Projekt Abschluss: Notenverwaltung

Projektarchivierung und Abschlussauswertung

Inhalt

[Aufgabe 1](#_Toc136871510)

[Interessanteste Aspekte 2](#_Toc136871511)

[Entity-realisierende Objekte funktional einschließlich Beziehungen 2](#_Toc136871512)

[Dependency Injection manuell implementiert 2](#_Toc136871513)

[Zeitraum 2](#_Toc136871514)

[Besprechung/Abgabe wann? 3](#_Toc136871515)

[(eigene) Ziele vor dem Projekt 3](#_Toc136871516)

[Fehler die im Projekt existieren 3](#_Toc136871517)

[Umsetzung 3](#_Toc136871518)

[Interessanteste Probleme die im Projekt behandelt wurden (wissenswertes, aber auch größte Hürden und wichtigste Fehler die man nicht mehr machen will) 4](#_Toc136871519)

[Kurzübersicht entwickelter Packages und Klassen 4](#_Toc136871520)

[DatabaseWrapper, Model und Entity 5](#_Toc136871521)

[Probleme während der Implementierung 5](#_Toc136871522)

[Zirkuläre Referenzierung und Endlosschleife 5](#_Toc136871523)

[Nicht gelöschtes Reader-Objekt von Sqlite verhindert das die Datenbankverbindung geschlossen wird 6](#_Toc136871524)

[AttributToValueDescription beschreibt auch Relationen 6](#_Toc136871525)

[Fazit - Erkenntnisse - habe ich Fehler gemacht, Ziele nicht erreicht? Ziele die in Umsetzung nicht genannt wurden können hier trotzdem angesprochen werden. 6](#_Toc136871526)

[Wichtigste Verbesserungsvorschläge für nächstes Projekt 8](#_Toc136871527)

[Dokumentation: Projekteinrichtung und Versionen der verwendeten Bibliotheken (wichtig, da ansonsten u.U. der Quellcode bei den neuen Bibliotheken vor Kompilierung angepasst werden muss) – ggf. Bibliotheksdateien in Extraordner archivieren 8](#_Toc136871528)

[Anleitung zur Ausführung des Projekts 10](#_Toc136871529)

[Archivierungsvorgang 10](#_Toc136871530)

# Aufgabe

Erstellung eines Objektrelationalen Mappers (ORM) in C#, welcher genutzt werden kann, um eine Datenbank für eine Notenverwaltung zu verwalten.

Der ORM soll die Möglichkeit bieten die zuerst anzubindende SQLite-Datenbank durch eine andere Datenbank auszutauschen, zum Beispiel MySQL.

Die Notenverwaltung soll es Lehrern und Schülern ermöglichen Noteneinzusehen, Noten zu vergeben, Schülern Klassen, Bildungsgängen und Kursen zuzuordnen.

Optional ist ein Frontend, welches die Notenverwaltung steuern kann. Eine rudimentäre Konsolenausgabe, welche die Funktionalität des ORM demonstrieren, ist ausreichend.

Abgesehen von Standard-Bibliotheken dürfen keine weiteren Bibliotheken oder Frameworks eingesetzt werden.

# Interessanteste Aspekte

## Entity-realisierende Objekte funktional einschließlich Beziehungen

Schreibweise z.B: Hans.Create(); Hans.Update(); Hans.Delete(); Hans.FindById(14);

Es ist mir tatsächlich gelungen, dass Entitätsobjekte im Programmcode als Listen ihre Beziehungen als weitere verwendbare Entitätsobjekte referenziert haben.

Beispiel A: Ein Kurs hat viele Teilnehmer, ich kann von einem Kurs-Objekt für einen bestimmten Kurs, z.B. „Mathematik Leistungskurs 2023/2“ alle teilnehmenden Schüler als Objekte in einer Liste abfragen und verarbeiten.

Beispiel B: Eine Klasse hat viele Schüler, ich kann von einem Klassen-Objekt für eine bestimmte Klasse, z.B. „ICD 13“ alle eingetragenen Schüler als Objekte in einer Liste abfragen und verarbeiten.

***Code für Beispiel B:***

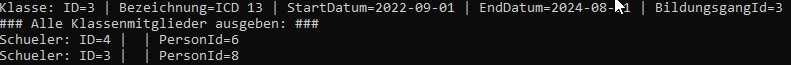
Class icd13 = new Class("ICD 13", (new SimpleDate("2022-09-01")).ToDateTime, (new SimpleDate("2024-08-31")).ToDateTime, kunstPaedagogik);

icd13.create();

student1.AddToClass(icd13);

student1.classes.ForEach(singleClass => singleClass.Print());

***Ausgabe:***



## Dependency Injection manuell implementiert

Ich habe Dependecy Injection manuell implementiert und das DatabaseStorage global in andere Klassen injeziert.

# Zeitraum

9. Februar 2023 – 2. Juni 2023 dh. 4 Monate

investierte Arbeitszeiten / Rhythmus:

* 2x 90 Minuten ca. aller zwei Wochen in der Schule
* Aller drei Wochen 4 Arbeitsstunden
* Kurz vor Abgabe: Dienstag 5 Stunden, Mittwoch nach der Schule von 17 bis 20 Uhr, donnerstags nach der Schule von 17 bis 24 Uhr, freitags von 8:30 bis zur Abgabe 14:50 Uhr
* Abschlussauswertung am 5. Juni: 4 Arbeitsstunden

# Besprechung/Abgabe wann?

2. Juni, 14:50 – Abgabefrist war 1. Juni 24 Uhr nachts

Bewertung: 100 %, Kommentar: super! ☺

Es wurden ausführliche Gespräche über den Umfang der Aufgabenstellung und die gewünschte Abgabeform eine Woche vor Abgabe geführt. Ergebnis: statt einem Konsolenfrontend, reicht eine einfache Demonstration, also eine einfache Debugausgabe der Konsolenanwendung ohne Möglichkeit für individuelle Aktionen oder Menüführung.

# (eigene) Ziele vor dem Projekt

* Dependency Injection selbst programmieren
* Jedes Objekt, welches Entität erbt, kann als Relation in der Datenbank persistiert werden
* Z.B. Schüler.Create() soll möglich sein
  + Eleganz des Objekthandlings war mein persönliches Ziel, dem Lehrer hätte auch ein Database.Create(Schüler) gereicht
  + Vor allem diese Umsetzung hat lange gebraucht, da musste ich lange herumexperimentieren
* Nicht zu viel Zeit neben der Schule investieren, da Lehrer meinte wir hätten in den Unterrichtsstunden Zeit. Desweiteren wollte ich nicht mein Verhalten von der Ameisensimulation wiederholen: Hier habe ich freiwillig bestimmt 20-30 Stunden investiert neben der Schule.
  + Dies ist mir nicht gelungen, siehe investierte Zeit unter Zeitraum.

# Fehler die im Projekt existieren

leicht zu findende Fehler die zu Demozwecken auch später reproduziert werden:

Es wird ein Fehler angezeigt, dass die Datenbanksuche einen Fehler ergab, im Demobereich „einen weiteren Studenten der Klasse hinzufügen.“

Die Relationen „NotenTyp“ und „BildungsgangHatFach“ wurden nicht implementiert.

Es ist nicht möglich ein Objekt eines Schülers (Student-Klasse) zu instanziieren, und von diesem Schüler alle seine Kurse abzufragen, und von diesen seinen Kursen alle seine Noten abzufragen. Hierfür müsste aber lediglich die nötige Relation in Participant implementiert werden. Es ist kein Fehler im Programmcode, und es ist auch möglich mit dem bestehenden ORM diese Daten zu modellieren.

# Umsetzung

C#

Visual Studio 2022

Tools: Umbrello (generiert Klassendiagramme f. C#); draw.io (für ERD); Gimp (Klassendiagramm von Umbrello als Screenshot herausgezogen, und dann in Gimp die Schnipsel zusammen gefügt)

Bibliothek f. Sqlite-Datenbank: System.Data.SQLite

Welche Sprache? Strukturen? Ziele?

Dependency Injection

Attributsbeschreibungen nicht mittels XML sondern einer Entitätsinterface erbenden Model-Klasse, welche in zwei Funktionen ihren Datenbankaufbau beschreibt.

Datenbankanbindung ist komplett gekappselt in DatabaseStorage, Schnittstellenfunktionen des Interface sind lediglich:

* Delete
* InsertSingleEntity – enstpricht „Create“
* UpdateSingleEntity
* FindById
* FindByKeyValue (nicht implementiert)
* InsertManyToManyRelationsIfMissing

# Interessanteste Probleme die im Projekt behandelt wurden (wissenswertes, aber auch größte Hürden und wichtigste Fehler die man nicht mehr machen will)

## Kurzübersicht entwickelter Packages und Klassen

* GlobalObjects
  + Stellt global ein Objekt von DatabaseStorage mittels Dependency Injection Technologie zur Verfügung
* DatabaseWrapper
  + Datenbankanbindung und Entity-Elternklasse wird hier implementiert
    - DatabaseStorage
      * Interface, welches mittels SqliteDatabaseStorage realisiert wird
      * Persistiert Relationen zu Objektbeschreibungen mittels der Klasse AttributeToValuesDescription
    - DatabaseBuilder
      * Interface welches mittels SqliteDatabaseBuilder realisiert wird
      * Stellt die SqliteConnection Verbindung zur Datenbank her
      * Erstellt und löscht alle Datenbanktabellen
      * Schreibt grundlegende Beispieldaten für Entwicklungszwecke in die Datenbank
    - Entity
      * Abstrakte Klasse, welche durch AttributeToValuesDescription-Objekten mit der DatabaseStorage kommunziert
    - AttributesToValuesDescription
      * Container-Klasse, welche zwischen Entity-realisierenden Klassen und der DatabaseStorage-Klasse die Kommunikation ermöglicht
      * Hierfür werden die Attribute und Beziehungen mithilfe der folgenden drei Klassen beschrieben:
        + KeyValue
        + OneToXRelationKeyValue
        + OneToManyKeyValue
    - Statements
      * Package welches die nötigen Sqlite-Skripte für die Erstellung der Datenbanktabellen und von Beispieldaten für die Datenbank zu Verfügung stellt
      * Die Skripte werden vom DatabaseBuilder genutzt um die Datenbanktabellen zu erstellen oder zu löschen, und die Beispieldaten in die Tabellen einzufügen.
* Model
  + Konkrete Entitätsobjekte wie z.B. Schüler oder Lehrer werden hier als im restlichen Programm (also nur in der Main-Funktion) verwendbare Klassen beschrieben
  + Die Realisierung der Interface-Methoden ermöglicht es auch alle Beziehungstypen zu deklarieren mittels „OneToXKeyValue“ und „OneToManyKeyValue“
  + Entity-realisierende Klassen
    - Person
    - Student
    - Teacher
    - Grade
    - Course (Kurs, z.B. Mathematik Leistungskurs Sek. 2)
    - Subject (Fach, z.B. Mathematik)
    - Participant (Schüler, welcher als Teilnehmer in einem Kurs etwas lernt)
    - BranchOfStudy
    - Class
    - Lecturer (dh. Lehrer welcher als Dozent einen Kurs gibt)
  + Tablenotation:
    - Liste mit konstanten Strings, die die Tabellen und Attribute der Datenbank beschreiben.
* Programm
  + Main-Funktion demonstriert die Funktionalität der Model Objekte, und dass diese als Entitäten vom DatabaseWrapper-Package erfolgreich in der Datenbank als Relationen abgespeichert werden

## DatabaseWrapper, Model und Entity

Die abstrakte Klasse „Entity“ wird im DatabaseWrapper-Package implementiert. Objekte die diese Klasse realisieren, können von der Datenbank als Relationen gespeichert, abgerufen, aktualisiert und gelöscht werden.

Die Entity-realisierenden Objekte sind im Package „Model“ im Projekt zu finden. Hier finden sich also die Schüler-, Lehrer-, Noten-, und Bildungsgänge-Klassen (uvm.), mit welchen man im restlichen Programm arbeiten kann.

## Probleme während der Implementierung

### Zirkuläre Referenzierung und Endlosschleife

Beispiel: Ein Schüler besucht viele Kurse, und jeder Kurs hat viele Schüler. Bei jeder Erstellung und Aktualisierung von Schülern oder Kursen, möchte der Entwickler für beide Objekte Listen der besuchten Kurse und der eingetragenen Schüler aktualisieren.

Die Aktualisierung eines Schülers soll also alle dessen besuchten Kurse in der Datenbank abfragen, und diese Liste als Attribut in diesen Schüler verfügbar machen. Nun werden aber bei einer Abfrage der Kurse diese auch aktualisiert, es wird abgefragt welche Schüler in einem Kurs sind, und die eingetragenen Schüler in der Liste eines Kurses aktualisiert. Das Erstellen des Schüler-Objekts in einem Kurs benötigt aber erneut die Abfrage aller seiner Kurse.

Lösung: Es wurde die Variable „recursionLevel“ genutzt, welche bei internen Aktualisierungsabfragen zwischen Entitäts-Objekten und der DatabaseStorage erhöht wird. Falls die Rekursionstiefe größergleich 3 ist, wird von weiteren Hinzufügen von Kurs-Objekten die in Beziehung zum Schüler-Objekt sein sollen abgesehen.

### Nicht gelöschtes Reader-Objekt von Sqlite verhindert das die Datenbankverbindung geschlossen wird

Innerhalb der Klasse SqliteDatabaseStorage wird die Sqlite Datenbank mithilfe eines Objekts der Klasse SqliteConnection angesprochen. Dieses SqliteConnection Objekt nutzt ein Objekt der Klasse SqliteDataReader um die Ergebnisse einer Datenbankabfrage bereit zu stellen. Falls es während der Verarbeitung der Ergebnisse zu einer Exception kommt, kann es aber in manchen Fällen passieren, dass das SqliteDataReader-Objekt nicht gelöscht wird. Die weiterhin bestehende Referenz auf den Reader verhindert, dass die SqliteConnection geschlossen wird. Nach nur wenigen Abfragen blockiert die Sqlite-Datenbank, und gab die Fehlermeldung zurück, dass es zu viele simultane Abfragen gäbe.

Die Lösung hierfür war, dass für jedes der genannten Objekte ein neuer using-Block deklariert wurde, welcher sicher stellt, dass auch bei Fehlern diese Objekte dann zuverlässig wieder gelöscht werden.

## AttributToValueDescription beschreibt auch Relationen

Die Klasse wird von Entity-realisierenden Klassen genutzt, um eine Beschreibung des Objekts zu erstellen, wie es als Relation in der Datenbank existiert. Desweiteren speichert die Klasse die Attribute, und transferiert sie zwischen den DatabaseWrapper und Model-Bereich des Programms.

Auch Relationen werden in diesen Objekt beschrieben.

Attribute sind mithilfe von der Klasse KeyValue beschrieben.

Ein-zu-Eins- und Eins-zu-Viele Beziehungen werden mithilfe von Objekten der Klasse OneToXRelationKeyValue beschrieben.

Viele-Zu-Viele-Beziehungen nutzen hierfür die Klasse ManyToManyKeyValue.

**Problem:** Die daraus resultierenden Methoden in den Entity-realisierenden Klassen sind zu kompliziert, hier wäre ein Refactoring sehr gut.

# Fazit - Erkenntnisse - habe ich Fehler gemacht, Ziele nicht erreicht? Ziele die in Umsetzung nicht genannt wurden können hier trotzdem angesprochen werden.

* Entwicklung
  + Es war sehr gut, dass ich die Exceptions von vorneherein ausgegeben habe
  + An manchen Stellen hätte ich sofort die Exception werfen müssen, und das Programm abbrechen sollen, da ich dann Programmabbrüche an anderen Stellen hatte
    - Generell hatte ich oft das Problem, dass ich bei manchen Fehlern erst eine Weile suchen musste – meine Exceptions sind nicht da aufgetreten, wo die eigentlichen Fehler aufgetreten sind
    - Hier wäre es gut, eine erwartete Eingabe- und erwartete Ausgabevalidierung einzubauen.
    - Exceptions hätte ich nicht so schnell abfangen dürfen während der Entwicklung

„Der Alles auf einmal Ansatz…“:

* + - Ich hätte modularer entwickeln können?
      * Eine Entwicklung mit Stubs- und Mockobjekten hätte ich mir überlegen können, ich habe immer über alle Programmteile hinweg entwickelt. Das hat wirklich viele stressige Fehler verursacht.
        + Dagegen spricht, dass ich diese Art der Aufgabenstellung, dass erste mal hatte, und das Programm gar nicht so gut planen konnte
        + Dies zeigt sich vor allem sehr gut an dem Problem der Relationen! Hier waren die Schnittstellen kaum planbar. Es war hier sogar gut, dass ich nicht alles vorher modular geplant habe
        + Ob der „alles auf einmal“ Ansatz gut war, kann ich abschließend nicht wirklich beurteilen
    - Das ich in einem Art TDD-Stil alle Features nacheinander für die Entitätsobjekte zur als Kommentare, und danach nacheinander als Objekt-Aktivitäten implementiert habe, war ein guter Workaround im Angesicht der geringen Zeit.
* Projektmanagement
  + Die Aufgabenstellung wurde im Zeitumfang stark unterschätzt
  + Es war aber auch nicht möglich noch mehr Arbeitszeit hinein zu stecken, ohne andere Pflichten auf der Arbeit zu stark hinten anzustellen.
  + Das Gespräch zu suchen mit dem Lehrer war sehr hilfreich, und hat mir viel Stress abgenommen.
    - Somit wurden auch nochmal die Anforderung verringert, ursprünglich hatte ich verstanden, dass ein Konsolenfrontend mit Menüführung und Benutzeraktionen zwingen notwendig ist.
    - Es hat auch die Erwartungshaltung des Lehrers verringert. Tatsächlich hat ein ruhiges Gespräch den Lehrer auch zum Umdenken gebracht, ich vermute, dass dieses Gespräch dazu geführt hat, dass der Lehrer erkannte, dass wir 20 bis 40 Arbeitsstunden abseits der Schule hinein gesteckt haben
      * Marcel meinte auch, dass es 3 Hauptaufgaben gab:
        + Datenbank anbinden (Sqlite)
        + Klassenstruktur (ORM), Austauschbare Sqlite mit Mysql
        + Funktionalität das Model-Objekte werden persistiert werden können

Das Relationen erstellt werden, war also eigentlich ein Zusatzpunkt der über die Ansprüche hinaus geht

* + Der Lehrer hat im Gespräch im Nachhinein erklärt, dass es absichtlich so war, dass das Projekt zu groß ist, er ging von vorneherein davon aus, dass wir nicht fertig werden.
    - Es sollte eine Vorbereitung sein, wie im realen Arbeitsalltag Kunden ihre Wünsche formulieren, und wie man an zu großen Projekten seine Zeit nicht gemanagt bekommt.
    - Die Diskussionsfähigkeiten sollten verbessert werden, Diskussionen gehören zum Arbeitsalltag mit Kunden dazu, genauso wie kritisches Hinterfragen.
    - Vor allem sollte gelernt werden auch bei halbfertigen Projekten einen Abschluss zu finden, welche die minimalen Anforderungen gerade zu erreichen, und die Dokumentation zu schnell wie möglich abzuschließen, damit der Kunde ein Papier in der Hand hat.
* Informatik – Theorie (z.B. welche Patterns oder Architekturen wann sinnvoll sind)
  + Eine Recherche wie ORMs gut aussehen wäre zum Abschluss angebracht
  + Vor der Implementierung gängige Ansätze zu recherchieren wollte ich nicht, um mir meine eigenen Ideen auszudenken – das möchte ich auch weiterhin so handhaben

# Wichtigste Verbesserungsvorschläge für nächstes Projekt

* Vor allem die Implementierung der Relationen war am komplexesten. Hier kann noch viel gelernt und verbessert werden.
  + Hier waren die Schnittstellen kaum planbar. Es war hier sogar gut, dass ich nicht alles vorher modular geplant habe
* Ob der „alles auf einmal“ Ansatz gut war, kann ich abschließend nicht wirklich beurteilen
* Vor der Implementierung gängige Ansätze zu recherchieren wollte ich nicht, um mir meine eigenen Ideen auszudenken – das möchte ich auch weiterhin so handhaben
* Exceptions hätte ich nicht so schnell abfangen dürfen während der Entwicklung
  + Generell hatte ich oft das Problem, dass ich bei manchen Fehlern erst eine Weile suchen musste – meine Exceptions sind nicht da aufgetreten, wo die eigentlichen Fehler aufgetreten sind
* Das ich in einem Art TDD-Stil alle Features nacheinander für die Entitätsobjekte zur als Kommentare, und danach nacheinander als Objekt-Aktivitäten implementiert habe, war ein guter Workaround im Angesicht der geringen Zeit.
* Das Logging hätte ich gerne von vorneherein implementiert, dann hätte ich oft nicht so viel „System.Console.WriteLine(….) schreiben müssen“. Dies hat doch etwas genervt.
* Lehrer hat vorgeschlagen alle Objekte als Views abzufragen, hier müssen keine Joins an die Datenbank übergeben werden.

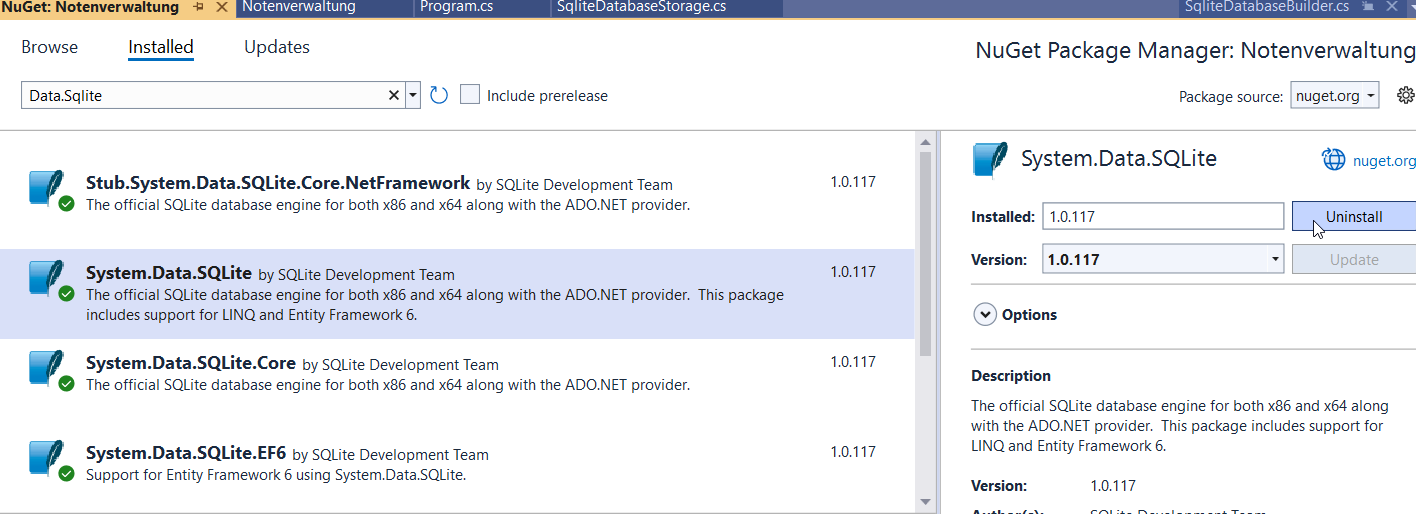
# Dokumentation: Projekteinrichtung und Versionen der verwendeten Bibliotheken (wichtig, da ansonsten u.U. der Quellcode bei den neuen Bibliotheken vor Kompilierung angepasst werden muss) – ggf. Bibliotheksdateien in Extraordner archivieren

**Lediglich die Sqlite-Bibliothek muss noch eingebunden werden. – Vorgehen:**

System.Data.SQLite, Version=1.0.117.0

Vorgehen der Einbindung:

Tutorial von: <https://www.codeguru.com/dotnet/using-sqlite-in-a-c-application/>

1. To connect SQLite with C#, we need drivers. Install all required SQLite resources from the NuGet package, as pictured in Figure 1. Don’t forget to click “Manage NuGet Packages” from the pop-up menu.
   1. = Rechtslick auf Projekt > NuGet Packages
2. Suche der Data.Sqlite Bibliothek
   1. -> Installieren
   2. 
   3. Basisscript zur Ansteuerung:

using System;

using System.Collections.Generic;

using System.Data.SQLite;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SQLiteDemo

{

class Program

{

static void Main(string[] args)

{

SQLiteConnection sqlite\_conn;

sqlite\_conn = CreateConnection();

CreateTable(sqlite\_conn);

InsertData(sqlite\_conn);

ReadData(sqlite\_conn);

}

static SQLiteConnection CreateConnection()

{

SQLiteConnection sqlite\_conn;

// Create a new database connection:

sqlite\_conn = new SQLiteConnection("Data Source=

database.db;Version=3;New=True;Compress=True;");

// Open the connection:

try

{

sqlite\_conn.Open();

}

catch (Exception ex)

{

}

return sqlite\_conn;

}

static void CreateTable(SQLiteConnection conn)

{

SQLiteCommand sqlite\_cmd;

string Createsql = "CREATE TABLE SampleTable

(Col1 VARCHAR(20), Col2 INT)";

string Createsql1 = "CREATE TABLE SampleTable1

(Col1 VARCHAR(20), Col2 INT)";

sqlite\_cmd = conn.CreateCommand();

sqlite\_cmd.CommandText = Createsql;

sqlite\_cmd.ExecuteNonQuery();

sqlite\_cmd.CommandText = Createsql1;

sqlite\_cmd.ExecuteNonQuery();

}

static void InsertData(SQLiteConnection conn)

{

SQLiteCommand sqlite\_cmd;

sqlite\_cmd = conn.CreateCommand();

sqlite\_cmd.CommandText = "INSERT INTO SampleTable

(Col1, Col2) VALUES ('Test Text ', 1);";

sqlite\_cmd.ExecuteNonQuery();

sqlite\_cmd.CommandText = "INSERT INTO SampleTable

(Col1, Col2) VALUES ('Test1 Text1 ', 2);";

sqlite\_cmd.ExecuteNonQuery();

sqlite\_cmd.CommandText = "INSERT INTO SampleTable

(Col1, Col2) VALUES ('Test2 Text2 ', 3);";

sqlite\_cmd.ExecuteNonQuery();

sqlite\_cmd.CommandText = "INSERT INTO SampleTable1

(Col1, Col2) VALUES ('Test3 Text3 ', 3);";

sqlite\_cmd.ExecuteNonQuery();

}

static void ReadData(SQLiteConnection conn)

{

SQLiteDataReader sqlite\_datareader;

SQLiteCommand sqlite\_cmd;

sqlite\_cmd = conn.CreateCommand();

sqlite\_cmd.CommandText = "SELECT \* FROM SampleTable";

sqlite\_datareader = sqlite\_cmd.ExecuteReader();

while (sqlite\_datareader.Read())

{

string myreader = sqlite\_datareader.GetString(0);

Console.WriteLine(myreader);

}

conn.Close();

}

}

}

# Anleitung zur Ausführung des Projekts

Im Ordner „Release“ müssen neben der Exe-Datei auch alle Bibliotheksdateien da sein.

# Archivierungsvorgang

Screenshots in Reihenfolge der Anwenderaktivitäten

Ausführbare Anwendung in extra Ordner,

Entwicklungsprojektdateien in extra Ordner

Kompletten aktiven Entwicklungsordner extra archivieren

Dokumentation Diagramme, Anwenderdokumentation, Entwicklerdokumentation in extra Ordner ablegen.

Entwicklerdokumentation enthält u.a. Projektkonventionen die eingehalten werden wie z.B. Codestyle, Namingconventions

Finalen Git-Push mit hiesigen Dateien als Archiv