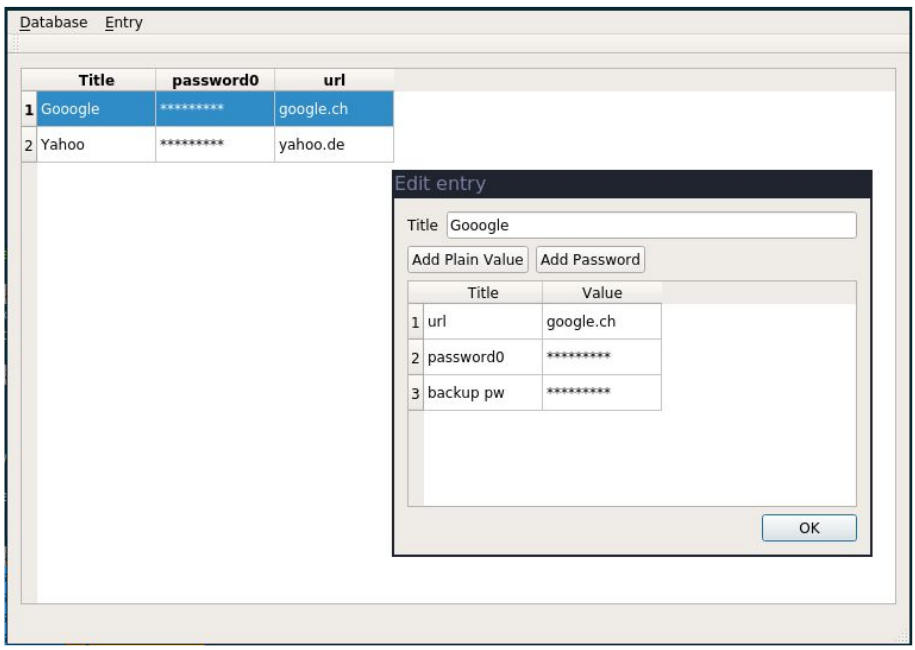
Berner
Fachhochschule

Projektarbeit Zirconiumpass

C++ in Embedded Systems



Ziel dieser Projektarbeit im Rahmen des Vertiefungsmodules C++ in Embedded Systems ist die Anwendung des Rational Unified Process (RUP) von der Analyse über das Design bis zur Implementierung eines Passwortmanagers.

Studiengang: Elektro- und Kommunikationstechnik
Autoren: Timo Lang, Aaron Schmocker
Dozent: Ivo Oesch
Datum: 21.05.2017

Berner Fachhochschule
Elektro- und Kommunikationstechnik

Inhalt

1 Einleitung

Dieses Projekt entstand im Rahmen des Moduls C++ in Embedded Systems. Das Projekt ergänzt die theoretischen Grundlagen welche im Verlauf des Moduls vermittelt wurden um eine praktische Arbeit. Das Projekt soll punkto Analyse und Design, Codierung und Dokumentation als Beispielprojekt für nachfolgende Projekte verwendet werden können.

1.1 Aufgabenstellung und Pflichtenheft

Die Aufgabenstellung die wir uns selber gestellt haben umfasst folgende Punkte:

- Programmierung, Analyse und Design eines Passwortmanagers
- Verschlüsselung aller Einträge mittels state-of-the-art Cryptographie
- Einträge in der Datenbank können verschlüsselt oder unverschlüsselt angelegt werden.
- Software-Design welches höchst mögliche Sicherheit ermöglicht um ein Ausspähen der entschlüsselten Daten aus dem Memory zu erschweren/verhindern.
- Clipboard-Support mit automatischer Löschung nach Timerablauf
- Verwendung des JSON-Formats als Datenbank-Format und zur Serialisierung
- Minimales GUI
- Testen der wichtigsten Funktionen mittels Unittests

2 Analyse und Design

2.1 Anwendungsfälle

In diesem Kapitel werden mit Hilfe eines Anwendungsfalldiagramms die Anforderungen analysiert.

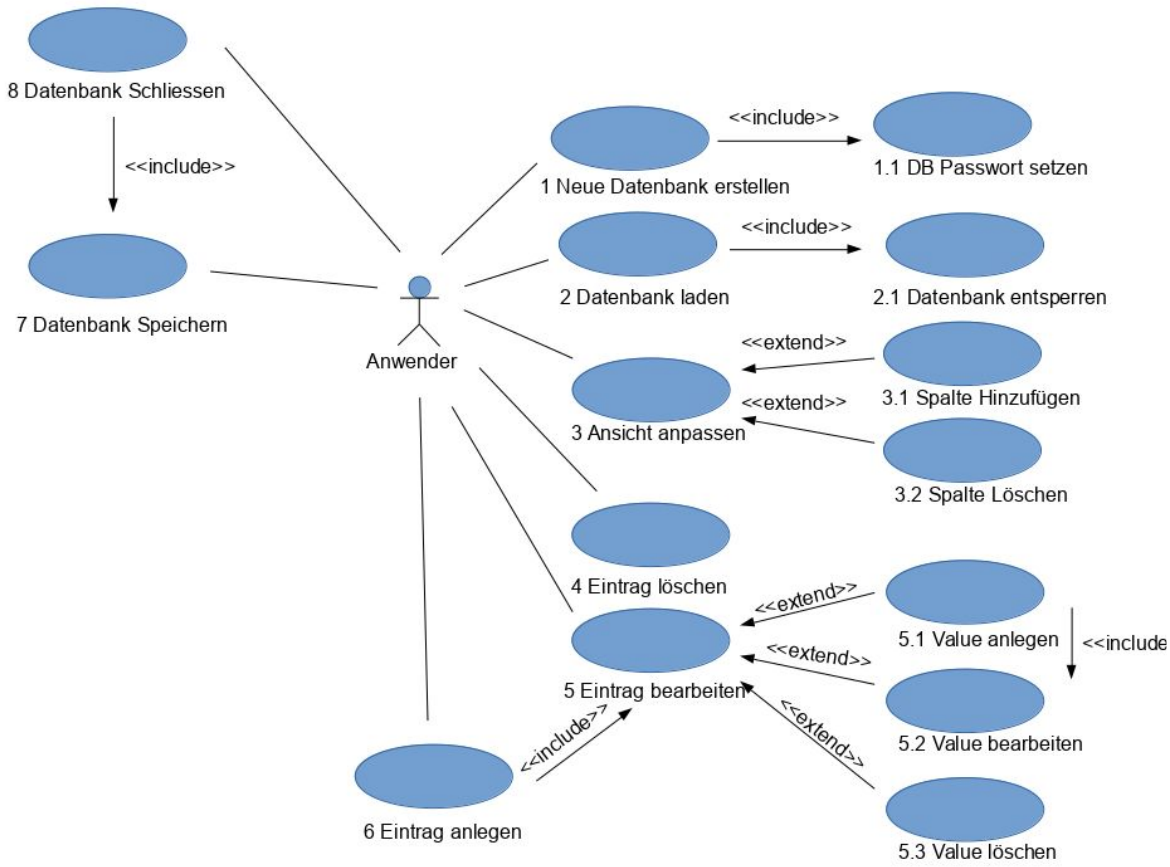


Abbildung 2.1: Anwendungsfall-Diagramm des Passwortmanagers

2.2 Neue Datenbank erstellen

Beschreibung: Im Passwortmanager wird eine neue Datenbank angelegt.
Vorbedingung: Keine

Beschreibung des Ablaufs

E1 Der Benutzer wählt “Datenbank erstellen” aus und setzt ein Passwort (Fall 1.1).
E1A1 Das System leitet aus dem Passwort einen Masterkey her.
E1A2 Das System erstellt eine Datenbank mit Header.
E1A3 Das System setzt den Verschlüsselungs Initialisierungsvektor.

Auswirkungen: Es wird eine Datenbank erstellt und verschlüsselt.
Weitere Informationen: Die Datenbank sollte anschliessend noch vom Benutzer gespeichert werden

Diagramme: Sequenzdiagramm 2.3

2.3 Datenbankeintrag erstellen

Beschreibung: Der Benutzer trägt ein Passwort in die Datenbank ein
Vorbedingung: Es muss eine Datenbank geladen sein (Durch Fall 1 oder Fall 2)

Beschreibung des Ablaufs

E1 Der Benutzer klickt auf “Neuen Eintrag anlegen” .
E1A1 Der Dialog zum Bearbeiten eines Eintrags öffnet sich.
E2 Der gibt einen Titel für den Eintrag ein
E3 Der Benutzer entscheidet zwischen der Eingabe eines Passworts oder eines unverschlüsselten Wertes.
E3A1 Das System erstellt entweder einen encrypted- oder einen plain Wert und speichert ihn im aktuellen Eintrag.

Auswirkungen: Es wird entweder ein verschlüsselter oder unverschlüsselter Eintrag in der Datenbank angelegt.
Weitere Informationen: Keine.

Diagramme: Sequenzdiagramm 2.3 und 2.4

2.4 Klassendesign

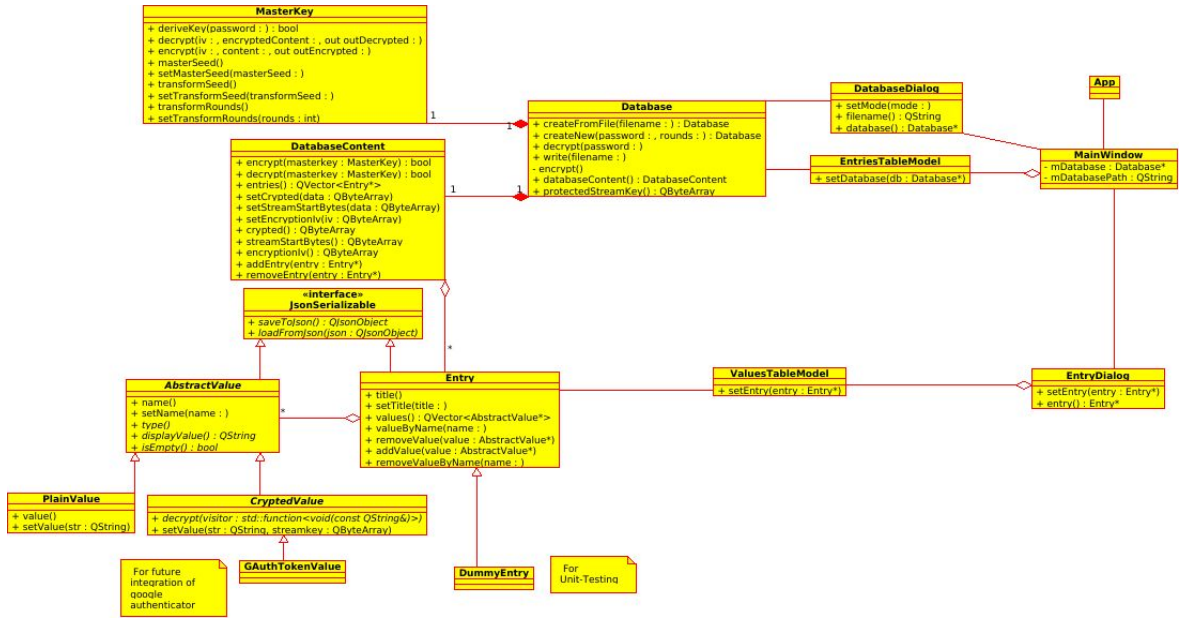


Abbildung 2.2: Klassendiagram des Projekts (befindet sich in voller Auflösung im Projektordner)

Beim Erstellen der Klassenhierarchie wurden folgende Schwerpunkte gelegt:

- Klassen so entwerfen, dass eine gewisse Sicherheit “by-design” entsteht. Zum Beispiel nicht Schlüssel + verschlüsselte Daten als Member der gleichen Klasse (nebeneinander im Memory).
- Erweiterbarkeit für neue Value Typen ermöglichen (z.B. Google Authenticator OneTimePads)
- Testbarkeit gewährleisten durch starke Abstraktion und verwendung von Factories.

Die Klassen Database, MasterKey und DatabaseContent bilden die Grundlage für den kryptografischen Teil der Applikation. Die Database-Klasse bildet den Einstiegspunkt um eine Datenbank-Datei zu laden und zu speichern. Intern wird dabei die MasterKey-Klasse verwendet um einen Schlüssel herzuleiten, der anschliessend verwendet wird, um JSON zu entschlüsseln oder verschlüsseln. Der entschlüsselte Inhalt der Datenbank wird von der DatabaseContent Klasse verwaltet.

Dadurch das MasterKey und DatabaseContent in separaten Klassen sind, lassen sich weitere Schutzmassnahmen gegen fremdes Ausspähen des Memorys einfach einbauen. Intial war die Idee die, MasterKey und DatabaseContent in einem speziellen Memory-Bereich der CPU anzulegen, und diese mittels mlock gegen fremdes Auslesen zu schützen. Dies wurde aber noch nicht implementiert, kann aber einfach erweitert werden.

Grundsätzlich wird **nie** ein Passwort im Klartext im Memory gespeichert. Selbst wenn die Datenbank entsperrt ist, sind alle Passwörter noch einzeln verschlüsselt. Um die Passwörter im Klartext zu lesen muss CryptedValue::decrypt aufgerufen werden, welches via Visitor-Pattern das Passwort im Klartext zur Verfügung stellt, bevor der Memory-Bereich automatisch überschrieben wird. Folgender Pseudocode-Ausschnitt verdeutlicht die Situation und zeigt auch wie sie trotzdem ausgenutzt werden kann:

```
void CryptedValue::decrypt(const QByteArray &streamkey, std::function<void (const QString&)> visitor)
{
   QString decryptedString =  decrypt(mBytearray, streamkey);

   visitor(decryptedString); // let visitor look at it

   decryptedString.fill('0'); // overwrite memory
}

// ------------------------------------------------------------------------------

void SomeClass::someMethod() {

   // Good: Password is not leaked
   cryptedValue->decrypt(mDatabase->protectedStreamKey(),[](const QString& password) {

      QApplication::clipboard()->setText(password);

      QTimer::singleShot(10000, [clipboard](){
         QApplication::clipboard()->clear();
      });

   });


   // Bad: Password is leaked
   QString passwordCopy;
   cryptedValue->decrypt(mDatabase->protectedStreamKey(),[&passwordCopy](const QString& password) {
      // Note the explicit capture that I needed, in order to leak it
      passwordCopy = password;
   });
}
```

Die Klassen Entry, AbstractValue sowie deren Subklassen PlainValue und CryptedValue bilden die Grundlage für das Datenmodell. Wie bereits erwähnt, beinhaltet die Datenbank ein verschlüsseltes JSON-Dokument. Dieses JSON-Dokument kann in eine Liste von Entry-Instanzen deserialisiert werden. Eine Entry-Instanz steht dabei für einen Eintrag in der Passwortliste (z.B. “Google”). Jede Entry-Instanz besitzt neben ihrem Titel ein oder mehrere Value-Instanzen. Eine Value-Instanz steht dabei für einen Wert für den Eintrag (z.B. ein Passwort, eine Url oder ein Notizfeld). Jenachdem ob es sich bei dem Wert um etwas besonders schützenswertes (wie ein Passwort) handelt oder nicht, wird entweder eine PlainValue oder CryptedValue-Instanz erstellt, respektive der Wert wird im Klartext im verschlüsselten JSON-Dokument abgelegt oder vor dem Ablegen noch zusätzlich verschlüsselt.

Die folgende Tabelle zeigt einige Beispieldaten und wie diese in deserialisierter Form abgelegt werden.

Entry Title *Entry::title()*	**Value Name** *AbstractValue::name()*	**Value Content**	**Klasse**
Google.ch			*Entry*
	password	1234	*CryptedValue*
	backup password	4321	*CryptedValue*
	url	google.ch	*PlainValue*
Yahoo.com			*Entry*
	password	passw0rd	*CryptedValue*

Um die erwähnten Klassen einfach testbar zu machen, benutzen diese Intern das Factory-Pattern zum Erzeugen neuer Instanzen. Damit wird ermöglicht das die Unit-Tests ihre eigenen Versionen von Entry & AbstractValue-Klassen einschleusen können.

Um das GUI zu erstellen wurden einige weitere Klassen angelegt, welche meistens von einer bestehenden QT-Klasse erben. Die App-Datei bildet den Einstiegspunkt der Applikation (main-Funktion) und lädt die MainWindow Klasse. Das MainWindow nutzt den DatabaseDialog (QDialog-Subklasse) um eine Datenbank zu laden oder erstellen. Anschliessend werden die Einträge im Haupfenster in einer QTableView anzeigt, welche vom EntriesTableModel (QAbstractTableModel-Subklasse) versorgt wird. Soll ein Eintrag bearbeitet werden, so wird dazu ein EntryDialog (QDialog-Subklasse) erstellt, worin wiederum eine QTableView angezeigt wird um die AbstractValues mittels der ValuesTableModel-Klasse darzustellen.

2.5 Sequenzdiagramme

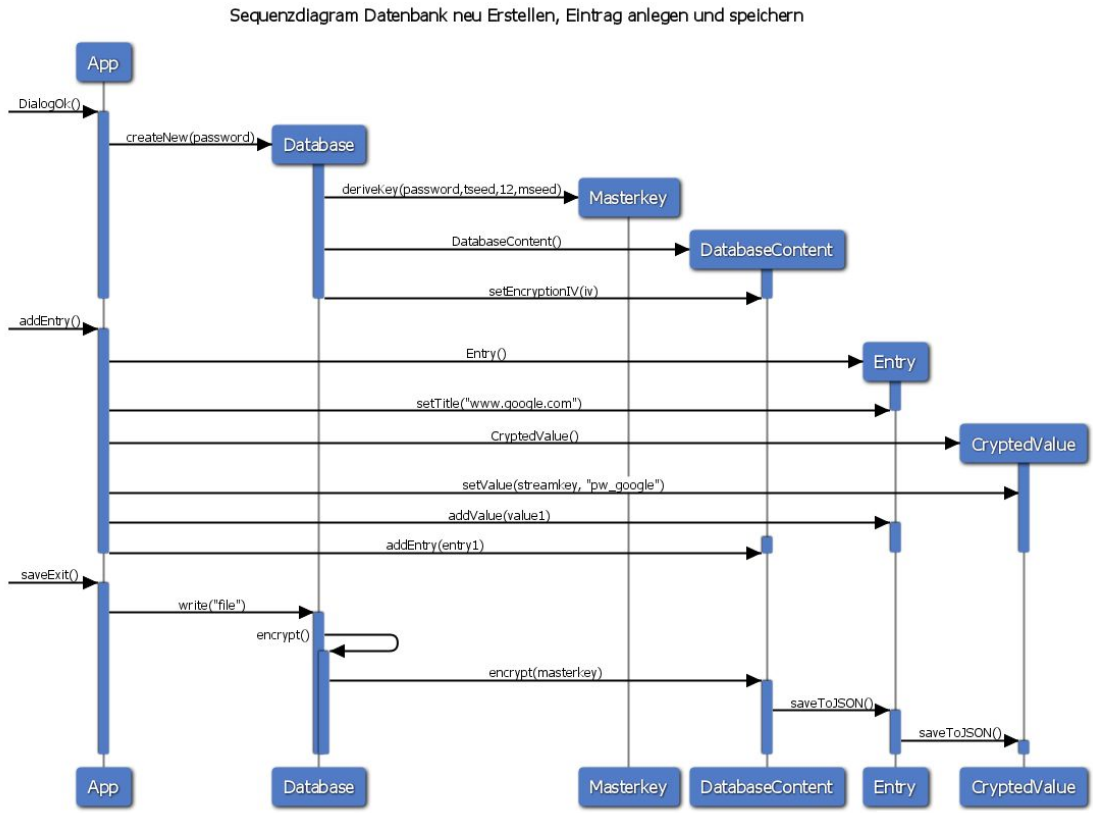


Abbildung 2.3: Sequenzdiagramm zum Erstellen und Abspeichern eines Eintrags

Obiges Sequenzdiagram beschreibt den Ablauf beim Erstellen einer Datenbank, Anlegen eines Wertes und anschliessendem Speichern der Datenbank.
Wird vom Benutzer eine neue Datenbank erstellt, so öffnet sich ein Dialog wo er der Datenbank einen Namen und ein Password geben muss. Das System leitet anschliessend aus dem Passwort einen sogenannten Masterkey her, erstellt einen DatabaseContent und setzt den Initialisierungsvektor im Header des DatabaseContents. Dieser Initialisierungsvektor wird für die Verschlüsselung des JSONs benötigt.
Ist die Datenbank erstellt, so kann der Benutzer einen neuen Eintrag hinzufügen. Diesem Eintrag können mehrere extra verschlüsselte oder normale Werte hinzugefügt werden.
Das System setzt den Titel des Eintrages und fügt den Wert hinzu. Falls es sich um ein extra verschlüsselten Wert handelt so wird dieser beim Anlegen sofort verschlüsselt.
Wenn der Benutzer die Datenbank abspeichert so werden alle Einträge ins JSON Format serialisiert. Anschliessnd ruft die Datenbank ihre Verschlüsselungsfunktion auf und verschlüsselt das gesamte JSON mit dem Masterkey, bevor die Datenbank auf die Disk geschrieben wird.

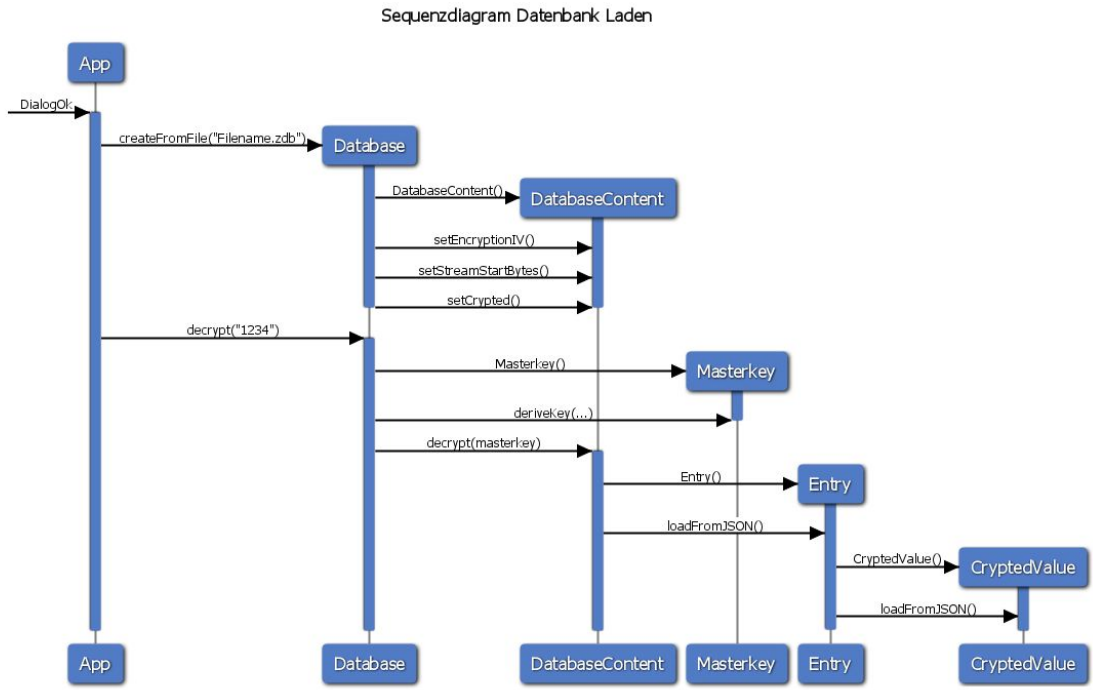


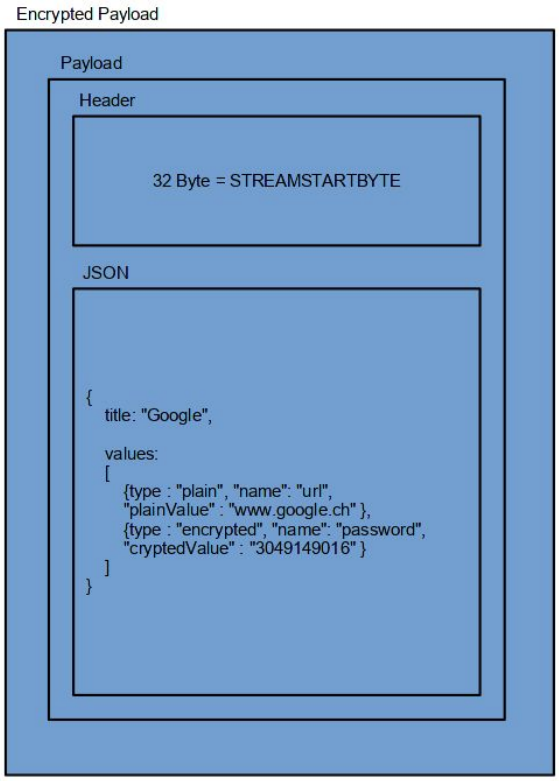
Abbildung 2.4: Sequenzdiagramm zum Laden und Bearbeiten eines Eintrags

Möchte der Benutzer eine bereits vorhandene Datenbank laden so klickt er im GUI unter dem Eintrag “Datenbank” auf “Laden”. Es wird ein Dialog angezeigt in welchem ein Pfad und ein Passwort angegeben werden müssen. Sind beide Einträge valid, so erstellt das System ein Datenbankobjekt (createFromFile) und setzt den Initialisierungsvektor und die StreamStartBytes auf dem DatabaseContent Objekt. Anschliessend wird der Inhalt der Datenbank mit dem Masterkey entschlüsselt. Um zu Prüfen ob die Enschlüsselung erfolgreich war, wird der Start des entschlüsselten Sektors mit den StreamStartBytes verglichen. Falls die Entschlüsselung erfolgreich war, wird für jeder Eintrag aus dem JSON Format in ein Eintragobjekt deserialisiert (via loadFromJSON()). Dabei werden ebenfalls alle dem Eintrag hinzugefügten Values deserialisiert.

Anschliessend ist das System geladen und ist nun dazu bereit Werte auszulesen oder neue hinzuzufügen.

3 Software

3.1 Aufbau des Datenbankfiles



Formatübersicht:

1) Magic Byte 0x7a6972636f6 (14 Bytes)
2) Version
3) MasterKey $TRANSFORMSEED (32 Byte)
4) MasterKey $TRANSFORMROUNDS (4 Byte
5) MasterKey $MASTERSEED (32 Byte)
6) DatabaseContent $ENCRYPTION_IV (16 Byte)
7) DatabaseContent $STREAMSTARTBYTES (32 Bytes)
8) Database $PROTECTED_STREAM_KEY (32 Bytes)
9) $PAYLOAD (siehe Bild)

Abbildung 3.1: Aufbau der Datenbank

3.2 Aufbau eines Entrys in JSON

```
{
   title: "Google",

   values:
   [
      {type : "plain", "name": "url", "plainValue" : "google.ch" },
      {type : "encrypted", "name": "password", "cryptedValue" : [12,113,43,12,42,12] }
   ]
}
```

Beispielaufbau eines Entry Objekts in der Datenbank mit einem PlainValue und einem CryptedValue

3.3 Verwendete Libraries

Aller Code des Projekts wurde während der Projektarbeit eingenhändig verfasst, mit Ausnahme des Codes im Ordner *src/crypto*. Im Ordner *src/crypto* befinden sich Klassen von KeepassXC welche grundlegende kryptografische Funktionen bereitstellen (AES256, sicherer Zufallszahlengenerator etc). Diese Klassen haben eine Abhängigkeit zu der Bibliothek **gcrypt**.

3.4 Unittests

Um die Funktion der Software zu Prüfen haben wir parallel zur Entwicklung Unittests erstellt.

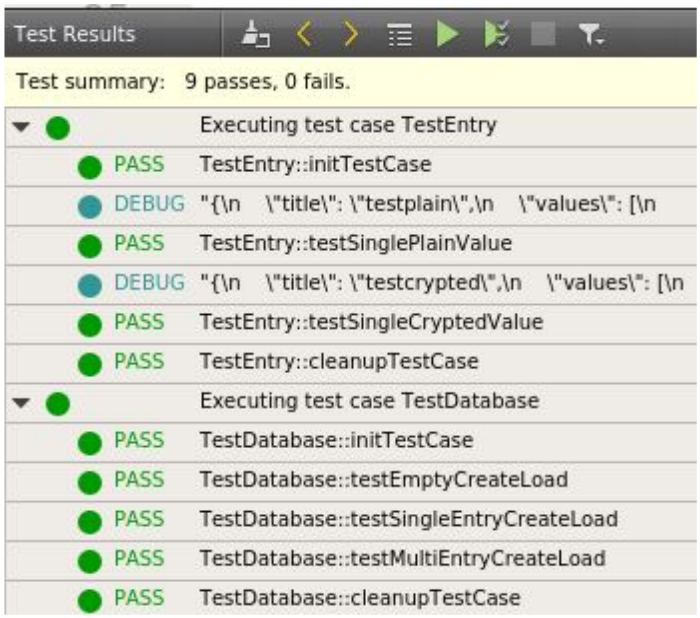


Abbildung 3.2: Unittests für das Projekt

Die Unittests umfassen die Grundfunktionen des Passwortmanagers wie beispielsweise das Erstellen und Laden einer Datenbank, oder das Anlegen eines verschlüsselten oder unverschlüsselten Wertes. Hier besteht aber ganz klar noch Erweiterungsbedarf, dem wir durch die knappe Zeitbemessung nicht ganz gerecht werden konnten.

4 Bedienungsanleitung

4.1 Kompilieren und Erstellen

Die Applikation ist in 3 Teile aufgeteilt:

- libzircopass: Kompletter Source-Code der Applikation, ohne main Funktion. Wird als Shared-Library kompiliert
- Unit Tests: Sind eigenständige QTest Applikationen, welche libziropass einbinden
- App: Starter-Applikation mit main(), welche das Hauptfester öffnet. Diese nutzt intern ebenfalls libziropass.

Die Applikation wurde für das Linux Betriebssystem entwickelt. Um das Projekt auf Linux zu erstellen, kann dieses einfach in QT-Creator importiert werden und damit kompiliert werden. Die benötigte Bibliothek gcrypt, sollte auf Linux vorinstalliert sein.

Die Applikation sollte ebenfalls auf Window kompilierbar sein. Allerdings sind dazu noch einige Anpassungen am Project-File vonnöten, welche wir aus Zeitgründen nicht vornehmen konnten. Konkret müssen vermutlich folgende Dinge angepasst werden:

- Includepath & Librarypath für die gcrypt Bibliothek auf Windows
- Libraryname der libzircopass (Windows benötigt meistens libzircopass0 statt libziropass)

4.2 Erste Schritte

Beim ersten Start des Programms sollte eine Datenbank angelegt werden (CTRL + N). Wählen Sie dazu ein sicheres Passwort. Anschliessend können Einträge hinzugefügt werden (CTRL + E). Normalerweise jedem Eintrag nebst einem Titel noch ein Passwort hinzugefügt. Anschlessend sollte die Datenbank gespeichert werden (CTRL + S).

Beim nächsten Start des Programms kann die bestehende Datenbank wieder geladen werden (CTRL + O). Um schneller an oft benötigte Values zu gelangen, können die Spalten im Hauptfenster erweitert werden.

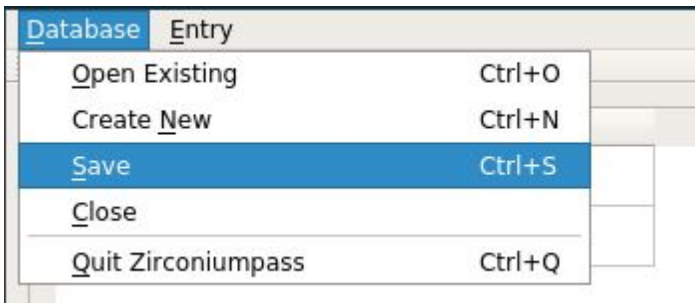
4.3 Tipps

Im Hauptfenster

- Doppelklick auf Eintrags-Titel öffnet den Eintrag im Editor.
- Doppelklick auf einer anderen Spalte, kopiert die Zelle in die Zwischenablage
- Rechtsklick auf Zelle zeigt alle Option. Die markierte Option wird bei einem Doppelklick ausgeführt.

- Rechtsklick auf Tabellenheader erlaubt das Hinzufügen/Entfernen oft benötigter Spalten. Diese Einstellung wird auf dem Computer gespeichert.

- Tastenkombinationen sind in den Menü-Einträgen hinterlegt



Im Eintrag bearbeiten Dialog

- Doppelklick auf Zelle, ermöglicht bearbeiten
- Rechtsklick auf Zelle zeigt mehr Optionen an
- Die Zwischenablage wird nach 10 Sekunden automatisch geleert, wenn es sich um ein Passwort handelt.

5 Schlussfolgerungen

Durch die Projektarbeit konnten wir die gelernte Theorie in die Praxis umsetzen. Besonders wertvoll war die Erfahrung dass es oft schwierig und doch wichtig ist ein Design von Anfang an gut durchzuplanen und nicht im Laufe des Projektes noch erweitern zu müssen.
Da für uns berufsbegleitende Studenten die Zeit oft ein wenig knapp ist mussten wir im Design einige Abstriche machen, konnten aber doch alle Hauptvorgaben erfüllen.

5.1 Stand der Dinge

Der Passwortmanager erfüllt alle Anforderungen des Pflichtenhefts. Es ist möglich eine verschlüsselte Datenbank zu erstellen oder zu laden. Weiter lassen sich sowohl verschlüsselte als auch unverschlüsselte Einträge in der Datenbank erstellen. Besonderer Wert wurde darauf gelegt die abgespeicherten Werte vor einem Ausspähen aus dem Memory zu bewahren. Das GUI ist in der aktuellen Version sehr minimal gehalten, ermöglicht es aber alle Funktionen der Datenbank zu verwenden. Durch das GUI lassen sich auch Einträge ins Clipboard des Computers verschieben und werden nach einer definierten Zeit wieder gelöscht.

5.2 Ausblick und Erweiterungen

Ein Punkt für den die Zeit leider nicht gereicht hat ist das automatische generieren von sicheren Passwörtern mittels Zufallszahlen, einem Seed oder Muster. So liessen sich beispielsweise aus bereits vorhanden Passwörtern neue ableiten.
Ein weiterer interessanter Punkt wäre der Support eines Zwei-Faktor-Authentisierung Verfahrens wie es beispielsweise Google anbietet. Wenn man in einem Eintrag der Datenbank nebst Usernamen und Passwort noch den SecretKey ablegen würde könnte man mittels der Google-Auth-Library und der aktuellen Zeit auch noch ein valides Token für den Login generieren.