Transferleistung Theorie/Praxis [6

Matrikelnummer:	8252
Freigegebenes Thema:	Analyse der Verwendung von Datentypen in Swift Projekten
Studiengang, Zenturie:	A17b

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

Das Konzept der typisierten Daten ist in der heutigen Programmierung nicht mehr wegzudenken. Da Computer nur auf Bitmustern arbeiten und diese Daten keine Typisierung haben, muss das Konzept von Datentypen softwareseitig implementiert sein. Die meisten Programmiersprachen stellen dafür ein Typsystem zur Verfügung, um die Verwendung zu vereinfachen.[12] Viele Programmiersprachen bieten eine Reihe von fertigen Datentypen an, die verwendet werden können. Dazu gehören boolesche Werte, Zahlen, Zeichenketten und komplexere Strukturen wie Listen und Mengen. Darüber hinaus können häufig eigene Datentypen definiert werden.

Häufig gibt es zum Lösen eines Problems verschiedene Lösungsmöglichkeiten. Die unterschiedlichen Lösungen machen dabei auch Gebrauch von unterschiedlichen Typen. Dafür kann es verschiedene Gründe geben. Eines der wichtigsten Konzepte der Informatik ist zum Beispiel die Abstraktion. Dies spiegelt sich auch in den Typsystemen wieder. In objektorientierte Programmiersprachen kann man zum Beispiel mit der Möglichkeit Verhalten und den Zustand zu vererben Abstraktion einbauen. Außerdem kann mithilfe von Schnittstellendefinitionen die Schnittstelle von der konkreten Implementation getrennt werden. Wenn andere Abstraktionen eingesetzt werden, muss auch auf andere Typen zurückgegriffen werden.[14], [15]

Ziel der Arbeit ist es, die Verwendung von verschiedenen Arten von Typen zu analysieren. Dabei wird zunächst eine Methode entwickelt, die die für die Analyse notwendigen Informationen aus bestehenden Projekten extrahiert. Anschließend werden die so gesammelten Daten analysiert.

Es ist nicht das Ziel der Arbeit, die Ergebnisse der Analyse zu bewerten. Das heißt im speziellen auch, dass für die Projekte keine Verbesserungsvorschläge gemacht werden können. Es werden zunächst lediglich Muster, Unterschiede und Gemeinsamkeiten ermittelt.

2 Methodik

2.1 Auswahl der Projekten

Um die Verwendung von verschiedenen Arten von Datentypen zu untersuchen, werden einige bestehende Softwareprojekte analysiert. Dabei sollen unter anderem auch die internen Projekte aus der Firma mit in den Vergleich einbezogen werden. Diese sind in der Programmiersprache *Swift* implementiert. Da sich die Typesysteme von verschiedenen Programmiersprachen signifikant voneinander unterscheiden können, sodass ein Vergleich nur schwer umsetzbar ist, wird sich in der Analyse auf Softwareprojekte in der Programmiersprache *Swift* beschränkt. Dabei können die Projekte auch *C*, *C++*, *Objective-C* und *Objective-C+* verwenden, da die Typen

dieser Programmiersprachen transparent importiert werden[1].

Zunächst sollen die beiden internen Projekte *mail-ios* und *cloud-ios* aus Tabelle 1 analysiert werden. Beide Projekte sind iOS Apps, die sowohl auf iPhones als auch iPads laufen.

Name	Type
mail-ios	iOS App
flex-ios	iOS App

Tabelle 1: Interne Projekte *(Eigene Auflistung)*

Darüber hinaus sollen bekannte *Swift* Projekte verwendet werden. Dafür wurde auf GitHub unter dem Topic Swift die Projekte nach der Programmiersprache Swift gefiltert und nach der Anzahl der Sterne sortiert. Diese sich ändernde Liste kann über `https://github.com/topics/swift?l=swift&o=desc&s=stars` abgerufen werden. Zum aktuellen Zeitraum wurden etwa 23000 Projekte gefunden.

Von diesen wurden die ersten 10 Projekte genommen. In der Liste tauchen auch ein paar Projekte auf, die lediglich kuratierte Listen andere Projekte sind, selbst jedoch keinen Code enthalten[1]. Diese Projekte wurden übersprungen. Daraus ergibt sich die in Tablelle 2 dargestellte Liste von Projekten.

Name	Type	URL
Alamofire	Paket	`https://github.com/Alamofire/Alamofire`
ShadowsocksX-NG	macOS App	`https://github.com/shadowsocks/ShadowsocksX-NG`
iina	macOS App	`https://github.com/iina/iina`
SwiftyJSON	Paket	`https://github.com/SwiftyJSON/SwiftyJSON`
ReactiveCocoa	Paket	`https://github.com/ReactiveCocoa/ReactiveCocoa`
vapor	Paket	`https://github.com/vapor/vapor`
Hero	Paket	`https://github.com/HeroTransitions/Hero`
RxSwift	Paket	`https://x.com/ReactiveX/RxSwift`
Kingfisher	Paket	`https://github.com/onevcat/Kingfisher`
SnapKit	Paket	`https://github.com/SnapKit/SnapKit`

Tabelle 2: Ausgewählte Open-Source Projekte *(Eigene Auflistung)*

2.2 Untersuchung von Projekten

Bei den einzelnen Projekte soll untersucht werden, welche Arten von Datentypen verwendet werden. Dafür muss zunächst ermittelt werden, welche Datentypen verwendet werden. Diese Information lässt sich jedoch nicht mithilfe einer Syntaxanalyse des Source Codes ermitteln.

[1]Zum Beispiel: `https://github.com/dkhamsing/open-source-ios-apps`

Dies hat verschiedene Gründe. Zunächst ist Swift keine rein explizite Programmiersprache. Bei expliziten Programmiersprachen muss der verwendete Datentyp immer angegeben werden[12]. An vielen Stellen taucht der verwendete Datentyp jedoch nicht im Source Code auf. Zweitens kann bei importieren Datentypen die Art des Datentypens erst durch das richtige Linken gegen die Abhängigkeiten bestimmt werden. Zuletzt können verschiedene Datentypen gleich heißen, sodass auch hier eine Syntaxanalyse die benötigten Informationen nicht liefern kann. Damit scheidet eine reine Syntaxanalyse aus.

Neben dem Swift Compiler gibt es das Framework *SourceKit*. SourceKit ist ein Framework, das zur Unterstützung von Integrierten Entwickungsumgebungen (IDEs) entwickelt wird. Einige der angebotenen Features benötigen ebenfalls mehr als eine reine Syntaxanalyse. Das Framework kann unter anderem Informationen wie Optionen für eine Autovervollständigung von Quellcode bestimmen. Da dafür die Information, welcher Datentyp vorliegt, benötigt wird, kann das Framework die Datentypen bestimmen.[3]

Daher benötigt SourceKit das gesamte Projekt in einem kompilierfähigen Zustand, um all Informationen ausgeben zu können. Das bedeutet aber auch, dass zunächst alle Dependencies richtig installiert sein müssen.

Als Framework stellt SourceKit nur eine C API zur Verfügung, die genutzt werden kann. Daher wurde das Tool *SourceKitten* erstellt, welches eine Komandozeilenschnittstelle bereitstellt, um mit SourceKit zu interagieren. SourceKitten bietet verschiedene Operation an. Zum Beispiel kann man für eine Stelle im Source Code Autovervollständigungsvorschläge ausgeben lassen.[16]

Am Interessantesten für die Analyse ist der Befehl `sourcekitten doc`. Dieser erstellt eine JSON Dokumentation für das Projekt, die unter anderem auch alle verwendeten Typen enthält. In dieser Dokumentation sind die Typen auf mehrere Arten enthalten. Dabei sind einige Informationen maschinenlesbar und menschenlesbar abgelegt.

Im Folgenden soll beispielhaft aus dem Projekt *Alamofire* eine Definition von einer Instanzvariable untersucht werden. Der Eintrag in der JSON Dokumentation enthält dabei unter anderem diese Attribute:

deklaration: `public var emptyRequestMethods: Set<HTTPMethod>`

Der Ausschnitt aus dem Sourcecode, für den der Eintrag ist. Es handelt sich um eine öffentliche Variable mit dem Namen `emptyRequestMethods`.

kind: `source.lang.swift.decl.var.instance`

Es handelt sich um eine Instanzvariable.

typename: `Set<HTTPMethod>`

Der Typename, wie er aus der Deklaration übernommen wurde

typeusr: `$sShy9Alamofire10HTTPMethodVGD`

Der gemangelte Typ.

Zunächst können *deklaration* und *typename* nicht verwendet werden, da diese die vorher genannten Probleme eine Syntaxanalyse haben. *kind* kann verwendet werden, um die Definitionen zu filtern, die eine Verwendung von einem Datentypen darstellen. So verwendet das Deklarieren einer Variable einen Datentypen während das Deklarieren einer Klasse noch keine Verwendung darstellt.

Von besonderem Interesse ist hier jedoch *typeusr*. Dies enthält den gemangelte Typen des Eintrags. Typemangling beziehungsweise Mangling allgemein ist in Swift notwendig, um eindeutig Objekte zu referenzieren. Die gemangelten Namen sind deshalb immer eindeutig und enthalten bereits viele Informationen wie unter anderem den Typen und auch die Art des Typen.[2] Insbesondere bei generischen Typen können dabei mehrere rohe Typen verwendet werden. In diesem Beispiel sind das `Set` und `HTTPMethod`. Beide sind relevant, um die Verwendung von Typen zu analysieren. In Tabelle 3 ist dargestellt, wie der gemangelte Typ aus dem Beispiel gelesen werden muss. Wie die einzelnen Symbole zu interpretieren sind, ist in [2] dokumentiert.

Symbol	Bedeutung
`$s`	Es handelt sich um Swift stable mangling
`Sh`	S prefix für Swift Modul, h für Set
`y`	Anfang Bound Generic Type: Generischer Type mit gebundenen Typen
`9Alamofire`	Name des Moduls mit Längenprefix
`10HTTPMethod`	Name des Typens mit Längenprefix
`V`	Es handelt sich um ein Struct
`G`	Ende Bound Generic Type; Es gibt keine weiteren Typ Parameter
`D`	Ende der Typedefinition

Tabelle 3: Beispiel für einen gemangelten Typen *(Eigene Auflistung)*

Um diese komplexen Namen einlesen zu können, gibt es das Paket `CwlDemangle`. Diese kann gemangelte Namen einlesen und wie in Tabelle 3 angedeutet die einzelnen Komponenten des Namen bereitstellen.[13] Diese Komponenten können dann vergleichsweise einfach rekursiv untersucht werden, sodass die relevanten Teile extrahiert werden können.

Insgesamt kann nun für ein ganzes Projekt die Dokumentation mit SourceKitten als CLI für SourceKit erstellt, mithilfe von CwlDemangle eingelesen und anschließend ausgewertet werden. Dies wird soweit zusammengefasst, dass die wesentliche Information von einem Projekt in einer Tabelle mit 4 Spalten dargestellt werden kann: Modulname, Typename, Art des Typen und Anzahl der Verwendungen.

Die Schritte zum Erstellen der SourceKitten Dokumentation der einzelnen Projekte ist in

Abschnitt I zusammengefasst. Unter `https://github.com/XXXX/swift-type-analyzer`[2] ist der Code zum Verarbeiten der Dokumentation zu finden.

3 Auswertung

3.1 Arten von Typen

Die nach der in Abschnitt 2.2 beschriebenen Methode gesammelten Daten können nun ausgewertet werden. Dabei wird beim Vergleichen verschiedener Projekte oder Projektgruppen immer die relativen Anteile verglichen. Unterschiede in den absoluten Häufigkeiten sind zu erwarten und hängen vermutlich vor allem mit der Größe der Projekte zusammen. Andere Aspekte wie verwendete Abstraktion oder granularere Aufteilung in mehr Typen können hier ebenfalls die absoluten Häufigkeiten beeinflussen. Diese Aspekte sollen aber in der weiteren Analyse nicht betrachtet werden.

Zunächst können die verwendeten Arten von Typen über alle 12 Projekte analysiert werden. Insgesamt wurden in allen Projekten zusammen 44790 Typeverwendungen gefunden. Diese teilen sich auf 7 verschiedene Arten von Typen wie in Abbildung 1 dargestellt ungleichmäßig auf[3]. Die einzelnen Arten werden nun im folgenden detaillierter betrachtet.

Struct Structs sind ein Verbund von einem Zustand und Verhalten. Der Zustand wird dabei in Form von gespeicherten Eigenschaften und das Verhalten in Form von Funktionen abgebildet. Structs können nicht von anderen Structs erben (Inheritance). Außerdem sind Structs *value types*, dass heißt der Zustand wird kopiert, wenn zum Beispiel eine zweite Variable angelegt wird. Viele der eingebauten Standardtypen wie *Int* oder *String* sind Structs.[10] Insgesamt sind Structs die am häufigsten verwendete Typart mit insgesamt 11538 Verwendungen.

Enum Enums verhalten sich ähnlich wie Structs. Es gibt zwei verschiedene Arten von Enums, die sich in der Art der Zustandsspeicherung unterscheiden. Die erste Variante ist ähnlich zu Enums in C und wird durch eine endliche Liste an Zustanden abgebildet, die ein Enum annehmen kann. Die zweite Variante ist ähnlich zu Unions in C. Es gibt eine endliche Liste von Fällen, wobei jeder Fall unterschiedliche assoziierte Zustände haben kann.[5] Die zweite Variante wird unter anderem bei dem eingebauten Typen `Result` verwendet. Dabei gibt es zwei Fälle. Der erste Fall repräsentiert den Erfolgsfall mit dem entsprechenden Wert. Der zweite Fall

[2]Ich habe meinen Namen in der URL aufgrund des Double-Blind-Verfahrens der Transferleistungen geschwärzt. Bei Bedarf ist das Projekt aber (mit meinem Namen) auf GitHub zu finden.

[3]Alle nachfolgenden Abbildungen verwenden das selbe Farbschema

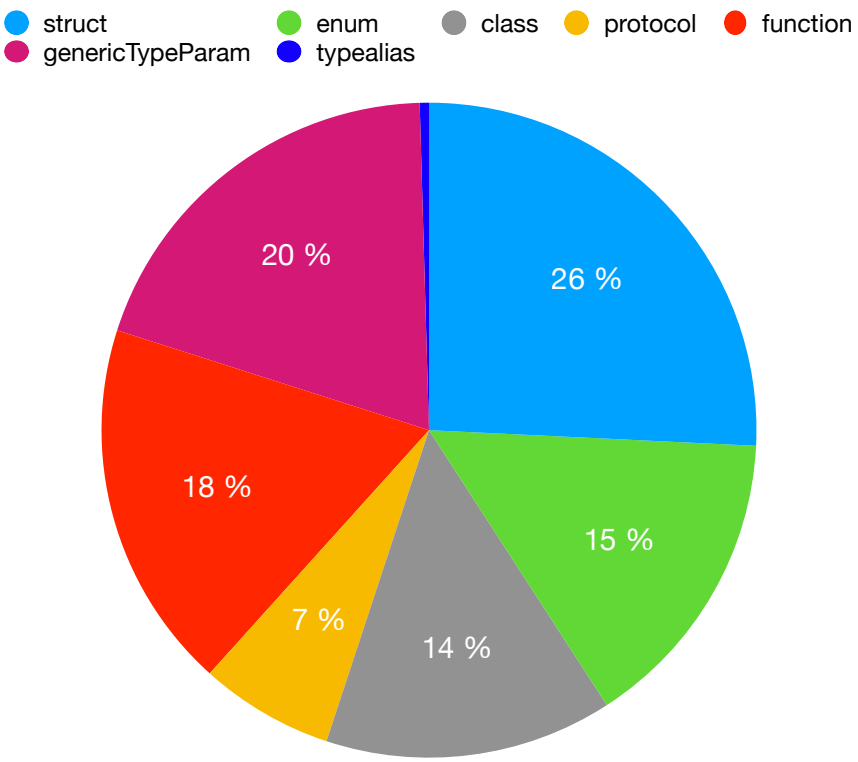


Abbildung 1: Verwendete Typen *(Eigene Darstellung)*

repräsentiert den Fehlerfall mit dem entsprechenden Fehler. Dabei kann immer nur einer der beiden Fälle vorhanden sein:[9]

```
enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}
```

Insgesamt wurden Enums 6763 mal in allem untersuchten Projekten verwendet.

Class Klassen sind ebenfalls ähnlich zu Structs. Sie erlauben aber zusätzlich das Erben von Zustand von Verhalten von anderen Klassen. Außerdem sind Klassen *reference types*. Das bedeutet, dass zum Beispiel beim Anlegen einer zweiten Variable lediglich eine Referenz auf die vorherige Instanz erstellt wird. Der Zustand wird nicht kopiert.[10] Insgesamt wurden Klassen 6364 mal in allen Projekten verwendet.

Protocol Protokolle sind die Definition von einer Schnittstelle, welche von einem Struct, einem Enum oder einer Klasse implementiert werden kann. Das Verwenden von Protokollen erlaubt das Festlegen, welchen minimalen Zustand von Verhalten ein Typ implementieren muss,

um verwendet zu werden. Dies erlaubt das Trennen von der Implementierung von der Schnittstelle. Neben einer reinen Schnittstellendefinition erlauben Protokolle auch das Definieren von Standardverhalten, sodass die konkreten Typen diesen nicht implementieren müssen. Das Protokoll kann dabei entscheiden, ob der konkrete Typ diese überschreiben kann.[8] Insgesamt wurden Protokolle 2961 mal in allen Projekten verwendet.

Function Funktionen bilden das Verhalten in Swift Programmen ab. Dabei können Funktionen an verschiedenen Stellen definiert werden. Zunächst können Funktionen als Teil des Verhaltens von Structs, Enums, Klassen und Protokollen definiert sein. Funktionen können aber auch Teil des Zustands sein und wie zum Beispiel Structs in Variablen gespeichert werden.[4], [6] Da sich die Struktur von Funktionen grundlegend von der der anderen Arten von Typen unterscheiden, werden Funktionen bei den weiteren Vergleichen nicht weiter betrachtet.

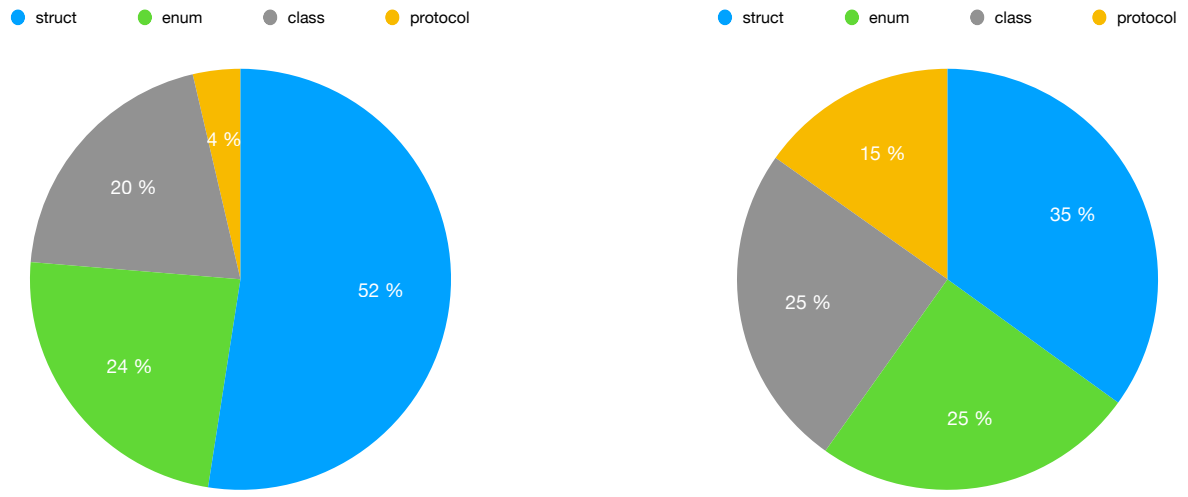
genericTypeParam Alle anderen Typen können generisch implementiert werden. Das bedeutet, dass ein oder mehrere Typen, die in dem generischen Typen verwendet werden, bei der Verwendung des generischen Typen festgelegt und nicht von dem Typen selbst vorgegeben werden. Ein Beispiel hierfür sind Arrays, die einen beliebigen Typen speichern können. Wird nun dieser Typ in der Implementation von Arrays verwendet, ist der tatsächliche Type und damit auch die eigentliche Art des Typen nicht bekannt.[7] Daher werden auch generische Type Parameter nicht weiter betrachtet.

Typealias Ein Typealias ist ein anderer Name für einen Typen. Dabei kann sich dahinter ein beliebiger anderer Typ verbergen. Dazu zählen auch wieder ein anderer Typealias. Da damit der tatsächliche Type und damit auch die eigentliche Art des Typen nicht bekannt ist, werden Typealiase nicht weiter betrachtet.[11] Insgesamt spielen Typealiase auch nur eine geringe Bedeutung mit 203 Verwendungen. Dies entspricht etwa 0.45% aller Typverwendungen.

3.2 Vergleich App und Paket

Nun soll untersucht werden, wie sich die Typverwendung zwischen Paketen und Apps unterscheidet. Vier der 12 untersuchten Projekten sind Apps. Dazu gehören die zwei internen Projekte (Tabelle 1) und zwei der open-source Projekte (Tabelle 2). Alle anderen Projekte sind Pakete.

Die relative Verteilung der untersuchten Typarten des Sourcecodes von den ausgewählten Apps und Pakete ist in Abbildung 2 dargestellt. Zunächst stellt der Anteil der Protokolle den größten Unterschied dar. In den untersuchten Paketen ist der Anteil der Protokolle mehr als



(a) App (b) Paket

Abbildung 2: Vergleich App und Paket *(Eigene Darstellung)*

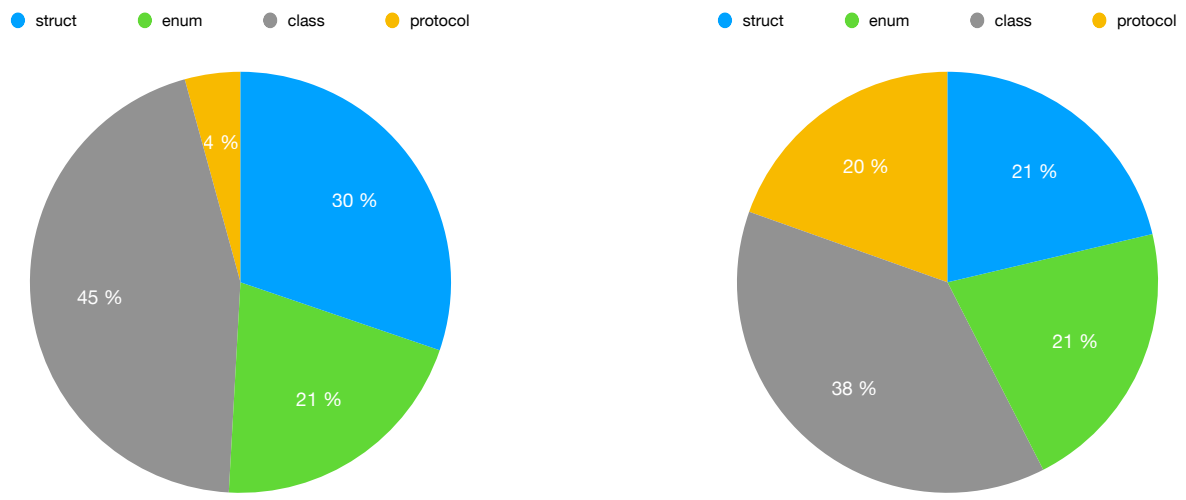
drei Mal so hoch wie in den untersuchten Apps. In den App Projekten werden Structs sehr viel mehr und Klassen etwas mehr eingesetzt. Der Anteil von Enums unterscheidet sich nur minimal.

Um die Verteilungen weiter zu betrachten, können alle Typen, die von Swift mitgeliefert werden, aus der Verteilung ausgeschlossen werden. Hierfür können alle Typen aus den Modulen `Swift` und `Foundation` gefiltert werden. Das Modul `Swift` stellt die Standardbibliothek von Swift dar und enthält viele Basistypen. In den untersuchten Projekten wurden davon am häufigsten `Optional`[4], `String`, `Bool`, `Array` und `Int` verwendet. Das Modul `Foundation` erweitert die Standardbibliothek um weitere Basistypen und ist bei den meisten Swiftinstallation vorhanden. Die am häufigsten verwendeten Typen hier sind `URL`, `Data`, `URLRequest` und `Date`.

Da es sich bei diesen Typen vor allem um Basistypen handelt, die von vielen Programmen gebraucht werden, kommen diese verhältnismäßig oft vor. In allen Projekten sind insgesamt 11762 der verwendeten Typen aus diesen beiden Modulen. Damit machen diese etwas mehr als 26% der verwendeten Typen aus. Daher ist es interessant nur den Teil zu betrachten, der innerhalb der Projekte entwickelt oder explizit als Abhängigkeit hinzugefügt wurde.

Die Verteilung ohne diese beiden Module ist in Abbildung 3 dargestellt. In beiden Verteilungen gab es einige größere Verschiebungen. In den App Projekten werden deutlich mehr Klassen und dafür deutlich weniger Structs verwendet. Dies liegt daran, dass Typen für UI häufig Klassen sind. Der Anteil der Enums hat sich dagegen nur geringfügig von 24% auf 21% verringert, während sich der Anteil der Protokolle gar nicht verändert hat. Bei den Paketen gibt es bei

[4] `Optional` ist ein generischer Type, um einen Wert oder die Abwesenheit eines Wertes zu speichern. Dies entspricht dem NULL Pointer aus vielen anderen Programmiersprachen.



(a) App (b) Paket

Abbildung 3: Vergleich App und Paket ohne Standardmodule *(Eigene Darstellung)*

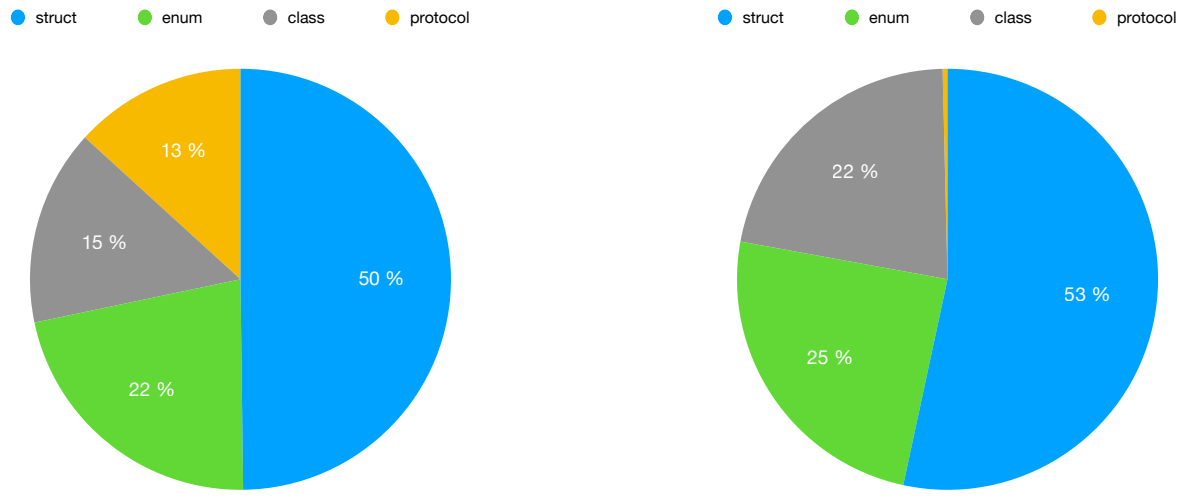
den Structs, Enums und Klassen eine ähnliche Veränderung. Der Anteil der Protokolle hat sich jedoch von 15% auf 20% erhöht. Damit ist der Anteil der Protokolle bei den Paketen nun etwa fünf Mal so hoch wie bei den Apps. Dies liegt daran, dass die von den Paketen definierten Typen im Vergleich zu den Apps häufiger Protokolle sind.

3.3 Vergleich Intern und Extern

Nun soll untersucht werden, wie sich die Typverwendung zwischen den internen und externen Projekten unterscheiden. Im Abschnitt 3.2 wurde bereits festgestellt, dass es zwischen Paketen und Apps einige Signifikante Unterschiede gibt. Da es keine internen Pakete als Vergleich gibt, sollen im Folgenden nur die zwei internen Apps mit den zwei externen Apps verglichen werden. Da damit die Anzahl der untersuchten Projekte sehr gering ist, sollten die folgenden Daten vorsichtig interpretiert werden. Es können trotzdem unterschiedlich Verwendungen aufgedeckt werden.

Wie in Abschnitt 3.2 wird wieder zunächst die Verteilung aller Typen analysiert und anschließend die Betrachtung auf die Typen, die nicht aus den Standardmodulen kommen, eingeschränkt.

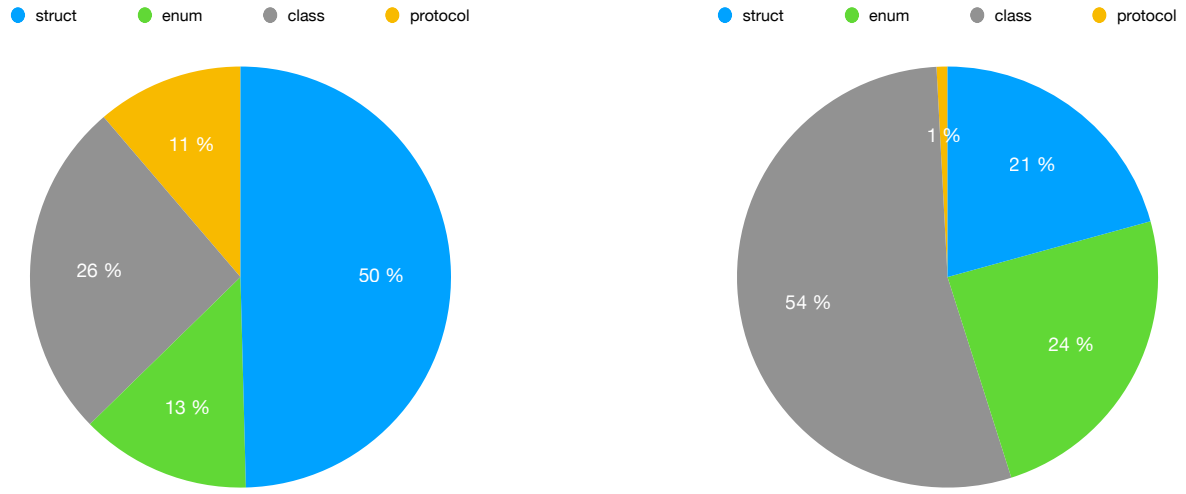
In Abbildung 4 sind die Verteilungen aller Typen dargestellt. Aus der vorangegangenen Analyse ging hervor, dass die untersuchten Apps relativ viele Structs verwenden. Hierbei unterscheiden sich die internen und externen Apps auch nur minimal. Es sind jeweils etwa 50% der Typeverwendungen Structs. Insgesamt ist die Verwendung von konkreten Typen bei den externen Apps jeweils etwas höher. Dafür ist die Verwendung von Protokollen bei den internen



(a) App (b) Paket

Abbildung 4: Vergleich Intern und Extern *(Eigene Darstellung)*

Apps etwa 14 mal so höher als bei den externen.



(a) Intern (b) Extern

Abbildung 5: Vergleich Intern und Extern *(Eigene Darstellung)*

Betrachtet man nun nur die Typen, die nicht aus den Standardmodulen kommen, verändert sich die Verteilung wie in Abbildung 5a dargestellt nur wenig. Die Anteile von Structs und Protokollen verändern sich nur minimal. Der Anteil der Klassen wird deutlich mehr und der Anteil der Enums nimmt deutlich ab. Dies liegt unter anderem daran, dass einige viel verwendete Typen der Standardlibraries wie `Optional` und `Result` als Enum implementiert sind, während Typen für UI häufig Klassen sind. Bei den externen Apps gab es deutlich stärkere Veränderungen. Hier hat sich der Anteil der Enums und Protokolle nur minimal verändert,

dafür haben sich die Anteile von Structs und Klassen von 53% und 22% etwa vertauscht, sodass die neuen Anteile nun 21% und 54% sind. Dies liegt daran, dass die externen Apps besonders viele `String`s und `Bool`s verwenden, selbst aber fast keine Structs definieren.

4 Zusammenfassung

Insgesamt konnte festgestellt werden, dass sich die Nutzung der Typen bei den verschiedenen Projektgruppen stark unterscheiden. Teilweise kann bereits aus den Daten abgeleitet werden, weshalb die Unterschiede auftreten. Die untersuchten App Projekte haben in der Regel mehr konkrete Typen verwendet. Insbesondere der Anteil der Klassen ist höher, da Klassen für das Erstellen der Graphischen Benutzeroberfläche nötig sind. Die Paket Projekte dagegen nutzen einen höheren Anteil von Protokollen.

Aus den Daten lässt sich jedoch nicht ableiten, ob die aktuelle Verteilung der Typverwendungen sinnvoll ist. In einem nächsten Schritt kann nun untersucht werden, ob das Verwenden bestimmter Typarten die Qualität des Codes verbessern kann. Außerdem könnte nun untersucht werden, ob die gefundenen Unterschiede zwischen den Projektgruppen sinnvoll ist.

In der Zukunft könnten außerdem noch weitere Analysen mit den gesammelten Daten gemacht werden. In dieser Arbeit wurden die Analysen vor allem mit den verschiedenen Typarten gemacht. Die gesammelten Daten erlauben jedoch noch genauere Analysen auf Typenebene. Dafür wird vermutlich jedoch eine größere Datenbasis benötigt.

Literatur

[1] Apple Inc. (16. Dez. 2020). „Conventions for foreign symbols,“ Adresse: https://github.com/apple/swift/blob/main/docs/ABI/Mangling.rst#conventions-for-foreign-symbols (besucht am 20. 12. 2020).

[2] ——, (16. Dez. 2020). „Mangling,“ Adresse: https://github.com/apple/swift/blob/main/docs/ABI/Mangling.rst (besucht am 20. 12. 2020).

[3] ——, (22. Dez. 2020). „SourceKit,“ Adresse: https://github.com/apple/swift/tree/main/tools/SourceKit (besucht am 20. 12. 2020).

[4] ——, (16. Sep. 2020). „Closure,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/Closures.html (besucht am 20. 12. 2020).

[5] ——, (16. Sep. 2020). „Enumerations,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html (besucht am 20. 12. 2020).

[6] ——, (16. Sep. 2020). „Functions,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/Functions.html (besucht am 20. 12. 2020).

[7] ——, (16. Sep. 2020). „Generics,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/Generics.html (besucht am 20. 12. 2020).

[8] ——, (16. Sep. 2020). „Protocols,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (besucht am 20. 12. 2020).

[9] ——, (2020). „Result — Apple Developer Documentation,“ Adresse: https://developer.apple.com/documentation/swift/result (besucht am 20. 12. 2020).

[10] ——, (16. Sep. 2020). „Structures and Classes,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html (besucht am 20. 12. 2020).

[11] ——, (16. Sep. 2020). „Typealias,“ Adresse: https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID27 (besucht am 20. 12. 2020).

[12] L. Cardelli und P. Wegner, „On Understanding Types, Data Abstraction, and Polymorphism,“ *ACM Comput. Surv.*, Jg. 17, Nr. 4, S. 471–523, Dez. 1985, ISSN: 0360-0300. DOI: 10.1145/6041.6042. Adresse: https://doi.org/10.1145/6041.6042.

[13] CwlDemangle Contributors. (12. Juni 2020). „CwlDemangle,“ Adresse: https://github.com/mattgallagher/CwlDemangle (besucht am 20. 12. 2020).

[14] B. Liskov und S. Zilles, „Programming with Abstract Data Types,“ *SIGPLAN Not.*, Jg. 9, Nr. 4, S. 50–59, März 1974, ISSN: 0362-1340. DOI: 10.1145/942572.807045. Adresse: https://doi.org/10.1145/942572.807045.

[15] P. A. Sivilotti und M. Lang, „Interfaces First (and Foremost) with Java,“ in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, Ser. SIGCSE '10, Milwaukee, Wisconsin, USA: Association for Computing Machinery, 2010, S. 515–519, ISBN: 9781450300063. DOI: 10.1145/1734263.1734436.

[16] SourceKitten Contributors. (11. Dez. 2020). „SourceKitten,“ Adresse: https://github.com/apple/swift/tree/main/tools/SourceKit (besucht am 20. 12. 2020).

I Project Setup

```
# Alamofire
sourcekitten doc > ../../input/alamofire.json
# ShadowsocksX-NG
pod install
sourcekitten doc -- -workspace ShadowsocksX-NG.xcworkspace \
    -scheme ShadowsocksX-NG > ../../input/shadowsocksX-NG.json
# iina
pod install
sourcekitten doc > ../../input/iina.json
# SwiftyJSON
sourcekitten doc -- -workspace SwiftyJSON.xcworkspace \
    -scheme "SwiftyJSON iOS" > ../../input/swifty-json.json
# ReactiveCocoa
carthage bootstrap --platform ios
sourcekitten doc -- -workspace ReactiveCocoa.xcworkspace \
    -scheme ReactiveSwift-iOS > ../../input/reactive-cocoa.json
# Vapor
swift package generate-xcodeproj
sourcekitten doc > ../../input/vapor.json
# Hero
sourcekitten doc -- -workspace Hero.xcworkspace -scheme Hero \
    > ../../input/hero.json
# RsSwift
sourcekitten doc -- -workspace Rx.xcworkspace -scheme RxSwift \
    > ../../input/RxSwift.json
# Kingfisher
sourcekitten doc -- -workspace Kingfisher.xcworkspace -scheme Kingfisher \
    > "../../input/kingfisher.json"
# SnapKit
sourcekitten doc -- -workspace SnapKit.xcworkspace -scheme SnapKit \
    > ../../input/snapkit.json
```