kritischer APIs

Zunächst muss geklärt werden, ob auf der iOS-Plattform die Voraussetzungen erfüllt sind, um die Menge der kritischen APIs eindeutig zu bestimmen. In Abschnitt 2.1.2 wurden dazu vier Vorbedingungen aufgestellt:

- ✓ *Die Menge aller APIs ist klar definiert und bekannt:* Sämtliche öffentliche APIs des iOS SDKs sind sowohl in Header-Dateien als auch in einer Online-Dokumentation[112] aufgelistet und beschrieben.
- ✓ *Es werden nur definierte und bekannte APIs genutzt:* Um mit den in Abschnitt 3.3.5 diskutierten App Store Review Guidelines konform zu gehen, dürfen nur öffentliche APIs verwendet werden.
- ✓ *Eine Plattformversion ist durch eine Versionsnummer eindeutig bestimmt:* Die Plattformversion entspricht unter iOS der Version des SDKs[113] und ist durch eine Major-, Minor- und Subminor-Komponente eindeutig bestimmt.
- ✓ *Zu jeder API ist bekannt, auf welchen Versionen der Plattform sie verfügbar ist:* Wurde eine API nach dem initialen SDK-Release (Version 2.0) eingeführt, verfügt sie in der entsprechenden Header-Datei über Attribute[114], die über die Mindestversion informieren. Wurde ein Framework als Ganzes neu eingeführt, fehlt diese Information u. U. in den Header-Dateien, kann aber in der Apple-Dokumentation[115] gefunden werden.

Darauf aufbauend müssen für jedes iOS-Projekt sowohl das Deployment Target[116] als auch das Base SDK[117] gesetzt werden, wobei der Standardwert für beide Einstellungen die neuste verfügbare Version ist. Es lässt sich somit festhalten, dass die in Abschnitt 2.1.2 aufgestellte Definition einer kritischen API unter iOS problemlos anwendbar ist.

Automatische Detektion

> *„Lint: Werkzeug zur vollautomatischen Erkennung kritischer APIs in Android-Projekten"*

Weder in Xcode noch in Clang oder dem Static Analyzer wurde bislang die Funktionalität implementiert, auf kritische APIs im Quellcode hinzuweisen. Die alternative Entwicklungsumgebung

[112] `http://developer.apple.com/library/ios/navigation/#section=Frameworks`
[113] Mit Ausnahme der Subminor-Version, die keinen Einfluss auf das SDK hat
[114] `http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html`
[115] `https://developer.apple.com/library/IOS/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html`
[116] Compiler-Flag: `IPHONEOS_DEPLOYMENT_TARGET`
[117] Compiler-Flag: `SDKROOT`

JetBrains **AppCode**[118] weist auf kritische APIs dagegen standardmäßig hin, legt die Warnungen allerdings sehr streng aus. In Abbildung 3.13 ist eine solche Warnung dargestellt.

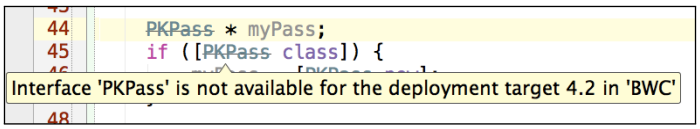


Abbildung 3.13: Kritische API in AppCode

Sowohl die textuelle als auch die visuelle Darstellung (Durchstreichen der API) suggerieren, dass die API in diesem Projekt prinzipiell nicht verwendet werden darf. Hinzu kommen einige Detailprobleme der Umsetzung. Zwar ist die in der Abbildung dargestellte Klasse unter iOS 4.2 tatsächlich nicht vorhanden, der `class`-Selektor (Abbildung 3.13, Zeile 45) kann jedoch trotzdem verwendet werden, um die Verfügbarkeit einer Klasse zu überprüfen, genau wie die bloße Referenzierung einer Klasse (Abbildung 3.13, Zeile 44) keinerlei Laufzeitprobleme verursacht[119]. Die Warnungen lassen sich global oder für bestimmte Dateien einer Anwendung deaktivieren, allerdings besteht keine dokumentierte Möglichkeit, die Warnungen für einzelne Codezeilen oder -bereiche zu unterdrücken. Die Ausgabe der Warnungen erscheint nicht im Build-Log[120] und steht ausschließlich AppCode selbst zur Verfügung, kann also nicht von anderen Werkzeugen (beispielsweise zur Auflösung kritischer APIs) weiterverwendet werden. Für Entwickler, die AppCode der Standard-IDE Xcode vorziehen, können die zusätzlichen Warnungen eine nützliche Hilfe sein, andererseits auch zu Trugschlüssen bei der Auflösung führen.

Während der Anfertigung dieser Diplomarbeit wurde das Werkzeug **Deploymate**[121] veröffentlicht, das ausschließlich die Detektion kritischer APIs zum Ziel hat. Es setzt auf den Clang-Bibliotheken auf, ist aber wie auch AppCode kommerzielle Software und für Dritte nicht modifizier- oder erweiterbar. Aufgrund der hohen Spezialisierung werden einige Problem- und Sonderfälle von Deploymate besser unterstützt als in AppCode, beispielsweise ist Deploymate der Weak-Linking-Mechanismus bekannt und `class`-Nachrichten werden nicht als problematisch markiert. Andere kritische APIs werden derzeit nicht gefunden, dazu gehören `performSelector:`-Aufrufe kritischer Methoden, Unterklassen kritischer Klassen und Kompatibilitätsprobleme in XIB-Dateien. Die Neuheit des Werkzeugs lässt auf baldige Aktualisierungen hoffen, die teilweise bereits konkret angekündigt worden sind[122], darunter die Möglichkeit, einzelne Warnungen mit `#pragma`-Anweisungen zu unterdrücken [Vas13]. Deploymate ist nicht in Xcode integriert, der Nutzer muss das Werkzeug separat starten und die kritischen APIs später in Xcode selbst auflösen.

Manuelle Detektion mithilfe der Dokumentation

> „*API-Dokumentation: Information über die mindestens benötigte Plattformversion für jede API; Filtermöglichkeiten*“

[118] `http://www.jetbrains.com/objc/` – die folgenden Erkenntnise wurden durch die Arbeit mit AppCode 2.1 gewonnen
[119] Grund dafür ist der Weak-Linking-Mechanismus [app06]
[120] Angesichts der Tatsache, dass AppCode den Build-Prozess komplett an das Apple-Werkzeug `xcodebuild` auslagert, ist dies keine Überraschung
[121] `http://www.deploymateapp.com/` – die folgenden Erkenntnise wurden durch die Arbeit mit Deploymate 1.1.4 gewonnen
[122] Im FAQ des Werkzeugs werden einige mögliche Features gelistet: `http://www.deploymateapp.com/faq/`

Die SDK-Dokumentation kann auf vielfältige Weise eingesehen werden und enthält **Verfügbarkeitsinformationen** für sämtliche APIs. Außerhalb von Xcode steht die Dokumentation online und offline[123] im HTML-Format zur Verfügung. Xcode selbst bietet einen eigenen Browser für die Dokumentation an und fasst die wichtigsten Informationen im Quick Help Inspector direkt neben dem Quellcodeeditor zusammen, wie in Abbildung 3.14 dargestellt.

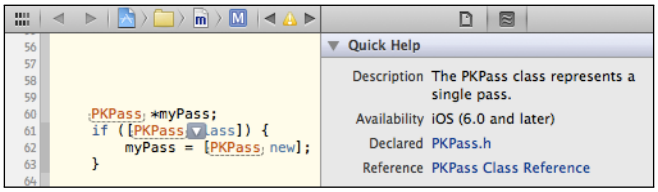


Abbildung 3.14: Kurzdokumentation in Xcode

Neben der aktuellen stehen auch alle vorherigen Versionen der API-Dokumentation zum Download zur Verfügung. Übertragen werden die Dokumentationspakete über das Atom-Format[124], sodass eine Einbindung in eigene Werkzeuge denkbar ist. Eine Filterfunktion nach der Verfügbarkeit von APIs fehlt in allen Darstellungsformen der Dokumentation.

> *„API Diffs: Ergänzung zur Dokumentation, indem der Änderungsverlauf in den Vordergrund gestellt wird"*

Mit jeder Aktualisierung des SDKs veröffentlicht Apple die API Diffs online im HTML-Format[125]. Der Vergleich erfolgt jeweils zur direkten Vorgängerversion. Die API Diffs sind Teil der SDK-Dokumentation und können somit im Xcode-Browser angezeigt und über den Atom-Feed geladen werden.

Sonstige Detektionsmöglichkeiten

Die Verfügbarkeitsinformationen in den Header-Dateien der Frameworks werden über Attribute bereitgestellt. Die Attribute sind jedoch nicht fest in den Quellcode geschrieben, sondern werden über Makros (**Availability Macros**) eingesetzt. In [mur11a] ist ein Ansatz geschildert, der ausnutzt, dass Xcode zwar keine kritischen, aber veraltete APIs markiert. Vor dem Import der Frameworks werden die Makros so überschrieben, dass sämtliche kritischen APIs mit `deprecated`-Attributen versehen werden. Dieser Ansatz ist offensichtlich unsauber und die Makro-Definitionen sollten vor dem Bauen der Anwendung stets entfernt werden[126], er bietet jedoch eine schnelle und praktikable Möglichkeit, alle kritischen APIs innerhalb Xcodes aufzuzeigen. Ein in [app03] beschriebener Mechanismus basiert ebenfalls auf Makros und teilt die gleichen Vor- und Nachteile.

[123] Um die gesamte Dokumentation herunterzuladen kann Xcode verwendet werden
[124] Spezifikation des Atom Syndication Formats: `http://tools.ietf.org/html/rfc4287/`
[125] API Diffs zwischen iOS 6.0 und iOS 6.1:
`https://developer.apple.com/library/ios/#releasenotes/General/iOS61APIDiffs/index.html`
[126] Die Verfügbarkeit der Symbole kann Einfluss auf den Link-Vorgang haben, wie unter Abschnitt 3.4.2 geschildert ist

3.4.2 Auflösung kritischer APIs

Auflösung zur Compile-Zeit

> *„Makros: Präprozessor-Makros, um inkompatiblen Code vor dem Kompilieren zu entfernen"*

Die Programmiersprache C, und damit auch Objective-C, unterstützt Präprozessormakros, wodurch inkompatibler Code zur Compile-Zeit aus einer Anwendung entfernt werden kann. Dieser Ansatz ist nicht mit dem Apple App Store kompatibel, da für verschiedene iOS-Versionen auch verschiedene Apps gebaut und später bereitgestellt werden müssten (s. Abschnitt 3.3.5). Deshalb wird dieser Ansatz im weiteren Verauf dieser Arbeit nicht beachtet.

Auflösung zur Laufzeit durch explizite Abfragen

> *„Reflection: Nutzung von APIs, die nicht im Base SDK enthalten sind und nicht vom Compiler geprüft werden"*

Die Runtime-Bibliothek von Objective-C erlaubt es, Klassen zu benutzen, die zur Compiler-Zeit nicht bekannt sind. In Codebeispiel 3.7 wird gezeigt, wie das Laden einer solchen Klasse umgesetzt werden kann.

```
1  // Variable des generischen Typs 'id':
2  // Die Klasse ist zur Compile-Zeit nicht bekannt und kann nicht als Typ
3  // verwendet werden
4  id instance = nil;
5  // Versuch, die Klassendefinition zu laden
6  Class theClass = NSClassFromString(@"ClassName");
7  if (theClass) {
8      // Klasse konnte geladen werden: Instanziierung
9      instance = [theClass new];
10 }
11
12 // Benutzung der Instanz
13 [instance performSelector:@selector(doSomething)];
```

Codebeispiel 3.7: Dynamisches Laden und Benutzen einer Klasse in Objective-C

Die Syntax zur Benutzung der dynamisch geladenen Klasse (Abbildung 3.7, Zeile 13) weicht von der Standardnachrichtensyntax ab, weil die Schnittstelle zur Compile-Zeit nicht bekannt ist. Darunter leidet die Lesbarkeit des Quellcodes und die Fehleranfälligkeit erhöht sich aufgrund der fehlenden Compilerprüfung. Daher sollte auf diesen Ansatz nur in Ausnahmefällen zurückgegriffen werden. Er wird in den folgenden Kapiteln nicht zum Tragen kommen, da er der grundsätzlichen Idee und Motivation dieser Arbeit, stets das neuste SDK zu verwenden, widerspricht.

„Konditionaler Code: Nutzung einer API nur nach Abfrage einer Version oder einer Funktionalität"

Die Abfrage der Version der aktuellen Laufzeitplattform wird vom UIKit-Framework unterstützt, wie in Codebeispiel 3.8 demonstriert ist.

```
1 float version = [UIDevice currentDevice].systemVersion.floatValue;
2 if (version >= 5.0f) {
3     // Code, der nur auf iOS 5+ ausgeführt werden soll
4 }
```

Codebeispiel 3.8: Abfrage der aktuellen iOS-Version

Alternativ kann die Verfügbarkeit von Klassen, Methoden und Funktionen explizit zur Laufzeit erfragt werden. Codebeispiele dazu werden in den Best Practices in Abschnitt 3.4.3 nachgereicht.

Auflösung in einer Zwischenschicht

„Abstraktionsschicht: Nutzung von Design Patterns, um verschiedene Implementierungen über ein einheitliches Interface ansprechen zu können"

Eine Abstraktionsschicht ist im Allgemeinen unabhängig von Programmiersprachen und daher ohne Weiteres auf der iOS-Plattform umsetzbar. Die Einbindung einer solchen Abstraktionsschicht wird am Ende dieses Unterabschnitts diskutiert.

„Support Libraries: Nachimplementierung und Imitierung von Framework-Klassen durch statisch verlinkbare Klassen"

Apple stellt ausschließlich die dynamisch verlinkbaren Frameworks zur Verfügung, statisch verlinkbare Teilimplementierungen analog zu den Android Support Libraries sind nicht verfügbar. Aus technischer Sicht spricht jedoch nichts gegen die Erstellung von Support Libraries, für einzelne Frameworkklassen wurden ähnliche Ansätze von Dritten bereits veröffentlicht[127].

„Google Play Services: Konzeptionell ähnlich zu dynamischen Bibliotheken, die sich gekapselt als App auf verschiedenen Plattformversionen (nach)installieren lassen"

Eine vergleichbare Anwendung wird von Apple nicht angeboten. Die eingeschränkten Kommunikationsmöglichkeiten zwischen iOS-Apps untereinander sprechen gegen eine Implementierung einer solchen Service-App durch Dritte, auch die direkte Installation dynamischer Frameworks ist auf unmodifizierten Endgeräten nur Apple möglich.

[127] Für die mit iOS 6 eingeführte Klasse `UICollectionView` existiert eine Nachimplementierung, die bis iOS 4.3 kompatibel ist: `https://github.com/steipete/PSTCollectionView/`

„AOP: Vermeidung von verteiltem konditionalen Code durch die zentrale Definition von Aspekten; Implementierung programmiersprachenabhängig auf der Sender- oder Empfängerseite"

Anstatt den Anwendungscode direkt gegen eine Abstraktionsschicht oder Support Library zu schreiben, kann AOP dazu verwendet werden, den eigentlichen Anwendungscode unverändert zu lassen und die kritischen Frameworkaufrufe transparent zu einer Zwischenschicht umzuleiten. Weder für C noch für Objective-C gibt es etablierte AOP-Technologien, die mit AspectJ für Java vergleichbar wären, aber kleinere Projekte und Forschungsarbeiten existieren. Alle bekannten AOP-Technologien für C modifizieren den Sendercode durch einen speziellen Compiler, während AOP-Ansätze für Objective-C die Nachrichtenbindung zur Laufzeit ausnutzen können.

3.4.3 Best Practices und Status Quo

Bislang wurden alle bekannten Methoden zur Detektion und Auflösung kritischer APIs berücksichtigt. Für einige Teilbereiche gibt es offizielle Empfehlungen und Best Practices, die Apple in der Dokumentation an die Entwickler weitergibt. Die in dieser Arbeit schlussendlich erarbeitete Gesamtlösung sollte sich so nah wie möglich und sinnvoll an die Best Practices halten, um erfahrenen iOS-Entwicklern längere Einarbeitungszeiten zu ersparen und auf die von Apple angebotenen Hilfestellungen aufzubauen.

Grundsätzlich müssen einige **Regeln** eingehalten werden, von denen manche in anderen Abschnitten bereits genannt worden sind. Die Benutzung privater APIs ist nicht nur in Hinblick auf die App Store Review Guidelines untersagt, sondern auch aus rein technischer Sicht problematisch. Im Gegensatz zu öffentlichen APIs können sich private APIs mit jeder SDK-Version ohne Vorankündigung ändern. Die Verfügbarkeit privater APIs ist nicht dokumentiert, weshalb private APIs nie zur Compile-Zeit als kritisch erkannt werden können. Allgemeiner kann formuliert werden, dass von der Verwendung sämtlicher Implementierungsinterna abgesehen werden muss. In [Mas10] werden als Beispiele neben privaten APIs auch Zeitspannen von Animationen, Größenangaben in Pixeln und die Anordnung (z-Index) verschiedener Ebenen einer View-Hierarchie genannt. Generell ist vom Reverse Engineering der Frameworks abzusehen, bzw. sollten sich die daraus gewonnenen Erkenntnisse nicht direkt im Code widerspiegeln. Bei der Arbeit mit Categories muss darauf geachtet werden, Framework-Methoden nicht unbeabsichtigt zu überschreiben. Dies gilt auch für private, potenziell unbekannte Methoden, weshalb für Methoden in Categories generell ein Namenspräfix verwendet werden sollte [Mas10]. Die Einhaltung der genannten Regeln beeinflusst sowohl die Auf- als auch Abwärtskompatibilität einer Anwendung [Mas10].

In der Apple-Dokumentation wird darauf hingewiesen, dass ausschließlich das Base SDK zur Codeprüfung verwendet wird:

„Symbol lookup, code completion, and header file opening are based on the headers in the base SDK." [app10b]

Durch die alleinige Berücksichtigung des neusten SDKs werden kritische APIs maskiert. Dieses Problem wird von den derzeitigen Werkzeugen, Richtlinien und Dokumentationen vollkommen ignoriert. Es wird angenommen, dass jeder Entwickler die Mindestversion einer API entweder kennt oder in der Dokumentation nachschlägt. Erst für die Auflösungsphase werden für verschiedene API-Typen konkrete **Implementierungsvorschläge** unterbreitet. Dabei wird die Strategie verfolgt, die jeweilige Funktionalität explizit abzufragen anstatt die Laufzeitversionsnummer auszuwerten [app13b, TB11, Mas10]; dieses Vorgehen wird auch in Apple-unabhängiger Literatur empfohlen [Red11, Kapitel 8 Versioning]. Im Gegensatz zur Versionsnummernabfrage erfasst diese Strategie auch APIs, die nicht nur an die Systemversion, sondern auch die konkrete Hardware gebunden sind [Mas10]. Die folgenden Vorschläge zur Verfügbarkeitsabfrage sind in [Mas10], [app10b] und [app13b] übereinstimmend nachzulesen. Grundlegend für die Referenzierung kritischer Symbole zu Abfragezwecken ist der **Weak-Linking**-Mechanismus, der mit Mac OS X 10.2 eingeführt wurde[128] und in [app10b], [app06] und [app03] im Detail erklärt wird. Die Essenz ist, dass der dynamische Linker[129] nicht gefundene, als `weak` deklarierte Symbole auf `NULL` setzt.

Abfrage der Verfügbarkeit einer Methode

In Objective-C kann zur Laufzeit geprüft werden, ob eine Klasse oder Instanz auf eine gegebene Nachricht reagieren kann. Sofern der Nachrichtenmechanismus nicht bewusst manipuliert wurde, lässt sich dadurch auch das Vorhandensein einer Methodenimplementierung testen. Zur Abfrage kann der in Codebeispiel 3.9 enthaltene Code verwendet werden.

```
1  // Test anhand einer Klasse
2  if ([NSObject instancesRespondToSelector:@selector(doSomething)]) {
3      // Alle Instanzen antworten auf 'doSomething'-Nachrichten
4  }
5
6  // Test anhand einer Instanz
7  NSObject *instance = [NSObject new];
8  if ([instance respondsToSelector:@selector(doSomething)]) {
9      // Die gegebene Instanz antwortet auf 'doSomething'-Nachrichten
10 }
```

Codebeispiel 3.9: Verfügbarkeitsprüfung einer Objective-C-Methode

Sind mehrere Methoden in einem Protocol zusammengefasst, lässt sich mit einer Abfrage testen, ob ein Objekt alle verpflichtenden Methoden dieses Protocols implementiert. Dabei muss beachtet werden, dass lediglich die Angabe des Protocols im Header der Klasse über den Rückgabewert entscheidet [app13c]. Werden verpflichtende Methoden nicht tatsächlich implementiert, werden zwar Warnungen generiert[130], zur Laufzeit ist dieser Umstand jedoch nicht erkennbar[131]. Die API zum Test der Konformität mit einem Protocol existiert sowohl als Instanz- als auch als

[128] Die Technik ist unter allen iOS-Versionen verfügbar [app10b]

[129] Als dynamischer Linker wird *dyld* verwendet: `http://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/dyld.1.html`

[130] Dazu muss das `-Wprotocol`-Compiler-Flag gesetzt sein

[131] Es sei denn, alle Methoden werden einzeln auf ihre Verfügbarkeit überprüft

Klassenmethode, die Nutzung der Instanzmethode ist in Codebeispiel 3.10 beispielhaft dargestellt.

```
1 NSObject *instance = [NSObject new];
2 if ([instance conformsToProtocol:@protocol(NSCoding)]) {
3     // Die gegebene Instanz gibt an, das Protocol zu implementieren
4 }
```

Codebeispiel 3.10: Konformitätsprüfung eines Objekts gegen ein Protocol in Objective-C

Abfrage der Verfügbarkeit einer C-Funktion

Bevor eine kritische C-Funktion verwendet wird, kann ihre Existenz zur Laufzeit über den Namen des Symbols überprüft werden. Dies ist in Codebeispiel 3.11 dargestellt.

```
1 ABAddressBookRef aBook = nil;
2 // Die Funktionsparameter sind nicht Teil des Symbolnamens
3 if (ABAddressBookCreateWithOptions != NULL) {
4     // Die Funktion kann verwendet werden
5     aBook = ABAddressBookCreateWithOptions(NULL, nil);
6 }
```

Codebeispiel 3.11: Konditionale Verwendung einer C-Funktion

Abfrage der Verfügbarkeit einer Klasse

Eine Möglichkeit der Verfügbarkeitsabfrage einer Klasse wurde schon in Codebeispiel 3.7 aufgezeigt. Ab einem Base-SDK von iOS 4.2, einem Deployment Target von iOS 3.1 und der Benutzung eines LLVM-basierten Compilers[132] kann eine einfachere Variante gewählt werden, die in Codebeispiel 3.12 aufgezeigt wird [app10b].

```
1 // Die Referenzierung der Klasse ist mit Weak-Linking unkritisch
2 PKPass *instance;
3 if ([PKPass class]) {
4     // Die Klasse ist vorhanden und kann benutzt werden
5     instance = [PKPass new];
6 } else {
7     // Die Klassenmethode hat 'nil' zurückgegeben:
8     // Die Klasse ist nicht vorhanden
9 }
```

Codebeispiel 3.12: Konditionale Verwendung einer Objective-C Klasse

Anstelle der `class`-Methode (Codebeispiel 3.12, Zeile 3) kann auch eine beliebige andere Klassenmethode verwendet werden, da alle Klassenmethoden im Fall einer fehlenden Implementierung

[132] Clang oder llvm-gcc

`nil` zurückgeben. Somit können kritische Klassen in einem optimistischen Ansatz einfach verwendet werden, weil jeder Instanziierungsversuch `nil` zurückgibt und (Folge-)Nachrichten an `nil` in Objective-C keinen Laufzeitfehler verursachen. Damit die Anwendung nicht bereits beim initialen Laden abstürzt, müssen alle **kritischen Frameworks** als „optional" gekennzeichnet werden, um den Weak-Linking-Mechanismus anzustoßen. Das Setzen der Option in Xcode ist in Abbildung 3.15 gezeigt. Dies gilt nur für vollkommen neue Frameworks. Werden neue Klassen zu bestehenden Frameworks hinzugefügt, wird der Weak-Linking-Mechanismus automatisch anhand der Availability Macros verwendet [app10b].

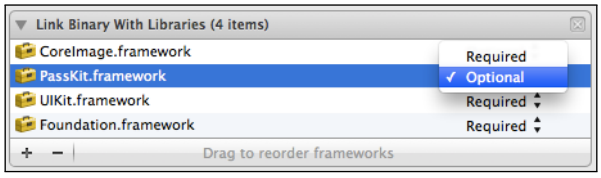


Abbildung 3.15: Weak-Linking von Frameworks in Xcode

Abfrage der Verfügbarkeit einer Konstanten

Die Werte einiger Konstanten, darunter Enumerations, werden bereits zur Compile-Zeit festgelegt (**Compile-Zeit-Konstanten**). Eine Enumeration aus dem PassKit-Framework ist exemplarisch in Codebeispiel 3.13 abgebildet.

```
typedef NS_ENUM(NSInteger, PKPassKitErrorCode) {
                                // Laufzeitwerte:
    PKUnknownError = -1,        //             -1
    PKInvalidDataError = 1,     //              1
    PKUnsupportedVersionError,  //              2
    PKInvalidSignature,         //              3
} NS_ENUM_AVAILABLE_IOS(6_0);
```

Codebeispiel 3.13: Beispieldeklaration einer Compile-Zeit-Konstanten

Die symbolischen Namen der Enum-Konstanten werden nur zur Entwicklungs- und Compile-Zeit referenziert. Der Compiler transformiert die Namen in die explizit oder implizit durch die Reihenfolge zugewiesenen Werte (im Beispiel vom Typ `NSInteger`), die unabhängig von SDK- und Plattformversionen sind. Der Zugriff auf diese Werte an sich kann demnach nicht zu Kompatibilitätsproblemen führen, genausowenig wie die Werte zur Laufzeit auf ihre Verfügbarkeit geprüft werden können. Es ist zu beachten, dass die Semantik der Werte durchaus versionsabhängig sein kann. Die Auflösung zur Laufzeit ist nicht möglich, also sollten Warnungen auf kritische Enums und Enum-Konstanten hinweisen.

Insbesondere Konstanten nicht-primitiver Datentypen werden ihre dann unveränderlichen Werte erst zur Laufzeit zugewiesen (**Laufzeitkonstanten**). In den Apple-Frameworks werden solche Konstanten üblicherweise als `extern`-Variablen deklariert, ein Beispiel befindet sich in Codebeispiel 3.14.

```
extern NSString *const PKPassKitErrorDomain NS_AVAILABLE_IOS(6_0);
```

Codebeispiel 3.14: Beispieldeklaration einer Laufzeitkonstanten

Im Gegensatz zu Compile-Zeit-Konstanten darf der Wert einer Laufzeitkonstanten nur abgefragt werden, wenn die Konstante tatsächlich deklariert und definiert worden ist. Dies lässt sich zur Laufzeit analog zu Codebeispiel 3.15 prüfen, indem die Adresse der Konstanten auf `NULL` geprüft wird.

```
if (&PKPassKitErrorDomain != NULL) {
    // Konstante kann verwendet werden
}
```

Codebeispiel 3.15: Konditionale Verwendung einer kritischen Laufzeitkonstanten

3.5 ZUSAMMENFASSUNG

In diesem Kapitel wurde ein **Gesamtüberblick** über die iOS-Plattform gegeben. Allgemeine Aspekte wie die primäre Programmiersprache Objective-C, das iOS SDK und die Entwicklungsumgebung Xcode wurden einleitend vorgestellt. In Abschnitt 3.2 wurde eine Übersicht der Software- und Geräteversionen tabellarisch dargestellt und diskutiert, mit dem Schluss, dass die Fragmentierung unter iOS ein überschaubares Problem ist und sich die Installationsbasis der jeweils aktuellen Version über die Jahre hinweg vergrößert hat.

Anschließend wurden für diese Arbeit besonders relevante **Teilaspekte** in separaten Abschnitten behandelt. Objective-C kann sich vor allem durch seine hohe Laufzeitdynamik von anderen kompilierten, objektorientierten Sprachen abheben. Das Binden einer Nachricht an eine Methodenimplementierung erfolgt in bis zu fünf Stufen, die in Abschnitt 3.3.1 detailliert vorgestellt worden sind. Der Build-Prozess in Xcode lässt sich einerseits durch Skripte erweitern, kann andererseits auch vollständig durch Skripte und somit externe Build-Systeme ersetzt werden. Zur Kompilierung des Quellcodes vertraut Apple seit einiger Zeit auf den LLVM-basierten Clang-Compiler, der von Apple maßgeblich mitentwickelt wird und einen großen Anteil an der schnellen Weiterentwicklung der Programmiersprache trägt. LLVM unterstützt auch die moderne, non-fragile Objective-C ABI, deren Vorteile in Abschnitt 3.3.4 zusammengefasst worden sind. Insbesondere behebt die neue ABI das Fragile Base Class Problem. Beim Einsatz fortgeschrittener APIs und Techniken muss sichergestellt werden, dass die resultierenden Anwendungen mit dem App Store als wichtigstem Vertriebskanal kompatibel bleiben. Diese Problematik wurde in Abschnitt 3.3.5 besprochen.

Im letzten Teil dieses Kapitels wurden die in Kapitel 2 gesammelten Techniken zur Detektion und Auflösung kritischer APIs mit den vorhandenen Werkzeugen und technischen Möglichkeiten der iOS-Plattform abgeglichen. Die **Detektion** ist derzeit nur mit externen Werkzeugen oder temporären Hacks möglich, die Wiederverwendbarkeit der Resultate für andere Werkzeuge ist nicht gegeben. Die derzeitigen Best Practices schlagen die direkte **Auflösung** durch konditionalen Code vor, ganzheitliche und in die vorhandenen Entwicklungswerkzeuge integrierte Ansätze zum Erlangen und Sicherstellen von Abwärtskompatibilität konnten nicht gefunden werden.

4 KONZEPTE FÜR DIE ABWÄRTSKOMPATIBILITÄT VON iOS-APPS

Im vorangegangenen Kapitel wurden Techniken untersucht, die einzelne Schritte des Detektions- und Auflösungsprozesses unter iOS unterstützen oder ganz übernehmen. In diesem Kapitel sollen die genannten Techniken zu einer Gesamtlösung zusammengefügt werden. Zunächst werden in Abschnitt 4.1 die Anforderungen an eine solche Gesamtlösung erörtert. Der Inhalt des Abschnitts stützt sich auf im Rahmen dieser Arbeit durchgeführte Interviews mit Entwicklern vornehmlich mobiler Anwendungen. In Abschnitt 4.2 wird grundlegend erklärt, wie sich die bereits bekannten Phasen der Detektion und Auflösung kritischer APIs sowie eine neu eingeführte Vorbereitungsphase gegenseitig beeinflussen. Dabei soll von konkreten Techniken und Umsetzungen zunächst abstrahiert werden. In den darauffolgenden Abschnitten erfolgt eine Abwägung zwischen den existierenden Techniken für die drei Phasen, mit dem Ziel, am Ende des Kapitels für jede Phase eine begründete Auswahl präsentieren zu können. Während innerhalb dieses Kapitels mehrere Konzeptideen und Teilkonzepte miteinander konkurrieren, soll zum Schluss des Kapitels ein Gesamtkonzept stehen, dessen Umsetzung Thema des 5. Kapitels ist.

4.1 ANFORDERUNGSANALYSE

Die im Folgenden zusammengestellten Anforderungen sollen sowohl bei der Auswahl der Techniken und Teilkonzepte als auch bei der finalen Evaluation in Kapitel 6 als Richtlinien dienen. Um die Anforderungen möglichst objektiv zu gestalten, wurden insgesamt zwölf Entwickler in **persönlichen Gesprächen** um ihre Meinung gebeten. Die Ergebnisse dieser Interviews erheben keinen Anspruch auf statistische Belastbarkeit und absolute Allgemeingültigkeit. Durch die große Übereinstimmung in wichtigen Fragen geben sie jedoch eine Richtung vor, an der es sich zu orien-

tieren gilt. Alle befragten Entwickler sind im Umfeld der mobilen Anwendungsprogrammierung tätig, elf von zwölf verfügen bereits über Erfahrung mit der App-Entwicklung für iOS.

AF.1 **Integration:** Die Integrierbarkeit in die bestehenden Entwicklungswerkzeuge ist ein absolutes Pflichtkriterium. Auf einer Skala von 1 (unwichtig) bis 5 (sehr wichtig) wurde die Xcode-Integration durchschnittlich mit 4,4 bewertet, keiner der befragten Entwickler gab eine Bewertung unter 3 ab. Einige Entwickler nehmen dafür in Kauf, dass die Xcode-Installation selbst verändert wird, was zu Einschränkungen beim Update-Prozess von Xcode führen kann. Die Unverändertheit von Xcode wurde durchschnittlich mit 3,1 bewertet. Dennoch sollte Xcode nach Möglichkeit nicht verändert werden, um das entstehende Konzept und Werkzeug auf lange Sicht mit Xcode kompatibel zu halten ohne fortlaufend Anpassungen durchführen zu müssen.

AF.2 **Priorisierung der nativen Implementierung:** Falls eine kritische API verfügbar ist, sollte diese stets gegenüber einer möglichen Ausweichvariante bevorzugt werden. Diese Anforderung kann mit den Best Practices aus Abschnitt 3.4.3 begründet werden und reduziert den Aufwand, die Ausweichvariante vollständig aus einem Projekt zu entfernen, wenn dessen Deployment Target zu einem späteren Zeitpunkt erhöht wird. Außerdem ist davon auszugehen, dass die Funktionalität auf Legacy-Plattformen nicht immer vollständig nachimplementiert wird, sondern oft vereinfacht oder weggelassen wird.

AF.3 **Performance:** Je nach Umsetzung der Detektionsphase kann deren Durchlauf Zeit kosten. Ist der Zeitaufwand besonders hoch, muss erwogen werden, die Detektion selektiv auszuführen, andernfalls kann sie bei jedem Build-Vorgang aktiviert bleiben. Die Ansprüche an die Compile-Zeit-Performance waren unter den befragten Entwicklern gemischt, durchschnittlich mit einem Wert von 2,6 aber eher gering. Sehr wichtig (Ø 4,0) ist dagegen, dass die Auflösungsphase, die teilweise zur Laufzeit der Anwendung stattfinden kann, die Laufzeit-Performance der Anwendung nicht spürbar beeinträchtigt.

AF.4 **App-Store-Kompatibilität:** Die Bedeutung des App Stores wurde bereits besprochen, die Kompatibilität der gebauten Anwendungen ist ein Pflichtkriterium für die praktische Anwendbarkeit eines Konzepts für die Abwärtskompatibilität. Dies bestätigt die durchschnittliche Bewertung von 4,9 unter den befragten Entwicklern. Die Anforderung impliziert zudem, dass die Stabilität der Anwendung nicht beeinträchtigt wird.

AF.5 **Zentrale Auflösung:** Alle befragten Entwickler bevorzugen eine Auflösung, die den bestehenden Code nicht oder nur minimal verändert und die eigentliche Auflösungslogik an zentraler Stelle implementiert, beispielsweise in einem Unterprojekt. Dies trägt zur Übersichtlichkeit des Anwendungscodes bei und erleichtert die Identifikation und Reversibilität der durch das Werkzeug durchgeführten Änderungen.

AF.6 **Umfangreiche Diagnostikinformationen:** Von einigen der befragten Entwickler wurde explizit gewünscht, dass trotz der zentralen Auflösung ersichtlich bleibt, welche Zeilen im Anwendungscode den Eingriff des Werkzeugs zur Auflösung von Inkompatibilitäten ausgelöst haben. Zusätzlich sollte zur Entwicklungszeit erkennbar sein, ab welcher Version eine kritische API verfügbar ist, und auf welchen Systemen die Ausweichimplementierung greift.

4.2 GESAMTARCHITEKTUR

Die Gesamtproblematik wird im Folgenden in drei Teilbereiche unterteilt, die in Abbildung 4.1 dargestellt sind. In Phase 1, der **Vorbereitungsphase**, werden Verfügbarkeitsinformationen aufbereitet, die für die darauffolgende Detektionsphase die Basis darstellen. Die entscheidende Fragestellung der ersten Phase ist, wie die Menge aller kritischen APIs effektiv erfasst und bereitgestellt werden kann. Je nach Umsetzung der Detektionsphase und dem Vorhandensein maschinenlesbarer Verfügbarkeitsdaten der APIs kann die erste Phase vollständig entfallen.

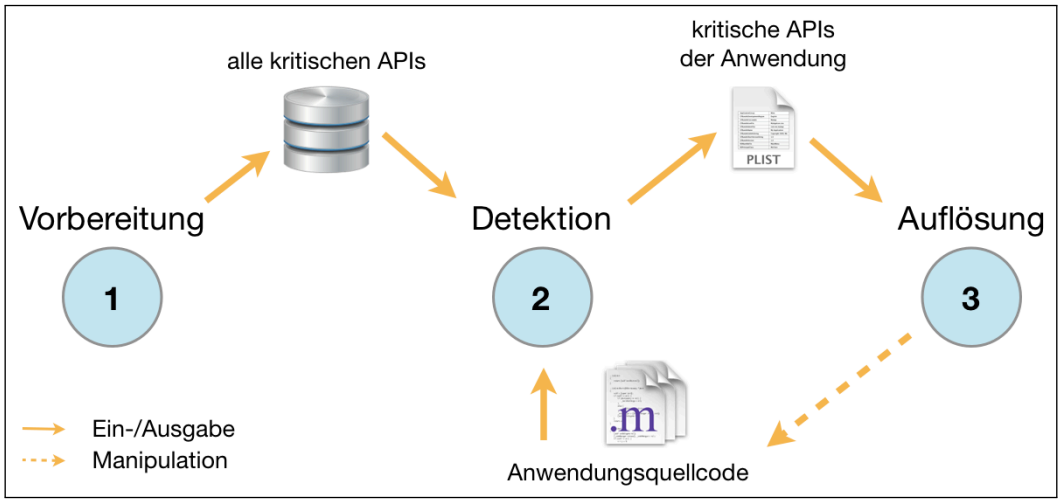


Abbildung 4.1: Zusammenspiel der Vorbereitungs-, Detektions- und Auflösungsphase

Die **Detektionsphase** muss die in der Anwendung verwendeten APIs mit der Menge aller kritischen APIs abgleichen. Auch hier ist die entscheidende Frage, wie die Suche im Quellcode effektiv und zuverlässig durchgeführt werden kann. Für eine automatisierte Auflösung in Phase 3 muss jede, in der Anwendung gefundene kritische API wie in Abbildung 4.1 dargestellt in eine Datei exportiert werden, oder auf anderem Wege an die Auflösungsphase weitergereicht werden. Die **Auflösungsphase** soll Code generieren, der Abstürze auf Legacy-Systemen verhindert und darüber hinaus dem Entwickler die Möglichkeit bietet, Ausweichcode einzufügen. Es stellt sich in erster Linie die Frage, auf welche Weise der generierte Code in das vorhandene Projekt integriert wird.

Alle Teilkonzepte sollen möglichst allgemeingültig beschrieben werden. Andererseits ist das Ziel dieser Arbeit, dass das finale Konzept für iOS-Anwendungen optimiert ist, sodass die **Umsetzbarkeit** auf dieser Plattform von Beginn an eine wichtige Rolle spielt und zu keinem Zeitpunkt vollständig ausgeblendet wird.

4.3 PHASE 1: VORBEREITUNG

Die Verfügbarkeit der iOS-Framework-APIs ist an drei Stellen dokumentiert: In der HTML-API-Dokumentation, in den API Diffs im HTML-Format und in den Header-Dateien. Neben der Frage

nach der optimalen **Datenquelle** muss entschieden werden, ob die gesamte Dokumentation durchsucht werden soll, oder nur die für die vorliegende Anwendung relevanten Dateien. Das Einlesen der gesamten Dokumentation ist ein zeitaufwändiger Prozess, dessen Ergebnisse jedoch für verschiedene Projekte und viele Detektionsdurchläufe wiederverwendet werden können, wenn sie in einer Datenbank abgespeichert werden. Sofern es möglich ist, die für die Anwendung relevanten Dokumentationsdateien leicht einzugrenzen, können auch nur diese eingelesen werden und die Ergebnisse direkt im Speicher gehalten werden.

Das **Parsen** von HTML-Dateien ist prinzipiell langsamer und instabiler als das Parsen von C-Header-Dateien. Die HTML-Dateien enthalten nicht nur für die automatische Verarbeitung irrelevante Layoutinformationen, sondern können auch strukturell jeder Zeit und beliebig von Apple verändert werden, sodass Parser neu geschrieben bzw. konfiguriert werden müssen, um die gewünschten Daten zu extrahieren. Für C-Header sind dagegen Bibliotheken zum Parsen verfügbar[133], u. a. die Kernbibliothek des Clang-Compilers, *LibClang*[134]. Denkbar ist auch, quelloffene Dokumentationswerkzeuge wie *appledoc*[135] oder *Doxygen*[136] so zu modifizieren, dass sie die Verfügbarkeit von Symbolen nicht nur darstellen, sondern in ein eigenes Format oder eine Datenbank exportieren.

Werden die API-Dokumentation oder die Header-Dateien als Datenquelle genutzt, reicht die Betrachtung der Version des neusten SDKs, in der pro API die Verfügbarkeit explizit genannt wird. Andererseits lässt sich die Anzahl der **relevanten Dateien** nicht durch Heranziehen des Deployment Targets eingrenzen. Bei der Nutzung der API Diffs müssen lediglich die Diff-Dateien bis einschließlich der ersten Version über dem Deployment Target betrachtet werden, alle vorherigen API-Änderungen sind irrelevant[137]. Die Gesamtanzahl und -größe der zu verarbeitenden Dateien ist bei den Diff-Dateien voraussichtlich am geringsten, weil keine für diesen Prozess irrelevanten Informationen darin enthalten sind.

Aufgrund der Konzentriertheit der relevanten Daten scheinen die API Diffs die effizientere der beiden HTML-Varianten zu sein. Die Header-Dateien bieten den Vorteil der stabileren Werkzeuge zum Extrahieren der gewünschten Daten. Zudem lassen sich die relevanten Header-Dateien über die Import-Anweisungen eines Projekts bestimmen.

4.4 PHASE 2: DETEKTION

Die Umsetzung der Detektionsphase wird sich an **Lint** für Android orientieren. Speziell die IDE-Integration von Lint soll als Vorbild dienen, darüber hinaus müssen die gefundenen kritischen APIs für die dritte Phase exportierbar sein. Die beiden existierenden Werkzeuge, AppCode und Deploymate, erfüllen die Anforderung AF.1 (Integration) wegen der fehlenden Xcode-Integration nicht. Zudem bietet keines der beiden Werkzeuge eine Export-Möglichkeit der kritischen APIs.

[133] Auch HTML-Parser sind frei verfügbar, allerdings ist in HTML nur die Syntax, nicht die Semantik der dargestellten Inhalte definiert – C-Parser können die eingelesenen Daten sofort deuten, sofern es sich nicht um Kommentare handelt

[134] `http://clang.llvm.org/docs/Tooling.html`

[135] `http://gentlebytes.com/appledoc/`

[136] `http://www.doxygen.org/`

[137] Möchte man die Daten einmalig einlesen und cachen oder in einer Datenbank ablegen, können ebenso alle Diff-Dateien eingelesen werden

Beide Werkzeuge sind nicht quelloffen und können demnach nicht für die Zwecke dieser Arbeit modifiziert werden.

Eine Möglichkeit, kritische APIs ausfindig zu machen, ist das **Herabsetzen des Base SDKs**. Kompiliert die Anwendung fehlerfrei, nachdem das Base SDK dem Deployment Target gleichgesetzt wurde, enthält die Anwendung keine kritischen APIs. Andernfalls werden kritische APIs als Fehler angezeigt – diese Vorgehensweise wird für Android-Apps in [Mei12] erwähnt. Im Gegensatz zu Android lassen sich verschiedene iOS SDKs jedoch nicht trivial in Xcode verwalten. Um die Anwendung schlussendlich fehlerfrei mit dem ursprünglich gewünschten Base SDK zu bauen, ist ein zusätzlicher Kompilierungsvorgang notwendig. Damit für akkurate Diagnostikinformationen (Anforderung AF.6) die genaue Mindestversion einer API ermittelt werden kann, müssen weitere Kompilierungsvorgänge durchgeführt werden, was insbesondere bei großen Projekten mit einer großen Spanne zwischen Base SDK und Deployment Target zu einer Vervielfachung des Zeitaufwands führt und dadurch mit AF.3 (Performance) in Konflikt steht. Für kritische APIs unter Verwendung eines zu geringen Base SDKs generiert der Compiler Fehler statt Warnungen. Fraglich ist, ob der Kompilierungsvorgang trotz dieser Fehler überhaupt fortgesetzt werden kann, ob durch die fehlenden Symbole möglicherweise kritische APIs übersehen werden, und wie die generierten Fehler anschließend weiterverwendet werden können. Insgesamt lässt ein solcher Ansatz einen hohen Aufwand und große Performanceprobleme erwarten. Ein Vorteil ist, dass die Vorbereitungsphase entfallen kann.

Vergleichbare Probleme bringen Konzepte mit sich, die auf der Redefinition der **Verfügbarkeitsmakros** basieren. Für genaue Diagnostikinformationen müssen mehrere Kompilierungsvorgänge durchgeführt werden (mindestens zwei) und die Resultate sind schwer wiederverwendbar. Vorteilhaft ist, dass die Verwaltung der SDKs entfällt und Warnungen statt Fehlern erzeugt werden. Der Vorteil der entfallenen Vorbereitungsphase bleibt: Die Verfügbarkeitsinformationen werden überhaupt nicht genutzt, sondern durch `deprecated`-Attribute ersetzt[138]. Mit zusätzlichen Skript-Build-Phasen (vgl. Abschnitt 3.3.2) kann die Redefinition der Makros und das wiederholte Starten des Build-Prozesses leicht automatisiert werden. Die Xcode-Integration (AF.1) kann somit als gegeben angenommen werden.

Die nächste Kategorie von Detektionsansätzen beruht auf der **expliziten Suche** kritischer APIs im Quellcode. Zuvor muss die Vorbereitungsphase durchlaufen werden. Es bestehe die Annahme, dass die Datenbank der kritischen APIs nach der Vorbereitungsphase m Einträge enthält. Zudem sei n die Anzahl der Quellcodesymbole im Anwendungscode, und u ($u \leq n$) die Anzahl der verschiedenen Quellcodesymbole. Die Suche kann in zwei Richtungen durchgeführt werden. Einerseits kann für alle m Symbole der Datenbank eine Suche über alle n Quellcodesymbole erfolgen. Diese Vorgehensweise (**Quellcodesuche**) ist besonders günstig, wenn die Datenbank (m) klein ist, beispielsweise, weil in der Vorbereitungsphase lediglich für das Projekt relevante Dateien betrachtet worden sind. Allerdings müssen sämtliche Quellcodesymbole m-mal verglichen werden. Alternativ kann der Quellcode einmal vollständig durchlaufen werden und für alle n Quellcodesymbole eine Datenbankabfrage durchgeführt werden (**Datenbanksuche**), was im Vergleich zum ersten Szenario eine erhöhte Anzahl von Datenbankzugriffen zur Folge hat. Optimiert werden kann diese Herangehensweise, indem die Datenbankabfragen gecacht werden und nur u Anfragen gestellt werden. Weil die tatsächlichen Ausprägungen von m, n und u letztlich

[138] Die anschließende Detektion von `deprecated` APIs wird bereits unterstützt

nur gemutmaßt werden können und stark von der Größe des Projekts sowie der Art der Vorbereitungsphase abhängen, müssten im Zweifel beide Szenarien praktisch erprobt und verglichen werden, es sei denn, andere Gründe sprechen klar für eine der beiden Suchvarianten.

Die Quellcodesuche kann mit einem naiven Suchskript unter Verwendung von *grep*[139] umgesetzt werden. Für jede in der Datenbank erfasste kritische API müsste eine grep-Suche über alle Dateien des Projekts ausgeführt werden. Die rein **textuelle Suche** stößt allerdings schnell an ihre Grenzen, wie mithilfe von Codebeispiel 4.1 demonstriert werden soll.

```
1 MyClass *instance = [MyClass new];
2 PKPass *thePass = [PKPass new];
3 thePass.serialNumber = @"SN01";        // PKPass Property (kritisch)
4 instance.serialNumber = @"SN02";       // MyClass Property (nicht kritisch)
5 // [thePass setSerialNumber:@"SN01"];  // Alternative Syntax
```

Codebeispiel 4.1: Exemplarische Problemfälle einer rein textuellen Quellcodesuche

Sowohl die `PKPass`-Klasse als auch die `serialNumber`-Property selbiger Klasse seien in der Datenbank der kritischen APIs eingetragen. Die Textsuche nach `.serialNumber` würde aber neben der tatsächlich kritischen API (Codebeispiel 4.1, Zeile 3) auch Properties gleichen Namens von Instanzen anderer Klassen finden (Zeile 4). Für ein textbasiertes Werkzeug ist es keine triviale Aufgabe, vom Variablennamen einer Instanz zuverlässig auf die Klasse der Instanz zu schließen. Dazu kommt, dass der Zugriff auf eine Property in Objective-C auf syntaktisch verschiedene Weisen erfolgen kann. Eine semantisch äquivalente Alternative ist als Kommentar in Zeile 5 dargestellt – ein Suchskript sollte beide Syntaxvarianten finden und als semantisch identisch klassifizieren können. Ein weiteres Problem in Zeile 5 ist, dass ein Suchskript erkennen müsste, dass es sich um einen Kommentar und nicht um ausführbaren Code handelt. Obwohl das gewählte Beispiel in Objective-C geschrieben ist, ist das Grundproblem nicht programmiersprachenabhängig. Vielmehr fehlen einer textuellen Suche semantische und kontextuelle Informationen, die sich nur mühsam extrahieren lassen.

Stattdessen sollte auf vorhandene Compiler-Werkzeuge und -Bibliotheken zurückgegriffen werden, die Quellcode zuverlässig parsen können und eine **symbolische Suche** unterstützen. Im Idealfall lässt sich die symbolische Suche der IDE automatisiert verwenden[140]. Ansonsten bietet sich für C-basierte Sprachen die LibClang-Bibliothek nicht nur zum Parsen der Header-Dateien (s. Abschnitt 4.3), sondern auch zur Suche an. Um die IDE-Integration (Anforderung AF.1) zu wahren, sollte ein auf LibClang aufbauendes Werkzeug über die Kommandozeile ausführbar sein, um es dann über Skripte in den Buildprozess einzubinden. Ein solches Werkzeug kann neben der Quellcodesuche auch die Datenbanksuche umsetzen, indem der gesamte Quellcode geparst wird und für jedes neue Symbol eine Datenbankabfrage ausgeführt wird.

Im Rahmen der Ideenfindung für diese Arbeit wurde untersucht, ob sich die vom Compiler produzierten **Objektdateien** für eine Suche nach kritischen Symbolen eignen. Mithilfe von *otool*[141]

[139] `http://unixhelp.ed.ac.uk/CGI/man-cgi?grep`

[140] An dieser Stelle soll vorweggenommen werden, dass dies für Xcode nicht der Fall ist. Xcode stellt eine AppleScript-API bereit, die allerdings keine Möglichkeit der symbolischen Suche vorsieht. Außerdem ist die Implementierung der AppleScript API in Xcode 4 lückenhaft.

[141] `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/otool.1.html`

lassen sich Referenzen auf externe Symbole ausgeben; auf diese Informationen greift auch der Linker zu. Alternativ kann das Kommandozeilenwerkzeug *nm*[142] zur Inspektion verwendet werden. Ein auf den Objektdateien basierender Ansatz wurde jedoch nicht weiter verfolgt, weil die Verbindung zum Quellcode verloren geht. Wird ein kritisches Symbol im Binärcode gefunden, kann nicht ohne Weiteres zurückverfolgt werden, in welcher Zeile des Quellcodes die kritische API referenziert wurde. Mit der Anforderung AF.6 (Umfangreiche Diagnostikinformationen) ist dies nicht vereinbar, zumal nur statisch verlinkte Symbole in den Objektdateien referenziert werden. Objective-C-Methoden können auf diese Weise nicht gefunden werden[143]. Mit hinreichend Aufwand, bis hin zu einem Disassembly-Vorgang, lassen sich genauere Details über die Verwendung bestimmter APIs herausfinden [Ash11], doch dieser Aufwand ist nur gerechtfertigt, wenn der Quellcode nicht vorliegt. Letzteres wird im Rahmen dieser Arbeit als gegeben angenommen. Wird die Detektion kritischer APIs in kompilierten Bibliotheken Dritter gefordert, muss dieser Ansatz weiterverfolgt werden.

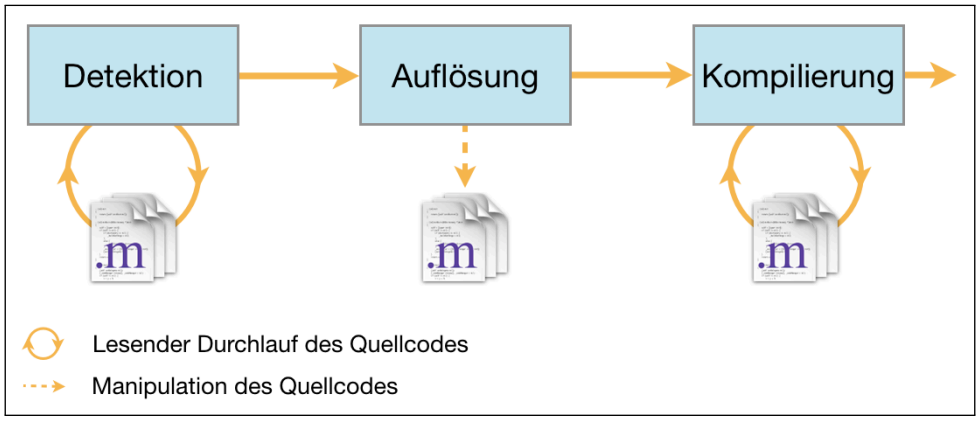


Abbildung 4.2: Mehrfaches Durchlaufen des Quellcodes für Detektion und Kompilierung

Wird ein Werkzeug nach einem der genannten Konzepte implementiert, kann es in Xcode durch Skripte in den Build-Prozess eingebunden werden. Der gesamte Quellcode muss zur Detektion mindestens einmal durchlaufen werden. In der bestehenden Kompilierungsphase der IDE wird der Quellcode ein weiteres Mal durchlaufen, wie Abbildung 4.2 verdeutlichen soll. Zwar ist die Compile-Zeit-Performance kein besonders hoch gewichtetes Anforderungskriterium (AF.3), dennoch sollte sie nach Möglichkeit optimiert werden. Deshalb muss in Erwägung gezogen werden, die Detektion direkt in den Compiler zu integrieren und für den IDE-Build-Prozess den **modifizierten Compiler** zu nutzen. Dieser Ansatz ist in Abbildung 4.3 grafisch dargestellt. Weil die Detektion nicht vor dem Ende des Kompilierens abgeschlossen ist, kann die Auflösungsphase erst nach dem Kompilieren starten[144]. Wird während der Auflösungsphase Code geändert oder neu hinzugefügt, muss der Kompilierungsprozess neu gestartet werden, damit die gebaute Anwendung den aktuellen Codestand repräsentiert. Im Allgemeinen ist der zweite Kompilierungsvorgang deutlich schneller, weil temporäre Build-Caches genutzt werden können[145]. Selbst im Fall

[142] `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/nm.1.html`
[143] Die Namen der Selektoren lassen sich im Textsegment finden, Ort und Art der Verwendung sowie die Empfängerklasse sind daraus jedoch nicht ersichtlich
[144] Einzelne Auflösungsschritte können theoretisch bereits nach der Detektion der ersten kritischen APIs erfolgen, die Auflösungsphase kann jedoch in keinem Fall vor dem Ende der Detektions- und damit auch Kompilierungsphase abgeschlossen werden
[145] Eine modulare Anwendungsstruktur und lose Abhängigkeiten begünstigen die Effizienz von Build-Caches und inkrementellen Builds [Lak05, Kapitel 0 Introduction]

einer Rekompilierung ist deshalb nicht davon auszugehen, dass der gesamte Quellcode zweimal durchlaufen wird.

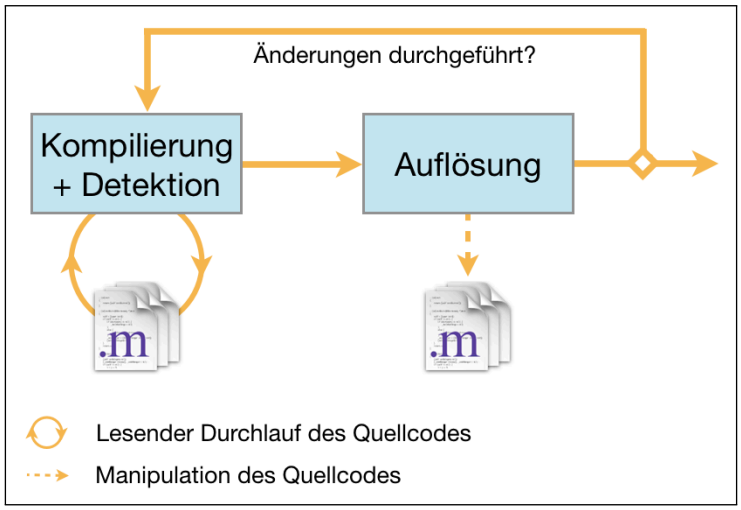


Abbildung 4.3: Detektion und Kompilierung in einem Durchlauf des Quellcodes

Bevor die eigentliche Kompilierung startet, werden vom **Präprozessor** Makros expandiert und die Import-Anweisungen durch den gesamten Inhalt der referenzierten Header-Dateien ersetzt. Sofern die Verfügbarkeitsinformationen in den Headern enthalten sind, befinden sich nach der Präprozessorphase alle Verfügbarkeitsinformationen der importierten APIs in der expandierten Version der Quellcodedatei. Die Vorbereitungsphase kann in diesem Fall direkt im Compiler implementiert werden, weil der die expandierten Dateien vollständig, zeilenweise durchläuft und sich die Deklarationen und Verfügbarkeitsinformationen der kritischen APIs zwangsweise vor deren erster Referenzierung im Anwendungscode befinden. Der Ablauf ist beispielhaft anhand einer gekürzten und kommentierten, expandierten Quellcodedatei in Codebeispiel 4.2 dargestellt.

```
1  // Annahme: iOS Deployment Target < 6.0
2  /**
3   *    Beginn der Vorbereitungsphase
4   *    Die folgenden Zeilen wurden vom Präprozessor eingefügt,
5   *    ausgelöst durch '#import <PassKit/PKPass.h>'
6   */
7
8  // Verfügbarkeitsattribute einer kritischen API (eingeführt in iOS 6)
9  __attribute__((visibility("default"))) __attribute__((availability(ios,
           introduced=6.0)))
10 @interface PKPass : NSObject
11     // [gekürzt] Deklarationen von PKPass
12 @end
13
14 /**
15  *    Abschluss der Vorbereitungsphase / Beginn der Detektionsphase
16  *    Menge aller kritischen APIs: {PKPass}
17  */
```

```
18 // Eigentlicher Anwendungscode
19 @implementation MyClass
20 - (id) myMethod {
21     // Nutzung einer kritischen API
22     PKPass *pass = [PKPass new];
23     return pass;
24 }
25 @end
26
27 /**
28  *    Abschluss der Detektionsphase
29  *    Menge der verwendeten kritischen APIs: {PKPass}
30  */
```

Codebeispiel 4.2: Kommentierte, expandierte Quellcodedatei (Ausschnitt)

Die Verfügbarkeitsinformationen müssen nicht in einer Datenbank abgelegt werden, sondern können in Datenstrukturen des Compilers im Speicher gehalten werden. Sofern die Verfügbarkeitsattribute in einem standardisierten Format angegeben werden und der Compiler den Standard implementiert, ist auch die Vorbereitungsphase bereits fertig implementiert. Andernfalls muss der Parser des Compilers so modifiziert werden, dass er die Verfügbarkeitsinformationen speichert. Weil alle wesentlichen Operationen (das Expandieren der Imports und das Parsen der expandierten Dateien) in jedem Fall bereits durchgeführt werden, verursacht die Vorbereitungsphase keine zusätzlichen Kosten. In Kombination mit einer direkt beim Kompilieren vorgenommenen Detektion auf Basis der sich im Speicher befindenden Verfügbarkeitsdaten kann von minimalen Performance-Beeinträchtigungen ausgegangen werden. Entscheidend für die Umsetzbarkeit des Konzepts ist letztlich, ob der Compiler modifizierbar ist, und ob sich der modifizierte Compiler nahtlos in das Build-System der IDE eingliedern lässt.

4.5 PHASE 3: AUFLÖSUNG

Die trivialste Form der Auflösung ist das Kapseln einer kritischen API im Anwendungscode. Der Code zum Kapseln verschiedener Typen von APIs kann in den Best Practices in Abschnitt 3.4.3 nachgeschlagen werden. Die Nachteile dieses Ansatzes wurden bereits in Abschnitt 2.2.2 besprochen und liegen insbesondere in der schlechten Lesbarkeit des Codes, einer hohen Redundanz und schwieriger Reversibilität. Zudem wurde der Ansatz durch Anforderung AF.5 (Zentrale Auflösung) explizit ausgeschlossen.

4.5.1 Generierung der Kompatibilitätsbibliothek

In Anlehnung an die Android Support Libraries kann eine eigene Bibliothek generiert werden, die eine der Framework-Schnittstelle ähnelnde API bereitstellt. Diese Bibliothek soll im Folgenden

als **Kompatibilitätsbibliothek** bezeichnet werden. In Sprachen mit Unterstützung für Namensräume können sich die Namen der APIs des Frameworks und der Kompatibilitätsbibliothek bis auf den Namensraum gleichen[146]. In Objective-C gibt es keine Namensräume, weshalb zur Vermeidung von Namenskonflikten ein Präfix für die APIs einer eigenen Bibliothek verwendet werden sollte.

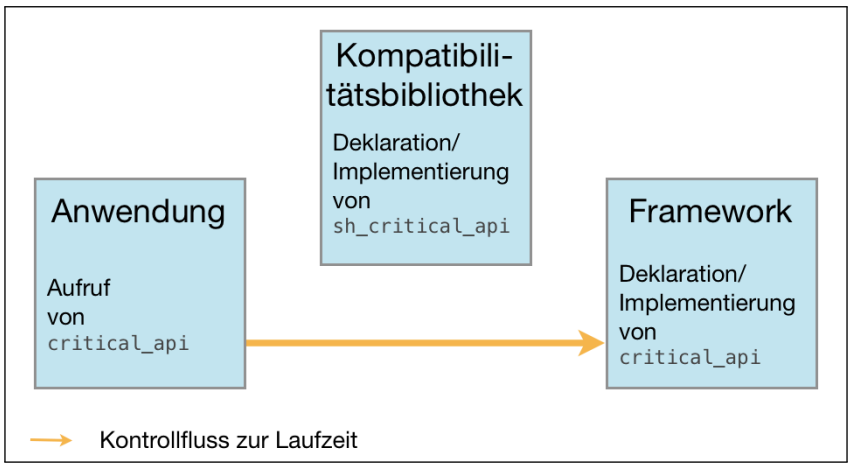


Abbildung 4.4: Kompatibilitätsbibliothek vor ihrer Einbindung

Der Code, der an Stelle der Implementierung einer Framework-API ausgeführt werden kann, wird als **Ausweichimplementierung** (engl.: „Fallback implementation") bezeichnet. Um die Kompatibilitätsbibliothek nicht unnötig groß und unübersichtlich zu gestalten, sollte sie nur APIs bereitstellen, die für die vorliegende Anwendung relevant sind. Relevant sind alle kritischen APIs, die während der Detektionsphase in der Anwendung gefunden worden sind. Die Aufgabe der Auflösungsphase ist es zunächst, für all diese kritischen APIs eine mit dem entsprechenden Präfix ausgestattete API in der Kompatibilitätsbibliothek zu deklarieren und zu definieren. Dieser Zustand ist in Abbildung 4.4 dargestellt. Die Ausweichimplementierung kann dabei im einfachsten Fall leer bleiben, sodass ein Absturz verhindert wird, die Funktionalität auf Legacy-Systemen aber nicht erhalten bleibt. Der Anwendungsentwickler kann die API-Definition in der Kompatibilitätsbibliothek jedoch nutzen, um eine eigene Implementierung bereitzustellen. Dies kann eine vollständige oder partielle Nachimplementierung der Framework-API sein, oder ein Hinweis für den Nutzer, dass diese Funktionalität auf seinem Gerät nicht zur Verfügung steht. Denkbar ist auch, für spezifische APIs eine allgemeingültige Ausweichimplementierung vorzugeben.

4.5.2 Einbindung der Kompatibilitätsbibliothek

Die Einbindung der Kompatibilitätsbibliothek kann statisch oder dynamisch erfolgen. **Statisch** bedeutet in diesem Fall, dass bereits zur Compile-Zeit sämtliche Referenzen auf kritische Framework-APIs durch Aufrufe der entsprechenden APIs aus der Kompatibilitätsbibliothek ersetzt werden. Wie aus Abbildung 4.5 ersichtlich ist, entscheidet die Kompatibilitätsbibliothek zur Laufzeit, ob ein Aufruf an die native Implementierung weitergeleitet (Anforderung AF.2) oder die Ausweichimplementierung genutzt wird.

[146] Dieser Weg wird von den Android Support Libraries beschritten

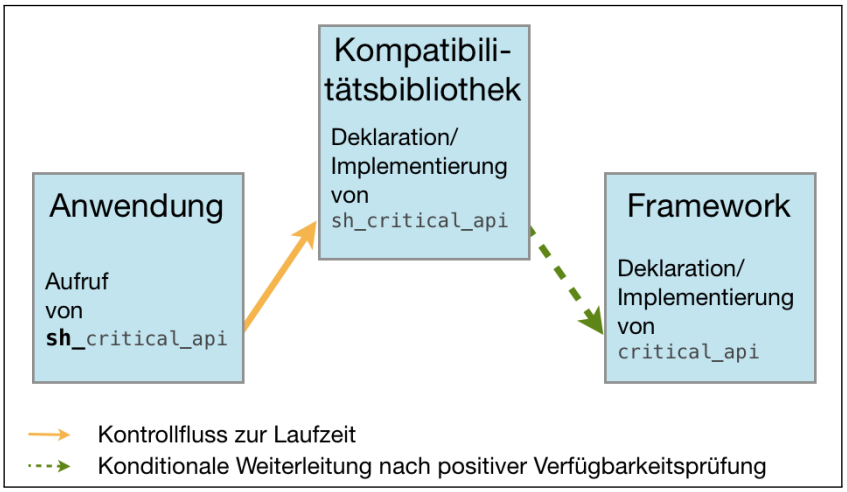


Abbildung 4.5: Statische Einbindung der Kompatibilitätsbibliothek

Die Laufzeitperformance (Anforderung AF.3) wird bei der statischen Einbindung nur minimal durch die Verfügbarkeitsabfrage und anschließende Weiterleitung beeinflusst. Von einer spürbaren Beeinträchtigung kann nicht ausgegangen werden. Lediglich Anforderung AF.5 (Zentrale Auflösung) kann nicht ideal erfüllt werden, weil die eigentliche Auflösung zwar zentral in der Bibliothek stattfindet, die Einbindung aber eine geringfügige Modifikation des bestehenden Quellcodes erfordert. Konkret müssen die kritischen API-Aufrufe um den Präfix der Kompatibilitätsbibliothek erweitert werden (in Abbildung 4.5 wurde der eingefügte `sh_`-Präfix fett markiert).

Um den Anwendungsquellcode vollkommen unberührt zu lassen, muss die Einbindung der Kompatibilitätsbibliothek **dynamisch** erfolgen[147]. Diese Strategie ist in Abbildung 4.6 dargestellt. Generell sendet die Anwendung alle Aufrufe an das native Framework. Erst wenn die API dort nicht gefunden wurde, wird der Aufruf an die Kompatibilitätsbibliothek weitergeleitet. Die Umleitung ist jedoch nicht trivial, weil die Frameworks nicht im Quellcode vorliegen und auf iOS-Geräten vorinstalliert sind.

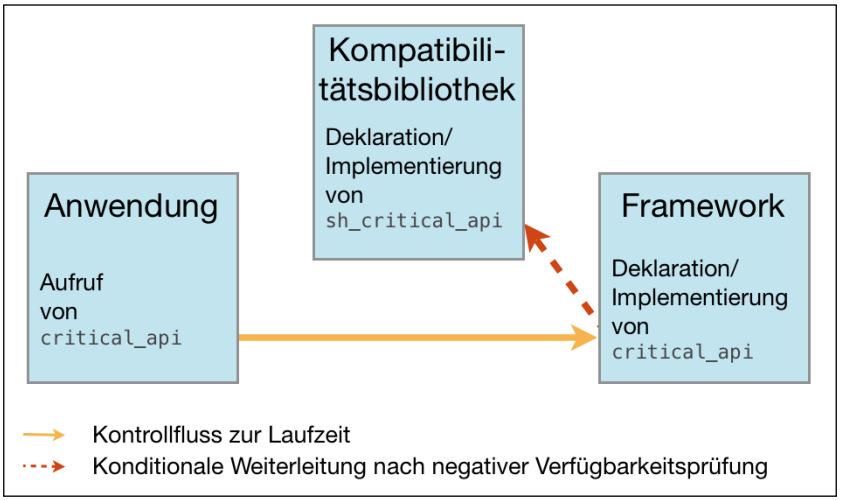


Abbildung 4.6: Dynamische Einbindung der Kompatibilitätsbibliothek

[147] Die Bibliothek muss trotzdem statisch verlinkt werden; gemeint ist die Einbindung in den Kontrollfluss

Unter der Annahme, dass eine produktiv einsatzfähige AOP-Technologie existiere, kann für jede kritische API ein Pointcut spezifiziert werden. Die `around`-Advice zu diesen Pointcuts enthalten die Verfügbarkeitsabfrage und je nach Ergebnis eine Ausführung der nativen Implementierung (dem Join Point) oder eine Weiterleitung an die Kompatibilitätsbibliothek. Ein `before`-Advice ist nicht ausreichend, weil darin zwar die konditionale Weiterleitung an die Kompatibilitätsbibliothek realisiert werden kann, die native API aber in jedem Fall zusätzlich aufgerufen wird. Die eingangs genannte Annahme einer stabilen AOP-Technologie ist weder für Objective-C noch für C gegeben. In Abschnitt 2.2.2 wurde aufgezeigt, wie AOP in einer dynamischen Sprache wie Smalltalk oder Objective-C konzeptionell umgesetzt werden kann.

In Objective-C lässt sich alternativ ausnutzen, dass die Verfügbarkeit von Methoden implizit von der Laufzeitumgebung geprüft wird. Sei `critical_sel` der Selektor einer kritischen Methode. Wird in der Selektorentabelle des Empfängers eine Methode zu `critical_sel` gefunden, folgt die sofortige Ausführung der Methode. Andernfalls setzt der in Abschnitt 3.3.1 beschriebene, fünfstufige Auflösungsmechanismus ein. Um die Laufzeitperformance (Anforderung AF.3) zu wahren, sollte die Auflösung frühstmöglich in der zweiten Stufe erfolgen[148]. In dieser Stufe (Permanente Auflösung) kann die Klasse eine Ausweichimplementierung nachreichen, indem sie die gewünschte Implementierung unter dem `critical_sel`-Selektor verfügbar macht. Dazu muss die Klasse die `resolveInstanceMethod:`-Methode implementieren[149]. Alle weiteren Aufrufe der gleichen kritischen Methode nutzen die nachgereichte Implementierung, ohne den fünfstufigen Auflösungsmechanismus erneut zu durchlaufen. Die Ausweichimplementierung kann der Kompatibilitätsbibliothek entnommen werden. In diesem Fall werden die Klassen der Kompatibilitätsbibliothek nicht direkt genutzt, sondern dienen als Code-Repository, aus dem sich andere Klassen zur Laufzeit bedienen können. Offensichtlich sind die unmodifizierten Frameworkklassen nicht in der Lage, die Ausweichimplementierungen der Kompatibilitätsbibliothek zu entnehmen. Jede Frameworkklasse, die kritische Methoden deklariert, muss deshalb zur Laufzeit[150] modifiziert werden, um in `resolveInstanceMethod:` das beschriebene Verhalten zu implementieren. Die Umsetzung mithilfe von Categories und Method Swizzling wird später in Abschnitt 5.2.3 genau beschrieben.

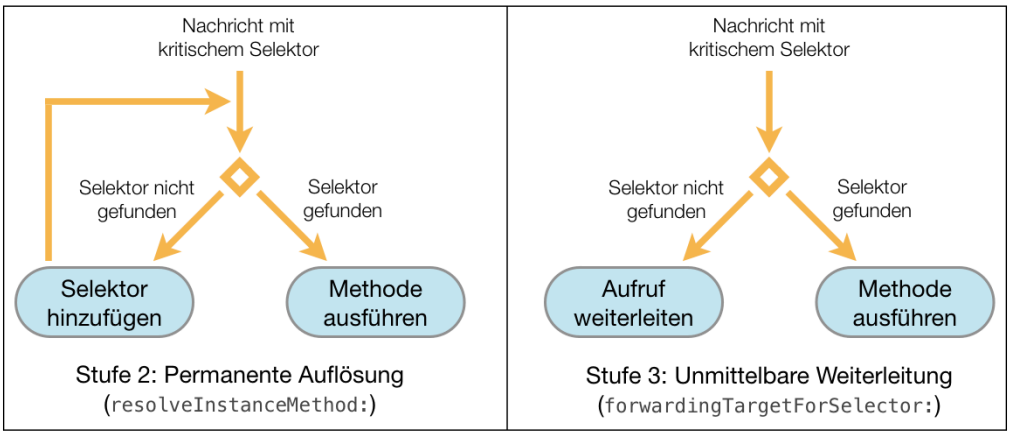


Abbildung 4.7: Vergleich zweier Möglichkeiten der dynamischen Nachrichtenauflösung

[148] Die erste Stufe ist das reguläre Auflösen mithilfe der Selektorentabelle
[149] Im Fall einer Klassenmethode muss `resolveClassMethod:` implementiert werden
[150] Vor der Benutzung einer kritischen Methode

In Abbildung 4.7 werden die **dynamische Nachrichtenauflösung** in Stufe 2 und in Stufe 3 in Form von Aktivitätsdiagrammen verglichen. In Stufe 2 wird der Ausweichpfad lediglich einmal pro kritischer API durchlaufen, weil die Pfadbedingung („Selektor nicht gefunden“) durch die Pfadaktion („Selektor hinzufügen“) für zukünftige Durchläufe ausgeschlossen wird. Erfolgt die Nachrichtenauflösung durch eine unmittelbare Weiterleitung in Stufe 3, wird der Ausweichpfad entweder nie (native API vorhanden) oder bei jeder Nachricht (native API nicht vorhanden) durchlaufen, was in letzterem Fall den Nachrichten-Overhead geringfügig erhöht. Auch für die Auflösung in Stufe 3 gilt, dass existierende Frameworkklassen zur Laufzeit modifiziert werden müssen, um die Weiterleitungslogik hinzuzufügen.

In den beiden Ansätzen der dynamischen Nachrichtenauflösung werden die Frameworkklassen so modifiziert, dass sie die Ausweichimplementierung selbstständig zur Laufzeit finden und nutzen können. Die Manipulation der Frameworkklassen ist unabhängig von der Anzahl der kritischen APIs, die von der dynamischen Auflösung letztlich betroffen sind. Anstatt den Frameworkklassen einen solchen Auflösungsmechanismus hinzuzufügen, können den Klassen direkt die fehlenden APIs **präventiv** hinzugefügt werden. Weil die Klassen nicht quelloffen sind, muss auch dies zur Laufzeit umgesetzt werden. Beispielhaft ist dies in Codebeispiel 4.3 gezeigt.

```
1  // Menge der aufzulösenden Methoden
2  NSArray *criticalAPIs = @[@"criticalAPI1", @"criticalAPI2"];
3  for (NSString *criticalAPI in criticalAPIs) {
4      SEL criticalAPISel = NSSelectorFromString(criticalAPI);
5      Class targetClass = [NSObject class];
6
7      // Die Methode soll nur hinzugefügt werden, wenn sie nativ nicht
8      // vorhanden ist
9      if (![targetClass instancesRespondToSelector:criticalAPISel]) {
10         // Details der Ausweichimplementierung müssen von Hilfsfunktionen
11         // der Kompatibilitätsbibliothek bereitgestellt werden
12         char *types = sh_getTypesForSel(criticalAPISel);
13         IMP impl = sh_getImplementationForSel(criticalAPISel);
14
15         // Hinzufügen einer Ausweichimplementierung
16         class_addMethod(targetClass,
17                         criticalAPISel,
18                         impl,
19                         types);
20       }
21 }
```

Codebeispiel 4.3: Hinzufügen von Ausweichimplementierungen zur Laufzeit

Die Ausführung des abgebildeten Codes[151] bewirkt das Hinzufügen von zwei Methoden zur `NSObject`-Klasse, sofern die Methoden nicht bereits vorhanden sind. Der Code muss durchlaufen werden, bevor kritische API-Aufrufe die Klasse oder deren Instanzen erreichen können. Denkbar ist ein Hinzufügen der Methoden direkt nach dem Start der Anwendung oder im Anschluss

[151] Ohne die Implementierung der beiden Hilfsfunktionen (Zeile 12/13) ist der Code nicht ausführbar

an das Laden einer Klasse. So oder so müssen die kritischen Methoden einer Klasse in einem Zug hinzugefügt werden, unabhängig davon, ob sie später überhaupt verwendet werden. Für die dynamische Nachrichtenauflösung in Stufe 2 muss ähnlicher Code generiert werden, der jedoch zum spätestmöglichen Zeitpunkt (engl.: „**lazy**“) für jede nicht gefundene Methode einzeln ausgeführt wird. Die explizite Verfügbarkeitsprüfung (Zeile 9) entfällt dann ebenso wie die Iteration über alle kritischen Selektoren einer Klasse (Zeile 3). Werden alle Ausweichimplementierungen wie in Codebeispiel 4.3 nacheinander durch Aufrufe an die Runtime-Bibliothek hinzugefügt (Zeile 16), besteht bei einer hohen Anzahl kritischer Methoden das Risiko einer zeitlichen Verzögerung, die mit Anforderung AF.3 (Laufzeitperformance) unvereinbar wäre. Insbesondere sollten Verzögerungen beim initialen Start einer Anwendung vermieden werden [Tur12].

Um die Funktionsaufrufe an die Runtime-Bibliothek zu minimieren, kann ein Versuch unternommen werden, eine Ausweichklasse zu generieren, der bereits zur Compile-Zeit alle kritischen APIs samt Ausweichimplementierung hinzugefügt werden. Zur Laufzeit muss die Ausweichklasse in die bestehende **Klassenhierarchie** eingefügt werden, was mithilfe der Runtime-Bibliothek umgesetzt werden kann. Die Implementierung kann mit Codebeispiel 4.4 nachvollzogen werden, während Abbildung 4.8 das Resultat grafisch darstellt.

```
1  // Annahme: UIView enthalte kritische Methoden
2  Class targetClass = [UIView class];
3  // SHView sei die Ausweichklasse, bereitgestellt von der
4  // Kompatibilitätsbibliothek
5  Class fallbackClass = [SHView class];
6
7  // Bestimmung der bisherigen Superklasse
8  Class originalSuperclass = class_getSuperclass(targetClass);
9  // Die Ausweichklasse soll von der gleichen Superklasse wie UIView erben
10 class_setSuperclass(fallbackClass, originalSuperclass);
11 // UIView soll direkt von der Ausweichklasse und indirekt von der
12 // ursprünglichen Superklasse erben
13 class_setSuperclass(targetClass, fallbackClass);
```

Codebeispiel 4.4: Einfügen einer Ausweichklasse in die Klassenhierarchie in Objective-C

Kritische Methodenaufrufe, die die Empfängerklasse selbst nicht verarbeiten kann (Abbildung 4.8, `criticalAPI2`), werden automatisch an die Ausweichklasse weitergeleitet. Ist eine native Implementierung vorhanden (`criticalAPI1`), überschreibt diese die Ausweichimplementierung der eingefügten Ausweichklasse. Bei der Umsetzung muss beachtet werden, dass `super`-Aufrufe der Empfängerklasse von der Ausweichklasse ignoriert und stattdessen von der ursprünglichen Superklasse verarbeitet werden müssen. Dies ist in Abbildung 4.8 am Beispiel der nicht-kritischen `init`-Methode dargestellt. Außer in Fällen, in denen eine kritische API fehlt, muss die Ausweichklasse vollkommen transparent sein, um das Verhalten der Frameworkklassen nicht durch ungewünschte Seiteneffekte zu manipulieren.

Zusammenfassend lässt sich sagen, dass das Einbinden der Kompatibilitätsbibliothek auf viele verschiedene Weisen umgesetzt werden kann. Die einfachste ist die statische Umleitung aller kritischen API-Aufrufe, was geringfügige Änderungen im Anwendungsquellcode erfordert. Andern-

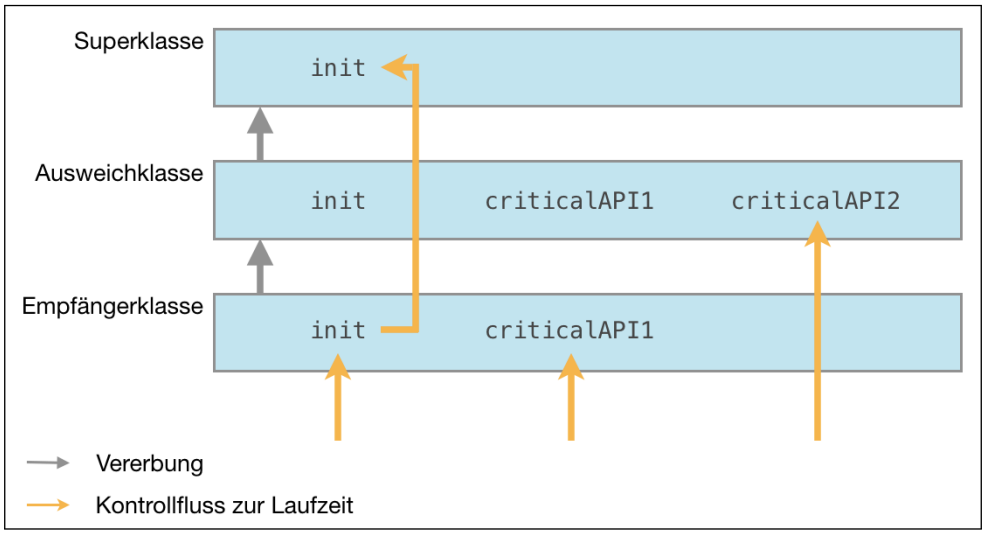


Abbildung 4.8: Manipulierte Klassenhierarchie nach Einfügen einer Ausweichklasse

falls muss die Programmiersprache die Möglichkeit bieten, API-Aufrufe zur Laufzeit nach Bedarf umzuleiten. Dies kann präventiv geschehen, indem alle notwendigen Umleitungen beim Start der Anwendung aktiviert werden. Andernfalls werden die Umleitungen erst geschaltet, wenn der kritische API-Aufruf erstmalig gescheitert ist. Dieses Vorgehen wird als lazy bezeichnet.

4.6 ZUSAMMENSETZUNG UND KONKRETISIERUNG EINES GESAMTKONZEPTS

Alle untersuchten und die für das Gesamtkonzept gewählten Teilkonzepte werden in Abbildung 4.9 veranschaulicht. Aus der Abbildung ist auch ersichtlich, welche Teilkonzepte sich gegenseitig bedingen und welche Techniken unabhängig voneinander eingesetzt werden können. Die Teilkonzepte der Detektions- und der Auflösungsphase werden durch den Dateiexport der Detektionsergebnisse voneinander entkoppelt. Die in der Abbildung markierten Entscheidungen für oder gegen bestimmte Teilkonzepte werden in den folgenden Unterabschnitten schrittweise begründet.

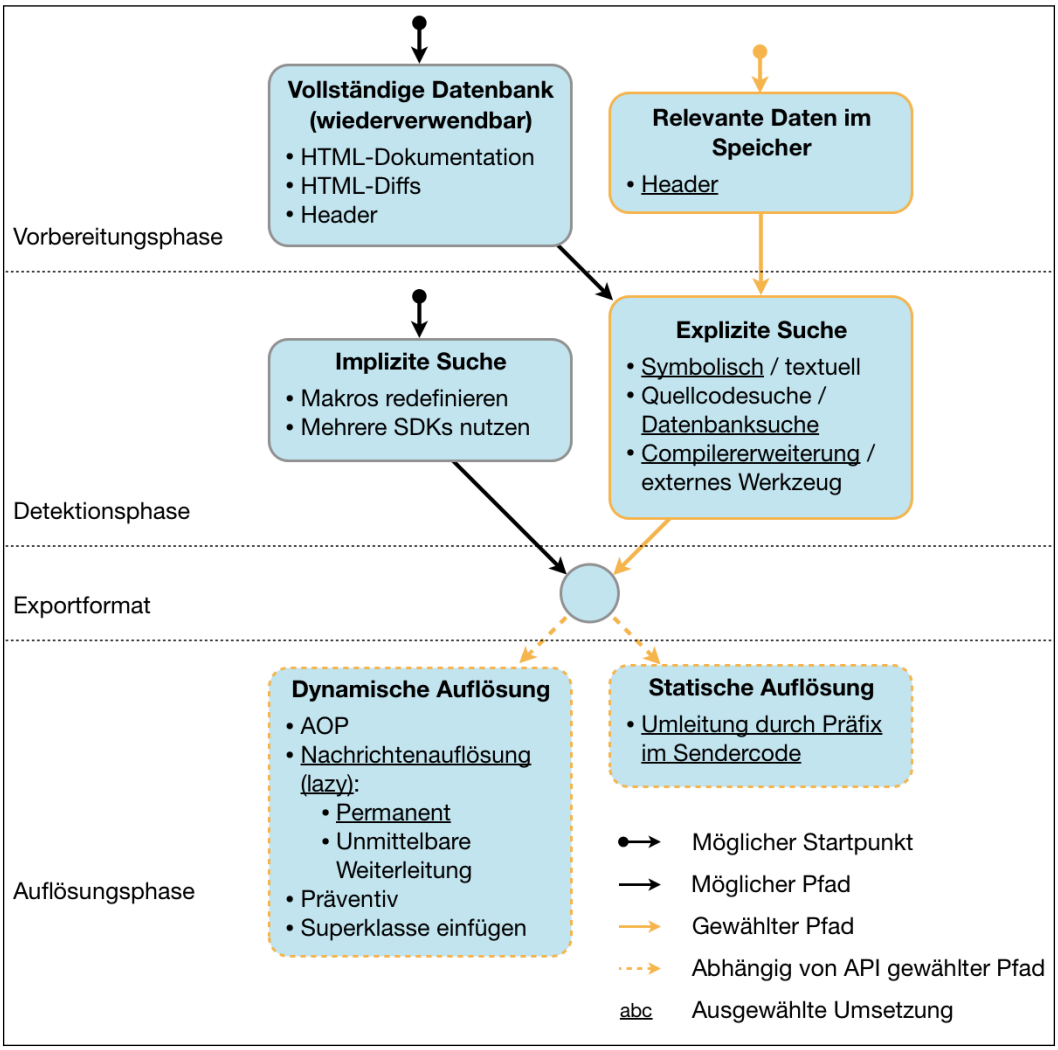


Abbildung 4.9: Auswahl der Teilkonzepte für ein Gesamtkonzept

4.6.1 Vorbereitungs- und Detektionsphase

Wenngleich die Vorbereitungsphase in der zeitlichen Abfolge vor der Detektionsphase liegt, muss zuerst entschieden werden, wie die Detektionsphase umgesetzt werden soll. Die Vorbereitungs-

phase dient lediglich dazu, eine effiziente Durchführung der Detektionsphase zu ermöglichen und muss dieser angepasst werden.

Schon während der Betrachtung der verschiedenen Detektionskonzepte in Abschnitt 4.4 wurde deutlich, dass nur **compilergestützte Werkzeuge**[152] eine zuverlässige, performante Suche nach kritischen APIs im Quellcode ermöglichen. Ferner ist zu erwarten, dass compilergestützte Werkzeuge auf die expandierten Dateien des Präprozessors zugreifen können und die Vorbereitungsphase keiner eigenen Implementierung bedarf.

Wird auf Basis von Compiler-Bibliotheken ein alleinstehendes Werkzeug entwickelt, muss der gesamte Quellcode einmal für die Detektion und ein weiteres Mal für die Kompilierung durchlaufen werden. Damit der zusätzliche Zeitaufwand tolerierbar bleibt, muss in der Detektionsphase ein potenziell komplexer Caching-Mechanismus implementiert werden, um nur in neu hinzugefügten oder veränderten Quellcodedateien nach kritischen Symbolen zu suchen. Deshalb sieht das Konzept dieser Arbeit vor, die Detektion direkt im Kompilierungsvorgang durchzuführen, indem der bestehende **Compiler modifiziert** wird. Wie in Abschnitt 4.4 unter Einbeziehung des Codebeispiels 4.2 dargelegt wurde, erfolgt die Vorbereitungsphase automatisch im Rahmen des Kompilierungsvorgangs. Die Verfügbarkeitsinformationen der iOS-Frameworks werden über standardisierte Attribute bereitgestellt, die sowohl GCC als auch Clang einlesen. Bevor die Symbole des Anwendungsquellcodes geparst werden, befinden sich somit sämtliche relevanten Verfügbarkeitsinformationen bereits im Speicher, ohne dass dafür zusätzliche Zeit oder Ressourcen verbraucht worden sind. Beim Parsen eines Symbols muss nun lediglich ein Vergleich zwischen der Verfügbarkeit der Symboldeklaration und dem Deployment Target implementiert werden. Das sequentielle Parsen des Codes und der Abgleich mit der Datenbasis entspricht konzeptionell der in Abschnitt 4.4 beschriebenen Datenbanksuche, wobei sich die Datenbasis direkt im Speicher befindet. Der Compiler verfügt über eine umfangreiche Optimierung durch Caches, sodass bei inkrementellen Builds nur veränderte Quellcodedateien berücksichtigt werden.

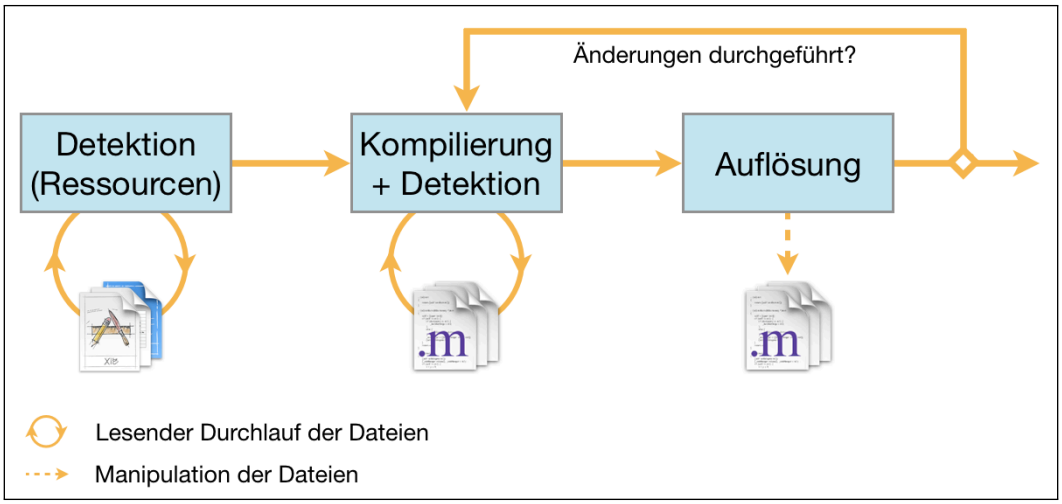


Abbildung 4.10: Detektion und Kompilierung in einem Durchlauf des Quellcodes: Erweiterung für Ressourcendateien

[152] Hiermit sind sowohl Werkzeuge gemeint, die auf Compiler-Bibliotheken basieren, als auch solche, die in den bestehenden Compiler integriert sind

Nicht alle Kompatibilitätsprobleme werden durch kritische APIs im Quellcode verursacht, auch bestimmte Konfigurationseinstellungen in der Projektdatei und die Verwendung kritischer APIs in den XML-basierten GUI-Dateien (XIB- und Storyboard-Dateien) können Laufzeitprobleme auf Legacy-Systemen verursachen. Weil diese Dateien (**Ressourcendateien**) nicht vom C-Compiler durchlaufen werden, müssen sie in einer zusätzlichen Build-Phase nach kritischen APIs durchsucht werden. Die Erweiterung um diese Build-Phase ist in Abbildung 4.10 grafisch dargestellt. Eine Auflösung der Kompatibilitätsprobleme in den Ressourcendateien ist im Rahmen dieser Arbeit nicht vorgesehen, stattdessen soll auf mögliche Konfigurationsprobleme hingewiesen werden und gegebenenfalls ein Vorschlag zur Behebung unterbreitet werden.

Die Ergebnisse der Detektionsphase sollen direkt in der IDE in Form von Warnungen sichtbar werden und in ein maschinenlesbares Dateiformat exportiert werden, um von der Auflösungsphase oder anderen Werkzeugen weiterverarbeitet werden zu können. Zu den Namen der gefundenen APIs müssen möglichst umfangreiche Metainformationen wie der Name der Quellcodedatei, die Zeilennummer, und die genaue Verfügbarkeit exportiert werden. So kann später nachvollzogen werden, welche Stellen des Anwendungscodes die Generierung von Ausweichimplementierungen ausgelöst haben, und unter welchen Voraussetzungen die Ausweichimplementierung greift.

4.6.2 Auflösungsphase

Die Auflösungsphase nutzt die maschinenlesbare Ausgabe der Detektionsphase und soll als **separate Build-Phase** umgesetzt werden. Die direkte Integration der gesamten Auflösungsphase in den Kompilierungsvorgang ist denkbar, aber problematisch, weil der Compiler selbst stetig weiterentwickelt wird. Ein regelmäßiger Merge-Vorgang des modifizierten und unmodifizierten Compilercodes ist nur dann ohne erheblichen Aufwand möglich, wenn sich die Modifikationen auf kleine Teile der Codebasis beschränken. Die Codegenerierung der Auflösungsphase würde jedoch erhebliche Änderungen am Compilercode bedingen, die es zu vermeiden gilt. Auch um das Vertrauen der Entwickler in den Compiler aufrecht zu halten, sollten die Änderungen minimal sein und weder den Quell- noch den generierten Maschinencode beeinflussen. Eine klare Trennung zwischen der Detektions- und der Auflösungsphase hat den weiteren Vorteil, dass die Implementierung einer Phase leicht ausgetauscht werden kann, ohne die andere Phase zu beeinträchtigen.

Die Auflösung, angefangen mit der Einbindung der Kompatibilitätsbibliothek, muss sich an den Typ einer kritischen API anpassen. Erlaubt ein API-Typ die verfügbarkeitsabhängige, empfängerseitige Umleitung, sollen die Aufrufe der APIs dynamisch an die Kompatibilitätsbibliothek umgeleitet werden, da so der Anwendungsquellcode unverändert bleiben kann. Als Lösungsmöglichkeiten diskutiert wurden die Umleitung mittels AOP, die dynamische Nachrichtenauflösung in Phase 2 (permanente Auflösung) oder 3 (Weiterleitung), die präventive permanente Auflösung, und das Einfügen einer Ausweichklasse in die Klassenhierarchie. Die Implementierung einer allgemein verwendbaren, stabilen AOP-Technologie für Objective-C ginge weit über die eigentlichen Ziele dieser Arbeit hinaus. Die Manipulation der Klassenhierarchie erfordert die Benutzung der `class_setSuperclass()`-Funktion, von deren Benutzung in der Dokumentation abgeraten wird, was möglicherweise Auswirkungen auf die App-Store-Kompatibilität (s. Abschnitt 3.3.5)

und damit Anforderung AF.4 hat. Zudem haben praktische Versuche im Rahmen dieser Arbeit ergeben, dass die Stabilität der Anwendung durch ABI-Probleme beeinträchtigt wird. Ähnliche Probleme mit `class_setSuperclass()` wurden in einem Open-Source-Projekt[153] von Steinberger dokumentiert [Ste13]. Wie in Abschnitt 4.5 bereits erklärt wurde, ist die **spätestmögliche, permanente Nachrichtenauflösung** der Nachrichtenweiterleitung oder der präventiven Auflösung aus Performancegründen vorzuziehen. Deshalb wird dieser Ansatz für das Konzept zur Auflösung dynamisch veränderbarer APIs ausgewählt und später implementiert.

Von den in dieser Arbeit relevanten API-Typen werden ausschließlich Objective-C-Methoden dynamisch aufgelöst. Objective-C-Klassen lassen sich zwar zur Laufzeit manipulieren und neu hinzufügen, allerdings fügen sich zur Laufzeit erzeugte Klassen nicht transparent in die Anwendung ein, weil sie unter Verwendung der Standardsyntax nicht gefunden werden. Dies soll anhand des Codebeispiels 4.5 verdeutlicht werden.

```
1  // Verfügbarkeitsprüfung
2  if (![PKPass class]) {
3      // Klasse wird zur Laufzeit erzeugt und registriert
4      // Anm.: Die Funktion muss separat implementiert werden
5      sh_createAndRegisterClassWithName(@"PKPass");
6  }
7
8  PKPass *instance;
9  // Standardsyntax zum Laden einer nativen Klasse
10 instance = [[PKPass alloc] init];
11 // Syntax zum Laden einer nativen oder dynamisch hinzugefügten Klasse
12 instance = [[NSClassFromString(@"PKPass") alloc] init];
```

Codebeispiel 4.5: Dynamisches Laden einer kritischen Objective-C-Klasse

Auf Legacy-Systemen würde die Allokation der Klasse mit `alloc/init` und der Verwendung des Klassensymbols (Codebeispiel 4.5, Zeile 10) fehlschlagen[154], nur das Laden der Klasse mithilfe von Schnittstellen der Runtime-Bibliothek oder darauf aufbauenden Hilfsfunktionen (Zeile 12) führt zum Erfolg. Das Hinzufügen eines Präfixes zum Klassennamen ist besser nachvollziehbar und leichter umzusetzen als die Veränderung der Allokationssyntax aller kritischen Klassen. Kritische Klassen werde daher genauso wie C-APIs **statisch** durch das Hinzufügen eines Präfixes zur Kompatibilitätsbibliothek umgeleitet. Wird eine statisch aufzulösende, kritische API erkannt, sollte der Compiler zusätzlich zu der normalen Warnung einen Änderungsvorschlag unterbreiten, aber den Code nicht selbstständig ändern.

Das **Gerüst** der Kompatibilitätsbibliothek soll anhand der Ergebnisse der Detektionsphase **automatisch generiert** werden. Die Kompatibilitätsbibliothek spiegelt eine Untermenge[155] der Framework-Schnittstellen, wobei sich die APIs der Kompatibilitätsbibliothek (Ausweich-APIs) durch einen Präfix von den nativen APIs unterscheiden. Auch Teile der Funktionalität der Kompatibilitätsbibliothek können automatisch generiert werden. Dabei muss erneut zwischen statisch

[153] `https://github.com/steipete/PSTCollectionView/`
[154] Der Wert von `instance` bliebe bei `nil`
[155] Nicht verwendete und nicht kritische APIs werden übersprungen

und dynamisch aufgelösten APIs unterschieden werden. Bei dynamisch aufgelösten APIs erfolgt die Verfügbarkeitsabfrage vor dem Aufruf der Ausweich-API. Die gesamte Implementierung der Ausweich-API entspricht daher der Ausweichimplementierung, wie aus Codebeispiel 4.6 ersichtlich wird.

```
// Ausweich-Methode zu 'criticalAPI:'
// Die Methode wird nur aufgerufen, wenn die native API fehlt
- (void) sh_criticalAPI:(id)sender {
    /*
     *  Ausweichimplementierung
     */
}
```

Codebeispiel 4.6: Stub einer dynamisch aufgelösten API am Beispiel einer Objective-C-Methode

Dagegen werden statisch aufgelöste Aufrufe unabhängig von der Verfügbarkeit einer API zunächst auf die Ausweich-API umgeleitet, deren Implementierung die Verfügbarkeitsabfrage und einen verzweigten Kontrollfluss enthält. Wie aus Codebeispiel 4.7 hervorgeht, enthält einer der Pfade die Weiterleitung zur nativen API und der alternative Pfad die eigentliche Ausweichimplementierung. Die konkrete Ausgestaltung der Stubs ist abhängig vom Typ einer API und orientiert sich an den Best Practices aus Abschnitt 3.4.3.

```
// Ausweich-Funktion zu 'CriticalAPI()'
// Die Funktion wird immer anstelle der nativen API aufgerufen
void SH_CriticalAPI(id sender) {
    if (CriticalAPI != NULL) {
        // Pfad 1: Weiterleitung an die native API
        CriticalAPI(sender);
        return;
    } else {
        /*
         *  Pfad 2: Ausweichimplementierung
         */
    }
}
```

Codebeispiel 4.7: Stub einer statisch aufgelösten API am Beispiel einer C-Funktion

Neben den Stubs und dem Weiterleitungscode können auch Teile der Ausweichimplementierung automatisch generiert werden, wenn zusätzliches Wissen über die API bereitgestellt wird. Die Bereitstellung des Zusatzwissens kann über ein öffentliches Git-Repository (**Code-Repository**) erfolgen, sodass die Entwicklergemeinschaft aktiv mitwirken und von einmal bereitgestelltem Wissen profitieren kann. Es sollen drei Fälle unterschieden werden:

- **Leere oder eigene Ausweichimplementierung:** Es wird kein zusätzliches Wissen über die API bereitgestellt, weshalb die Ausweichimplementierung leer bleibt. Abstürze werden bereits durch das Vorhandensein bzw. die Einbindung einer Ausweich-API vermieden.

Eine Warnung kann den Entwickler darauf hinweisen, dass ein Stub generiert wurde. Der Entwickler kann anschließend entscheiden, ob das bloße Überspringen der API (leere Ausweichimplementierung) ausreichend ist, oder er selbst einen Teil der Funktionalität nachimplementiert. In jedem Fall sollte es dem Entwickler möglich sein, die Warnung nach der Begutachtung des generierten Codes einfach zu entfernen.

- **Direktes Einfügen einer allgemeingültigen Ausweichimplementierung:** Für einige kritische APIs lässt sich eine allgemeingültige, anwendungsübergreifende Ausweichimplementierung definieren. Ein solcher Fall ist in Codebeispiel 4.8 gezeigt. Die im Beispiel gewählte Funktion überprüft die mit iOS 6 eingeführte und seitdem verpflichtende Abfrage nach dem Zugriffsrecht auf das Adressbuch. Weil das Zugriffsrecht auf Legacy-Systemen vom Nutzer nicht entzogen werden kann, ist die abgebildete Ausweichimplementierung allgemeingültig. Im Code-Repository muss eine Abbildung vom eindeutigen Namen einer API auf den einzufügenden Code hinterlegt werden.

- **Umleitung auf eine allgemeingültige Ausweichklasse:** Wenn die allgemeingültige Ausweichimplementierung aus wenigen Zeilen Code oder der Umleitung auf eine andere, nichtkritische API besteht, kann die Ausweichimplementierung direkt in die Ausweich-API eingefügt werden. Ist die Ausweichimplementierung komplexer, sollte sie in eine zusätzliche Klasse ausgelagert werden. Für einige kritische Klassen und Funktionalitäten sind abwärtskompatible Open-Source-Lösungen verfügbar. Ziel der Auflösung soll es sein, solche Lösungen automatisch einzubinden. Im Code-Repository muss anstelle der gesamten Ausweichimplementierung nur der Name der zusätzlichen Klasse hinterlegt werden. Eine Erweiterung dieses Ansatzes sieht vor, dass neben dem Klassennamen auch das Quellcoderepository der zusätzlichen Klasse(n) angegeben und genutzt werden kann.

```
1  // Kritische Funktion, die die Zugriffsrechte auf das Adressbuch erfragt
2  ABAuthorizationStatus SH_AddressBookGetAuthorizationStatus() {
3      if (ABAddressBookGetAuthorizationStatus != NULL) {
4          // Nativ verfügbar: Weiterleitung
5          return ABAddressBookGetAuthorizationStatus();
6      } else {
7          // In früheren Versionen kann der Zugriff nicht verweigert werden
8          return kABAuthorizationStatusAuthorized;
9      }
10 }
```

Codebeispiel 4.8: Allgemeingültige Ausweichimplementierung am Beispiel einer C-Funktion

Mit dem vorgestellten Konzept umfasst die Auflösung einer API die Einbindung einer automatisch generierten Ausweichimplementierung, die bei entsprechendem Zusatzwissen die native API vollständig nachbildet. Fehlt dieses Zusatzwissen oder kann nicht von einer allgemeingültigen Ausweichimplementierung ausgegangen werden, wird dem Entwickler ein Codegerüst zur Verfügung gestellt, das die Priorisierung der nativen API (Anforderung AF.2) bereits umsetzt. Alle Deklarationen der Kompatibilitätsbibliothek sollten mit den zusätzlichen Informationen aus der Detektionsphase in Form von Kommentaren annotiert werden, um Anforderung AF.6 (Umfangreiche Diagnostikinformationen) zu genügen.

4.6.3 Bedingungen an die Umsetzbarkeit des Konzepts

Bei der Erstellung des Konzepts wurde von Beginn an auf die Umsetzbarkeit auf der iOS-Plattform geachtet. Nichtsdestotrotz können Teile des Konzepts auch auf anderen Plattformen umgesetzt werden. Im Folgenden sollen alle wichtigen Grundvoraussetzungen zusammengefasst und kurz erläutert werden. Ist eine Bedingung nicht gegeben, können in den Abschnitten 4.3 bis 4.5 möglicherweise alternativ anwendbare Konzepte gefunden werden.

- *Kritische APIs bestimmbar:* Analog zu Abschnitt 3.4.1 dieser Arbeit muss vor einer Portierung des Konzepts überprüft werden, ob die Definition einer kritischen API anwendbar ist.
- *Annotierte Header-Dateien:* Die Annotation der Framework-APIs mit ihren Verfügbarkeitsinformationen in Form von standardisierten Attributen erlaubt den faktischen Wegfall[156] der Vorbereitungsphase.
- *Quelloffener Compiler:* Zur Integration der Detektionsphase in den bestehenden Compiler muss dessen Quellcode zugänglich sein. Um Kompatibilitätsprobleme auf binärer Ebene zu vermeiden, sollte kein anderer als der Standardcompiler zugrunde gelegt werden.
- *Einbindung des modifizierten Compilers:* Damit die Detektionsphase in die IDE-Workflows eingebunden werden kann, muss der Standard-Compiler durch den um die Detektionsphase erweiterten Compiler ersetzt werden können.
- *Erweiterung des Build-Prozesses:* Um den Build-Prozess um die Phasen der Detektion in Ressourcendateien und der Auflösung zu erweitern, muss der Build-Prozess modifizier- oder erweiterbar sein, beispielsweise durch das Hinzufügen eigener Skripte.
- *Steuerung des Build-Prozesses:* Muss der Build-Vorgang im Anschluss an die Auflösung neu gestartet werden, sollte dies automatisch ausgelöst werden. Wenn ein programmgesteuerter Neustart nicht möglich ist, sollte zumindest ein Abbruch des aktuellen Build-Vorgangs stattfinden, um den Entwickler auf die Veränderung hinzuweisen.
- *Darstellung von Warnungen:* Warnungen, die insbesondere während der beiden Detektionsphasen erzeugt werden, sollten sich nahtlos in die IDE integrieren und von bestehenden Warnungen nicht unterscheidbar sein.
- *Verfügbarkeitsprüfung zur Laufzeit:* Um native APIs nicht grundsätzlich zu überschreiben, muss vor dem Aufruf der Ausweichimplementierung die Verfügbarkeit der nativen Implementierung überprüft werden.
- *Dynamische Auflösungsmöglichkeiten:* Die dynamische Auflösung kann nur durch die Unterstützung der Programmiersprache sauber umgesetzt werden. Andernfalls kann auf die statische Auflösung zurückgegriffen werden, die im Konzept dieser Arbeit für C-Funktionen und Objective-C-Klassen zum Einsatz kommt. Existiert eine produktiv einsetzbare AOP-Technologie, kann die Einbindung der Kompatibilitätsbibliothek besonders elegant umgesetzt werden, weil der eigentliche Auflösungsmechanismus hinter der AOP-Technologie verborgen bleibt.

[156] Die Vorbereitungsphase findet dennoch statt, muss aber nicht gesondert implementiert oder eingebunden werden

Die Erfüllung der meisten Bedingungen unter iOS wurde in den vergangenen Kapiteln bereits ausführlich besprochen, die Diskussion der bislang übergangenen Voraussetzungen wird in Kapitel 5 nachgereicht.

4.7 ZUSAMMENFASSUNG

Nach gründlicher Diskussion verschiedener Teilkonzepte wurde in diesem Kapitel ein Gesamtkonzept für die Detektion und Auflösung kritischer APIs erarbeitet. Einen zentralen Stellenwert nimmt der Compiler ein, der so modifiziert wird, dass er kritische Symbole im Quellcode findet und exportiert. Die Datenbasis liefern zuvor eingelesene Attribute der API-Deklarationen aus den Header-Dateien. Für die Detektion kritischer APIs in Ressourcendateien ist eine zusätzliche Build-Phase nötig.

Die Auflösungsphase wird in einer weiteren Build-Phase umgesetzt und generiert Code, der den Absturz auf Legacy-Geräten verhindert und im Idealfall die volle Funktionalität der nativen APIs nachimplementiert. Die Einbindung des generierten und in eine Bibliothek ausgelagerten Codes in die bestehende Anwendung erfolgt je nach Art der API statisch oder dynamisch. Der gesamte, stark vereinfachte Ablauf eines Build-Prozesses ist in Codebeispiel 4.9 in Form von Pseudocode und Kommentaren dargestellt.

```
/*  ##########################################################################
 *  (1) Ressourcendetektion: Iteriere über unterstützte Ressourcendateien
 *      Umsetzung: Skript-Build-Phase                                        */
for (File *f in supportedResourceFiles) {
    // (1.1) Datei nach kritischen APIs oder Konfigurationen durchsuchen
    if ( /* Probleme gefunden */ )
        // (1.2) Warnungen in Xcode anzeigen
}
/*  ##########################################################################
 *  (2) Kompilierung inkl. Detektion: Iteriere über Quellcodedateien
 *      Umsetzung: Modifikation des Standard-Compilers                       */
for (File *f in sourceFiles) {
    // (2.1) Präprozessor: Imports auflösen [unverändert]
    // (2.2) Parser: Eingefügte Symboldeklarationen einlesen [unverändert]
    // (2.3) Parser: Anwendungsquellcode einlesen [modifiziert]
    for (Symbol *s in sourceCodeSymbols) {
        // (2.4) Verfügbarkeit eines Symbols prüfen
        if (s.availability > DEPLOYMENT_TARGET) {
            // (2.5) Warnung in Xcode anzeigen
            // (2.6) Warnung/Symboldefinition exportieren
            if ( /* Symbol muss statisch aufgelöst werden */ )
                // (2.7) Änderungsvorschlag: s.name = PREFIX + s.name;
        }
    }
}
/*  ##########################################################################
 *  (3) Auflösung: Iteriere über Exportliste der Detektion
 *      Umsetzung: Skript-Build-Phase                                        */
BOOL requiresRebuild = NO // Rekompilierung bei Codeänderungen
for (Symbol *s in exportedSymbols) {
    if ( /* Ausweich-API existiert bereits */ )
        continue;
    if ( /* Symbol kann dynamisch aufgelöst werden */ )
        // (3.1) Generiere Code für dynamische Einbindung
    // (3.2) Generiere API-Stub
    if ( /* Symbol muss statisch aufgelöst werden */ )
        // (3.3) Füge Verfügbarkeitsabfrage in Stub ein
    if ( /* Symbol wird im Code-Repository gefunden */ )
        // (3.4) Füge Ausweichcode in Stub ein
    requiresRebuild = YES;
}
if (requiresRebuild)
    // Springe zu (2)
```

Codebeispiel 4.9: Gesamtkonzept zur Detektion und Auflösung kritischer APIs in Pseudocode

5 PROTOTYPISCHE IMPLEMENTIERUNG

Als Beleg für die Umsetzbarkeit des vorgestellten Konzepts wurde im Rahmen der Arbeit ein Prototyp implementiert, der alle wesentlichen Elemente des Konzepts beinhaltet und Thema dieses Kapitels ist. In Abschnitt 5.1 werden die generellen Anforderungen an das Konzept um implementierungsspezifische Anforderungen erweitert. Abschnitt 5.2 fasst die wichtigsten Implementierungsentscheidungen und -techniken zusammen, die im Prototyp Anwendung gefunden haben. Unter Abschnitt 5.3 wird erklärt, warum einige Sonderfälle einer speziellen, unter Umständen vom allgemeinen Konzept abweichenden Behandlung bedürfen. Die Installation und Nutzung des implementierten Werkzeugs wird in Abschnitt 5.4 beschrieben, anschließend folgt eine Zusammenfassung aktueller Limitierungen, die den Prototyp derzeit von einem produktiv einsetzbaren Werkzeug unterscheiden. Das Kapitel schließt mit einer Zusammenfassung in Abschnitt 5.6.

5.1 ERWEITERTE ANFORDERUNGEN

Alle in Abschnitt 4.1 zusammengetragenen Anforderungen gelten weiterhin und werden um folgende, implementierungsspezifische Anforderungen ergänzt. Auch diese Anforderungen stützen sich auf die Ergebnisse der initialen Entwicklerbefragung.

AF.7 **Ausführung bei jedem Build:** Die zwölf befragten Entwickler haben sich einstimmig dafür entschieden, dass die Detektion kritischer APIs bei jedem Kompilierungsvorgang ohne Zutun des Entwicklers durchgeführt werden soll. Als Alternative wurde die Integration in den Static Analyzer, der vom Entwickler selbstständig gestartet werden muss, angeboten. Durch die Entscheidung für den Kompilierungsvorgang können Fehler zuverlässiger[157] und rechtzeitig gefunden werden. Die Anforderung kann als Erweiterung von AF.1 (Integra-

[157] Zuverlässiger in dem Sinne, dass nicht jeder Entwickler den Static Analyzer regelmäßig nutzt

tion) verstanden werden und stärkt die Relevanz der Compile-Zeit-Performance (Anforderung AF.3) wegen der hohen Ausführungsfrequenz.

AF.8 **Ausführung mit Protokollierung:** Alle befragten Entwickler haben sich für eine unterbrechungsfreie Ausführung des Werkzeugs ausgesprochen. Diese Entscheidung ist für die Codegenerierung während der Auflösungsphase relevant. Stehen verschiedene Codevarianten zur Auswahl, soll der Entwickler nicht während des Build-Vorgangs zu einer Entscheidung gedrängt werden. Stattdessen sollen mögliche Probleme und Optionen protokolliert werden, sodass sie im Anschluss an den Build-Prozess inspiziert werden können. Diese Anforderung baut auf AF.6 (Umfangreiche Diagnostikinformationen) auf.

5.2 MASSNAHMEN UND IMPLEMENTIERUNG

Der Prototyp dieser Arbeit besteht aus drei einzelnen Werkzeugen sowie einer Installationsanwendung. Die Werkzeuge implementieren die drei Phasen des Gesamtkonzepts und werden in den folgenden Abschnitten vorgestellt. Die Implementierung erfolgt auf Basis der aktuellen, öffentlich verfügbaren Plattformen und Werkzeuge:

- Mac OS X 10.8 (Mountain Lion)
- Xcode 4.6
- iOS 6.1

Die Frameworks stellen APIs verschiedenster Typen zur Verfügung, doch nicht für alle Typen ist die Detektion und Auflösung gleichermaßen sinnvoll. Untenstehend ist zusammengefasst, welche Mechanismen für welche API-Typen angewandt werden.

- **Detektion und dynamische Auflösung:** Ausschließlich Objective-C-*Methoden* (Instanzmethoden, Klassenmethoden und Properties) werden dynamisch aufgelöst, die Begründung kann in Abschnitt 4.6.2 nachgelesen werden.
- **Detektion und statische Auflösung:** Objective-C-*Klassen* und C-*Funktionen* werden statisch aufgelöst.

Werden Symbole statisch oder dynamisch aufgelöst, soll die Warnung über eine kritische API standardmäßig[158] nicht angezeigt werden, schließlich wird der Grund der Warnung in der Auflösungsphase behoben[159].

- **Detektion:** Einige kritische Symbole verursachen keine direkten Laufzeitprobleme, weil sie bereits zur Compile-Zeit aufgelöst werden. Dazu gehören die in Abschnitt 3.4.3 betrachteten Compile-Zeit-Konstanten (*Enums*) und Objective-C-*Protocols*[160]. Die Auflösung

[158] Die Warnungen können über Compiler-Flags auf Wunsch aktiviert werden

[159] Bei der statischen Auflösung wird ein Änderungsvorschlag (Präfix) unterbreitet – wird dieser akzeptiert, ist die API auch nicht mehr als kritisch erkennbar

[160] Die in den Protocols definierten Methoden können Laufzeitfehler verursachen, nicht aber die Schnittstellenbeschreibungen selbst

erübrigt sich in diesen Fällen, die Detektion soll dennoch durchgeführt werden. Kritische Laufzeit-Konstanten (*Extern-Variablen*) lösen dagegen Laufzeitkonflikte aus. Allerdings werden Extern-Variablen in der Regel für vorgegebene Parameter-Werte[161] der Framework-APIs genutzt, und die Auflösung der Konstantenwerte hilft nicht darüber hinweg, dass ältere Frameworks die kritischen Werte nicht entgegennehmen oder der Entwicklerintention entsprechend verarbeiten können. Die automatische Auflösung würde den Fokus des Entwicklers auf die Auflösung des Konstantenwertes lenken, tatsächlich kritisch ist aber die Nutzung des aufgelösten Wertes zur Kommunikation mit dem Framework. Daher soll eine Warnung generiert werden, die Auflösung aber unterbleiben.

5.2.1 Detektion in Ressourcendateien: xcodeprojectcheck

Für die Detektion in Ressourcendateien wurde das Kommandozeilenwerkzeug *xcodeprojectcheck* in Objective-C implementiert. Auf dem derzeitigen Stand erkennt es **kritische**, nicht als optional markierte **Frameworks** im App-Target einer Projektdatei. Weil die Frameworks als Ganzes keine maschinenlesbaren Verfügbarkeitsinformationen bereitstellen, werden diese Informationen auf Grundlage der offiziellen Dokumentation[162] in Form einer PLIST-Datei bereitgestellt. Wegen der im Vergleich zu den einzelnen APIs überschaubaren Anzahl an Frameworks wird die PLIST derzeit manuell verwaltet und vom Installationsprogramm mitgeliefert, aber die automatische Extraktion der Daten aus der HTML-Dokumentation ist zukünftig denkbar.

Xcode-Projektdateien[163] liegen ungeachtet der `pbxproj`-Dateierweiterung im PLIST-Format vor, sodass der lesende Zugriff nach kurzer Einarbeitung in die Dateistruktur unproblematisch ist[164]. Wie auch bei XIB- und Storyboard-Dateien ist die semantische Struktur der Projektdateien nicht standardisiert oder dokumentiert, weshalb keine allgemeingültigen Ratschläge zur Verarbeitung der Dateien gegeben werden können. Alle in den Werkzeugen angewandten Erkenntnisse über die Ressourcendateien wurden durch Reverse Engineering herausgefunden.

Die **Einbindung** des Kommandozeilenwerkzeugs ist über Skript-Build-Phasen möglich, wie in Abbildung 5.1 gezeigt. Zuvor wird das Werkzeug während der Installation in ein bekanntes Unterverzeichnis des Projekts kopiert.

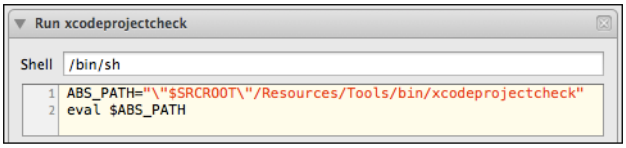


Abbildung 5.1: Einbindung von xcodeprojectcheck als Xcode-Build-Phase

Vor der Ausführung einer Skript-Phase setzt Xcode eine Vielzahl von Umgebungsvariablen, darunter das aktuelle Deployment Target und das Basisverzeichnis des Xcode-Projekts. Der Zugriff auf

[161] Beispielsweise Fehlerdomains und Hash-Schlüssel

[162] `http://developer.apple.com/library/ios/#documentation/miscellaneous/conceptual/iphoneostechoverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html`

[163] Die Projektstruktur wird in der `project.pbxproj`-Datei gespeichert, die sich innerhalb des `.xcodeproj`-Bundles befindet

[164] Zahlreiche Objective-C-APIs bieten einfache Lese- und Schreiboperationen auf PLIST-Dateien an, sodass für dieses Format keine eigenen Parser implementiert werden müssen

die Umgebungsvariablen mit Objective-C wird in Codebeispiel 5.1 demonstriert. Die Ausführung des xcodeprojectcheck-Werkzeugs erfordert keine gesonderten Kommandozeilenparameter, ist dafür aber von den Umgebungsvariablen abhängig.

```
1 // Bedingung: Verlinkung des Foundation-Frameworks
2 NSDictionary *environmentVars = [[NSProcessInfo processInfo] environment];
```

Codebeispiel 5.1: Zugriff auf die Umgebungsvariablen in Objective-C

In den Skriptphasen und den darin ausgeführten Programmen können durch einfache String-Konventionen **Warnungen** ausgegeben werden, die Xcode zusammen mit allen anderen Warnungen anzeigt. Die Syntax kann dem Codebeispiel 5.2 entnommen werden. Auf diese Weise werden auch die xcodeprojectcheck-Warnungen generiert, von denen eine exemplarisch in Abbildung 5.2 dargestellt ist.

```
1 printf("%s:%d: warning: Critical framework '%s'.\n",
2        filePath,
3        lineNumber,
4        frameworkName);
```

Codebeispiel 5.2: Generierung einer Xcode-Warnung aus einem C-Kommandozeilenwerkzeug

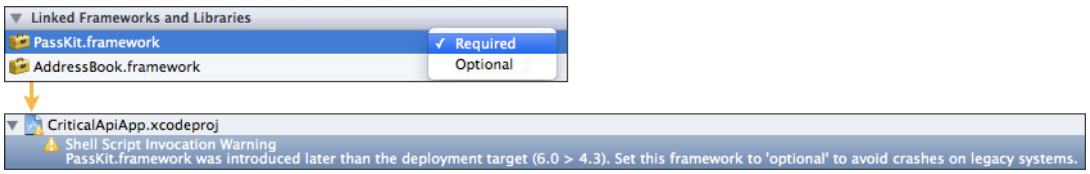


Abbildung 5.2: Warnung über ein kritisches Framework

Die Suche nach kritischen APIs in XIB- und Storyboard-Dateien ist nicht fertig implementiert, wurde aber bei der Konzeption des Werkzeugs bedacht und soll in einer späteren Version umgesetzt werden. Der Quellcode des Werkzeugs ist in Anlage B.1 enthalten.

5.2.2 Detektion im Quellcode: Modifizierter Clang-Compiler

Wie in Abschnitt 3.3.3 dargelegt wurde, hat Apple den GCC-Compiler inzwischen vollständig durch Clang ersetzt. Als Grundlage für eine Modifikation kommt deshalb nur der in C++ geschriebene Clang-Compiler in Frage. In [Her12] wird argumentiert, dass Clang auch für Compiler-Erweiterungen jenseits des Apple-Ökosystems GCC und anderen Compilern bzw. Bibliotheken vorzuziehen ist. Das Quellcode-Repository des Clang-Projekts umfasst 23 Bibliotheken, aus denen sich sechs ausführbare, Clang-basierte Werkzeuge und der Clang-Compiler selbst zusammensetzen. Insgesamt wurden Modifikationen an sechs der Bibliotheken durchgeführt, die im Folgenden begründet werden. Bei den Modifikationen wurde darauf geachtet, die in [llv13] aufgeführten Coding-Standards weitestgehend einzuhalten. Die Clang-Plugin-API wurde nicht in Betracht gezogen: Googles Chromium-Entwicklerteam rät ausdrücklich von der Nutzung ab, weil sie instabil und kaum dokumentiert sei [chr11].

Der Kompilierungsvorgang setzt sich intern aus mehreren Phasen zusammen[165], wobei jede der Phasen von einer separaten Bibliothek implementiert wird. Die *clangDriver*-Bibliothek koordiniert die Ausführung der einzelnen Phasen des Clang-Compilers.

Die umfangreichsten Änderungen wurden an der *clangSema*-Bibliothek durchgeführt. Das Sema-Modul führt eine **semantische Analyse** des Quellcodes durch und erzeugt schließlich den AST. Zur `Sema`-Klasse wurde eine private Funktion namens `CheckSafeAPIUsage()` hinzugefügt, die als Parameter eine Symboldeklaration und Kontextinformationen erwartet. In dieser Kernfunktion findet der Vergleich zwischen der **Verfügbarkeit** einer Symboldeklaration und dem Deployment Target statt. Weil die Funktion für alle potenziell kritischen Symbole aufgerufen wird, ist es entscheidend, dass sie frühstmöglich abbricht, wenn es sich nicht um ein kritisches Symbol handelt. Dass die Nutzung und Implementierung dieser Funktion keinen signifikanten Performancenachteil zur Folge hat, soll später in der Evaluation nachgewiesen werden. Wurde ein Symbol als kritisch eingestuft, wird erstens eine Warnung generiert und zweitens die Symboldeklaration zum späteren Export übergeben.

Die `CheckSafeAPIUsage()`-Funktion wird in vier existierenden Funktionen des Sema-Moduls aufgerufen. Alle Aufrufe bestehen aus wenigen Zeilen Code, der nur den Funktionsaufruf selbst und gegebenenfalls eine Anpassung der übermittelten Parameter enthält. Die folgenden Funktionen wurden um den Aufruf erweitert:

- `DiagnoseUseOfDecl()`: Detektion kritischer C-Funktionen, Enums, Typedefs und Extern-Variablen
- `BuildInstanceMessage()`: Detektion kritischer Objective-C-Methoden; erfasst werden auch Properties und sogenannte „Opaque Expressions“[166]
- `BuildClassMessage()`: Detektion kritischer Klassen und Klassenmethoden
- `ParseObjCSelectorExpression()`: Detektion kritischer Instanz- und Klassenmethoden, die mit `performSelector:` aufgerufen werden

Der Export kritischer Symbole in eine Datei soll erst am Ende der Quellcodeanalyse durchgeführt werden, um die Anzahl der benötigten Festplattenzugriffe zu minimieren; deshalb werden die gefundenen kritischen Symbole zunächst temporär im Speicher gehalten. Das modifizierte **Driver-Modul** löst nach der abgeschlossenen semantischen Analyse den Exportvorgang aus. Der Export wird standardmäßig übersprungen und kann durch das Setzen des `-export-unsafe-api-usage`-Flags aktiviert werden. Zur Definition des zusätzlichen Flags und zur Koordination des Export-Vorgangs mussten die Driver- und Frontend-Bibliotheken geringfügig modifiziert werden.

Auf die Schnittstellen der **Exportfunktionalität** muss sowohl aus dem Sema-Modul (Weitergabe der Symbole) als auch aus dem Driver-Modul (Start des Exportvorgangs) zugegriffen werden. Um keine zusätzlichen Abhängigkeiten einzuführen, aber den Konfigurationsaufwand eines neuen

[165] Eine Einführung in die wichtigsten Phasen eines Compilers wird in [Cha04] gegeben

[166] Einige rein syntaktische Spracherweiterungen werden vom Compiler in normale Methodenaufrufe umgewandelt, sind aber im Quellcode nicht als solche erkennbar – ein Beispiel ist die Subscripting-Syntax, die später in Abschnitt 5.3 thematisiert wird

Moduls zu vermeiden, wurde die Exportfunktionalität in der *clangBasic*-Bibliothek implementiert. Das Basic-Modul ist die einzige direkte, gemeinsame Abhängigkeit der Driver- und Sema-Module. Im Unterschied zu den anderen Modifikationen wurde die Exportfunktionalität in Objective-C statt C++ implementiert, um die einfache Generierung von PLIST-Dateien mit dem Foundation-Framework nutzen zu können. Eine beispielhafte Export-PLIST befindet sich in Anhang A.1. Durch die Bindung an das Apple-Framework geht die Plattformunabhängigkeit verloren, was jedoch aufgrund der Spezialisierung des Werkzeugs keine tatsächliche Einschränkung darstellt[167]. Damit sowohl die *LibClang*-Bibliothek als auch Clang erfolgreich gebaut werden können, mussten die Build-Konfigurationen beider Module angepasst werden[168]. Im Basic-Modul wurde zudem die Deklaration der hinzugekommenen Warnungen und deren Flags vorgenommen:

- `-Wunsafe-api-usage`: Generierung von Warnungen für alle kritischen APIs
- `-Wfix-unsafe-api-usage`: Generierung von Änderungsvorschlägen (sogenannten *Fix-Its*) für statisch aufzulösende APIs
- `-Wunfixable-unsafe-api-usage`: Generierung von Warnungen für kritische APIs, die nicht automatisch aufgelöst werden

Die **Warnungen** wurden in den vorhandenen Diagnosemechanismus von Clang integriert, sodass die Anzeige einer Warnung keines eigenen Codes bedarf und bestehende Konventionen zur Nutzung der Flags[169] Anwendung finden. Die Anzeige von zwei beispielhaften Warnungen in Xcode ist in Abbildung 5.3 dargestellt.

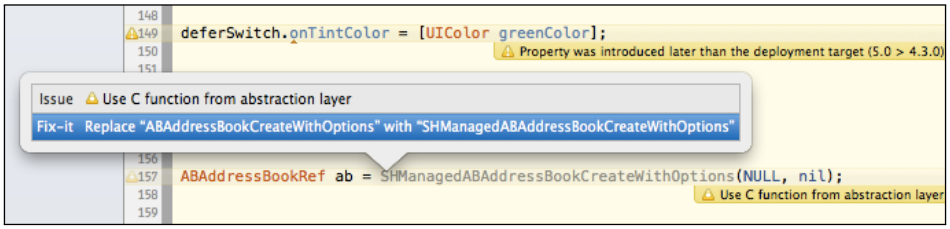


Abbildung 5.3: Warnungen über dynamisch und statisch aufzulösende, kritische APIs

Als letztes Modul wurde *clangSerialization* modifiziert. Um die Verwendung inkompatibler Versionen der Clang-Module auszuschließen, existiert im unmodifizierten Serialization-Modul ein Vergleich der SVN-Revisionsnummern. Durch die beschriebenen Anpassungen und die Auslagerung in ein externes Repository schlägt dieser Test fehl, weshalb er deaktiviert werden musste. Der gesamte Code des Compilers befindet sich in Anhang B.2, zu dem eine Liste der veränderten und neuen Dateien beigefügt wurde.

Zuletzt stellt sich die Frage, wie der modifizierte Compiler in Xcode eingebunden werden kann. Eine Option ist das **Überschreiben** der Clang-Programmdatei[170], die Teil der Xcode-Anwendung

[167] Die Entwicklung von iOS-Apps ist ausschließlich unter Mac OS X möglich
[168] Die Abhängigkeit für die Clang-Programmdatei wurde dem Driver-Modul hinzugefügt; zukünftig ist eine Angabe der Abhängigkeit im Basic-Modul wünschenswert, die transitiv an die abhängigen Module weitergereicht wird
[169] Bsp.: Der Präfix `no` deaktiviert eine Warnung, d. h. `-Wnounsafe-api-usage` unterdrückt sämtliche Warnungen über kritische APIs
[170] Clang-Pfad in der Standardinstallation:
`/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/`

ist. Mit der Binärdatei müssen die Header-Dateien[171] aktualisiert werden. Damit es nicht zu Konflikten zwischen anderen, Clang-basierten Xcode-Funktionen kommt, sollte auch die LibClang-Bibliothek[172] aktualisiert werden. Diese Vorgehensweise wurde nicht umgesetzt, weil der modifizierte Compiler dann sämtliche Xcode-Builds durchführen würde, die bestehende Installation aber nach Anforderung AF.1 möglichst unverändert bleiben soll. Es ist zu erwarten, dass viele Entwickler vor dem Überschreiben des Standardcompilers zögern würden und die Akzeptanz des Werkzeugs darunter leiden könnte.

Weniger radikal ist die Einbindung über ein Xcode-**Plugin**, wie es in [Ros11] vorgeschlagen wird. Ein Versuch, den modifizierten Compiler auf diese Weise einzubinden, scheiterte jedoch. Ferner ist zu beachten, dass die Plugin-API von Xcode nicht dokumentiert ist und sich jederzeit ändern kann. Die Einbindung wurde letztlich durch das Setzen des `CC`**-Flags** auf den Pfad der modifizierten Programmdatei realisiert. Dass das Flag für jede Build-Konfiguration einzeln gesetzt werden kann, wird genutzt um den modifizierten Compiler ausschließlich für Debug-Builds zu verwenden. Die Build-Konfigurationen werden nicht in Xcode selbst, sondern in der Projektdatei gespeichert. Dies bringt den Vorteil mit sich, dass auch andere IDEs den modifizierten Compiler verwenden. Erfolgreich getestet wurde dies mit AppCode.

Weder die Einbindung als Plugin noch das Setzen des `CC`-Flags hat Einfluss auf die (Lib)Clang-basierten Xcode-Funktionen, weshalb die Warnungen über kritische APIs nicht während des Editierens, sondern nur nach einem Build angezeigt werden. Es hat sich das Problem herausgestellt, dass die neu hinzugefügten Diagnose-Flags[173] die Autocomplete-Funktion von Xcode deaktivieren. Als Grund wird vermutet, dass Xcode die Flags an die unmodifizierte LibClang-Bibliothek weiterleitet, die unter diesen Umständen keine Ergebnisse liefert. Um LibClang nicht zu überschreiben, wurden die gewünschten Flags im modifizierten Compiler standardmäßig aktiviert, sodass sie nicht explizit gesetzt werden müssen. Im Repository des Projekts, das sich in Anhang B.2 befindet, wurden zwei zusätzliche Branches für unterschiedliche Flag-Konfigurationen erstellt.

5.2.3 Auflösung: unsafeApiResolver

Das Kommandozeilenwerkzeug *unsafeApiResolver* setzt die in Abschnitt 4.5 beschriebene Auflösung für Methoden, Klassen und Funktionen um. Die Eingabe für das in Objective-C geschriebene Programm ist die vom modifizierten Compiler generierte und im Build-Verzeichnis abgelegte Export-PLIST. Der Pfad des Build-Verzeichnisses und andere Eingabeparameter werden analog zum xcodeprojectcheck-Werkzeug über die Umgebungsvariablen bezogen, sodass die Einbindung in ein Xcode-Projekt über eine Skript-Build-Phase bis auf den Programmnamen identisch ist.

Nach dem Start überprüft unsafeApiResolver, ob alle **Rahmenbedingungen** erfüllt sind. Dazu zählt, dass die Kompatibilitätsbibliothek in Form eines Unterprojekts bereits eingebunden ist und die Export-PLIST gefunden wird. Das Programm verfügt über einen einfachen Cache, sodass die

[171] `../lib/clang/<version>`, ausgehend vom Clang-Pfad
[172] `libclang.dylib` unter `../lib`, ausgehend vom Clang-Pfad
[173] Allgemeine Flags wie `-export-unsafe-api-usage` sind nicht betroffen

Verarbeitung nur fortgesetzt wird, wenn die PLIST seit dem letzten Durchlauf verändert worden ist.

Zur Codegenerierung wird über alle Einträge der Export-PLIST iteriert. Der Typ einer kritischen API entscheidet schrittweise über folgende Aspekte:

- **Typ des Containers:** Jede API wird in einem bestimmten Kontext, im Folgenden Container genannt, deklariert und definiert. Klassen werden in Dateien deklariert, pro Ausweichklasse wird deshalb eine Header- und eine Implementierungsdatei generiert. Die Generierung basiert auf Templates, die in den Ressourcen des Werkzeugs[174] enthalten sind. Die Klassen-Templates enthalten die verfügbarkeitsabhängige Verzweigung zwischen nativer Implementierung und Ausweichimplementierung, lediglich die Namen der Original- und Ausweichklasse müssen an dafür markierten Stellen eingesetzt werden.

  Templates:[175] `ManagedClass.h/m`

  Der Container einer Methode ist eine Klasse. Für kritische Methoden gibt es dabei zwei mögliche Szenarien. Ist die Containerklasse selbst kritisch, wird die Ausweichmethode zur generierten, statisch eingebundenen Ausweichklasse hinzugefügt. Wenn die Methode Teil einer nicht kritischen Klasse ist, wird eine Category zu dieser Klasse generiert[176]. Ein Ausschnitt des Category-Templates ist in Codebeispiel 5.3 abgebildet.

```
1  // Aufruf von der Runtime, wenn 'sel' nicht gefunden wurde
2  + (BOOL)resolveInstanceMethod:(SEL)sel {
3      // Hier müssen noch die kritischen Selektoren eingefügt werden
4      if (NO_SELECTOR_YET) {
5          // Bestimmung des Selektors der Ausweichmethode (Präfix)
6          // Die Ausweichmethode wird in der gleichen Category definiert
7          SEL fallbackSel = [SHLibraryHelper
                   createManagedVersionOfSelector:sel];
8          // Implementierungsdetails bestimmen
9          Method fallbackM = class_getInstanceMethod([self class],
                   fallbackSel);
10         const char *types = method_getTypeEncoding(fallbackM);
11         IMP impl = method_getImplementation(fallbackM);
12         // Ausweichmethode der Originalklasse hinzufügen
13         class_addMethod([self class], sel, impl, types);
14         // Methode wurde hinzugefügt: Aufruf kann erneut erfolgen
15         return YES;
16     }
17     return [super resolveInstanceMethod:sel];
18 }
```

Codebeispiel 5.3: Template für die dynamische Methodenauflösung (Ausschnitt)

[174] Die Ressourcen sind nicht direkt im Kommandozeilenwerkzeug enthalten, sondern werden vom Installationsprogramm bereitgestellt

[175] Pfad innerhalb des Projektverzeichnisses: `Resources/Templates/Files/`

[176] Pro Frameworkklasse wird nur eine Category generiert, die die Ausweichmethoden aller kritischen Methoden dieser Klasse definiert

Die Ausführung des dargestellten Codes bewirkt, dass die Ausweichmethoden verwendet werden, wenn die native Schnittstelle fehlt. Durch die Umsetzung als Category wird die möglicherweise vorhandene Implementierung von `resolveInstanceMethod:` überschrieben. Damit auch die Implementierungen der Containerklasse oder anderer darauf definierter Categories genutzt und priorisiert werden können, muss die Originalimplementierung per Method Swizzling gesichert und in `resolveInstanceMethod:` aufgerufen werden (in Codebeispiel 5.3 nicht dargestellt). Das Vertauschen der Selektoren muss in der `load`-Klassenmethode der Category implementiert werden[177]. Weil das Überschreiben von `resolveInstanceMethod:` nur in wenigen Ausnahmefällen zu Problemen führt, wurde in der prototypischen Umsetzung auf Method Swizzling verzichtet.

Templates: `LoadManipulator.h/m`

Alle Ausweichfunktionen werden gesammelt in einer Header-Datei deklariert und in der zugehörigen Implementierungsdatei definiert. Durch die statische Einbindung sind keine weiteren Vorkehrungen nötig.

Templates: `SHManagedCFunctions.h/m`

- **Deklaration:** Die folgenden zwei Schritte sind nur für Methoden und Funktionen relevant. Die Deklaration und Implementierung einer Klasse wird, abgesehen von ihren Methoden, bereits vom ersten Schritt vollständig abgedeckt.

  Die Deklaration einer Ausweichfunktion setzt sich auf einfache Weise aus der Originaldeklaration und einem Präfix zusammen. Die Deklaration wird der Containerdatei für Ausweichfunktionen hinzugefügt.

  Die Deklaration einer Methode hängt wiederum von der Art der Containerklasse ab. Im Fall einer kritischen Containerklasse wird die Deklaration der Originalmethode der Ausweichklasse hinzugefügt. Ist die Containerklasse nicht kritisch, wird die Originaldeklaration um einen Präfix erweitert und dem Container, einer Category, hinzugefügt. Der Präfix vermeidet, dass die möglicherweise nativ vorhandene Originalimplementierung durch die Category überschrieben wird. Zur Einbindung der Ausweichmethode muss der Originalselektor dem Auflösungsmechanismus bekannt gemacht werden. Sei `criticalAPI` ein kritischer Beispielselektor. In Zeile 4 des Codebeispiels 5.3 muss nun der initial als `NO` definierte Platzhalter `NO_SELECTOR_YET` mit dem Ausdruck `sel == @selector(criticalAPI)` überschrieben werden.

- **Implementierung**: Die Implementierung der Ausweichfunktionen und -methoden folgt direkt dem unter 4.6.2 aufgestellten Konzept, es muss jedoch erneut unterschieden werden, ob die Containerklasse einer Methode kritisch ist. Wenn dies der Fall ist, muss innerhalb der Ausweichimplementierung eine Verfügbarkeitsüberprüfung stattfinden, weil die Methode samt ihrer Containerklasse statisch eingebunden wird. Die konkreten Snippets der Ausweichimplementierungen, die sich u. a. durch die Art des Rückgabewertes unterscheiden, können im `CodeSnippets`-Verzeichnis unter `Resources/Templates` gefunden werden. Im Unterverzeichnis `ExtendedCodeSnippets` befindet sich ein als Git-Submodule eingebundenes Code-Repository zur Bereitstellung allgemeingültiger Ausweichimplementierungen. Das Code-Repository befindet sich auch in Anlage B.4.

[177] Zur Begründung, s. Abschnitt 3.3.1

	Container	Einbindung	Präfix
Methode einer kritischen Klasse	Ausweichklasse der kritischen Klasse	statisch	nein
Methode einer unkritischen Klasse	Category zu der unkritischen Klasse	dynamisch	ja
Objective-C-Klasse	`.h`/`.m`-Dateien (pro Klasse)	statisch	ja
C-Funktion	Datei	statisch	ja

Tabelle 5.1: Auflösung verschiedener APIs mit unsafeApiResolver

Die beschriebenen Vorgehensweisen werden in Tabelle 5.1 zusammengefasst. Vor jedem Schritt der Codegenerierung wird geprüft, ob der zu generierende Code bereits existiert. Einmal generierte Dateien, Deklarationen und Definitionen werden unter keinen Umständen überschrieben oder entfernt. Die in der Export-PLIST enthaltenen Kontextinformationen werden zur Header-Dokumentation genutzt. Neu generierte Ausweichklassen und die Containerdatei für Funktionen werden automatisch einem Sammelheader der Bibliothek hinzugefügt, sodass diese analog zu den Plattform-Frameworks durch das Hinzufügen eines einzigen Imports genutzt werden kann. Alle dynamischen Frameworks der App werden auch von der Kompatibilitätsbibliothek aus verlinkt, damit die Typinformationen der Deklarationen und Ausweichimplementierungen verfügbar sind. Um die generierten Dateien zum Xcode-Projekt hinzuzufügen, wird auf ein in Python implementiertes Werkzeug namens *mod-pbxproj*[178] zurückgegriffen. Falls nötig, wird der Build-Prozess über AppleScript neu gestartet. Weil die AppleScript-API von Xcode 4 nicht implementiert ist, wird dazu GUI-Scripting[179] verwendet. Der Quellcode von unsafeApiResolver befindet sich in Anhang B.3.

5.3 SONDERFÄLLE

Das Paradigma des Konzepts ist es, sämtliche Versionsabfragen in der Kompatibilitätsbibliothek durchzuführen. Findet der Compiler eine kritische API im Anwendungsquellcode, wird deshalb nicht näher untersucht, unter welchen Bedingungen die API erreicht werden kann. Eine solche Betrachtung des Kontexts kann im Compiler aus Performancegründen nicht integriert werden, wäre aber eine Option im Static Analyzer. Allerdings wurde für Klassen eine einfache Erkennung von **False Positives** realisiert, indem `class`-Nachrichten an kritische Klassen bewusst ignoriert werden. Unter der Annahme, dass der Weak-Linking-Mechanismus verwendet wird, kann so die Verfügbarkeit einer Klasse geprüft werden, ohne dass das Klassensymbol als kritisch markiert wird. Werden andere Symbole fälschlicherweise als kritisch markiert, obwohl deren konditionale Verwendung im Anwendungscode sichergestellt wurde, kann das in Codebeispiel 5.4 gezeigte Makro zur Unterdrückung von Warnungen verwendet werden. Derartige Makros sind für alle Clang-basierten Warnungen möglich, der Protoyp definiert Makros für die drei neu hinzugefügten Warnungen. Über das `-Wnounsafe-api-usage`-Flag können die Warnungen in einer ganzen Datei unterdrückt werden.

[178] `https://github.com/kronenthaler/mod-pbxproj/`

[179] `https://developer.apple.com/library/mac/#documentation/AppleScript/Conceptual/AppleScriptX/Concepts/as_related_apps.html`

```
1 // Die Verwendung der 'class'-Methode erzeugt keine Warnungen
2 if ([PKPass class]) {
3     // Der Kontext wird nicht analysiert
4     // Warnungen müssen explizit unterdrückt werden
5     __SUPPRESS_CRITICAL_API_BEGIN__
6     PKPass *myPass = [PKPass new];
7     __SUPPRESS_CRITICAL_API_END__
8 }
```

Codebeispiel 5.4: Makros zur Unterdrückung von Clang-Warnungen

Nicht alle Kompatibilitätsprobleme werden von Xcode ignoriert. Mit der Einführung von ARC wurden auch **Weak-Pointer** (Schlüsselwort `__weak`) unter iOS eingeführt, die keinen Besitz des Zielobjekts[180] ergreifen, aber bei der Deallokation des Objekts automatisch auf `nil` gesetzt werden. Somit werden Abstürze durch Zugriffe auf undefinierte Speicherbereiche vermieden. Weak Pointer sind von der Laufzeitplattform abhängig und erst ab iOS 5 verfügbar. Bei der Verwendung in Projekten mit einem niedrigeren Deployment Target weist Xcode auf diese Problematik hin.

Auch die Verwendung kritischer APIs, einschließlich Auto Layout[181], in XIB- und Storyboard-Dateien kann von den unmodifizierten Entwicklerwerkzeugen erkannt werden. Die **Layout-Dateien** haben jedoch ein eigenes Deployment Target, das nicht mit dem der Anwendung synchronisiert und initial auf die Version des Base SDKs gesetzt wird. Nach einer manuellen Korrektur weist der XIB-Compiler *ibtool*[182] auf Kompatibilitätsprobleme hin. In einer geplanten Weiterentwicklung des xcodeprojectcheck-Werkzeugs muss nicht die Detektion an sich umgesetzt, sondern lediglich das Deployment Target korrigiert werden.

Ein letzter hier untersuchter Sonderfall ist das mit iOS 6 bzw. Clang 3.1 vorgestellte Array- und Dictionary-**Subscripting**, das den lesenden und schreibenden Zugriff auf Objekte u. a. vom Typ `NSArray` und `NSDictionary` syntaktisch vereinfachen soll. Die Zeilen 4 und 6 des Codebeispiels 5.5 zeigen den Vergleich zwischen der Subscripting-Syntax und dem herkömmlichen Zugriff über die API einer Klasse.

```
1 NSArray *array = @[@"One", @"Two", @"Three"];
2 id obj;
3 // Zugriff per Subscripting
4 obj = array[0];
5 // Zugriff über die NSArray-API
6 obj = [array objectAtIndex:0];
7 // Zugriff über die Subscripting-API
8 obj = [array objectAtIndexedSubscript:0];
```

Codebeispiel 5.5: Array-Subscripting in Objective-C

Der Compiler übersetzt die Subscripting-Syntax in das Versenden einer Nachricht [cla13], nutzt dafür aber nicht den `NSArray`-spezifischen Selektor `objectAtIndex:`. Stattdessen wurden vier neue

[180] Im Sinne des Memory Managements
[181] `http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/AutolayoutPG/`
[182] `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/ibtool.1.html`

Methoden für den lesenden und schreibenden Zugriff auf index- und schlüsselbasierte Klassen eingeführt, von denen eine in Zeile 8 des Codebeispiels 5.5 verwendet wird (lesend, indexbasiert). Durch die Abstraktion von dem konkreten Interface von `NSArray` bzw. `NSDictionary` steht Subscripting auch anderen Klassen offen, die mindestens eine der neuen Methoden adaptieren. Allerdings müssen auch die bestehenden Klassen erst um diese Methoden erweitert werden, weshalb Subscripting nicht abwärtskompatibel ist. Während die Header-Informationen die Verfügbarkeit ab iOS 6 garantieren [nsa12, nsd12], ist Array- und Dictionary-Subscripting laut Dokumentation bereits ab iOS 5 nutzbar [obj12]; Versuche im Rahmen dieser Arbeit haben sogar eine Verfügbarkeit unter iOS 4 ergeben. Allerdings ist die Verfügbarkeit an weitere Rahmenbedingungen gebunden[183], weshalb der modifizierte Compiler derzeit die konservativen Informationen der Header-Dateien verwendet. Die Verfügbarkeit der Subscripting-Methoden auf Legacy-Systemen stellt Apple durch die dynamische Manipulation der Framework-Klassen sicher [Lut12], ähnlich zu den in dieser Arbeit gewählten Techniken.

5.4 INSTALLATION UND NUTZUNG

In Anhang B.5 befindet sich ein **Installationsprogramm**, das die entwickelten Werkzeuge zur Detektion und Auflösung in ein bestehendes iOS-Projekt einbindet. Das Installationsprogramm kann zwei Modi konfigurieren:

- **Detektionsmodus:** In diesem Modus wird ausschließlich die Detektionsphase in das Projekt eingebunden. Dazu wird der modifizierte Clang-Compiler in ein Unterverzeichnis des Projektverzeichnisses kopiert[184] und anschließend in den Build-Einstellungen referenziert. Für den Compiler-Build wurde der `customised_clang_detection`-Branch verwendet, der die `-Wunsafe-api-usage`-Warnungen automatisch aktiviert, ohne dass das Flag explizit gesetzt werden muss. Die Maxime des Detektionsmodus ist es, die Projektstruktur unverändert zu lassen. Daher wird auf die Einbindung des xcodeprojectcheck-Werkzeugs zur Detektion in Ressourcendateien verzichtet.

  Zur Anzeige der Warnungen muss das Projekt lediglich mit Xcode gebaut werden.

- **Detektions- & Auflösungsmodus:** Die Auswahl dieses Modus installiert alle implementierten Werkzeuge, um die Ressourcendetektion, Quellcodedetektion und Auflösung zu unterstützen. Der Compiler wurde aus dem `customised_clang_resolution`-Branch gebaut, sodass die Flags `-Wunfixable-unsafe-api-usage` und `-Wfix-unsafe-api-usage` standardmäßig gesetzt sind, nicht aber `-Wunsafe-api-usage`. Somit werden nur Warnungen für APIs angezeigt, die nicht von der Auflösungsphase erfasst werden, sowie Änderungsvorschläge zur statischen Auflösung. Neben den Werkzeugen werden auch deren Ressourcen (z. B. Code-Templates) installiert und die Kompatibilitätsbibliothek wird der Anwendung als Unterprojekt hinzugefügt.

  Nach der Installation kann der Build-Vorgang in Xcode ausgelöst werden, um die Detektion und Auflösung anzustoßen. In den Quellcodedateien des Unterprojekts befinden sich

[183] Eine Bedingung ist die Verlinkung einer ARC-Bibliothek [Ste12]

[184] Neben der ausführbaren Programmdatei werden auch die nicht quelloffenen ARC-Bibliotheken aus der lokalen Xcode-Installation und die zum Build passenden Header-Dateien in `../lib` kopiert

Warnungen, die auf leere Ausweichimplementierungen hinweisen, wie in Abbildung 5.4 dargestellt. Zur Entfernung der Warnung muss lediglich die mit `#warning` beginnende Zeile entfernt werden.

Abbildung 5.4: Warnung über eine fehlende Ausweichimplementierung

In der GUI des Installationsprogramms sind alle Modifizierungen ausführlich dokumentiert, sodass die Änderungen bei Problemen oder zur manuellen Konfiguration nachvollzogen werden können. Vor dem ersten Build nach der Installation sollte ein Clean-Vorgang ausgeführt werden, da der Compiler andernfalls auf inkompatible Build-Caches zugreifen könnte. Alle Werkzeuge des Prototyps sind ab Mac OS X 10.7 (Lion) lauffähig.

5.5 AKTUELLE LIMITIERUNGEN

Das Konzept wurde beinahe vollständig umgesetzt und der Prototyp ist für reale Projekte anwendbar. Bei der Implementierung wurde Wert auf eine stabile Ausführbarkeit der Grundfunktionen gelegt, weshalb einige Sonderfälle aus Zeitgründen noch nicht oder nicht vollständig umgesetzt worden sind. Die Limitierungen und einschränkenden Annahmen auf dem aktuellen Stand sollen in diesem Abschnitt kurz genannt werden.

- *Enumerations:* Kritische Enumerations werden derzeit nicht erkannt, weil Clang die Verfügbarkeitsattribute in der Vorbereitungsphase nicht korrekt einliest. Der Bug muss in Clang selbst behoben werden und ist unabhängig von den Modifizierungen dieser Arbeit.
- *Layout-Dateien:* Die Synchronisierung des Deployment Targets der App mit den Deployment Targets der XIB- und Storyboard-Dateien ist nicht fertig implementiert worden, weshalb kritische APIs in Layout-Dateien nicht erkannt werden. Das Deployment Target der Layout-Dateien kann manuell korrigiert werden, um die Detektion zu aktivieren.
- *Method Swizzling:* Um sicherzustellen, dass die generierten Categories in keinem Fall dynamische Auflösungsmechanismen der nativen Klassen überschreiben, soll Method Swizzling angewandt werden. Die genaue Vorgehensweise wurde in Abschnitt 5.2.3 beschrieben, muss aber noch in Code umgesetzt und getestet werden.
- *Code-Repository:* Das Code-Repository unterstützt momentan nur die Bereitstellung einzelner, direkt einzufügender Snippets für Ausweichimplementierungen. Die in Abschnitt 4.6.2 vorgeschlagene Referenzierung bestehender Ausweichklassen ist noch nicht implementiert.
- *Workspaces:* Die Auflösungs- und Installationsprogramme gehen von alleinstehenden Xcode-Projekten aus. Sind die Projekte Teil eines Xcode-Workspaces, schlägt die automatische Konfiguration und Modifizierung der Projektdateien fehl.

Das Projekt ist vollständig quelloffen und soll über die Abgabe dieser Diplomarbeit hinaus weiterentwickelt werden. Die aktualisierten Limitierungen und Annahmen sind im öffentlichen Git-Repository des Projekts aufgeführt, dessen URL sich in Anhang B.5 befindet.

5.6 ZUSAMMENFASSUNG

Die in diesem Kapitel vorgestellten Werkzeuge vereinen eine Vielzahl von **Technologien**, die die Funktionalität des Prototyps und dessen nahtlose Einbindung in Xcode realisieren. Der Großteil des Quellcodes wurde in C++ und Objective-C geschrieben, um die bestmögliche Performance zu erreichen. Zur Modifikation der Projektdateien kommt Python-Code zum Einsatz, zur Steuerung der Xcode-Builds wird AppleScript verwendet und der Aufruf der Werkzeuge erfolgt über Shell-Skripte.

Bei der **Einbindung** der Werkzeuge wird nicht Xcode selbst, sondern eine ausgewählte Projektdatei modifiziert, sodass keinerlei Auswirkungen auf andere Projekte entstehen. Die Einbindung der zusätzlichen Werkzeuge, xcodeprojectcheck und unsafeApiResolver, verlief ohne Probleme und nutzt die von Xcode angebotenen Möglichkeiten, den Build-Prozess über eigene Skript-Phasen beinahe beliebig zu erweitern. Um die Verwendung des modifizierten Compilers in der existierenden Kompilierungsphase zu erwirken, gibt es keinen offiziell empfohlenen Ansatz. In Abwägung der Risiken, speziell hinsichtlich der Aufwärtskompatibilität von Xcode, wurde der Compiler letztlich über ein Xcode-unabhängiges Build-Flag eingebunden.

In Abschnitt 5.3 wurden **Sonderfälle** betrachtet, die vom allgemeinen Konzept und dem daraus entwickelten Prototyp nicht oder nur unzureichend erfasst werden. Geplant oder bereits fertig implementiert wurden die Sonderbehandlungen für ausgewählte False Positives, Layout-Dateien und Subscripting.

Zur **Installation** wurde eine Mac-App erstellt, sodass die Konfiguration und Einbindung der Werkzeuge nicht manuell durchgeführt werden muss. Zur Nutzung des Prototyps sind keine speziellen Kenntnisse notwendig, weil alle Werkzeuge in die bekannten Xcode-Workflows eingebunden sind. Warnungen und Quellcodekommentare weisen den Anwendungsentwickler auf Stellen hin, die einer manuellen Prüfung bedürfen. Um eine breite Nutzerbasis zu erreichen, wurden alle entwickelten Komponenten unter **Open-Source**-Lizenzen veröffentlicht. In den Repositories, die in Anhang B.5 verlinkt sind, befinden sich fertige Builds der Werkzeuge.

6 EVALUATION

Die Evaluation dieser Arbeit ist in zwei Teilen erfolgt. In Abschnitt 6.1 wird demonstriert, dass der Prototyp den aufgestellten Anforderungskriterien genügt, wobei der Analyse objektive **Testreihen** und Fakten zugrunde gelegt worden sind. Um zu zeigen, dass auch die realen und subjektiven Anforderungen von Entwicklern erfüllt werden, wurden **Entwicklertests** durchgeführt. Die Durchführung und Ergebnisse werden in Abschnitt 6.2 beschrieben.

6.1 ERFÜLLUNG DER ANFORDERUNGEN

Die folgenden Unterpunkte beziehen sich auf die in den Abschnitten 4.1 und 5.1 aufgestellten Anforderungen.

AF.1 **Integration:** Abgesehen vom Installationsvorgang muss der Anwendungsentwickler die gewohnte Entwicklungsumgebung zu keinem Zeitpunkt verlassen. Sowohl die Detektion als auch die Auflösung sind in den vorhandenen Build-Vorgang eingebunden, sodass sich an den Workflows keine Änderungen ergeben. Die Ausgabe der Werkzeuge erfolgt in Form der bekannten Xcode-Warnungen, wobei sich die Warnungen des modifizierten Compilers auch in die bestehenden Mechanismen zur Unterdrückung (`#pragma`-Befehle), (De)aktivierung (Flags) und Fehlerbehebung (Fix-Its) integrieren. Live-Warnungen werden derzeit nicht unterstützt, weil dazu eine bestehende Bibliothek überschrieben werden müsste[185]. Für die Kompatibilitätsbibliothek werden native Xcode-Subprojekte verwendet.

Ein entscheidender Unterpunkt des Integrationskriteriums ist die **Unverändertheit** der IDE-Installation, um Xcodes Aktualisierbarkeit nicht zu beeinträchtigen. Auf Plugins und andere direkte Xcode-Modifikationen konnte vollständig verzichtet werden und die positiven Auswirkungen wurden bereits praktisch erprobt. Die Entwicklung des Prototyps ist ausschließlich auf Basis der aktuellen, öffentlich verfügbaren Xcode-Version 4.6 unter Mac OS X 10.8

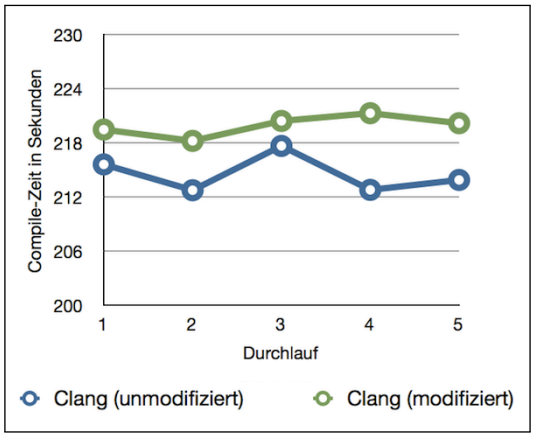
[185] Wird die Veränderung in Kauf genommen und ein anderer Installationsprozess gewählt, stehen auch Live-Warnungen zur Verfügung

erfolgt, zum Testen der Werkzeuge wurde iOS 6.1 zugrunde gelegt. Alle Komponenten des Prototyps funktionieren auch mit der privaten Beta-Version von Xcode 5 unter Mac OS X 10.8 sowie Mac OS X 10.9 (Beta 1) und im Zusammenspiel mit iOS 7 (Beta 1). Auf Details kann aus Gründen der Geheimhaltung[186] nicht eingegangen werden, alle getesteten Beta-Versionen unterscheiden sich jedoch substanziell von den öffentlich verfügbaren Versionen.

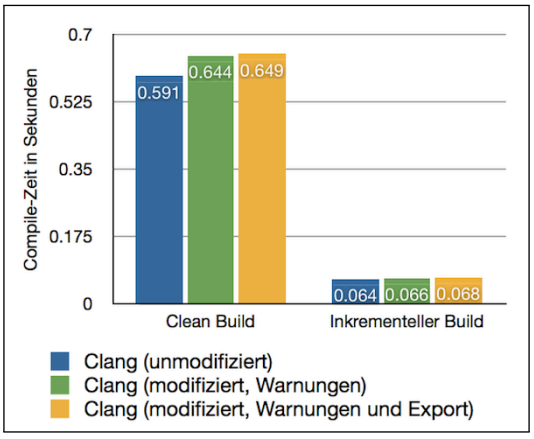
AF.2 **Priorisierung der nativen Implementierung:** Die Priorisierung der nativen Implementierung ist bei der dynamischen Auflösung von Objective-C-Methoden fest im Konzept verankert, da die Ausweichimplementierung erst eingebunden wird, wenn die Bindung an die native Implementierung fehlgeschlagen ist.

Im Fall von kritischen Klassen und C-Funktionen ist die Priorisierung der nativen Implementierung Teil der automatisch eingefügten Code-Templates. Der Entwickler wird durch generierte Quellcodekommentare auf den Zweck und die Auswirkungen hingewiesen, Warnungen markieren die Stellen der einzufügenden Ausweichimplementierung.

AF.3 **Performance:** Zur Evaluierung der Performance wurden verschiedene Tests durchgeführt. Im ersten Test wurde der reine **Compiler-Overhead** an einem großen Projekt, das keinerlei kritische APIs enthält, gemessen. In diesem Fall bringt der Overhead dem Anwendungsentwickler keinen Mehrwert und sollte deshalb besonders gering sein. Als zu kompilierende Codebasis wurde das in C++ geschriebene LLVM-Projekt verwendet. Der Vergleich fand zwischen dem unmodifizierten Compiler aus dem Repository des Clang-Projekts und einer modifizierten Version, die auf dem gleichen Commit des Repositories aufsetzt, statt. Das LLVM-Projekt wurde mit beiden Compilern sechs mal kompiliert[187], der langsamste Durchlauf wurde gestrichen. Aus Abbildung 6.1a geht hervor, dass der modifizierte Compiler um wenige Sekunden langsamer ist. Angesichts der Gesamtdauer der Kompilierung von über drei Minuten ist der zeitliche Overhead vertretbar, prozentual liegt der Overhead bei durchschnittlich 2,5 %. Die genauen Rahmenbedingungen und Ergebnisse können in Anlage A.2 eingesehen werden.



(a) LLVM-Quellcode ohne kritische APIs



(b) iOS-Quellcode mit kritischen APIs

Abbildung 6.1: Vergleich der Compile-Zeiten zwischen dem unmodifizierten und dem modifizierten Clang-Compiler

[186] Die Beta-Versionen werden exklusiv für Mitglieder des kostenpflichtigen iOS Developer Programs bereitgestellt, alle Informationen zu Betas sind vertraulich

[187] Zwischendurch wurde jeweils ein Clean-Vorgang durchgeführt

Beim Build-Vorgang einer iOS-App mit kritischen APIs lässt sich ein prozentual größerer zeitlicher Unterschied feststellen, schließlich müssen die Warnungen und Diagnoseinformationen erfasst und schlussendlich exportiert werden. Verglichen wurden die selben Compiler, die auch für die LLVM-Builds (Abbildung 6.1a) verwendet worden sind. Als zu kompilierende Codebasis wurde die in Abschnitt 6.2 für die Entwicklertests bereitgestellte iOS-Beispielanwendung herangezogen, die knapp 30 kritische APIs verschiedener Typen enthält. Der modifizierte Compiler wurde in zwei Konfigurationen getestet. In beiden Konfigurationen wurde das `-Wunsafe-api-usage`-Flag zur Anzeige der Warnungen gesetzt, in der letzten Konfiguration zusätzlich das `-export-unsafe-api-usage`-Flag zum Export in eine Datei. Die Messergebnisse in Abbildung 6.1b zeigen einen erkennbaren Unterschied der Compile-Zeiten zwischen dem unmodifizierten und dem modifizierten Compiler für Clean Builds. Bei inkrementellen Builds ist der Unterschied dagegen kaum messbar, Gleiches gilt für den Unterschied zwischen den beiden Flag-Konfigurationen des modifizierten Compilers. Prozentual liegt der Overhead bei Clean-Builds bei 9.8 %[188], bei inkrementellen Builds bei 6,2 %. Aufgrund der sehr geringen Build-Zeiten können jedoch konstante Overheads die relativen Unterschiede verfälschen. Zudem besitzt die getestete Anwendung eine ungewöhnlich hohe Dichte kritischer APIs. Die in Abbildung 6.1b dargestellten Messwerte wurden aus dem Mittel von sechs Messreihen und der Streichung der Schlechtesten ermittelt, alle Details der Messung können in Anhang A.3 nachgeschlagen werden. Aufgrund der fehlenden Vergleichbarkeit zwischen den Standard-Build-Phasen eines Xcode-Projekts und dem modifizierten Build-Vorgang inklusive Codegenerierung wurde auf Testreihen mit eingebundener Auflösungsphase verzichtet.

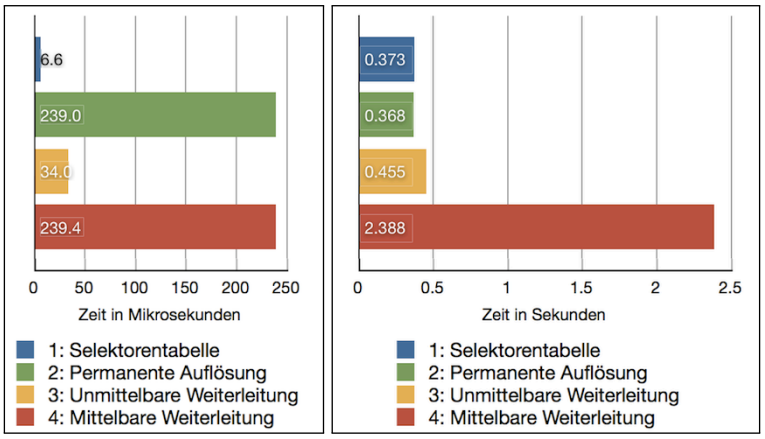
Die beiden Messreihen haben gezeigt, dass die Compiler-Performance davon abhängt, wieviele kritische APIs gefunden werden. Der Overhead korreliert daher mit dem Mehrwert für den Entwickler. Wichtiger als die Compile-Zeit-Performance ist die Performance der Anwendung zur **Laufzeit**, da Unterschiede in diesem Bereich direkt vom Anwender bemerkt werden, ohne dass die Vorteile der verwendeten Mechanismen für diesen offensichtlich sind. Die klare Priorität der Laufzeit-Performance ist auch in der initialen Entwicklerbefragung deutlich geworden (s. Abschnitt 4.1).

Relevant für die Laufzeitperformance ist nur der generierte Code der Kompatibilitätsbibliothek. Bei der statischen Auflösung lässt sich der Overhead leicht aus dem Code ablesen und besteht im Wesentlichen aus einer `if`-Verzweigung und einem zusätzlichen Aufruf im Fall einer verfügbaren, nativen API. Die Verzweigung ist im Rahmen der Möglichkeiten der verwendeten Programmiersprachen[189] alternativlos und wurde den Best Practices aus Abschnitt 3.4.3 entnommen, weshalb die Performance nicht gesondert untersucht wurde. Von einem signifikanten Overhead kann nicht ausgegangen werden.

Interessanter ist die Betrachtung des **dynamischen Auflösungsmechanismus**. Anhand der in Anlage A.4 beigelegten iOS-Anwendung wurde der Overhead der Nachrichtenauflösung in verschiedenen Stufen[190] gemessen, wobei die erste Stufe der normalen Nachrichtenauflösung entspricht, die als Referenzwert dient. In dieser Arbeit wurde die Auflösung kritischer APIs in Stufe 2 (Permanente Auflösung) vorgeschlagen und umgesetzt. Wie aus Abbildung 6.2a abgelesen werden kann, ist die erstmalige Auflösung eines Selektors in

[188] Gemessen zwischen dem unmodifizierten Compiler und der langsameren Konfiguration
[189] C (Funktionen) und Objective-C (Klassen)
[190] Die fünf Stufen der Nachrichtenauflösung wurden in Abschnitt 3.3.1 erstmalig diskutiert

Stufe 2 vergleichsweise teuer, weil ein neuer Eintrag in der Selektorentabelle hinzugefügt werden muss. Der absolute Overhead ist mit einer viertel Millisekunde dennoch vernachlässigbar.



(a) Erster Aufruf (b) Gesamtzeit für 100.000 Aufrufe

Abbildung 6.2: Zeitlicher Overhead der dynamischen Nachrichtenauflösung

Die Stärke des gewählten Ansatzes zeigt sich bei weiteren Aufrufen des gleichen Selektors, nachvollziehbar anhand von Abbildung 6.2b. Der einmalig hinzugefügte Selektor ist von nativ vorhandenen Selektoren nicht zu unterscheiden und die Auflösung erfolgt faktisch in Stufe 1. Entsprechend gleichen sich die Gesamtzeiten der Auflösungsmechanismen in Stufe 1 und 2 bei hinreichend vielen Aufrufen[191]. Die Auflösung in Stufe 3 hat einen deutlich geringeren initialen Overhead (ca. 30 Mikrosekunden), allerdings sind auch Folgeaufrufe etwas teurer als im Falle der normalen Auflösung in Stufe 1. Die mittelbare Weiterleitung (Stufe 4) eignet sich nur nach einer Abwägung in Sonderfällen, da sich der vergleichsweise hohe initiale Overhead auch auf fortlaufende Aufrufe überträgt.

Die Tests wurden auf einem iPhone 4S unter iOS 6.1 ausgeführt. Jeder in Abbildung 6.2 eingetragene Wert wurde aus dem Durchschnitt von fünf Messwerten gebildet, wobei zuvor aus sechs Messwerten der Schlechteste gestrichen wurde. Die Methodenimplementierung des aufzulösenden Selektors wurde leer gelassen, allerdings können die eingefügten Aufrufe zur Messung die absoluten Werte beeinträchtigen. In Anhang A.5 befinden sich die detaillierten Ergebnisse und Rahmenbedingungen des Tests.

AF.4 **App-Store-Kompatibilität:** Es wird kein mit den App-Store-Richtlinien in Konflikt stehender Code generiert. Die Legalität des Method Swizzlings und des Überschreibens von Framework-Methoden mit Categories wurde in Abschnitt 3.3.5 erörtert. Für Release-Builds wird der unmodifizierte Apple-Compiler verwendet, um Probleme auf binärer Ebene und beim Code-Signing der Anwendung grundsätzlich auszuschließen.

AF.5 **Zentrale Auflösung:** Das Kriterium der zentralen Auflösung wird durch die Auslagerung des generierten Codes in eine Bibliothek erfüllt. Zur Einbindung dieser Bibliothek wird der bestehende Code minimal und auf nachvollziehbare Weise[192] verändert, wenn die Möglichkeiten der Programmiersprache keine dynamische Auflösung erlauben.

[191] In den durchgeführten Testreihen war die Auflösung in Stufe 2 sogar schneller; die Ursache dafür können Caches oder Messfehler sein

[192] Die Modifikation ist an einem einheitlichen Präfix erkennbar und muss manuell bestätigt werden

AF.6 **Umfangreiche Diagnostikinformationen:** Die Diagnostikinformationen sollen helfen, die automatischen Eingriffe des Prototyps nachzuvollziehen. In Codebeispiel 6.1 ist die generierte Dokumentation einer kritischen Methode abgebildet.

```
1  /**
2   *  This Property was generated by Unsafe API Resolver
3   *  References:
4   *  [SecondViewController.m:79]
5   *
6   *  Declared in:
7   *  Class 'UISwitch'
8   *
9   *  Available since:
10  *  iOS 6.0
11  *
12  */
13 - (void) SHManagedSetTintColor:(UIColor *)tintColor;
```

Codebeispiel 6.1: Generierte Headerdokumentation einer kritischen Methode

Aus der Dokumentation geht hervor, welche Zeilen des Anwendungsquellcodes für den Eingriff auslösend waren (Codebeispiel 6.1, Zeile 4), welche Frameworkklasse die kritische Methode deklariert (Zeile 7) und für welche Versionen die vorliegende Ausweichimplementierung relevant ist (Zeile 10). Umgekehrt können die kritischen Aufrufe im Anwendungscode durch Warnungen hervorgehoben werden. Die Warnungen enthalten einen kurzen Erklärungstext sowie die minimale SDK-Version einer API. Perspektivisch kann erwogen werden, die Auflösung kritischer APIs im Anwendungsquellcode durch Kommentare oder Annotationen deutlicher und unabhängig vom Compiler hervorzuheben.

AF.7 **Ausführung bei jedem Build:** Der geringe Performance-Overhead des Compilers erlaubt die Durchführung der Detektion bei jedem Build-Vorgang. Umgesetzt wurde die Anforderung durch die Verwendung des modifizierten Compilers in der vorhandenen Kompilierungsphase von Xcode. Der modifizierte Compiler wird bewusst nur für Debug-Builds verwendet.

AF.8 **Ausführung mit Protokollierung:** Der Build-Vorgang und die eingebundenen Werkzeuge laufen stets unterbrechungsfrei durch. Im seltenen Fall mehrerer möglicher Implementierungen einer Ausweich-API wird die unwahrscheinlichere Variante als Kommentar hinzugefügt. Eine entfernbare Warnung lenkt die Aufmerksamkeit des Entwickler auf Stellen, die einer besonderen Prüfung bedürfen. Ein solcher Fall ist exemplarisch in Codebeispiel 6.2 dargestellt.

```
1  - (NSUInteger) hash {
2      if (self.nativeImpl) {
3          #warning decide whether you want to use the native
                   implementation, if available
4          return [self.nativeImpl hash];
5          //return [super hash];
6      }
```

```
7
8       // Call super or provide custom fallback implementation
9       return [super hash];
10  }
```

Codebeispiel 6.2: Protokollierung der Auflösungsvarianten

Die abgebildete `hash`-Methode wurde einer kritischen Klasse hinzugefügt, die wiederum statisch aufgelöst wird. Die Generierung wurde ausgelöst, weil im Anwendungscode eine `hash`-Nachricht an die kritische Klasse entdeckt wurde. Dabei wird die Methode nicht von der kritischen Klasse selbst, sondern von der Basisklasse `NSObject` deklariert. Ob die Implementierung von der kritischen Klasse verändert wird, kann nicht automatisch ermittelt werden. Der Entwickler muss daher an zwei Stellen (Codebeispiel 6.2, Zeile 3-5 und Zeile 8-9) die Entscheidung treffen, ob die Implementierung der nicht kritischen Basisklasse ausreichend ist (`super`-Aufruf), oder ob die native Implementierung der kritischen Klasse bzw. eine Ausweichimplementierung aufgerufen werden muss.

Die Anforderungen AF.1 - AF.8 wurden von Beginn an als Pflichtkriterien betrachtet und bei allen Design- und Implementierungsentscheidungen berücksichtigt. Deshalb werden aus objektiver Sicht alle Anforderungskriterien vom Prototyp erfüllt.

6.2 ENTWICKLERTESTS

6.2.1 Durchführung

Im Rahmen der Evaluation dieser Arbeit wurden zehn iOS-Entwickler gebeten, den Prototyp anhand einer mit kritischen APIs ausgestatteten **Beispielanwendung** zu testen. Den Entwicklern wurde eine ausführliche Anleitung mit Hintergrundinformationen zur Wirkungsweise des Prototyps zur Verfügung gestellt, sodass nicht nur die Nutzung, sondern auch das zugrundeliegende Konzept bewertet werden konnte. Die Entwicklergruppe bestand hauptsächlich aus Studenten, die unterschiedlich viel Erfahrung mit der Entwicklung von iOS-Anwendungen besitzen; viele der Entwickler waren bereits an App-Store-Anwendungen beteiligt.

Die in Anlage A.7 enthaltene Beispielanwendung basiert auf iOS 6 und enthält insgesamt 20 verschiedene kritische APIs, von denen die meisten zum Absturz unter iOS 5 und iOS 4 führen. Neben der Beantwortung eines **Fragenkatalogs** war die Hauptaufgabe, die gegebene Anwendung so zu modifizieren, dass Abstürze verhindert werden. In wenigen Fällen sollten auch triviale Ausweichimplementierungen eingebunden werden.

Der Fragenkatalog befindet sich in Anhang A.6. Die Bearbeitung der Aufgabe und das Beantworten der Fragen wurde von den Entwicklern selbstständig und ohne Hilfestellung durchgeführt. Zum Großteil der Fragen wurden Antwortmöglichkeiten vorgegeben, die einer fünfstufigen Likert-Skala entsprechen [Lik32] und sich an der bekannten Notenskala von 1 bis 5 orientieren.

Zu jeder Frage oder These konnten die Evaluationsteilnehmer somit zwischen vollkommener Zustimmung, entschiedener Ablehnung, abgeschwächten Positionen und einer neutralen Antwort wählen. Insbesondere, aber nicht ausschließlich bei ablehnenden Haltungen sollten die Entwickler ihre Entscheidung begründen, um Rückschlüsse auf Verbesserungsmöglichkeiten ziehen zu können. Die fünfstufige Skala war nicht für alle Fragen anwendbar, die alternativen Antworten gehen in diesen Fällen aus dem Fließtext oder den Grafiken der Auswertung hervor. Es wurden Fragen zu den Themenbereichen der Konzeption und der Usability gestellt, die Auswertung der Funktionalität wurde auf Basis der praktischen Aufgabe durchgeführt. Die Ergebnisse werden nachfolgend in den entsprechenden Unterabschnitten ausgewertet und grafisch dargestellt, die anonymisierten Daten der Entwicklertests und der initialen Entwicklerumfrage befinden sich in Anhang A.8. Aufgrund der begrenzten Teilnehmerzahl sollen die Diagramme keine statistische Belastbarkeit suggerieren, sondern lediglich die Ergebnisse der Evaluation visualisieren.

6.2.2 Ergebnisse: Konzeption

Mit der grundlegenden Herangehensweise stimmten fast alle befragten Entwickler überein. Der These, dass das Konzept ausgereift und durchdacht sei, stimmten alle Entwickler zu, 40 % gaben ihre „vollkommene Zustimmung“ an[193]. Noch deutlicher (70 %) war die vollkommene Zustimmung für die Auslagerung von Verfügbarkeitsprüfungen und Ausweichcode in ein Unterprojekt. Dem entgegen steht ein Entwickler, der die direkte Auflösung im Anwendungscode bevorzugt.

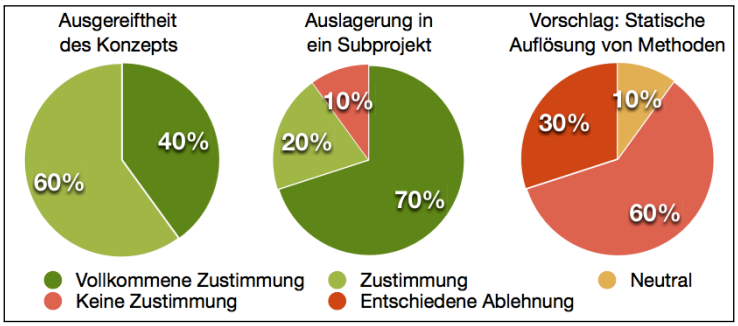


Abbildung 6.3: Evaluationsergebnisse zum Konzept der Arbeit

Um den Zuspruch zur dynamischen Methodenauflösung zu evaluieren, wurden die Entwickler gefragt, ob sie stattdessen die statische Auflösung analog zu Funktionen und Klassen bevorzugen würden. Dabei brachten 90 % ihre (teils entschiedene) Ablehnung zum Ausdruck, Zustimmung gab es keine. Die Verteilung der Antworten kann aus Abbildung 6.3 abgelesen werden.

6.2.3 Ergebnisse: Usability

Einen besonders hohen Stellenwert haben die Fragen zur Usability eingenommen, weil dieser Aspekt schlecht objektiv oder faktisch erfasst und bewertet werden kann. Die Umfrageergebnisse dieses Bereichs wurden in Abbildung 6.4 veranschaulicht. Aus den abgebildeten Diagrammen zu

[193] Dabei ist zu beachten, dass das Konzept im Rahmen der Evaluation nicht in dem Detailgrad der vorliegenden Arbeit vorgestellt werden konnte

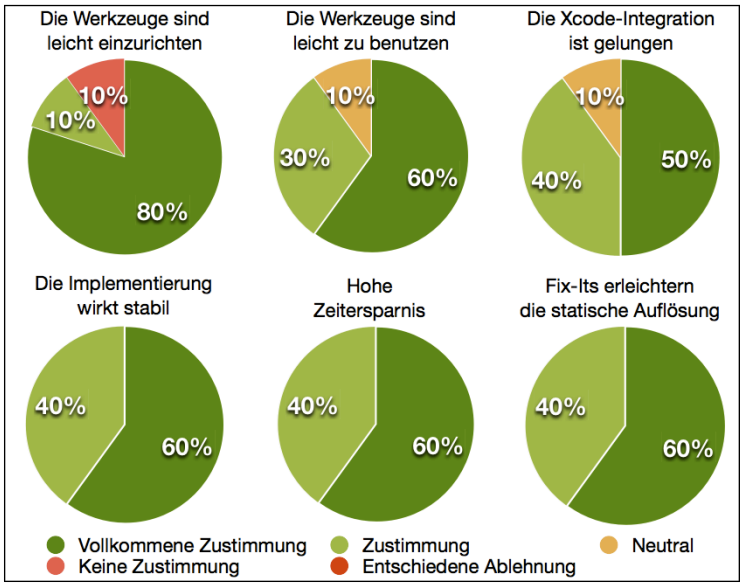


Abbildung 6.4: Evaluationsergebnisse zur Usability des Prototyps

verschiedenen Thesen kann ein breiter Zuspruch zu den gewählten Ansätzen zur Einrichtung und Integration des Prototyps geschlussfolgert werden. Alle befragten Entwickler bescheinigen der Nutzung des Prototyps eine Zeitersparnis im Vergleich zur manuellen Auflösung und bestätigen den Mehrwert der compilergestützten Änderungsvorschläge (Fix-Its). Vereinzelt wurden Verbesserungswünsche zur Integration und Bedienbarkeit der Werkzeuge geäußert, die in Abschnitt 6.2.5 berücksichtigt werden.

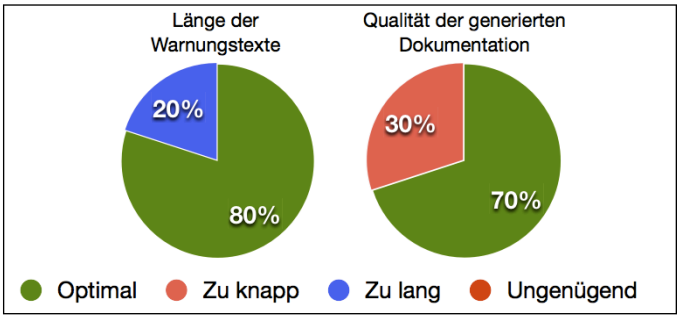


Abbildung 6.5: Evaluationsergebnisse zur Qualität der Warnungen und Dokumentation

Die Qualität der generierten Warnungen (Detektionsmodus) und der generierten Dokumentation wurde konkret hinterfragt, die Ergebnisse wurden in Abbildung 6.5 dargestellt. Dabei zeigt sich, dass einige Entwickler kürzere Warnungen bevorzugen würden, was möglicherweise an der eingeschränkten Anzeige innerhalb Xcodes liegt. Weil die zukünftige Xcode-Version den Warnungen mehr Platz einräumt, sollte dieser Aspekt nach dem öffentlichen Release von Xcode 5 erneut evaluiert werden. Alle Entwickler gaben an, dass die Warnungen nachvollziehbar sind (nicht grafisch dargestellt). Die generierte Dokumentation empfinden dagegen 30 % der befragten Entwickler als zu knapp, weshalb zusätzliche Informationen in Betracht gezogen werden sollten.

6.2.4 Ergebnisse: Funktionalität

Vor der Verwendung des Prototyps wurden die Teilnehmer der Evaluation darum gebeten, die Beispielanwendung innerhalb eines festgelegten Zeitrahmens nach kritischen APIs zu durchsuchen. Insgesamt befanden sich 20 verschiedene kritische APIs in der Anwendung, von denen maximal 19 von einem Entwickler gefunden wurden; durchschnittlich waren es lediglich 9,5. Dies beweist deutlich die fehlenden oder ungeeigneten Hilfsmittel zur Detektion kritischer APIs in den von Apple bereitgestellten Entwicklungswerkzeugen.

Nach der Durchführung der Evaluation wurden die bearbeiteten Projekte aller zehn Entwickler hinsichtlich ihrer tatsächlichen Abwärtskompatibilität untersucht. Dazu wurden die Anwendungen unter iOS 6.1, iOS 5.1 und iOS 4.3 getestet. Wie aus Abbildung 6.6 hervorgeht, konnten acht der zehn Anwendungen absturzfrei auf allen Systemen ausgeführt werden, drei Anwendungen hatten fehlende oder mangelhafte Ausweichimplementierungen[194], und zwei Anwendungen sind auf Legacy-Systemen abgestürzt.

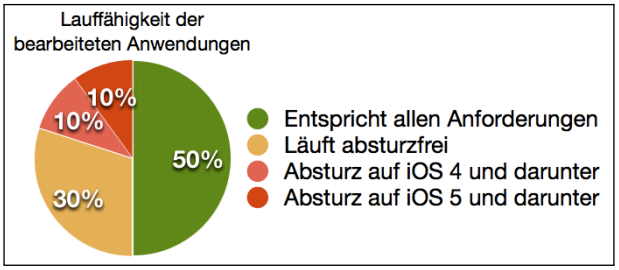


Abbildung 6.6: Auswertung der bearbeiteten Evaluationsprojekte

Die beiden Abstürze resultieren nicht aus dem vom Prototyp generierten Code, sondern fehlerhaften Ausweichimplementierungen der Evaluationsteilnehmer. Dabei haben sich zwei Ausweichimplementierungen als anfällig erwiesen:

- `presentViewController:animated:completion:` Die Methode zur modalen Präsentation eines View-Controllers wurde mit iOS 6 eingeführt. Drei mal wurde die in der Evaluationsanleitung explizit genannte Ausweich-API[195] nicht oder fehlerhaft eingefügt. In dem konkreten Fall hätte sich die Fehlerrate durch das Hinterlegen der Ausweich-API im Code-Repository reduzieren lassen, im Allgemeinen gehört die Verhinderung dieser Art von Programmierfehlern jedoch nicht zu den Zielen dieser Arbeit.

- `init`-Methode einer kritischen Klasse: Drei Entwickler sind an der korrekten Implementierung einer Initialisierungsmethode gescheitert. Auch dabei handelt es sich letztlich um allgemeine Programmierfehler, die unabhängig von der Verwendung des Prototyps sind. Allerdings ist die Generierung von Initialisierungsmethoden ein häufig auftretender Fall, sodass die generierte Ausweichimplementierung an diese Fehlerquelle angepasst werden kann.

[194] Das Ausweichverhalten wurde in der Evaluationsanleitung genau spezifiziert, inklusive Hinweisen zur Implementierung

[195] `presentModalViewController:animated:`

6.2.5 Weiteres Feedback

Neben den Fragen mit vorgegebenen Antwortmöglichkeiten wurde den Entwicklern die Möglichkeit eingeräumt, Bedenken, Verbesserungs- und Erweiterungsvorschläge zu nennen. Mehrfach genannt wurden die folgenden Wünsche und Probleme:

- **Clean Builds vermeiden:** Code-Änderungen in der Kompatibilitätsbibliothek werden nicht immer sofort berücksichtigt, wodurch zusätzliche Clean Builds notwendig werden. Dies wiederum schränkt die Einsatzfähigkeit bei großen Projekten ein. Das Problem ist bekannt und wird nach aktuellem Kenntnisstand von Xcode verursacht. Ein hier aus Gründen der Geheimhaltung nicht näher genannter Bugfix und erste Tests mit Xcode 5 legen nahe, dass das Problem bereits von Apple behoben wurde und somit unabhängig vom vorliegenden Prototyp ist.
- **Versionsinformation zu Beginn der Warnungen:** Zwei Entwickler wünschen sich eine Neuformatierung der Warnungen, sodass die minimal benötigte iOS-Version zu Beginn einer Warnung steht. Dies hängt mit der aktuellen Darstellung von Warnungen in Xcode zusammen, bei der längere Texte abgeschnitten werden. Eine Neuformatierung ist nicht geplant, um das Format der Warnungen konsistent mit den existierenden Warnungen zu halten. Sämtliche Clang-Warnungen beginnen mit einer knappen Problembeschreibung, an die gegebenenfalls Details und konkrete Werte angehangen werden. Die in Abschnitt 6.2.3 angesprochenen Veränderungen in Xcode 5 lassen auf eine verbesserte Anzeige in zukünftigen Versionen hoffen.
- **Uninstaller:** Die Bereitstellung einer Anwendung zum Deinstallieren der Werkzeuge wurde mehrfach gewünscht. Die Umsetzung soll in Zukunft erwogen werden, ist allerdings nicht trivial, da nachträglich nicht aus allen Einstellungen hervorgeht, ob sie bei der Installation vorgenommen worden sind oder zuvor bereits gesetzt waren. Die genaue Dokumentation der durchgeführten Veränderungen im Installationsprogramm oder der Einsatz von Versionsverwaltungssoftware (SCM) erlauben bis auf Weiteres eine manuelle Deinstallation.
- **Workspace-Support:** Die fehlende Kompatibilität mit Xcode-Workspaces wurde bereits in Abschnitt 5.5 angesprochen und ist wichtig für die praktische Anwendbarkeit des Prototyps. Der Hauptgrund für den Einsatz von Workspaces ist das beliebte *CocoaPods*-Werkzeug[196] zur Verwaltung von Abhängigkeiten.

6.2.6 Fazit

Die Entwicklertests haben bestätigt, dass das Grundkonzept für die Auflösung und Detektion kritischer APIs die Entwicklung abwärtskompatibler iOS-Apps erleichtert und die meisten der befragten Entwickler die gewählte Umsetzung und Xcode-Integration befürworten. Alle befragten Entwickler können sich den **Einsatz des Prototyps** im Detektionsmodus auf dem derzeitigen Stand bereits an produktiven Apps vorstellen, 70 % der Befragten würden auch den Auflösungsmodus auf produktive Apps anwenden, wie dies in Abbildung 6.7 illustriert wurde.

[196] `http://cocoapods.org/`

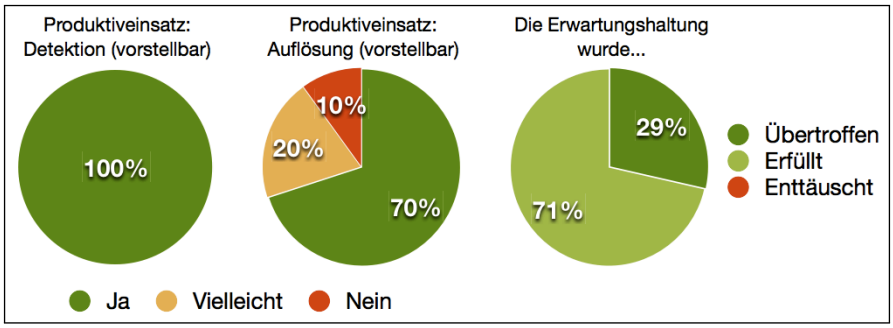


Abbildung 6.7: Evaluationsergebnisse zur Gesamtbewertung

Der Eingriff in die Projektstruktur und die Verwendung von Präfixen anstelle der unveränderten Framework-APIs sind für einen der Entwickler ein Ausschlusskriterium der automatischen Auflösung. Die beiden unentschlossenen Evaluationsteilnehmer haben verschiedene Motive für ihre angegebene Antwort: Ein Entwickler bindet seine Entscheidung an die Kompatibilität mit CocoaPods, ein anderer erwägt, komplett auf kritische APIs zu verzichten bzw. das Deployment Target nach oben zu korrigieren. Von den zehn Teilnehmern der Entwicklertests haben sieben an der initialen Umfrage teilgenommen. Die **Erwartungen** dieser Entwickler an den Prototyp konnten in allen Fällen erfüllt werden; zwei Entwickler gaben an, dass ihre Erwartungen übertroffen worden sind.

Im Rahmen der Entwicklertests wurde der Prototyp auf drei Betriebssystemen (Mac OS X 10.7, 10.8 und 10.9 Beta), zwei verschiedenen Xcode-Versionen (Xcode 4.6 und 5 Beta) und mit zwei iOS SDKs (iOS 6.1 und 7 Beta) getestet, ohne dass von versionsspezifischen Problemen berichtet wurde. Die **Stabilität** der Implementierung muss sich jedoch noch durch den Einsatz an komplexeren iOS-Projekten beweisen.

Die von den Entwicklern bearbeiteten iOS-Projekte haben gezeigt, dass die **Codegenerierung** optimiert werden kann, um Fehlerquellen zu minimieren. Zur Behebung des in Abschnitt 6.2.4 besprochenen Problems der fehleranfälligen Initialisierungsmethoden wurde der Prototyp bereits weiterentwickelt. Der zur Evaluation verwendete Stand wurde in den Repositories (Anhang B.2 und Anhang B.5) mit dem `FIRST_DEVELOPER_TESTS`-Tag versehen, der finale Stand zur Abgabe dieser Diplomarbeit wurde mit `DIPLOMA_THESIS_SUBMISSION` getaggt.

6.3 ZUSAMMENFASSUNG

Die Kombination aus objektiven Betrachtungen und Messungen sowie teils subjektivem Feedback von iOS-Entwicklern erlaubt eine umfangreiche Evaluation dieser Arbeit. Alle aufgestellten Anforderungskriterien aus den Kapiteln 4 und 5 konnten grundsätzlich erfüllt werden, wobei der zeitliche Overhead beim Build-Vorgang möglicherweise noch reduziert werden kann. Der deutlich wichtigere Overhead der Nachrichtenauflösung zur Laufzeit befindet sich bereits im kaum messbaren Bereich. Die Anforderung, die Xcode-Installation unverändert zu lassen, setzt die Grenzen der Integration des Prototyps. Wird eine engere Integration gefordert, beispielsweise für die Unterstützung von Live-Warnungen, müssen dafür Abstriche bei der Unveränderheit von Xcode in

Kauf genommen werden. Die Entscheidung über die Balance zwischen Integration und Unveränderbarkeit wurde in dieser Arbeit nur im Installationsprogramm getroffen, die einzelnen Werkzeuge sind davon unabhängig.

Die Auswertung der Entwicklertests zeichnet ein positives Gesamtbild mit Optimierungsmöglichkeiten im Detail, insbesondere der Codegenerierung und den Diagnose- bzw. Dokumentationsinformationen. Bereits getroffene Maßnahmen zur Verbesserung wurden im Fazit der Entwicklertests in Abschnitt 6.2.6 beschrieben.

Beide Evaluationsphasen bestätigen, dass das in Kapitel 4 aufgestellte Konzept die Abwärtskompatibilität von iOS-Apps wirkungsvoll steigern kann, umsetzbar ist und von Entwicklern, die mit den Standardwerkzeugen der Plattform vertraut sind, akzeptiert wird.

7 ZUSAMMENFASSUNG UND AUSBLICK

Zusammenfassung

In dieser Arbeit wurde untersucht, inwieweit Abwärtskompatibilitätsprobleme in iOS-Apps automatisch gefunden und aufgelöst werden können. Die derzeit lückenhafte Werkzeugunterstützung sorgt sowohl auf der Nutzer- als auch auf der Entwicklerseite für Probleme. Drei von zwölf im Rahmen der Arbeit befragten Entwickler haben angegeben, dass sie schon einmal als Nutzer[197] auf die Installation einer App aufgrund von Kompatibilitätsproblemen verzichten mussten. Die Hälfte der zwölf Entwickler sah sich in der Vergangenheit bereits mit Kompatibilitätsproblemen während der App-Entwicklung konfrontiert, wobei die Probleme in den meisten Fällen[198] durch den Verzicht auf neue APIs gelöst wurden. Dieses Vorgehen ist mehr als unbefriedigend, weil der Fortschritt der Plattform von den Apps nur mit großer Verzögerung genutzt wird. Insbesondere unter iOS bedeutet dies zudem eine Priorisierung einer beinahe marginalen Anzahl von Nutzern, während die Mehrheit der Nutzer eine neue Version des Betriebssystems innerhalb weniger Wochen adaptiert. Der Fokus dieser Arbeit liegt deshalb auf der **Priorisierung der neusten Version** und auf einfachen, werkzeuggestützten Möglichkeiten zum Herstellen der Kompatibilität mit Legacy-Systemen.

Bislang haben sich, unabhängig von der betrachteten Plattform, kaum wissenschaftliche Arbeiten mit dem Thema auseinandergesetzt. Zur **Grundlagenforschung** in Kapitel 2 wurden daher vor allem Ansätze aus der Praxis analysiert und abstrahiert. Betrachtet wurden auch Arbeiten zu verwandten Themen wie der Cross-Platform-Entwicklung. Die Recherche ergab eine Liste von Ansätzen, die nach ihren Anwendungsbereichen (Detektion oder Auflösung) gruppiert wurden. Einige der Ansätze sind eng an eine Plattform gebunden, andere sind technologieunabhängige Konzepte und Vorschläge. Es konnte kein Konzept oder praktischer Ansatz gefunden werden, der sowohl die Detektion als auch die Auflösung umfasst.

[197] Die Zahlen beziehen sich auf eine initiale Entwicklerumfrage, an der iOS- und Android-Entwickler teilgenommen haben
[198] Vier von sechs Entwicklern

Die Liste aller möglichen Ansätze wurde in Kapitel 3 eingeengt, indem überprüft wurde, welche der Ansätze unter iOS und im Zusammenspiel mit Apples Entwicklungsumgebung Xcode anwendbar sind. Um den **Auswahlprozess** nachvollziehbar zu gestalten, wurde zu Beginn des Kapitels eine Einführung in die wichtigsten Komponenten und Rahmenbedingungen der Plattform gegeben. Einige der Ansätze greifen in bestehende Compile- und Laufzeitmechanismen ein. Weil ein detailliertes Wissen in diesen Bereichen nicht vorausgesetzt wird, wurden diese ausführlich beschrieben.

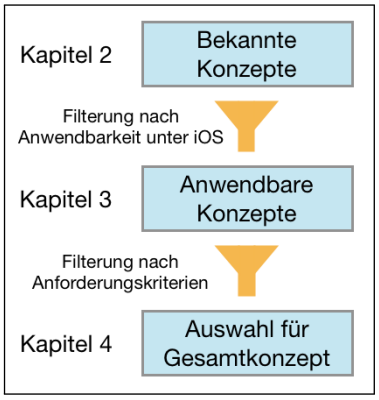


Abbildung 7.1: Kapitelweise Auswahl der Ansätze für das Gesamtkonzept

In Kapitel 4 wurden zuerst Anforderungskriterien aufgestellt, nach denen eine weitere Auswahl und schließlich die Erstellung eines Gesamtkonzepts erfolgt ist. Die schrittweise Auswahl der Teilkonzepte über drei Kapitel ist in Abbildung 7.1 dargestellt. Das **Gesamtkonzept** setzt sich aus den drei Phasen der Vorbereitung (Sammlung von Verfügbarkeitsinformationen), der Detektion und der Auflösung zusammen, wobei die Vorbereitungsphase bei dem gewählten Ansatz bereits vom Compiler implementiert wird. Aus Performancegründen findet auch die Detektion während der Kompilierung statt und nutzt die vorhandenen Verfügbarkeitsattribute der Header-Dateien. Die Auflösung zur Laufzeit erfolgt in einer separaten Bibliothek, deren APIs je nach Typ statisch oder dynamisch eingebunden werden.

Die prototypische Implementierung aus Kapitel 5 hat die **Umsetzbarkeit** des Konzepts demonstriert und darüber hinaus die Grundlage für die Entwicklertests gelegt. Die hauptsächlich durch einen Fragebogen erhobenen Ergebnisse dieser Tests wurden im 6. Kapitel ausgewertet und haben wertvolle Erkenntnisse über Erweiterungs- und Verbesserungsmöglichkeiten geliefert. Den Einsatz der prototypischen Werkzeuge zur Detektion können sich alle befragten Entwickler bereits zum jetzigen Zeitpunkt an produktiven Apps vorstellen, für den Produktiveinsatz der Auflösungswerkzeuge muss insbesondere die Kompatibilität mit Xcode-Workspaces hergestellt werden. Im Rahmen der Evaluation wurden auch Messreihen durchgeführt, die einen erkennbaren, aber dennoch geringen Overhead der Werkzeuge beim Build-Vorgang ergeben haben. Dagegen beeinträchtigen die gewählten Ansätze zur Auflösung kritischer APIs die Laufzeit-Performance in keinster Weise. Insgesamt konnte nach der Evaluation ein sehr **positives Fazit** gezogen werden.

Ausblick

Nicht zuletzt aufgrund des großen Zuspruchs seitens der in die Evaluation eingebundenen Entwickler soll der entstandene Prototyp über die Diplomarbeit hinaus weiterentwickelt werden. Dabei werden die Grenzen der automatischen Detektion und Auflösung auszuloten sein. Mit iOS 7 wird Apple erstmals größere Änderungen hinsichtlich der GUI-Gestaltung und Interaktionsparadigmen vornehmen, sodass die Abwärtskompatibilität nicht auf die funktionale API-Ebene reduziert werden kann. Während die Verfügbarkeit einer API leicht werkzeuggestützt überprüfbar ist, gestaltet sich die Erkennung der **semantischen Veränderung** einer API ungleich schwieriger.

Um die Problematik der Abwärtskskompatibilität stärker in den Fokus der Entwickler zu rücken, wäre zumindest die Detektion kritischer APIs in den von Apple ausgelieferten Werkzeugen wünschenswert. Eine stabile, standardisierte[199] **AOP**-Technologie könnte den dynamischen Auflösungsmechanismus und die damit verbundene Codegenerierung deutlich vereinfachen. Die Umsetzung einer Auflösung mit Aspekten wurde bereits in Abschnitt 4.5 erklärt, aber aufgrund der fehlenden Basistechnologie nicht weiter verfolgt.

Aufgrund der Tatsache, dass die iOS-Plattform auf der Basis von Mac OS X entwickelt wurde und sich die Entwicklerwerkzeuge, Frameworks und Programmiersprachen stark ähneln, ist eine **Portierung** der Ergebnisse auf Mac OS X trivial. In Kapitel 4 wurde eine Übersicht gegeben, welche Voraussetzungen das Konzept stellt, um auch auf anderen Plattformen umgesetzt zu werden. Zukünftige Arbeiten können auf den dargelegten Erkenntnissen aufbauen, um die Abwärtskompatibilität von Software auf anderen Plattformen sicherzustellen und das Konzept für die Besonderheiten der Zielplattform optimieren. Insbesondere im mobilen Bereich ist durch die steigende Diversität von Hard- und Software nicht zu erwarten, dass sich die Problematik der Abwärtskompatibilität in Zukunft auf bestimmte Plattformen begrenzt oder allgemein an Bedeutung verliert.

[199] Hierbei kann es sich auch um einen Defacto-Standard handeln, wichtig ist vor allem die breite Anwendbarkeit auf alle iOS-Projekte

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

CODEBEISPIELVERZEICHNIS

LITERATURVERZEICHNIS

[AMC+07] ALVES, Vander ; MATOS, Pedro Jr. ; COLE, Leonardo ; VASCONCELOS, Alexandre ; BORBA, Paulo ; RAMALHO, Geber: Transactions on aspect-oriented software development IV. Version: 2007. http://dl.acm.org/citation.cfm?id=1793854.1793861. Berlin, Heidelberg : Springer-Verlag, 2007. – ISBN 3–540–77041–0, 978–3–540–77041–1, Kapitel Extracting and evolving code in product lines with aspect-oriented programming, 117–142

[anda] *Creating Backward-Compatible UIs*. Dokumentation. http://developer.android.com/training/backward-compatible-ui/new-implementation.html, Abruf: 01. März 2013

[andb] *<uses-sdk> Element*. Dokumentation. http://developer.android.com/guide/topics/manifest/uses-sdk-element.html, Abruf: 20. Juni 2013

[and13a] *Google Play Services: Reference*. Dokumentation. http://developer.android.com/reference/gms-packages.html. Version: Mai 2013, Abruf: 16. Mai 2013

[and13b] *Platform Versions*. Dokumentation. http://developer.android.com/about/dashboards/. Version: Juni 2013, Abruf: 20. Juni 2013

[app03] *Technical Note TN2064: Ensuring Backwards Binary Compatibility - Weak Linking and Availability Macros on Mac OS X*. Dokumentation. http://developer.apple.com/legacy/library/technotes/tn2064/_index.html. Version: Februar 2003, Abruf: 16. Mai 2013

[app06] *Framework Programming Guide: Frameworks and Weak Linking*. Dokumentation. https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WeakLinking.html. Version: November 2006, Abruf: 14. Mai 2013

[app09] *Objective-C Runtime Programming Guide*. Dokumentation. https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/. Version: Oktober 2009, Abruf: 26. März 2013

[app10a] *Objective-C Runtime Reference*. Dokumentation. https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/ObjCRuntimeRef/. Version: Juni 2010, Abruf: 26. März 2013

[app10b] *SDK Compatibility Guide*. Dokumentation. http://developer.apple.com/library/ios/#documentation/developertools/conceptual/cross_development/. Version: November 2010, Abruf: 19. Februar 2013

[app12a] *Dynamic Library Programming Topics: Overview of Dynamic Libraries*. Dokumentation. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/OverviewOfDynamicLibraries.html. Version: Juli 2012, Abruf: 22. Februar 2013

[app12b] *iOS Software License Agreement*. Dokumentation. http://images.apple.com/legal/sla/docs/iOS6.pdf. Version: 2012, Abruf: 20. März 2013

[app12c] *iOS Technology Overview*. Dokumentation. http://developer.apple.com/library/ios/#documentation/miscellaneous/conceptual/iphoneostechoverview/. Version: September 2012, Abruf: 14. März 2013

[app12d] *Programming with Objective-C: Customizing Existing Classes*. Dokumentation. http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html. Version: Dezember 2012, Abruf: 13. Mai 2013

[app13a] *Apple Developer Library Search: 'API Diffs'*. Dokumentation. http://developer.apple.com/library/ios/search/?q=API+Diffs. Version: Februar 2013, Abruf: 26. Februar 2013

[app13b] *iOS App Programming Guide: Advanced App Tricks*. Dokumentation. http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AdvancedAppTricks/AdvancedAppTricks.html. Version: April 2013, Abruf: 15. Mai 2013

[app13c] *NSObject Class Reference*. Dokumentation. https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/Reference/Reference.html. Version: Januar 2013, Abruf: 26. März 2013

[Ash11] ASH, Mike: *Friday Q&A 2011-12-02: Object File Inspection Tools*. Blog. http://www.mikeash.com/pyblog/friday-qa-2011-12-02-object-file-inspection-tools.html. Version: Dezember 2011, Abruf: 09. Juni 2013

[asp05] *Chapter 5. Load-Time Weaving*. Dokumentation. http://eclipse.org/aspectj/doc/next/devguide/ltw.html. Version: 2005, Abruf: 08. März 2013

[BFJR98] BRANT, John ; FOOTE, Brian ; JOHNSON, Ralph E. ; ROBERTS, Donald: Wrappers to the Rescue. In: *In Proceedings ECOOP '98, Volume 1445 Of LNCS*, Springer-Verlag, 1998, S. 396–417

[BH90] BÖCKER, Heinz-Dieter ; HERCZEG, Jürgen: What Tracers are Made Of. In: *ECOOP/OOPSLA '90 Conference Proceedings* Bd. 25, 1990

[bla11a] *Develop applications for different BlackBerry Device Software versions and BlackBerry smartphone models*. Dokumentation. http://supportforums.blackberry.com/t5/Java-Development/Develop-applications-for-different-BlackBerry-Device-Software/ta-p/444827. Version: März 2011, Abruf: 06. März 2013

[bla11b] *Previous BlackBerry JDE Versions*. Dokumentation. http://developer.blackberry.com/java/download/previousjdeversions/. Version: Dezember 2011, Abruf: 06. März 2013

[Cha04] CHAMBERS, Craig: *CSE 401: Introduction to Compiler Construction*. Vorlesungsfolien. http://www.cs.washington.edu/education/courses/cse401/04au/slides/09-29.slides.pdf. Version: 2004, Abruf: 10. Juni 2013

[chr11] *WritingClangPlugins*. Blog. http://code.google.com/p/chromium/wiki/WritingClangPlugins. Version: September 2011, Abruf: 29. Januar 2013

[cla13] *Objective-C Literals - Clang 3.4 documentation*. Dokumentation. http://clang.llvm.org/docs/ObjectiveCLiterals.html. Version: Juni 2013, Abruf: 12. Juni 2013

[Coo13] COOK, Tim: *WWDC 2013 Keynote*. Konferenzvideo. http://www.apple.com/apple-events/june-2013/. Version: Juni 2013, Abruf: 20. Juni 2013

[Dre11] DREPPER, Ulrich: How To Write Shared Libraries. (2011), Dezember

[Duc99] DUCASSE, Stéphane: Evaluating message passing control techniques in Smalltalk. In: *Journal of Object-Oriented Programming (JOOP)* 12 (1999), S. 39–44

[DW10] DAINTITH, John ; WRIGHT, Edmund: *A Dictionary of Computing*. 6. Edition. Oxford University Press, 2010

[ecl09] *Frequently Asked Questions: Basic AOP and AspectJ Concepts*. Dokumentation. http://eclipse.org/aspectj/doc/released/faq.php#concepts. Version: März 2009, Abruf: 08. März 2013

[Eim05] EIMERS, Dick: *Dynamic Updating for the Java Virtual Machine using Load-time Structural Reflection*. Utrecht University, Masterarbeit, Februar 2005

[eng13] *Apple: over 500 million iOS devices sold*. Webseite. http://www.engadget.com/2013/01/23/apple-over-500-million-ios-devices-sold/. Version: Januar 2013, Abruf: 30. Januar 2013

[eth12] *After Early Boom, New Numbers Suggest iOS 6 Adoption May Be Reaching A Plateau*. Webseite. http://techcrunch.com/2012/10/05/after-early-boom-new-numbers-suggest-ios-6-adoption-may-be-reaching-a-plateau/. Version: Oktober 2012, Abruf: 29. Januar 2013

[FCDR95] FORMAN, Ira R. ; CONNER, Michael H. ; DANFORTH, Scott H. ; RAPER, Larry K.: Release-to-release binary compatibility in SOM. In: *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 1995 (OOPSLA '95). – ISBN 0–89791–703–0, 426–438

[Gal10] GALLAGHER, Matt: *Dynamic ivars: solving a fragile base class problem*. Blog. `http://www.cocoawithlove.com/2010/03/dynamic-ivars-solving-fragile-base.html`. Version: März 2010, Abruf: 12. Mai 2013

[gam13] *iOS 6 penetration percentage*. Webseite. `http://www.game4mob.com/index.php/tech-articles/67-ios-5-penetration-percentage`. Version: Januar 2013, Abruf: 29. Januar 2013

[GGKS07] GAN, Zhi ; GUO, Ying C. ; KURANE, Rahul ; SRINIVASAN, Aravind: *Extending Portability of Java Code Through the Use of AOP*. Patent, Patentnummer US 2008/0222607 A1, März 2007

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994

[Hag12] HAGEDORN, Sebastian: *Konzepte für Mehrmandantenfähigkeit von komplexen iOS-Apps*. TU Dresden, Belegarbeit, November 2012

[hei13] *WhatsApp unterstützt iPhone 3G nicht mehr*. Webseite. `http://www.heise.de/mac-and-i/meldung/WhatsApp-unterstuetzt-iPhone-3G-nicht-mehr-1775951.html`. Version: Januar 2013, Abruf: 30. Januar 2013

[Her12] HERBERT, Laurence K.: *C/C++ Student Feedback & Learning Tool – A Static & Dynamic Analyser for C & C++*. The University of Nottingham, Masterarbeit, Mai 2012

[Hir03] HIRSCHFELD, Robert: AspectS - Aspect-Oriented Programming with Squeak. In: *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. London, UK, UK : Springer-Verlag, 2003 (NODe '02). – ISBN 3–540–00737–7, 216–232

[ios13] *Optimizing Apps for iOS 6*. Webseite. `https://developer.apple.com/devcenter/ios/checklist/`. Version: Juni 2013, Abruf: 21. Juni 2013

[Isa11] ISAACSON, Walter: *Steve Jobs*. 1. Edition. Simon & Schuster, 2011

[Jah12] JAHCHAN, Rudy: *Monkey-Patching iOS with Objective-C Categories Part I: Simple Extensions and Overrides*. Blog. `http://blog.carbonfive.com/2012/01/23/monkey-patching-ios-with-objective-c-categories-part-1-simple-extensions-and-overrides/`. Version: Januar 2012, Abruf: 25. März 2013

[Kre10] KREMENEK, Ted: *LLVM Technologies in Depth*. Konferenzvideo. `https://developer.apple.com/videos/wwdc/2010/`. Version: Juni 2010, Abruf: 24. Juni 2013

[Lak05] LAKOS, John: *Large-scale C++ software design*. 16. Edition. Addison Wesley Longman Publishing Co., Inc., 2005

[Lat06] LATTNER, Chris: *Introduction to the LLVM Compiler Infrastructure*. Konferenzvortrag. `http://www.nondot.org/sabre/Resume.html#talks`. Version: April 2006, Abruf: 29. März 2013

[Lat07] LATTNER, Chris: *The LLVM Compiler System*. Konferenzvortrag. `http://www.nondot.org/sabre/Resume.html#talks`. Version: März 2007, Abruf: 29. März 2013

[Lev12] LEVIN, Jonathan: *Mac OS X and iOS Internals : To the Apple's Core*. 1. Edition. Wrox, 2012

[Lik32] LIKERT, Rensis: A Technique for the Measurement of Attitudes. In: *Archives of Psychology* 22 (1932), Nr. 140, S. 1–55

[llv13] *LLVM Coding Standards - LLVM 3.4 documentation*. Dokumentation. `http://llvm.org/docs/CodingStandards.html`. Version: Juni 2013, Abruf: 09. Juni 2013

[Lut12] LUTHI, Cédric: *Friday Q&A 2012-06-22: Objective-C Literals*. Kommentar. `http://www.mikeash.com/pyblog/friday-qa-2012-06-22-objective-c-literals.html#comment-c4ce71c3d04c25f8f40c2b5c5e130a29`. Version: Juni 2012, Abruf: 13. Juni 2013

[Mas10] MASON, Henry: *Future Proofing Your Application - Forewarned is forearmed*. Konferenzvideo. `https://developer.apple.com/videos/wwdc/2010/`. Version: Juni 2010, Abruf: 24. Juni 2013

[Mei12] MEIER, Reto: *Professional Android 4 Application Development*. 3. Edition. Apress, 2012

[MS98] MIKHAJLOV, Leonid ; SEKERINSKI, Emil: A Study of The Fragile Base Class Problem. In: *European Conference on Object-Oriented Programming*, Springer-Verlag, 1998, S. 355–382

[mur11a] MURAT: *iPhone app developed with SDK 4.2, requires backward compatibility with iOS 3.1.3 .. easy way? (Answer)*. Blog. `http://stackoverflow.com/a/7867986/2050985`. Version: Oktober 2011, Abruf: 17. Mai 2013

[Mur11b] MURPHY, Mark: *Beginning Android 3*. Apress, 2011

[NAR08] NICOARA, Angela ; ALONSO, Gustavo ; ROSCOE, Timothy: Controlled, systematic, and efficient code replacement for running java programs. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA : ACM, 2008 (Eurosys '08). – ISBN 978–1–60558–013–5, 233–246

[ND13] NORBYE, Tor ; DUCROHET, Xavier: *Google I/O 2013 - What's New in Android Developer Tools*. Konferenzvideo. `http://www.youtube.com/watch?v=lmv1dTnhLH4`. Version: Mai 2013, Abruf: 18. Mai 2013

[nsa12] *NSArray.h*. Quellcode/Header, 2012

[nsd12] *NSDictionary.h*. Quellcode/Header, 2012

[obj12] *Objective-C Feature Availability Index*. Dokumentation. http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/ObjCAvailabilityIndex/index.html. Version: Dezember 2012, Abruf: 04. April 2013

[ora05] *Binary Compatibility*. Dokumentation. http://docs.oracle.com/javase/specs/jls/se5.0/html/binaryComp.html. Version: 2005, Abruf: 22. Februar 2013

[ora13] *iOS Version Statistics – January 14th 2013*. Blog. http://www.14oranges.com/2013/01/ios-version-statistics-january-14th-2013/. Version: Januar 2013, Abruf: 29. Januar 2013

[oxf10] *Oxford Dictionary of English*. 3. Edition. Oxford University Press, 2010

[Pas86] PASCOE, Geoffrey A.: Encapsulators: A New Software Paradigm in Smalltalk-80. In: *OOPSLA '86 Proceedings*, 1986

[Pow10] POWELL, Adam: *How to have your (Cup)cake and eat it too*. Blog. http://android-developers.blogspot.de/2010/07/how-to-have-your-cupcake-and-eat-it-too.html. Version: Juli 2010, Abruf: 01. März 2013

[Red11] REDDY, Martin: *API Design for C++*. Morgan Kaufmann, 2011

[red12] *Red Hat Enterprise Linux: Application Compatibility Specification*. Webseite. http://www.redhat.com/f/pdf/rhel/RHEL6_App_Compatibility_WP.pdf. Version: Februar 2012, Abruf: 15. Februar 2013

[Ree11] REESE, George: *API Versioning*. Webseite. http://broadcast.oreilly.com/2011/10/api-versioning.html. Version: Oktober 2011, Abruf: 05. März 2013

[Ros11] ROSE, Jordy: *Using Clang from SVN in Xcode*. Blog. http://belkadan.com/blog/2011/07/Using-Clang-from-SVN-in-Xcode/. Version: Juli 2011, Abruf: 10. Juni 2013

[Sag98] SAGAR, Ajit: *Reflection & Introspection: Objects Exposed*. Webseite. http://java.sys-con.com/node/35980. Version: Mai 1998, Abruf: 06. März 2013

[Sau12] SAUVE, Chris: *iOS Ebb and Flow*. Webseite. http://pxldot.com/18754186750. Version: März 2012, Abruf: 20. Juni 2013

[SDH05] STUART, Jeffery A. ; DASCALU, Sergiu M. ; HARRIS, Frederick C.: Towards A Unified Approach for Cross-Platform Software Development. In: *Proceedings of the 14th International Conference on Intellegent and Adaptive Systems and Software Engineering (IASSE-2005)*, 2005, S. 235–242

[See12] SEELEMANN, Max: *Objective-C kompakt - Ein Kurs für Umsteiger und Fortgeschrittene*. dpunkt.verlag, 2012

[Smi13] SMITH, David: *iOS Version Stats*. Blog. http://david-smith.org/iosversionstats/. Version: Januar 2013, Abruf: 29. Januar 2013

[SR07] SAVGA, Ilie ; RUDOLF, Michael: Refactoring-Based Support for Binary Compatibility in Evolving Frameworks. In: *GPCE '07 Proceedings of the 6th international conference on Generative programming and component engineering* (2007), S. 175–184

[SR09] SAVGA, Ilie ; RUDOLF, Michael: Designing Adapters for Binary Backward Compatibility in Refactored Framework APIs / TU Dresden. 2009. – Forschungsbericht

[Sta11] STALLMAN, Richard: *About the GNU Project*. Webseite. `http://www.gnu.org/gnu/thegnuproject.html`. Version: September 2011, Abruf: 29. März 2013

[Ste12] STEINBERGER, Peter: *Using Subscripting With Xcode 4.4 and iOS 4.3+*. Blog. `http://petersteinberger.com/blog/2012/using-subscripting-with-Xcode-4_4-and-iOS-4_3/`. Version: Juli 2012, Abruf: 14. Juni 2013

[Ste13] STEINBERGER, Peter: *PSTCollectionView.m*. Quellcode. `https://github.com/steipete/PSTCollectionView/commit/41535edfae6d233a0b89879e7b3f20b5c51a466b`. Version: Juni 2013, Abruf: 03. Juni 2013

[sto13] *App Store Review Guidelines*. Webseite. `https://developer.apple.com/appstore/resources/approval/guidelines.html`. Version: 2013, Abruf: 12. Mai 2013

[TB11] TURNER, Jim ; BEHRENS, Jake: *Improving the Stability of Your App - Making the CrashMan go away*. Konferenzvideo. `https://developer.apple.com/videos/wwdc/2011/`. Version: Juni 2011, Abruf: 24. Juni 2013

[Tho13] THOMPSON, Mattt: *AFHTTPRequestOperation.m*. Quellcode. `https://github.com/AFNetworking/AFNetworking`. Version: Mai 2013, Abruf: 13. Mai 2013

[Tur12] TURNER, Jim: *Up and Running - Making a great impression with every launch*. Konferenzvideo. `https://developer.apple.com/videos/wwdc/2012/?id=225`. Version: Juni 2012, Abruf: 29. Mai 2013

[ult12] ULTRAMIRACULOUS: *App Store - Method Swizzling Legality (Question)*. Blog. `http://stackoverflow.com/questions/8834294/app-store-method-swizzling-legality`. Version: Januar 2012, Abruf: 13. Mai 2013

[Vas13] VASIC, Ivan: *Dealing with false positives*. Blog. `http://www.deploymateapp.com/news/dealing-with-false-positives/`. Version: April 2013, Abruf: 15. Mai 2013

[VBAM09] VILLAZÓN, Alex ; BINDER, Walter ; ANSALONI, Danilo ; MORET, Philippe: Advanced Runtime Adaptation for Java. In: *Proceedings of the eighth international conference on Generative programming and component engineering*. New York, NY, USA : ACM, 2009 (GPCE '09). – ISBN 978–1–60558–494–2, S. 85–94

[WK08] WOJTCZYK, M. ; KNOLL, A.: A Cross Platform Development Workflow for C/C++ Applications. In: *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, 2008, S. 224 –229

[xco12] *Xcode 4.5 Release Notes*. Dokumentation. `https://developer.apple.com/library/mac/releasenotes/DeveloperTools/RN-Xcode/#//apple_ref/doc/uid/TP40001051-SW174`. Version: 2012, Abruf: 22. März 2013

[xco13] *Xcode 4.6 Release Notes*. Dokumentation. `http://developer.apple.com/library/ios/releasenotes/DeveloperTools/RN-Xcode/index.html#//apple_ref/doc/uid/TP40001051-SW216`. Version: 2013, Abruf: 29. März 2013

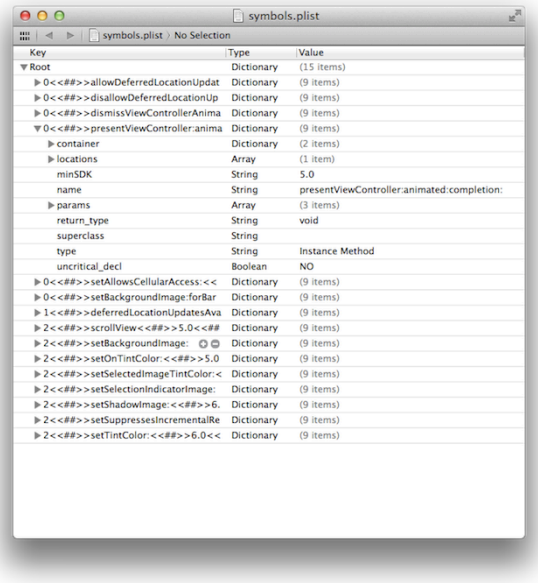
A DATEIEN

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

A.1 EXPORT-PLIST

Pfad: `Appendix/symbols.plist`

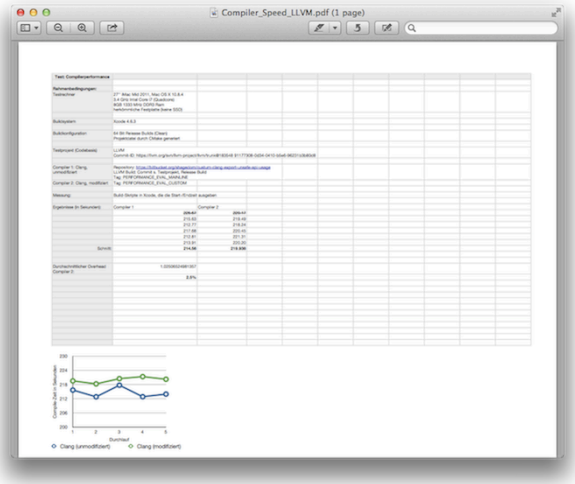
Vorschau:



A.2 TESTERGEBNISSE: LLVM-BUILDS

Pfad: `Appendix/Compiler_Speed_LLVM.pdf`

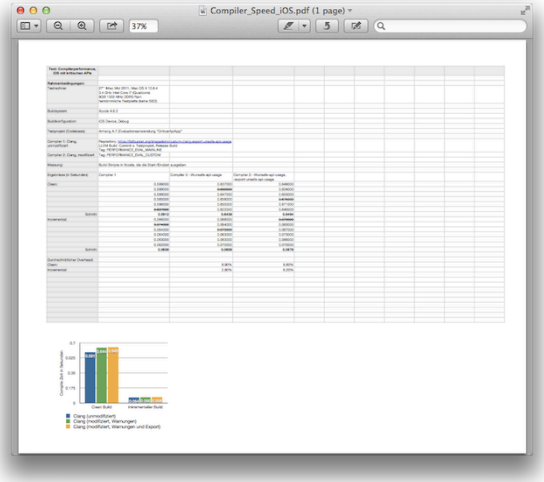
Vorschau:



A.3 TESTERGEBNISSE: iOS-APP-BUILDS

Pfad: `Appendix/Compiler_Speed_iOS.pdf`

Vorschau:



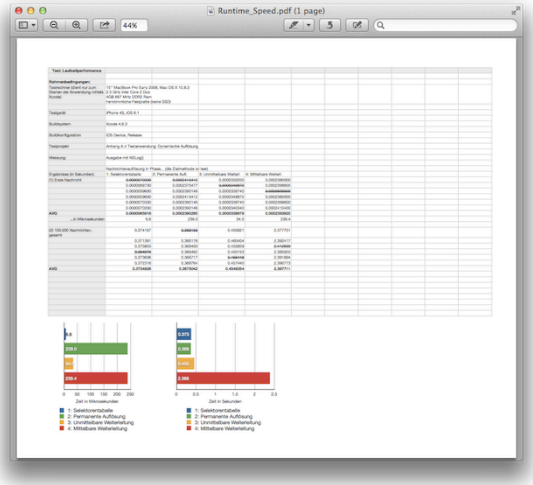
A.4 TESTANWENDUNG: DYNAMISCHE AUFLÖSUNG

Pfad: `Appendix/ResolveSpeedTest.zip`

A.5 TESTERGEBNISSE: DYNAMISCHE AUFLÖSUNG

Pfad: `Appendix/Runtime_Speed.pdf`

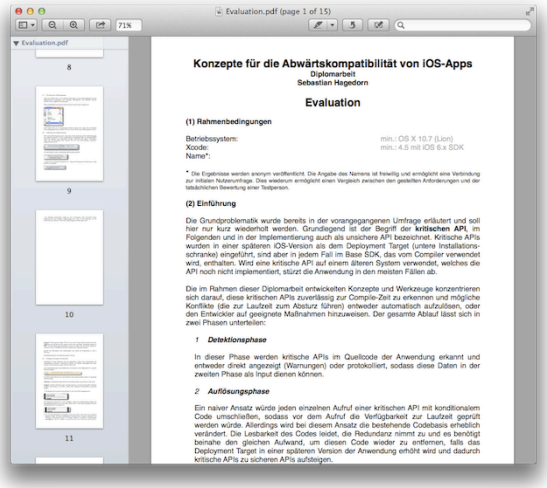
Vorschau:



A.6 EVALUATIONSBOGEN

Pfad: `Appendix/Evaluation.pdf`

Vorschau:



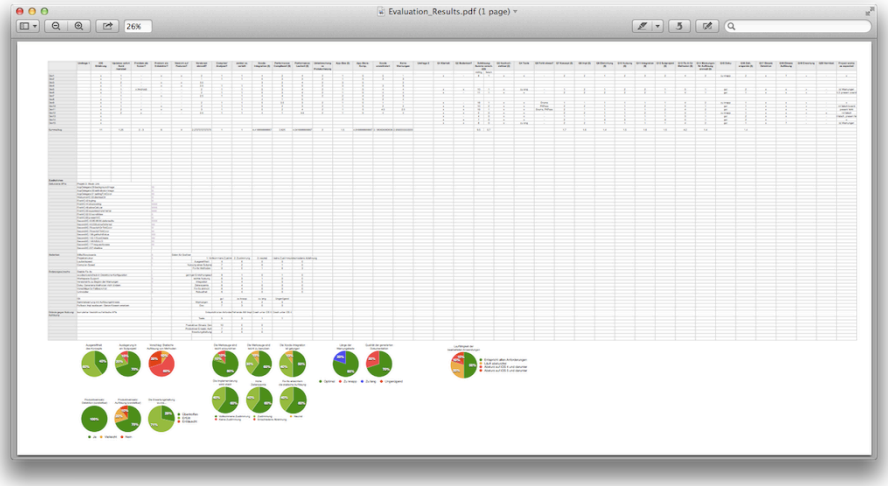
A.7 EVALUATIONSANWENDUNG

Pfad: `Appendix/CriticalApiApp.zip`

A.8 EVALUATIONSERGEBNISSE (ANONYMISIERT)

Pfad: `Appendix/Evaluation_Results.pdf`

Vorschau:



B PROTOTYP

B.1 xcodeprojectcheck

Pfad: `Prototype/prototype.zip`: `/Tools/XcodeProjectCheck/`

Repository: `https://bitbucket.org/shagedorn/unsafe-api-resolver`

B.2 CUSTOM CLANG COMPILER

Pfad: `Prototype/custom_clang.zip`

Repository: `https://bitbucket.org/shagedorn/custum-clang-export-unsafe-api-usage`

Modifizierte und neu hinzugefügte Dateien:

clangBasic:
`include/clang/Basic/APIExporter.h` [neu]
`include/clang/Basic/APIExporterCWrapper.h` [neu]
`include/clang/Basic/APIExporterSymbol.h` [neu]
`include/clang/Basic/DiagnosticSemaKinds.td` [modifiziert]
`lib/Basic/APIExporter.mm` [neu]
`lib/Basic/APIExporterSymbol.mm` [neu]
`lib/Basic/CMakeLists.txt` [modifiziert]

clangDriver:
`include/clang/Driver/Options.td` [modifiziert]
`tools/driver/cc1_main.cpp` [modifiziert]
`tools/driver/CMakeLists.txt` [modifiziert]

clangFrontend:
`include/clang/Frontend/FrontendOptions.h` [modifiziert]
`lib/Frontend/CompilerInvocation.cpp` [modifiziert]

clangSema:
`include/clang/Sema/Sema.h` [modifiziert]
`lib/Sema/SemaExpr.cpp` [modifiziert]
`lib/Sema/SemaExprObjC.cpp` [modifiziert]

clangSerialization:
`lib/Serialization/ASTReader.cpp` [modifiziert]

libClang:
`tools/libclang/CMakeLists.txt` [modifiziert]

B.3 unsafeApiResolver

Pfad: `Prototype/prototype.zip`

Repository: `https://bitbucket.org/shagedorn/unsafe-api-resolver`

B.4 CODE REPOSITORY

Pfad: `Prototype/snippets.zip`

Repository: `https://bitbucket.org/shagedorn/extendedsnippets`

B.5 INSTALLER

Pfad: `Prototype/prototype.zip`: `/Installer/Unsafe API Usage Tool Installer/`

Repository: `https://bitbucket.org/shagedorn/unsafe-api-resolver`

C CD