Programmentwicklung Advanced II
Modulübergreifende Projektarbeit

Generics

Ausgangslage

Sie haben in den letzten Aufgaben eine View inkl. ViewModel erstellt. Im ViewModel machen Sie Datenabfragen gegen Ihre Datenbank und zeigen die Daten in der View an. Um die Datenzugriffslogik (Datenabfragen usw.) zu separieren, soll diese Architektur nun durch einen DataAccess-Layer erweitert werden.

Auftrag

Abbildung 1 zeigt die klassische Three-Layer-Architecture für MVVM. In dieser Aufgabe soll nun für Ihre Applikation ein DataAccess-Layer implementiert werden.

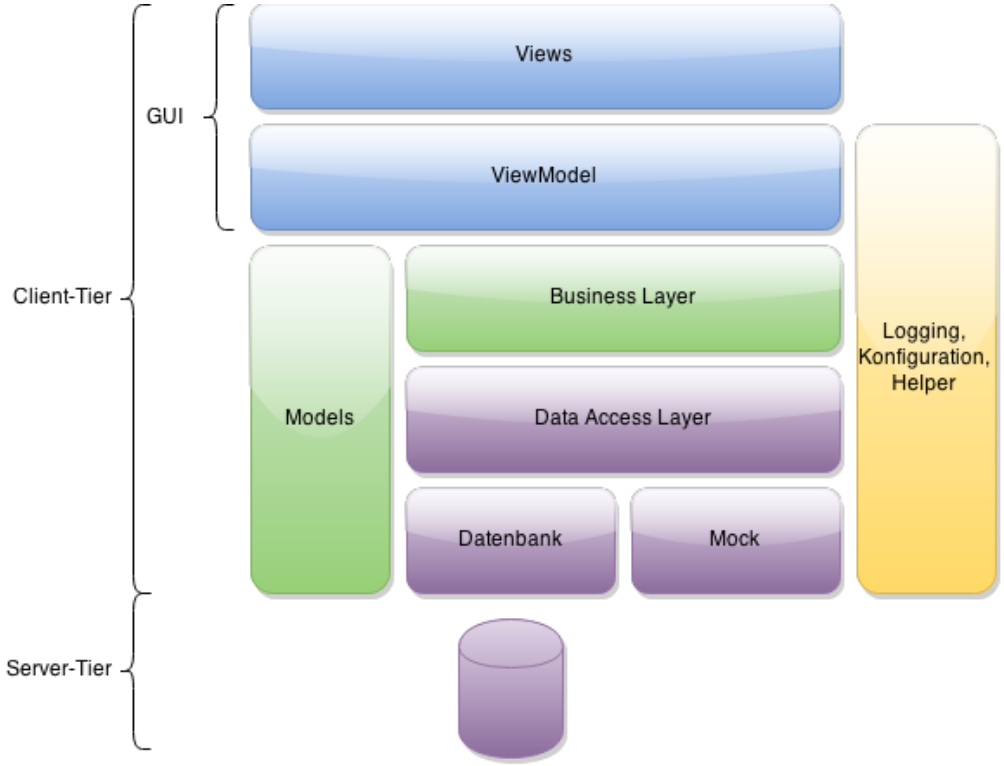

Abbildung 1: Three-Layer-Architecture MVVM

In diesem Layer soll das Repository-Pattern verwendet werden. Weitere Informationen zum Repository-Pattern finden Sie hier:
https://www.norberteder.com/das-repository-pattern-anhand-eines-beispiels-inkl-tests/

Das Repository kapselt die Logik für die Datenbankzugriffe (d.h. den kompletten ADO.NET-Teil) und gibt die Models an den BusinessLayer bzw. an die ViewModels weiter.
Im Gegensatz zu obigem Artikel soll ein generisches abstraktes Basis-Repository implementiert werden. Dieses Basis-Repository kann in der Folge als Basisklasse für konkrete Repositories verwendet werden.

Nachfolgend im Listing 1 ist das Interface für das generische Basis-Repository abgebildet. Beachten Sie die Beschreibung der einzelnen Member.

```
public interface IRepositoryBase<M>{
  /// <summary>
  /// Liefert ein einzelnes Model-Objekt vom Typ M zurück,
  /// welches anhand dem übergebenen PrimaryKey geladen wird.
  /// </summary>
  /// <typeparam name="P">Type des PrimaryKey</typeparam>
  /// <param name="pkValue">Wert des PrimaryKey</param>
  /// <returns>gefundenes Model-Objekt, ansonsten null</returns>
  M GetSingle<P>(P pkValue);

  /// <summary>
  /// Fügt das Model-Objekt zur Datenbank hinzu (Insert)
  /// </summary>
  /// <param name="entity">zu speicherndes Model-Object</param>
  void Add(M entity);

  /// <summary>
  /// Löscht das Model-Objekt aus der Datenbank (Delete)
  /// </summary>
  /// <param name="entity">zu löschendes Model-Object</param>
  void Delete(M entity);

  /// <summary>
  /// Aktualisiert das Model-Objekt in der Datenbank hinzu (Update)
  /// </summary>
  /// <param name="entity">zu aktualisierendes Model-Object</param>
  void Update(M entity);

  /// <summary>
  /// Gibt eine Liste von Model-Objekten vom Typ M zurück,
  /// die gemäss der WhereBedingung geladen wurden. Die Werte der
  /// Where-Bedingung können als separat übergeben werden,
  /// damit diese für PreparedStatements verwendet werden können.
  /// (Verhinderung von SQL-Injection)
  /// </summary>
  /// <param name="whereCondition">WhereBedingung als string
  /// z.B. "NetPrice > @netPrice and Active = @active and
  ///       Description like @desc</param>
  /// <param name="parameterValues">Parameter-Werte für die Wherebedingung
  /// bspw: {{"netPrice", 10.5}, {"active", true}, {"desc", "Wolle%"}}</param>
  /// <returns></returns>
  List<M> GetAll(string whereCondition, Dictionary<string, object> parameterValues);

  /// <summary>
  /// Gibt eine Liste aller in der DB vorhandenen Model-Objekte vom Typ M zurück
  /// </summary>
  /// <returns></returns>
  List<M> GetAll();

  /// <summary>
  /// Zählt in der Datenbank die Anzahl Model-Objekte vom Typ M, die der
  /// Where-Bedingung entsprechen
  /// </summary>
  /// <param name="whereCondition">WhereBedingung als string
  /// z.B. "NetPrice > @netPrice and Active = @active and
  ///       Description like @desc</param>
  /// <param name="parameterValues">Parameter-Werte für die Wherebedingung
  /// bspw: {{"netPrice", 10.5}, {"active", true}, {"desc", "Wolle%"}}</param>
  /// <returns></returns>
  long Count(string whereCondition, Dictionary<string, object> parameterValues);

  /// <summary>
  /// Zählt alle Model-Objekte vom Typ M
  /// </summary>
  /// <returns></returns>
  long Count();

  /// <summary>
  /// Gibt den Tabellennamen zurück, auf die sich das Repository bezieht
  /// </summary>
  string TableName { get; }

}
```

Listing 2: Interface für IRepositoryBase

Wie bereits erwähnt, soll die Datenbankanbindung mit ADO.NET erfolgen (gemäss Modul DB sowie Einführung in DBAdv).

Dieses Interface kann nun in Form einer abstrakten Klasse `RepositoryBase` implementiert werden (siehe Listing 2). Diese abstrakte Basisklasse beinhaltet in der Folge die Logik, die für alle Repositories gemeinsam ist. Als Beispiel könnte die Methode `Count()` dort implementiert werden. Dies könnte wie in Listing 2 dargestellt aussehen.

```
public abstract class RepositoryBase<M> : IRepositoryBase<M>{
    protected RepositoryBase() {
        // TODO: Load ConnectionString
        this.ConnectionString = "<ConnString>";
    }

    protected string ConnectionString { get; }

    public long Count() {
        using (var conn = new MySqlConnection(this.ConnectionString)) {
            using (var cmd = conn.CreateCommand()) {
                conn.Open();
                cmd.CommandText = $"select count(*) from {this.TableName}";
                return (long) cmd.ExecuteScalar();
            }
        }
    }

    public abstract string TableName { get; }

    /* TODO: Implement Interface Members */
}
```

Listing 2: abstrakte Basisklasse RepositoryBase

Anschliessend kann die Klasse `RepositoryBase` für konkrete Repositories verwendet werden. Listing 3 zeigt bswp. die Signatur für das konkrete `LocationRepository`, welches Location-Objekte laden, speichern usw. kann.

```
public class LocationRepository : RepositoryBase<Location>{
    public override string TableName => "Location";

    /* TODO: Implement Members */
}
```

Listing 3: konkrete Implementierung des Repositories für Location

Das konkrete Repository kann auch weitere Methoden aufweisen, als das `RepositoryBase` bzw. das Interface `IRepositoryBase` vorgibt. Z.B. könnte es eine Methode geben, die ein bestimmtes StoredProcedure aufruft oder Daten auf Basis eines komplexen Joins lädt usw. D.h. Logik, die spezifisch nur für einen bestimmten Model-Typ benötigt wird.

Ihre Aufgabe besteht nun darin, das Repository-Pattern in generischer Form gemäss obiger Beschreibung in Ihrer Anwendung umzusetzen. Konkret implementieren Sie in Ihrer Anwendung einen DataAccess-Layer, welcher alle Datenbankzugriffe kapselt. D.h. in Ihren ViewModels bzw. Models darf es anschliessend keine direkte Interaktion mit der Datenbank mehr geben.
Implementieren Sie mindestens zwei Repositories:

- Locations (Tabelle Location – achten Sie, dass Sie die Hierarchie im Model auch abbilden – d.h. Locations können hierarchisch strukturiert werden)
- Loggings (View vLogEntries – überlegen Sie sich, ob es sinnvoll ist, alle Methoden zu implementieren. Methoden die Sie nicht implementieren könnten z.B. eine `NotImplementedException` auslösen.)

Das könnte bspw. so aussehen wie in Abbildung 2 dargestellt.

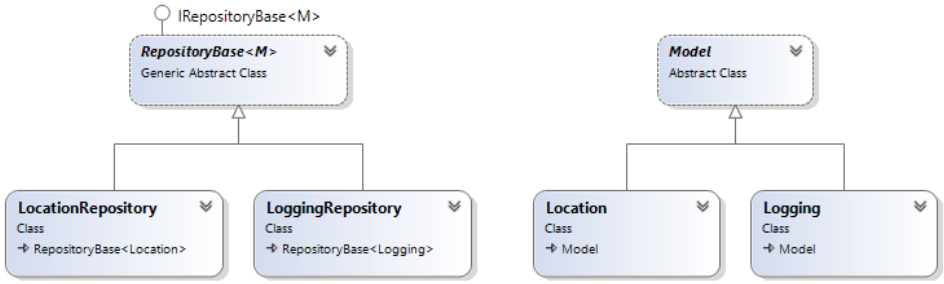

Abbildung 2: Datenmodell Repositories und Models

Die Implementierung eines DataAccess-Layers gibt Ihnen anschliessend die Möglichkeit, die Datenquelle (bzw. das Framework für die Interaktion mit der Datenquelle) zukünftig einfach ausgetauscht werden kann. Dies werden wir uns zunutze machen, wenn wir im Rahmen des Moduls «Datenbanken Advanced» EntityFramework für die Interaktion mit der Datenbank verwenden.

Lieferergebnis	Als Abgabe wird der komplette Sourcecode als ZIP-Datei erwartet. Zusätzlich soll eine Kurzdokumentation erstellt werden, aus der die Handhabung Ihrer Anwendung hervorgeht.
Bewertung	Gemäss Initialdokument
Termin	Gemäss Moodle-Abgabe