Morik Password Manager

Programmentwurf

von

Moritz Gutfleisch

und

Erik Zimmermann

Abgabedatum:	30. April 2022
Bearbeitungszeitraum:	04.10.2021 - 30.04.2022
Matrikelnummer, Student:	9540703, Moritz Gutfleisch
Matrikelnummer, Student:	2152543, Erik Zimmermann
Kurs:	TINF19B1
Gutachter der Dualen Hochschule:	Daniel Lindner

Inhaltsverzeichnis

1 Einleitung

Morik ist ein Passwort Manager, der die gespeicherten Passwörter verschlüsselt in einer Datenbank aufbewahrt. Zusätzlich zu den Passwörtern wird ein Name des Eintrags gespeichert sowie ein optionaler Login. Neue Einträge können sehr einfach hinzugefügt werden. Von jedem vorhandenen Eintrag kann der Benutzer das Passwort im Klartext abfragen oder aber den Eintrag bearbeiten sowie löschen. Möchte der Benutzer ein neues Passwort für einen Eintrag vergeben, so kann dies entweder von Hand getan werden oder der Benutzer verwendet den Passwortgenerator von Morik, wodurch zufällige Passwörter variabler Länge generiert werden. Dies steigert die Sicherheit der Passwörter im Vergleich zur manuellen Eingabe.

Um die Applikation bauen zu können, ist es notwendig, einige Abhängigkeiten zu installieren. Dafür kann das „apt_install_script“ mit Superuser-Rechten ausgeführt werden, das die benötigten Pakete automatisch installiert. Anschließend kann das „run_tests“-Skript ausgeführt werden, das zuerst die Applikation baut und anschließend direkt die Tests ausführt. Somit sind mögliche Fehler bspw. durch Inkompatibilitäten direkt nach dem Bauen durch die Tests identifizierbar.

In den folgenden Kapiteln wird auf die einzelnen Kriterien des Programmentwurfs mit Bezug zum Code von Morik eingegangen.

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.

2.1.3 User Interface

Dieses Plugin Implementiert eine Benutzerschnittstelle für das Programm. Eine solche Schnittstelle ermöglicht Interaktion mit dem Password Manager für einen Benutzer, durch Umsetzung der zentralen Schleife. Bei der Schnittstelle handelt es sich um ein *CommandLineInterface*, welches das *AbstractUserInterface* Interface aus der Applikationsschicht implementiert. Auch hier findet folglich eine Dependency-Inversion statt, da das Plugin abhängig vom Interface aus der Applikationsschicht ist.

2.1.4 Pseudo-Zufallszahlen-Generator

Die Implementierung des *PasswordGenerator*s braucht eine Möglichkeit (Pseudo-)Zufallszahlen zu generieren. Das Interface *PseudoRandomNumberGenerator* ist die Schnittstelle in der Applikationsschicht, die definiert wie ein solcher Generator implementiert werden muss. Konkrete Implementierungen werden von Plugins vorgenommen, indem das Interface vom Plugin implementiert wird. In unserem Fall wird dies von der *MersenneTwister* Klasse übernommen. Durch die invertierte Abhängigkeit, könnte jedoch jederzeit auf eine andere Implementierung umgestellt werden.

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 dessen.
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.

2.2.2 User Interface

Die Klasse *UIDataHelper* implementiert die Funktionalität, die ein Benutzer zur Verfügung hat. Namentlich handelt es sich dabei um das Anlegen und Bearbeiten von Passwörtern. Diese Klasse ist separat vom entsprechenden Plugin, damit die Funktionalität nicht abhängig ist, von spezifischen Plugins. Neue User Interfaces können also implementiert/hinzugefügt werden, ohne diese Funktionen erneut zu implementieren.

2.3 Applikationskern

Der Kern des Programmes, befindlich im application Ordner, ist zuständig für die Umsetzung der Domänenkonzepte und enthält zusätzlich Pure Fabrication Klassen (bspw. *InstanceManager*) und Interfaces, welche von den Plugins implementiert werden (siehe oben). Der Applikationskern hat keine externen Abhängigkeiten, durch konsequente Anwendung der Dependency-Inversion.

3 Programming Principles

In diesem Kapitel wird unser Projekt im Bezug auf die SOLID, DRY und GRASP Prinzipien evaluiert.

3.1 SOLID

3.1.1 Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip besagt, dass Klassen und Funktionen stets nur einen Grund zur Änderung haben sollen. Bei Morik gilt dies für alle Klassen. Ein einfaches Beipiel ist die *EntryFactory*, welche Entries erstellt und nur verändert werden müsste, um dies anders zu erreichen. Das Prinzip ist ebenfalls erfüllt vom *Cipher* Interface, welches sowohl ver- als auch entschlüsseln kann. Dies sind zwar unterschiedliche Operationen, jedoch ergibt es keinen Sinn die eine zu ändern, ohne die andere gleichermaßen anzupassen. Der einzige Grund die Klasse zu verändern, ist um die Verschlüsselung zu ändern. Ein Beispiel des Single-Responsibility-Prinzips bei Methoden sind die des *Cipher* Interface. Ver- und Entschlüsselung sind hier in Methoden getrennt, statt über eine Flag, o.ä., zu differenzieren.

3.1.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip besagt, dass Klassen, die bereits korrekt funktionieren, erweitert werden können (bspw. durch Einführung neuer abgeleiteter Klassen). Die Interfaces in Morik (*Cipher*, *DatabaseInterface*, etc.) setzen die Einführung von Implementierungen voraus um verwendet zu werden und können folglich einfach um weitere Subklassen erweitert werden. Auch der Großteil der anderen Klassen könnte prinzipiell durch Vererbung erweitert werden, auch wenn es dazu oft keinen Grund gibt. Bei Klassen, für die wir explizit vermeiden wollen, dass von ihnen geerbt wird gibt es das **final** Schlüsselwort in C++. Dies verwenden wir z. B. für das ValueObject *EntryID*. Dadurch wird aktiv

vermieden, dass Funktionalität hinzugefügt wird, da ValueObjects keine Funktionalität haben sollen. Streng genommen, verletzt dies das Open-Closed Prinzip.

Trotzdem gilt das Prinzip für den Großteil der Code-Basis.

3.1.3 Liskov-Substitution-Prinzip

Es gibt mehrere Stellen in unserer Anwendung, an denen Liksov-Substitution demonstriert werden kann. Das Prinzip ist überall erfüllt, da wir stets mit den Interfaces arbeiten, ohne downcasting anzuwenden.

Ein gutes Beispiel um dies zu demonstrieren ist das *Cipher* interface.

```
1 class Cipher {
2 public:
3     virtual std::string encrypt(std::string plain, std::string ↙
          ↳ master) const = 0;
4
5     virtual std::string decrypt(std::string cipher, ↙
          ↳ std::string master) const = 0;
6 };
```

Die Domain Services, die dieses Interface verwenden um Verschlüsselung durchzuführen akzeptieren einen Pointer auf ein *Cipher* Objekt als Konstruktor Argument. Es hat keinen Einfluss auf deren Funktionalität ob das übergebene Objekt vom Typ *CBC_Cipher<AES>*, *CBC_Cipher<Serpent>* oder eine beliebige andere Implementierung des Interfaces ist.

Es ist zwar wichtig, dass zur Verschlüsselung der gleiche Algorithmus verwendet wird, wie zur Entschlüsselung. Dies ist jedoch irrelevant im Bezug auf dieses Prinzip, denn egal mit welchem Subtyp man *Cipher* ersetzt, die Funktionalität ist unverändert: Ein String kann unter Angabe eines Schlüssels verschlüsselt oder entschlüsselt werden. Solange der Subtyp das Interface korrekt implementiert, sind Substitutionen möglich. Problematische Verwendungen können natürlich zu Problemen führen, beispielsweise die Verwendung verschiedener *Cipher* Instanzen für Ver- und Entschlüsselung würde bedeuten, dass die Instanzen nur paarweise ersetzt werden dürfen, da sonst falsche Ergebnisse entstehen würden.

3.1.4 Interface-Segregation-Prinzip

Interfaces sollen nur Methoden haben, die eng an ihren Zweck gebunden sind. Morik hält sich an dieses Prinzip, es existieren keine unnötigen Methoden in den Interfaces. Bspw. um ein neues *Cipher* zu implementieren, muss nur Ver- und Entschlüsselung implementiert werden, sonst nichts.

3.1.5 Dependency-Inversion-Prinzip

Dieses Prinzip besagt, dass high-level Komponenten keine Abhängigkeiten auf lower-level Komponenten haben sollen. Ein klassisches Beispiel, ist die Abhängigkeit von Plugins. Die Klasse in der Applikationsschicht, die die vom Plugin bereitgestellte Funktionalität verwendet, soll nicht von der Plugin-spezifischen Implementierung abhängig sein. Eine solche Abhängigkeit würde die Funktionalität der Applikation abhängig von der Funktionalität des Plugins machen. Durch Definition eines Interfaces in der Applikationsschicht, das in der Pluginschicht implementiert werden muss, kann dies verhindert werden. Die tatsächlichen Instanzen der Plugin-Klassen werden durch Dependency Injection übergeben. In unserem Fall gibt es beispielsweise die Schnittstelle zur Datenbank, in der Dependency Inversion demonstriert wird. Die abstrakte Klasse *AbstractSqlDatabase* in der Adapterschicht, wird von der konkreten Implementierung einer SQL-Datenbank (der Klasse *SQLiteDatabase*) implementiert.

3.2 DRY

Morik ist größtenteils frei von Redundanzen, bei denen tatsächliche Funktionalität mehrmals implementiert wird. Natürlich gibt es einige Methodenhierarchien, die leicht redundant sind (bspw. *PasswordEncryptor::encrypt -> Cipher::encrypt -> CBC_Cipher::encrypt*), jedoch sind diese notwendig um den Architektur-Anforderungen zu entsprechen. Außerdem gibt es bspw. Wiederholungen beim Einlesen der Benutzereingaben im *CommandLineInterface*[1]. Auf weitere Wiederholungen im Code wird in Kapitel 7 eingegangen.

[1] siehe Zeile 18-26 und Zeile 43-51

Ein Beispiel für die geziehlte Vermeidung von Redundanz ist der Einsatz von Templates der *CBC_Cipher* Klasse, um verschiedene Blockchiffren verwenden zu können, ohne mehrere Klassen von Hand zu implementieren[2].

3.3 GRASP

Im Folgenden wird erläutert, inwiefern Morik die Prinzipien Low Coupling und High Cohesion der GRASP Prinzipien erfüllt.

3.3.1 Low Coupling

Morik verwendet Interfaces und damit polymorphe Methodenaufrufe an vielen Stellen[3]. Damit sind die einzelnen Komponenten nur schwach an die Implementierungen dieser Interfaces gekoppelt.

Auch gibt es generell kaum statische Methodenaufrufe, wobei zumindest im Verschlüsselungs-Plugin die Schlüsselherleitung als statische Methode realisiert ist[4]. Dadurch ist *CBC_Cipher* abhängig von dieser Methode. Dies ist jedoch die einzige Klasse, die an diese statische Methode gekoppelt ist.

Ein weiterer Fall von Coupling, ist der *InstanceManager*. Dieser stellt einige global verfügbare Instanzen zur Verfügung. Klassen wie bspw. die *EntryFactory* sind davon abhängig, dass diese Instanzen gesetzt/valide sind. Dies wird in Kauf genommen, da garantiert werden soll, dass diese Instanzen überall verwendet werden, wo die entsprechenden Klassen gebraucht werden.

3.3.2 High Cohesion

Die Klassen in Morik sind alle sehr übersichtlich. Klassen wie der Domain Service *PasswordEncryptor* machen genau das, was der Name impliziert (in diesem Fall Passwörter

[2] siehe https://github.com/moorts/Morik/blob/main/src/plugins/encryption/CBC_Cipher.h
[3] bspw. PasswordEncryptor ruft Methoden des Cipher Interface auf, EntryRepository ruft Methoden des DatabaseInterface auf
[4] siehe https://github.com/moorts/Morik/blob/main/src/plugins/encryption/DefaultHash.h

verschlüsseln). Wir haben darauf geachtet, dies bei allen Klassen zu erreichen. Das sorgt für hohe Kohäsion, da keine Attribute und Methoden für Dinge notwenig sind, die nicht direkt zur Funktionalität gehören. Manche Klassen, wie das *DatabaseInterface*, haben zwar eine leicht höhere Anzahl Methoden, jedoch sind diese logisch kohärent: Das Einfügen, Entfernen und Bearbeiten von Einträgen sind Konzepte die klar zusammen gehören.

4 Domain Driven Design

4.1 Ubiquitous Language

Die Analyse der Ubiquitous Language ergab die in Tabelle 4.1 aufgeführten 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 („sensitive 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 den 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 „Entry“ 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. Was die PasswordLength jedoch angeht, war uns

Domänenexperte (KeePass/1Password)	Implementierung (Morik)
Password	PlaintextPassword
Password	EncryptedPassword
Entry/Item	Entry
UUID	EntryId
Title	EntryName
User Name	Login
Vault	EntryRepository
Password Generator	PasswordGenerator
Length of generated password	PasswordLength
?	PasswordEncryptor
?	PasswordDecryptor

Tabelle 4.1: Ubiquitous Language Gegenüberstellung

die Formulierung von KeePass zu lang, weshalb wir uns die Präposition „of“ und das Adjektiv „generated“ gespart haben. Außerdem ist das Adjektiv sowieso überflüssig, da nur beim Generieren der Passwörter überhaupt explizit eine Länge gesetzt werden muss, denn wenn der Benutzer das Passwort selbst eingibt, so legt er diese implizit fest. Weder KeePass noch 1Password benennen die Teile ihrer Software, die die Daten ver- und entschlüsseln. Stattdessen ist die Rede von „encrypted data“ und „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. Auch die *PasswordLength*, die die Länge des zu generierenden Passworts speichert, ist ein Value Object, da eine gleiche PasswordLength bedeutet, dass die Passwörter gleich lang sind. 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 das *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, anstelle dass die Auswertung, ob der Test das richtige Ergebnis 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 Abschnitt 5.2 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 Abschnitt 5.3 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 irrelevanten Aspekte der Applikation getestet, sondern wir fokussieren uns, wie bereits bei „Thorough“ genannt, auf das Testen der wichtigsten Funktionalitäten.

5.2 Code Coverage

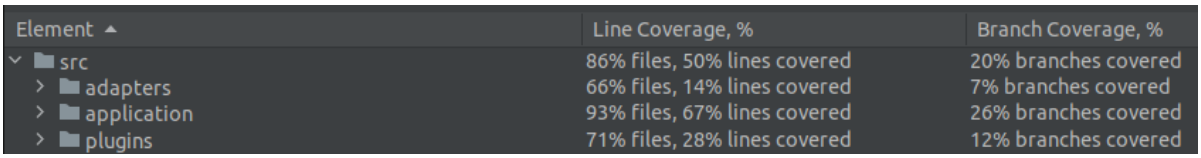

Abbildung 5.1: Line Coverage und Branch Coverage Messung

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 50%, während eine Branch Coverage von nur 20% erreicht wird. Eine Line Coverage von 50% klingt dabei erst mal gar nicht so schlecht, da die Hälfte des Codes getestet wird. Die Branch Coverage jedoch weist darauf hin, dass die getestete Hälfte des Codes nur 20% der gesamten Funktionalität widerspiegelt. Dies zeigt erneut, dass es einfacher ist eine hohe Line Coverage zu erreichen, als eine hohe Branch Coverage. Ermittelt wurden diese Zahlen durch das Ausführen der Tests mit Coverage mithilfe von JetBrains IDE CLion.

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 EntryRepository getestet werden kann, die die Übergabe eines AbstractDatabaseInterface im Konstruktor erwartet, ohne zuvor ein vollständiges Objekt einer AbstractDatabaseInterface-Implementierung erzeugen zu müssen. 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, ohne ein Objekt einer AbstractSqlDatabase-Implementierung erzeugen zu müssen.

6 Entwurfsmuster

6.1 Factory Method

Das Entwurfsmuster der Factory Method wird von der *EntryFactory* umgesetzt. Um ein Entry aus den Standarddatentypen zu erzeugen hat die EntryFactory zwei verschiedene statische Factorymethoden. Die Methode *createEntry* nimmt alle notwendigen Daten als Standarddatentyp, also die EntryId als unsigned int, den EntryName, Login und EncryptedPassword als string, und erzeugt daraus die notwendigen Value Objects, um letztendlich den Entry zu erzeugen. Die Methode *createEntryFromPlaintext* hingegen nimmt ebenfalls alle vier notwendigen Parameter als Standarddatentypen entgegen, geht jedoch davon aus, dass das Passwort nicht verschlüsselt ist, weshalb vorerst ein Plaintext-Password aus dem übergebenen String erzeugt wird, das dann an den PasswordEncryptor Domain Service übergeben wird, um ein EncryptedPassword zu erzeugen. Schließlich wird auch hier ein Entry aus den erzeugten Value Objects erzeugt und zurückgegeben. Die EntryFactory vereinfacht also das Erzeugen von Entries dadurch, dass nicht immer erst alle vier Value Objects, die notwendig sind um den Konstruktor des Entrys aufzurufen, erzeugt werden müssen, sondern direkt als Standarddatentyp übergeben werden können und die Factory diese Umwandlung intern vornimmt. Eine Visualisierung der EntryFactory in Form eines UML-Diagramms kann in Abbildung 6.1 betrachtet werden.

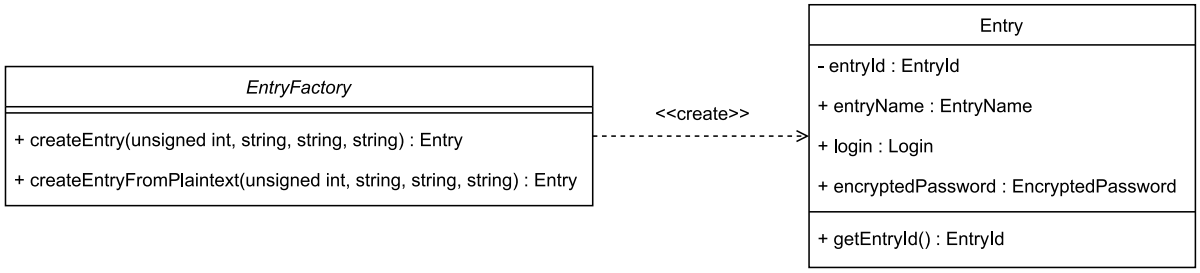

Abbildung 6.1: UML-Diagramm EntryFactory

7 Refactoring

In diesem Kapitel werden einige der gefundenen Code Smells und der angewendeten Refactorings in Morik erklärt. Bei den Code Smells handelt es sich dabei um verbesserungswürdige Codestellen, während die Refactorings konkrete Lösungen für solche Stellen bieten, ohne dabei das externe Verhalten, wie beispielsweise die Schnittstelle oder die Bedeutung des Ergebnisses, zu verändern.

7.1 Code Smells

7.1.1 Duplicated Code

Bei dem ersten identifizierten Code Smell handelt es sich um duplicated code, also doppelt vorhandenen Code. Dies führt dazu, dass Änderungen an einer Stelle nichts an der Funktionsweise der anderen Stellen ändert, was mehrfachen Pflegeaufwand zur Folge hat. Der Code Smell kann in dieser Version des DatabaseInterfaces gesehen werden. Es geht dabei um den Codeabschnitt, der prüft, ob beim Ausführen des SQL-Statements ein Fehler aufgetreten ist. Dieser Codeabschnitt ist in den Methoden *insertEntry* (Z. 30-34), *removeEntry* (Z. 41-45) und *modifyEntry* (Z. 53-57) vorhanden.

7.1.2 Long Method

Ein weiterer Code Smell, der in Morik vorhanden ist, ist long method, also eine lange Klassenmethode. Dieser Code Smell ist in der Methode *mainloop* der Klasse *CommandLineInterface* zu beobachten, die über 200 Zeilen lang ist. Die Methode enthält dabei mehrere unabhängige Funktionalitäten, wie beispielsweise das Auflisten aller Entries sowie das Abfragen der Benutzereingaben zum Erstellen eines neuen Entrys. Da diese Funktionalitäten nichts miteinander zu tun haben, sollten sich diese in verschiedenen Methoden befinden. Aus diesem Grund ist die Methode, so wie sie momentan ist, zu lang.

7.1.3 Switch Statement

Der dritte Code Smell, der gefunden wurde, ist das Switch Statement. Dieses ist nicht flexibel, da es ausschließlich in-place wachsen kann und hat außerdem eine fehleranfällige Syntax, da sehr leicht das *break* am Ende eines Falls vergessen werden kann, was dazu führt, dass mehrer Fälle ausgeführt werden. Dieser Code Smell wurde ebenfalls in der *mainloop*-Methode der *CommandLineInterface*-Klasse gefunden und ist in Zeile 27 zu sehen. Es kann davon ausgegangen werden, dass in Zukunft weitere Optionen hinzukommen, die der Benutzer auswählen kann. Um diese neuen Optionen einzubauen, muss das Switch Statement für jede Option um einen weiteren Fall ergänzt werden. Wird ein Fall vergessen oder wird das *break* am Ende eines neuen Falls vergessen, so führt dies zu einem Fehler. Da also zu erwarten ist, dass an dieser Stelle Erweiterungen geschehen, ist das Switch Statement hier fehl am Platz.

7.2 Refactorings

7.2.1 Extract Method

Das erste Refactoring beseitigt den ersten Code Smell, also den duplicated code im DatabaseInterface. Um dies zu beheben wurde das Refactoring Extract Method angewendet, was in diesem Commit beobachtet werden kann, wodurch der duplizierte Code in eine private Methode namens *executeWithErrorCheck* ausgelagert wurde. Das Refactoring führt dazu, dass zukünftig nur noch eine zentrale Stelle anzupassen ist, sollte das Verhalten bei einem Fehler verändert werden müssen. Entsprechend ist nur noch eine Änderung notwendig und jede Codestelle, die die Funktion aufruft ändert sich gleichermaßen. Der Pflegeaufwand nimmt also stark ab, während das Ausführen des Codes einheitlicher abläuft.

7.2.2 Rename Method

Ein weiteres Refactoring, das angewendet wurde ist Rename Method. Dies wurde in diesem Commit durchgeführt und sorgt für eindeutigere Namen der Methoden. Der Leser dieser drei Methodenaufrufe muss demnach nicht mehr die Parameter inspizieren, um

herauszufinden welche der drei Methoden tatsächlich aufgerufen wird, sondern kann dies direkt am Methodennamen erkennen und weiß daher schneller welches Feld des Entrys bei der betrachteten Codestelle modifiziert wird.