Morik Password Manager

Programmentwurf

von

Moritz Gutfleisch

und

Erik Zimmermann

Abgabedatum:	30. April 2022
Bearbeitungszeitraum:	04.10.2021 - 30.04.2022
Matrikelnummer, Student:	0000000, Moritz Gutfleisch
Matrikelnummer, Student:	2152543, Erik Zimmermann
Kurs:	TINF19B1
Gutachter der Dualen Hochschule:	Daniel Lindner

Abstract

- English -

This is the starting point of the Abstract. For the final bachelor thesis, there must be an abstract included in your document. So, start now writing it in German and English. The abstract is a short summary with around 200 to 250 words.

Try to include in this abstract the main question of your work, the methods you used or the main results of your work.

Abstract

- Deutsch -

Dies ist der Beginn des Abstracts. Für die finale Bachelorarbeit musst du ein Abstract in deinem Dokument mit einbauen. So, schreibe es am besten jetzt in Deutsch und Englisch. Das Abstract ist eine kurze Zusammenfassung mit ca. 200 bis 250 Wörtern.

Versuche in das Abstract folgende Punkte aufzunehmen: Fragestellung der Arbeit, methodische Vorgehensweise oder die Hauptergebnisse deiner Arbeit.

Inhaltsverzeichnis

Abkürzungsverzeichnis

SQL Structured Query Language

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

2 Clean Architecture

2.1 Plugins

2.1.1 Datenbank

Die Datenbank stellt eines der Plugins dar. Als Technologie wurde hier eine SQLite3 Datenbank verwendet. Die konkrete Implementierung der Schnittstelle zur SQLite-Datenbank befindet sich in der Klasse *SQLiteDatabase*. Diese Klasse erbt von der abstrakten Klasse *AbstractSqlDatabase*, die sich in der Adapterschicht befindet. Hierdurch wird eine Dependency Inversion erreicht, da nun eine *AbstractSqlDatabase* von anderen Klassen verwendet werden kann, um SQL-Befehle auf der Datenbank auszuführen, statt eine konkrete *SQLiteDatabase* zu verwenden, was die Abhängigkeit von innen nach außen laufen lassen würde.

2.1.2 Verschlüsselung

Ein weiteres Plugin ist die Verschlüsselungsbibliothek. Als solche wurde Cryptopp[1] verwendet. Die abstrakte Klasse *Cipher* definiert die Schnittstelle in der Applikationsschicht, die das Plugin implementieren muss. Eine konkrete Implementierung dieser Schnittstelle befindet sich in der Klasse *CBC_Cipher*. Dadurch findet auch hier eine Dependency Inversion statt, da *Cipher* verwendet werden kann, um Verschlüsselung zu verwenden, statt eine Dependency auf eine konkrete Klasse der Pluginschicht zu brauchen. Die Schnittstelle muss lediglich einen String ver- und entschlüsseln können unter Angabe des Klar- bzw. Geheimtextes und eines Schlüssels. *CBC_Cipher* implementiert dies für die Chiffren, die im *BLOCK* enum aufgezählt sind. Momentan handelt es sich dabei um AES und Serpent, neue Chiffren können allerdings leicht hinzugefügt werden, solange sie von Cryptopp unterstützt werden. Diese werden als Block-Chiffren mit Cipher Block Chaining (CBC) als Betriebsmodus verwendet. Um andere Betriebsmodi, oder Nicht-Blockchiffren zu

[1] https://cryptopp.com/

verwenden, müssten weitere Implementierungen des *Cipher* Interfaces hinzugefügt werden. Dies ist ohne Weiteres möglich, solange ein *String* zür Schlüsselherleitung hinreichend ist.

2.2 Adapter

2.2.1 Datenbank

Die Klasse *DatabaseInterface* implementiert die eigentliche Funktionalität in Form der SQL-Anweisungen und führt diese über eine konkrete Implementierung der *AbstractSqlDatabase* in der Datenbank aus. Somit geht keine Funktionalität verloren, wenn das Plugin durch eine andere Datenbanktechnologie ausgetauscht wird. Die Umsetzung der Funktionalität als SQL-Anweisungen bedeutet jedoch, dass ein Austausch des Plugins nur ohne Weiteres möglich ist, wenn die neue Datenbank ebenfalls eine SQL-Datenbank ist. Handelt es sich bei der neuen Datenbank jedoch beispielsweise um eine noSQL-Datenbank, so muss die Funktionalität, also die Abfragen, angepasst werden. Der Klasse *DatabaseInterface* wird im Konstruktor eine *AbstractSqlDatabase* übergeben, was die Dependency Injection umsetzt. Zusätzlich erbt die Klasse *DatabaseInterface* von der abstrakten Klasse *AbstractDatabaseInterface*, die sich im Kern befindet, sodass die Abhängigkeit vom Adapter auf den Kern zeigt. Dies setzt also die Dependency Inversion um und ermöglicht Klassen im Kern eine Abhängigkeit auf das AbstractDatabaseInterface zu haben statt auf eine konkrete Implementierung davon.
Auf die Benutzung von Prepared Statements wurde innerhalb des Adapters verzichtet, da die Datenbank lokal ist und nur der Benutzer Befehle auf ihr ausführt. Würde die Datenbank über eine öffentliche Schnittstelle angesteuert werden, so wäre dies nicht zu vernachlässigen.

3 Programming Principles

4 Domain Driven Design

4.1 Ubiquitous Language

Die Analyse der Ubiquitous Language ergab die zuvor genannten Programmteile, deren Bezeichnungen mit denen anderer Passwortmanager verglichen werden. KeePass unterscheidet in der Ubiquitous Language nicht zwischen den zwei von uns identifizierten Arten von Passwörtern, nämlich PlaintextPassword und EncryptedPassword. Stattdessen werden diese zwei Umstände meist mit Adverben beschrieben (ßensitive data is stored encryptedly", "make sensitive data available unencryptedly") oder mithilfe mehrerer Substantive ("passwords as plain-text"). Da wir im Source Code weder mit Adverben noch mit unnötig langen Konstrukten aus mehreren Substantiven arbeiten wollen, wir jedoch trotzdem eine Unterscheidung der beiden Zustände eines Passworts benötigen, haben wir uns für die beiden genannten Varianten entschieden. Was das Entry angeht, so verwenden wir den gleichen Begriff wie KeePass (1Password zieht hier Item vor). Was die EntryId angeht, so verwenden wir keine UUID, da wir nur Entries verwalten und demnach keine eindeutige Kennung über mehrere Tabellen hinweg benötigen. Das Präfix Ëntry"vor der Id dient dem besseren Nachvollziehen von was es die Id ist. Das Gleiche gilt für das Präfix des EntryName. Hier haben wir uns für Name statt wie KeePass für Title entschieden, da wir dies für eindeutiger halten. Außerdem haben wir uns gegen User Name entschieden und haben stattdessen Login gewählt als Begriff, da unser Passwort Manager darauf abzielt dem Benutzer beim Anmelden zu helfen. Aus diesem Grund kann im Feld Login die konkrete Zeichenkette gespeichert werden, die der Benutzer zum Anmelden braucht, sei es tatsächlich der Benutzername oder aber die Email Adresse. Entsprechend uneingeschränkt sollte auch die Bezeichnung dieses Feldes sein, was bei User Name nicht der Fall ist. 1Password nennt den Ort, an dem die Passwörter gespeichert werden den Vault. Wir haben uns stattdessen für EntryRepository entschieden, um zum einen klar zu machen, dass es sich um einen Aufbewahrungsort handelt und zum anderen eindeutig festzulegen, dass dies der Aufbewahrungsort speziell für Entries ist. Dies beseitigt sämtliche Fragen, was genau dort aufbewahrt wird. Beim PasswordGenerator jedoch sind wir uns einig mit der Bezeichnung von KeePass. Weder KeePass noch 1Password benennen

Domänenexperte (KeePass/1Password)	Implementierung (Morik)
Password	PlaintextPassword
Password	EncryptedPassword
Entry/Item	Entry
UUID	EntryId
Title	EntryName
User Name	Login
Vault	EntryRepository
Password Generator	PasswordGenerator
?	PasswordEncrypter
?	PasswordDecryper

Tabelle 4.1: Ubiquitous Language Gegenüberstellung

die Teile ihrer Software, die die Daten ver- und entschlüsseln. Stattdessen ist die Rede von ëncrypted dataünd "decrypted data". Da Passwörter die einzigen Felder sind, die von uns ver- und entschlüsselt werden, haben wir uns dafür entschieden die dafür zuständigen Programmteile mit PasswordEncryptor und PasswordDecryptor zu bezeichnen, sodass wir trotzdem die von KeePass und 1Password verwendeten Wortstämme verwenden, daraus jedoch Substantive machen.

Um die Ubiquitous Language des Programms auch außerhalb des Source Codes zu verwenden, haben wir ebenfalls die Spalten der Datenbank an die festgelegten Begriffe angepasst. Dies kann in Abbildung 4.1 beobachtet werden.

```
EntryId  EntryName  Login                      EncryptedPassword
-------  ---------  -------------------------  ------------------------
1        TestEntry  max.mustermann@beispiel.de  SafeAndEncryptedPassword
```

Abbildung 4.1: Datenbank-Spaltennamen nach Ubiquitous Language

4.2 Value Objects

Das erste Value Object ist das *PlaintextPassword*, welches das vom Benutzer eingegebene abzulegende Passwort beinhaltet bevor es verschlüsselt wird beziehungsweise nachdem es entschlüsselt wird. Es handelt sich dabei um ein Value Object, da es sich um das gleiche PlaintextPassword handelt wenn der Benutzer zwei mal die gleiche Zeichenkette eingibt. Das *EncryptedPassword*, das das Passwort beinhaltet nachdem es verschlüsselt ist, gilt

ebenfalls als Value Object. Dies gilt aus dem gleichen Grund wie beim PlaintextPassword, also dass die gleiche Zeichenkette auch die Gleichheit des Objekts impliziert. Außerdem gilt der *EntryName* als Value Object, der den Namen, welchen der Benutzer für den Eintrag vergibt, speichert. Hierbei handelt es sich ebenfalls um ein Value Object, da die gleiche eingegebene Zeichenkette bedeutet, dass es sich um den gleichen Namen für den Eintrag handelt. Auch der vom Benutzer optional hinzugefügte *Login* eines Entrys gilt als Value Object, ebenfalls aus dem Grund dass die gleiche Zeichenkette bedeutet, dass das Objekt das Gleiche ist. Zu guter Letzt gilt auch die *EntryId* als Value Object, die die ID des Entrys speichert. Hier gilt, dass der gleiche Integer Wert schlussfolgern lässt, dass es das gleiche Objekt der Klasse EntryId ist.

4.3 Entities

Ein *Entry* beschreibt einen konreten Eintrag in der Datenbank. Der Entry beinhaltet die ID des Eintrags, den Namen des Eintrags, einen optionalen Login und das verschlüsselte Passwort. Man stelle sich Folgendes vor. Der Benutzer löscht einen Entry und erstellt danach einen neuen. Bei dem neuen Entry vergibt er den gleichen Namen, Login und das gleiche Passwort wie bei dem zuvor gelöschten Entry. Handelt es sich deshalb um den selben Entry wie der vorherige? Nein, denn der wurde gelöscht. Darum ist der Entry eine Entity und kein Value Object.

4.4 Aggregate

Es gibt genau ein Aggregat. Dieses Aggregat beinhaltet einen Entry, eine EntryId, einen EntryName, einen Login und ein EncryptedPassword. Diese Teile sind als Aggregat zu betrachten, da sie immer gemeinsam von Interesse sind. Ein Benutzer möchte nicht einfach irgend ein Passwort wissen ohne den zugehörigen Namen des Eintrags zu kennen. Die Aggregat-Root ist dabei der Entry, wobei über die EntryId des Entrys auf das jeweilige Aggregat zugegriffen wird.

4.5 Repositories

Da es nur ein Aggregat gibt, gibt es ebenfalls genau ein Repository. Dieses Repository ist die *EntryRepository* und stellt den Zugriff auf die Datenbank dar, die die Einträge des Benutzers persistiert. Das EntryRepository liefert demnach vorhandene Entries zurück, speichert neue Entries, löscht nicht mehr benötigte und passt Werte eines Entrys an neue Werte an.

4.6 Domain Services

Der Generator für neue Passwörter (*PasswordGenerator*), sowie die Verschlüsselung eines PlaintextPasswords (*PasswordEncryptor*) und die Entschlüsselung eines EncryptedPasswords (*PasswordDecryptor*) werden als Domain Services realisiert.

5 Unit Tests

Die Unit Tests von Morik wurden so geschrieben, dass sie uns beim Entwickeln neuer Funktionalitäten helfen. Als Test-Framework wird dabei *GoogleTest* verwendet. Um eine gute Übersicht über die Tests zu haben, sind diese genau in der gleichen Ordnerstruktur angeordnet wie der Produktivcode, nur eben im *tests*-Ordner. Im Folgenden wird genauer auf die Umsetzung der ATRIP-Regeln, das Messen der Code Coverage und den Einsatz von Mock-Objekten eingegangen.

5.1 ATRIP-Regeln

Diese Regeln definieren eine Reihe von Eigenschaften, die gute Unit Tests erfüllen sollten. "ATRIP" steht dabei für automatic, thorough, repeatable, independent und professional. Automatic, repeatable und independent sind dabei harte Regeln, was bedeutet, dass diese Eigenschaften messbar sind, während thorough und professional weiche Regeln sind, sodass diese nur bedingt messbar sind. Im Folgenden wird näher auf die einzelnen Regeln und deren Erfüllung seitens Morik eingegangen.

5.1.1 Automatic

Die Tests laufen alle eigenständig ab. Es gibt also keinen Test, der bspw. eine Eingabe erwartet. Außerdem prüfen die Tests ihre Ergebnisse selbst, indem Assertions verwendet werden. Dadurch gibt es eine deutliche Übersicht, ob alle Tests erfolgreich ausgeführt wurden, statt dass die Auswertung, ob der Test das richtige Ergebnisse produziert hat, manuell durchgeführt werden muss.

5.1.2 Thorough

Diese Regel besagt, dass alles Notwendige des Systems getestet ist. Dies erfüllt Morik nicht, da hierfür mehr Tests benötigt werden. Wir haben uns beim Schreiben der Tests

jedoch trotzdem auf das Testen der wichtigen Funktionalitäten fokussiert. Demnach testen die bestehenden Tests hauptsächlich das Ver- und Entschlüsseln der Passwörter (bspw. bei den *encryptor_tests*), sowie das Lesen dieser aus der Datenbank (bspw. bei den *repository_tests*). Dies sind die wesentlichen Bestandteile eines Passwort Managers, jedoch werden auch hiervon nicht alle Aspekte getestet. Auf eine Einschätzung wie "gründlich" die Tests tatsächlich sind, wird in der Sektion "Code Coverage" eingegangen.

5.1.3 Repeatable

Unsere Tests sind wiederholbar. Sie können also immer wieder ausgeführt werden und liefern immer das gleiche Ergebnis, sofern keine Änderungen am Code vorgenommen wurden. Dies liegt daran, dass die Tests nicht von äußeren Umständen abhängen. Weitere Einblicke wie wir die Tests von äußeren Abhängigkeiten lösen, werden in der Sektion "Mocks" gegeben.

5.1.4 Independent

Jeder unserer Tests testet eine dedizierte Funktionalität wie bspw. das Lesen eines Eintrags aus der Datenbank oder das Ver- und Entschlüsseln von Passwörtern. Hierbei wird das Ver- und Entschlüsseln, wie man in *crypto_tests* sieht, als eine Funktionalität aufgefasst, da man ein Passwort nur entschlüsseln kann wenn es zuvor verschlüsselt wurde und das Verschlüsseln eines Passworts ohne die Intention es wieder zu entschlüsseln sinnfrei ist. Da jeder Test genau eine Funktionalität testet, gibt es keine Abhängigkeiten zwischen den Tests, weshalb auch die Ausführungsreihenfolge irrelevant ist.

5.1.5 Professional

Der Testcode von Morik ist professionell geschrieben. Dies bedeutet, dass er leicht verständlich ist aufgrund von sprechenden Bezeichnern und dem Befolgen der AAA-Normalform. Außerdem werden keine unrelevanten Aspekte der Applikation getestet, sondern wir fokussieren uns, wie bereits bei "Thorough" genannt, auf das Testen der wichtigsten Funktionalitäten.

5.2 Code Coverage

Element	Line Coverage, %	Branch Coverage, %
adapters	100% files, 21% lines covered	9% branches covered
application	90% files, 52% lines covered	28% branches covered
plugins	75% files, 60% lines covered	32% branches covered

Abbildung 5.1: Line Coverage und Branch Coverage Messung

Wie zuvor bereits genannt, ermöglicht die Code Coverage eine Einschätzung wie ”gründlich” die Tests tatsächlich sind, indem gemessen wird wie viel Code beim Testen durchlaufen wird im Vergleich zu wie viel Code insgesamt in der kompletten Applikation durchlaufen werden kann. Hierbei werden zwei Metriken besonders häufig verwendet. Zum einen gibt es die Line Coverage, die die ausgeführten Codezeilen zählt, während es zum anderen die Branch Coverage gibt. Diese zählt die ausgeführten Verzweigungsmöglichkeiten. Dabei ist letztere Metrik aussagekräftiger für die tatsächliche ”Gründlichkeit” der Tests, während die Line Coverage allerdings leichter zu messen ist. Dass die Branch Coverage aussagekräftiger ist, kommt vor Allem daher wie einfach es ist die Line Coverage zu erhöhen. Beispielsweise könnte eine Funktion, welche tatsächlich getestet wird, so umgeschrieben werden, dass sie mehr Zeilen Code einnimmt. Dadurch steigt die Line Coverage, da der getestete Code relativ gesehen nun einen größeren Anteil der gesamten Codebasis einnimmt als zuvor. Die Branch Coverage jedoch steigt nicht, da sich nichts daran geändert hat welche Funktionen aufgerufen werden oder welche Verzweigungen innerhalb der Funktion tatsächlich durchlaufen werden.
Wie in Abbildung 5.1 gezeigt, erreicht Morik mit den Tests eine Line Coverage von 44.3%, während eine Branch Coverage von nur 23% erreicht wird. Dies zeigt erneut, dass es einfacher ist eine hohe Line Coverage zu erreichen, als eine hohe Branch Coverage. Ermittelt werden diese Zahlen durch das Ausführen der Tests mit Coverage mithilfe von JetBrains IDE CLion.

Zahlen und screenshot anpassen

5.3 Mocks

Einige der Klassen von Morik sind abhängig von anderen Klassen, um richtig funktionieren zu können. Um diese Klassen jedoch trotzdem testen zu können, ohne vorher alle

Abhängigkeiten konstruieren zu müssen, werden an ausgewählten Stellen Mock-Objekte eingesetzt. Für das Erzeugen der Mock-Objekte wird *gmock* verwendet, das in GoogleTest enthalten ist. So kann in der Klasse *AbstractDatabaseInterfaceMock* bspw. das Mocken eines AbstractDatabaseInterface beobachtet werden, sodass in *repository_tests* die Entry-Repository getestet werden kann, die die Übergabe eines AbstractDatabaseInterface im Konstruktor erwartet. Ein weiteres Mock-Objekt ist das *AbstractSqlDatabaseMock*, das eine AbstractSqlDatabase mockt. Dies wird verwendet, um in *databaseInterface_tests* den DatabaseInterface-Adapter testen zu können, der wiederum die Übergabe einer AbstractSqlDatabase im Konstruktor erwartet.

6 Entwurfsmuster

7 Refactoring