31. MAI 2021

PROGRAMMENTWURF "MEDIT"

TECHNISCHE DOKUMENTATION

KEVIN REIS

INHALT

1 Über das Projekt 1

2 Domain Driven Design 2

2.1 Analyse der „Ubiquitous Language“ 2

2.1.1 Markup Language 2
2.1.2 Texteditor 2
2.1.3 Befehle – Command 2
2.1.4 Textformatierung – Text Action 3
2.1.5 Dokument 4
2.1.6 Dateireferenz 4

2.2 Analyse der verwendeten Muster 4

2.2.1 Value Objects 4
2.2.2 Entities 4
2.2.3 Aggregates 5
2.2.4 Repositories 5
2.2.5 Domain Services 6

3 Clean Architecture 7

4 Programming Principles 9

4.1 SOLID 9

4.1.1 Single-Responsibility-Prinzip 9
4.1.2 Open-Closed-Prinzip 9
4.1.3 Liskovsches Substitutionsprinzip 9
4.1.4 Interface-Segregation-Prinzip 10
4.1.5 Dependency-Inversion-Prinzip 10

4.2 GRASP 11

4.2.1 Information Expert 11
4.2.2 Creator 11
4.2.3 Controller 11
4.2.4 Low Coupling 11
4.2.5 High Cohesion 12
4.2.6 Polymorphismus 12
4.2.7 Pure Fabrication 12
4.2.8 Indirection 13

4.2.9 Protected Variations 13
4.3 DRY 13
5 Refactoring 14
5.1 Mapper für FileEntity und FileReference 14
5.2 Änderung des Entwurfsmusters für TextActions 15
6 Entwurfsmuster 17
Verweise iii
Textblockverzeichnis iii

1 Über das Projekt

Im Rahmen des Programmentwurfs für die Vorlesung „Advanced Software Engineering“ wird eine Android-App namens „MeDit“ entwickelt. Mit dieser App können Markdown-Dokumente erstellt, bearbeitet und gespeichert werden.

Markdown ist eine Sprache, mit der Texte verfasst werden und einfach zu HTML konvertiert werden können. Ziel der Sprache ist es, möglichst einfach lesbar und zum Schreiben zu sein. [1]

Mit diesem Editor soll es zudem möglich sein, verschiedene Formatierungsoptionen auswählen und anwenden, sowie ein gerendertes Ergebnis des Dokuments darstellen zu können. Erweitert wird das Textbearbeitungsprogramm durch die Darstellung mathematischer Ausdrücke im LaTeX-Format.

Die Applikation wird in Kotlin entwickelt und ist für die Android API-Versionen 28 aufwärts mit Target 30.

2 Domain Driven Design

2.1 Analyse der „Ubiquitous Language“

2.1.1 Markup Language

Unter einer Markup Language ist eine Sprache zu verstehen, die ein elektrisches Dokument gliedert, die z. B. Bilder, Links oder Überschriften besitzt. In dieser App wird überwiegend auf Markdown gesetzt. Ein anderes, bekanntes Beispiel ist HTML, welches als Grundlage für das World Wide Web und für die Browsers genutzt wird.

2.1.2 Texteditor

Als Texteditor versteht man einen Bereich der Applikation, in dem Veränderungen an einem Dokument möglich sind. So kann ein Text verfasst, ein Dokument geladen und gespeichert werden, sowie Textmanipulationen wie Ersetzen oder Einfügen durchgeführt werden. `TextEditor` ist als Interface definiert, da die Verwendung einer abstrakten Klasse in einem Controller in Android unflexibel ist.

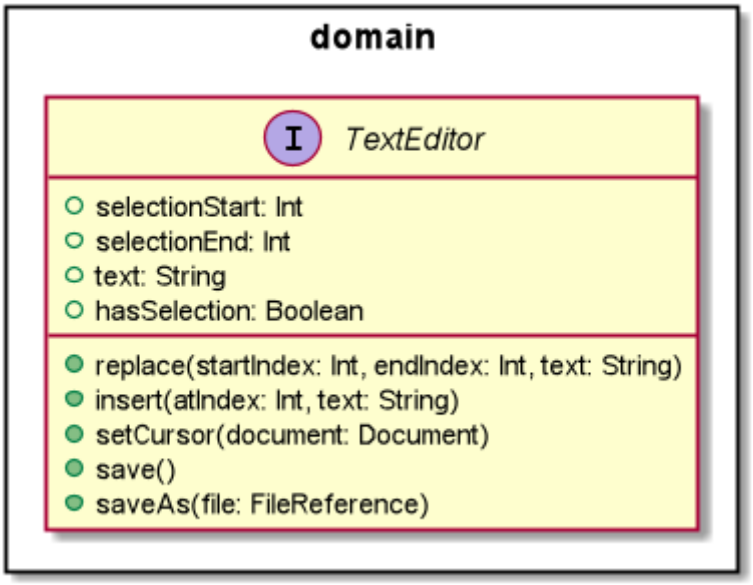


Abbildung 1: UML-Diagramm "TextEditor"

2.1.3 Befehle – Command

Befehle, oder auch `Command`, werden durch eine Eingabe des Users verursacht. In diesem Kontext fallen z. B. auch Textformatierungen.

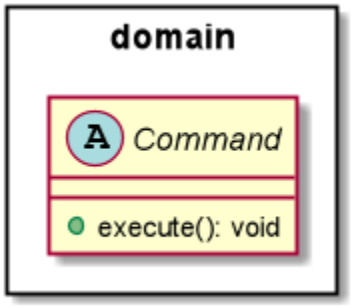


Abbildung 2: UML-Diagramm "Command"

2.1.4 Textformatierung – Text Action

Jede Textmanipulation bzw. -formatierung, die die Darstellung der Schrift beeinflusst, wird „text action“ genannt. Im Rahmen einer Kommandomusterimplementierung sind diese Text Actions als `InlineTextActionCommand` und `BlockTextActionCommand`. `InlineTextActionCommand` ist ein Befehl, um eine Textformatierung mitten in einer Zeile („inline“) durchzuführen, `BlockTextActionCommand` fügt an der Stelle des Cursors im Texteditor ein Block („block“) mit der gewählten Formatierung ein. Beispiele für inline text actions sind **fettgedruckt**, *kursiv* oder ~~durchgestrichen~~. Unter „block text actions“ fallen Codeblöcke (vgl. Textblock 1), Zitate (vgl. Textblock 2), Tabellen (vgl. Textblock 5), Bilder (vgl. Textblock 6), nummerierte (vgl. Textblock 4) und unnummerierte Listen (vgl. Textblock 3). Die Tabellen, nummerierte und unnummerierte Listen sind allerdings kein Bestandteil der Implementierung.

```
```
// Mein Beispielcode
public class Test {
    public static void main(string[] args) {
        System.out.println(“Hello world!”);
    }
}
```
```

Textblock 1: Codeblock

```
> Ein Zitat in Markdown.
> Vielleicht noch mit ein paar Zeilen mehr.
```

> Ein Zitat in Markdown.
> Vielleicht noch mit ein paar Zeilen mehr.

Textblock 2: Zitatblöcke

```
* Eine Liste
  * Mit paar
  * und vielleicht
    * noch mehr
    * Unterpunkten

- Eine Liste
    - Mit paar
    - und vielleicht
        - noch mehr
        - Unterpunkten
```

Textblock 3: Unnummerierte Liste mit Einrückung

```
1. Eine Liste
  1. Mit paar
  2. und vielleicht
    1. noch mehr
    2. Unterpunkten

1. Eine Liste
    a. Mit paar
    b. und vielleicht
        i. noch mehr
        ii. Unterpunkten
```

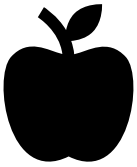
Textblock 4: Nummerierte Liste mit Einrückung

```
| Überschrift 1 | Überschrift 2 |
| ------------- | ------------- |
| Inhalt 1      | Inhalt 2      |
```

Überschrift 1	Überschrift 2
Inhalt 1	Inhalt 2

Textblock 5: Tabelle

```
![Bildinfo](http://mein.link/)
```



Textblock 6: Bilder

2.1.5 Dokument

Ein Dokument (Klasse `Document`) bezeichnet man als eine Einheit, in der ein Titel, ein Inhalt und eine Referenz auf die zugehörige Datei in einem Dateisystem hinterlegt ist.

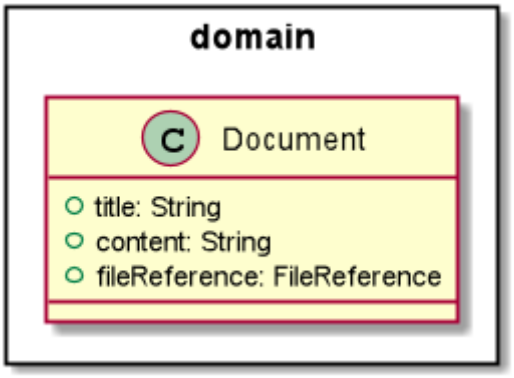


Abbildung 3: UML-Diagramm "Document"

2.1.6 Dateireferenz

Eine Dateireferenz (Klasse `FileReference`) ist eine Abstraktion der Informationen zu einer Datei. Sie besitzt einen Dateinamen, die Dateiendung, der Pfad zur Datei und einen Zeitstempel des letzten Zugriffs. Die Dateireferenz eignet sich gut für eine

Speicherung in einer Datenbank für letzte Zugriffe und ist Bestandteil eines Dokuments.

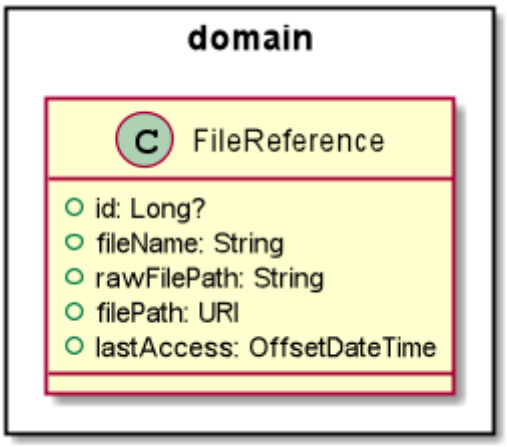


Abbildung 4: UML-Diagramm "FileReference"

2.2 Analyse der verwendeten Muster

2.2.1 Value Objects

Value Objects sind Klassen, die weder eine eindeutige Identität noch veränderbar sind. Diese sind im Projekt als `FileReference` (siehe Abbildung 4) und `Document` (siehe Abbildung 3) im Domain-Modul wiederzufinden. Die Besonderheit bei `FileReference` ist jedoch, dass das Datum des letzten Zugriffs bei einem Speicher- oder Ladevorgang aktualisiert werden muss. Da eine direkte Veränderung der Property `lastAccess` allerdings die Regel der Unveränderbarkeit verletzt, existiert ein Domain Service `FileReferenceDataUpdater`, der in Kapitel 2.2.5 näher beschrieben wird.

2.2.2 Entities

Entitäten besitzen eine eindeutige Identität innerhalb des Domänenmodells, können verändern werden und repräsentieren tatsächliche Objekte in einer Domäne. In diesem Projekt ist eine Entität als `FileEntity` zu finden. Sie ist eine Abbildung einer `FileReference`, die eindeutig über ihre Datenbank-ID wiedergefunden werden kann, und in einer Datenbank abgelagert wird. Der Use Case für diese Entity ist das Festhalten der zuletzt zugegriffenen Dokumente. Für die Konvertierung von `FileEntity` zu `FileReference` und umgekehrt ist der `FileEntityToFileReferenceMapper` zuständig.

2.2.3 Aggregates

In diesem Projekt wurden keine Aggregates verwendet. Ein Konzept wäre das `DocumentAggregate` im Domain-Modul, um komplexe Beziehungen zwischen Entitäten und Value Objects zu vermeiden. Sie besitzt eine Referenz auf einem `Document`-Objekt und auf einem `FileReference`-Objekt.

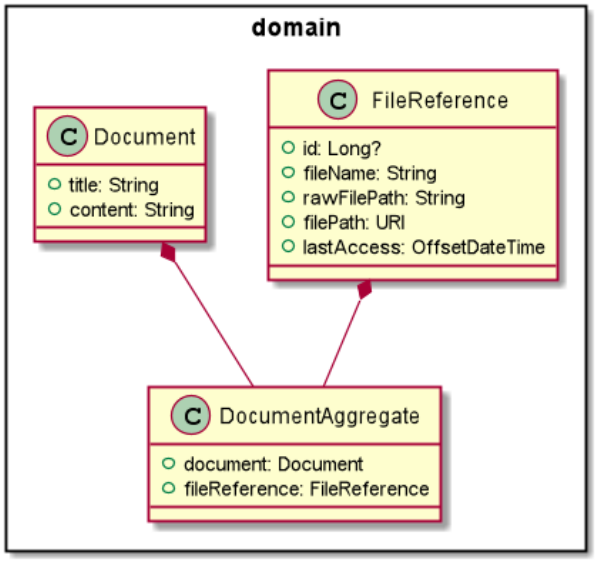


Abbildung 5: Konzeptionelles Klassendiagramm von "DocumentAggregate"

In der Regel werden Aggregates in Repositories verwendet. Da aber `Document` Inhalte eines Dokuments zwischenspeichert und nicht in einer Datenbank lagert, wird `DocumentAggregate` zum Zeitpunkt der Abfrage zuletzt genutzter Dateien keine Referenz auf ein `Document`-Objekt oder eine Referenz auf ein „leeres“ `Document`-Objekt besitzt. Demnach ist die Verwendung eines `DocumentAggregate` in diesem Kontext redundant.

2.2.4 Repositories

Repositories dienen als Data Access Object, wodurch Daten aus beliebiger Quelle abgefragt werden können. Sie kapseln die Logik für Persistenz und Erzeugung von Daten ab.

Im Projekt werden Repositories für u. a. die Speicherung und das Abrufen von zuletzt verwendeten Dokumenten implementiert. Ein Interface `RecentFileRepository` befindet sich in der Domainschicht der App, worin CRUD-Operationen als Funktionen definiert sind. Da die Dateiquelle unabhängig vom Projekt selbst gewählt werden kann und in der Android-App diese Daten aus einer Android Room-Datenbank abgerufen

werden, wird die Klasse `RoomRecentFileRepository`, die das Interface `RecentFileRepository` implementiert, in der Pluginschicht `plugins-android-database` zu finden sein. Sie nimmt im Konstruktor zusätzlich ein DAO entgegen, welches von Android Room als Interface bereitgestellt wird. Das Room-DAO-Interface implementiert eine ähnliche Schnittstelle wie `RecentFileRepository`, arbeitet allerdings mit `FileEntity` statt `FileReference`. In der Abbildung 6 wird zugunsten einfacher Darstellung `FileEntity` ausgelassen.

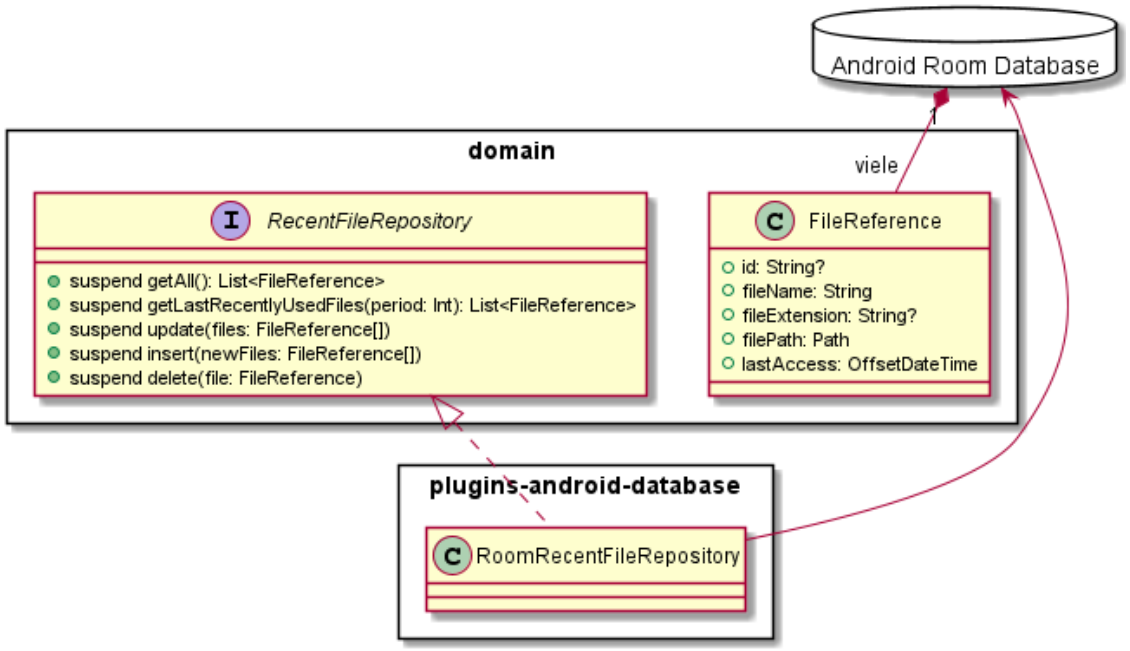


Abbildung 6: Klassendiagramm mit Beziehungen von "RecentFileRepository"

2.2.5 Domain Services

Um das Aktualisieren von Properties in einer Dateireferenz zu vereinfachen, ist ein `FileReferenceDataUpdater` im Modul `application` implementiert, welcher über Dependency Injection verfügbar ist. Die Provider-Methode befindet sich in `AppCoreModule` im `application`-Modul. Grund für diese Implementierung ist die Unveränderbarkeit von Value Objects. Um z. B. den Zeitstempel des letzten Zugriffs zu aktualisieren, ist das Klonen des Objekts erforderlich. Wenn dieser Vorgang an verschiedenen Stellen im Code notwendig ist, kann dadurch schnell redundanter Quellcode entstehen. `FileReferenceDataUpdater` ermöglicht das Updaten des Zeitstempels und der zugehörigen Datenbank-ID, sofern eine Update-Operation auf die Datenbank erfolgen soll, aber noch festgestellt werden muss, ob ein Eintrag für diese Dateireferenz bereits existiert.

3 CLEAN ARCHITECTURE

Das Projekt ist in sechs Modulen eingeteilt, welches fünf Schichten der Clean Architecture repräsentieren.

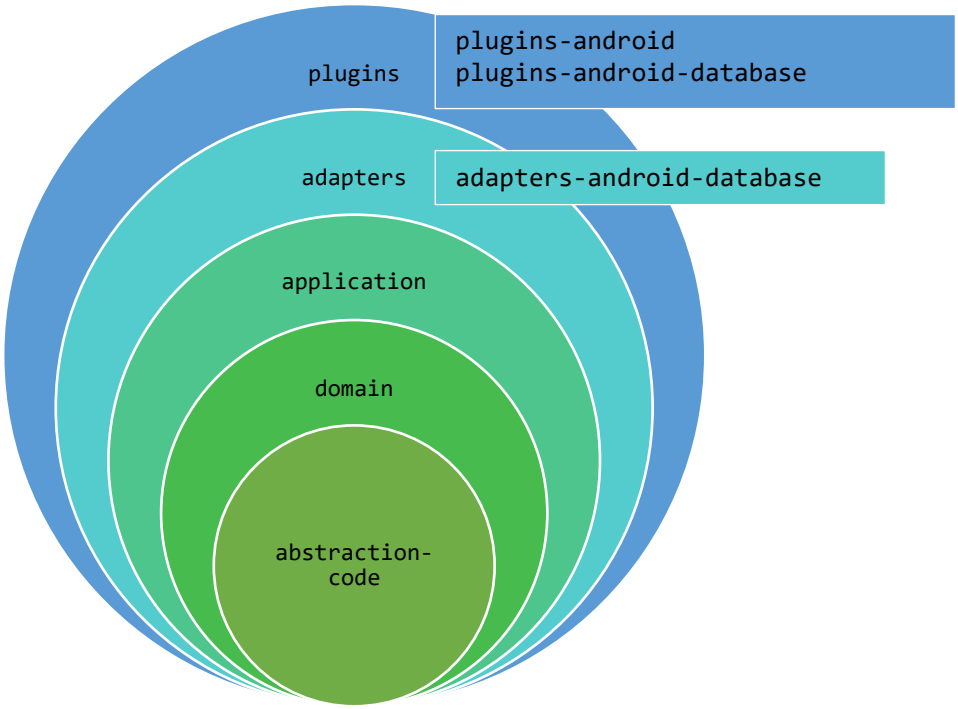


Abbildung 7: Schematische Darstellung der Clean Architecture

Anzumerken hierbei ist die Position der Android-Schicht (`plugins-android`). Sie befindet sich in der äußersten (Plugin) Schicht des Clean Architecture-Modells, da es sich um betriebssystemspezifische Implementierungen handelt. Dieser Ansatz ermöglicht eine einfachere Erweiterung der Betriebssystemunterstützung der MeDit-App, sodass in Zukunft z. B. eine Desktop-Version der App entwickelt werden kann.

An dieser Stelle befindet sich auch ein Modul für eine Android Room-Datenbank (`plugins-android-database`). Sie soll das Speichern von beliebigen Inhalten in einer SQLite-Datenbank ermöglich, die lokal auf dem jeweiligen Smartphone gesichert wird. Diese Funktionalität wird u. a. für das Darstellen von auf zuletzt zugegriffene Dokumente verwendet.

Um die verwendete Entität `FileEntity` in die Domäne zu übertragen, befindet sich auf der Adapterschicht das Modul `adapters-android-database`. Dort ist ein `FileReferenceToFileEntityMapper` enthalten, welches die Transformation von Pluginspezifischem ins Domänenspezifischem und umgekehrt ermöglicht. Zusätzlich

sind in diesem Modul `TypeConverter` implementiert, die URIs und Zeitstempel in Strings serialisieren, um diese Daten in Datenbanken speichern zu können.

In der Applikationsschicht sind die Implementierungen eines Executors von Text Action-Befehlen (siehe Kapitel 2.1.4) und ein Service zur Modifikation von FileReference (`FileReferenceDataUpdater`, nähere Beschreibung in Kapitel 6) enthalten.

Die Domainschicht wird durch das Modul `domain` repräsentiert und beinhaltet domänenspezifische Implementierungen, wie `Document`, `FileReference` und `Command`.

Im Modul `abstraction-code` befinden sich gemeinsame Bibliothekenabhängigkeiten, die überall im Projekt genutzt werden können. Android Hilt ist nicht enthalten. Diese Bibliothek erweitert Dagger für eine bessere Benutzung im Android-Kontext. Da es sich bei `abstraction-code` um eine pure Kotlin/Java-Library handelt, kann sich in diesem Modul kein Android-spezifischer Code befinden. Neben dieser Einschränkung finden sich in einigen `build.gradle`-Dateien aller Module ggf. redundante Einträge von Abhängigkeiten, die bereits in `abstraction-code` vorhanden sind. Aus unbekannten Gründen funktionieren aber transitive Abhängigkeiten in Gradle in diesem Projekt nicht zuverlässig.

4 PROGRAMMING PRINCIPLES

4.1 SOLID

4.1.1 Single-Responsibility-Prinzip

Für das Single-Responsibility-Prinzip gilt das Beispiel `TextActionCommandExecutor`. Diese Klasse ist dafür zuständig, Befehle (`Command`) mit einem momentanen Stand (für Dokumente der Inhalt) entgegenzunehmen, den übergebenen Befehl auszuführen und ein Logbuch über ausgeführte Befehle auf den jeweiligen Stand des Inhalts zu führen. Diese ist vor allem für die Umsetzung des Rückgängig-Mechanismus vorteilhaft. Sofern eine Operation rückgängig gemacht werden soll, kann der letzte Stand aus dem Logbuch entfernt werden. Man erhält daraufhin das entfernte Objekt zurück. Zwar besitzt die momentane Implementierung eine teils höhere Kopplung zwischen `TextEditor` und des `BlockTextActionCommand` bzw. `InlineTextActionCommand`, allerdings kann ein Refactoring durchgeführt werden, um die Verantwortung des Ortes der Befehlsausführung des `TextActionCommandExecutor` zu überlassen.

4.1.2 Open-Closed-Prinzip

Beispiel sind `BlockTextActionCommand` und `InlineTextActionCommand`. Sie implementieren das `Command`-Interface, was beide Klassen in ihrem Ursprung gleich macht. Betrachtet man nun `MarkdownTextActionCommands`, so bemerkt man, dass einige Inline bzw. Block Actions teils modifiziert werden. So ist zwar `HeaderCommand` ein `BlockTextActionCommand`, allerdings ist die vollständige Funktionalität des „Umranden“ eines Blocks nicht notwendig. Start- und Endblock werden überschrieben, sodass nur die „#“ als eingesetzt werden. Das zeigt, dass Command fundamental und funktional gleich sind, allerdings beliebig erweitert und angepasst werden können.

4.1.3 Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip ist theoretisch auf `BlockTextActionCommand` und `InlineTextActionCommand`, allerdings auch nur auf diese Klassen anwendbar. Verschiedene Implementierungen des `BlockTextActionCommand` können untereinander ausgetauscht werden und weiterhin gleiche Funktionen mit

unterschiedlichen Zeichenketten für Start- und Endblock genutzt werden; Ähnliches gilt für `InlineTextActionCommand`. Begibt man sich jedoch auf eine Ebene tiefer, unterhalb der Kommandos, so lässt sich das Prinzip schlecht anwenden. Command sind in diesem Fall sehr abstrakt gehalten, wodurch hinter der abstrakten Methode `execute()` der gleiche Sinn des Ausführens eines Befehls liegt, allerdings können die Implementierung der Kommandos unterschiedlich sein. Somit kann das Liskovsche Substitutionsprinzip nicht auf `BlockTextActionCommand` und `InlineTextActionCommand` angewandt werden, sofern man sich auf das Interface `Command` reduziert.

4.1.4 Interface-Segregation-Prinzip

Dass Interfaces auf kleinste Funktionalitäten beschränkt werden können, zeigt z. B. die UI-Komponente `SelectableEditText` im Modul `plugins-android`. Sie entstand dadurch, dass `AppCompatEditText` der Android Library keine Möglichkeiten bietet, Observer/Listener auf Veränderungen der Selektionen innerhalb des Textfeldes zu registrieren/hinzuzufügen. So wurde `SelectableEditText` als Erbe von `AppCompatEditText` implementiert und liefert ein Interface `OnSelectionChangedListener`, welches die einzige Methode `onSelectionChanged()` besitzt. Dieses Interface können Klassen implementiert, wenn sie sich selbst als Observer für die UI-Komponente registrieren wollen. Dieses Interface erfüllt damit v. a. das Single-Responsibility-Prinzip, indem `OnSelectionChangedListener` nur ein Zweck als Observer erfüllt.

4.1.5 Dependency-Inversion-Prinzip

Dependency Inversion wird durch die Verwendung des Frameworks Dagger und Android Hilt ermöglicht. Dabei ist z. B. im Modul `plugins-android` ein AppModul definiert, welches als höchstes Modul in Dagger gesehen werden kann. Davon ausgehend werden abhängig Module miteingebunden, wie `FileManagementModule`, welche sich mit Dateioperationen und -verwaltung in Datenbanken beschäftigt. `FileManagementModule` hängt wiederum von `AndroidDatabaseModule` und `AndroidDatabaseAdapterModule` ab, die jeweils in `plugins-android-database` und `adapters-android-database` implementiert sind. Die Module sind teils hierarchisch angeordnet, mit `AppModule` stets an der Spitze. Mithilfe des Dependency-

Injection-Prinzips sollen Zyklen im Abhängigkeitsgraphen von Modulen und daraus resultierenden Objekte vermieden werden.

4.2 GRASP

4.2.1 Information Expert

Beispiel ist `AndroidFileLoader`. Diese Klasse ist für das Laden, Auslesen und Speichern von Dokumenten zuständig. Beim Laden nimmt `AndroidFileLoader` eine URI entgegen, womit einerseits der Inhalt der darauf zeigenden Datei, als auch der Dateiname abgefragt werden kann. Daraus werden jeweils eine `FileReference` und ein `Document` erstellt, die das jeweilige Objekt als Rückgabewert erhält. So agiert `AndroidFileLoader` als Information Expert, da dieser mithilfe gegebener Daten weitere Daten berechnen kann.

4.2.2 Creator

Als Beispiel dient `CommandHistory`. Diese Klasse hält eine `History`, die in einem Stack `CommandHistoryEntry` mit einer Liste an Strings lagert. Sofern ein neuer `Command` via `push()` zum Verlauf hinzugefügt wird, wird dementsprechend eine `CommandHistoryEntry` erstellt. Damit sind `CommandHistory` und `CommandHistoryEntry` auch stark gekoppelt, da ohne weiteres keine eigene Instanz von `CommandHistoryEntry` der `History` zugefügt werden kann. Sie muss von `CommandHistory` erzeugt werden.

4.2.3 Controller

`RoomRecentFileRepository` zählt bspw. als Controller, da dieser für die Abfragen und Verwaltung von `FileReference` in der Datenbank zuständig ist. Sofern eine Datei geöffnet oder gespeichert wird, wird die dazugehörige Dateireferenz aktualisiert oder hinzugefügt. Dadurch entsteht eine hohe Kohäsion zwischen Controller und z. B. dem `EditorFragment`, welches mithilfe des Controllers Datenabfragen und -manipulationen durchführen muss.

4.2.4 Low Coupling

Durch z. B. die Einführung von `RecentFileRepository` als Interface zum Abrufen von zuletzt verwendeten Daten existiert hier eine geringe Kopplung zwischen demjenigen, die die Daten benötigen und demjenigen, die die Daten zur Verfügung

stellen. Klassen können Daten abrufen, ohne die Herkunft der Daten zu kennen. Wenn eine andere Datenbank verwendet werden muss, müssen nur die jeweiligen Klassen angepasst werden, die `RecentFileRepository` implementieren.

4.2.5 High Cohesion

`MarkdownRenderFragment` hat eine starke Abhängigkeit zu `MarkdownMarkupLanguageRenderer.Factory`, da ein `MarkupLanguageRenderer` ein notwendiges Kriterium ist, um innerhalb des `MarkdownRenderFragments` einen Markdown-Text zu parsen und darzustellen. Sonst ist keine weitere Verwendung des `MarkdownMarkupLanguageRenderer` zu erkennen. Demnach ist auch ersichtlich, dass diese Funktionen zusammenarbeiten müssen. Weitere Beispiel sind `EditorFragment` und `EditorViewModel`. Jede erfüllt eigene Funktionalität; `EditorFragment` ist Controller einer View und `EditorViewModel` das zugehörige `ViewModel`, um Daten zur Darstellung über UI zu beobachten oder zu aktualisieren. Alleinstehend würden beide Klassen keinerlei Funktionalität bringen.

4.2.6 Polymorphismus

Polymorphismus wird in diesem Projekt auch verwendet, um die verschiedenen Variationen von `Commands` (genauer `BlockTextActionCommand` und `InlineTextActionCommand`) zu realisieren. Die Implementierung sind in der Klasse `MarkdownTextActionCommands` im Domain-Modul wiederzufinden. Sie unterscheiden sich in den zu verwendeten Zeichenketten und dem Aufbau des Start- und Endblocks. Jedoch ist die Umsetzung der Veränderung, die auf den Textinhalt des Texteditors angewandt wird, etwa gleich.

4.2.7 Pure Fabrication

Pure Fabrication ist in der Domainschicht des Projekts zu finden. Dort wird ein Repository `RecentFileRepository` als Interface definiert, welches die nötigen Methoden für Datenmanipulationen von `FileReference` in Datenbanken festlegt. Da allerdings dieses Repository in ihrer Funktionalität nicht in der Domainschicht implementiert wird, besteht hier eine starke Trennung zwischen Domäne und Technologien. So ist in diesem Projekt ein Repository in `plugins-android-database` implementiert, welches Android Room verwendet, um Referenzen auf die zuletzt verwendeten Dateien zu persistieren.

4.2.8 Indirection

Als klassisches Beispiel für Indirection ist eine Art Modell-View-Controller-Muster in Android. Views werden über XML-Dateien definiert, die in Activities oder Fragments als Layout genutzt werden. Diese Klassen agieren zusätzlich als Controller, die z. B. Daten an ViewModels auslagern. Als Beispiel im Projekt dient `FileExplorerRecentListFragment`, welches von `EditorActivity` genutzt wird. Dieses Fragment besitzt `FileExplorerRecentListViewModel`, welches für die Beschaffung der darzustellenden Daten zuständig ist. Das Fragment kann sich als Observer auf die zur Verfügung stehenden Daten des ViewModels eintragen, die bei Veränderungen aktualisiert werden. Darüber hinaus das Fragment selbst über das ViewModel Veränderungen an den bestehenden Datensätzen durchführen. Jegliche Veränderungen verursachen in der Regel eine Aktualisierung der View, sofern der zugegehörige Controller (hier `FileExplorerRecentListFragment`) für die zu beobachtenden Daten als Observer eingetragen ist.

4.2.9 Protected Variations

Diese sind vor allem in den Variationen von `InlineTextActionCommand` und `BlockTextActionCommand` zu sehen. Sie sind fundamental gleich, existieren aber in unterschiedlichen Versionen wie `BoldCommand` in `MarkdownTextActionCommands`. Zudem spiegeln sie keinen Zustand wider, sondern führen immer die gleiche Logik auf einen Texteditor aus.

4.3 DRY

Dieses Prinzip lässt sich v. a. im Bereich des Kommadoverhaltensmusters (siehe Kapitel 6) beobachten. Durch die Definition von `BlockTextActionCommand` und `InlineTextActionCommand` sollen wiederholte Implementierungen von Block Actions und Inline Actions vermieden werden, da die Logiken für die jeweiligen Markdown-Spezifikationen identisch sind.

Ein weiteres Beispiel ist der `FileReferenceDataUpdater`, der in Kapitel 2.2.5 näher beschrieben ist. Hierbei gilt es, an verschiedenen Stellen im Code wiederholte Neuinstanziierungen von Value Objects o. Ä. zu vermeiden. Somit kann ein „Schrotflintenkugel“-Effekt vermieden werden, bei dem für kleinere Änderungen in Klasse viel mehr Stellen im gesamten Code angepasst werden müssen.

5 REFACTORING

5.1 MAPPER FÜR FILEENTITY UND FILEREFERENCE

Ursprünglich wurden solche Mappingfunktionen als eine Art „statische“ Erweiterung implementiert, um `FileEntity` in `FileReference` und umgekehrt zu überführen. Ein Beispiel kann im Commit 0a3eb6f99e452a98ce6d4177a58759a38bd22e85 eingesehen werden. Das hat den Nachteil, dass diese Mappingfunktionalität z. B. in Unit Tests schwer mockbar sind.

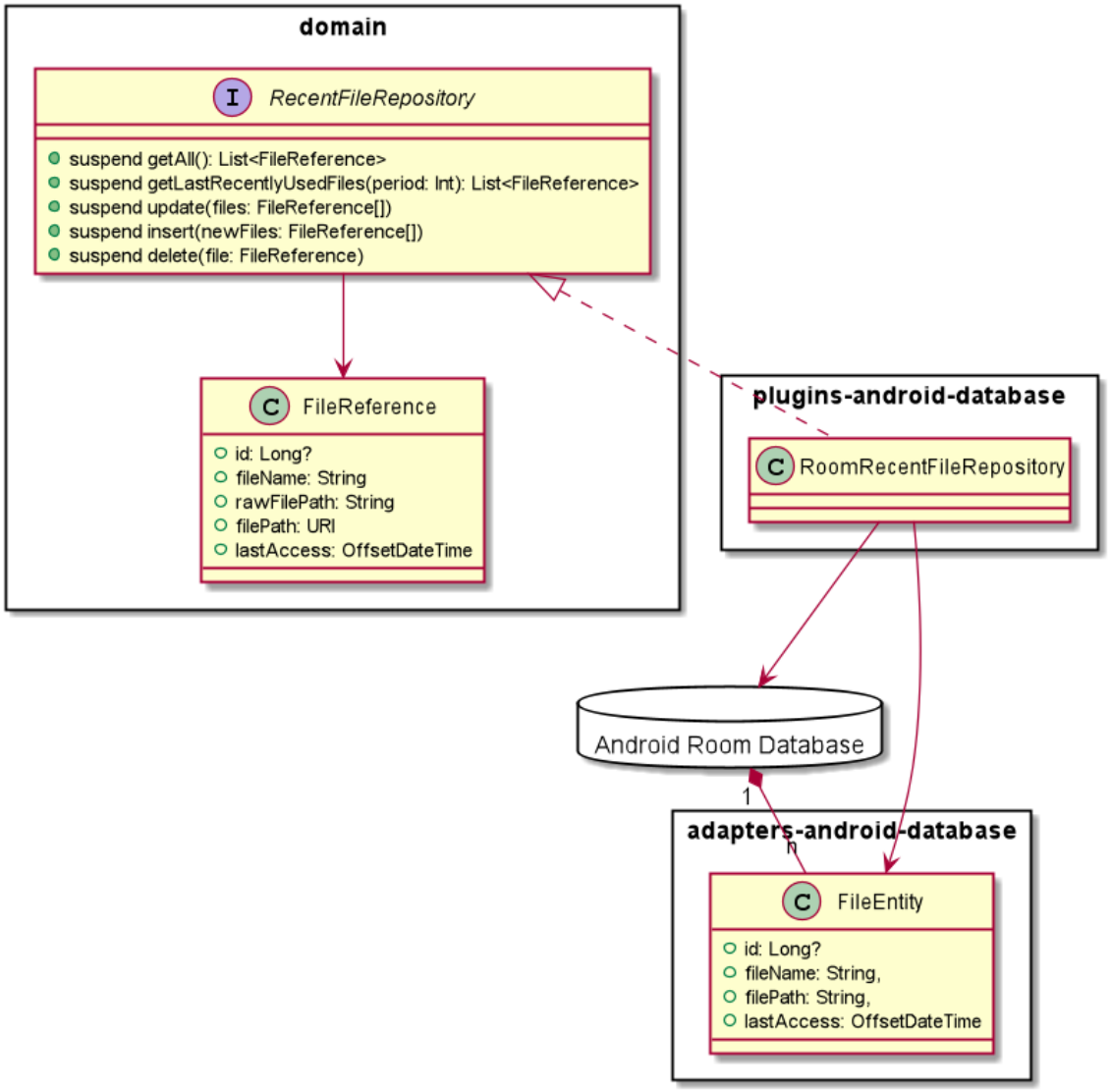


Abbildung 8: Klassendiagramm vor dem Refactoring mit DataMapper

Deshalb wird innerhalb der Domainschicht ein allgemeines Interface `DataMapper` definiert, welches als primitive Datentypen ein **S**ource und ein **T**arget annehmen. Ein

Mapping ist jeweils mit den Methoden `toSourceType()` und `toTargetType()` möglich.

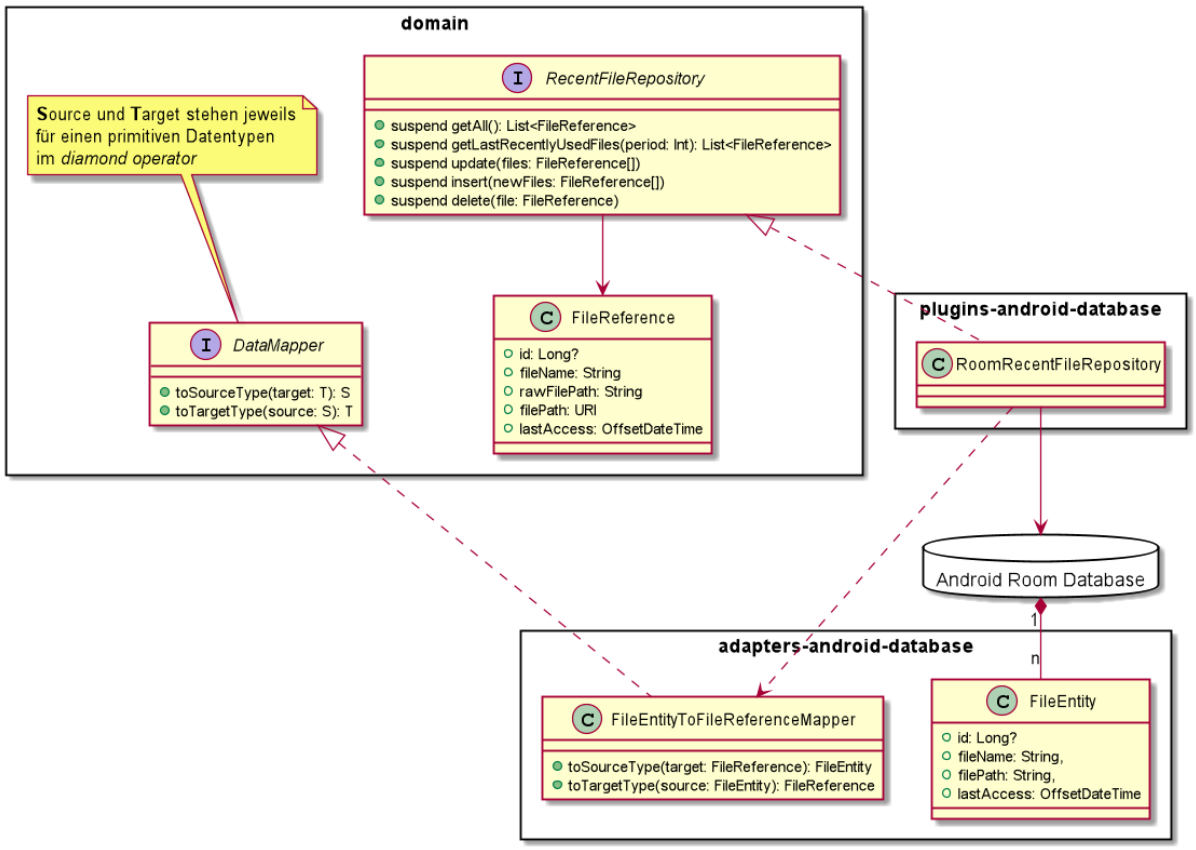


Abbildung 9: Klassendiagramm nach Refactoring mit DataMapper

In `adapters-android-database` wird anschließend eine Klasse `FileEntityToFileReferenceMapper` erstellt, die das Interface `DataMapper` implementiert. Für spätere Dependency Injection wird außerdem eine Provider-Methode (Dagger-Kontext) im Gradle-Modul zugehörigen `AndroidDatabaseModule` implementiert. Dieses Modul wird in `plugins-android` in `AppModule` miteingeschlossen. Durch Dependency Injection kann nun eine Instanz des `FileEntityToFileReferenceMapper` dem `RoomRecentFileRepository` übergeben werden, um `FileEntity` und `FileReference` umzumappen.

5.2 Änderung des Entwurfsmusters für TextActions

Am Anfang der Entwicklung wurden die Text Actions so konzipiert, dass `BlockTextAction` und `InlineSpanTextAction` von der abstrakten Klasse `TextAction` erben. Diese abstrakte Klasse gibt vor, die Funktion `apply(textEditor: TextEditor)` zu implementieren. Im Idealfall sollte in dieser

Funktion der Anwendungsprozess einer Textformatierung mithilfe des übergebenen Texteditors stattfinden. Daher muss der Controller, der die Textverarbeitung steuert, das Interface `TextEditor` implementieren.

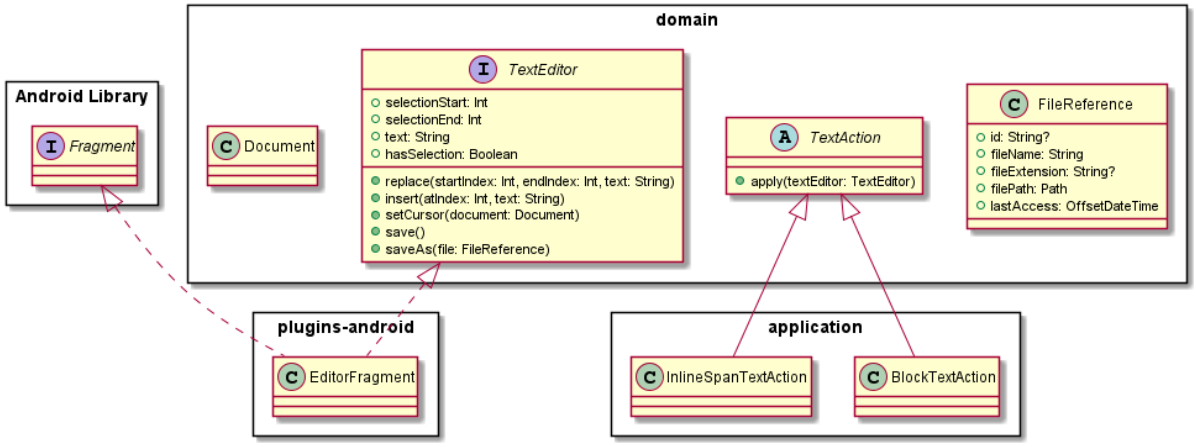


Abbildung 10: Klassendiagramm mit TextAction im Besuchermuster

Da die Interaktion der Android App überwiegend im Controller einer View implementiert wird, hat das Interface `TextEditor` außer zur Umsetzung des oben genannten Visitor-Patterns keinerlei Funktion.

Gängiger ist es, ein Kommando-Verhaltensmuster zu implementieren, da sie z. B. Benutzerbefehle widerspiegeln können. Mit diesem Muster können die zugehörigen Textformatierungskommandos besser und von Controller einer View isoliert getestet werden. Damit entfällt die Notwendigkeit, dass die Unit Test in einer emulierten Android-Umgebung getestet werden müssen. Diese Art von Tests ist mit dem Aufwand des Hochfahrens einer Android-Umgebung notwendig und ggf. ressourcenhungrig. Eine genauere Beschreibung der Verwendung des Kommando-Verhaltensmusters kann in Kapitel 6 Entwurfsmuster eingesehen werden.

6 ENTWURFSMUSTER

Wie bereits in Kapitel 5.2. erwähnt wird das Kommandomuster verwendet. Das Muster wird gewöhnlich für Eingabe über grafische Benutzeroberflächen genutzt. So werden Benutzerbefehle in ausgesonderten Kommando-Klassen und -Objekte abgekapselt. Der Vorteil gegenüber des Besuchermusters ist u. a. eine einfachere Implementierung komplexer Logik wie z. B. von Rückgängig-Mechanismen, Warteschlangen zur Bearbeitung von Kommandos und Logging. Ein möglicher Nachteil ist die steigende Anzahl an Klassen, die erstellt werden müssen, um verschiedene Kommandos zu implementieren.

Als Basisklasse dient die abstrakte Klasse `Command`, die eine abstrakte Methode `execute()` besitzt, über die ein Kommando ausgeführt wird. Um nun die Block Actions und Inline Actions zu realisieren, wurden die Klassen `BlockTextActionCommand` und `InlineTextActionCommand` erstellt, die einen `TextEditor` und eine Zeichenkette entgegennehmen. `BlockTextActionCommand` erstellt bei Ausführung über `execute()` einen Block, der oben und unten jeweils mit der übergegebenen Zeichenkette umrandet wird, `InlineTextActionCommand` fügt durch definierte Start- und Endblöcke einen zusammengesetzten Block an die Stelle, wo sich der Cursor im Textfeld des Texteditors befindet.

Mithilfe `BlockTextActionCommand` und `InlineTextActionCommand` kann nun versucht werden, verschiedene Markup Languages zu implementieren. In der Klasse `MarkdownTextActionCommand` im Domain-Gradle-Modul ist eine beispielhafte Implementierung von Textformatierungskommandos gemäß Markdown-Spezifikation zu finden.

Der Zustand vor der Anwendung des Kommandoverhaltensmusters kann in der Abbildung 10 eingesehen werden.

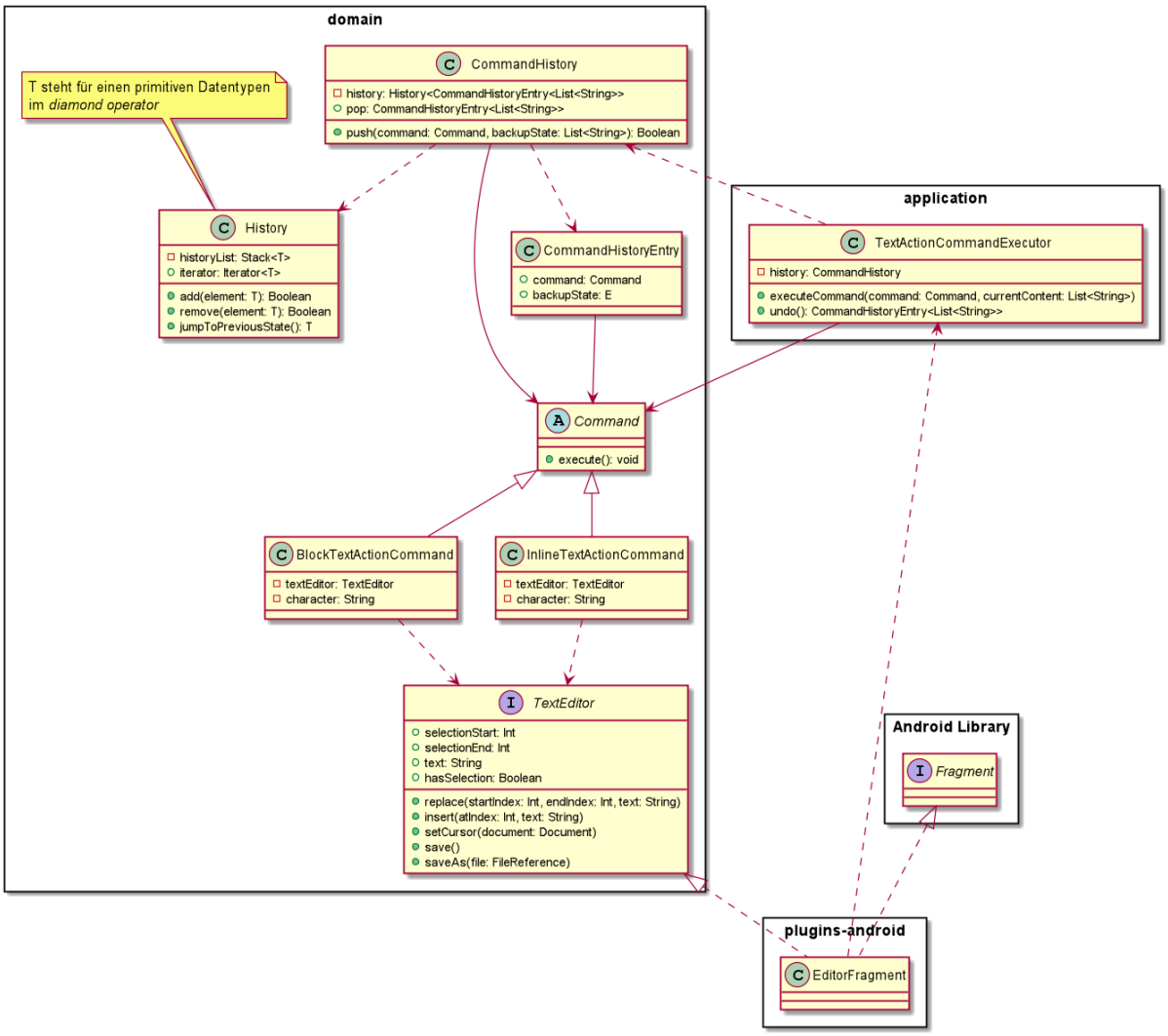


Abbildung 11: Refactoring nach Anwendung des Kommandomusters für Text Actions

VERWEISE

[1] J. Gruber, „Daring Fireball: Markdown Syntax Documentation,“ [Online]. Available: https://daringfireball.net/projects/markdown/.

TEXTBLOCKVERZEICHNIS

Textblock 1: Codeblock 3
Textblock 2: Zitatblöcke 3
Textblock 3: Unnummerierte Liste mit Einrückung 4
Textblock 4: Nummerierte Liste mit Einrückung 4
Textblock 5: Tabelle 4
Textblock 6: Bilder 4

ABBILDUNGSVERZEICHNIS

Abbildung 1: UML-Diagramm "TextEditor" 2
Abbildung 2: UML-Diagramm "Command" 3
Abbildung 3: UML-Diagramm "Document" 4
Abbildung 4: UML-Diagramm "FileReference" 4
Abbildung 5: Konzeptionelles Klassendiagramm von "DocumentAggregate" 5
Abbildung 6: Klassendiagramm mit Beziehungen von "RecentFileRepository" 6
Abbildung 7: Schematische Darstellung der Clean Architecture 7
Abbildung 8: Klassendiagramm vor dem Refactoring mit DataMapper 14
Abbildung 9: Klassendiagramm nach Refactoring mit DataMapper 15
Abbildung 10: Klassendiagramm mit TextAction im Besuchermuster 16
Abbildung 11: Refactoring nach Anwendung des Kommandomusters für Text Actions 18