Programmentwurf
einer
Notizverwaltungsanwendung
in Java
für
Advanced Software Engineering

Quelle: https://eventfaq.de/auch-persoenliche-notizen-unterfallen-der-dsgvo

Name:	David Baur
Matrikelnummer:	6977148
Kurs:	TINF18 B5

Inhaltsverzeichnis

Grundprinzip der Anwendung

Die Notizverwaltungsanwendung ist eine Anwendung, bei der der Benutzer über die Kommandozeile Notizen erstellen, lesen, manipulieren und löschen kann. Darüber hinaus ist es möglich, Notizen per Mail zu versenden. Außerdem können Analysen für die Notizen wie z.B. Spellcheck und WordCount erstellt werden und auf Wunsch in einem Logfile gespeichert werden.

Ein Teil der bisher implementierten Funktionalität ist über den display Befehl zu erreichen. Hierbei wird in einer GUI eine Übersicht über die vorhandenen Notizen gegeben. Eine Übersicht über die sonst vorhandenen Befehle liefert der help bzw. help+ Befehl.

Unit Tests

ATRIP Regeln

Das Akronym ATRIP steht für Automatic, Thorough, Repeatable, Independent und Professional. Automatic bedeutet in diesem Zusammenhang, dass das Aufrufen der Tests sowie das Prüfen der Ergebnisse mit einem Klick geschehen kann. Mit Thorough wird die Gründlichkeit bzw. Sorgfältigkeit betont mit welcher der Code getestet wird. Auch Randfälle sollten abgedeckt werden und insgesamt eine hohe Abdeckung erreicht werden. Repeatable bedeutet, dass bei jeder Neuausführung von Tests immer dasselbe Ergebnis herauskommen muss (pass/ fail). Independent meint, dass der Code möglichst unabhängig voneinander getestet werden soll. Dies kann z.B. durch den Einsatz von Fake- bzw. Mockobjekten erreicht werden. Mit Professional ist gemeint, dass die Tests grundsätzlich in einer hohen Qualität geschrieben werden. Es sollen demnach dieselben „Regeln“ (z.B. SOLID) gelten wie für Produktivcode.

Umsetzung von ATRIP

In der Anwendung wurden 13 Unit Test Klassen erstellt, um die wichtigsten logiklastigen Funktionalitäten zu testen. Hierbei wurde z.B. UI Code nicht getestet, weil dies als kompliziert und nicht zielführend erachtet wurde. Zusätzlich wurden im package testhelper/fakes Fakeobjekte erstellt, um die Funktionalitäten isoliert testen zu können.

Das I von ATRIP wurde auch dadurch umgesetzt, dass jeweils nur ein assert pro Unittest aufgerufen wird. Falls doch mehrere assert statements in einem Unittest aufgerufen werden, stellen diese die gleiche Funktionalität sicher.Ein Beispiel dazu ist in der countOccurrenceOfWordTest Methode der SingleNoteWordFinderTest Klasse zu finden, in welcher mehrere asserts verwendet werden, um sicherzustellen, dass auch bestimmte Randfälle hinsichtlich dem korrekten Zählen getestet werden. Es wird aber dieselbe Methode getestet.

Das A von ATRIP wurde durch den Einsatz des Junit Frameworks „automatisch“ umgesetzt, da die Tests alle direkt nacheinander oder nach Wunsch auch einzeln mit einem Klick in der IDE durchgeführt werden können.

Das T von ATRIP wurde versucht einzuhalten. Hierbei ist allerdings zu erkennen, dass die Gründlichkeit vermutlich noch nicht auf einem Topniveau ist. Dieses Topniveau ist allerdings schwierig zu erreichen, da es sehr viele unterschiedliche Eingaben geben kann, die einzeln validiert werden müssten (vermutlich auch zusätzliche Klassen im Produktivcode). Da es sich jedoch nicht um eine hochsicherheitskritische Anwendung handelt, wurde sich bei den Tests auf die gewöhnlichen Eingaben und Variationen beschränkt.

Das R von ATRIP wurde eingehalten, da jeder geschriebene Unittest bei mehrfacher Ausführung dasselbe Ergebnis liefert. Hierbei wird vorausgesetzt, dass sich der Produktivcode zwischen den einzelnen Tests nicht verändert. Von Zufall basierten Algorithmen wurde abgesehen, da diese das Repeatable-Prinzip verletzen würden.
Um zu prüfen, dass das Repeatable Prinzip eingehalten wurde, wurden die Tests 10 Mal hintereinander ausgeführt und die Ergebnisse verglichen.

Um das P von ATRIP zu erfüllen gibt es viele Herangehensweisen. Die Tests der Anwendung wurden z.B. nach dem Arrange Act Assert Prinzip geschrieben. In der Arrange Phase werden dabei die für den Test benötigten Objekte instanziiert. Daraufhin wird in der Act phase die zu testende Methode aufgerufen. Abschließend wird in der Assert Phase eine Annahme über das Ergebnis überprüft.

Grundsätzlich wurde zur weiteren Umsetzung des ATRIP Ps darauf geachtet, Programmierprinzipien einzuhalten (vgl. SOLID/ GRASP), um die Übersichtlichkeit, Wiederverwendbarkeit und Testbarkeit des Testcodes zu gewährleisten. Der Testcode wurde in dieser Hinsicht mit dem selben Anspruch wie der Produktivcode geschrieben. Ein Beispiel dafür ist die Filewriter Klasse im testhelper package, welche in mehreren Unittests als Teil der Arrange Phase verwendet worden ist. Man hätte hier statt der Klasse auch Code replizieren können. Dies würde allerdings gegen das DRY Prinzip verstoßen.

Programming Principles

SOLID

Das Single Responsibility Prinzip nach Uncle Bob besagt, dass jede Klasse genau eine Aufgabe besitzen soll. Um dem Prinzip Folge zu leisten wurden schon während dem Enwurf, aber auch bei der Implementierungsphase und den folgenden Refactorings darauf geachtet, Klassen nicht mit Funktionen zu überladen. Stattdessen wurde Funktionalität so unabhängig wie möglich umgesetzt. Die einzelnen Klassen wurden mit passenden, einheitlichen Namen versehen, um die angestrebte Übersichtlichkeit auch tatsächlich zu erreichen. Testbarkeit und Wiederverwendbarkeit werden durch die Einhaltung des Prinzips auch erhöht.

Das Open-Closed Principle besagt, dass Klassen offen für Erweiterungen sein soll. Im gleichen Zug sollen Klassen aber auch geschlossen für Modifikationen von außen sein. Das Prinzip wird in der Anwendung u.a. durch die Einführung einer abstrakten Klasse “AbstractCommand” angewandt. Diese bildet die Basis für Ereiterungen in Form von Custom Command Klassen, die von “AbstractCommand” erben (z. B. NoteDeclarationCommand). Es ist dabei nicht möglich, Klassen zu erstellen, die von der “Schablone” (der abstrakten Klasse) abweichen.

Das Liskov Substitution Principle besagt, dass eine abgeleitete Klasse an jeder beliebigen Stelle ihre Basisklasse ersetzen sollen kann. Dabei sollen keine unerwünschten Nebeneffekte auftreten. In der Anwendung ist das Prinzip umgesetzt. Das heißt konkret dass es keine Superklassen gibt, die allgemeinere Methodenparameter verwenden als Unterklassen.Beispielsweise nehmen die Methoden des Sorter Interfaces „allgemeine“ Typen wie Map entgegen bzw. geben diese zurück. Die Klassen, die das Interface implementieren wie z.B. RhymesNoteSorter, besitzen ausschließlich Methoden, die mit Typen auf derselben („allgemeinere“ Typen wären auch möglich) Abstraktionsebene operieren.

Das Interface Segregation Principle besagt, dass einzelne, kleinere Interfaces besser sind als ein großes Interface, das für einen generellen Zweck entworfen wird.
In der Anwendung wurde versucht, dieses Prinzip umzusetzen. In Commit 43be7426e79aae0c71fe53bbac936c0b146ef6ab wurde das Interface Declarator, das als großes Interface eingeführt werden sollte, in zunächst zwei Interfaces unterteilt: Ein Interface FileCreator und ein Interface HeaderAdder mit entsprechenden Methoden. Zu einem späteren Zeitpunkt eingeführte Deklarationsklassen besitzen nach diesem Refactoring die Möglichkeit, die jeweiligen Funktionalitäten unabhängig voneinander zu implementieren. Auch Testen lässt es sich einfacher. Die Übersichtlichkeit wird erhöht. Zwei Interfaces sind nach Meinung des Verfassers definitiv noch nicht „zu viele“.

Das Dependency Inversion Principle gibt Richtlinien für Abhängigkeiten einzelner Module. In der Anwendung wurde besipielsweise umgesetzt, dass Abstraktionen nicht von Details abhängen, wie in Commit 52fd9df5e31aee611cbffaa502bae5e439479e52 skizziert. Weiter wurde darauf geachtet, dass Details von Abstraktionen abhängen. Das ist alleine durch die angemessene Benutzung von Interfaces (die keine Marker Interfaces sind) bzw. Abstrakten Klassen erzwungen werden.

Mit dem Commit 743a63260c427cf79019fa21930b4564d989ba17 wird deutlich, dass eine Struktur implementiert wurde, die eine Umkehrung von Abhängigkeiten unterstützt. Ohne diese Architektur müsste die Methode „execute“ der „NoteSorterCommand“ Klasse Logik für jede einzelne implementierung verwenden. Dies wäre zumindest potenziell sehr aufwendig und würde sich negativ auf die Übersichtlichkeit auswirken.

GRASP

Das Information Expert Pattern zielt darauf ab, dass neue Funktionalitäten derjenigen Klasse zugeordnet wird, die das meiste Vorwissen besitzt. Das Prinzip wurde in der Anwendung umgesetzt. Ein Beispiel, das die Umsetzung des Patterns beispielhaft an zwei Methoden erläutert, ist in 277f23d887071a555ea19cda1679c7c9293a54be beschrieben. Static helper Methoden können verhindert werden, was im Sinne der Objektorientierung ist.

Das Creator Pattern definiert Richtlinien, unter welchen Bedingungen Instanzen einer Klasse B in einer Klasse A erzeugt werden dürfen. Das Prinzip wurde zumindest im Großteil der Anwendung umgesetzt. Ein Beispiel für die Umsetzung des Creator Patterns ist in Commit f43a42afe832639fe408260e8ce773644a0a9ce3 zu sehen. Hierbei ist die Klasse A (SingleNoteDispatcher) abhängig von Klasse B (MimeMessage). Weitere erstellte Objekte tragen zur Erstellung von Klasse B bei.

Über einen Controller können sämtliche Informationen (beispielsweise Benutzereingabe einer GUI) an entsprechenden Klasse der Domänenschicht weitergeleitet werden. Dieser Ansatz garantiert noch kein sauberes Design, begünstigt ein solches aber zumindest. In der Anwendung wurde das Design nicht umgesetzt. An Stellen im Code, wo das Pattern Anwendung finden könnte, wurde sich für einen anderen Ansatz entschieden. Beispielsweise wurde im Commit 8dfb3f28fb6a8d3cfce6b1a0122662412697cb73 kurz erklärt, wie Polymorphie anstelle des Controller Patterns benutzt wurde.

Low Coupling bedeutet im GRASP Kontext, dass nur nötige Abhängigkeiten in der Anwendung platziert werden. Dies bringt einige Vorteile mit sich wie z.B. leichtere Anpassbarkeit, bessere Testbarkeit und erhöhte Wiederverwendbarkeit.

Ein Beispiel dafür, dass in der Anwendung das Low Coupling Prinzip auftritt, ist im Commit dae3073efe638cff773d065f7b4ebb87d59e593b beschrieben. Häufig erreicht man Low Coupling durch die Einführung einer weiteren Abstraktionsschicht wie im Commit skizziert. Häufig ist es auch sinnvoll ein Interface via Dependency Injection zu übergeben. In der Anwendung macht das an der Stelle wenig Sinn. Man müsste die Definition der Abstrakten Klasse ändern bzw. weitere Klassen einführen, was an der Stelle nach Meinung des Verfassers etwas über das Ziel hinausschlagen würde.

Versteckte Kopplungen treten auf, wenn zwei Methoden voneinander abhängen und in der falschen Reihenfolge aufgerufen werden können. Sie kann z.B. durch die Parametrisierung der „zeitlich zweiten“ Methode geschehen.
In der Anwendung werden potenzielle Versteckte Kopplungen u.a. durch private Konstruktoren in Kombination mit Static Factory Methoden vorgebeugt (vgl. f8e01ea520cde0ce8b744955171780308c9a5428).

High Cohesion bedeutet, dass die einzelnen Methoden einer Klasse einen hohen funktional-/ inhaltlichen Zusammenhang besitzen. Dadurch wird das Low Coupling Prinzip unterstützt. In der Anwendung ist High Cohesion zu großen Teilen vorhanden. Zum Beispiel in Commit e3e6bccab8f6aecdd187452e674f775ab55ae038 wurde festgestellt, dass NoteStack und Notes sehr eng zusammengehören. Da die Methoden von NoteStack in seinen Methoden häufig auf Notes bzw. dessen Felder zurückgreift, ist der inhaltlich- funktionale Zusammenhalt in der Klasse sehr hoch (nicht maximal). Das Ergebnis von einer erfolgreichen Umsetzung von High Cohesion ist, dass Funktionalität dort ist, wo man sie erwartet.

Polymorphie ist für gewöhnlich Bestandteil eines Projektes, das in einer Objektorientierten Sprache umgesetzt ist. Es sagt aus, dass es bestimmte Ausprägungen einer Einheit bzw. Klasse gibt, die den Programmablauf entsprechend beeinflussen. Polymorphie kann den Einsatz von Switch Statements verhindern und ist dabei weniger fehleranfällig. In der Anwendung wird Polymorphie z.B. durch Interfaces erzwungen (z.B. c8e9ca4c894b79a1e2a4ddd2e93e445ea73c79a8).

Pure Fabrication bedeutet, dass eine Klasse eingeführt wird, die statische Hilfsmethoden anbietet. Das heißt, dass diese Klasse kein Experte für die in der Methode verarbeiteten Informationen ist. Durch die Verwendung kann High Cohesion + Low Coupling unterstützt werden, da unpassende Codefragmente aus einer Klasse losgelöst werden können. Man sollte das Pure Fabrication Pattern allerdings mit Bedacht einsetzen, da bei häufiger Nutzung die Gefahr besteht, die Vorteile der Objektorientierung zu verlieren. In der Anwendung wird beispielsweise eine mathematische Berechnung von Prozentwerten in eine Klasse PercentageCalculator ausgelagert, um generelles von domänenspezifischem Wissen zu trennen. Eine Referenz hierfür enthält der Commit 2c7190f741426b00e52eb835b3c4f48905c3e943. Generell wurde versucht, den Anteil der Hilfmethoden gering zu halten.

Mit Indirection können Abhängigkeiten gebrochen werden, indem z.B. eine Mediatorklasse eingeführt wird, über welche die Kommunikation zwischen zwei oder mehreren Klassen läuft. Auch das Controller Pattern kann als Umsetzung des Indirection Prinzips verstanden werden, da eine Schicht zwischengeschaltet wird und die Kommunikation daher nicht direkt geschehen muss. Für Indirection lassen sich in der Anwendung wenige Beispiele finden.

Protectet Variations verfolgen den Ansatz, einzuelne “Bereiche” der Anwendung voneinander zu isolieren (i.d.R. durch Einführung einer Abstraktionsschicht). Das Prinzip hängt sehr eng mit dem aus SOLID bekannten Open-Close Prinzip zusammen. In der Anwendung wird das Prinzip durch Interfaces und Abstrakte Klasse umgesetzt.

DRY

DRY ist die Abkürzung für Don’t Repeat Yourself. Um das Prinzip zu erfüllen, dürfen also keine Dopplungen in der Codebasis auftreten. Falls Dopplungen vorhanden sind, können diese u.a. durch die Nutzung von Methoden aufgelöst werden. Auf diese Weise können diese möglichst atomaren Einheiten in anderen Bereichen der Anwendung wiederverwendet werden. In der Anwendung wurde das Prinzip weitestgehend eingehalten. Drei Formatierungsklassen wurden durch die Klasse NoteSorterFormatter ersetzt, da sie mehr oder weniger den selben Code enthielten. Der Commit dazu ist bb4d7f02e21d2d75d70084edec645e039fe1fcde.

Clean Architecture

Es wurde festgestellt, dass eine konkrete Klasse zur Konvertierung von Noteobjekten zu Notes fehlt. Deshalb wurde eine Klasse Noteadapter eingeführt, welche eine Methode convertToNote besitzt.

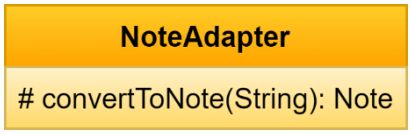

Abbildung 1: InitializeDisplayNotes aus DisplayNotes vor Refactoring

Die Komplexität der statischen initializeNote() Methode in der Note Klasse wurde verringert, da die Funktionalität der Adapterschicht in die separate Klasse ausgelagert wurde. Der Commit b9cf0ff7d7782b3d3a7e7ed1a0a9ad002c693c3b enthält die genaue Implementation der Methode.

Refactoring

In der Klasse DisplayNotes wurden Codesmells festgestellt. Die statische Initialisierungsmethode der Klasse hat über 50 Codezeilen enthalten, was darauf hindeutet, dass wenige Methoden zur Unterteilung benutzt worden sind. Aus diesem Grund wurde ein Refactoring des Konstruktors vorgenommen, sodass als Ergebnis nur noch 16 Zeilen Code im Konstruktor vorhanden sind. Nicht nur wirkt der Code dadurch übersichtlicher. Er ist durch die sprechenden Methodennamen und sinnvolle Gruppierung auch einfacher zu verstehen, zu warten und zu testen (auch wenn das bei Unittests für GUI-Methoden nicht so sehr der Falls ist). Auch die Wieder-verwendbarkeit ist potenziell erhöht worden.

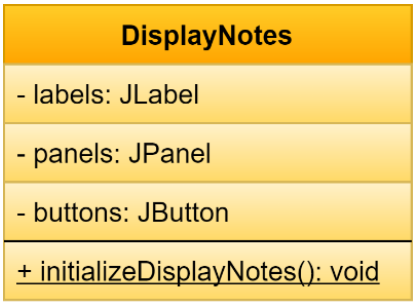

Abbildung 2: InitializeDisplayNotes aus DisplayNotes vor Refactoring

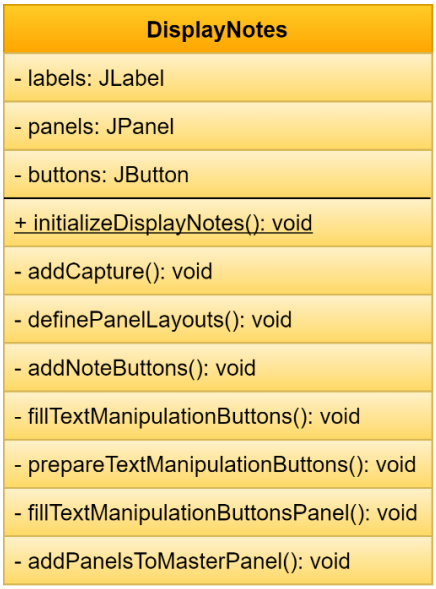

Abbildung 3: InitializeDisplayNotes aus DisplayNotes nach Refactoring

Die Membervariablen wurden in den UML Diagrammen auf der vorigen Seite aus Gründen der Einfachheit zusammengefasst. Der Fokus liegt bei diesem Refactoring auf den zusätzlichen Methoden, die in Abbildung 3 dargestellt werden.

Ein weiterer CodeSmell wurde in der Klasse SingleNotedispatcher gefunden. Hier wurde der Methode createMessage(…) fünf und der Methode createAttachment(…) vier Parameter übergeben. Derartige umfangreiche Parameterlisten können das Verständnis von Methodenaufrufen erschweren. Empfohlen ist in solchen Fällen eine Reduktion der Parameter durch die Zusammenfassung der Parameter in Objekte, sofern dies möglich ist. In der Anwendung wurde deshalb eine SendingInformation Klasse erstellt, die die Parameter hält. Diese Klasse wurde wie im Abschnitt Entwurfsmuster beschrieben mit dem Builder Entwurfsmuster erstellt. Sie besteht genau genommen aus zwei Klassen, namentlich SendingInformation und Builder.

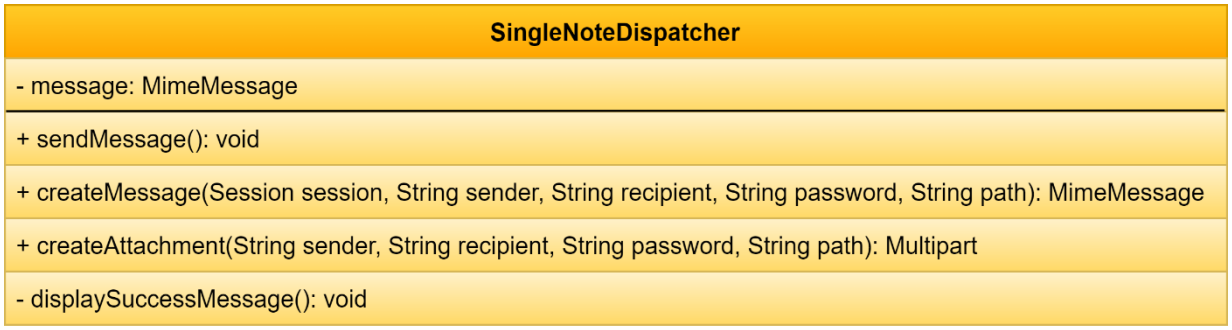

Abbildung 4: SingleNoteDispatcher vor Refactoring

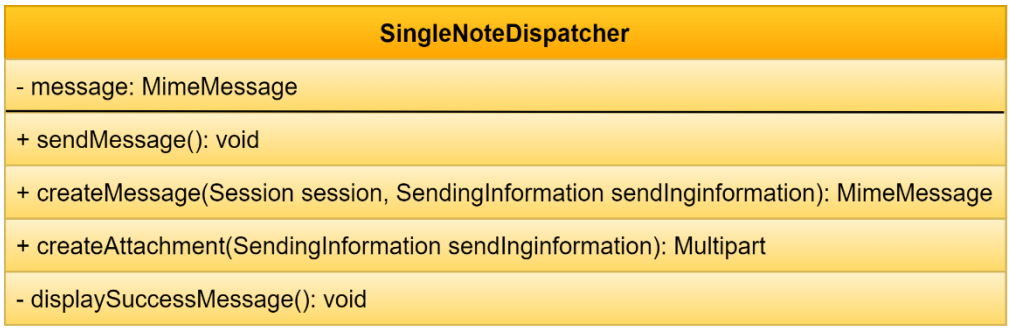

Abbildung 5: SingleNoteDispatcher nach Refactoring

Der veränderte Zustand ist in Commit fa4d4858978014955458fcd392618f4cbf3b62a1 referenziert und kurz beschrieben.

Entwurfsmuster

In der Klasse SendingInformation wurde das Builder Pattern eingeführt. Im Vorfeld wurde festgestellt, dass der konventionelle Konstruktor der Klasse vier Parameter entgegennimmt. Dies ist nach Meinung des Verfassers bereits ziemlich viel. Die Erstellung von Objektinstanzen ist potenziell unübersichtlich, auch weil der Entwickler (ohne Hinweise der DIE zumindest) nicht genau wissen kann, welche Parameter er dem Konstruktor für die Initialisierung übergeben muss. Weiter kann es sein, dass man nach späteren Erweiterungen der Klasse, mehrere Konstruktoren zur Erzeugung benötigt. Diese Probleme werden mit dem Builder Pattern behoben. Außerdem kann die Klasse als immutable markiert werden. Der größte Nachteil ist die notwendige Erstellung einer Builderklasse, was i.d.R. in mehr Codezeilen resultiert.
In f595342dee6e1d8c6ea42080b882f42ae80099cc befindet sich eine Referenz zur Sendinginformationklasse, der eine Innere Klasse Builder hinzugefügt worden ist.

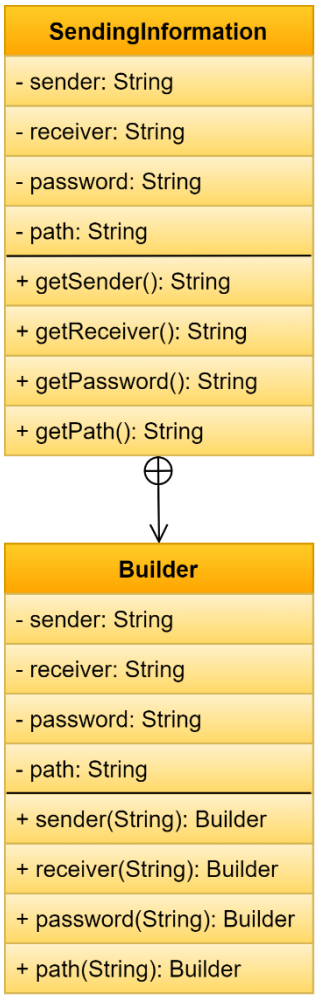

Abbildung 6: Anwendung des Builderpatterns in der Anwendung

Legacy Code

In der Klasse NoteCounterRaw wurde eine überflüssige Abhängigkeit in der Methode countWordsOfNote() identifiziert. Diese nahm einen Parameter vom Typ Note entgegen. Diese Art der Hartcodierung erschwerte die flexible Nutzung der Methode, weshalb ein neues Interface als Abstraktionsebene eingeführt wurde, welche als Parameter verwendet wurde. Das verwendete Interface namens WordListHolder erzwingt die Definition von einer countWords() Methode, die einen Stringarray zurückgibt. Damit wurde die Abhängigkeit gebrochen und ein flexiblerer Einsatz (auch hinsichtlich Tests) ist möglich. Die Änderungen sind in Commit c01b0903229678f051c5b8997fa61ffc78d402d0 einzusehen.

In Commit e18328f2e1ac870184be3e6f46c90599f9071651 ist dargestellt, wie die Klasse OverviewSpellChecker refactored wurde, um eine ungewollte Abhängigkeit zu brechen. Die Methode openFileDialogue() nahm ein Result Objekt entgegen, was eine direkte Kopplung an die Klasse bedeutete. Nach dem Refactoring nimmt die Methode als Parameter das Interface StringRepresentation entgegen, welches die Möglichkeit offenhält, andere Implementationen als Result zu nutzen und ein Fake-/ Mockobjekt zum Testen zu verwenden.

Domain Driven Design

Entity	
Bezeichnung	Note
Beschreibung	Hält Informationen über eine Notiz und enthält grundsätzlich hilfreiche Methoden zur weiteren Verarbeitung wie removeEmptyLines()
Attribute	Public Path completePath Public String noteName Public String[] wordList Public String[] lineList Public String content Protected String contentForGraphicalProcessing

Abbildung 7: Note Entity

Entity (enhält Aggregation von Note)	
Bezeichnung	NoteStack
Beschreibung	Hält strukturierte Informationen über alle Notizen des festgelegten Verzeichnisses
Attribute	Protected List<Path> pathList Protected List<String> noteContenList Protected List<String[]> separatedWordListList Public Set<String> noteNames Public List<Note> notes

Abbildung 8: NoteStack Entity

Entity	
Bezeichnung	NoteButtonActionListener
Beschreibung	Listener für GUI Buttons. Auf Klick wird die entsprechende Note angezeigt und die Analysebuttons aktiviert
Attribute	Private DisplayNotes displayNotes Private String noteName

Abbildung 9: NoteButtonActionListener Entity

Entity	
Bezeichnung	SingleNoteWordFinder
Beschreibung	Findet eingegebene Zeichenfolge für eine definierte Notitz und gibt die Häufigkeit des Auftretens mit Lokalisierung an
Attribute	Private Map<Integer, Integer> wordOccurrence

Abbildung 10: SingleNoteWordFinder Entity

Value Object	
Bezeichnung	Globals
Beschreibung	Enthält vordefinierte Pfade für Notes und Logfiles sowie eine Scanner Instanz, die zentrale Funktionen in der Anwendung haben
Attribute	Public static String path_for_notes Public static String path_for_logfiles

Abbildung 11: Globals Value Object

Value Object	
Bezeichnung	SendingInformation
Beschreibung	Enthält Informationen, die zum versenden von Note Files über E-Mail wichtig sind. Kann über Builder gebaut werden. Ist unveränderlich (keine Vererbungshierarchie)
Attribute	Public String Recipient Public String Sender Public String Path Public String Password

Abbildung 12: SendingInformation Value Object

Value Object	
Bezeichnung	LineOverwriterInformation
Beschreibung	Enthält Informationen, die für das Übeschreiben einer Notizzeile nötig sind. Dies passiert beispielsweise durch den NotelineEditor.
Attribute	Private Path completePath Private int indexLineNumber Private String replacementLine

Abbildung 13: LineOverwriterInformation Value Object

Schwächen der Anwendung

Auch wenn in der Anwendung einige sinnvollen Prinzipien umgesetzt wurden, offenbart die Anwendung auch Schwächen. Für den Anwendungsfall ausreichend aber für Erweiterungen/ Umstrukturierungen problematisch ist, dass das lokale Filesystem das Zentrum der Anwendung hinsichtlich der Persistenz darstellt. Das bedeutet, dass ein späteres Austauschen der Persistenzlogik nur durch grundlegende Neustrukturierungen/ Codeveränderungen an mehreren Stellen durchgeführt werden könnte.

Abbildungsverzeichnis