Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Refactoring Prozess einer mobilen Applikation basierend auf Clean Architecture Prinzipien

Anh Dung Nguyen
Matrikelnummer: 5094546
anhdun95@fu-berlin.de

Betreuer/in: Prof. Dr. Lutz Prechelt
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr.-Ing. Volker Roth

Berlin, 29. Juli 2022

Zusammenfassung

Die GitJournal Applikation ist ein open-source Projekt und eine plattformübergreifende Anwendung. Es dient zum Bearbeiten von Notizen und Synchronisierung mit einem Git Plattform. Die Anwendung ist mit der Zeit und verschiedene Mitwirkenden schnell gewachsen und wurde auch komplexer. Es gab keine klar definierte Softwarearchitektur für das App, welches dazu führt, dass mit der zunehmenden Komplexität, auch die Schwierigkeitsgrades erhöht, neue Funktionen und Notizenformatte einzubauen. Um dieser Komplexität eine Struktur zu verleihen, wird das GitJournal App mit der Hilfe von den klassischen Refaktorisierungsmethoden und der Prinzipien der "Clean Architecture" refaktorisiert.

Freie Universität Berlin

Fachbereich Mathematik, Informatik und Physik

SELBSTSTÄNDIGKEITSERKLÄRUNG

Name: NGUYEN	(BITTE nur Block- oder Maschinenschrift verwenden.)
Vorname(n): ANH DUNG	
Studiengang: INFORMATIK BACHELOR	
Matr. Nr.: 5094546	

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 29.07.2022 Unterschrift: A. Ng

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung & Motivation

In dem Bereich-Softwareingenieur wird viel über Software Refaktorisierung, während der Entwicklung und Wartung gesprochen. Man findet vergleichsweise wenig über Architektur Refaktorisierung. Wenn man über Software Architektur redet, dann meistens als etwas, was vor der Entwicklung entschieden werden muss. Das führt oftmals dazu, dass während der Entwicklung technische Schulden entstehen, wie bei dem Ergebnis dieser Umfrage deutlich zeigt:

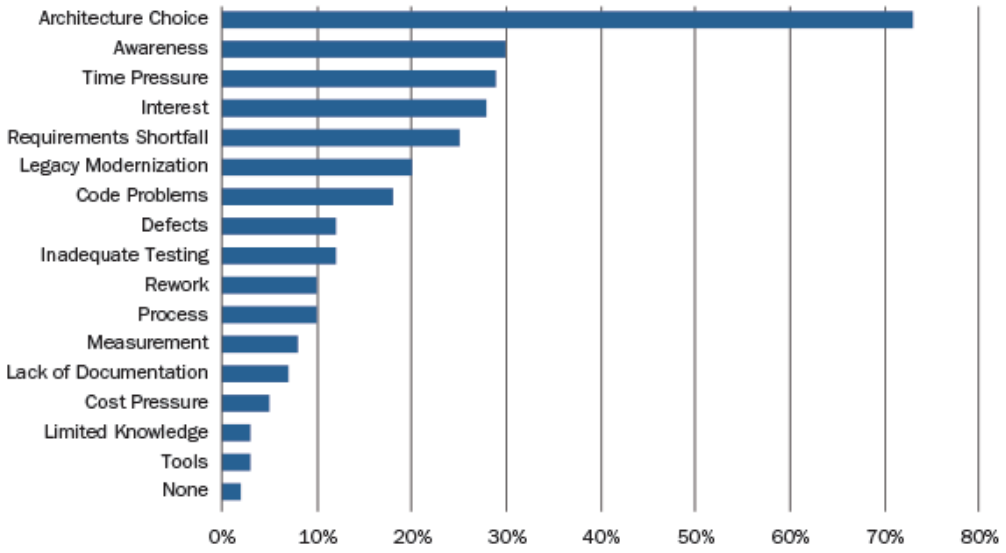

Abbildung 1: Umfrage: Gründe für technischen Schulden [1]

In diese Arbeit wird versucht, die beiden Themen zusammenzubringen und der Frage nachgegangen, ob man eine Softwarearchitektur während der Entwicklung, mit den Prinzipien und Techniken, die bereits bei Refaktorisierung bekannt sind, zu nutzen, um die Architektur zu verändern. Die Architektur, die in dieser Arbeit verwendet wird, ist die Clean Architecture von Robert C. Martin. Die Prinzipien der Clean Architecture und die „Code Smells“ der Refaktorisierung von Martin Fowler werden als Leitfaden für den Refaktorisierung Prozess benutzt.

1.2 Refaktorisierung nach Martin Fowler

Refactoring, (auch Refaktorisierung oder Restrukturierung), bezeichnet das Umgestalten eines Computerprogramms um dieses zu verbessern. Das Konzept der Refaktorisierung wurde durch Martin Fowler zusammen mit Kent Beck in dem Buch „Refactoring - Improving the Design of Existing Code“, welches im Jahr 2000 zum ersten Mal erschienen ist, der breiten Masse der Software Gemeinschaft verbreitet.

In dem Buch beschreibt Martin Fowler Refaktorisierung als einen Prozess, der „ein Softwaresystem so zu modifizieren“ [3]. Dabei soll das Verhalten und die Funktionalität des Systems nicht verändert werden, sondern nur die interne Struktur des Codes. Dadurch sollte das Softwaredesign während der Entwicklung verbessert werden. Dabei betont Martin Fowler die Art der Veränderungen, die bei diesen Prozess durchgeführt werden sollen. Die Veränderungen sollten nämlich klein und einfache Schritte sein und zwar so klein wie möglich. Diese kleinen Veränderungen werden einen akkumulierten Effekt haben und dadurch das Design des gesamten Systems verbessern.

Eine Schwierigkeit bei dem Refactoring ist, zu identifizieren, welche Stelle der Software verändert werden soll. Und genau hier bieten die sogenannten „Code-Smells“ (Schlechte Gerüche im Code) eine Abhilfe. „Code-Smells“ sind Arten von Programmteilen, die problematisch sein können, hier liegt die Betonung auf „können“, das heißt, dass diese nicht unbedingt schlecht sind. Die „Code-Smells“ gibt dem Softwareentwickler ein Signal, welche Stelle der Software genauer untersucht werden soll. Einen Katalog an „Code-Smells“ findet man in dem Buch und auch auf der Website des Buches, beschrieben [1].

Nach dem Identifizieren der Stelle im Code, welche problematisch ist, kann man diese umstrukturieren und die Stelle verbessern. Dafür hat Martin Fowler eine Reihe von Refaktorisierungs Techniken in seinem Buch beschrieben, welche auch auf seiner Website nachzulesen sind [2]. Diese Techniken sind kleine Code Veränderungen, zum Beispiel das Extrahieren einer Funktion oder einer Variable, oder das Umbenennen einer Variable. Diese Techniken sollen unterschiedlich kombiniert werden, um ein Problemteil zu verbessern.

1.3 Clean Architecture nach Robert C.Martin

In dem Buch „Clean Architecture“ [9] begründet Robert C. Martin seine Gründe für den Aufbau seiner Software Architektur, welches er wie in der Titel seines Buches, als „Clean Architecture“ bezeichnet.[3],

Als Grundbaustein seiner Argumente, erläutert Martin die sogenannte SOLID Prinzipien [9]. SOLID ist ein Akronym und steht für [4]:

- S: Das Single-Responsibility-Prinzip
- O: Das Open-Closed-Prinzip
- L: Das Liskov’sche Substitutionsprinzip
- I: Das Interface-Segregation-Prinzip
- D: Das Dependency-Inversion-Prinzip

[1] https://refactoring.guru/refactoring/smells, Zugriff am 25.07.2022

[2] https://refactoring.com/catalog/, Zugriff am 25.07.2022

[3] Anmerkung: Clean Architecture ist die Bezeichnung der Architektur. Daher werden in der Arbeit diese nicht in Deutsche übersetzt

[4] Anmerkung: Die Bezeichnungen der Prinzipien werden im Buch nicht übersetzt.

Das Single-Responsibility-Prinzip wird in der Software Industrie unterschiedlich definiert, nach Robert Martin soll ein Modul (üblicherweise eine Klasse, kann aber auch mehrere Klasse beinhalten) nur aus einen Grund verändert werden[9].

Das Open-Closed-Prinzip besagt, dass ein System nur mit neuen Funktionalitäten erweitert werden sollen, anstatt das Verhalten des Systems selbst zu verändern. Dafür muss das System seine Bestandteile sauber trennen, zum Beispiel basierend auf das Single-Responsibility-Prinzip[9].

Das Liskov'sches Substitutionsprinzip oder auch Ersetzbarkeitsprinzip besagt, dass Klassen durch seine Unterklassen ersetzbar sein sollen. [5] In der Softwarearchitektur kann dieses Prinzip oft bei Schnittstellen mit mehrere Implementierungen wieder finden.

Häufig finden wir eine Klasse A, die von eine andere Klasse B abhängig ist, welche mehr Funktionen haben, als die Klasse A benötigt. Das ist was das Interface-Segregation-Prinzip dagegen wirken möchte. Dieses Prinzip besagt, dass in eine Anwendung die Klassen nicht von Schnittstellen abhängig sein sollen, wenn diese nicht gebraucht werden.

Das letzte Prinzip in der Akronym ist das Dependency-Inversion-Prinzip und bedeutet die Umkehr der Abhängigkeiten. Diese ist sehr verbreitet in der Software Welt, dank der Verbreitung der Dependency-Injection Containern wie in „Spring", welche Entwickler die Implementierung dieser Entwurfsmuster abnimmt. (Dadurch wurde diese Prinzip in der Welt der Softwareentwicklung weniger gut verstanden) [4]

Aufbauend auf diese 5 Prinzipien beschreibt Robert C. Martin weitere 6 Komponentenprinzipien, diese sind Prinzipien für die Komponentenebene und definiert die Merkmale von Module. Diese werden in 2 Gruppen unterteilt.

Die erste Gruppe beschäftigt sich mit der Kohäsion eines System:

- Das Reuse-Release-Equivalence-Prinzip (REP)
- Das Common-Closure-Prinzip (CCP)
- Das Common-Reuse-Prinzip (CRP)

Das Reuse-Release-Prinzip besagt, dass Klassen und Module, welche wiederverwendet werden sollen, auch gemeinsam als Release zur Verfügung stehen sollen. Dieses Prinzip stellt die Einheiten mit der Wiederverwendbarkeit und des Release in Zusammenhang dar [8].

Das Common-Closure-Prinzip ist das Single-Responsibility-Prinzip auf der Komponenten Ebene. Das Prinzip besagt, dass eine Einheit nur auf einen Grund verändert werden soll. Deswegen sollen „Typen, von denen zu erwarten ist, dass sie gemeinsam geändert werden müssen, sollten in derselben physischen Einheit zusammengefasst werden" [7].

Martin erläutert, dass aufgrund der höheren Wichtigkeit der Wartbarkeit einer Anwendung gegenüber der Wiederverwendbarkeit, sollten die Teile, die öfter modifiziert werden, auch in einer Komponent zusammengefasst werden und an dieser Komponente werden dementsprechend Anpassungen vorgenommen, anstatt mehrere Änderungen in mehrere Komponenten durchzuführen[9].

Das Common-Reuse Prinzip repräsentiert auf der Komponentenebene das „Interface-Segregation“ Prinzip von den SOLID Prinzipien. Laut dieses Prinzips sollen Module und Klasse in einer Komponente zusammengeführt werden, wenn diese „im Allgemeinen gemeinsam wiederverwendet werden“ [9].

Bei der zweiten Gruppe geht es um die Beziehungen der Komponente, und diese werden auch als Prinzipien der (losen) Kopplung bezeichnet:

- Das Acyclic-Dependencies Prinzip (ADP)
- Das Stable-Dependencies Prinzip oder auch das Prinzip der stabilen Abhängigkeiten (SDP)
- Das Stable-Abstractions Prinzip (SAP)

Das Acyclic-Dependencies-Prinzip oder das Prinzip der azyklischen Abhängigkeiten. Hierbei müssen die Abhängigkeiten zwischen den Komponenten einem gerichteten azyklischen Graphen entsprechend. Das heißt, die Abhängigkeitsstruktur darf keinen Zyklen haben [6]. Das Prinzip der stabilen Abhängigkeiten besagt, dass Komponente nur von stabilen Komponente abhängen sollen. Das heißt „Die Abhängigkeiten sollten in der Richtung der Stabilität verlaufen“[9].

Bei dem Stable-Abstractions Prinzip geht es um den Grad der Abstraktheit eines Komponents, dieses soll dementsprechend gleich der Grad seiner Stabilität sein. Dadurch sollte die Komponente mehr erweiterbar sein entsprechend seiner Stabilität Grad.

Zusammenfassend haben wir die SOLID Prinzipien, welche als Grundbausteine dienen und geben vor, wie die Code Struktur organisiert werden sollen. Auf eine höhere Ebene gibt es die Komponente Prinzipien für hohen Kohäsion und losen Kopplung. Diese geben eine Anleitung, wie Klassen in Module strukturiert werden sollen und auch wie die Komponente zueinander stehen sollen.

Auf Grundlage der oben genannten Prinzipien, hat Robert C. Martin seine „Clean Architectur“ begründet. Das Clean Architecture ist keine komplett neue Idee. Das Architektur basiert auf das „Hexagonal“ Architektur und auch „Onion“ Architektur und gehört zu den Kategorien der Schichten Architektur.

Die Architektur besteht aus 4 verschiedene Schichten. Im inneren ist die Schicht der Entitäten bzw. Geschäftsobjekten, welche das Kern der Software entspricht. Diese Schicht beinhaltet die Geschäftsregeln des Unternehmens bzw. die ällgemeingültige und hochschichtigsten Regeln"der Anwendung.

Als nächste Schicht sind die „Usecases“ bzw. Anwendungsfälle, welche „anwendungsspezifische Geschäftsregeln“ [3] des Programms enthält. Diese Schicht verwaltet „den Datenfluss zu und von den Entitäten“ [3].

Die dritte Schicht enthält die Schnittstellen, welche die Geschäftsregeln (Business-Logik) der Anwendung mit den Treiber und Frameworks verbindet. In der äußeren Schicht sind Sachen wie Datenbanken und die grafischen Benutzerschnittstelle angesiedelt.

Eine Besonderheit bei dieser Schichten Architektur, ist die nach innen gerichtete Abhängigkeit. Äußere Schichten sind von den inneren Schichten abhängig und auch nur in diese eine Richtung.

Das primäre Ziel dieser Architektur ist Grenzen zwischen den Elemente bzw. Module zu erschaffen. Und die unterschiedlichen Stabilitäten und Zusammenhängenden Verantwortlichkeiten zu gruppieren. Dadurch sollte die Anwendung unabhängig von Technologien und Framework die Anwendung konstruiert werden, dank der Schichtentrennung. Je innerer einer Schicht ist, desto mehr stabil ist diese und desto weniger häufig werden die Module der entsprechenden Schicht verändert. Deswegen sind auch die Implementierung der Anforderungen und Anwendungsfälle in der inneren Schichten. Da diese die Regeln der Anwendung definieren. Und solche Regeln verändern im Laufe des Leben der Software nicht oft oder gar nicht. Eine „ToDo“ Anwendung, die ein neues „ToDo“ hinzufügen soll, welche dann als „erledigt“ markiert werden kann, wird im Laufe der Anwendung wahrscheinlich nicht ändern. Aber die Benutzeroberfläche könnte zum Beispiel ein neues Aussehen bei dem Hinzufügen bekommen. Solche Sachen sind deswegen in der äußeren Schicht. Diese hat den Merkmal, dass sie öfter verändert werden und daher nicht stabil sind.

Da die Abhängigkeit nach innen und nur in eine Richtung gerichtet sind, stellt sich die Frage, wie kann die innere Schicht, die äußere Schicht aufrufen. Wie kann zum Beispiel die „Usecases“ Schicht die Datenbank Schicht eine Anfrage schicken? Genau hier kommt die „Dependency Injection“ von den SOLID Prinzipien ins Spiel. Durch diesen Mechanismus kann der Fluss der Kontrolle umgekehrt werden. Diese Softwaremuster wird bei den Schnittstellen genutzt und stellt sicher, dass die Abhängigkeit immer nach innen gerichtet sind.

Das Ziel dieser Architektur ist das Software unabhängig von der Wahl der Framework, Datenbank grafische Komponente und jeglichen externen Komponenten zu machen. Dazu ist diese Architektur Testfähig.

Eine Idee, die immer wieder in dem Buch „Clean Architecture“ und auch in verschiedene Präsentation darüber zum Vorschein kommt, ist die Verschiebung von Entscheidungen. Es besagt, dass eine gute Architektur erlaubt, so viele Entscheidungen wie möglich zu einem späteren Zeitpunkt zu verlegen.

Dieses davor genannten SOLID und Komponenten-Prinzipien kann man in den „Clean Architecture“ sehr gut erkennen. Die Entscheidungen über welche Datenbank oder welche Frontend Framework genutzt werden sollen, ist nicht unveränderbar. Das Software ist so aufgebaut, dass man die Teile später ersetzen kann, falls diese nicht mehr den Anforderung der Anwendung entspricht. Beispielsweise wenn wir ein SQL Datenbank später vielleicht durch eine NoSQL Datenbank austauschen möchten, können wir diese tun, in dem wir die Schnittstellen der Anwendung mit den neuen Datenbank implementieren. Die Schnittstellen für das Lesen, Schreiben, etc. ist noch die selbe wie vorher. Das bedeutet, die innere Schicht sind von der Veränderung nicht betroffen. Diese werden immer noch die selben Schnittstellen nutzen und delegieren die genaue Implementierung an den äußeren Schicht.

1.4 Die Mobile Anwendung für die Refaktorisierung

Bei dieser Arbeit wird die mobile Anwendung „GitJournal“ für die Refaktorisierung genutzt. Die Anwendung ist in der Dart Programmiersprache geschrieben und benutzt das Flutter Framework für die Erstellung des iOS und Android Apps. Im Folgenden werde ich kurz über die einzelnen Bestandteile der Anwendung geben und

ein paar Besonderheiten bei der hier verwendete Technologie (Dart/Flutter) erklären.

1.4.1 GitJournal Applikation

Die GitJournal App ist ein Open-Source Projekt [5] und ist als Anwendung im Apple App Store [6] und Google Play Store [7] zu finden. Das heißt, die Anwendung ist in „Produktion“ und wird laufend weiter entwickelt. Die Kernfunktion der Anwendung ist das Erstellen, Bearbeiten und Löschen von Notizen, welche in verschiedenen Formen sein können. Das aktuell verfügbare Format ist Markdown und Tagebuch Eintrag, weitere Formate sind derzeit in der Entwicklung. Die Notizen sind in Ordnern strukturiert und die Anwendung bietet verschiedene Ansichten für die Ordner. Diese Notizen und Ordner sollen in einen Git Host wie GitHub, Gitlab, etc. gespeichert und synchronisiert werden.

1.4.2 Dart/Flutter

Dart ist eine Objekt-orientierte Programmiersprache, die 2009 erschienen ist und Flutter ist ein Framework, welches in 2018 von Google vorgestellt wurde. Dieses Framework erlaubt für verschiedenen Plattformen eine Benutzeroberfläche mit demselben Code zu erstellen, unter anderem auch für iOS und Android. In dieser Arbeit wird auf Flutter und Dart nicht im Detail eingegangen. Jedoch werden im Folgenden ein paar Besonderheiten erklärt, welche für die Refaktorisierung der Anwendung von Bedeutung sein kann.

- **ChangeNotifier** ist eine Elternklasse in Dart. Diese wird meist verwendet, um Benutzeroberfläche zu informieren, dass Daten verändert wurden und diese aktualisiert werden muss um die neuen Daten anzuzeigen.
- **Schnittstellen in Dart:** In Dart gibt es keine explizite Schlüsselwort oder Implementierung von Schnittstellen, wie zum Beispiel in JAVA mit „interface“. In Dart deklariert man dafür eine abstrakte Klasse, welche dann später implementiert werden sollen. In Dart gibt es das Konzept der impliziten Schnittstelle. Das bedeutet, eine Klasse kann auch als eine Schnittstelle dienen. Das ist ähnlich wie „abstract class“ in JAVA mit den Unterschied, dass implizite Schnittstellen in Dart keine abstrakte Methode haben kann.[2]
- **mixin** in Dart ermöglicht das Erweitern einer Klasse mit zusätzliche Funktionen, ohne diese vererben zu müssen. Das **mixin** ermöglicht die Wiederverwendung einer Klasse und deren Methoden.[2]

1.4.3 Ist-Zustand der Anwendung

Die Entwicklung der GitJournal App wurde am 12. Mai 2018 begonnen. Seitdem sind viele neue Funktionen und Änderungen implementiert worden. Das ist einer

[5] https://github.com/GitJournal/GitJournal
[6] https://apps.apple.com/app/gitjournal/id1466519634&utm_source=github&utm_medium=link
[7] https://play.google.com/store/apps/details?id=io.gitjournal.gitjournal&utm_source=github&utm_medium=link

der Gründe, warum die interne Struktur der Anwendung Verbesserungen benötigt. Durch den Austausch mit dem Ersteller der Anwendung, konnten ein paar Stellen in der Software entdeckt werden, welche eine Refaktorisierung benötigen:

1. Unsaubere Trennung der unterschiedlichen Notizen Formate. Aktuelle bietet die Anwendung Markdown, Yaml, Txt und Org Mode Formate an. Diese Formate sind alle zusammen unter einem Dach und sauber voneinander getrennt, was die Skalierbarkeit der Formate hemmt und es erschwert Fehler zu identifizieren.

2. Ähnlich wie die Notizen, sind die verschiedenen Ordner Strukturen nicht sauber voneinander getrennt.

3. Die Oberflächen Logik sind mit der Git und lokale Speicherung Logik vermischt. Es erschwert die Erweiterung der Applikation in andere Plattformen, welches mit Flutter möglich ist. Außerdem sind Bugs schwer zu identifizieren und zu beseitigen. Eine saubere Abgrenzung diese Teile muss dringend geschehen. Damit die Entwicklung voran kommen kann.

1.4.4 Auswahl des Projekts

Die GitJournal Applikation ist open-source und das Quellcode ist groß genug, wodurch sich eine Refaktorisierung lohnen würde. Außerdem wird die Software laufend weiterentwickelt, dadurch kann ich zum Beispiel durch die Bugs Report besser abschätzen, an welcher Stelle mehr Probleme auftauchen. Diese Merkmale bieten eine gute Grundlage für mein Refaktorisierung Experiment.

1.4.5 Ziel

Das Hauptziel diese Arbeit ist den Prozess der Refaktorisierung zu untersuchen. Dabei soll getestet werden, ob die Idee von kleinen Refactoring Schritten, welche von Fowler beschrieben wurde, auch auf der Architektur-ebene anwendbar ist. Darüber hinaus, sollte die Schwierigkeiten und potenzielle Problem bei der Refaktorisierung aufgedeckt und untersucht werden. Zum Schluss hoffe ich, dass ich durch die Arbeit Muster der Refaktorisierung identifizieren zu können und eventuell Richtlinien daraus ziehen kann.

1.5 Allgemeine Vorgehensweise

In dieser Arbeit geht es um einen Selbstversuch die Prozesse hinter einer architektonischen Refaktorisierung zu untersuchen. Dabei habe ich während der Durchführung meine Gedanken und Vorhaben in Form einer Tabelle dokumentiert. In regelmäßigen Abständen habe ich meine Überlegungen bei den aktuellen Schritten und auch bei Hindernissen aufgeschrieben.

2 Die Refaktorisierung

2.1 Umstrukturierung des GitJournal Moduls

2.1.1 Problem

Beim Betrachten der GitJournalRepo Klasse fällt auf, dass die Klasse sehr groß ist. Der Grund für die enorme Größe der Klasse ist die Verletzung der Single-Responsibility Prinzip. Die Klasse ist verantwortlich für unterschiedliche Bereiche. Die Verantwortlichkeit der Klasse kann in folgende Bereiche gruppiert werden:

- Manipulieren der Notizen
- Manipulieren der Ordnern
- Interagieren mit Git

Darüber hinaus, wird das Open-Closed-Prinzip nicht betrachtet. Wenn man zum Beispiel eine Funktion für das Exportieren der Notizen hinzufügen möchtest, werden die **GitJournalRepo** Klasse modifiziert, obwohl nicht alle Teile der Klasse davon betroffen ist.

Im Folgenden wird deswegen versucht **GitJournalRepo** Klasse in den einzelnen Bereiche zu zerlegen bis die einzelne Klasse den Single Responsibiliy Prinzip erfüllen. Außerdem ist die Klasse GitJournalRepo eine ChangeNotifier Klasse, wie in der Einführung bereits erklärt, hat diese Klasse die Fähigkeit die Benutzeroberfläche zu verändern. Durch die umfangreiche Verantwortungen der GitJournalRepo Klasse kann es öfter dazu kommen, dass Teile der grafischen Oberfläche ungewollt aktualisiert werden.

2.1.2 Vorgehensweise

Als erstes wird auf die Methoden, die für die Manipulation von Notizen zuständig sind, fokussiert. Dabei wird zuerst eine separate Klasse erstellt, damit die entsprechenden Methoden in diese Klasse bewegt werden können. Diese Klasse wird **NoteUsecases** benannt.

Als erstes wird die Die Methode **addNote** betrachtet und refaktorisiert. Diese hat zwei Funktionen. Zum einen wird hier das Notiz lokal gespeichert und zum anderen wird hier auf Git einen Commit gemacht und dann gespeichert. Daher habe ich entschieden, diese Methode auf zu splitten. Eine Methode ist für die lokale Speicherung zuständig und die andere für die Interaktion mit Git. Allerdings gibt es noch eine Besonderheit bei der Methode. Hier wird die ganze Funktionalität innerhalb ein Lock durchgeführt, welches verhindert werden soll, dass mehrere Git Commit gleichzeitig durchgeführt werden. Dieses Lock Instanz wird nicht nur von der Methode **addNote** benutzt, sondern auch anderen Operationen für Notizen verwendet. Da die Methode auf gesplittet und dann in andere Klasse bewegt werden sollen, hatte ich die Idee dieses Lock Instanz als globale Variable zu extrahieren. Dadurch kann mehrere Klasse auf den Lock Instanz für Git zugreifen.

Da es mehrere Schlüssel (Lock) Instanzen bei der GitJournalRepo Klasse gibt, habe ich diese zusammen in eine „Singleton“ Klasse gepackt, um die mehrmalige Initialisierung der Schlüssel zu vermeiden. Die Vorgehensweise hat die Nachteil, dass die Klasse immer da ist, auch wenn sie nicht gebraucht wird.

Die beiden Teile der Methode konnte problemlos extrahiert werden. Jedoch ist die Git Methode als auch das Aktualisieren der Benutzeroberfläche und das Hochzählen der Tracking Variable, welche die Anzahl der Git Veränderungen speichert, in ein Schlüsselmechanismus umgeben. Bei der Schlüssel handelt es sich um einen Lock Mechanismus, der den kritischen Abschnitt beschützt werden soll.

Die Methode *notifyListener*, welche für das Aktualisieren der Benutzeroberfläche zuständig ist, kann nur innerhalb der GitJournalRepo Klasse aufgerufen werden, da die GitJournalRepo Klasse diese Funktionalität von der Klasse *ChangeNotifier* vererbt und die neue Klasse **JournalUsecases** nicht. Das erschwert das Bewegen der **addNote** Methode in **JournalUsecases** Klasse. Eine Möglichkeit ist, diese Methode von der neue Klasse aus aufzurufen. Der Nachteil hier ist die Kopplung mit der GitJournalRepo Klasse. Eine zweite Möglichkeit ist, das Aufrufen der Methode, welche für die Aktualisierung der Benutzeroberfläche zuständig ist, nicht innerhalb der Git Schlüssel aufzurufen, sondern nach dem die Methode in der Schlüssel fertig ausgeführt ist.

Auf Grund der Einzel-Thread der Programmiersprache Dart, gibt es eigentlich keinen wirklichen Lock Mechanismus. Die hier benuzte Lock für die Git Operation kann durch einen async/await Mechanismus ersetzt werde. Außerdem ist die **notifyListeners** Methode nicht asynchron. Daher habe ich mich entschieden, die notifyListeners nach der Ausführung der Git Methoden auszurufen. Das erleichtert das Extrahieren des Code Block, weil die Grenze zwischen die verschiedenen Verantwortlichkeiten der Methode **addNote** deutlicher geworden ist. Damit kann ich die Methode in einzelne kleinere Methoden extrahieren, nämlich in diese Teilmethoden:

1. Notiz in das lokale Speicher zu schreiben
2. Notiz in Git hinzufügen und einen Git Commit auszuführen
3. Die Nummer der Veränderung hochzuzählen
4. Die Methode zur Aktualisierung der Benutzeröberfläche auszurufen
5. Am Ende wird zusätzlich die Methode **syncNote** in *unawaited* Block synchron ausgeführt

Die neue Klasse **NoteUsecases** hat eine Abhängigkeit mit der Klasse **GitNoteRepository**, welche die benötigte Git Methode enthält. Außerdem benötigt die Klasse für die lokale Speicherung noch **NotesCache** und **NotesFolderFS**. Nach dem Extrahieren und Bewegen in neue Klasse, sieht die Methoden Aufruf wie folgt aus:

Die Methode **updateNote** ist hat die selben Struktur wie die Methode **addNote**. Demzufolge konnte ich wie oben beschrieben die gleichen Schritte der Refaktorisierung durchführen.

Die nächste Methode, **saveNoteToDisk**, ist etwas einfacher! Wie der Name schon sagt, ist diese Methode nur für die lokale Speicherung der Notiz zuständig und hat auch keine direkte Abhängigkeiten oder Git-Schlüssel. Diese kann problemlos in die

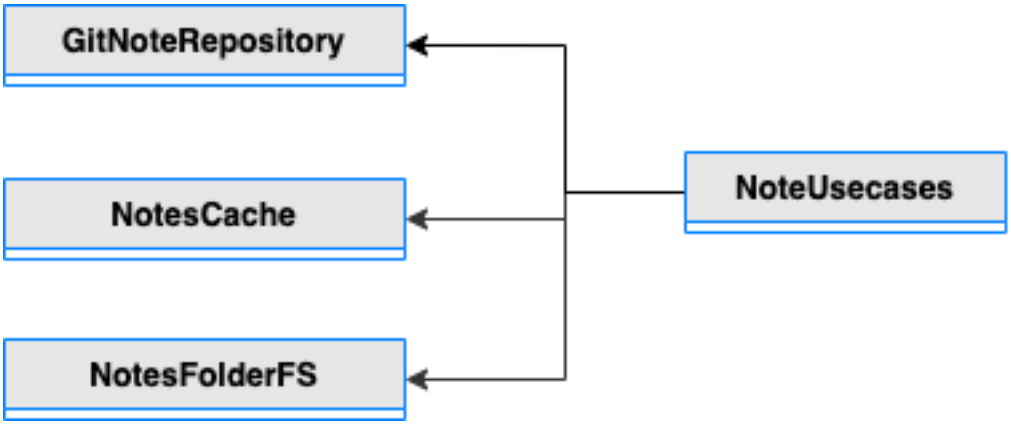

Abbildung 2: NoteUsecases Klasse

NoteUsecases bewegt werden. Nach dem Bewegen der **saveNoteToDisk** fällt mir auf, dass andere Klasse wie ober beschrieben, auch die Operation lokales Schreiben benötigt. Daher konnte ich hier das Teil der lokale Speicherung bei der Methode wie **addNote** durch die **saveNoteToDisk** ersetzen und dadurch Code Duplikation vermeiden. Darüber hinaus bietet sich hier an, den Name der Methode zu verändern, um einheitlich mit den restlichen Schema zu machen. Die Methode **saveNoteToDisk** wird auf **saveNoteToStorage** benannt, da hier die Methode NoteStorage.save aufgerufen wird.

Das Refaktorisieren der **renameNote** Methode funktioniert genauso wie bei der **addNote**.

Bei der Methode **moveNotes** erkenne ich die Einzel Teil Funktionen wieder wie bei der Methode **addnote** und **updateNote**. Jedoch anders als die beiden Methode, wird alle Teile innerhalb der Git-Schlüssel ausgeführt. Dieses *Inkonsistenz* hat mich etwas verwirrt. Die Frage ist, ob der Autor des Code dieses Struktur absichtlich aus einen bestimmten Grund gewählt hat, oder würde die Struktur wie bei **addNote** Methode auch genauso gut funktionieren? Wenn ich die **removeNotes** betrachte, dann sehe ich, dass diese genauso strukturiert ist wie die **moveNotes** Methode. Ich vermute, dass der Autor diese beide Methoden so strukturiert hat, weil es hier um Methode handelt, welche mehrere Notizen bearbeitet. Aber meine Theorie ist, dass ein Struktur wie bei der **addNote** Methode auch genauso gut funktionieren würde.

Ich habe die Struktur des ursprünglichen Codes für die Refaktorisierung beibehalten. Die Struktur ist anders als bei der Methode **addNote** und sieht wie folgt aus: - Methoden Aufruf –> Git-Schlüssel mit Veränderung der Notizen auf lokalen Speicher und mit dem Git System –> Zum Schluss wird beim Erfolgreichen Ausführung die Veränderung Zähler inkrementiert und die Benutzeroberfläche aktualisiert. Wie oben schon erwähnt, ist die Methode **removeNotes** gleich strukturiert wie bei der **moveNotes** Methode und kann daher ähnlich refaktorisiert werden.

Das Entfernen einer Notiz kann mit der Methode **undoRemoveNote** rückgängig gemacht werden. Auch hier werden lokale als auch Git Operationen innerhalb der Git-Schlüssel ausgeführt und kann ähnlich wie oben extrahiert werden.

Damit habe ich alle Funktionalitäten der Notizen Manipulation mit zwei einfachen Refaktorisierung Mechanismus refaktorisiert, nämlich **extract Method** und **mo-**

ve Method Was mich dabei noch stört, ist den Aufruf der Methode zur Aktualisierung der Benutzeroberfläche (*notifyListeners*). Diese war ursprünglich innerhalb der Git-Schlüssel, aber aufgrund des Bewegen der Methoden in eine andere Klasse kann diese nur nach der Git-Schlüssel ausgeführt werden und nicht mehr innerhalb.

Eine Lösung dafür wäre die Git-Schlüssel nicht mit in die neue Klasse zu übertragen, sondern bei der **GitJournalRepo** Klasse zu belassen. Dann sollen die einzelne Teil Methoden Einzel aufgerufen werden, anstatt einen Methodenaufruf welches alles dann erledigt und das Ergebnis zurück ermittelt.

Wenn man die Methode **addNote** wieder als Beispiel nimmt, dann wird die Methode bei diese Refaktorisierung so aussehen:

- Rufe die Methode zum Hinzufügen des Notiz in den lokalen Speicher auf, diese Methode kann über das Objektklasse **NoteUsecases** aufgerufen werden
- Eintreten der kritischen Abschnitt
- Rufe die Methode zum Hinzufügen in das Git System, auch über die **NoteUsecases** Objektklasse, auf
- Inkrementiere den Veränderung Zähler
- Rufe die **notifyListener** Methode auf, um die Benutzeroberfläche zu ändern
- Verlasse den kritischen Abschnitt
- Gebe das hinzugefügte Notiz zurück

Diese Vorgehensweise hat der Vorteil, dass die Folge des Methodenaufrufs wie bei den ursprünglichen Code beibehalten werden kann. Und damit laufe ich nicht den Gefahr, etwas zu verändern, was nicht Vorgesehen war.

Obwohl bei der ersten Vorgehensweise ich relativ sicher bin, dass das Programm noch ordnungsgemäß funktioniert, da die Test Fälle nach der Refaktorisierung noch durchgegangen sind. Aber die zweite Vorgehensweise ist meine Meinung nach trotzdem sicherer, weil hier die Logik der Schlüssel und den kritischen Abschnitt nicht verändert wird.

Der Vorteil der zweite Vorgehensweise wird deutlich wenn man a die Refaktorisierung der Methode **moveNotes** anwendet. Die Struktur sieht wie folgt aus:

- Eintreten der kritischen Abschnitt
- Rufe über der Klasse **NoteUsecases** die Methode zum Bewegen der Notizen in den lokalen Speicher auf
- Rufe die Methode zum Bewegen in das Git System, auch über die **NoteUsecases** Objektklasse, auf
- Inkrementiere den Veränderung Zähler
- Rufe die **notifyListener** Methode auf, um die Benutzeroberfläche zu ändern
- Verlasse den kritischen Abschnitt

- Gebe eine List der Notizen mit den neuen Dateistruktur zurück

Nach dem Refaktorisierung sieht die Klassen so aus:

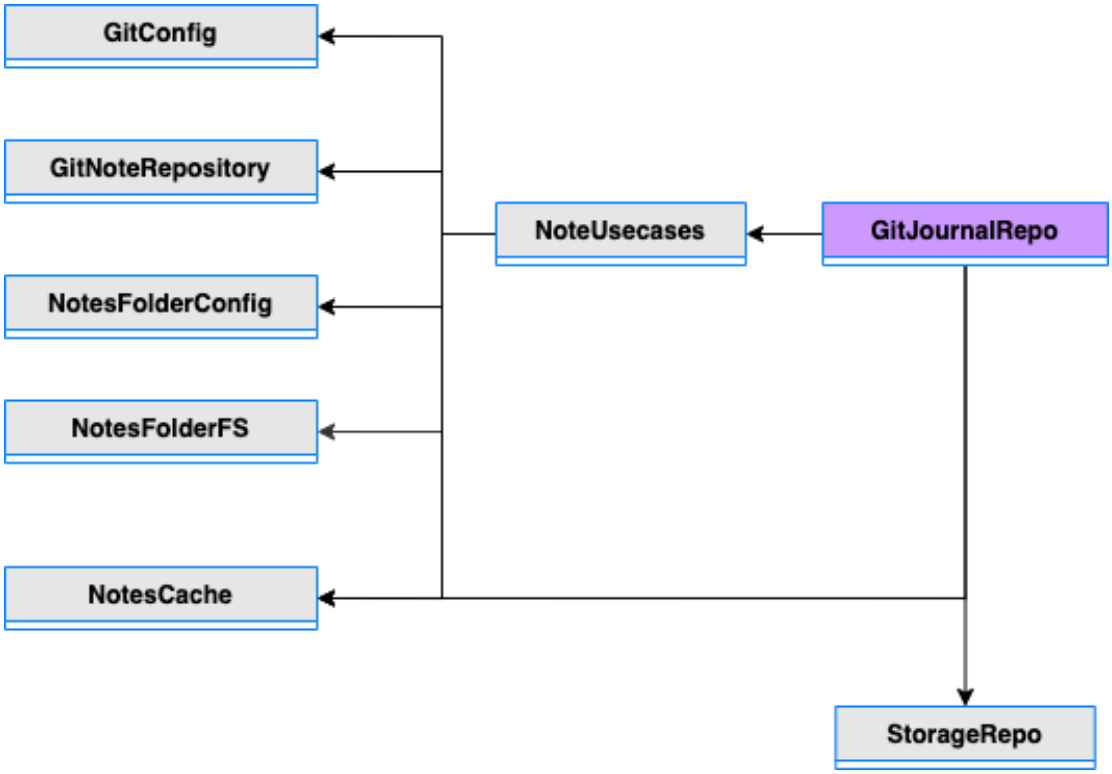

Abbildung 3: NoteUsecases Klasse nach der Refaktorisierung

An diese Stelle sind auch die Abhängigkeiten mit der Klasse*NotesCache* in der GitJournalRepo Klasse nicht mehr vorhanden. Dank das Bewegen der Notizen Methoden in eine separate Klasse. Daher kann diese von der GitJournalRepo auch entfernt werden.

Relativ ähnlich aufgebaut, sind auch die Operationen für die Ordner. Auch hier gibt es Inkonsistenzen bezüglich der Struktur der Methoden wie bei den Notizen. Daher habe ich gleich die zweite Vorgehensweise wie oben beschrieben angewendet. Die Git-Schlüssel bleiben bei der GitJournalRepo Klasse und die Teilfunktionen der Ordner werden extrahiert und in eine neue Klasse bewegt. Nach den gleichen Schema wie bei der Refaktorisierung der Notizen, habe ich die neue Klasse "FolderUsecases"benannt.

Zur Veranschaulichung, werde ich im Folgenden die Refaktorisierung der Methode **createFolder** beschreiben.

Betrachtet man es mal die Struktur der Methode, dann fällt gleich auf, dass die Teilfunktionen innerhalb der Git-Schlüssel aufgeführt werden. Die genaue Struktur sieht so aus:

- Eintreten der kritischen Abschnitt

- Erstelle ein neues **NotesFolderFS** Objekt und rufe **create** auf um die Ordner in der lokale Speicher zu erstellen
- Füge einen Referenz mit dem Eltern Ordner Objekt
- Rufe die Git-Klasse auf und füge die neu erstellte Ordner in das Git System ein
- Inkrementiere den Veränderung Zähler
- Rufe die **notifyListener** Methode auf, um die Benutzeroberfläche zu ändern
- Verlasse den kritischen Abschnitt
- Lade alles neu mit der **syncNotes** Methode

Hier kann man klar die lokalen Operationen und die Git Methode jeweils eine Sub Klasse extrahieren. Außerdem habe ich erst bei diese Refaktorisierung gemerkt, dass das Hochzählen der Veränderung Variable und das Aktualisieren der Benutzeroberfläche immer wieder auftaucht, bei der Notizen Methoden, als auch jetzt bei den Ordner Methoden. Daher habe ich diese beide in eine neue Methode mit der Name *notifyNewChange* extrahiert und dann rückwirkend bei den oben benannten Notizen Funktion ersetzt, bis auf die Methode **undeoRemoveNote** welche die Änderung rückgängig macht und anstatt den Zähler zu inkrementieren, muss bei diese Methode den Zähler dekrementiert werden. Daher habe ich das Dekrementieren und das Aktualisieren der Benutzeroberfläche in eine neue Methode, namens **undoPreviousChange**, extrahiert. Dadurch vermeidet man den Duplizierten Code und es erhöht auch die Leserlichkeit.

Nun, nach den extrahieren der einzelne Teilmethoden der **createFolder** Funktion, kann ich diese in die Klasse **FolderUsecases** bewegen und die Methodenaufruf sieht wie folgt aus:

- Eintreten der kritischen Abschnitt
- Rufe über die **FolderUsecases** Klasse die Methode zur Erstellen der Ordner auf den lokalen Speicher auf, welche auch einen Referenz mit dem Eltern Ordner Objekt hinzufügt
- Rufe über die **FolderUsecases** Klasse die Methode zur Erstellen der Ordner im Git-System auf
- Rufe **notifyNewChange** auf, welche den Zähler inkrementiert und die Benutzeroberfläche aktualisiert
- Verlasse den kritischen Abschnitt
- Lade alles neu mit der **syncNotes** Methode

Die **createFolder** sieht nach der Refaktorisierung nicht nur kompakter aus, sondern trennt auch schön von der Implementierung Details der einzelnen Funktionalitäten. Alle andere Ordner Methoden können wie **createFolder** refaktorisiert werden. Bei der **renameFolder** Methode bin ich allerdings auf eine Schwierigkeit gestoßen.

Der Test für den Methode konnte nicht erfolgreich durchgelaufen werden. Daraufhin habe ich meine Refaktorisierung rückgängig gemacht und bei der orginale Version ist der Test durchgelaufen, also müsste irgendwas bei der Umstrukturierung schief gelaufen sein. Beim genauen Betrachten und Durchgehen der Methode ist mir aufgefallen, dass ich bei der Refaktorisierung einen Fehlerzustand falsch refaktorisiert habe. Nach diesem Vorfall habe ich erst das Nutzen von Test Code und denen Notwendigkeit bei der Refaktorisierung gesehen, weil ohne diese Test, hätte ich nicht gemerkt, dass ich diese Fehlerzustand falsch behandelt habe.

Nach dem alle Ordner Methoden extrahiert und in eine neue Klasse neu geordnet sind, kann ich zu den letzten Teil Bereiche der **GitJournalRepo** Klasse fokusieren.

Hier bietet sich auch gut an, eine neue Klasse für die Git-Methoden zu erstellen. Ich habe diese **GitUsecases** benannt, um den Schema gleich zu bleiben. Anders als bei den Notizen und Ordner Methoden, haben die Git-Methoden keine direkte Abhängigkeiten mit der **GitJournalRepo** Klasse. Das erleichtert die Refaktorisierung sehr, da es sich hierbei um den sogenannten Vermittler „Code Smell“ [3] handelt. Der Aufrufer ruft die Git-Methoden über die Klasse, spricht ich könnte die ganzen Git-Methoden direkt in die neue Klasse bewegen und bei den Aufrufer der Methode die Klassenobjekt ändern. Als Beispiel nehme ich im Folgenden die Methode **branches**, welche die Git Zweige laden und zurückgeben sollte. Diese Methode ruft am Anfang eine static Methode von der Klasse **GitAsyncRepository** auf, welche das Repository laden soll. Danach werden die lokalen Zweige und remote Zweige zusammengefügt und in eine Liste zurückgegeben. Die Methode **branches** ist abgekapselt von der **GitJournalRepo** Klasse. Wenn man diese in eine andere Klasse bewegt, gibt es daher auch keinen Fehler bei der **GitJournalRepo** Klasse. Gleicher Fall ist es auch bei den restlichen Git Methoden. Außer bei der Methoden **checkoutBranch** und **resetHard**. Bei diesen Methoden werden die Notizen und Ordner neu geladen. Daher muss hier nach dem Bewegen, die Methode **loadNotes** von der **GitJournalRepo** Klasse aufgerufen werden. Damit besteht weiterhin eine Abhängigkeit mit der **GitJournalRepo** Klasse. Wenn man die **checkoutBranch** in der gesammten Anwendung anschaut, dann kommt diese Methode nicht so häufig auf. Um die Klassen und deren Funktionalitäten besser zu trennen, habe ich entschieden, eine Methode bei der **GitJournalRepo** zu erstellen, welche diesen Fall behandelt. Das bedeutet, wenn der User einen neuen Branch wechselt, muss er auch eine Methode von der **GitJournalRepo** Klasse aufrufen, welche die entsprechenden Notizen des neuen Branch neu lädt. Zum Schluss habe ich die Name der "GitJournalRepo"Klasse verändert, da diese nicht die richtige Bezeichnung für die Funktionalitäten dieser Klasse mehr entsprechen. In der Klasse werden nun hauptsächlich die Funktionen der Usecases Klasse ausführt und die grafische Komponente benachrichtigt, zu aktualisieren, hat diese Klasse mehr die Funktion Controller/Presenter. Deswegen habe ich diese umbenannt auf "GitJournalPresenter".

Nach dem Refaktorisieren der drei Teilbereiche der **GitJournalRepo** Klasse, sieht das UML Diagramm wie folgt aus:

2.1.3 Fazit

Bei dieser Refaktorisierung hatte ich Schwierigkeiten am Anfang die Struktur und die einzelnen Verantwortlichkeiten einer Methode, beispielsweise die Notizen Metho-

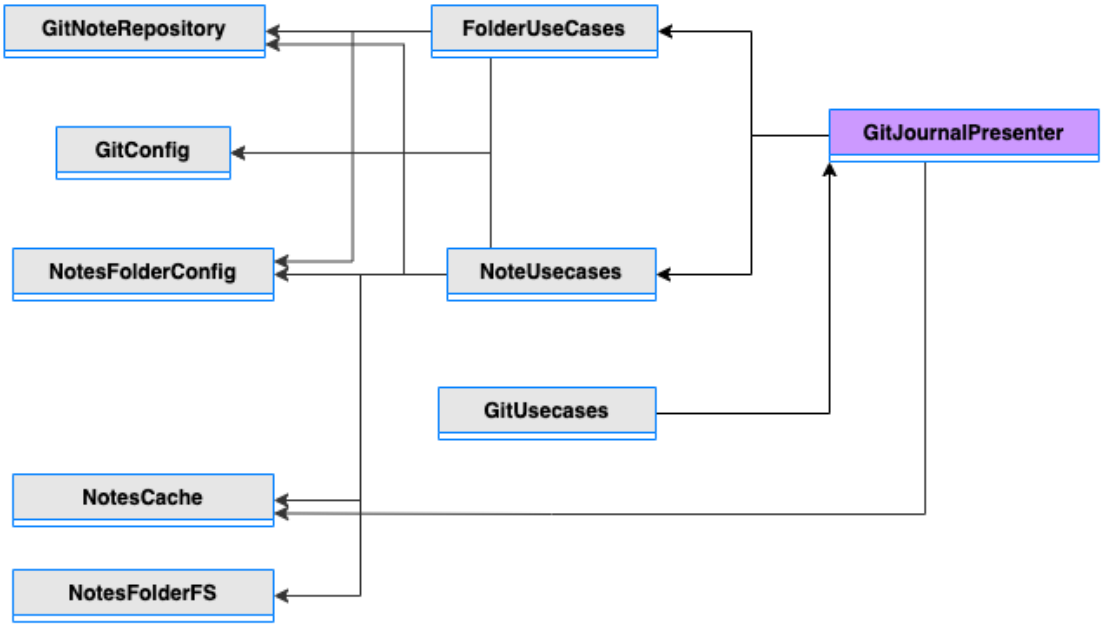

Abbildung 4: Alle Usecases-Klasse nach der Refaktorisierung

den. Auch Besonderheiten der Methoden Struktur, wie das Schloss für den kritischen Abschnitt. Besonders, weil es hier keine einfache Möglichkeit gab, die Funktionsfähigkeit bei der Refaktorisierung zu garantieren. Es gab keinen spezifischen Test für den kritischen Abschnitt Mechanismus und das gab mir kein sicheres Gefühl bei der Refaktorisierung und deswegen habe ich mich bei den unterschiedlichen Möglichkeiten der Refaktorisierung, am Ende doch für einen relativ sichereren Weg entschieden, wo die Struktur des Codes nicht beeinflusst wird.

Nach dem ich ein paar Methoden der Notizen in einzelne Teile zerlegt und extrahiert habe, habe ich diese in eine neue Klasse gruppiert. Dies hat den Rest sehr vereinfacht. Die Methoden für Ordner sind auch ähnlich aufgebaut und daher konnte ich auf die gleiche Art und Weise diesen refaktorisieren. Nach dem ich den Mechanismus herausgefunden habe, konnte ich im NormalFall eine Methode unter 10 Minuten refaktorisieren und mit den vorhandenen Test Fällen überprüfen. Natürlich gab es auch ein paar Ausnahmen, wie zum Beispiel wenn die Struktur der Methode nicht dem Rest entspricht und ich nicht sicher war, ob diese Struktur absichtlich so gemacht ist!

Betrachtet man das obige UML-Diagramm (Abbildung 4), dann fällt auf, dass die **GitJournalPresenter** Klasse. Nun nicht direkt die Logik der Funktionen aufruft, sondern über die Üsecases"Klassen. Auch die Git Methoden würden aus der Klasse extrahiert. Die üblich gebliebene Aufgabe der **GitJournalPresenter** Klasse ist das Aktualisieren der Benutzeroberfläche. Durch die Refaktorisierung ist die Verantwortlichkeiten der Klasse kleiner geworden, aber die Klasse übernehme noch die Aufgabe zum Aufrufen der Notizen als auch die Ordner Funktionen und dadurch das Aktualisieren der grafischen Oberfläche für diese beiden Bereiche. Das bedeutet, das Single-Responsibility Prinzip ist hier noch nicht ganz erfüllt.

Außerdem werden nun die **GitJournalPresenter** Klasse nicht mehr abhängig sein

von den Implementierungsklassen für die lokale Speicherung und für die Git Interaktionen. Stattdessen rufen die Usecases Klassen diese auf und sind direkt abhängig. Das ist nicht ideal, da diese Klassen nicht die geschäftliche Anwendungsfälle enthält und mehr Anfällig von Modifikationen sind im Gegensatz zu den Usecases. Nach dem Stable-Dependency Prinzip sollte das andersherum sein, **GitNoteRepository** sollte zum Beispiel von **FolderUseCases** abhängig sein.

Ein positiver Aspekt ist die Aufteilung der Bestandteile. Wie auf dem UML-Diagramm zu erkennen ist, gibt es 3 Schichten. Die Use-Cases in der Mitte, die Presenter und Benutzeroberfläche sind auf der Rechten Seite und die Klassen für Git und lokale sind auf der linken Seite. Die Presenter, die Code für Speicherung und die externen Schnittstellen sind in der äußeren Schicht um den Use-Case.

2.2 Umstrukturierung des GitJournal ChangeNotfier Moduls

2.2.1 Problem

Die Klasse **GitJournalRepo** hat, wie bei den obigen Kapitel schon beschrieben, viele Verantwortlichkeiten. Ich habe bereits die logischen Teile, welche die Anwendungsfälle enthält, refaktorisiert, aber die Klasse ist auch gleichzeitig verantwortlich für das Aktualisieren von mehreren Benutzeroberflächen. Hier wird nicht gezielt die Benutzeroberfläche aktualisiert, sondern sobald eine Veränderung auftritt, egal ob die Veränderung wegen Notizen oder Ordner verursacht wurde, werden die grafischen Elemente neu geladen. Auch wenn das Framework Flutter effizient ist, kann solche unnötige Aktualisierung der Oberfläche großen Einfluss auf die Leistung der Applikationen haben. Selbst wenn man im nach hinein versucht, die Elemente der Benutzeroberfläche zu optimieren, wie zum Beispiel durch das Zwischenspeichern, ist es schwierig und nicht effizient. Besser wäre es die Struktur der Anwendung zu refaktorisieren und so umzustrukturieren, dass nur Elemente neu geladen werden, welche auch tatsächlich neu geladen werden müssen.

2.2.2 Vorgehensweise

Im Folgenden wird versucht, das Aktualisieren der Benutzeroberfläche zu begrenzen. Die folgenden Refaktorisierung sind erst mal nicht mit den Refaktorisierungen im vorherigen Teil verbunden, sondern wird getrennt behandelt und später zusammengefügt. Das bedeutet, dass die Refaktorisierung vom Ist-Zustand aus, durchgeführt wird.

Schnell aufgefallen sind hier die Git-Schlüssel, welche den kritischen Abschnitt der Git Methoden beschützt. Aus der Erfahrung der vorherigen Refaktorisierung, kann ich diese Variable extrahieren und in eine separate Klasse bewegen. Um sicherzustellen, dass die Variable nur einmal initialisiert wird, muss die Objektklasse als Singleton umgewandelt werden. Von da an ist die Refaktorisierung relativ einfach. Als Beispiel werde ich im Folgenden die Refaktorisierung der Methode **addNote** beschreiben. Für die Refaktorisierung musste ich die komplette Methode extrahieren, dann in die neue Klasse, welche als **JournalNote** bezeichnet wird, verschieben. In der Methode fehlt die Referenz zur Klasse **GitNoteRepository**, die für Git Operationen zuständig ist. Deswegen habe ich diese in dem Konstruktor der **JournalNote** Klasse referenziert.

Ein Problem, das hier entstanden ist, sind die 3 Methoden nach der Operation zum Hinzufügen der Notizen. Diese sind das erhöhen der Zähler für Änderungen, das Aktualisieren der Benutzeroberfläche und das Neuladen der Notizen. Diese Methoden sind abhängig von der ursprünglichen Klasse **GitJournalRepo**. Für das Erste habe ich entschieden, eine Referenz der Klasse **GitJournalRepo** mit in die neue Klasse zu nehmen und dann die 3 oben genannte Methode von der ursprüngliche Klasse **GitJournalRepo** aufrufen zu lassen. Damit konnte ich den Rest der Methoden refaktorisieren, mit dem Problem, das die noch Abhängigkeit mit der ursprüngliche Klasse haben.

Nach dem Verschieben der Methoden für Notizen in die **JournalNote** Klasse und die Methoden für Ordner in die **JournalFolder** Klasse, ist mir aufgefallen, dass das Aktualisieren der Benutzeroberfläche, sprich das Aufrufen der **notifyListener** Methode, nicht unbedingt von der **GitJournalRepo** Klasse aufgerufen werden muss. Ich kann die **JournalNote** und **JournalFolder** Klasse jeweils von der **ChangeNotifier** Klasse vererben und habe dadurch auch den Zugriff auf die Methode **notifyListener**. Ich muss hier dann nur noch diese mit der Benutzeroberfläche verbinden, damit die Benutzeroberfläche neu geladen werden kann, wenn die **notifyListener** von der **JournalNote** bzw. **JournalFolder** aufgerufen wird. Dementsprechend habe ich die beiden Klassen auf **JournalFolderPresenter** und **JournalNotesPresenter** umbenannt.

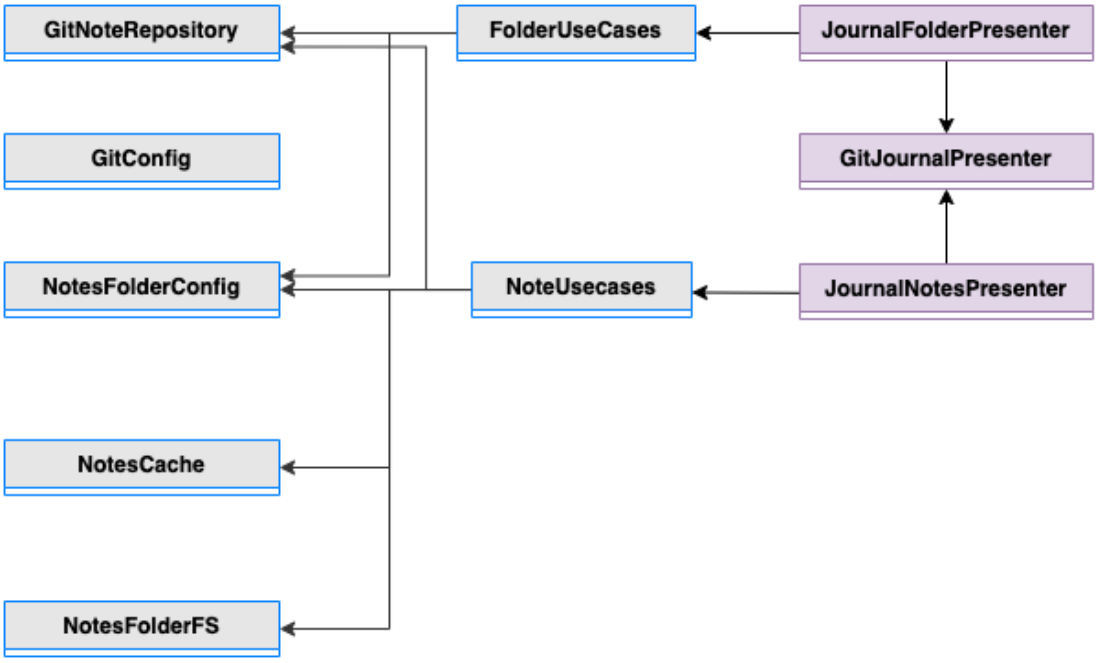

Abbildung 5: NoteUsecases Klasse nach der Refaktorisierung

Um komplett unabhängig von der **GitJournalRepo** Klasse zu sein, muss ich noch das Modifizieren der Änderungszähler in eine separate Klasse extrahieren, damit alle Presenter Klassen auf diese zugreifen können.

Das Synchronisieren der Notizen verhindert allerdings das Vorhaben von der **GitJournalRepo** Klasse unabhängig zu sein, weil bei der Synchronisation wird „git

fetch“, „git merge“ und „git push“ aufgerufen und bei jedem dieser Schritte wird die Methode **notifyListener** zum Aktualisieren der grafischen Elemente aufgerufen. Eine Möglichkeit dafür ist diese Synchronisation von der jeweiligen Presenter Klasse selbst ausführen zu lassen. Das heißt, diese Methoden werden in den **JournalNotesPresenter** und **JournalFolderPresenter** dupliziert. Für die Unabhängigkeit, musste ich diese Duplizierung in Kauf nehmen.

2.2.3 Fazit

Nach dieser Refaktorisierung kommt dieser Code Bereich dem Single-Responsibility-Prinzip viel näher. Das Open-Closed Prinzip wird auch erhöht. Hier werden Modifikationen von bestehenden Klassen bei neuen Funktionalitäten in der Zukunft minimiert, da es hier eine klare Trennung zwischen den verschieden Verantwortlichkeiten gibt.

Bei diese Refaktorisierung fällt es mir ziemlich leicht. Es liegt zum einen daran, dass ich diese Code Bereiche durch die vorherige Refaktorisierung Prozess bereits kenne und zum anderen sind die Methoden ähnlich aufgebaut, somit sind die Schritte der Umstrukturierung auch ähnlich. Die meiste Zeit habe ich damit verbracht, die dazugehörigen Test Fälle auch in eine getrennte Testklasse zu verschieben und auftretende Fehler zu beseitigen. Außerdem ist bei der Refaktorisierung der **removeNote** Methode ein Fehler aufgetreten, welche mir nicht ersichtlich war. Bei der Untersuchung stellte sich heraus, dass es ein Fehler der Anwendung ist und nicht durch die Refaktorisierung entstanden ist.

Nichtsdestotrotz habe ich für diese Refaktorisierung unter 7 Stunden gebraucht, was relativ schnell ist im Gegensatz zu den anderen Refaktorisierungen.

2.3 Umstrukturierung der „NotesFolder“ Klassen

2.3.1 Problem

Aufgrund des nicht erweiterbaren Aufbaus der Objektklasse **NotesFolder** und der zusammenhängenden Klasse, entstand mit der Zeit der Codebasis eine verwirrende Struktur und machte das ganze schwieriger zu erweitern und potenzielle Fehler zu beseitigen.

Auf dem ersten Blick sind diese sprechen diese Klassen alle miteinander und rufen sich gegenseitig auf. Auch gibt es keine klaren Hierarchien und Strukturen. Eine Besonderheit bei der **NotesFolder** Klasse ist die Reactiveness. Die Klasse besitzen unterschiedliche Observer Methoden, welche die Benutzeroberflächen aktualisieren sollten, wenn eine Veränderung der Notizen oder Ordner durchgeführt wurde. Und da liegt auch das größte Problem bei der ganzen Konstruktionen, welche es nicht leicht macht, die zu strukturieren. Dazu sind auch nicht viele Tests Fälle für diese Klassen vorhanden, welches den Prozess der Refaktorisierung nicht sicher macht. Aus diesem Grund habe ich erst mal die Tests Fälle betrachtet und versucht diese zu erweitern. Damit ich mehr Sicherheit bei dem weiteren Refaktorisierung Prozess habe.

Wenn man die Abhängigkeiten um die NotesFolder Klasse aufzeichnet, dann sehen diese wie folgt aus:

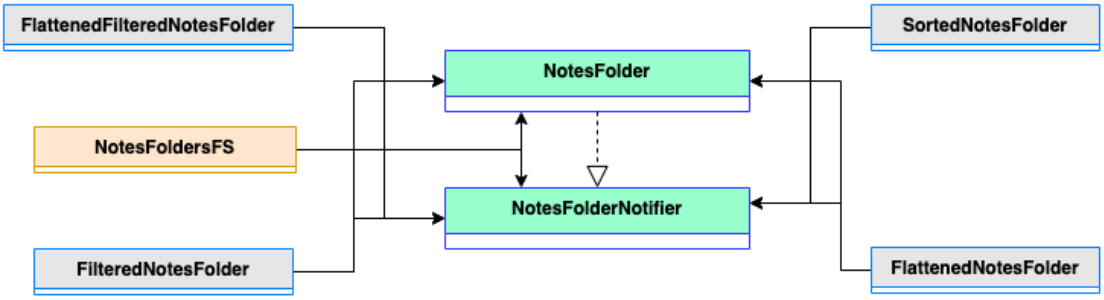

Abbildung 6: Ordnern Klassen

Dabei sollten die Klasse **NotesFolder** und **NotesFolderNotifier** als Base dienen, sodass man die verschiedenen Ordner Typen erweitern kann. Darunter gehören folgenden Ordner Klassen:

- SortedNotesFolder
- FlattenedNotesFolder
- FlattenedFilteredNotesFolder
- FilteredNotesFolder
- NotesFolderFS: wobei diese genauer betrachtet, auch eine Basis Klasse ist!

Das Problem, das ich auf dem ersten Blick sehe, war die transitive Abhängigkeit der Subklassen mit den Basis Klassen und vice versa. Beispielsweise implementiert die Klasse FlattenedNotesFolder die Schnittstelle *NotesFolder* und gleichzeitig die Klasse **NotesFolderNotifier** benutzt. Das Problem kommt, wenn die Klasse *NotesFolder* auch die Klasse **NotesFolderNotifier** implementiert. Das bedeutet, die Klasse FlattenedNotesFolder implementiert die Klasse NotesFolderNotifier durch die Schnittstelle *NotesFolder* mit der Klasse **NotesFolderNotifier**. Es ist auch gleichzeitig eine kreisförmige Abhängigkeit, denn hier implementiert die Methode der Klasse **NotesFolderNotifier** sich selbst. Eine klare Trennung zwischen dem Deklarieren und die Implementierung ist hier nicht gegeben. Das Acyclic-Dependencie-Prinzip des Clean Architectures wird hier verletzt.

Und so sehen auch die anderen Objektklassen aus. Diese Selbstimplementierung wird bei jeden *NotesFolder* Objektklasse benötigt und nicht nur bei **FlattenedNotesFolder**. Ein weiteres Problem bei diesen Konstruktion sind die nicht benutzten Methoden. Eine Klasse muss immer alle Methoden der Klasse **NotesFolderNotifier** implementieren, auch wenn diese nicht verwendet werden.

Ähnlich wie die anderen Klassen hat die Klasse **NotesFolderFS** auch die transitive Abhängigkeit, aber darüber hinaus hat die auch eine Abhängigkeit mit anderen Klassen und wird in der Anwendung oft als einen Art Base bzw. Standard Klasse für die lokale Speicherung genutzt.

2.3.2 Vorgehensweise

Im Folgenden werde ich beschreiben, wie ich diesen Bereich versucht habe zu refaktorisieren. Als ich den Versuch gestartet habe, konnte ich die Refaktorisierung nicht erfolgreich durchführen, die Funktionalität der Anwendung würde dabei stark verändert und die Veränderungen müsste zurück gesetzt werden.

Wie bei dem vorherigen Kapitel besitzen **NotesFolderNotifier** Observer, Methoden für Notizen und Ordner. Eine Möglichkeit voranzukommen, wäre erstmal diese beiden Gruppen von Methoden zu trennen.

Aber ein anderes Problem, das vorher geklärt werden soll, ist die transitive Abhängigkeit. Wie kann ich diese auflösen? Meine erste Idee war, die Observer Listen von der Klasse **NotesFolderNotifier** zu trennen. Ich habe alle Oberserver Listen in eine neue Klasse **NotesFolderObserver** bewegt und gleichzeitig eine Schnittstelle dafür implementiert. Die abstrakte Klasse **NotesFolder** kann jetzt direkt die abstrakte Klasse **NotesFolderObserver** (Interface) vererben und nicht mehr die Klasse **NotesFolderNotifier**.

Nun wenn die Klasse **FlattenedFilteredNotesfolder** die Observer Listen anwenden möchte, muss diese mit den Methoden der Klasse **NotesFolderObserverImpl** erweitert werden.

Jedoch treten Fehler auf bei den **FlattenedFilteredNotesfolder** Klasse. Das Problem liegt daran, dass die Methoden der **FlattenedFilteredNotesFolder** Klasse direkt das NotesFolder Objekt benutzt. Beispielsweise bei diese Methode:

```
addFolder(NotesFolder folder) { ... }.
```

Dadurch ruft das Objekt **NotesFolder** auch die Observer Methoden auf. Das ist auch eine Besonderheit bei der Dart Programmiersprache. Es gibt in Dart keine *Interface*, diese wird durch abstrakte Klassen implementiert. Aber abstrakte Klassen können auch für andere Zwecke verwendet werden.

Um diesen Fehler zu beseitigen, muss die abstrakte Klasse **NotesFolder** die andere abstrakte Klasse **NotesFolderObserver** auch vererben. Damit hat die Klasse **FlattenedFilteredNotesFolder** zwei abstrakte Klassen, welche als „Interface“ dienen.

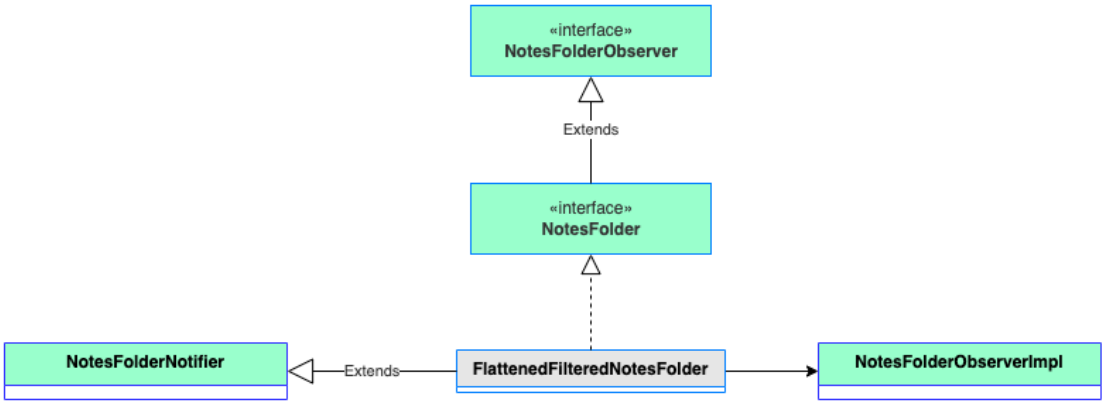

Abbildung 7: FlattenedFilteredNotesFolder Abhängigkeiten

```
class FlattenedFilteredNotesFolder extends NotesFolderNotifier
   with NotesFolderObserverImpl implement NotesFolder {
 ...
}
```

Dadurch ist die Abhängigkeit der Klassen nicht besser als am Anfang und sieht etwas schlimmer aus bei den einzelnen Subklassen wie **FlattenedFilteredNotesFolder** Der einzige Vorteil hier wäre die kleinere **NotesFolderNotifier** Klasse. Weitere Verbesserungen könnten sein, die NotesFolderObserver Klasse zwei verschieden Klassen, eine für Notizen Observer Listen und eine andere für die Ordners Observer Listen, zu verändern. Damit könnte das andere oben genannte Problem der fehlenden Trennung der unterschiedlichen Teilbereiche gelöst werden, aber das Hauptproblem, das mit der kreisförmige Abhängigkeiten der Klassen, ist immer noch nicht gelöst!

Ich habe mich dennoch entschieden, an der Stelle weiterzumachen. Mit der Hoffnung, die Probleme nach und nach auflösen zu können.

Des Weiteren habe ich die Subklassen der Ordner, also von der **NotesFolder**, wie oben beschrieben refaktorisiert, mit dem Nachteil, dass jede Subklasse die **NotesFolderNotifier** Klassen vererbt und die Methode der Klasse **NotesFolderObserverImpl** benutzt, um die abstrakte Klasse **NotesFolder** implementieren zu können.

Da alle Subklassen mit den Klassen **NotesFolderNotifier** und **NotesFolderObserverImpl** erweitert werden und darüber hinaus, hat die **NotesFolderNotifier** Klasse keine andere Arten von Unterklassen außer den Ordner Subklassen, konnte ich hier die **NotesFolderNotifier** direkt an die Methoden von der **NotesFolderObserverImpl** Klasse vererben. Und somit wenn eine Subklasse wie **FlattenedFilteredNotesFolder** die Klasse **NotesFolderNotifier** vererbt, dann vererbt diese auch die Methoden der Klasse **NotesFolderObserverImpl**. Dadurch schauen die einzelnen Subklasse nicht mehr so groß aus wie davor und die Struktur sieht nun wie folgt aus:

Zu diesem Zeitpunkt der Refaktorisierung haben wir das Problem mit der Abhängigkeit (Problem 1) auflösen können. Die **NotesFolderNotifier** Klasse wird nicht mehr genutzt um sich selbst zu implementieren.

Betrachtet man die **NotesFolderObserver** Klasse weiter, dann fällt auf, dass diese Methoden für Notizen als auch Ordnern beinhalten. Diesen Anhaltspunkt habe ich mir zur Nutze gemacht und die Klasse erst mal getrennt. Methoden für Notizen in eine und Methoden für Ordner in andere Klasse.

Deswegen muss auch die **NotesFolder** Klasse zwei abstrakte Klasse anstatt eine und die **NotesFolderNotifier** muss auch zwei Implementierungsklassen anstatt eine inkludieren. Hier muss man die *mixin* Funktionaliät von Dart benutzen, da Dart nur eine Vererbung (*extends*) erlaubt. Damit habe ich nur eine kleinere Observer Klasse erreicht. Daher müsste ich eigentlich auch die **NotesFolder** Klasse und die **NotesFolderNotifier** Klasse jeweils in 2 Klassen trennen, eine für die Notizen und eine für die Ordner. Das geht leider nicht so einfach, da die Subklassen wie **FlattenedFilteredNotesFolder**, **FilteredNotesFolder**, etc. immer beide Arten von Methoden, Notizen und Ordnern, aufrufen.

Ein weiteres Merkmal bei den *NotesFolder*n Klassen ist die Häufigkeit der Benutzung diese Klassen. Die **FilteredNotesFolder** Klasse wird zum Beispiel nur in einer Stelle der Anwendung genutzt. Darüber hinaus, rufen alle Subklassen außer **Notes-**

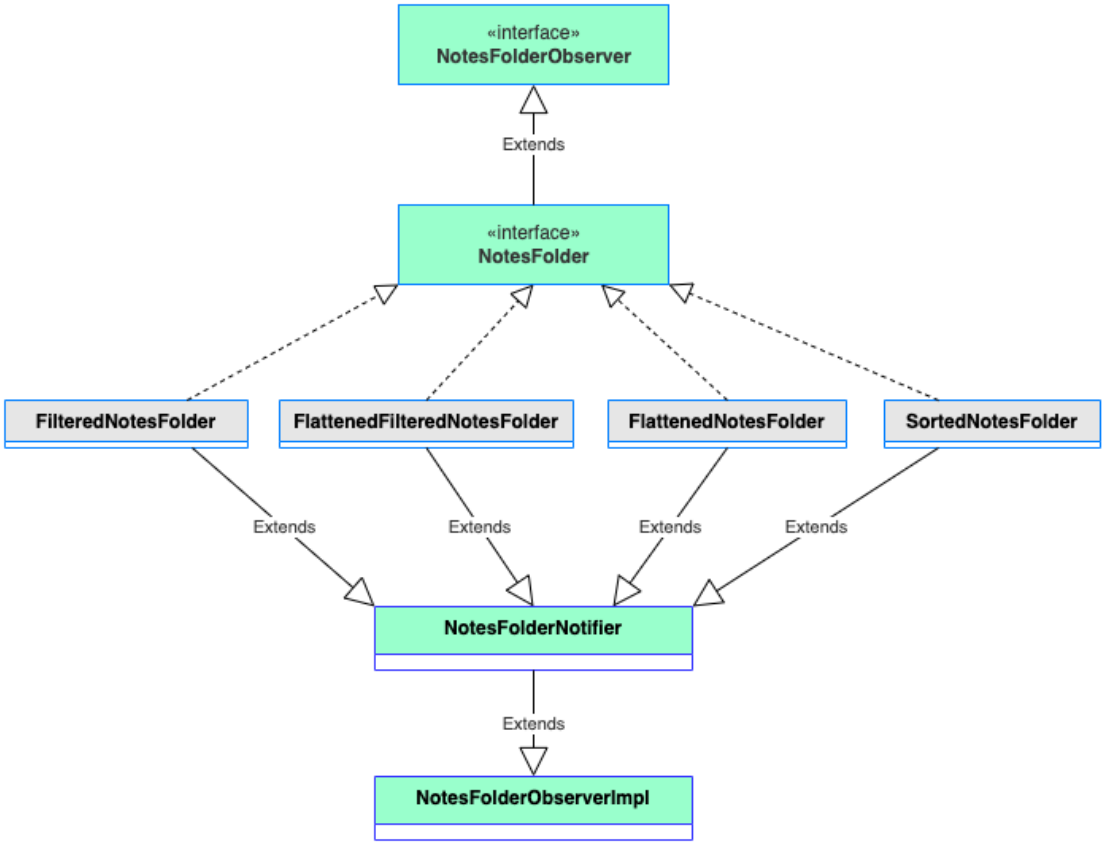

Abbildung 8: NotesFolder Klassen nach der Refaktorisierung

FolderFS die gleichen 6 "listener"Methoden auf und am Ende die gleichen 6 Methoden um die "listenerßu entfernen.

Dieses Muster habe ich als Anhaltspunkt für die nächste Refaktorisierung genommen und habe diese Methoden extrahiert und in eine neue gemeinsame Klasse verschoben.

Dann könnte ich die *mixin* Fähigkeit der Programmiersprache Dart nutzen. Durch *mixin* kann eine Klasse mehrere Klassen mit zusätzlichen Funktionen erweitern, ohne eine Klasse zu vererben. Nach dem Extrahieren der Oberserver Listen kann die Klasse **NotesFolder** nun diese Observer Funktionaliäten direkt erweitert werden und gleichzeitig hat die **NotesFolder** Klasse noch die Erweiterung von den restlichen Methoden der Klasse **NotesFolderNotifier**

Anders als bei anderen Subklassen, welche die abstrakte Klasse **NotesFolder** implementiert, hat die Klasse **NotesFolderFS** mehrere Funktionen und wird überall in der Anwendung genutzt. Da stellt sich die Frage, ob diese noch den Subklassen angehören und ob man diese komplett von der ganzen *NotesFolder* Klassen Struktur, das bis jetzt beschrieben würde, trennen soll. Dann könnte man eine separate Implementierung Struktur für die Klasse **NotesFolderFS** benutzen und dadurch die abstrakte Klasse **NotesFolder** minimieren, weil viele Methoden und Variablen der Klasse **NotesFolder** nur von **NotesFolderFS** genutzt wird. Diese Überlegung habe ich als

Anmerkung bei meiner Dokumentation hinterlassen, aber ich konnte aus zeitlichen Gründen dieser nicht erfolgreich nachgehen.

2.3.3 Fazit

Leider habe ich nach der Refaktorisierung diesen Bereich, bei dem manuellen Testen der Anwendung festgestellt, dass die Ordner nach Operationen wie Umbenennung oder Löschen nicht mehr funktioniert. Ich habe daher die Refaktorisierung rückgängig gemacht, da ich nicht wusste woran genau diese Fehler aufgetreten ist. Ich habe weiter mit den anderen Refaktorisierung, aus der vorherigen Kapitel, weitergemacht. Ich habe einen zweiten Versuch gestartet, diese zu refaktorisieren, aber ich habe keine Lösung dafür gefunden, weil ich den Mechanismus nicht ganz verstehen konnte und auch weil ich nicht mehr genügend Zeit habe.

Bei dieser Refaktorisierung habe ich mir ziemlich schwer getan. Ein Grund war, dass ich am Anfang nicht verstanden habe wie dieser Code Bereich funktioniert.

Wenn ich meine Dokumentationen betrachte, sehe ich hier viele Einträge, wo ich nicht wusste wie ich vorgehen soll und wie die Klasse funktioniert und verwendet werden.

Description	Category
flattened notes folder communicate back and forth with notes_folder_notifier. hard to understand the flow and for me to extract them	
try to remove notesfolder dependency on notesfolder_notifier, to see what gonna break	
on testing, there is also an error in list_view.dart which is a widget class —> commenting out the code for now. after running the test again: the “load notes” test passed but “add notes” did not pass because “_noteAdded called on a note already added”	
Dont know what my next step should be	stuck
can’t see how to move forward from here, maybe i should go back before removing dependency in notesfolder and then try to refactor “NotesFolderNotifier” class instead.	stuck
I set a breakpoint and went through the “_addFolder” function. my hypothesis right now is that I could extract the large “NotesFolderNotifier” class into 2 seperate classes. To test this theory I will go back a few steps	thinking
revert changes back to da6cf0013b00ed25f47aab583526ca0a804b2499	redo
notice that there are 2 type of listener: 1 is for folder other one are for notes	small refactoring
attempt to have an extra class “NotesFolderObserver” which hold observerlist objects.	restructuring, medium refactoring
extract note observer list in to separate class, through mixin (with) instead of “extend” the abstract class “NotesFolder” can use both of the observer class	restructuring, medium refactoring
errors occurs in “notes_folder_notifier.dart” after deleting observerlists. I think the notifier class is relying on recursion to go through the note/folder tree structure. hard to understand the flow!	stuck
extending abstract class NotesFolder leads to requirement of overriding the extended class “NotesFolderObserver”	stuck
extends NotesFolder with NotesFolderObserver for now and use the same mixin logic as the original code. I want to fix the error in notifier class first and then go fix the ‘bad’ mixin logic	stuck, try out
change notes_folder_notifier methods and passing observer directly to those methods	try out
through previous changes FlattenedNotesFolder and NotesFolderFS class need to be adjusted. Also at list_view.dart a folder called addListener & remoteListener directly NotesFolderNotifier, but it is not possible anymore bc NotesFolder does not implements NotesFolderNotifier directly, therefore extra methods has to be added to NotesFolder abstract class	restructuring, fixing error
at the moment NotesFolder inherits from NotesFolderOberserver but implementation class like FlattenedNotesFolder has to add NotesFolderObserver as mixin in order to not have to override the NotesFolderObserver methods, which leads to weird dependency graph	stuck, thinking
	thinking

Abbildung 9: Einträge der Dokumentation während der Refaktorisierung (Links im Anhang)

Rückblickend frage ich mich, ob es nicht besser wäre, diesen Bereich umzuschreiben, anstatt zu versuchen die einzelnen Teile zu refaktorisieren. Es liegt an den vie-

len verschiedenen Abhängigkeiten dieser Klassen. Wenn eine Refaktorisierung vorgenommen wurde, muss man an verschiedene Stellen etwas anpassen.

3 Schlussfolgerung

Bei der Refaktorisierung habe ich nicht alle SOLID Prinzipien identifizieren können. Teilweise sind diese nicht eindeutig erkennbar. Bei manchen Fällen war ich mir nicht sicher, ob die vorliegende Stelle die SOLID Prinzipien erfüllt bzw. nicht verletzt. Darüber hinaus, fand ich es bei dem Single-Responsibility Prinzip schwer genau zu identifizieren, wo die genaue Grenze dieses Prinzip definiert ist. Ich habe zum Beispiel die Kategorien nach Klassen eingeordnet und voneinander getrennt, aber eine weitere Möglichkeit wäre, diese nach Operationen zu gruppieren, da bei bei der Entfernung Beispielsweise ein Ordner auch die dazugehörigen Notizen entfernen soll. Am Ende habe ich nicht genügend Zeit gehabt, diese Möglichkeit zu testen und zu vergleichen.

Interessant fand ich, dass die Code Smells gute Hinweise darauf geben kann, welche Prinzipien der "Clean Architecture"verletzt werden könnten. Beispielsweise geben lange Klasse und Methoden den Hinweis, dass das Single-Responsibility-Prinzip verletzt ist. Das Gleiche gilt für die „umfangreiche Klasse Code Smells“.

Am Ende bin ich mit der Refaktorisierung 1 und 2 zufrieden. Nach der Refaktorisierung, kann ich die Schichten der Clean Architecture erkennen, ohne diesen Bereich komplett neu zu schreiben. Das zeigt, dass es möglich ist, ein bestehendes Software mit kleinen Refaktorisierung Schritten auf das „Clean Architecture“ Modell zu refaktorisieren.

Literatur

[1] *A Field Study of Technical Debt*. https://insights.sei.cmu.edu/blog/a-field-study-of-technical-debt/. 2015. URL: https://insights.sei.cmu.edu/blog/a-field-study-of-technical-debt/.

[2] *A tour of the Dart language*. https://dart.dev/guides/language/language-tour. URL: https://dart.dev/guides/language/language-tour.

[3] Martin Fowler. *Refactoring - Wie Sie das Design bestehender Software verbessern (2. Auflage)*. MITP-Verlags GmbH & Co. KG, 2020.

[4] Matthias Kleine. *Das Dependency Inversion Prinzip*. http://prinzipien-der-softwaretechnik.blogspot.com/2013/01/das-dependency-inversion-prinzip.html. 2013. URL: http://prinzipien-der-softwaretechnik.blogspot.com/2013/01/das-dependency-inversion-prinzip.html.

[5] Matthias Kleine. *Das Liskovsche Substituionsprinzip*. http://prinzipien-der-softwaretechnik.blogspot.com/2013/01/das-liskovsche-substituionsprinzip.html. 2013. URL: http://prinzipien-der-softwaretechnik.blogspot.com/2013/01/das-liskovsche-substituionsprinzip.html.

[6] Matthias Kleine. *Das Acyclic-Dependencies Prinzip*. http://prinzipien-der-softwaretechnik.blogspot.com/2014/09/das-acyclic-dependencies-prinzip.html. 2014. URL: http://prinzipien-der-softwaretechnik.blogspot.com/2014/09/das-acyclic-dependencies-prinzip.html.

[7] Matthias Kleine. *Das Common-Closure Prinzip*. http://prinzipien-der-softwaretechnik.blogspot.com/2014/06/das-common-closure-prinzip.html. 2014. URL: http://prinzipien-der-softwaretechnik.blogspot.com/2014/06/das-common-closure-prinzip.html.

[8] Matthias Kleine. *Das Reuse-Release Equivalence Prinzip*. http://prinzipien-der-softwaretechnik.blogspot.com/2014/06/das-reuse-release-equivalence-prinzip.html. 2014. URL: http://prinzipien-der-softwaretechnik.blogspot.com/2014/06/das-reuse-release-equivalence-prinzip.html.

[9] Martin, Robert C. und Grenning, James und Brown, Simon. *Clean Architecture: Das Praxis-Handbuch für professionelles Softwaredesign*. MITP-Verlags GmbH & Co. KG, 2018.

A Anhang

- Quellcode der Refaktorisierung 1: https://github.com/avatarnguyen/GitJournal/tree/merge_refactor_git
- Quellcode der Refaktorisierung 2: https://github.com/avatarnguyen/GitJournal/tree/ref_presenter_classes
- Quellcode der Refaktorisierung 3: https://github.com/avatarnguyen/GitJournal/tree/ref_notes_folder_classes
- Dokumentation Tabelle: https://docs.google.com/spreadsheets/d/1hfWc588DnYNYYpYft5p4IC6
edit#gid=536766806