Morik Password Manager

Programmentwurf

von

Moritz Gutfleisch

und

Erik Zimmermann

Abgabedatum:	01. Februar 2018
Bearbeitungszeitraum:	01.10.2017 - 31.01.2018
Matrikelnummer, Student:	0000000, Moritz Gutfleisch
Matrikelnummer, Student:	0000000, 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.

2.1.3 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 *PasswordGenerators* 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 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.

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 demonstiert 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 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.

Ein Beispiel für die geziehlte Vermeidung von Redundanz ist der Einatz von Templates der *CBC_Cipher* Klasse, um verschiedene Blockchiffren verwenden zu können, ohne mehrere Klassen von Hand zu implementieren[1].

[1] siehe https://github.com/moorts/Morik/blob/main/src/plugins/encryption/CBC_Cipher.h

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[2]. 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[3]. 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*[4]. 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 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*[5], haben zwar eine leicht höhere Anzahl Methoden, jedoch sind diese logisch kohärent: Das

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

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 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. Was die PasswordLength jedoch angeht, war

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
?	PasswordEncrypter
?	PasswordDecryper

Tabelle 4.1: Ubiquitous Language Gegenüberstellung

uns 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 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. 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 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

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.

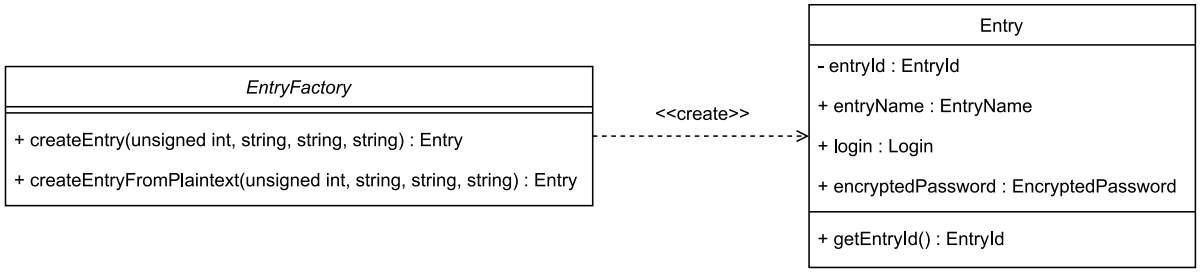

Abbildung 6.1: UML-Diagramm EntryFactory

7 Refactoring