OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Entwicklung eines Software-Tools zur automatisierten Ableitung eines Struktur- und Volumenmodells in eine SQLite Datenbank mit Hilfe der Revit API

Fakultät für Bauingenieurwesen
der Ostbayerischen Technischen Hochschule Regensburg

Masterarbeit

zur Erlangung des akademischen Grades
Master of Engineering

vorgelegt von

Friedrich Eder
geboren am 07.02.1988 in Deggendorf

am 07.02.2019

Erstprüfer: Prof. Dr.-Ing. T. Euringer
Zweitprüfer: Prof. Dr.-Ing. M. Obergrießer

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

Danksagungen

An dieser Stelle möchte ich mich bei allen Personen, die zur Entstehung dieser Masterarbeit beigetragen haben bedanken.

Mein Dank gilt insbesondere Herrn Professor Dr. Dipl.-Ing. Thomas Euringer für sein Interesse am Gelingen und seinem Einsatz als Betreuer während meiner Masterarbeit.

Herr Dipl.-Ing. Jürgen Maria Schmidt hat es erst möglich gemacht, die Masterarbeit extern bei der Firma Fides DV-Partner GmbH zu schreiben. Um mir das Pendeln zwischen Regensburg und München zu ersparen, brachte Herr Schmidt mir sehr viel Vertrauen entgegen und ermöglichte mir das Arbeiten am Heimarbeitsplatz. Hierfür möchte ich mich ausdrücklich bedanken.

Ein besonderes Dankeschön darf ich Herrn Dipl.-Ing. Gernot Frisch und Herrn Dipl.-Ing. Matthias Filus aussprechen, die sich stets Zeit für mich nahmen und an ihrem Erfahrungsschatz teilhaben ließen.

Herzlich danke ich auch meiner Frau, die mich während meines Studiums unterstützte und immer Interesse an meinem Fortgang zeigte.

Inhaltsverzeichnis

Eidesstattliche Erklärung . . . 2
Danksagungen . . . 3

1 Einleitung **5**

2 Aufgabenstellung **6**
2.1 Anforderungen an das Tool . . . 6
2.2 Lösungsansatz . . . 8

3 Theoretische Grundlagen **9**
3.1 Begriffsklärung . . . 9
3.2 Datenbanken . . . 11
3.2.1 Relationale Datenbanken . . . 11
3.2.2 Objektorientierte Datenbanken . . . 13
3.3 Autodesk Revit 2019 API . . . 14
3.3.1 Struktur der Schnittstelle . . . 14
3.3.2 Revit Modellelemente . . . 16

4 Umsetzung **17**
4.1 Überblick der Module . . . 18
4.2 Einbindung der Extension in das Revit-Framework . . . 19
4.3 Strukturierung der Modelldaten . . . 21
4.3.1 Datenhaltung . . . 22
4.3.2 Beschaffung und Verwaltung der Modelldaten . . . 28
4.4 Datenbankstruktur . . . 31
4.5 Schnittstelle zur SQLite Datenbank . . . 34
4.6 Anwendungsbeispiel . . . 37

5 Ausblick **44**

1 Einleitung

Die technischen Fortschritte der Bauindustrie werden in vielen Fällen nur dann ausgeschöpft, wenn eine einheitliche Datenbasis für ein entsprechendes Bauvorhaben vorhanden ist. In Zeiten moderner Vermessungstechniken, hohem Automationsgrad der Bautechnik und aufwendigen FE-Berechnungen sind große Datenmengen entweder Grundlage oder Ergebnis der Planung. Die zunehmende Informationsdichte macht eine konventionelle, (semi-)manuelle Verarbeitung der Daten nahezu unmöglich.

Über das Prinzip des *Building Information Modelling* bekommen die erfassten Daten einen Kontext der allen Projektbeteiligten in Planung und Ausführung als einheitliche Datenbasis dient. Dies ermöglicht Informationen kontextbezogen aufzubereiten, also bspw. die Bauwerksdaten entsprechend den Gewerken zu filtern und darüber wiederum Lese- und Schreibrechte einer Teilmenge der Daten zu verwalten.

Das BIM-Prinzip wird von zahlreichen Softwareherstellern umgesetzt und weiterentwickelt. Allerdings gibt es immer wieder Fälle in denen die Strukturen die bspw. ein IFC-Datensatz zur Verfügung stellt nicht ausreichen um ein bestimmtes Problem effizient und zukunftssicher zu lösen. In diesen Fällen müssen neue Schnittstellen geschaffen werden, die den Anforderungen gerecht werden.

Die Firma *FIDES-DV Partner GmbH* vertreibt und entwickelt Softwarelösungen für viele Bereiche des Bauwesens. Von selbst-entwickelter Berechnungssoftware für Tunnelbau und Geotechnik bis hin zu allumfassenden BIM-Strukturen beschäftigt sich die Firma mit dem Lösen der Aufgaben des modernen Bauwesens. Im Zuge der Weiter- und Neuentwicklung der FIDES-Softwarelösungen soll
eine Schnittstelle zwischen dem Autodesk BIM-Tool *Revit* und einer Datenbank
erstellt werden. Weil das Visualisieren und Planen oft nicht in der selben Software erfolgt in der die Berechnungen durchgeführt werden, muss im Laufe der Planung und Ausführung immer ein Datentransfer der jeweils relevanten Daten in das entsprechende Software-Tool erfolgen. Dies kann - wie oben bereits angesprochen - bei großen Datenmengen (Modellen) sehr aufwendig und fehleranfällig sein. Ein automatischer Export und Import der Modelldaten in und über eine zentrale Datenbasis kann hier eine große Erleichterung für den Sachbearbeiter darstellen.

2 Aufgabenstellung

Im Rahmen dieser Arbeit soll der Entwicklungsprozess und die Überlegungen für eine automatisierte Datenverarbeitung am Beispiel der Ableitung von Volumen- und Strukturmodellen aus *Revit* in eine Datenbank beschrieben werden. Nachfolgend werden die allgemeinen Anforderungen an die Software beschrieben.

2.1 Anforderungen an das Tool

Die erstellte Datenbank soll so aufgebaut sein, dass die Modelldaten einheitlich und nicht redundant gespeichert werden. Volumenmodelle sind über entsprechende Relationen in kleinere „Bausteine“ zu zerlegen. Zunächst soll nur der Datenexport umgesetzt werden. Der Import ist aber bei der Entwicklung mit zu berücksichtigen.

Es ist damit zu rechnen, dass über die Schnittstelle in Zukunft sehr große Datenmengen verarbeitet werden. Somit sollte ein Austauschformat gewählt werden, dass großen Datenmengen verarbeiten kann, ohne dabei zu viel Systemressourcen zu verwenden.

Bei großen Bauvorhaben, sind viele Projektbeteiligte an einer Datenbasis tätig. Bei Änderungen findet daher ein reger Austausch von Daten statt. Hier macht es also Sinn, den Export und Import über eine einzige Datei abzuwickeln, die direkt von der Festplatte des Benutzers weitergeleitet werden kann. Es sei aber angemerkt, dass das manuelle Verschicken von Dateien durch den Benutzer zu einem späteren Zeitpunkt - wenn durch die Datenbankschnittstelle auch die Versionsverwaltung der Datensätze erfolgt - nicht mehr nötig sein wird.

Für den Anwender und den Entwickler gleicher Maßen ist es sehr von Vorteil wenn die Datenbasis einsehbar und lesbar ist. Dies trägt dazu bei Fehler in den Quelldaten zu erkennen. Des Weiteren kann das „offene“ Datenhaltungskonzept einer Datenbank leichter überprüft und nachvollzogen werden. Würde man bspw. ein binäres Datenformat wählen, so ist es nahezu unmöglich nach dem Export die Daten in der Datei einzusehen und ggf. Fehler aufzudecken oder die Datenstruktur zu analysieren.

Neben den Anforderungen an die Funktionalität des Programms ist auch die Zukunftssicherheit und Erweiterbarkeit zu berücksichtigen. So ist es entscheidend für die Nutzung einer Software welche Systemvoraussetzungen zugrunde gelegt sind. Es sollten Situationen vermieden werden in denen bspw. bei der Umstellung auf ein neues Betriebssystem sämtliche Grundfunktionen des Tools wegfallen. Auch ist darauf zu achten das Softwareprojekt und die Objektstruktur so zu gliedern, dass die Objekte selbst über klar definierte Interfaces

miteinander kommunizieren, sodass gezielte Anpassungen an der Software nicht damit einhergehen, dass die gesamte Codebase erweitert werden muss.

Letztlich ist es bei der Erstellung einer neuen Software für das Arbeiten im Team von großer Bedeutung die Entwicklungsfortschritte so genau wie nötig und so einheitlich wie möglich zu dokumentieren. Der einzelne Entwickler sollte in steter Rücksprachen mit seinen Kollegen die Vorgehensweise hinterfragen und erläuternde Kommentare zu den Funktionen und Datenstrukturen hinterlassen.

Zusammenfassend ergibt sich aus den genannten Anforderungen folgende konkrete Probleme:

- Automatisiertes Auslesen der Revit Modelldaten
  - Datenaustausch über Revit API
  - Filtern nach definierten bzw. relevanten Daten
- Überführung der Objektorientierten Datenbank `Autodesk.Revit.DB` in eine relationale Datenbank
  - Schnittstelle zu einer Datenbank implementieren
  - Wahl geeigneter Tabellenbeziehungen (1:1, 1:n, m:n)
- Programmiersprache für das Tool festlegen
  - Funktionsumfang definieren
  - geeignetes Objektmodell für das Software-Tool entwickeln

2.2 Lösungsansatz

Die Daten des Struktur- und Volumenmodells werden innerhalb einer in *Revit* eingebetteten Extension ausgelesen. Dadurch ist zur Laufzeit des Programms durch die *Revit API* der nahezu vollständige Programmzugriff gewährleistet.

Die Extension wird in der Programmiersprache *C#* entwickelt. Diese moderne und vielseitige Programmiersprache stellt eine große Auswahl von Open-Source-Libraries für die Funktionserweiterung zur Verfügung. Dadurch ermöglicht *C#* einen recht schnellen Entwicklungsfortschritt, da man nicht ausschließlich auf selbst entwickelte Module zurückgreifen muss. Auf die Stabilität und „Zukunftssicherheit“ wirkt sich die Nutzung von Open-Source Modulen u.U. negativ aus, denn Fehlerkorrekturen oder Aktualisierungen an den eingebundenen Modulen sind (zeitnah) i.d.R. nur durch den Herausgeber möglich. Daher wird bei der Entwicklung nur auf die nötigsten externen Module zurückgegriffen (hier: `Autodesk.Revit`-API). Alle weiteren eingebundenen Klassenbibliotheken sind *.NET-Framework* (Windows) Standard-Systembibliotheken.

C# ermöglicht die Entwicklung von Software nach dem Paradigma der Objektorientiertheit (OOP). Da sich die gewünschte Funktionalität des Tools sehr gut in einer deklarativen Form über Datencontainer und Methoden beschreiben lässt, bietet sich ein Objektorientiertes Software-Konzept an. Bei sorgfältigen Vorüberlegungen wird eine Programmstruktur geschaffen die zum einen den logischen Zusammenhang der Datenbeschaffung und -verarbeitung verständlich abbildet und zum anderen eine eventuelle Funktionserweiterung durch klar definierte Module und Ausnutzung der Konzepte *Abstraktion* und *Polymorphie* erleichtert.

Der Datenaustausch zwischen *Revit* und den *FIDES*-Softwareprodukten wird über eine relationale *SQLite*-Datenbank realisiert. Der Vorteil einer *SQLite*-Datenbank liegt hier besonders darin, dass keine Serverinstanz auf dem Gerät des Endnutzers vorhanden sein muss. Eine *SQLite* Datenbank zeichnet sich besonders dadurch aus, dass die Datenbank in einer einzigen Datei gehalten wird, dies macht die Datenbank sehr portabel. Über eine *Dynamic Link Library (DLL)* werden die notwendigen Funktionen für eine Schnittstelle zur *SQLite*-Datenbank in das Programm eingebunden. Somit bleibt eine mitunter aufwendige Serverinstallation aus, denn die Datenbank wird über die Software selbst implementiert. Einzig die entsprechende *DLL* muss bei der Installation des Tools mit „installiert“ werden.

SQLite-Datenbanken können über das Tool *DB Browser for SQLite*[1] geöffnet und editiert werden. Dies erleichtert die Fehlersuche da die Datenbank direkt einsehbar ist und die enthaltenen Daten nicht binär Serialisiert werden. Relationale Datenbanken weisen im Vergleich zu anderen Formaten für die Datenhaltung und den Datenaustausch (*XML* oder *JSON*) eine wesentlich höhere Lese- und Schreibgeschwindigkeit auf und wie in 2.1 beschrieben ist mit sehr großen Datenmengen zu rechnen.

[1] https://sqlitebrowser.org/

3 Theoretische Grundlagen

3.1 Begriffsklärung

API (application programming interface)
Eine Programmierschnittstelle, die die Nutzung von Funktionen oder ganzen Programmen innerhalb selbst-entwickelte Software ermöglicht.

Dynamic Link Library (DLL)
Eine DLL-Datei enthält Methoden und Objektinformationen die in ein Projekt geladen werden können. In DLL-Dateien werden bei großen Softwareprojekten häufig einzelne Programmmodule gehalten. Hier nutzt man die Tatsache, dass DLL-Dateien auch erst zur Laufzeit des Programms geladen werden können. Dies reduziert den RAM-Bedarf einer Applikation enorm, z.b. bei FE-Berechnungen erst beim Auslösen der Berechnung der RAM-Bedarf steigt und nach Abschluss der Berechnung (wenn die DLL wieder entladen wurde) wieder abfällt. Auch bei der Einbindung von Schnittstellen über eine API kommen häufig DLL-Dateien zum Einsatz. Hier stellen Softwarehersteller eine abgeschlossene Funktions-Bibliothek zur Verfügung die nicht von „außen“ geändert werden kann.
Eine *Dynamic Link Library* ist kompiliert, das heißt ihr Inhalt kann üblicherweise nicht nachträglich geändert werden.

OOP (Objektorientierte Programmierung)
Das Prinzip der Objektorientierten Entwicklung beinhaltet, dass Programmteile ihrer semantischen Bedeutung nach gegliedert werden. Ein Objekt wird abstrakt als Klasse definiert, welche alle zum Objekt gehörigen Eigenschaften und Funktionen abbildet. Wenn man beispielsweise eine Fundamentplatte im Programm abbilden will, so hätte die Klasse *Fundamentplatte* bspw. die Eigenschaften *Dicke*, *Länge*, *Breite* und*Material*.
Eine Funktion der Klasse könnte dann über das Volumen und der Dichte als Eigenschaft des Materials das Eigengewicht ermitteln. Diese Zusammenhänge sind für den Mensch klar und logisch, sie ermöglichen - bei strikter Einhaltung dieses Prinzips - eine übersichtliche Programmstruktur.

SQL (structured query language)
Datenmanipulationssprache zur Definition von Datenstrukturen in relationalen Datenbanken. Die Sprache ermöglicht das Bearbeiten (Einfügen, Verändern, Löschen) und Abfragen von darauf basierenden Datenbeständen.

SQLite

Im Gegensatz zu den meisten anderen SQL-Datenbanken benötigt SQLite keinen separaten Serverprozess. SQLite liest und schreibt direkt in gewöhnliche Datenträgerdateien. Eine vollständige SQL-Datenbank mit mehreren Tabellen, Indizes, Triggern und Ansichten ist in einer einzigen Datenträgerdatei enthalten. Das Datenbankdateiformat ist plattformübergreifend. Eine Datenbank kann frei zwischen 32-Bit- und 64-Bit-Systemen kopiert werden. Diese Funktionen machen SQLite zu einer beliebten Wahl als Anwendungsdateiformat. [1]

3.2 Datenbanken

Als Datenbank bezeichnet man im Allgemeinen ein elektronisches Verwaltungssystem, dass immer dann verwendet wird, wenn große Datenmengen zu verwalten sind. Hier ist es besonders wichtig, dass die Datenbankstruktur effizient, eindeutig und dauerhaft mit den enthaltenen Daten umgeht, um logische Zusammenhänge digital abzubilden.
Entsprechend des nach Einsatzgebiets, kommen in der Praxis verschiedene Grundstrukturen einer Datenbank zum Einsatz.

3.2.1 Relationale Datenbanken

Bei dieser Form der Datenbank werden Daten über sogenannte Relationen abgebildet. Eine Relation entspricht einer Tabelle in der Datenbank. Eine Tabelle einer Relationalen Datenbank ist durch folgende Eigenschaften gekennzeichnet:

- eindeutiger Name der Tabelle
- mehrere Attribute (Spalten)
- keine bis beliebig viele Tabellenzeilen bzw. Datensätze
- ein Wert pro Attribut je Tabellenzeile
- Primärschlüssel, der jedes Tupel eindeutig identifiziert

Die Spalten einer Tabelle werden als Attribute oder Felder bezeichnet, die Zeilen entsprechen Tupel bzw. Datensätzen. Der Aufbau der Attribute der Tabelle wird als Struktur oder Schema bezeichnet. Die Summe der Relationsschemata der Datenbank entspricht dem Datenbankschema.[2]

Id	**Revit_Id**	**Material_Id**	**Tag**	**FaceList_Id**
52	321684765	1	Column	52
58	321698732	50	Floor	121
105	879867654	12	analyticalSolid	336

Tabelle 3.1: Beispiel einer Tabelle

Datentypen

Jedes Attribut wird einem bestimmten Datentyp zugeordnet. Relationale Datenbanken stellen je nach Implementierung verschiedene Datentypen zur Verfügung. Nachfolgend die für eine SQLite Datenbank relevanten Datentypen:

- INTEGER:
  Ganzzahliger Wert mit Vorzeichen, je nach Größe des Zahlewerts bis zu 8 bytes groß
- REAL:
  8-byte IEEE Gleitkommazahl, Genauigkeit bis zu 15 Stellen nach dem Komma
- TEXT:
  Text String im encoding-format der Datenbank (UTF-8, UTF-16BE oder UTF-16LE)
- BLOB (Binary Large Objects):
  Wird dann verwendet, wenn sehr große Daten wie bzw. Bilder oder Musik abgespeichert werden sollen

Datum und Uhrzeit kann über über folgende Datentypen abgebildet werden:

- INTEGER: Sekunden seit 1970-01-01 00:00:00 UTC
- REAL: Anzahl der Tage seit 12:00 Uhr Mittag in Greenwich am 24. Nov, 4714 v. Chr.
- TEXT: Datumsformat bspw.: "YYYY-MM-DD HH:MM:SS.SSS"

Anmerkung:
Im Gegensatz zu „reinem“ SQL wird in SQLite die Größe des Datentyps nicht fest vorgegeben. Vielmehr wird über den sog. Datentyp ein Wertebereich festgelegt. Je nach Größe des übergebenen Attributs werden für bspw. den Typ INTEGER 1, 2, 3, 4, 6 oder 8 Byte im Speicher der Datenbank reserviert. Dieses Verhalten wird im Allgemeinen als *Affinität der Datentypen* bezeichnet.

Beziehungen von Datensätzen

- **1:1 Beziehung:**
  Jeder Eintrag einer Tabelle ist genau **einem** Eintrag einer anderen Tabelle zugeordnet.

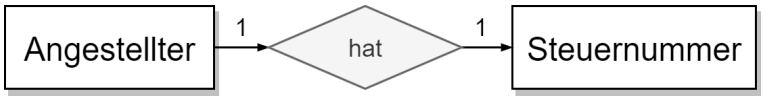


Abbildung 3.1: Beispiel 1:1 Beziehung

- **1:n Beziehung:**
  Jeder Eintrag einer Tabelle ist **einer oder mehreren** Einträgen einer anderen Tabelle zugeordnet.

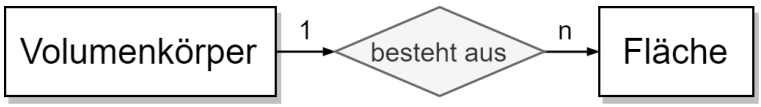


Abbildung 3.2: Beispiel 1:n Beziehung

- **m:n Beziehung:**
  Eine oder mehrere Einträge einer Tabelle können können **einem oder mehreren** Einträgen einer anderen Tabelle zugeordnet werden.

Abbildung 3.3: Beispiel m:n Beziehung

3.2.2 Objektorientierte Datenbanken

Wie bei objektorientierten Programmiersprachen (bspw. C++, C# oder Java) ist die Grundidee eines objektorientierten Datenbankmanagementsystems (OODBMS), Datenstrukturen zu Kapseln und über Methoden, Typenhierarchien und Polymorphie zu verwalten. Die Kapselung der Datenstrukturen soll ermöglichen, dass Teile der Struktur bei Bedarf ausgetauscht werden können, ohne dabei Nebenwirkungen auf andere Teile der gesamten Struktur mit sich zu bringen.
Die Vorteile der *OOP* sollen also auf Datenbankstrukturen angewandt werden, deren Objekte am Ende der Programmausführung persistent gespeichert werden sollen.
Die in Abschnitt 3.2.1 gezeigte flache Datenstruktur wird durch die Kapselung der Daten ersetzt.[3]

3.3 Autodesk Revit 2019 API

Das von *Autodesk* zur Verfügung gestellte *Application Programming Interface* ist mit allen Sprachen die das *Microsoft .NET Framework 4.7* unterstützen kompatibel. Die API ist so gestaltet, dass es möglich ist nahezu alle Funktionen die über das User-Interface zur Verfügung stehen auch in Form von programmierten Routinen umzusetzen.
Laut Autodesk [4] eignet sich die *Revit API* besonders für folgende Anwendungsbereiche:

- Erstellen von Add-Ins und Makros um wiederkehrende Abläufe zu automatisieren
- Umsetzung von Projektdesign-Standards über automatisierte Konformitätsprüfung
- Automatisierte Dokumentation eines Revit Projekts
- **Projekt- und Modelldaten für weiterführende Berechnungen exportieren**
- **Externe Datenquellen für die Modellerstellung einbinden**
- **Software von Drittanbietern in ein Autodesk Revit Produkt einbinden**

Anmerkung:
Alle Erläuterungen zur Funktion und Struktur der angesprochenen Objekte sind aus dem *„Revit API Developers Guide“* [4] entnommen und vom Verfasser ins Deutsche übersetzt.

3.3.1 Struktur der Schnittstelle

Die beiden „Top-Level“ Objekte der Revit API sind die *Applikation* und das *Dokument*. Ein *Dokument* wird innerhalb der *Applikation* gehalten und besteht aus einer Reihe von *Elementen*.

Applikation

Bezieht sich auf eine einzige Instanz der Autodesk Revit Anwendung und stellt Zugriff auf Dokumente, Optionen und weitere globale Einstellungen und Daten zur Verfügung.
Seitens der Schnittstelle wird zwischen zwei Objekten für den Applikationszugriff unterschieden:

- `Autodesk.Revit.UI.UIApplication` - Zugriff auf User-Interface Funktionalität (Hinzufügen von Steuerelementen, Zugriff auf aktives Dokument, etc.)
- `Autodesk.Revit.ApplicationServices.Application` - Zugriff auf alle weiteren Funktionen auf Applikations-Level

Dokument

Entspricht einer einzigen Revit Projektdatei, die ein Bauwerksmodell enthält. Eine Anwendungsinstanz kann mehrere Dokumente enthalten.
Seitens der Schnittstelle wird zwischen zwei Objekten für den Dokumentenzugriff unterschieden:

- `Autodesk.Revit.UI.UIDocument` - Zugriff auf User-Interface Funktionalität des Dokuments (Inhalt einer Auswahl, Modelldaten des aktiven Dokuments, etc.)
- `Autodesk.Revit.DB.Document` - Zugriff auf alle weiteren Funktionen auf Dokumenten-Level

Element

Der kleinste gemeinsame Nenner der Daten in einem Revit-Dokument ist das sogenannte *Element*. Das *Element* ist ein abstraktes Objekt, das als Basis für alle Bauwerksbezogenen Daten im *Dokument* verwendet wird. Neben einem eindeutigen Kenner, kann das *Element* u.a. einen Namen, eine Lage und eine Geometrie zur Verfügung stellen.

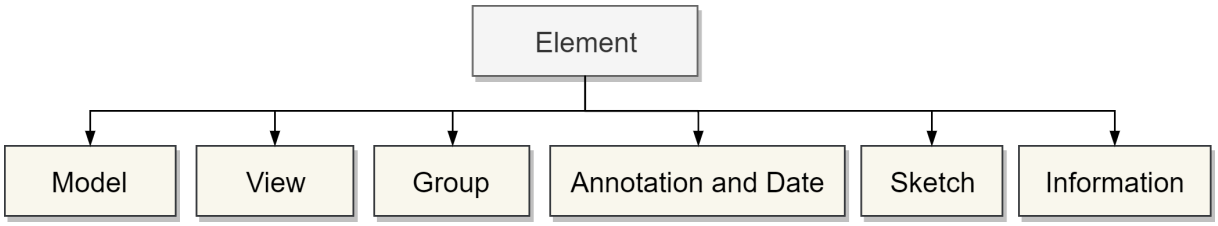


Abbildung 3.4: Hierarchie der Klasse *Element*

Wie in Abb. 3.4 erkennbar ist, stellt das *Element* einen Basis-Typ dar, der entsprechend seiner näheren Klassifizierung weitere Eigenschaften zur Verfügung stellt. In Revit gibt es insgesamt sechs verschiedene Element-Gruppen:

1. Modellelemente (*Model Elements*)
   - Instanzen von Revit-Familien (*Family Instance*)
   - Objekte die weitere Element-Instanzen beinhalten können (*Host Elements*)
   - Statisch relevante Elemente (*Structure Element*)
   - Materialeigenschaften für Berechnung und Rendering (*Material Elements*)
2. Ansichts Elemente (*View Elements*)
3. Gruppenelemente (*Group Elements*)
4. Anmerkung und Datum (*Annotation and Date Elements*)
5. Temporäre Skizzen und Entwürfe (*Sketch*)

6. Unsichtbare nicht physikalische Projekt und Applikationsdaten (*Information*)

Im Hinblick auf die Aufgabenstellung der zu entwickelnden Software wird sich im Folgenden auf die genauere Beschreibung der Modellelemente beschränkt.

3.3.2 Revit Modellelemente

Ein *Element* dessen Geometrie-Eigenschaft einen Verweis auf die Geometrie eines Objekts enthält, wird als Modellelement bezeichnet. Wie unter 3.3.1 beschrieben wurde, kann diese Geometrie in eine `FamilyInstance`, ein `HostElement` oder ein `Structure`-Objekt eingebunden sein. Unabhängig vom Typ sind die Geometrie-Daten immer gleich strukturiert.
In Abb. 3.5 ist die Hierarchie der Geometrie-Objekte dargestellt. Die strikte Kapselung der Objekte innerhalb der Revit API wird hier deutlich erkennbar. So sind Geometrieelemente wie *Faces* und *Edges* innerhalb der zugehörigen *Solids* gehalten und sind dadurch stark aneinander gebunden.
Dies ist besonders beim Auslesen der Objektdaten von Bedeutung, denn in den entsprechenden „Lese-Routinen“ muss diese verschachtelte Objektstruktur mit berücksichtigt werden.

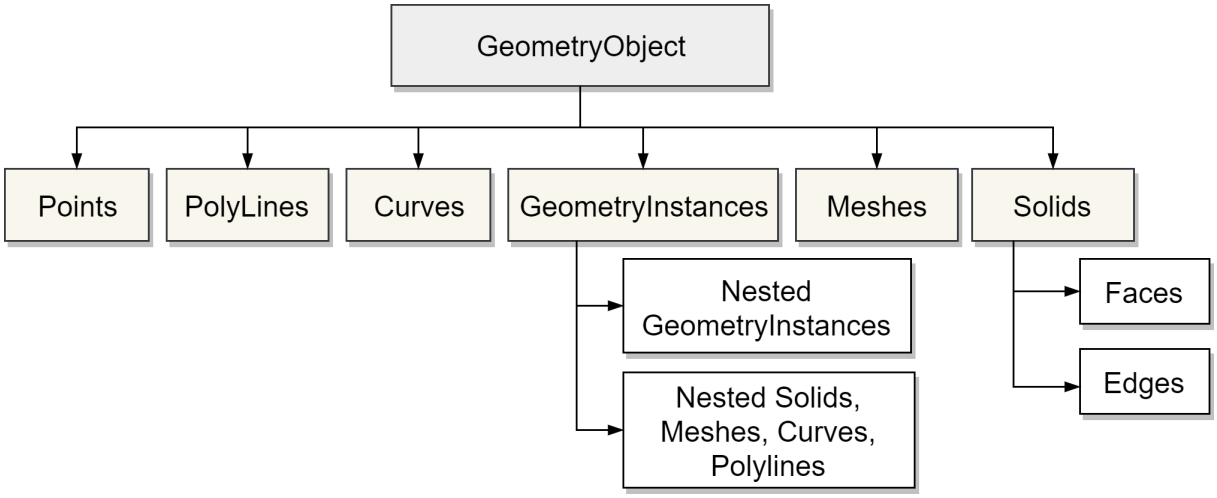


Abbildung 3.5: Hierarchie der Klasse *GeometryObject*

4 Umsetzung

Wie bereits unter Abschnitt 2.2 angesprochen, wird die Schnittstelle in Form einer Revit-Extension in C# entwickelt. Bei dem Objektorientieren Entwicklungsansatz steht der logische Zusammenhang zwischen den einzelnen Programmbausteinen im Vordergrund.

Der Ablauf des Programms kann allgemein folgendermaßen zusammengefasst werden:

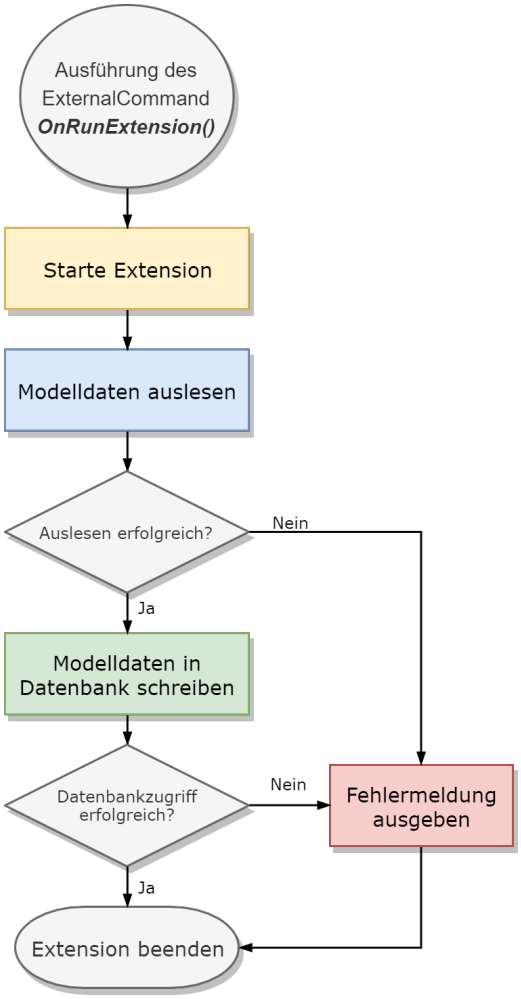


Abbildung 4.1: Ablaufschema der Extension

Nachfolgend wird zunächst das Objektmodell des Tools beschrieben.

4.1 Überblick der Module

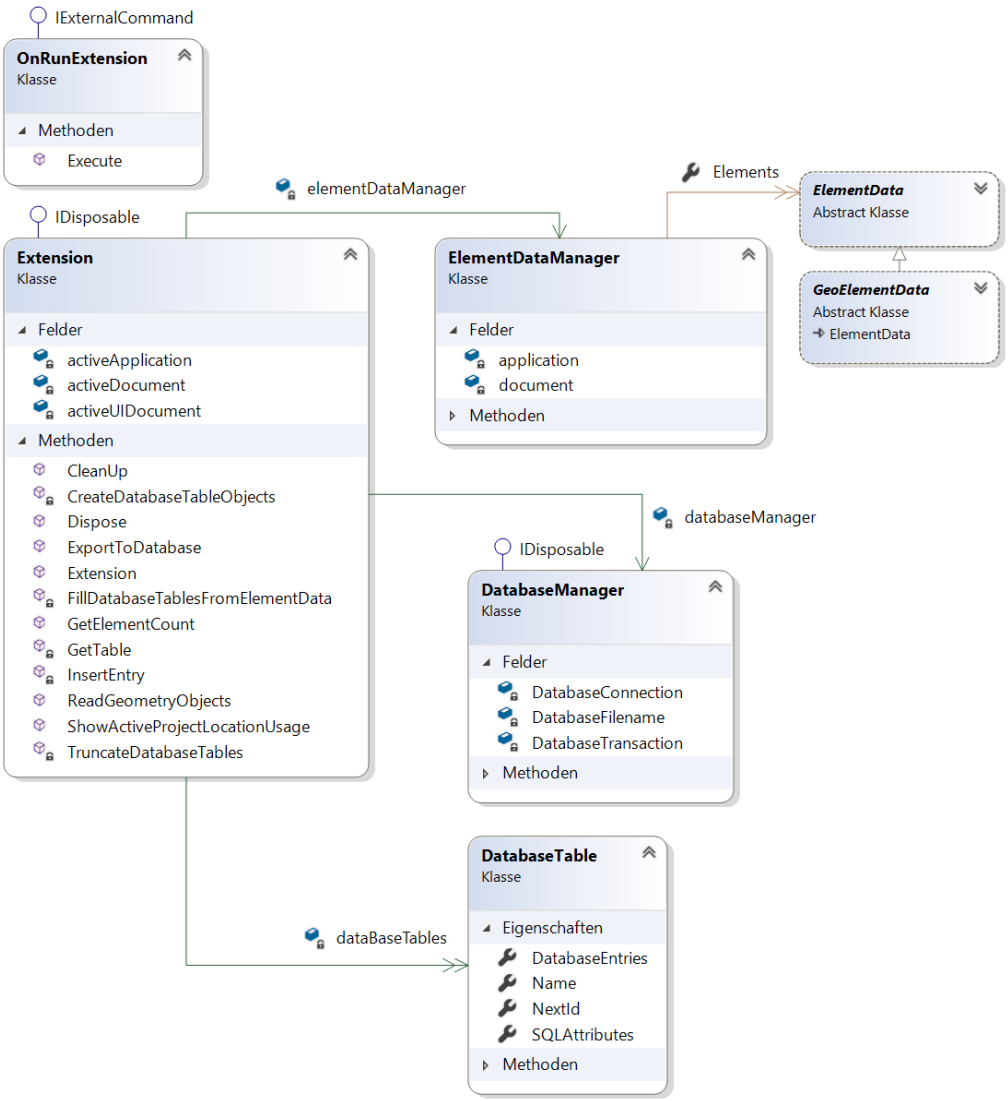


Abbildung 4.2: Objektmodell der Revit-Extension

Die Klasse `Extension` repräsentiert die Revit-Extension selbst. Sie verwaltet nebn der Verbindung zur Hostanwendung *Revit*, das Auslesen der Modelldaten (über eine Instanz der Klasse `ElementDataManager`), als auch den Datenbankexport über den `DatabaseManager` und einer Liste von Datensätzen des Typs `DatabaseTable`.

4.2 Einbindung der Extension in das Revit-Framework

Die Extension wird als Externes Kommando in die *Revit* Anwendung eingebunden. Das Tool *Addin-Manager* [1] ermöglicht das Laden von externen Klassenbibliotheken die als Extension in *Revit* gestartet werden können. In der DLL muss eine Klasse implementiert sein die das `Autodesk.Revit.UI.IExternalCommand`-Interface erweitert und die Methode `Execute` zur Verfügung stellt. In unserem Fall ist dies die Klasse `OnRunExtension`.
Die Implementierung der Methode `Execute` ist im Folgenden Codeausschnitt zu entnehmen:

```
32 public Result Execute(ExternalCommandData commandData,
33                       ref string message,
34                       ElementSet elements)
35 {
36   Extension ext = null;
37   try
38   {
39     Application app = commandData.Application.Application;
40     UIDocument uidoc = commandData.Application.ActiveUIDocument;
41     ext = new Extension(app, uidoc);
42     // first we try to read the geometry-objects,
43     if (0 != ext.ReadGeometryObjects())
44     {
45       // if this succeeds we export the geometry to the database
46       if (0 == extension.ExportToDatabase())
47       {
48         // if we cant export to the DB, return failure
49         return Result.Failed;
50       }
51     }
52     else
53     {
54       // otherwise return failure
55       return Result.Failed;
56     }
57   }
58   catch (Exception e)
59   {
60     // log the exception and stacktrace
61     Logging.AddToLog(e.Message, Logging.LOGTYPE.Exception);
62     Logging.AddToLog("TRACE: "+e.StackTrace, Logging.LOGTYPE.Exception);
63     ext.CleanUp();// Run all "cleanup" instructions
64     return Result.Failed;
65   }
66   return Result.Succeeded;
67 }
```

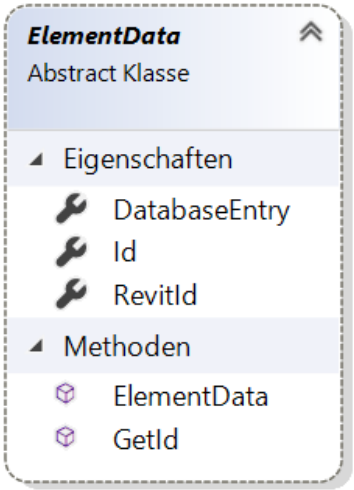
Codeausschnitt 4.1: Einstiegspunkt der Extension *(ExternalCommand.cs)*

[1] 2010 Autodesk, Inc.

Die Methode `OnRunExtension.Execute` ist also der Haupteinstiegspunkt für das Tool. Wird die Klassenbibliothek geladen und das Kommando `Execute` aufgerufen, so wird ein Objekt des Typs `Extension` instantiiert (s. Zeile 41). Diesem Objekt wird zum einen der Pointer auf die Revit-Applikation und zum anderen ein Pointer auf das aktive Dokument übergeben. Danach wird die Methode `Extension.ReadGeometryObjects` aufgerufen (s. Zeile 43). Der Rückgabewert der Methode entscheidet über den weiteren Verlauf der Funktion. Wird 0 zurückgegeben, so ist ein Fehler in der Methode `ReadGeometryObjects` aufgetreten und ein Export in die Datenbank kann nicht erfolgen, somit gibt die Methode `Execute` den Fehlercode *Result.Failed* zurück. Entspricht der Rückgabewert der Lese-Funktion nicht 0, so sind keine Fehler aufgetreten und der Datenbankexport über die Methode `Extension.ExportToDatabase` (s. Zeile 46) wird ausgeführt.
Die Extension wird innerhalb eines try-catch-Blocks ausgeführt. Besonders bei Schnittstellen zwischen mehreren Anwendungen - auf die der Entwickler mit unter keinen Einfluss nehmen kann - ist ein Fehlverhalten zur Laufzeit nicht auszuschließen. Wird eine *Exception* ausgelöst, so wird der zugehörigen Fehlercode, der Stacktrace (Stapel- bzw. Speicherzurückverfolgung) und der Typ der Ausnahme in einer eigenen Log-Datei gespeichert (s. Zeile 61 und 62). Damit bekommt der Entwickler Feedback über Abstürze und eventuelle zugehörige Programmfehler. Zudem werden über die Methode `Extension.CleanUp` alle bis dato nicht freigegeben Ressourcen (v.a. Arbeitsspeicher und Datenbankverbindungen) freigegeben.

4.3 Strukturierung der Modelldaten

Wie unter Abschnitt 3.3 beschrieben, enthält jedes Revit *Dokument* eine Reihe von *Element* Typen die in Summe das Bauwerksmodell abbilden. Diese sind so abstrakt strukturiert, dass über sie sämtliche Bauwerksrelevanten Daten abgebildet werden können.
Bei der Implementierung des Moduls das diese Daten ausliest und weiter verwendet, wurde daher auch der Begriff des „Element“ verwendet.
Ähnlich wie der Ansatz von Autodesk bildet die Klasse `ElementData` nur die Relevanten Grundinformationen eines Elements im Kontext der Extension ab. Die weitere Spezialisierung der Daten erfolgt in von `ElementData` abgeleiteten Klassen.
Dies wird durch das Schlüsselwort *abstract* hervorgehoben, denn abstrakte Klassen selbst können nicht instantiiert werden, wodurch der Entwickler gezwungen ist, die entsprechenden Objekte für eine konkretere Klassifizierung der Daten zu implementieren.

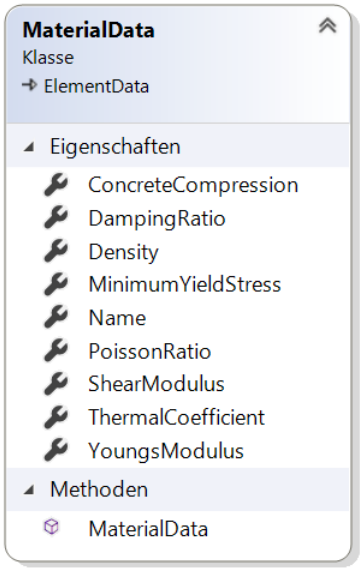


Der Pointer auf den zugehörigen Datenbankeintrag des Typs `DatabaseEntry` enthält einen Tabellennamen der zugehörigen Datenbank und eine Liste mit Einträgen vom Typ `SQLiteParameter` über die der SQLite-Query aufgebaut wird (siehe 4.4).
Der Primärschlüssel des Eintrags der Datenbank wird Programmintern verwaltet und entspricht dem 1-basierten Index des `ElementData`-Objekts in der Liste `ElementDataManager.Elements`.
Die Id des Elements wird innerhalb der `Autodesk.Revit.DB` Datenbank über die API vergeben.

Diese Eigenschaften sind die Basis für alle Elemente die in der Datenbank abgebildet werden. Das heißt, dass für die Datenhaltung zur Laufzeit jede Datenobjekt-Instanz auf ein Objekt des Typs `ElementData` reduziert werden kann.
Dadurch ist es möglich **alle Modellinformationen** in einer einzigen eindimensionalen Liste in der Klasse `ElementDataManager` zu halten, ohne dabei zwischen den expliziten Objektinstanzen (Solids, Faces, Edges, etc.) unterscheiden zu müssen. Letztlich befindet sich in der Liste der Elementdaten nur eine Adresse im Arbeitsspeicher die auf ein Objekt verweist, dass über seinen zugehörigen `DatabaseEntry` in die Datenbank geschrieben werden kann. Weil C# das Objektorientierte Entwickeln und damit den sog. Polymorphismus unterstützt, kann also der Verwaltungsaufwand auf ein Minimum reduziert werden.

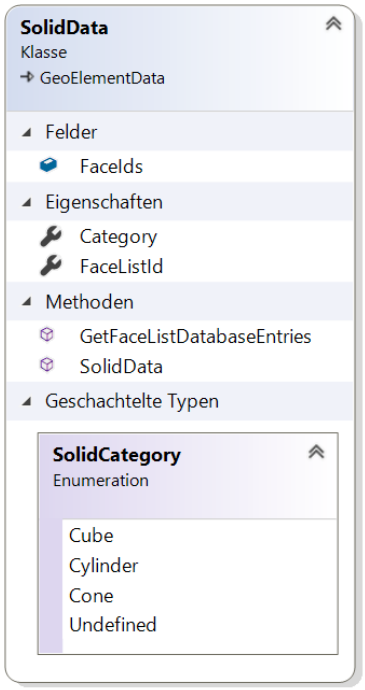
Es ist sichergestellt, dass bei der Erweiterung des Datenmodells der Element-Daten **keine** Anpassungen in der Datenhaltung (`ElementDataManager.Elements`) oder der Datenbankschnittstelle (`DatabaseManager`) erforderlich ist. Einzig die `SQLiteParameter` einer neuen - von `ElementData` abgeleiteten - Klasse sind entsprechend zu implementieren und an den `DatabaseEntry` anzuhängen.

4.3.1 Datenhaltung

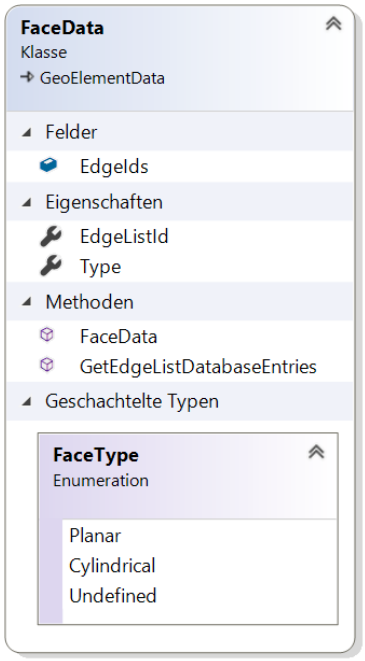


Die Klasse `MaterialData` ist von `ElementData` abgeleitet.
Ein Material wird durch seine physikalischen Eigenschaften und einen `Name` beschrieben.

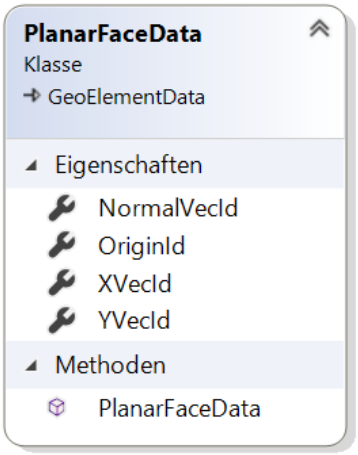
Die abstrakte Klasse `GeoElementData` dient als Basisklasse für alle Element-Daten die ein geometrisches Objekt repräsentieren. Sie erweitert `ElementData` um die Id des zugehörigen Materials und einer Eigenschaft `Tag` das für zusätzliche Kommentare und Beschreibungen der Geometrie dienen kann.



Die `SolidData` Klasse beschreibt einen beliebigen Volumenkörper.
Dieser ist durch eine Liste von Seiten (`FaceIds`) beschrieben und kann einer `Category` zugeordnet werden. Der Index der `FaceIds`-Liste in der Datenbank wird in der Eigenschaft `FaceListId` gehalten.
Die Kategorie dient zum einen dazu Fehler bei der Datenübertragung festzustellen. So kann bspw. ein Volumenkörper der Kategorie *Cube* nur aus Ebenen Flächen bestehen und die Anzahl der Seiten muss genau 6 sein.
Zum anderen kann die Kategorie auch als Filtereigenschaft verwendet werden, wenn große Datenmengen anfallen und nur Volumenkörper gesucht werden die eine bestimmte Form haben.



Die einzelnen *Faces* eines Volumenkörpers können in allgemeiner Form in der Klasse `FaceData` beschrieben werden. Eine Seite lässt sich über eine Liste von Kanten (`EdgeIds`) definieren.
Die Form der Seite wird in der Eigenschaft `Type` gehalten, hier sind Ebene (*Planar*) und gekrümmte (*Cylindrical*) Flächen implementiert.
Die Methode `GetEdgeListDatabaseEntries` liefert eine Liste von Datenbankeinträgen die die entsprechende Seite als Liste von Kanten mit einer identischen `EdgeListId` beschreiben (siehe Abschnitt 4.4).



Die geometrische Definition einer ebenen Fläche im Raum erfolgt über die Klasse `PlanarFaceData`.
Eine Ebene ist durch ihren Normalvektor (`NormalVecId`), Ursprung (`OriginId`) und Vektor in X und Y-Richtung (`XVecId`, `YVecId`) definiert(siehe Abbildung 4.3).

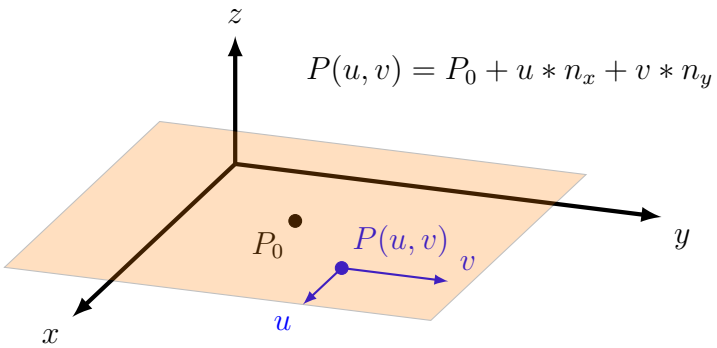
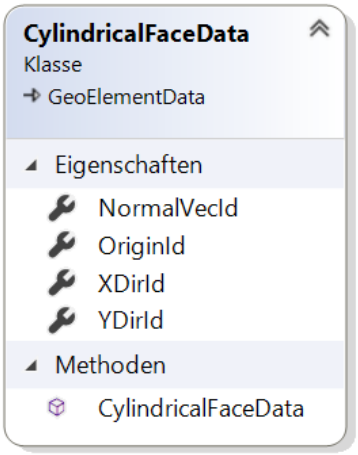


Abbildung 4.3: Ebenengleichung und -parameter einer Ebene (*PlanarFace*)



Eine gekrümmte, zylindrische Fläche im Raum wird durch die Klasse `CylindricalFaceData` beschrieben.
Sie ist ähnlich wie die Ebene durch ihren Normalvektor (`NormalVecId`), Ursprung (`OriginId`) und Richtungsvektor in X und Y-Richtung (`XDirId`, `YDirId`) definiert (siehe Abbildung 4.4).

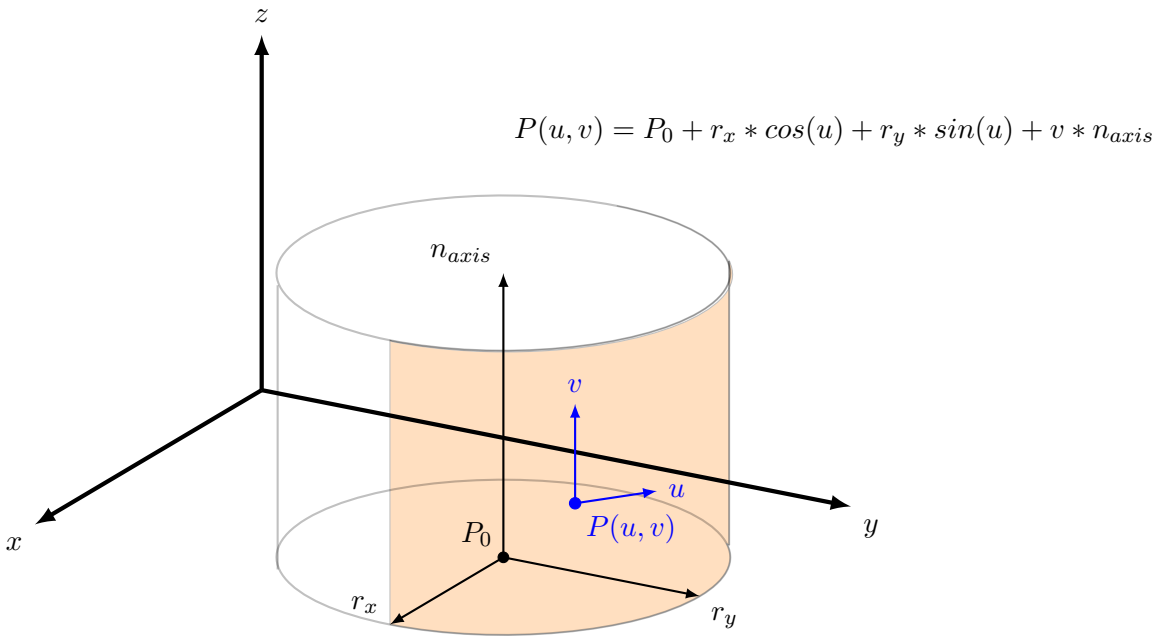
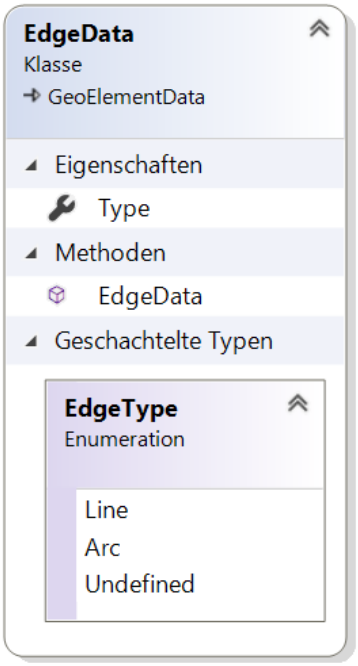
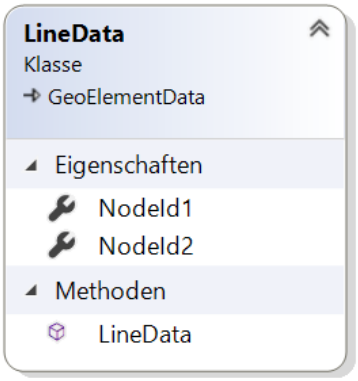


Abbildung 4.4: Ebenengleichung und -parameter einer einachsig gekrümmten Ebene (*CylindricalFace*)

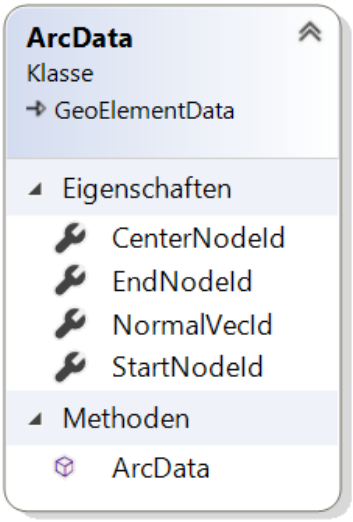
Die Radien r_x und r_y ergeben sich aus dem Produkt der Richtungsvektoren `XDir` bzw. `YDir` und dem Radius des beschriebenen Kreises (siehe `ArcData`).



Die Kante einer Fläche wird in `EdgeData` beschrieben. Den eindeutigen Kenner erbt die Klasse aus `GeoElementData` (und diese aus `ElementData`). Eine Kante kann auf verschiedene Arten beschrieben sein, konkret implementiert sind derzeit Linien und Bögen. Alle weiteren Formen sind *Undefinded*.



Eine Linie ist durch zwei Punkte im Raum eindeutig definiert.
In der Datenbank werden für jede Linie die Id's der beiden Punkte (`NodeId1` und `NodeId2`) gespeichert.



Ein Bogen verläuft von Anfangs- zum Endpunkt mit konstantem Abstand zum Mittelpunkt. Die Drehrichtung des Bogens ergibt sich aus der Richtung des Normalvektors `NormalVecId` der Ebene in der der Bogen liegt.
Der Normalvektors wird in der Parameterschreibweise in n_x und n_y aufgeteilt (siehe Abbildung 4.5).
Die Id's der Punkte (*Nodes*) werden in den Eigenschaften `StartNodeId`, `EndNodeId` und `CenterNodeId` gehalten.

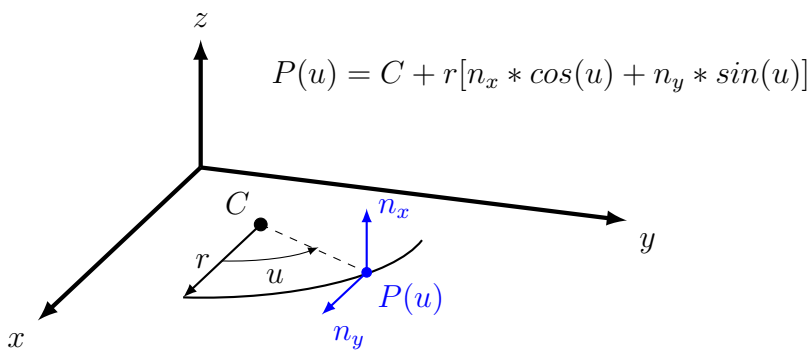
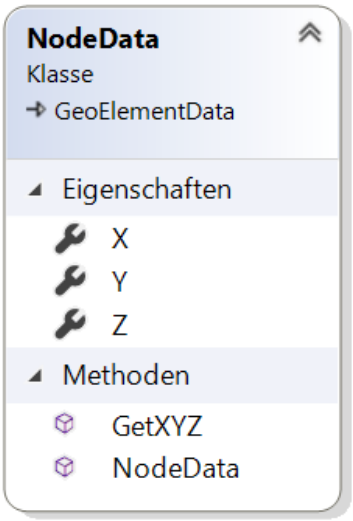


Abbildung 4.5: Parameter eines Bogen (*Arc*)



Die Informationen aller vorangegangenen Klassen münden in der Datenbank letztlich als Knoten/Punkt mit X-, Y- und Z-Koordinate.
Ein Objekt des Typs `NodeData` enthält die entsprechenden Koordinaten von allen in den übergeordneten Klassen beschriebenen geometrischen Daten. Alle Ursprungspunkte, Vektoren und Knoten werden über `NodeData` Objekte abgebildet.
Die Einheit der Koordinaten ist in Meter.

4.3.2 Beschaffung und Verwaltung der Modelldaten

Wie aus den zuvor erläuterten Strukturen hervorgeht, weisen die Klassen für die Datenhaltung keine umfangreichen Verwaltungsmethoden auf. Programm-intern wird zwischen Datenhaltung in allen von `ElementData` abgeleiteten Klassen und Datenverwaltung über die Klasse `ElementDataManager` unterschieden. Diese Trennung hat hier einige Vorteile:

- Klare Aufgabenverteilung der Programmmodule
  → Reduktion der Fehlerquellen
- Erleichterung beim Prototyping
  → Funktionalität muss nicht parallel zur Datenhaltung implementiert werden
- Speicherbedarf wird reduziert
  → Objektinstanz für die Datenhaltung enthält keine Verwaltungsstrukturen

Abbildung 4.6: Zusammenhang Datenhaltung - Datenverwaltung

Aus Abbildung 4.6 geht der angesprochenen Zusammenhang zwischen der Datenhaltung und der Datenverwaltung/Datenbeschaffung hervor.

Die Methode `ReadElementsFromDocument` in der Klasse `ElementDataManager` dient als Haupteinstiegspunkt für die Lese-Routine der Extension. Grundsätzlich sind zwei Modelldatentypen für den Export relevant:

1. Analytische Modelldaten (Statisches System)
2. Struktur-Elementdaten (Volumenmodell)

Diese beiden Datentypen werden innerhalb der Klasse `ElementDataManager` über eigene Methoden aus dem aktiven Revit Dokument gelesen.

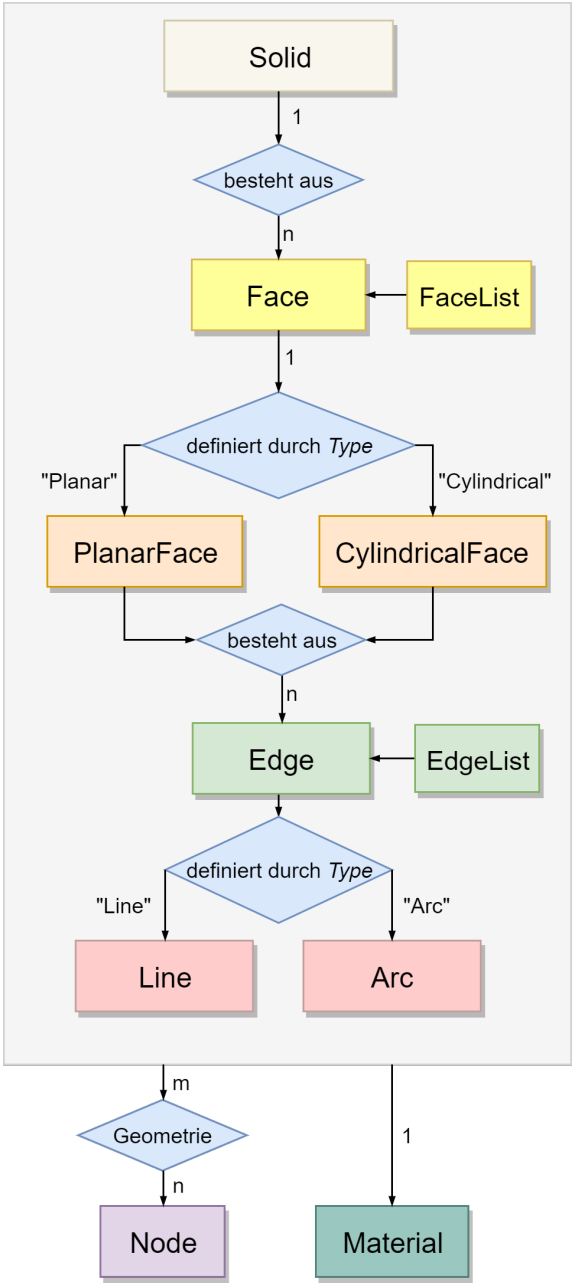
```
54 public int ReadElementsFromDocument()
55 {
56   // For now we only care for elements that have an analytical model
57   RevitDB.FilteredElementCollector analyticalModels =
58     new FilteredElementCollector(this.document)
59          .OfClass(typeof(AnalyticalModel));
60
61   // loop through all analytical models
62   foreach (AnalyticalModel analyticalModel in analyticalModels)
63   {
64     // Get Element Id of the structural element
65     // corresponding to the Analytical Model
66     RevitDB.ElementId elemId = analyticalModel.GetElementId();
67
68     // Get the structural element from the analytical Model via 'elemId'
69     // and read the model data of the structural element
70     RevitDB.Element structuralElem = this.document.GetElement(elemId);
71
72     // To keep the analytical Model connected to its FamilyInstance
73     // we read the volume model alongside with the analytical model
74     if (0 == ReadStructuralElementData(structuralElem))
75     {
76       Logging.AddToLog("Error in 'ReadStructuralModelData'",
77                        Logging.LOGTYPE.Error);
78       return 0;
79     }
80
81     // read the model data of the analytical model
82     if (0 == ReadAnalyticalModelData(analyticalModel))
83     {
84       Logging.AddToLog("Error in 'ReadAnalyticalModelData'",
85                        Logging.LOGTYPE.Error);
86       return 0;
87     }
88   }
89   return 1;
90 }
```

Codeausschnitt 4.2: Auslesen der Element-Daten *(ElementDataManager.cs)*

In den Zeilen 57 bis 59 werden über den sog. `FilteredElementCollector` alle die Elemente die dem „Bauplan“ eines `AnalyticalModel`-Typen entsprechen ausgelesen.
Wie in Abschnitt 3.3 beschrieben wurde, ist die objektorientierte Datenbank von Revit so aufgebaut, dass alle Modellelemente sowohl auf Analytische Modelle als auch auf Strukturelle Modelle verweisen.
Diese Referenzierung erleichtert nun die Suche nach relevanten Modelldaten, denn alle gefundenen Analytischen Modelle verweisen ebenso auf entsprechende Strukturelle Modelldaten. Hier ist es also nicht nötig nach beiden Typen zu Filtern, es genügt einen der beiden zu finden und über die Objektstruktur der Revit-API die fehlende Referenz herzustellen (Zeile 70).

4.4 Datenbankstruktur

In der SQLite Datenbank wird die verschachtelte Datenstruktur der Objektorientierten *Autodesk.Revit.DB*-Datenbank über Relationen mit unterschiedlichen Beziehungen abgebildet. Die in den einzelnen Datensätzen gehaltenen Daten wurden in Abschnitt 4.3.1 bereits erläutert.



Die Objekthierarchie die sich aus dem Revit-Datenmodell ergibt wird über eine *1:n*-Beziehungen abgebildet.

Generische Typen wie *Face* und *Edge* werden über das Attribut *Type* dem entsprechenden Datensatz zugeordnet, wo sie näher beschrieben werden.

Das Volumenmodell wird über die gezeigten Relationen letztlich über Punkte bzw. Vektoren beschrieben die in der Tabelle *Node* gehalten werden. Hier steht die Tabelle *Node*, also in einer *m:n*-Beziehung zu den Datensätzen die das Modell sozusagen „qualitativ“ beschreiben.

Einem Volumenkörper kann eindeutig ein Material zugeordnet werden, das heißt hier liegt eine *1:1*-Beziehung vor.

Abbildung 4.7: Relationsmodell der SQLite Datenbank

Die Datenbankeinträge der in Abb. 4.7 beschriebenen Relationen werden Programm-intern über die Klasse `DatabaseEntry` abgebildet. Die Einträge werden als Liste in `DatabaseTable`-Objekten gehalten, welche die Relationen selbst abbilden.

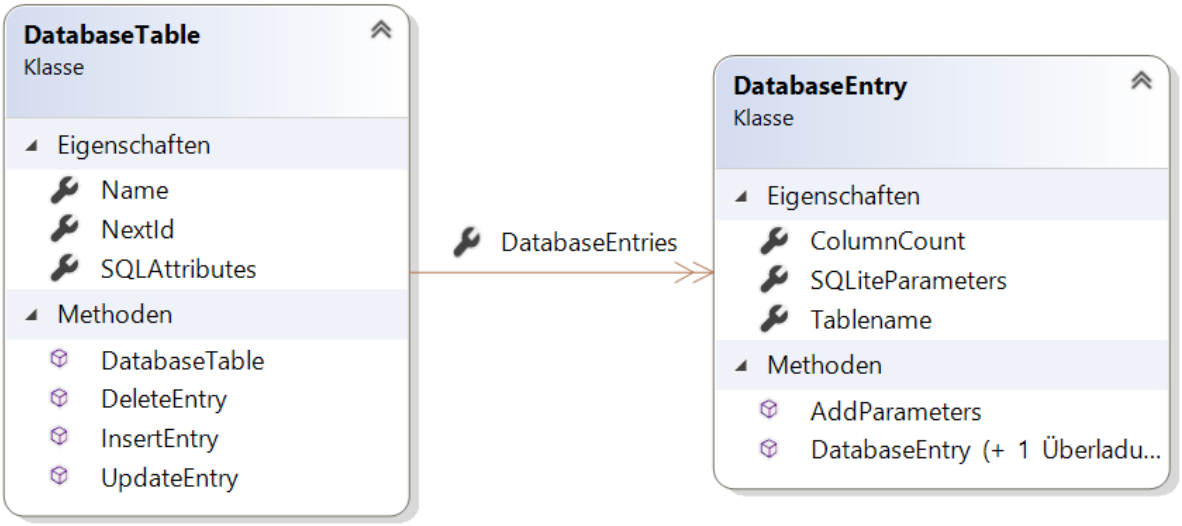


Abbildung 4.8: Zusammenhang *DatabaseTable - DatabaseEntry*

Ein Eintrag in der Datenbank (Entry) hat eine Datensatz von `SQLiteParameters` und den Namen der zugehörigen Tabelle `Tablename`. Wie in Abschnitt 4.3.1 beschrieben wurde, ist der Datenbankeintrag eine Eigenschaft der ausgelesenen Modellelemente `ElementData`. Jedes Modelldatenobjekt beschreibt den Datenbankeintrag mit den jeweils notwendigen Informationen. Dafür stellt der Datenbankeintrag die Methode `AddParameters` zur Verfügung. Die Klasse `DatabaseTable` fasst eine Reihen von Datenbankeinträgen (`DatabaseEntries`) zusammen und stellt `SQLAttributes` für die Datenbanktabelle selbst zur Verfügung. Zusätzlich sind Schnittstellen-Methoden für das Einfügen, Löschen und Ändern von einzelnen Einträgen implementiert.

Die Klasse `Extension` enthält eine Liste von `DatabaseTable`-Objekten und implementiert die Methode `FillDatabaseTablesFromElementData` die aus der Liste von `ElementData`-Objekten die Datenbankeinträge ausliest und das entsprechende Datenbank-Tabellen-Objekt „befüllt“.

```
248 private int FillDatabaseTablesFromElementData()
249 {
250   // each element has its own databaseentry (see: ElementData)
251   // the entry of the element is inserted in the database
252   foreach (var elemData in this.elementDataManager.Elements)
253   {
254     InsertEntryInCorrespondingTable(elemData.DatabaseEntry);
255
256     // both FaceData and SolidData have an additional set of entries
257     // they need to be added on their own.
258     // Additional implementations with similar
259     // structure are to be added below
260     if (elemData is FaceData faceData)
261     {
262       // FaceData contains a List of Entries for the Table "EdgeList"
263       foreach (DatabaseEntry e in faceData.GetEdgeListDatabaseEntries())
264       {
265         InsertEntryInCorrespondingTable(e);
266       }
267     }
268     else if (elemData is SolidData solidData)
269     {
270       // SolidData contains a List of Entries for the Table "FaceList"
271       foreach (DatabaseEntry e in solidData.GetFaceListDatabaseEntries())
272       {
273         InsertEntryInCorrespondingTable(e);
274       }
275     }
276   }
277   return 1;
278 }
```

Codeausschnitt 4.3: Befüllen der DatabaseTable Objekte *(Extension.cs)*

4.5 Schnittstelle zur SQLite Datenbank

Die Verbindung zur SQLite-Datenbank wird über die Klasse `DatabaseManager` verwaltet. Wie in Abschnitt 4.1 beschrieben, wird der Datenbankmanager innerhalb der Klasse `Extension` instantiiert. Dem Modul wird der Pfad zur Datenbank übergeben womit die Verbindung hergestellt werden kann.

```
19 public int InitDatabase(string databaseFilename)
20 {
21   // keep filename of database as property
22   this.DatabaseFilename = databaseFilename;
23   if (Util.IsFileLocked(databaseFilename))
24   {
25     string msg = "The SQLite-File is locked.\n";
26     msg += "Change the Filename or close the locking application ";
27     msg += "and try again.";
28     TaskDialog.Show("IOError", msg);
29     return 0;
30   }
31
32   // create the database-file
33   SQLiteConnection.CreateFile(this.DatabaseFilename);
34
35   // connect to the database and open the connection
36   string connection = "Data Source="+this.DatabaseFilename+";Version=3;";
37   this.DatabaseConnection = new SQLiteConnection(connection);
38   this.DatabaseConnection.Open();
39
40   // due to the m:n relation of certain tables FOREIGN-Keys are needed
41   SQLiteCommand command = new SQLiteCommand(this.DatabaseConnection);
42   BeginTransaction();
43   command.CommandText = "PRAGMA foreign_keys = ON;";
44   command.ExecuteNonQuery();
45   CommitTransaction();
46   CloseConnection();
47
48   return 1;
49 }
```

Codeausschnitt 4.4: Init-Methode des Datenbankmanagers *(DatabaseManager.cs)*

Der dargestellte Codeausschnitt zeigt die Initiierungsmethode des Datenbankmanagers. Soll eine neue Datenbankdatei angelegt werden, so muss zunächst geprüft werden, ob die Datei im übergebenen Pfad verwendet werden kann (s. Zeile 23ff). Andernfalls muss die Extension frühzeitig beendet werden, denn der Zugriff auf eine gültige Datenbankdatei ist unabdingbar für den Weiteren Programmverlauf.
Der Verbindungsaufbau erfolgt über einen *connectionString* der den Pfad zur Datenbank und die Version der Datenbank enthält. Weil einige Datenbanktabellen eine *m:n-Beziehung* zueinander haben, wird über das Schlüsselwort `foreign_keys` die Kaskadierte Beziehung der entsprechenden Tabellen ermöglicht.

Eine Instanz der Klasse Datenbankmanager hält selbst keine Daten, sondern fungiert nur als Schnittstelle zwischen der Datenhaltung in `ElementDataManager.Elements` und der SQLite-Datenbank. In Abb. 4.9 sind alle Member der Klasse dargestellt. Es wird ersichtlich dass bis auf *Dateiname*, *Verbindung* und aktive *Transaktion* keine Felder/Container implementiert sind. Dies verdeutlicht den Interface-Charakter der Klasse.

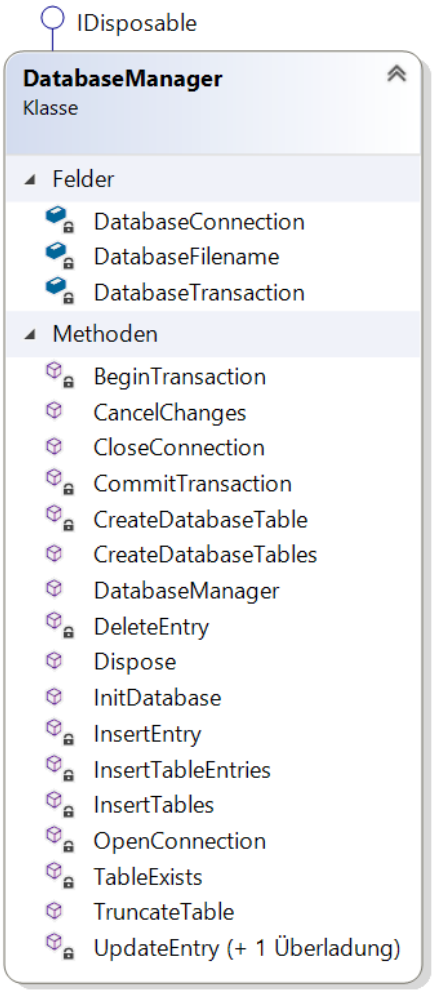


Abbildung 4.9: Klasse *DatabaseManager*

Die Datenbank wird in der Methode `CreateDatabaseTables` mit den Daten aus `DatabaseTable`-Objekten befüllt.

```
103 public int CreateDatabaseTables(List<DatabaseTable> databaseTables)
104 {
105   int returnValue = 1;
106   try
107   {
108     OpenConnection();
109     BeginTransaction();
110
111     foreach (DatabaseTable table in databaseTables)
112     {
113       // first, create the table with its columns
114       CreateDatabaseTable(table);
115
116       // second, add all entries from the DatabaseTable-Object
117       foreach (DatabaseEntry entry in table.DatabaseEntries)
118       {
119         InsertEntry(table.Name, entry);
120       }
121     }
122     CommitTransaction();
123     CloseConnection();
124   }
125   catch (Exception e)
126   {
127     Logging.AddToLog(e.Message, Logging.LOGTYPE.Error);
128     string msg="Exception in 'CreateDatabaseTables()':\n" + e.Message;
129     TaskDialog.Show("Exception", msg);
130     CloseConnection();
131     returnValue = 0;
132   }
133   return returnValue;
134 }
```

Codeausschnitt 4.5: Erzeuger-Methode des Datenbankmanagers *(DatabaseManager.cs)*

`CreateDatabaseTables` wird an einer einzigen Stelle in der Funktion `ExportToDatabase` ausgeführt. Das heisst, dass alle SQLite-Queries gebündelt und nacheinander an die Datenbank geschickt werden. Dies wirkt sich sehr positiv auf die Performance aus, was besonders bei großen Datenmengen einen Vorteil darstellt.
Darüber hinaus erübrigt sich durch den einmaligen, zentralen Aufruf der „Schreib-Routine“ auch eine mit unter aufwendige Verwaltung der Datenbankzugriffe. So wäre zwar durch ein asynchrones Beschreiben der Datenbank die Performance nochmal zu steigern, allerdings müssen dann entsprechende Verwaltungsstrukturen in der Datenbankschnittstelle geschaffen werden, damit zur Laufzeit keine Daten wegen unzulässigem Zugriff u.ä. verloren gehen. Diese Strukturen lassen sich aber auch nachträglich implementieren und können im Rahmen einer Weiterentwicklung der Schnittstelle umgesetzt werden (mehr dazu unter Abschnitt 5).

4.6 Anwendungsbeispiel

Die Funktionalität des Tools soll anhand einer einfachen Pfahlkopfplatte beispielhaft gezeigt werden.

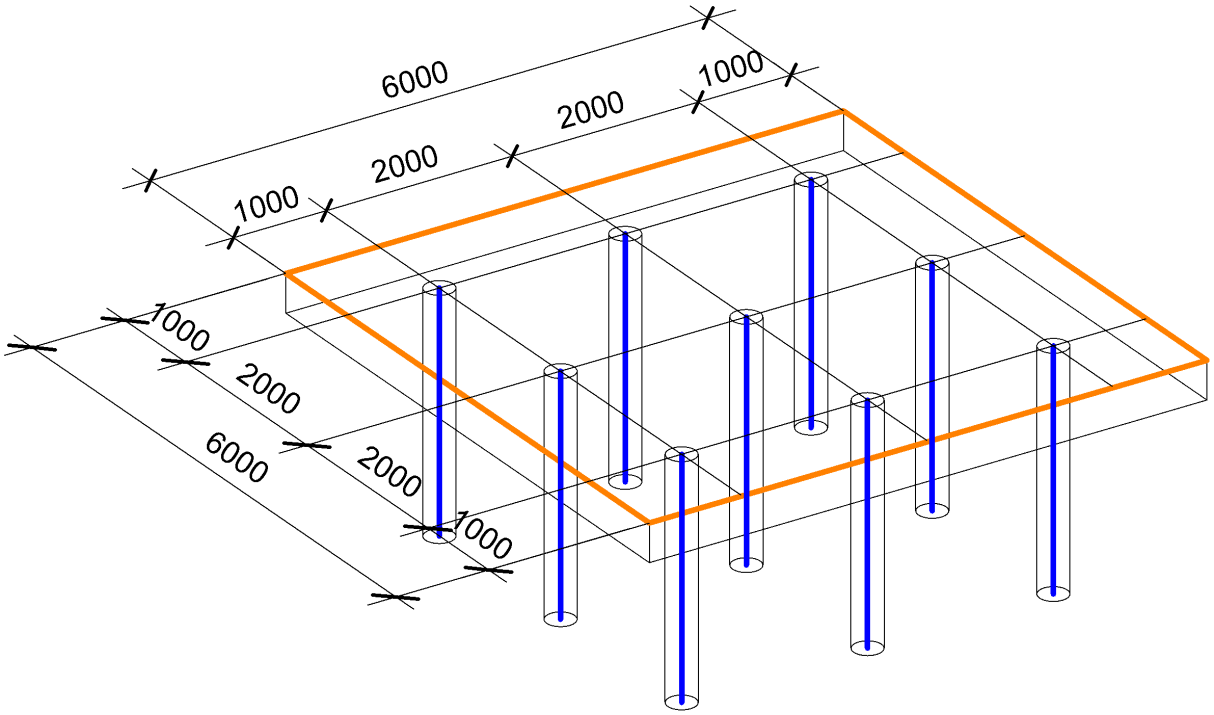


Abbildung 4.10: Beispielmodell „Pfahlkopfplatte“

Bezeichnung	Wert	Einheit
Plattenbreite	6	m
Plattenlänge	6	m
Plattendicke	40	cm
Anzahl der Pfähle	9	-
Pfahldurchmesser	300	mm
Pfahllänge	2.5	m

Tabelle 4.1: Geometrie der Pfahlkopfplatte

Bezeichnung	E-Modul [MPa]	Druckfestigkeit [MPa]
Ortbetonplatte	23250	24.13
Fertigteilpfahl	27791	34.48

Tabelle 4.2: Material der Pfahlkopfplatte

Das Volumenmodell der Fundamentplatte wird über eine Familien-Instanz des Typs *Structural Foundation: Slab* abgebildet. Diese - standardmäßig in Revit verfügbare - Familie enthält sowohl ein Struktur- bzw. Volumenmodell, als auch ein statisches System in Form einer Scheibe.
Die Pfähle werden über die Familie *Structural Column: Concrete Round Column* repräsentiert. Auf weiterführende Berechnungen hat es keinen Einfluss, dass statt der Familie des Typs *Pile* eine Stütze gewählt wurde. Viel mehr erleichtert die Stützen-Familie das Modellieren, denn im Gegensatz zum *Pile* stellt die Stütze neben dem Volumenmodell - wie die Platte - ein statisches System *Analytical Model* zur Verfügung. In Abbildung 4.10 sind die statischen Systeme farblich gekennzeichnet. Das Scheibenelement der Fundamentplatte ist orange umfahren, der Stab der einen Pfahl abbildet ist blau hervorgehoben.

Wie eingangs beschrieben wird die Extension derzeit über den sog. *Add-In Manager* eingebunden. Dieser ist in *Revit* unter dem Reiter „Add-Ins - External Tools“ zu finden:

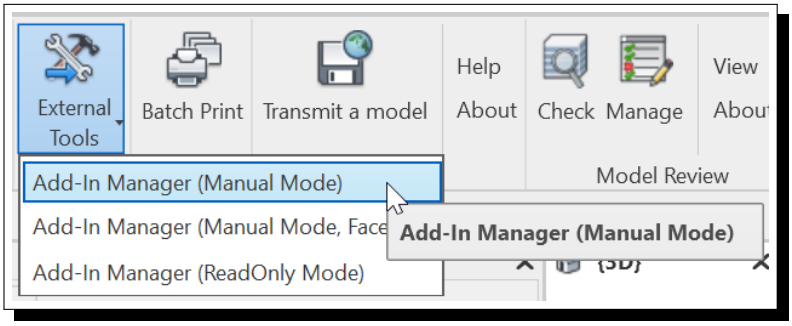


Abbildung 4.11: Add-In Managern in Revit 2019

Im sog „Manual Mode“ kann der Benutzer eine Typenbibliothek für die Verwendung in Revit laden und dann aus den in der *.dll enthaltenen Funktionen die von `IExternalCommand` abgeleitet sind auswählen.

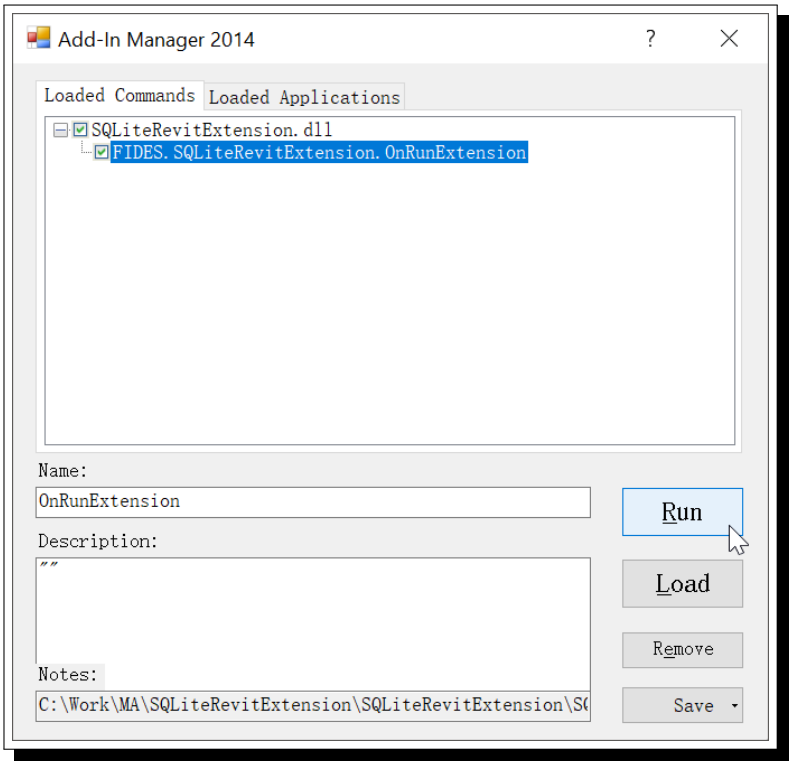


Abbildung 4.12: Auswahl und Aufruf des `ExternalCommands` im Add-In Manager

In unserem Fall ist die Klasse `OnRunExtension` die einzige in der *dll* die zur Verfügung steht. Mit den Klick auf „Run“ wird die unter Abschnitt 4.2 beschriebene Einstiegfunktion aufgerufen.
War die Extension erfolgreich, so bekommt der Benutzer zum Schluss Feedback über die Anzahl der Exportierten Elemente als auch die verstrichene Zeit.

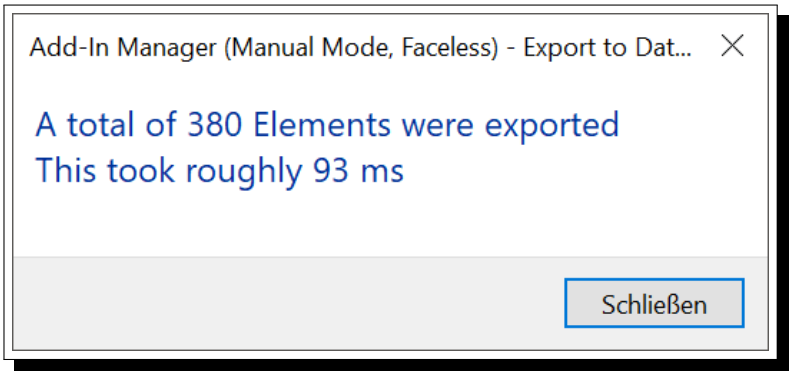


Abbildung 4.13: Schlussmeldung der Extension

Ein Blick in die Datenbank über das Tool *SQLite DB-Browser* zeigt dass alle Materialdaten entsprechend der Vorgaben übertragen wurden.

Tabelle: Material

	Id	Revit_Id	Name	PoissonRatio	YoungsModulus	ShearModulus	ThermalCoefficient	Density	DampingRatio	MinimumYieldStress	ConcreteCompression
	...	Fil...	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern
1	2	20489	Concrete, Cast-in-Place...	0.167	23250.0	9964.0	1.0e-05	2407.31172	0.15	-1.0	24.13293801
2	58	20490	Concrete, Precast	0.167	27791.0	11908.0	1.0e-05	2407.31172	0.15	-1.0	34.47562573

Abbildung 4.14: Tabelle *Material* der SQLite-Datenbank

Wie unter Abschnitt 4.4 beschrieben, ist die SQLite-Datenbank so aufgebaut, dass die Struktur- und Volumenmodelle über Hierarchische Relationen abgebildet sind. In diesem Beispiel wird dies exemplarisch an der Bodenplatte gezeigt.

Tabelle: Solid

	Id	Revit_Id	Material_Id	Tag	FaceList_Id
	...	Filtern	Filtern	Filtern	Filtern
1	53	932025687	2	Floor	53
2	57	944646215	2	analyticalSolid	57
3	59	933588679	58	Column	59

Abbildung 4.15: Tabelle *Solid* der SQLite-Datenbank

In der Tabelle *Solid* ist die Bodenplatte über die Id=53 zu finden (siehe *Tag* „Floor“). Die zugehörige Liste der Seiten ist in Tabelle *FaceList (Id=53)* abgelegt.

Tabelle: FaceList

	Id	Face_Id
	Fil...	Filt...
1	53	18
2	53	34
3	53	41
4	53	45
5	53	49
6	53	51

Tabelle: Face

	Id	Revit_Id	Material_Id	Tag	Type	EdgeList_Id
	...	Filtern	Filtern	Filtern	Filtern	Filtern
1	18	1637107398	2	400mm Foundation Sl...	Planar	18
2	34	1637107558	2	400mm Foundation Sl...	Planar	34
3	41	1637107590	2	400mm Foundation Sl...	Planar	41
4	45	1637106726	2	400mm Foundation Sl...	Planar	45
5	49	1637106758	2	400mm Foundation Sl...	Planar	49
6	51	1637106918	2	400mm Foundation Sl...	Planar	51

Abbildung 4.16: Tabelle *FaceList* und *Face* der SQLite-Datenbank

In der Tabelle *FaceList* finden sich genau 6 Einträge mit der Id 53. Dies passt zu dem Volumenmodell der Platte, welche auch genau 6 Seiten hat. Jeder Eintrag verweist wiederum auf eine *Face_Id* in der Tabelle *Face*. Hier werden die Seiten noch näher klassifiziert.
Wie bereits in Abschnitt 4.3.1 erläutert wurde, können die Seiten eines Volumenkörpers unterschiedlich geformt sein. Der *Type* der Seite zeigt hier den Typ *Planar* an, das heißt es handelt sich um eine ebene Fläche die in der Tabelle *PlanarFace* näher beschrieben wird. Außerdem findet sich ein Verweis auf die zugehörige Liste von Kanten (Edges) über den Kenner *EdgeList_Id*.

Tabelle: PlanarFace

	Id	Revit_Id	Material_Id	Tag	Normal_Vector_Id	Origin_Node_Id	X_Vector_Id	Y_Vector_Id
	...	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern	Filtern
1	18	1637107398	2	400mm Foundation Sl...	15	3	16	17
2	34	1637107558	2	400mm Foundation Sl...	32	20	33	17
3	41	1637107590	2	400mm Foundation Sl...	40	3	32	16
4	45	1637106726	2	400mm Foundation Sl...	16	4	32	17
5	49	1637106758	2	400mm Foundation Sl...	17	7	32	33
6	51	1637106918	2	400mm Foundation Sl...	33	10	32	40

Abbildung 4.17: Tabelle *PlanarFace* der SQLite-Datenbank

Die Tabelle beschreibt die Ebene *PlanarFace* über ihre Vektoren und den Ursprung.

Tabelle: EdgeList

	Id	Edge_Id
	Fil...	Filt...
1	18	5
2	18	8
3	18	11
4	18	13

Tabelle: Edge

	Id	Revit_Id	Material_Id	Tag	Type
	Fil...	Filtern	Filtern	Filtern	...
1	5	1617046822	2	400mm Foundation Sl...	Line
2	8	1617047014	2	400mm Foundation Sl...	Line
3	11	1617046182	2	400mm Foundation Sl...	Line
4	13	1617046022	2	400mm Foundation Sl...	Line

Abbildung 4.18: Tabelle *EdgeList* und *Edge* der SQLite-Datenbank

Eine Seite der Fundamentplatte wird über 4 *Edges* beschrieben. Die jeweilige *Edge_Id* referenziert einen Eintrag mit gleicher Id in der Tabelle *Edge*. Hier wird - wie auch bei der Tabelle *Face* - der *Type* des Geometrischen Objekts abgelegt. In unserem Fall handelt es sich um *Line* Objekte die die Kante der Platte abbilden. Dies zeigt an, dass eine nähere Definition in der Tabelle *Line* zu finden ist.

Tabelle: Line

	Id	Revit_Id	Material_Id	Tag	Node_Id1	Node_Id2
	...	Filtern	Filtern	Filtern	Filtern	Filtern
1	5	1617046822	2	400mm Foundation Sl...	3	4
2	8	1617047014	2	400mm Foundation Sl...	4	7
3	11	1617046182	2	400mm Foundation Sl...	7	10
4	13	1617046022	2	400mm Foundation Sl...	10	3

Abbildung 4.19: Tabelle *Line* der SQLite-Datenbank

Die *Line* verweist auf jeweils zwei *Node_Id*'s in der Tabelle *Node*.
Hier findet dann die Referenzierung ein Ende und die Geometrie wird über „Handfeste“ Koordinaten aufgelöst.

Tabelle: Node

	Id	Revit_Id	Material_Id	Tag	X	Y	Z
	F...	Filtern	Filtern	Filtern	Filtern	Filtern	F...
1	1	63318255	-1	StartupLocation	-3.41498490662583	9.2747274581249	0.0
2	3	49282189	2	400mm Found...	7.50545714232089	-0.827213330782175	-0.4
3	4	28110368	2	400mm Found...	7.50545714232089	-6.82721333078218	-0.4
4	7	14954540	2	400mm Found...	1.50545714232089	-6.82721333078218	-0.4
5	10	14979642	2	400mm Found...	1.50545714232089	-0.827213330782175	-0.4

Abbildung 4.20: Tabelle *Node* der SQLite-Datenbank

In Abbildung 4.20 sind die 4 Eckpunkte der Fundamentplatte gezeigt.
Allerdings liegt der Ursprung des Koordinatensystems an einem scheinbar zufälligem Ort und nicht - wie gewohnt - an einer Ecke oder im Schwerpunkt der Platte. Die gezeigten Koordinaten beziehen sich auf den Ursprung der Ebene über die die Seite der Platte beschrieben wird. Die Id des Ursprungsknoten n_o des lokalen Koordinatensytems der Platte ist in Abb. 4.17 gezeigt (*Origin_Node_Id* des *PlanarFace* mit Id=18) und ergibt sich zu:

$$n_o = \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix}$$

In unserem Fall ergeben sich die Eckpunkte im lokalen Koordinatensystem der Platte zu:

$$n_o - n_{Id3} = \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} - \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} = \begin{pmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \end{pmatrix}$$

$$n_o - n_{Id4} = \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} - \begin{pmatrix} 7.5055 \\ -6.8272 \\ -0.4000 \end{pmatrix} = \begin{pmatrix} 0.0000 \\ 6.0000 \\ 0.0000 \end{pmatrix}$$

$$n_o - n_{Id7} = \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} - \begin{pmatrix} 1.5055 \\ -6.8272 \\ -0.4000 \end{pmatrix} = \begin{pmatrix} 6.0000 \\ 6.0000 \\ 0.0000 \end{pmatrix}$$

$$n_o - n_{Id10} = \begin{pmatrix} 7.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} - \begin{pmatrix} 1.5055 \\ -0.8272 \\ -0.4000 \end{pmatrix} = \begin{pmatrix} 6.0000 \\ 0.0000 \\ 0.0000 \end{pmatrix}$$

Die Datenbank liefert also alle Informationen um ein Volumenmodell im Raum (in globalen und lokalen Koordinaten) zu beschreiben. Für die Pfähle sind die Daten nach gezeigtem Prinzip in der Datenbank gehalten. Mit Bezugnahme auf Abschnitt 4.4 kann also das Volumenmodell aus der Datenbank gelesen werden.

5 Ausblick

Wie im Anwendungsbeispiel in Abschnitt 4.6 gezeigt wurde, konnte die geforderte Funktionalität umgesetzt werden. Über die entwickelte Extension wird der Export eines Struktur- und Volumenmodells in eine SQLite-Datenbank ermöglicht.
Das beschriebene Objektmodell der Extension ist mit Hinblick auf die Erweiterung der Funktionalität entwickelt und stellt nach Ansicht des Verfassers eine gute Basis für die Weiterentwicklung des Tools dar.
Abschließend eine Reihe von Features, die - mit dem entwickelten Tool als Basis - umgesetzt werden können.

Interaktion durch den Benutzer

Im jetzigen Entwicklungsstand der Extension, liegt eine dynamische Programmbibliothek vor, die über ein externes Tool in das *Revit*-Framework eingebunden wird. Der Benutzer kann sich nur für den Zeitpunkt des Exports entscheiden und hat keinen Einfluss auf Inhalt, Pfad und Namen der Datenbank. Im Rahmen der Weiterentwicklung könnte eine grafische Benutzeroberfläche entwickelt werden, über die Einfluss auf genannte Parameter genommen werden kann.
Der Inhalt der Datenbank ließe sich über die Vorauswahl der zu exportierenden Objekte beeinflussen. Pfad und Name der Datenbank bspw. über einen Windows-üblichen „Speichern Unter...“-Dialog. Hierüber ließe sich auch das Problem umgehen, dass versehentlich alte Datenbanken unter gleichem Namen überschrieben werden.

Intensive „Stress-Tests“ der Datenbank

Wie in Abschnitt 2.1 erläutert sollen später auch sehr große Modelle und DGM-Daten über die Datenbank abgebildet werden. In der Theorie ist dies über Relationale Datenbanken wie SQLite durchaus möglich und gerade deswegen wurde eine Datenbank anstelle von XML u.ä. gewählt. Allerdings wurde das Tool und die implementiere Datenbank-Schnittstelle in einem ersten Schritt für kleinere übersichtlichere Modelle entwickelt.
Prinzipiell hat die Größe des Modells bzw. die Anzahl der Einträge in der Datenbank keinen Einfluss auf einen erfolgreichen Export. Dennoch ist zu prüfen ob und wie sich die Performance des Tools ändert, wenn sehr große Datenmengen anfallen.
Es ist zu erwarten, dass aufgrund der „single-threaded“ Struktur der Extension die Performance leidet wenn die Anzahl der Datenbank-Queries steigt. Auch der Bedarf an Arbeitsspeicher wird sich erhöhen, da - aus Performance-gründen - zunächst alle Tabelleneinträge im Speicher gehalten werden, um dann zentral in die Datenbank geschrieben zu werden.

Bidirektionaler Datenaustausch

Die Weiterverarbeitung der aus *Revit* exportierten Daten ist letztlich der Kern dieser Arbeit. Die Möglichkeiten die sich aus der einheitlichen Datenbasis ergeben sind vielzählig. Die weiterführenden Berechnungen statischer, dynamischer und kaufmännischer Art erzeugen wieder Ergebnisse die unter Umständen zu Veränderungen in der Datenbasis führen.
Ein klassisches Beispiel hierzu wäre - mit Bezugnahme auf das in Abschnitt 4.6 beschriebene Beispiel - eine Erhöhung der Plattendicke aus statischen Gründen. In einem solchen Fall, müsste derzeit das Modell in *Revit* wieder manuell geändert werden, ein weiterer Export ist nötig um die Datenbank zu aktualisieren, um am Ende bspw. die Kalkulation der Kosten erneut durchführen zu können.
Denkbar wäre aber auch die Schnittstelle so zu erweitern, dass nicht nur Modelldaten aus *Revit* gelesen werden, sondern auch Daten aus anderen Quellen in *Revit* importiert werden können.

Abbildungsverzeichnis

3.1 Beispiel 1:1 Beziehung 12
3.2 Beispiel 1:n Beziehung 13
3.3 Beispiel m:n Beziehung 13
3.4 Hierarchie der Klasse *Element* 15
3.5 Hierarchie der Klasse *GeometryObject* 16

4.1 Ablaufschema der Extension 17
4.2 Objektmodell der Revit-Extension 18
4.3 Ebenengleichung und -parameter einer Ebene (*PlanarFace*) 24
4.4 Ebenengleichung und -parameter einer einachsig gekrümmten Ebene (*CylindricalFace*) 25
4.5 Parameter eines Bogen (*Arc*) 27
4.6 Zusammenhang Datenhaltung - Datenverwaltung 28
4.7 Relationsmodell der SQLite Datenbank 31
4.8 Zusammenhang *DatabaseTable* - *DatabaseEntry* 32
4.9 Klasse *DatabaseManager* 35
4.10 Beispielmodell „Pfahlkopfplatte“ 37
4.11 Add-In Managern in Revit 2019 38
4.12 Auswahl und Aufruf des `ExternalCommands` im Add-In Manager 39
4.13 Schlussmeldung der Extension 39
4.14 Tabelle *Material* der SQLite-Datenbank 40
4.15 Tabelle *Solid* der SQLite-Datenbank 40
4.16 Tabelle *FaceList* und *Face* der SQLite-Datenbank 41
4.17 Tabelle *PlanarFace* der SQLite-Datenbank 41
4.18 Tabelle *EdgeList* und *Edge* der SQLite-Datenbank 42
4.19 Tabelle *Line* der SQLite-Datenbank 42
4.20 Tabelle *Node* der SQLite-Datenbank 43

Tabellenverzeichnis

3.1 Beispiel einer Tabelle 11

4.1 Geometrie der Pfahlkopfplatte 37
4.2 Material der Pfahlkopfplatte 37

Liste der Codeausschnitte

4.1 Einstiegspunkt der Extension *(ExternalCommand.cs)* 19
4.2 Auslesen der Element-Daten *(ElementDataManager.cs)* 29
4.3 Befüllen der DatabaseTable Objekte *(Extension.cs)* 33
4.4 Init-Methode des Datenbankmanagers *(DatabaseManager.cs)* 34
4.5 Erzeuger-Methode des Datenbankmanagers *(DatabaseManager.cs)* 36

Literaturverzeichnis

[1] D. Richard Hipp. Sqlite documentation. https://www.sqlite.org/about.html, January 2019.

[2] Elmar Fuchs. *SQL - Grundlagen und Datenbankdesign*. Herdt, Bodenheim, 2014.

[3] Udo Kelter. *Skript zum Lehrmodul 'Objektorientierte Datenbanksysteme'*, April 2004.

[4] Autodesk. Revit api developers guide, 2018. Zugriff am 03.01.2019.

Verwendete Programme

Bei der Erstellung dieser Arbeit wurden Folgende Programme verwendet:

Visual Studio Community 2017
Version 15.7.1

DB-Browser for SQLite
Version 3.10.1

Notepad++
Version 7.5.9

draw.io
Version 10.1.6-7

Greenshot
Version 1.2.10

Texmaker
Version 5.0.3

TeXworks
Version 0.6.2