# **Документация на проекта „Password Manager”**

## **1. Цел на проекта и йерархия на класовете**

Основната идея е да се пазят всички пароли в един двоичен файл, който се отключва с главна парола, означена тук като <filePass>. Целта е цялото съдържание на файла – както конфигурацията, така и самите записи – да бъде шифрирано, а добавянето на нов алгоритъм да изисква минимални промени в останалия код.

Ключовите класове са подредени йерархично. На върха стои абстрактния клас **Cipher** с методи за криптиране/декриптиране (encrypt, decrypt), сериализация (writeConfig, readConfig) и идентификация (name). От него наследяват конкретните реализации **CaesarCipher**, **TextCodeCipher** и **HillCipher**. Всеки от тези класове притежава:

* константен static const std::string ID – кратко име, чрез което шифъра се регистрира;
* безаргументен конструктор – нужен при десериализация от поток;
* статичен метод makeFromArgs – създава валиден обект от аргументи, подадени от конзолата;
* собствена логика във writeConfig и readConfig, за да съхрани специфичното си вътрешно състояние.

Над конкретните шифри се разполага **CipherFactory** – сингълтън, който по подадено ID връща завършен обект. Фабриката работи в две “посоки”: createFromArgs се използва, когато шифърът се създава от командата create, а createFromStream – когато се прочита вече съществуващ файл чрез open.

Тази архитектура се затваря от шаблона **AutoCreator<CipherType>**, който автоматично регистрира всеки шифър във фабриката по време на статичната инициализация.

Функционалната логика за самия файл е капсулирана в **PasswordManager** – клас, който пази указател към избрания Cipher, текущата <filePass> и вектор std::vector<PasswordEntry> \_entries. Всяка операция върху записа или върху файла минава през него. Конзолното взаимодействие се реализира от **CommandHandler**, който парсира входа и делегира действия към PasswordManager.

## **2. Плъгин архитектура и factory design pattern**

Регистрацията на нов шифър се свежда до един ред: static AutoCreator<NewCipher> \_; в края на собствения си .cpp файл. Това е възможно, защото AutoCreator наследява CipherCreator и изпълнява следните стъпки:

1. В конструктора си вика CipherFactory::registerCipher(this) и така се вписва в централния регистър.
2. Реализира createFromArgs чрез преход към NewCipher::makeFromArgs.
3. Реализира createFromStream, като създава обект с безаргументния конструктор на шифъра и подава входния поток на readConfig.

Така фабриката ни е напълно независима от конкретните типове, а целият “boiler‑plate” код, необходим при класическите **Creator**‑класове, отпада.

## 

## **Трудности и решения**

### **3.1 Създаване срещу отваряне на файл**

Командата **create** получава аргументи от потребителя и внедрява тези аргументи директно в избрания шифър. Обратната команда **open** първо изчита ID‑то и бинарната конфигурация на шифъра от файла, след което реконструира същия обект чрез createFromStream. Поддържането на тези две огледални операции наложи ясна концепция за сериализацията, заради което всеки шифър има виртуални методи: readConfig и writeConfig в интерфейса си.

**3.2 Оптимизация при savePassword**

Първата версия на savePassword презаписваше целия файл при всяка нова парола. За да избегна линейна сложност спрямо броя записи, във всеки файл се пази точната позиция на полето count. При savePassword се използва fstream в режим режим за писане и четене.

Алгоритъмът е следният:

* търси се \_countPos и се инкрементира броят на записите;
* позицията на файла се връща в края;
* сериализира се само новият PasswordEntry, без да се засяга предишното съдържание.

**3.3 Инвертиране на матрица при Hill Cipher**

Шифърът на Хил изисква модулна обратна на ключовата матрица. Намирането на такава обратима матрица и кодирането ѝ в диапазон ASCII 32–126 се оказа трудоемко, затова за функцията invertMatrix бе използвана генерирана с AI имплементация на алгоритъма на Гаус–Жордан. Признавам, че се колебаех между няколко имплементации и накрая използвах AI‑генериран вариант на Гаус–Жордан, най‑вече заради ограничения във времето и за да избегна ръчни грешки при модулните операции. След това адаптирах кода да работи с азбучния диапазон AsciiUtils::RANGE.

1. **Конзолен интерфейс**

Следните шаблони описват пълния синтаксис и задължителните аргументи.

* **create** <файл> <шифър> [аргументи на шифъра] <filePass> – създава нов архив.
  + **caesar** очаква едно цяло число shift.
  + **textcode** очаква път до текстов файл с корпуса.
  + **hill** очаква път до файл, съдържащ най‑напред n, а след това n×n цели числа – елементите на матрицата.
* **open** <файл> <filePass> – зарежда съществуващ архив.
* **save** <website> <user> <password> – добавя нов запис.
* **load** <website> [<user>] – без втори аргумент изброява всички записи за сайта; при подаден потребител връща само съответната парола.
* **update** <website> <user> <newPassword> – сменя съществуваща парола, ако новата стойност е различна.
* **delete** <website> [<user>] – трие сайт или единичен запис, в зависимост от това дали е подаден потребител.

При грешна команда или липсващи аргументи CommandHandler извежда съобщение за невалиден синтаксис.

## **Пълни детайли за writeConfig и readConfig**

### CaesarCipher

* **writeConfig** записва един int „shift“.
* **readConfig** чете същото число и го присвоява на полето shift.

### TextCodeCipher

* **writeConfig** записва size\_t count, след което извежда масива \_decode с дължина count. Таблицата \_encode не се пази, защото може да се пресметне.
* **readConfig** възстановява \_decode, създава \_encode с дължина 128 и попълва обратните съответствия; непознатите символи се маркират с -1.

### HillCipher

* **writeConfig** записва size\_t n, после n×n цели числа – елементите на \_key.
* **readConfig** чете размера, попълва \_key и незабавно пресмята \_invKey чрез invertMatrix, за да може методът decrypt да работи без излишни сметки.

## **6. Бъдещи разширения**

Във възможните бъдещи подобрения влизат графичен интерфейс (например Qt), поддръжка на множество пароли за една и съща двойка сайт/потребител чрез списъци. Нова версия на файловия формат с JSON заглавна част може да улесни обратната съвместимост и транспортирането между различни платформи.

С описаната архитектура проектът остава отворен за разширения без сериозни промени в основния кодов базов слой.