DOKUMENTATION PROGRAMMENTWURF

Advanced Software-Engineering

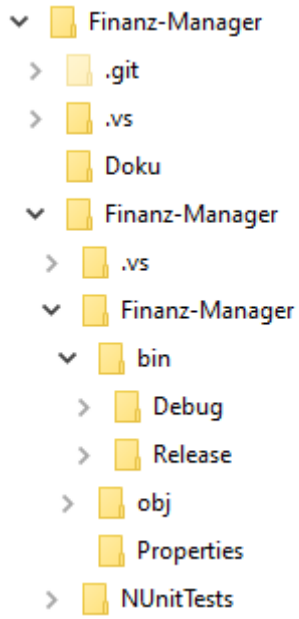
Nico Rahm

Inhalt

Link zum Repository: https://github.com/Nico-Rahm/Finanz-Manager

Allgemeines

Ordnerstruktur



Dieses Dokument ist unter „Doku“ zu finden.

Im Ordner „Finanz-Manager“ ist sämtlicher Programmcode untergebracht. Hier befinden sich die Visual-Studio-Solution und ein weiterer Ordner „Finanz-Manager“, der die Programmdateien enthält. Unter „bin“ sind die Builds zu finden. Unter „NUnitTests“ sind die Unit Test gespeichert.

Installierte NuGet-Pakete

Die Folgenden NuGet sind installiert:

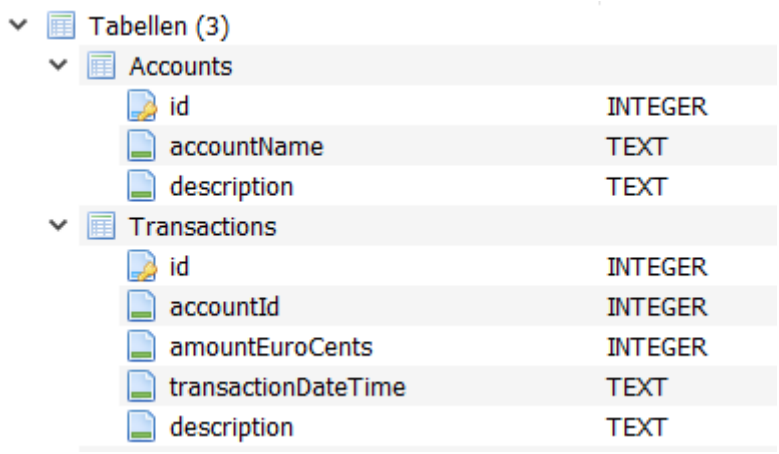
- System.Data.SQLite.Core
  Zum Ansprechen der SQLite-Datenbank
- Dapper
  zum einfachen Verbindungsaufbau und ausführen der SQL-Queries
- NUnit
  Das Framework für die Unit Tests

Framework-Abhängigkeiten

Die Folgenden Frameworks wurden verwendet:

- Microsoft.NETCore.App
- Microsoft.WindowsDesktop.App.WindowsForms

Datenbank Struktur

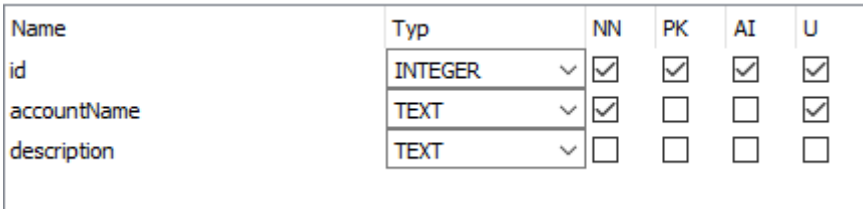


Legende für die Tabellen:

NN	Not Null	Das Feld darf nicht leer bleiben, beim Erstellen eines Datensatzes muss ein Wert angegeben werden.
PK	Primary Key	Primärschlüssel, identifiziert einen Datensatz eindeutig. Ist dieses Attribut gewählt, sind die Attribute NN und U automatisch mit ausgewählt.
AI	Auto inkrement	Der Wert dieses Feldes wird mit jedem neuen Datensatz automatisch inkrementiert. Beim Erstellen eines Datensatzes wird für dieses Feld kein Wert erwartet.
U	Unique	Der Inhalt dieses Feldes muss eindeutig sein.

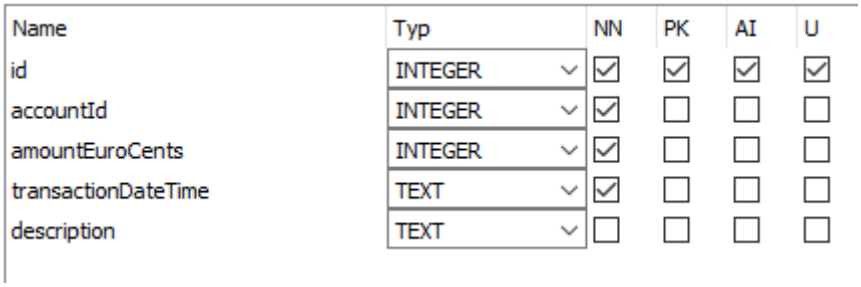
Die Datenbank besteht aus zwei Tabellen:

Accounts



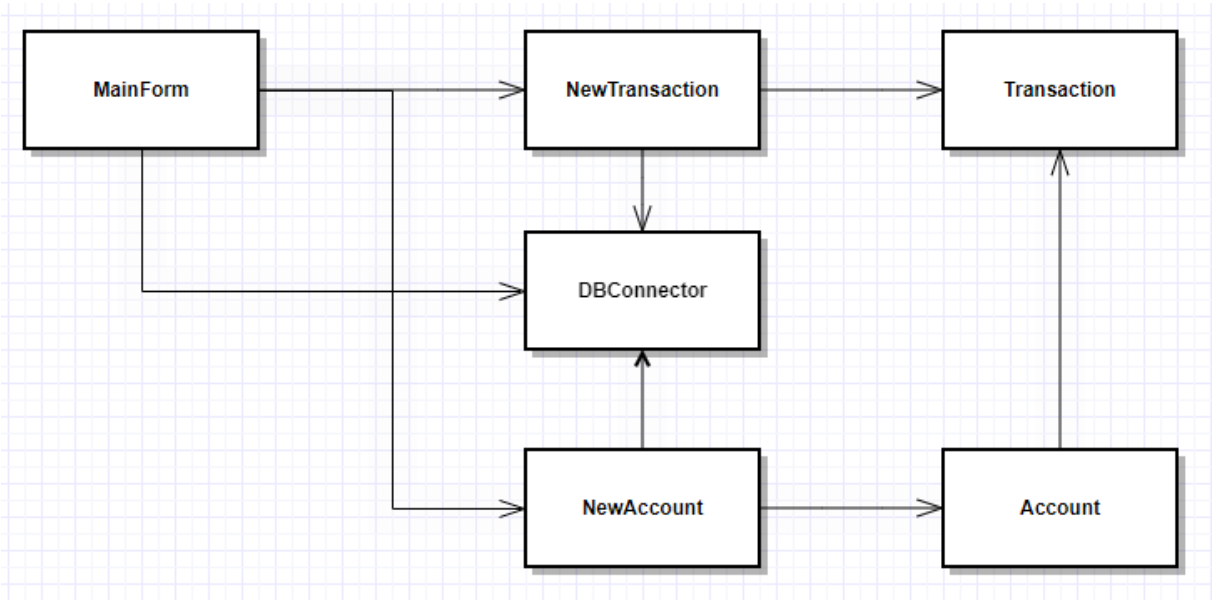
Enthält die einzelnen Accounts mit einer eindeutigen ID, einem Accountnamen und der optionalen Beschreibung.

Transactions



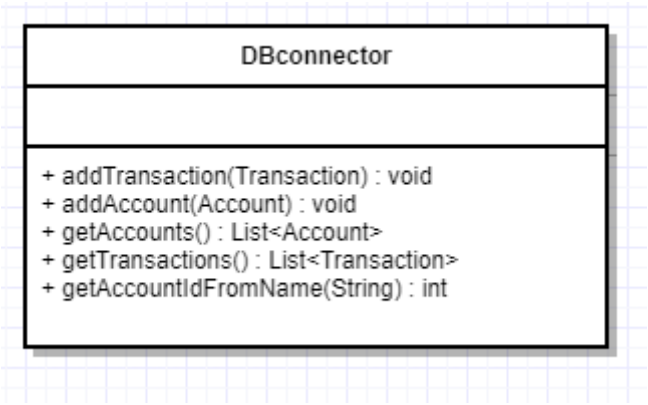
Enthält alle Transaktionen mit einer eindeutigen ID, der Account-ID, dem Betrag in Cent (Geldbeträge sollten aufgrund von möglichen Rundungsfehlern nicht als float/double gespeichert werden, sondern als Ganzzahl/Integer), Das Datum und Uhrzeit der Transaktion sowie eine optionale Beschreibung.

Funktion

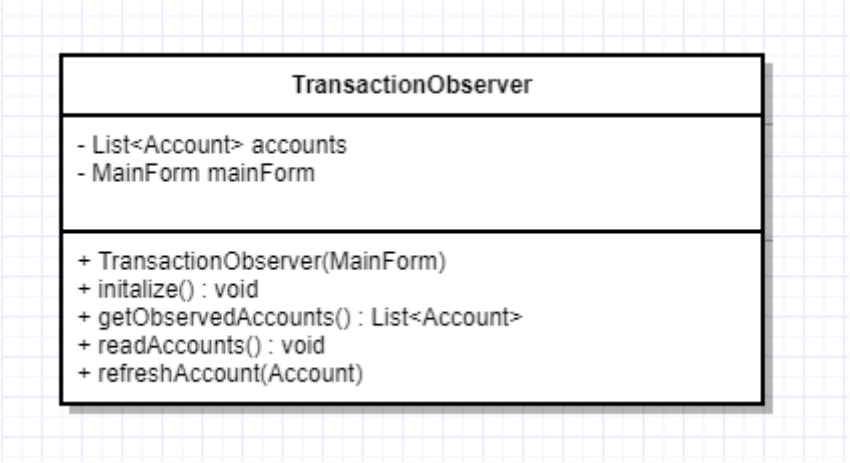


Von MainForm aus können neue Datensätze angelegt werden. Wird ein neues Konto (Account) angelegt, wird dieses über die NewAccount-Klasse, die ein Fenster zur Verfügung stellt, erzeugt.

Ebenso verhält sich eine neue Transaktion, die zusätzlich in der Transaktionen-Liste des zugehörigen Accounts landet. Über die statische DBconnector Klasse wird der Datenankzugriff geregelt. Es können Datensätze gelesen und geschrieben werden.



Hier ist die Statische Klasse „DBconnector“ dargestellt, die die Schnittstelle zwischen zur Datenbank bietet. Die Methoden „addTransaction“ und „addAccount“ legen einen Datensatz in der jeweiligen Tabelle an. Die Methode „getAccounts“ gibt eine Liste mit allen angelegten Accounts zurück. „getTransactions“ gibt eine Liste der Transaktionen eines bestimmten Accounts zurrück. „getAccountIdFromName“ nimmt einen Accountnamen an und wandelt ihn in die zugehörige ID um.



Die Attribute eines TransactionObserver-Objektes sind eine Liste der Account-Objekte, die überwacht werden und einen Verweis auf das MainForm-Objekt. Dieser wird dem Konstruktor beim Erstellen des Objekts als Parameter mitgegeben. Die initalize-Methode lädt zuerst alle Accounts und anschließend für jeden Account die entsprechenden Transaktionen. Die getObservedAccounts-Methode gibt die aktuelle Liste der Accounts zurück. Somit müssen diese nicht jedes Mal aus der Datenbank gelesen werden. Die Methode readAccounts liest die Accounts aus der Datenbank ein. RefreshAccount wiederum liest für den Angegebenen Account die Transaktionen aus der Datenbank.

Clean Architecture

Technologien

Es wurden die folgenden Technologien verwendet:

C#

Als Programmiersprache wurde C# ausgewählt, weil mir diese vertraut ist. Außerdem ist es mit Hilfe des .Net-Frameworks und Windows Forms einfach, Desktopanwendungen und deren GUI zu gestalten.

.Net Core 3.1.0

Das .Net-Framework wurde verwendet, da es das einfache Erstellen von GUI-Anwendungen für Windows-Maschinen ermöglicht. Frameworks besitzen den Nachteil, dass diese die Anwendung an sich binden. Es wird daher nahezu unmöglich, die Anwendung auf ein anderes Framework oder Libraries umzuziehen, ohne sie komplett neu zu schreiben.

SQLite

SQLite bietet eine sehr kompakte, dennoch vollständige Datenbank. Der große Vorteil ist, dass kein gesonderter Datenbank-Server installiert und konfiguriert werden muss. Eine SQLite-Datenbank ist in einer einzigen Datei untergebracht, die auf derselben Maschine gespeichert ist, die auch den Programmcode ausführt. Das macht sie Zwar nicht zentral administrierbar, aber jede Installation der Anwendung bringt seine eigene Datenbank mit sich und kann somit nur die eigenen Daten lesen und schreiben.

NUnit

Als Test-Framework wird NUnit eingesetzt. Es ermöglicht einfaches Testen von C# Code.

Schichtenarchitektur

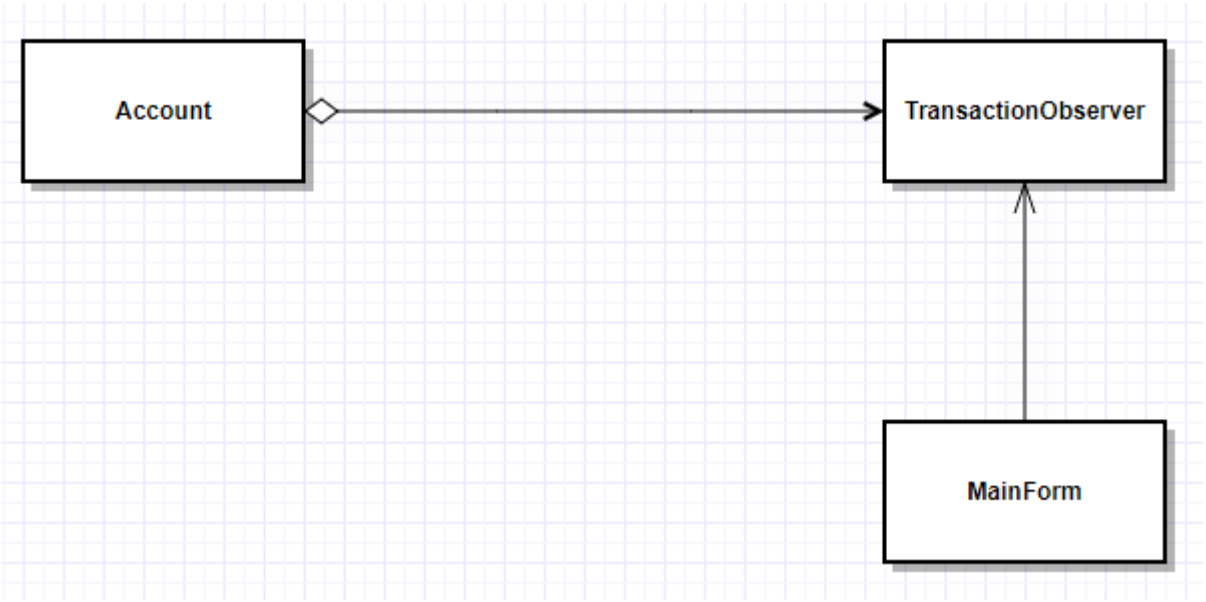
Die Anwendung wurde in zwei Schichten aufgebaut: der Application-Code und Plugins. Der Application Code umfasst die Klassen „Calculator“, „Dataset“, „Account“ und „Transaction“. Diese Klassen können einzeln kompiliert werden und sind weder vom .Net-Framework noch von anderen Klassen Abhängig.

Die GUI-Komponenten sind alle vom .Net-Framework abhängig und somit nicht teil des langlebigen Codes. Sie sind vom Lebenszyklus und der Verfügbarkeit des .Net-Frameworks abhängig.

Ebenso verhält sich die Klasse „DBconnector“. Sie ist die Schnittstelle zur SQLite Datenbannk. Anders als beim .Net-Framework lässt sich die Datenbank einfacher wechseln. Es muss lediglich die DBconnector-Klasse verändert werden, um eine andere Datenbanktechnologie verwenden zu können.

Entwurfsmuster

Implementiertes Entwurfsmuster: Beobachter



Ein TransactionOberserver-Objekt wird erzeugt, wenn ein neuer Account oder eine neue Transaction erzeugt wird. Der TransactionObserver hat eine Liste der Accounts, die er überwachen soll. Jedes Acount-Objekt wiederum hat eine Kiste der Transaktionen, die zu ihm gehören. Die MainForm kann den TransaktionObserver dazu veranlassen, auf neue Accounts zu prüfen. Dies wird zur Initialisierung beim Anwendungsstart genutzt. Wird ein neuer Account angelegt, wird er automatisch in die Liste der überwachten Accounts mit aufgenommen. Wird eine neue Transaktion angelegt, wird das dazugehörende Account Objekt dazu veranlasst, seine Transaktionen-List neu zu laden und so die neu angelegte Transaktion aufzunehmen.

Dieses Entwurfsmuster wird eingesetzt, da die Account-Objekte bei jedem Anlegen einer Transaktion ihren Zustand ändern.

Unit Tests

Für die Unit Tests wird das Framework „NUnit“ verwendet.

ATRIP

Automatic

Die Tests Laufen automatisch durch, kein Eingreifen erforderlich.

Thorough

Die Tests sind für dieses Projekt zu wenig und decken nur einen kleinen Teil des Codes ab.

Repeatable

Manche Tests sind nur wiederholbar, wenn sich der Zustand der Datenbank nicht ändert.

Independent

Die Tests sind voneinander unabhängig, sie laufen in egal welcher Reihenfolge.

Prifessional

Der Testcode ist leicht verständlich, die Tests sind kurz und lesbar.

Refactoring

Code Smells

Doppelter Code

```
public static int getAccountIdFromName(String pAccountName)
{
    using (IDbConnection cnn = new SQLiteConnection("Data Source =.\\Data.db; Version = 3"))
    {
        List<String> account = cnn.Query<String>("SELECT id FROM Accounts WHERE accountName = '" + pAccountName + "'").ToList();
        int accountId = Int32.Parse(account[0]);
        return accountId;
    }
}
```

Dieser Code dient dazu, anhand eines Account-Namens die Account-ID zu erhalten. Nach dem Aufbau der Datenbank-Verbindung wird die entsprechende SQL-Anweisung ausgeführt und das Ergebnis gespeichert. Der Rückgabewert wird in einen Interger umgewandelt und zurückgegeben. Diese Funktion wird benötigt, wenn ein Account oder eine Transaktion angelegt werden und die Liste der Transaktionen zu einem bestimmten Account gelesen werden soll. Dieser Code war somit in den Klassen MainForm (zum Abrufen der Transaktionen), NewTransaktion (zum Anlegen einer Transaktion) und NewAccount (zum Anlegen eines Accounts) enthalten. In diesem Zusand hat der Code eine gesonderte Funktion aufgerufen, die ein SQLQuery ausführet und das Ergebnis zurückgab. Um Doppelten Code zu vermeiden, wurde Diese Funktion in der Klasse DBconnector angelegt und die alte Funktion entfernt. Somit ist die Klasse auch kompakter.

Auskommentierter Code

```
/*
private static String loadConnectionString(string id = "Default")
{
    return "Data Source =.\\Data.db; Version = 3\" providerName=\"System.Data.SqlClient";
}
*/
```

Der oben gezeigte Kommentar aus der Klasse „DBconnector“ wurde als Test beim Einrichten der Datenbankverbindung erstellt, auskommentiert und vergessen. Solcher Code nimmt platz weg, verringert die Übersichtlichkeit des Codes und birgt die Gefahr, dass er später aus Unwissen wieder in den produktiven Code eingefügt wird und dann für Fehler sorgt.

```
public static void addAccount(Account pAccount)
{
    using (IDbConnection cnn = new SQLiteConnection("Data Source =.\\Data.db; Version = 3"))
    {
        //if(pAccount.getDescription() != "")
        //{
            cnn.Execute("INSERT INTO Accounts (accountName, description) values ('" + pAccount.getAccountName() + "', '" + pAccount.getDescription() + "')");
        //}
       // else
        //{
        //    cnn.Execute("INSERT INTO Accounts (accountName) values ('giroKonto')", pAccount);
        //}

    }
}
```

Die hier gezeigte Methode „addAccount“ hieß zuvor „AddDataset“ und sollte einen Datensatz in die Datenbank einfügen, egal ob es ein Account oder eine Transaction war. Diese Funktion war nie richtig implementiert, wesshalb der Code auch auskommentiert ist. Beim Implementieren der entsprechenden Funktionalität wurde festgestellt, dass es besser, einfacher und übersichtlicher ist, wenn es für beide Fälle eine gesonderte Methode gibt.

Nachdem die auskommentierten Codesegmente entfernt wurden, sieht die Methode „AddAccount“ wie folgt aus:

```
public static void addAccount(Account pAccount)
{
    using (IDbConnection cnn = new SQLiteConnection("Data Source =.\\Data.db; Version = 3"))
    {
        cnn.Execute("INSERT INTO Accounts (accountName, description) values ('" + pAccount.getAccountName() + "', '" + pAccount.getDescription() + "')");

    }
}
```

Große Klassen

```
public int calculateBalance(List<int> pAmounts)
{
    int balance = 0;
    foreach (int amount in pAmounts)
    {
        balance = +amount;
    }
    return balance;
}
```

Die MainForm Klasse wurde über den Prozess der Entwicklung immer größer, weshalb Funktionen wie „calculateBalance“, die aus einer Liste von Integer-Werten den Saldo berechnet, in die Klasse „Calculator“ ausgelagert. Dies ermöglicht außerdem, den Code mehrfach zu verwenden (siehe: Doppelter Code)

Refactorings

Extract Method

„Extract Method“ wurde angewendet, um den oben genannten Code Smell „Doppelter Code“ zu beheben. Der Code wurde aus drei verteilten Methoden in eine einzelne, neue Methode ausgelagert. Diese wird anstelle des alten Codes aufgerufen.

Somit gewinnen die drei Klassen MainForm, NewTransaction und NewAccount an Übersichtlichkeit und verlieren Länge. Sollte sich etwas im Code ändern, wenn beispielsweise eine Überprüfung hinzugefügt wird, kann dieser zentral modifiziert werden und gilt dann für alle aufrufe. Ist der Code verteilt, kann es passieren, dass eine Modifikation nur an zwei der drei Klassen durchgeführt wird, was einen Bug erzeugt.

Replace Temp with Query

```
public static void addTransaction(Transaction pTransaction)
{
    int accountId = pTransaction.getAccountId();
    int amount = pTransaction.getTransactionAmount();
    string dateTime = pTransaction.getTransactionDateTime();
    String description = pTransaction.getTransactionDescription();

    using (IDbConnection cnn = new SQLiteConnection("Data Source =.\\Data.db; Version = 3"))
    {
        cnn.Execute("INSERT INTO Transactions (accountId, amountEuroCents, transactionDateTime, description) values (" + accountId +
            ", " + amount + ", '" + dateTime + "', '" + description + "')");
    }
}
```

Die Methode „addTransaction“ der Klasse „Dbconnector“ erzeugt einen Datensatz in der Tabelle „Transactions“ mit hilfe des Transaction-Objekts, dass ihr beim Aufruf übergeben wird. Die

Zuweisung der Variablen kann weggelassen werden, indem die Getter-Aufrufe direkt in der SQL-Query ausgeführt werden.

```
public static void addTransaction(Transaction pTransaction)
{
    using (IDbConnection cnn = new SQLiteConnection("Data Source =.\\Data.db; Version = 3"))
    {
        cnn.Execute("INSERT INTO Transactions (accountId, amountEuroCents, transactionDateTime, description) values (" +
            pTransaction.getAccountId() + ", " + pTransaction.getTransactionAmount() + ", '" +
            pTransaction.getTransactionDateTime() + "', '" + pTransaction.getTransactionDescription() + "')");
    }
}
```

Die Methode ist jetzt kompakter und kann leichter geändert werden. Sollte sich der Code ändern und vergrößern, kann der Code leichter in eine andere Methode ausgelagert werden.