Софийски университет „Св. Климент Охридски“
Факултет по математика и информатика

Проект

на тема:

Мениджър на потребителски пароли

Дисциплина: Обектно ориентирано програмиране

Изготвил:

Златина Лилова

Информационни системи

2 курс, група 2, 3MI0700226

София

2025

Анализ на задачата и избрано решение

Настоящият проект за реализиране на Мениджър на Пароли се състои от три основни компонента – алгоритми за шифриране, команди, чрез които да се модифицира файл, в който се записват паролите и основен модул (ядро). Това е и първата стъпка от решението – да разбием цялостния проект на модули, които да разработваме един по един. След като направим това формално разделение е важно да разбием и самите модули на по – малки задачи – именно класовете и тяхната структура.

1. Модул за алгоритми за шифроване

Това е първият модул като там още самото задание подсказва как трябва да се подходи към проблема. Шифрите трябва да са взаимно заменяеми и другите модули да не се интересуват от това точно коя имплементация използваме – затова най – подходящо е тази част да бъде реализирана със Strategy Pattern в комбинация с Factory Pattern – дефиниране на интерфейс, който всички шифри да следват, след това реализация на шифроващите алгоритми един по един спрямо техните специфики и накрая създаване на клас Фабрика, чиято цел е на база подадено име да връща базов указател към съответния вид шифър. Тук се демонстрира полиморфизъм – другите модули не знаят, но не се и „интересуват“ от това кой шифър се ползва. Проблем, който възникна при реализацията на този модул е нуждата от структура/ клас за матрица, която се ползва активно при реализацията на Шифъра на Хил. С външна помощ от сайта tutorialspoint.com реализрах такъв клас, който се използва активно в имплементацията на този шифър.

2. Главен модул (Ядро)

В този модул се съдържа самия файл за писане и четене на пароли и неговата модификация. Първият проблем беше нуждата от структура подобна на Map – начин да асоцииграме ключ (потребителско име) със стойност (парола). Затова реализирах MyMap – шаблонен клас, чиято цел е да наподоби действието на речник. Следващата важна стъпка беше правилното разделение на модула на отделните му компоненти. За да може максимално да се придържаме към SOLID принципите, по – точно Single Responsibility принципа модула има следната структура:

- IOManager – сингълтън клас, който отговаря единствено за писане и четене на файл с пароли. Има минималистичен интерфейс – единствено пише и чете цялото съдържание на файла.
- FileEncryptor – по задание освен че самите пароли се криптират, трябва и целия файл да бъде шифрован с шифър по наш избор. Това е и главната цел на нашия клас – шифрира/дешифрира целия файл преди да бъде записан/прочетен с шифър, инжектиран в обекта при конструирането му. Тук преизползваме нашия модул за

шифри и сме избрали вече дефиниран такъв – улесняваме целия процес и преизползваме вече дефинирана логика

- PasswordFile – това е динамична структура, представляваща файла за съхранение на пароли, докато самия той се използва от потребителя. Има важни методи за сериализация и десериализация на тази динамична структура – именно това след това се подава надолу през предишните два класа, за да се запише успешно във файл.
- PassFileManager – последния клас входна структура на целия модул – той е сингълтън, тъй като менажира обект от тип PasswordFile, като спрямо изискванията можем да имаме само един отворен файл в рамките на програмата. Приема обект отвън при създаване или отваря сам такъв, ако вече е създаден, неговата основна цел е чрез композиция да менажира жизнения цикъл на инстанция на зададен от потребителя PasswordFile.

3. Модул за команди

Модулът за команди е реализиран на база Command Pattern – имаме един основен интерфейс, който всяка нова команда трябва да имплементира. Имаме конкретните класове на командите, които във виртуалната функция execute() извършват съответните модификации като си взаимодействат с PassFileManager-а. Последния компонент, нужен за реализацията на дизайн патърна е CommandManager-а – клас наподобяващ фабрика. Има две основни функционалности – да регистрира нови команди и да извиква метода им execute. Тук отново се вижда полиморфизъм – самия мениджър не се интересува от това как е имплементирана самата команда, стига да имплементира базовия интерфейс за команда.

Описание на класовете и структурата на модулите

1. Модул за шифри

Първо имаме базов клас/интерфейс, който се състои само от чисто виртуални функции – encrypt/decrypt – основните действия на един шифър и два метода getName, getConfig – които се ползват при сериализацията на файла за пароли.

```cpp
class Cipher
{
public:
    virtual std::string encrypt(const std::string& key) = 0;
    virtual std::string decrypt(const std::string& key) = 0;
    virtual std::string getName() = 0;
    virtual std::string getConfig() = 0;
    virtual ~Cipher() = default;
};
```

Класът на шифъра на Цезар имплементира съответните функции на базовия клас като единствената член данна е отместването, зададено от потребителя.

```cpp
class CeasarCipher : public Cipher
{
public:
    CeasarCipher(int shift);
    std::string encrypt(const std::string& key) override;
    std::string decrypt(const std::string& key) override;
    std::string getName() override;
    std::string getConfig() override;
private:
    int shift;
};
```

Класът за TextCode шифър е доста подобен на този за Цезаровия шифър. Единствената разлика е, че имаме функция populateMap – на база подадения стринг кодировката се пази в инстанция на шаблонната ни структура от тип речник и чрез функцията запълваме речника.

```cpp
class TextCodeCipher : public Cipher
{
public:
    TextCodeCipher(std::string text);
    std::string encrypt(const std::string& key) override;
    std::string decrypt(const std::string& key) override;
    std::string getName() override;
    std::string getConfig() override;

private:
    void populateMap();
    std::string text;
    MyMap<char, int> textCode;
};
```

Последния шифър на Хил съдържа правата и обратна матрица за шифроване и дешифроване и началния текст, за да се улесни сериализацята. Имаме две специфични функции setKey & paddPlainText – и двете се ползват при конструирането на матриците – едното представлява самото конструиране – setKey, другото е запълване на празни места ако подадения шифър не е точно n*n дълъг.

```
class HillCipher: public Cipher
{
public:
    HillCipher(int n, const std::string info);
    std::string encrypt(const std::string& key) override;
    std::string decrypt(const std::string& key) override;
    std::string getName() override;
    std::string getConfig() override;

private:
    std::string padPlainText(const std::string& plainText) const;
    bool setKey(const std::vector<std::vector<int>>& key_data);
    Matrix key_matrix;
    Matrix inverse_key_matrix;
    int n;
    std::string text;
};
```

Следва класа за матрица, който съдържа основните функции за една матрица – пресмятане на детерминанта, взимане на конкретен елемент на ред и стълб (с предефиниран function call оператор), адюнгиране за пресмятане на обратната матрица, пресмятане на обратна матрица ако има такава и умножение на две матрици.

Член данните са вектор от вектори, репрезентиращ матрицата и размерите на матрицата. Те се ползват за проверки дали съответните аритметични операции могат да бъдат изпълнени на конкретната матрица.

```
class Matrix {

public:
    Matrix(int r, int c);
    Matrix(const std::vector<std::vector<int>>& d);

    int getRows() const;
    int getCols() const;

    int& operator()(int r, int c);
    int operator()(int r, int c) const;
    Matrix operator*(const Matrix& other) const;
    void print() const;
    int determinant() const;
    Matrix adjugate() const;
    Matrix inverse() const;

private:
    int rows, cols;
    std::vector<std::vector<int>> data;
};
```

Последния клас от този модул е фактори класа, който е синглетон и има една единствена функция – да създаде инстанция на шифър по име и подадени параметри. Друга опция е реализиране на абстра ктно фактори, но за целите на проекта това е ненужно и би се получило прекалено усложняване за въвеждане на нов шифър.

```
class CipherFactory
{
public:
    static Cipher* createInstance(const std::string& name, const std::vector<std::string>& arg);
};
```

2. Основен модул

IOManager-а се грижи единствено за писане и четене във файл – има реализирани писане на сериализирани данни във файл, четене на такива от файл и прочитане само на първия ред (ползва се за проверка на паролата). Направен е като синглетон, защото в рамките на програмата няма логика да има повече от една инстанция от този клас.

```
class IOManager {
public:
    static IOManager* getInstance();
    std::string loadFile(const std::string& path);
    void saveFile(const std::string& path, const std::string& content);
    std::string readLine(const std::string& path);
private:
    IOManager() = default;
    IOManager(const IOManager&) = delete;
    IOManager& operator=(const IOManager&) = delete;
};
```

FileEncryptor-а както споменахме по – горе се ползва за шифроване на вече сериализирана информация за файл – преизползваме шифър, избран от нас. Отново в контекста на програмата няма логика да се копира обект от съответния тип. Възможно е отново като с по – горния мениджър да се реализира синглетон.

```
class FileEncryptor
{
public:
    FileEncryptor();
    ~FileEncryptor();
    FileEncryptor(const FileEncryptor&) = delete;
    FileEncryptor& operator=(const FileEncryptor&) = delete;

    std::string encryptFile(const std::string& plainText);
    std::string decryptFile(const std::string& encryptedText);

private:
    Cipher* cipher = nullptr;
};
```

PasswordFile-а е най комплексния клас в проекта – имаме функции за основните операции, които могат да се извършват – четене/писане/обновяване/триене, имаме двете важни функции за сериализация и десериализация. Пазим инстанция от шаблонния ни речник за сайтовете и самите входни данни на потребителите. Имаме също път към съответния файл, парола и използвания шифър – ползват се при сериализацията. FillInPasswords е функция за попълването на речника при десериализация – повтаряща се логика.

```
class PasswordFile
{
public:
    PasswordFile(const std::string& name, Cipher* cipher, const std::string& password);
    PasswordFile();
    ~PasswordFile();

    void savePassword(const std::string& site, const std::string& user, const std::string& pass);
    const std::string loadPassword(const std::string& site, const std::string& user) const ;
    const std::vector<WebsiteEntity> loadAllPasswords(const std::string& site) const;
    void updatePassword(const std::string& site, const std::string& user, const std::string& newPass);
    void deletePassword(const std::string& site, const std::string& user);
    void deleteAllPasswords(const std::string& site);

    void deserialize(const std::string& path, const std::string& plainText);
    std::string serialize();

    std::string getFilePath() const;

private:
    void fillInPassword(const std::string& site, const std::string& user, const std::string& pass);

    std::string filePath;
    std::string configPassword;
    Cipher* cipher;

    MyMap<std::string, std::vector<WebsiteEntity>> entries;
};
```

Следва

PassFileManager-а – синглетон, за който отново изрично са забранени копирането и конструктора. Основните функции са свързани именно с менажиране на жизнен цикъл на един файл: отваряне и затваряне. Тук се пази енкриптора на целия файл, защото това е входната точка към динамичната структура и оттук се задейства потока по записване във файл при затваряне и четене и поява на динамичен обект при отваряне.

```
class PassFileManager
{
public:
    static PassFileManager* getInstance();
    void openFile(const std::string& path);
    std::string getFilePassword(const std::string& path);
    void setFile(PasswordFile* passFile);
    PasswordFile* getFile();
    void closeFile();
    bool hasOpenFile() const;

    ~PassFileManager();

private:
    PassFileManager();
    PassFileManager(const PassFileManager&) = delete;
    PassFileManager& operator=(const PassFileManager&) = delete;


    PasswordFile* currentFile = nullptr;
    FileEncryptor* fileEncryptor = nullptr;

};
```

3. Команден модул

Базовия интерфейс е дефиниран в класа Command – това е функционален интерфейс само с една функция execute.

```
class Command
{
public:
    virtual void execute(const std::vector<std::string>args) = 0;
    virtual ~Command() = default;
};
```

Командите са почти идентични с изключение на open & create.

При create командата имаме функция createFile – създава празен файл със съответните път, парола и шифър. За шифъра ползва фактори класа.

```
class CreateCommand : public Command
{
public:
    void execute(const std::vector<std::string>args) override;

private:
    PasswordFile* createFile(const std::string& path, const std::string& cipherName,
        const std::vector<std::string>& args, const std::string& pass);
};
```

При командата за отваряне специфични функции са check password & openFile. Първата за момента проверява дали взетата от файла парола и подадената от потребителя съвпадат, но може и да се добави допълнителна логика при нужда. Втората именно се грижи да вземе паролата на файла, да извика функцията за сравнения и при съвпадащи пароли да извика функцията на мениджъра за отваряне на файл. Цялата тази логика се обединява във функцията execute.

```
class OpenCommand : public Command
{
public:
    void execute(const std::vector<std::string>args) override;
private:
    bool checkPassword(const std::string& pass, const std::string& readPass);
    void openFile(const std::string& path, const std::string& password);
};
```

За пример показваме функцията за Обновяване, останалите са със същия интерфейс. Всяка от тях проверява дали има отворен файл преди да извърши съответната модификация.

```
class UpdateCommand : public Command
{
public:
    void execute(const std::vector<std::string>args) override;
};
```

За да улесним работата на потребителя направихме и проста функция close, чиято цел е да затвори текущия файл с пароли и да може в рамките на едно пускане да се работи с няколко файла. Интерфейсът е същия като този по – горе.

Този модул завършва с мениджъра на команди – както споменахме по-рано има две важни функции за регистриране на нови команди, за изпълнение на команда. Ползва отново речник като асоциира името на командата с умен указател от съответния клас команда. Отново е сингълтън, тъй като не се нуждаем от няколко негови инстанции в рамките на програмата.

```
class CommandManager
{
public:
    static CommandManager* getInstance();
    void registerCommand(const std::string& name, std::shared_ptr <Command> command);
    void executeCommand(const std::string& inputLine);
private:
    CommandManager();
    MyMap<std::string, std::shared_ptr <Command>> commands;

};
```

Шаблонния клас за речник има основните функции, които тази структура от данни подържа – добавяне/ махане на елемент, взимане и проверка за съществуване на такъв, базов итератор, ползван за сериализацията. Представлява абстракция на вектор от структура ключ – стойност.

```
template <class Key, class Value>
class MyMap {
public:
    bool insert(const Key& key, const Value& val);
    bool remove(const Key& key);
    Value* get(const Key& key);
    const Value* get(const Key& key) const;
    bool contains(const Key& key) const;

    const Key& getByValue(const Value val) const;

    int getSize() const;


private:
    struct Entry {
        Key key;
        Value value;
    };

    std::vector<Entry> data;

public:
    typename std::vector<Entry>::iterator begin() { return data.begin(); }
    typename std::vector<Entry>::iterator end() { return data.end(); }

    typename std::vector<Entry>::const_iterator begin() const { return data.begin(); }
    typename std::vector<Entry>::const_iterator end() const { return data.end(); }

};
```

Последна е функцията main – в нея с един безкраен цикъл даваме възможност на потребителя да въвежда команди и чрез Командния мениджър да ги изпълняваме, ако съществуват и ако са с правилно подадени параметри, разделени с интервал.

```
int main() {

    std::string input;

    std::cout << "Password Manager is running. Type commands. Type 'exit' to quit.\n";

    while (true) {
        try {
            std::cout << ">> ";
            std::getline(std::cin, input);

            if (input.empty()) continue;
            if (input == "exit") break;

            CommandManager::getInstance()->executeCommand(input);
        }
        catch (std::exception& e) {
            std::cerr << e.what() << std::endl;
        }
    }

    std::cout << "Goodbye!";
    return 0;
}
```

Бъдещи подобрения

Възможно е да се реализира механизъм за undo на конкретна команда с помощта на мементо pattern. Лесно може да се добавят още команди - autosave на определен интервал, по – добра и динамична help команда. Също е възможна реализация на още шифри и комбинация на няколко в едно за още по – голяма защита на паролите.

Линк към кода в Github: https://github.com/zlatililova/Password-Manager