DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Entwerfen und Implementieren einer Verschlüsselungssoftware

Ausarbeitung

Bachelor of Science

Studiengang Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Nico Schrodt

Abgabedatum 23. April 2022

Bearbeitungszeitraum	5 + 6 Semester
Kurs	TINF19B3
Dozent	Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

1 Einführung

Dieses Kapitel befasst sich vorwiegend mit relevanten Grundlagen der Arbeit. Unter anderem wird das Ziel spezifiziert, elementare Aspekte der Arbeitsweise eines Prozessors werden erläutert und die verschiedenen Werkzeuge mit denen das Ziel realisiert wird werden aufgeführt.

1.1 Ziel der Arbeit

In dieser Arbeit soll ein Simulationsprogramm geschrieben werden, mit dem mehrere unterschiedliche 8-Bit Prozessoren simuliert werden können. Dazu sollen die Grundlegenden Eigenschaften in kurzen Lernprogrammen erläutert werden. Ebenfalls soll es eine interaktive Einweisung geben wie der Simulator verwendet werden kann.

1.2 Repository

Der Quellcode kann in folgendem GitHub-Repository abgerufen werden:
`https://github.com/NicoSchrodt/EncryptionService`

2 Clean Architecture

Der Sinn einer Clean Architecture ist es das Programm in klar definierte Schichten zu zerlegen die unabhängig voneinander ausgetauscht werden können. Dadurch soll idealerweise die Langlebigkeit und Wartbarkeit eines Projekts gewährleistet werden können.

2.1 Geplante Schichtenarchitektur

Für dieses Projekt sind zwei Schichten vorgesehen. Einmal die Benutzeroberfläche (GUI) welche mit Qt implementiert wird und die Logik, welche unter anderem die Verschlüsselung vornimmt. Der Benutzer soll ausschließlich mit der von Qt generierten Oberfläche interagieren z.B. durch Textfelder oder Knöpfe, welche vorkonfigurierte Befehle ausführen.

2.2 Umsetzung

Platzhalter

2.2.1 Benutzeroberfläche

Platzhalter

2.2.2 Verschlüsselungsdienst

Platzhalter

3 Entwurfsmuster

Das für den Umfang dieses Programmentwurfs verwendete Entwurfsmuster ist der Dekorierer. Aufgabe des Dekorierers ist es eine Klasse oder Funktion um einen oder mehrere Aspekte zu erweitern ohne die Klasse selbst zu verändern. Das Entwurfsmuster wurde in der Klasse 'EncrypterInterface.py' angewendet.

```
class EncrypterInterface:
    def __init__(self, reference):
        self.text = reference

    def encrypt(self, key):
        # Encrypt the character list in text-object
        pass

    def decrypt(self, key):
        # Decrypt the character list in text-object
        pass
```

'EncrypterInterface' dient wie der Name bereits verrät als Interface für konkrete Encrypter. Dabei ist aber nicht gewährleistet, dass die konkrete Implementierung die im Interface beschriebenen Funktion selbst implementiert. Der Dekorierer soll hier die Aufgabe Übernehmen, auf eine konkrete Implementierung zu kontrollieren und bei Fehlen dieser eine Exception auszulösen.

```
class EncrypterInterface(metaclass=abc.ABCMeta):
    def __init__(self, reference):
        self.text = reference

    @abc.abstractmethod
    def encrypt(self, key):
        # Encrypt the character list in text-object
        raise NotImplementedError

    @abc.abstractmethod
    def decrypt(self, key):
        # Decrypt the character list in text-object
        raise NotImplementedError
```

Die Funktion des Dekorierers beschränkt sich hier auf konkrete Implementierungen des EncrypterInterfaces, Also Klassen die von 'EncrypterInterface' erben. Das Interface selbst könnte potentiell immer noch instantiiert und verwendet werden ohne die Exception auszulösen.

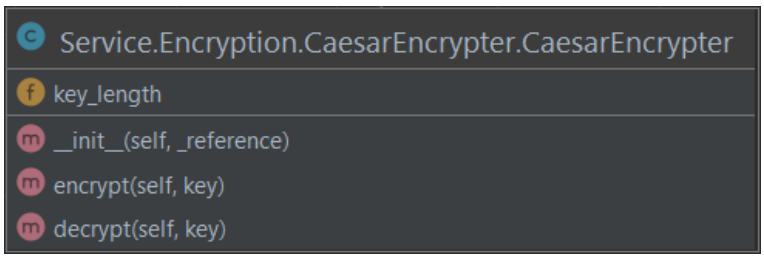
4 Programming Principles

In diesem Abschnitt werden kurz einige Programming Principles erläutert und deren Anwendung an Beispielen in diesem Projekt aufgezeigt.

4.1 SOLID

4.1.1 Single Responsibility Principle

Das Single Responsibility Principle steht für die Anforderung das jede Klasse nur eine einzige Aufgabe bzw. Verantwortung haben soll. Sinn dahinter ist es Komplexität und unerwünschte Kopplung zu vermeiden. Generell ist nämlich davon auszugehen das eine Klasse mit mehreren Verantwortungen Interaktionen zwischen diesen hat, was unter anderem das Ändern einzelner erschwert. Als Beispiel dafür wird in der unteren Abbildung eine konkrete Implementierung der Encrypter Klasse hergezogen.



Die Klasse hat effektiv eine Aufgabe. Sie erhält bei Instanziierung ein Textobjekt. Auf diesem Textobjekt werden Verschlüsselungen durchgeführt, dabei wird lediglich zwischen Ver- und Entschlüsseln unterscheiden.

4.1.2 Open/Closed Principle

Das Open/Closed Principle beschreibt das Designziel Klassen, Funktionen, etc. so aufzubauen das sie offen sind für Erweiterungen und geschlossen für Veränderungen. Konkret heißt das, neue Anforderungen sollen eher durch z.B. Vererbung realisiert werden, statt konkreten Modifikationen in der relevanten Klasse.

4.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle vermittelt das Prinzip, das jede Spezialisierung, z.B. durch Polymorphie bei Vererbung, an jeder Stelle verwendet werden können muss an der auch die Generalisierung verwendet wird. Beispielsweise soll also die Funktion einer Erbenden Klasse nicht zu einem Fehler führen an einer Stelle an der die Funktion der Ursprungsklasse funktioniert hat.

4.1.4 Interface Segregation Principle

Mit dem Interface Segregation Principle soll verhindert werden, dass Klassen ein über-spezifiziertes Interface verwenden. Ein verwendetes Interface soll also möglichst schlank sein und nicht zu viele Funktionen auf einmal anbieten. Damit soll verhindert werden, dass Klassen Zugriff auf Funktionen haben die sie gar nicht verwenden.

4.1.5 Dependency Inversion Principle

Das Dependency Inversion Principle beschreibt effektiv das Prinzip der Entkopplung. Klassen auf einer höheren Ebene bspw. der Logik eines Programms solle nicht von niedrigeren Klassen z.B. Benutzerinterfaces abhängen.

4.2 GRASP

4.2.1 Low Coupling

Platzhalter

4.2.2 High Cohesion

Platzhalter

4.3 DRY

DRY steht für 'Don't repeat yourself'. Zentraler Angelpunkt dieses Prinzips ist das Vermeiden von Code Duplikaten, sowie das Strukturieren des Programmcodes in einer Weise das nur logisch verknüpfte Elemente sich gegenseitig beeinflussen. Oder in anderen Worten, jedes logische Konstrukt im Quellcode muss durch eine klare von anderen Aspekten getrennte Struktur repräsentiert werden. Dadurch lassen sich z.B. einige Code Smells verhindern wie "Duplicated Codeöder SShotgun Surgery". Das Prinzip wurde angewendet hinsichtlich dem vermeiden von Code Duplikaten, welche sich unter anderem in zahlreichen Klassen die für die Verschlüsselung zuständig sind befanden, da der Prozess der Ver- und Entschlüsselung je nach Verfahren recht ähnlich ist (bspw. Vigenère- oder Caesar-Chiffre).

5 Refactoring

Das Ziel von Refactoring ist das Verbessern der Codequalität. Für diese Arbeit ist es unterteilt in das Identifizieren von 3 verschiedenen Code Smells und das Anwenden von 2 Refactorings.

5.1 Code Smells

Unter 'Code Smells' versteht man Stellen im Programmcode, welche Verbesserungspotential aufweisen bspw. bezüglich der Übersichtlichkeit.
(**Anmerkung:** Die hier aufgelisteten Code Smells sind womöglich nicht mehr in der neuesten Version des Projekts zu finden, sondern nur noch in älteren Commits)

5.1.1 Code Smells 1 Duplicated Code

Dieses Beispiel für einen 'Duplicated Code'- Code Smells ist in der 'Caesar-Encrypter.py'-Datei zu finden. Ausschlaggebend ist hierbei, dass das Verfahren zum Ver- und Entschlüsseln effektiv gleich ist mit der Ausnahme, welcher der Starttext ist und in welche Richtung (Positiv/Negativ) der Schlüssel anzuwenden ist.

```
def encrypt(self, key):
    local_list = self.text.character_list
    local_eligible_character_list = self.text.get_eligible_characters()
    num_elg_chars = len(local_eligible_character_list)
    for i in range(len(local_list)):
        index_key = local_eligible_character_list.index(key)
        try:
            index_char = local_eligible_character_list.index(local_list[i])
            index_new = index_char + index_key
            while index_new >= num_elg_chars:
                index_new -= num_elg_chars
            self.text.cipher_character_list.append(local_eligible_character_list[index_new])
        except ValueError:
            self.text.cipher_character_list.append(local_list[i])
            if local_list[i] != " ":
                print("Invalid character '" + local_list[i] + "' found, skipped. Please add a character-set which "
                                                              "contains character.")
```

```
def decrypt(self, key):
    local_list = self.text.cipher_character_list
    local_eligible_character_list = self.text.get_eligible_characters()
    num_elg_chars = len(local_eligible_character_list)
    for i in range(len(local_list)):
        index_key = local_eligible_character_list.index(key)
        try:
            index_char = local_eligible_character_list.index(local_list[i])
            index_new = index_char - index_key
            while index_new < 0:
                index_new += num_elg_chars
            self.text.character_list.append(local_eligible_character_list[index_new])
        except ValueError:
            self.text.character_list.append(local_list[i])
            if local_list[i] != " ":
                print("Invalid character '" + local_list[i] + "' found, skipped. Please add a character-set which "
                                                              "contains character.")
```

5.1.2 Code Smells 2 Long Method

Platzhalter

5.1.3 Code Smells 3 Large Class

Der dritte Code Smells zeigt eine 'Large Class'. Dieser bezeichnet eine große Klasse die unter anderem zu viele Instanzvariablen, Methoden oder allgemein Codezeilen aufweist.

5.2 Angewendete Refactorings

Platzhalter

5.2.1 Refactoring 1

Platzhalter

5.2.2 Refactoring 2

Platzhalter

6 Unit Tests

Platzhalter

6.1 Verwendete Unit Tests und getesteter Code

Platzhalter

6.1.1 Mocks

Platzhalter

6.1.2 Code Coverage

Platzhalter

6.2 Anwendung der ATRIP-Regeln

Platzhalter