



WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

ENTWICKLUNG EINES EDITORS FÜR GENX-DATEIEN

Fakultät für Physikalische Technik und Informatik
Fachgruppe Informatik

Entwicklung eines erweiterbaren Struktureditors für Genx-Dateien

Bachelorarbeit, vorgelegt von

Long Nguyen Hoang

im Februar 2026

betreut von der TKI mbh Chemnitz

Inhaltsverzeichnis

1. Einleitung	7
1.1. Motivation	7
1.2. Arbeitsziele	7
1.3. Abgrenzung	7
1.4. Vorgehensweise	7
1.5. Aufbau der Arbeit	7
2. Theoretische und technische Grundlagen	8
2.1. Autodesk AutoCAD	8
2.2. Map 3D Toolset	8
2.3. Feature Data Objects	8
2.4. Benutzung von Map 3D in der TKI mbH	8
2.5. Datenmodelle in Autodesk AutoCAD Map 3D	8
2.5.1. Themen	8
2.5.2. Objektklassen	9
2.5.3. Ansichten	9
2.5.4. Domänen	9
2.5.5. Tabellen	9
2.5.6. Versorgungsnetzwerk-Modelle	9
2.5.7. Logische Topologien	9
2.5.8. Sequenzen	9
2.6. Die GENX-Datei	10
2.7. Benutzung von Windows Presentation Foundation (kurz WPF) und C#	10
2.8. Model-View-ViewModel Entwurfsmuster	10
2.8.1. Model	10
2.8.2. View	10
2.8.3. ViewModel	11
2.8.4. Zusammenspiel von Model-View-ViewModel	11
2.9. Beispiel eines C# WPF Programms mit MVVM	11
2.9.1. Beispiel Model	11
2.9.2. Beispiel View	11
2.9.3. Beispiel ViewModel	11
2.9.4. Beispiel Command	12
3. Anforderungsanalyse	14
3.1. Ziele des Struktureditors	14
3.2. Nutzer des Struktureditors	14
3.3. Systemkontext	14
3.4. Funktionale Anforderungen	14
3.5. Nicht-funktionale Anforderungen	15
3.6. Einschränkungen	15

4. Konzept	16
4.1. Planung der Benutzeroberfläche	16
4.1.1. Menüleiste	17
4.1.2. Versionsanzeige	17
4.1.3. Darstellung des Datenmodells	17
4.1.4. Eigenschaftenbereich	17
4.2. Planung der Benutzerinteraktion	17
4.2.1. Interaktion für die Dateiverarbeitung	17
4.2.2. Interaktion mit dem Datenmodell	17
4.3. Umgang mit dem bestehenden Model	21
4.3.1. GenXParser	21
4.3.2. GenXStructure-Klasse und Datenmodell-Objekte	21
4.3.3. Probleme des bestehenden Models	21
4.3.4. Lösungskonzepte der Probleme	23
5. Umsetzung	24
5.1. Benutzeroberfläche	24
5.1.1. Container Struktur	24
5.1.2. UI-Elemente	26
5.1.3. Verbinden der View mit dem ViewModel	30
5.2. Umgang mit den Problemen des bestehenden Models aus Kapitel 4.3.3 . .	31
5.2.1. Lösung für die fehlende Themen Klasse	31
5.2.2. Lösung für verschachtelte Objekte	32
5.2.3. Lösung für unzureichende Zugriffsmodifikatoren	32
5.3. ViewModel	33
5.3.1. Haupt-ViewModel	33
5.3.2. Angleichen des ViewModels an das Model	33
5.3.3. Zusätzliches ViewModel für die Darstellung der Sammelknoten ei- nes Datenmodells	33
5.3.4. Erstellen der GenXElement-ViewModels	34
5.3.5. Anzeigen der Objekte in der Datenmodellldarstellung	37
5.3.6. Erstellung des Kontextmenüs eines Knotens	38
5.3.7. Commands	39
5.3.8. Auswahl eines Knotens	41
5.3.9. Anzeige der Eigenschaften eines Knotens	43
5.3.10. Datenmodellobjekte erstellen	44
6. Ergebnis	47
6.1. Überblick der Funktionen des Struktureditors	47
6.2. Dateiverarbeitung	48
6.2.1. Öffnen einer Genx-Datei (FA-1)	48
6.2.2. Speichern einer Genx-Datei (FA-2)	48
6.2.3. Erstellen eines neuen Datenmodells (FA-3)	48
6.3. Datenmodell-Bearbeitung	49
6.3.1. Erstellen neuer Datenmodellobjekte (FA-4)	49
6.3.2. Löschen bestehender Objekte (FA-5)	50
6.3.3. Bearbeiten von Objekteigenschaften (FA-6)	50
6.3.4. Erstellen eines neuen Strukturupdate (FA-7)	50
6.4. Darstellung	51
6.4.1. Datenmodell als hierarchische Struktur (FA-8)	51

6.4.2. Objekteigenschaften anzeigen (FA-9)	51
6.4.3. Anzeigen einer Änderung (FA-10)	52
6.5. Zusammenfassung des entwickelten Struktureditors	52
7. Fazit	53
7.1. Was wurde erreicht?	53
7.2. Herausforderungen	53
7.2.1. Unzureichende Analyse	53
7.2.2. Komplexes Model	53
A. Model-View-ViewModel-Beispiel	54

Abbildungsverzeichnis

2.1. Screenshot des Beispiel Programms	13
2.2. Screenshot des Beispielprogramms nach dem Klick auf „Alter erhöhen“ . .	13
4.1. Benutzeroberfläche des AutoCAD Struktureditors mit einem geladenen Datenmodell	16
4.2. Menüpunkt „Datei“ mit Neu, Öffnen, Speichern und Speichern unter	19
4.3. Kontextmenü eines Thema-Objektes in der Datenmodelldarstellung	19
4.4. Attributauswahl mit vorgegebenen Werten	20
4.5. Pop-Up-Fenster der Versionseingabe bei einem neuen Strukturupdate . . .	20
4.6. vereinfachtes Klassendiagramm von GenXStructure und Datenmodellobjekten	21
4.7. Links: Beispiel-Datenmodell mit verschachtelten Objekten, Rechts: Ergebnis der GenXParsers der Wohnung Feature-Klasse	22
5.1. Skizze des Grids und der StackPanel	24
5.2. Beispiel-Datenmodell mit drei verschachtelten Themen	31
6.1. Struktureditor nach dem Öffnen einer Genx-Datei	48
6.2. Kontextmenü des Themen-Sammelknotens	49
6.3. Kontextmenü eines Thema-Knotens	49
6.4. Struktureditor nach dem erstellen eines neuen Objekts	50
6.5. Darstellung eines Datenmodells in hierarchischer Struktur	51
6.6. Die Eigenschaften eines ausgewählten Knotens (Person Feature-Klasse) werden im Eigenschaftenbereich angezeigt	52

Tabellenverzeichnis

6.1. Übersicht der Anforderungen und deren Umsetzungsstatus	47
---	----

1. Einleitung

1.1. Motivation

Der neue Struktureditor soll TKI-intern benutzt werden und ein weiterer Schritt in Richtung AutoDesk-Unabhängigkeit sein. Ein eigener Struktureditor würde es erlauben, diesen nach den Anforderung des Unternehmens anzupassen. Es könnte beispielsweise eine Funktion eingefügt werden, die häufig wiederkehrende Strukturen mit einem Klick erstellt. Die Erstellung einer solchen Struktur würde eine schnellere und strukturell fehlerfreie Möglichkeit bieten ein Datenmodell zu erweitern.

1.2. Arbeitsziele

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung eines Struktureditor-Prototyps, der das Laden, Visualisieren, Bearbeiten und Speichern von Genx-Dateien ermöglicht.

1.3. Abgrenzung

Das im Rahmen dieser Bachelorarbeit entstehende Programm soll ein Prototyp eines Struktureditors sein. Dieser soll nicht vollumfänglich den bestehenden AutoCAD Struktureditor ersetzen.

1.4. Vorgehensweise

Für die Entwicklung des neuen Struktureditor wird zuerst der bestehende Struktureditor analysiert und anhanddessen Anforderungen an den neuen Struktureditor erhoben. Mit diesen Anforderungen wird ein Konzept und anhand des Konzeptes wird der Prototyp implementiert.

1.5. Aufbau der Arbeit

In Kapitel 2 werden die theoretischen Grundlagen für diese Arbeit erklärt. In Kapitel 3 werden die Anforderungen an den neuen Struktureditor gesetzt. In Kapitel 4 wird anhand der Anforderungen ein Konzept erstellt, wie die Anforderungen umgesetzt werden können. In Kapitel 5 wird auf die Implementierung des Struktureditors eingegangen. Es wird erklärt wie das Konzept und die Anforderungen umgesetzt wurden. In Kapitel 6 wird auf den zum Schluss entwickelten Struktureditor eingegangen, es wird thematisiert, welche Anforderungen umgesetzt wurden und wie diese Anforderungen im Programm aussehen.

2. Theoretische und technische Grundlagen

2.1. Autodesk AutoCAD

AutoCAD ist eine, von der Firma Autodesk, entwickelte „Computer-aided Design“-Software, oder kurz CAD-Software. AutoCAD erlaubt das Entwerfen und Bearbeiten von „2D-Geometrien“ und „3D-Modelle“. [3]

2.2. Map 3D Toolset

Das Map 3D-Toolset ist eine Funktionserweiterung für die AutoCAD-Software. Map 3D erlaubt den Zugriff und die Bearbeitung von Geodaten. Diese Geodaten werden mithilfe der „Feature Data Objects“-Schnittstelle bezogen.

2.3. Feature Data Objects

Feature Data Objects, oder kurz FDO, ist eine Schnittstelle mit der Geodaten aus unterschiedlich konfigurierten Datenquellen bezogen werden kann. Es ist eine Abstraktionsschicht die einen einheitlichen Zugriff auf unterschiedliche Datenquellen erlaubt. [2]

2.4. Benutzung von Map 3D in der TKI mbH

TKI ist ein Unternehmen, welches sich mit der Planung, dem Ausbau und der Dokumentation von Glasfasernetzen auseinandersetzt. Mithilfe von Map 3D werden die geplanten Glasfasernetze digital dokumentiert, um somit einen digitalen Zwilling zu erschaffen.

2.5. Datenmodelle in Autodesk AutoCAD Map 3D

In Autodesk AutoCAD Map 3D werden die darstellbaren Objekte in einem Datenmodell definiert. Diese Definition bestimmt beispielsweise, wie ein Objekt aufgebaut ist oder welche Unterobjekte es besitzt. Im folgenden werden die für die TKI mbH relevanten Datenmodellobjekte näher erklärt. Ausgelassen werden „Allgemeine Klassen“, „Allgemeine Tabellen“, „Flächen-Topologien“, „Verschnitte“ und „Lineare Referenzsysteme“.

2.5.1. Themen

Themen sind in einem Datenmodell Ordnerstrukturen, mit denen Objektklassen, Ansichten und weitere untergeordnete Themen von anderen Themen getrennt werden können. Es dient der Strukturierung und Abgrenzung von Themenbereichen. Beispielsweise kann es für die Trennung von „Gebäude“-Objekten und „Person“-Objekten die zwei Themen „Lageplan“ und „Verwaltung“ geben.

2.5.2. Objektklassen

Objektklassen, im Verlauf der Arbeit als Feature-Klasse bezeichnet, sind Abstraktionen von realen Objekten. Feature-Klassen besitzen eine Geometrie, eine Form, mit der das Objekt abstrakt dargestellt werden kann. Beispielsweise hat eine „Straßen“-Klasse die Geometrie einer Linie. Feature-Klassen bestehen auch aus verschiedenen Attributen, die das reale Objekt beschreiben. Bei einer „Straßen“-Klasse wäre das beispielsweise der Name der Straße. [1]

2.5.3. Ansichten

Ansichten, oder auch Views, sind definierte Datenabfrage auf eine bestimmte Datensammlung. Dies erlaubt einen vereinfachten, gefilterten Blick auf die Daten. Eine View kann beispielsweise alle Datensätze anzeigen, die ein bestimmtes Kriterium erfüllen. Zum Beispiel könnte eine View für die „Straßen“-Klasse, alle Straße herausfiltern die eine Länge von über 100 Metern haben.

2.5.4. Domänen

Domänen sind Mengen mit festgelegten gültigen Werten. Zum Beispiel könnte es eine Domäne „Straßenbeläge“ geben, welche die verschiedenen Arten von Straßebelägen angibt, wie beispielsweise „Asphalt“, „Pflasterstein“ oder „Schotter“. Domänen werden benutzt, um den Attributen einer Feature-Klasse nur mit gültigen Werten zu definieren.

2.5.5. Tabellen

Tabellen sind ein strukturiertes Sammelobjekt, welches Spalten (Attribute) und Zeilen (Datensätze) besitzt. Tabellen sind ähnlich wie Feature-Klassen Abstraktionen, nur besitzen Tabellen-Daten keine Geometrie-Daten.

2.5.6. Versorgungsnetzwerk-Modelle

Versorgungsnetzwerk-Modelle sind definierte Systeme aus Knoten oder Kanten-Objekten, die ein Verhalten darstellen. Damit kann definiert werden, wie sich Objekte zu einander verhalten.

2.5.7. Logische Topologien

Logische Topologien stellen die Verbindung zwischen Objekten dar, unabhängig von der Art und Weise, wie diese geometrisch gezeichnet werden. Beispielsweise können sich zwei Straßen auf einer geometrischen Darstellung kreuzen, ohne in Realität mit einander verbunden zu sein. Dies geschieht beispielsweise Brücken oder Tunneln, die über oder unter einer anderen Straße verlaufen.

2.5.8. Sequenzen

Sequenzen sind definierte Regeln, mit denen eine fortlaufenden Nummer erzeugt wird.

2.6. Die GENX-Datei

Ein Datenmodell wird in einer Genx-Datei abgespeichert. Dies ist ein auf XML-basierendes Datenformat, das die Datenmodell-Struktur beschreibt. In dem Datenmodell werden die in Abschnitt 2.5 erklärten Objekte gespeichert. Änderungen an einem Datenmodell werden als neue XML-Tags mit der „Update“-Operation gekennzeichnet. Diese Änderungen werden in einem `<UpdateVersion>`-Tag zusammengefasst und mit einer Versionsnummer gekennzeichnet. So wird in einer Genx-Datei nicht nur die endgültige Datenmodell-Struktur gespeichert, sondern mit einer Versionierung auch die Änderungen einer bestimmten Version.

2.7. Benutzung von Windows Presentation Foundation (kurz WPF) und C#

Für den neuen Struktureditor wird die Programmiersprache C# in Kombination mit WPF genutzt. Die TKI mbh entwickelt standardmäßig mit den Programmiersprachen VB.net oder C# in Kombination mit WPF. Da dieser Struktureditor im Unternehmen weiterentwickelt werden soll, können die schon erlangten Kenntnisse weiter genutzt werden. Anwendungen in WPF werden standardmäßig mit dem Model-View-Viewmodel-Entwurfsmuster oder kurz MVVM entwickelt.

2.8. Model-View-ViewModel Entwurfsmuster

Model-View-ViewModel, oder kurz MVVM, ist eine Variation des Model-View-Controller Entwurfsmusters. Es findet Nutzen in Anwendungen die Benutzeroberfläche mit Nutzdaten koppeln und dient der Trennung und Strukturierung einer Anwendung. Eine Anwendung, die MVVM nutzt, wird in drei Aufgabenbereiche geteilt. Dem Model, das für die Daten und die Geschäftslogik verantwortlich ist, der View, die für die Darstellung der Benutzeroberfläche verantwortlich ist, und dem ViewModel, welches die Daten aus dem Model in präsentabler Form an die View weiterleitet. [4]

2.8.1. Model

Das Model bildet die fachliche Grundlage einer Anwendung und ist unabhängig von der Benutzeroberfläche. Es besteht aus den grundlegenden Daten und der Geschäftslogik, auf der die Anwendung aufgebaut ist. Das Model ist zuständig für die Abstraktion realer Objekte und Entitäten, das können zum Beispiel Produkte, Bestellungen oder Kunden sein, die Umsetzung der Geschäftslogik, zum Beispiel, dass eine Bestellung einen Kunden und ein Produkt enthält, und die Verarbeitung der Daten.

2.8.2. View

Die View definiert die Benutzeroberfläche eines Programms und ist für die Darstellung der Daten verantwortlich. In WPF wird die Benutzeroberfläche deklarativ mit „Extensible Application Markup Language“, oder kurz „XAML“, beschrieben. Das heißt, dass schon vor dem Ausführen des Programms die grundsätzliche Benutzeroberfläche feststeht. Um die View interaktiv zu gestalten, wird der Inhalt der View mittels Data Binding mit dem ViewModel verknüpft.

2.8.3. ViewModel

Das ViewModel ist die Vermittlungsschicht zwischen der View und dem Model. Es stellt der View die Daten für die Benutzeroberfläche und die ausführbaren Befehle, wie Commands, zur Verfügung. Die Daten und die Commands werden über öffentliche Properties bereitgestellt und werden mit Data Binding an die View geknüpft, wo die Daten in den UI-Elementen angezeigt werden. Wenn sich die Daten im ViewModel ändern, wird der View das über Events mitgeteilt, sodass sich diese automatisch aktualisieren kann.

Die Daten des ViewModel stammen aus dem Model, auf diese kann das ViewModel zugreifen und es kann diese ändern. Diese Daten werden vom ViewModel in eine Darstellungs-geeignete Form verarbeitet und bereitgestellt.

2.8.4. Zusammenspiel von Model-View-ViewModel

Das Model-View-ViewModel Entwurfsmuster ist ein praktisches Hilfsmittel zum Erreichen einer losen Kopplung. Durch die Trennung einer Anwendung in die Teile Model, View und ViewModel, können Änderungen an den einzelnen Bereichen vorgenommen werden, ohne Änderungen an den restlichen Teilen vornehmen zu müssen. Es erlaubt auch, eine einfachere Zusammenarbeit zwischen Entwicklern und UX-Designern. Diese können die Benutzeroberfläche erstellen, ohne viel Wissen über die unterliegende Programmierung zu haben. Genauso können die Entwickler im Model Änderungen vornehmen, ohne die Benutzeroberfläche zu beeinflussen.

2.9. Beispiel eines C# WPF Programms mit MVVM

Zur Veranschaulichung des Model-View-ViewModel Entwurfsmusters wird ein einfaches Beispiel betrachtet. Der Quelltext des Beispiels befindet sich im

2.9.1. Beispiel Model

Das Model besteht aus einer Klasse Person (Siehe Listing A.1), die einen Namen und ein Alter hat und diese mit öffentlichen Properties weitergeben kann. Die Klasse Person hat eine öffentliche Methode „IncreaseAge()“, die das Alter um 1 erhöht.

2.9.2. Beispiel View

Die View (Siehe Listing A.2) definiert ein Fenster in dem sich ein StackPanel befindet. Ein StackPanel ist ein Objekt, welches seinen Inhalt entweder horizontal nacheinander oder vertikal untereinander anordnet. In diesem Beispiel werden ein TextBlock und ein Knopf untereinander angeordnet. Ein Screenshot der laufenden Anwendung ist in Abbildung 2.1 auf Seite 13

In WPF wird der View noch eine Datei angehängt, die Quelltext enthält, das sogenannte CodeBehind. Diese Datei hat direkten Zugriff auf benannte Xaml-Elemente und enthält Funktionalitäten, die die View betreffen. In diesem Fall (Siehe Listing A.3) wird eine Instanz des ViewModels als DataContext gesetzt. Dieser Schritt ist notwendig, um der View ein ViewModel zu zuweisen, von dem die benutzten Daten bezogen werden sollen.

2.9.3. Beispiel ViewModel

Das ViewModel (Siehe Listing A.4) verwaltet eine Instanz der Person-Klasse und stellt deren Daten als formatierten String der View zur Verfügung. Das Alter der Person-Instanz

kann über die Methode „IncreaseAge()“ erhöht werden, um die View über diese Änderung zu informieren, implementiert das ViewModel das „INotifyPropertyChanged“ Interface. Dies benachrichtigt die View beim Aufruf, dass sich eine Property verändert hat.

Um dem Knopf in der View eine Funktion zu geben, wird im ViewModel ein Command erstellt, der beim Auslösen das IncreaseAge() im ViewModel aufruft.

2.9.4. Beispiel Command

Die IncreaseAgeCommand-Klasse (Siehe Listing A.5) implementiert das Interface „ICommand“. Das Interface fordert zwei Methoden und ein Event. Die Execute-Methode enthält die Schritte die ausgeführt werden, wenn der Command ausgeführt wird, in diesem Fall wird auf dem ViewModel die IncreaseAge-Methode aufgerufen. Die CanExecute-Methode, die bestimmt, ob ein Command ausgeführt werden kann. Das Event ist das CanExecuteChanged-Event, das aufgerufen werden sollte, wenn sich die Ausführbarkeit eines Commands ändert. Da sich die Ausführbarkeit in diesem Beispiel nicht ändert, wird dieses Event nicht aufgerufen. Der Effekt des Commands ist in Abbildung 2.2 auf Seite 13 zu sehen. Das Alter hat sich um 1 erhöht, von 25 auf 26.

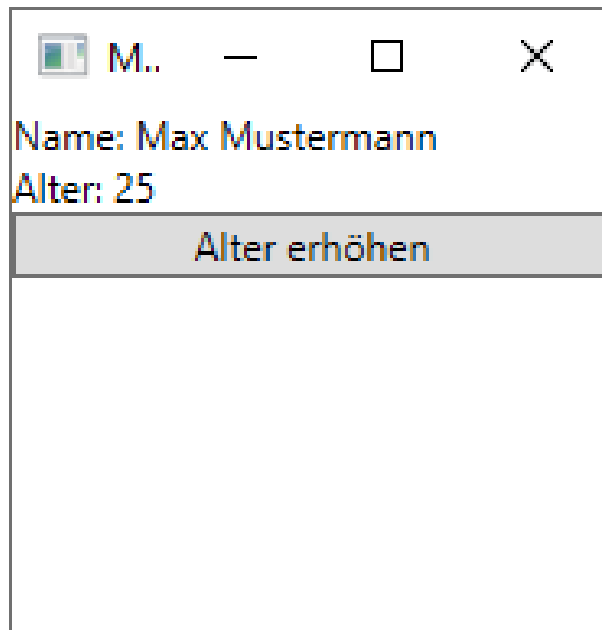


Abbildung 2.1.: Screenshot des Beispiel Programms

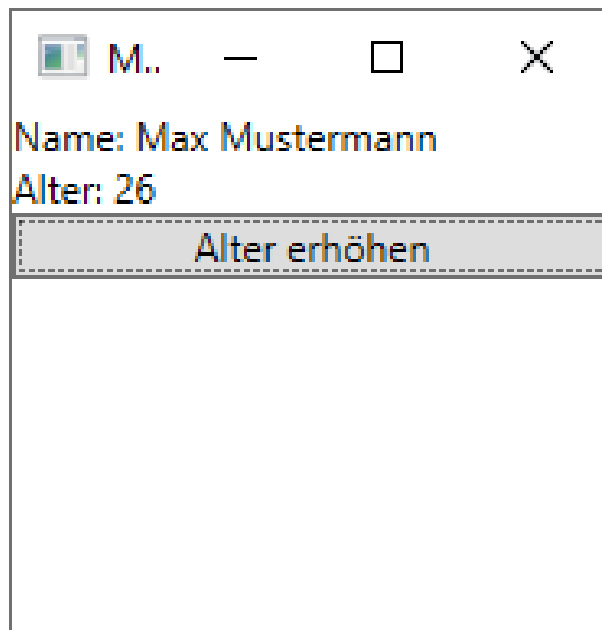


Abbildung 2.2.: Screenshot des Beispielprogramms nach dem Klick auf „Alter erhöhen“

3. Anforderungsanalyse

3.1. Ziele des Struktureditors

Ziel dieser Arbeit ist die Konzeption und die prototypische Umsetzung eines neuen Struktureditors für GenX-Dateien. Die Entwicklung eines neuen Struktureditors soll es erlauben unternehmenseigene Anforderungen einfacher und spezifischer umzusetzen, als der bestehende AutoCAD Struktureditor es erlaubt.

3.2. Nutzer des Struktureditors

Softwareentwickler Der Struktureditor wird von Softwareentwicklern genutzt, um Datenmodelle zu erweitern und anzupassen. Diese kennen die Funktionen der verschiedenen Datenmodell-Objekte und wissen diese zu bearbeiten oder zu erstellen.

3.3. Systemkontext

Der Struktureditor wird als Windows Desktop-Anwendung entwickelt. Grund dafür ist, dass der Editor unternehmensintern genutzt werden soll, um die Entwicklung von Datenmodellen zu erleichtern.

3.4. Funktionale Anforderungen

Für den neuen Struktureditor wurden folgende funktionale Anforderungen festgestellt:

Dateiverarbeitung

FA-1 Das Programm muss Genx-Dateien öffnen können.

FA-2 Das Programm muss Genx-Dateien speichern können.

FA-3 Das Programm muss ein neues Datenmodell erstellen können.

Datenmodell-Bearbeitung

FA-4 Das Programm muss neue Objekte erstellen können.

FA-5 Das Programm muss bestehende Objekte löschen können.

FA-6 Das Programm muss die Eigenschaften eines Objektes ändern können.

FA-7 Das Programm muss ein neues Strukturupdate erstellen können.

Darstellung

FA-8 Das Programm muss die Objekte des Datenmodells in einer hierarchischen Struktur darstellen.

FA-9 Das Programm muss die Eigenschaften eines Objektes anzeigen können.

FA-10 Das Programm muss Änderungen anzeigen können.

3.5. Nicht-funktionale Anforderungen

Für den neuen Struktureditor wurde folgende nicht-funktionale Anforderungen festgestellt.

FA-11 Das Programm muss intuitiv benutzbar sein.

FA-12 Das Programm soll eine konsistente Bedienung bieten.

Die typischen leistungsbezogenen Anforderungen wurde nicht als Anforderungen festgelegt, weil es sich bei dem entwickelten Programm um einen Prototypen handelt.

3.6. Einschränkungen

Weil dieses Programm unternehmensintern bei der TKI mbh genutzt und erweitert werden soll, ergeben sich folgende Einschränkungen.

- Entwicklung mit C#
- Entwicklung mit Windows Presentation Foundation
- Nutzung des MVVM-Entwurfsmusters
- Nutzung des bestehenden Models

Die TKI mbh entwickelt ihre Software standardmäßig mit VB.net und C#, deswegen soll dieser Struktureditor ebenfalls in C# geschrieben werden. Für Anwendungen mit einer Benutzeroberfläche nutzt die TKI mbh das Windows Presentation Foundation Framework, das sich gut mit VB.net oder C# kombinieren lässt. Für Anwendungen, die mit WPF entwickelt werden, verwendet die TKI mbh das Model-View-ViewModel-Entwurfsmuster zur Strukturierung des Programms (siehe Kapitel 2.8 auf Seite 10), weswegen der neue Struktureditor ebenfalls damit entwickelt werden soll. Zur internen Darstellung der Objekte eines Map 3d Datenmodells, wird ein, von der TKI mbh, bereits entwickeltes Model genutzt.

4. Konzept

4.1. Planung der Benutzeroberfläche

Damit der neue Struktureditor intuitiv benutzbar und konsistent bedienbar ist (siehe Anforderungen FA-11 und FA-12 auf Seite 15), wird die Benutzeroberfläche nach dem Vorbild des bestehenden Struktureditor gestaltet.

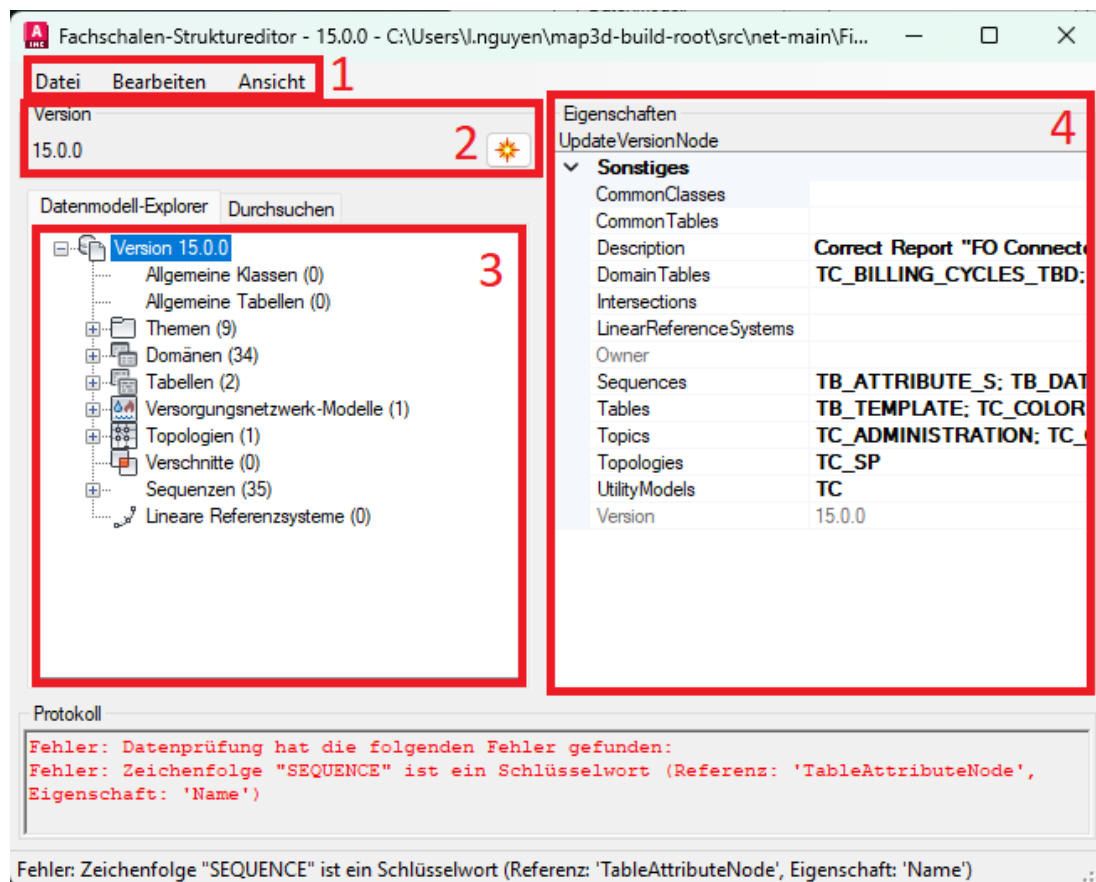


Abbildung 4.1.: Benutzeroberfläche des AutoCAD Struktureditors mit einem geladenen Datenmodell

Die für den neuen Struktureditor als wichtig befundenen Bestandteile sind:

1. Menüleiste
2. Versionsanzeige
3. Hierarchische Darstellung des geladenen Datenmodells
4. Eigenschaftenbereich

4.1.1. Menüleiste

Die Menüleiste erlaubt es dem Nutzer, über den Menüpunkt „Datei“, Genx-Dateien zu erstellen, zu öffnen und zu speichern. Der Menüpunkt „Bearbeiten“ erlaubt es ein Datenmodell aus einer Fachschale zu erstellen, diese Funktion wurde für den neuen Struktureditor jedoch als unwichtig befunden. Der Menüpunkt „Ansicht“ erlaubt das Ein- und Ausschalten des Protokolls und des Eigenschaftenbereichs. Der „Ansicht“-Menüpunkt wurde ebenfalls als unwichtig befunden, weil der Eigenschaftenbereich immer angezeigt werden soll und das Protokoll vollständig entfällt.

4.1.2. Versionsanzeige

Die Versionsanzeige zeigt die aktuellste Version des geladenen Datenmodells. Außerdem erlaubt es das Erstellen einer neuen Datenmodell-Version.

4.1.3. Darstellung des Datenmodells

Für die Navigation in einem Datenmodell ist die Hierarchische Darstellung dessen zuständig. Das geladene Datenmodell wird in einer Baumansicht angezeigt. Befüllte Knoten lassen sich auf- und zuklappen. Wenn ein Knoten ausgewählt wird im Eigenschaftenbereich die Eigenschaften des Knotens angezeigt.

4.1.4. Eigenschaftenbereich

Der Eigenschaftenbereich erlaubt das Bearbeiten der Eigenschaften eines ausgewählten Knotens.

4.2. Planung der Benutzerinteraktion

Wie die Benutzeroberfläche soll die Bedienung auch dem AutoCAD Struktureditor ähnlich sein. Im AutoCAD Struktureditor verläuft die Benutzerinteraktion meist über Dropdown-Menüs oder Kontextmenüs.

4.2.1. Interaktion für die Dateiverarbeitung

Die Interaktion mit einer Datei (siehe Anforderungen FA-1 bis FA-3) verläuft über den Menüpunkt „Datei“ in der Menüleiste (Siehe Nummer 1 in Abbildung 4.1 auf Seite 16). Bei einem Klick auf den Menüpunkt öffnet sich ein Dropdown-Menü mit den Menüpunkten „Neu“, „Öffnen“, „Speichern“ und „Speichern unter“. Diese Menüpunkte können während der Benutzung des Editors auch über Tastenkombinationen aufgerufen werden. Dies erlaubt das Erstellen eines Datenmodells, das Öffnen und das Speichern einer bestehenden Genx-Datei und das neu Erstellen einer Genx-Datei. Siehe Abbildung 4.2

4.2.2. Interaktion mit dem Datenmodell

Die Interaktionen zur Bearbeitung eines Datenmodells verläuft über die Datenmodell-darstellung und dem Eigenschaftenbereich (Siehe Nummer 3 und 4 in Abbildung 4.1 auf Seite 16).

Anforderung FA-4 und FA-5: Objekte erstellen und löschen Das erstellen neuer und löschen bestehender Objekte verläuft über das Kontextmenü eines Objektes in der Datenmodellldarstellung. Je nachdem welche Objektart, beim Öffnen des Kontextmenüs, angeklickt wurde, können die, für diese Objektart, gültigen Unterobjektarten erstellt werden. Zum Beispiel können bei einem Themen-Objekt Feature-Klassen, Ansichten und weitere Themen erstellt werden. Im Gegensatz dazu können bei einer Domäne nur Domäneneinträge erstellt werden. Analog dazu können ausgewählte Objekte über das Kontextmenü gelöscht werden. Siehe Abbildung 4.3

Anforderung FA-6: Eigenschaften bearbeiten Die Auswahl eines Objektes in der Datenmodellldarstellung aktualisiert den Eigenschaftenbereich und zeigt die Eigenschaften des ausgewählten Objektes an. Der Eigenschaftenbereich erlaubt es diese Eigenschaften zu bearbeiten, dabei kann die Eigenschaft entweder direkt beschrieben werden oder es kann ein vorgegebenen Wert ausgewählt werden. Beispielsweise kann bei einem Feature-Klasse-Objekt der Name direkt beschrieben werden, aber der Typ muss vorgegebenen Werten ausgewählt werden. Siehe Abbildung 4.4

Anforderung FA-7: Neues Strukturupdate erstellen Das Erstellen eines neuen Strukturupdates verläuft über einen Knopf in der Versionsanzeige (Siehe Nummer 2 in Abbildung 4.1 auf Seite 16). Bei einem Klick auf diesen Knopf öffnet sich ein Pop-Up-Fenster, in dem der Nutzer nach einer neuen Versionsnummer gefragt wird. Nach der Eingabe und der Bestätigung, werden Änderungen an dem Datenmodel der neu erstellten Version zugeschrieben. Siehe Abbildung 4.5

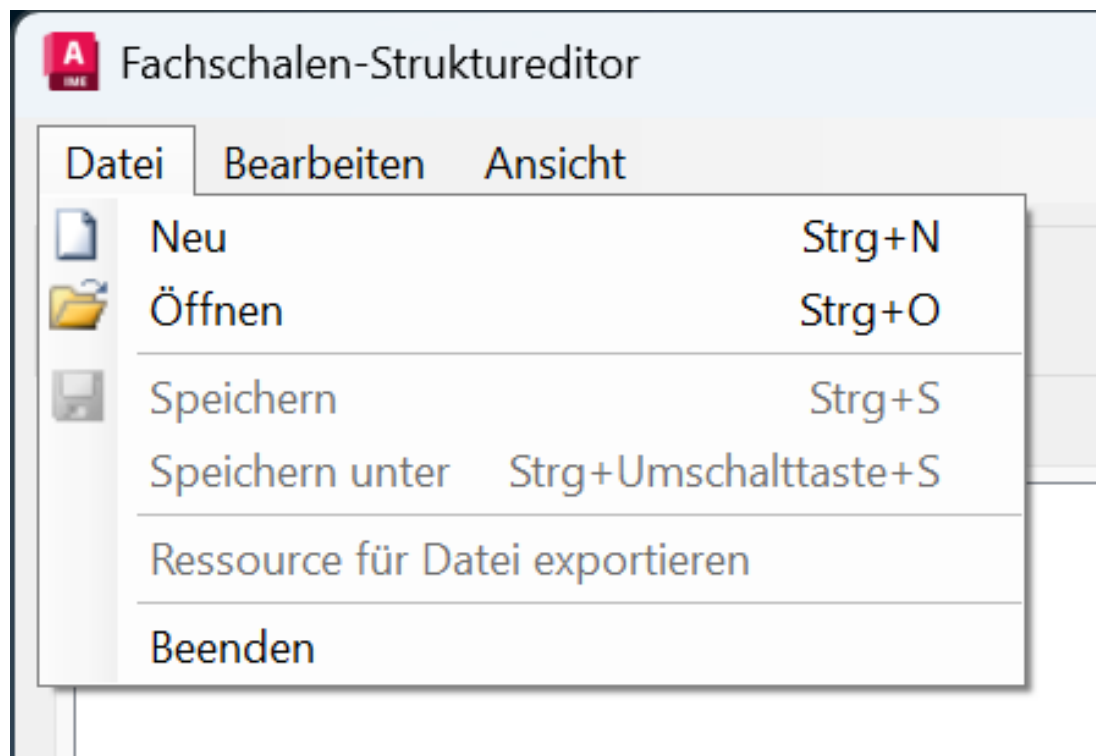


Abbildung 4.2.: Menüpunkt „Datei“ mit Neu, Öffnen, Speichern und Speichern unter

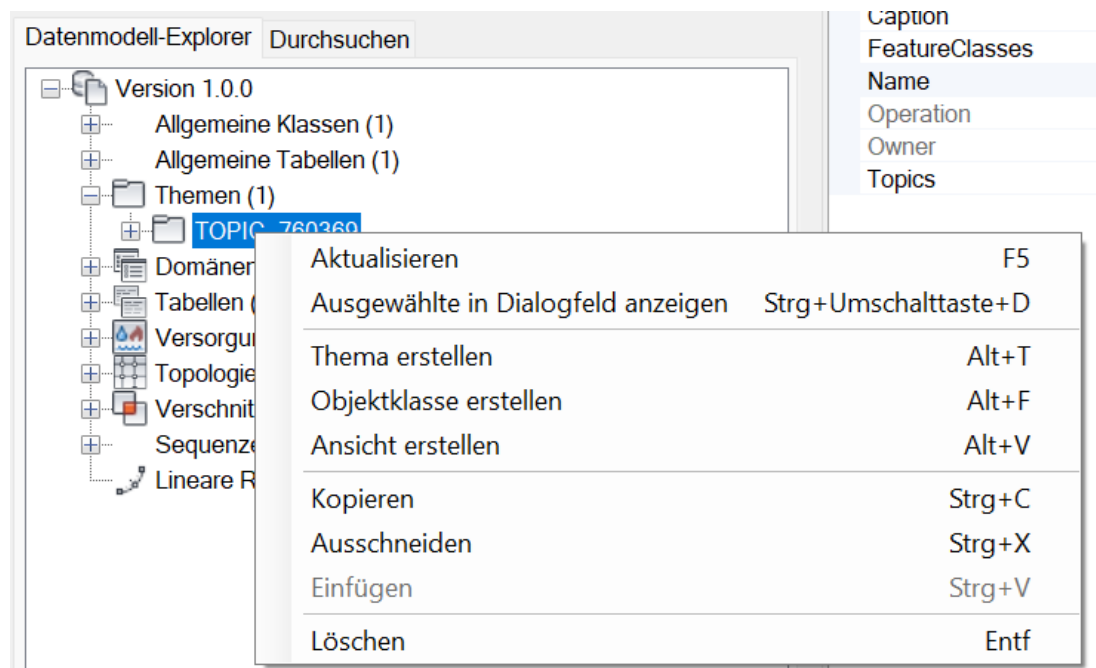


Abbildung 4.3.: Kontextmenü eines Thema-Objektes in der Datenmodellldarstellung

Eigenschaften
FeatureClassNode

▼ Sonstiges	
Attributes	ATTR_700980
Caption	
CommonClasses	
FeatureClasses	FEATURE_CLASS_978549
Indexes	INDEX_149041
ModelFeatureClass	
Name	FEATURE_CLASS_584003
Operation	Add
Owner	
Type	Sammlung ▼

Attribut
 Bemaßung
 Label
 Polygon
 Polylinie
 Punkt
Sammlung
 Verbund-Polygon
 Verbund-Polylinie
 Zentroid

Abbildung 4.4.: Attributauswahl mit vorgegebenen Werten

Versionsnummer erforderlich ×

Geben Sie eine neue Versionsnummer ein:

OK

Cancel

1.0.1

Abbildung 4.5.: Pop-Up-Fenster der Versionseingabe bei einem neuen Strukturupdate

4.3. Umgang mit dem bestehenden Model

Für den neuen Struktureditor soll ein bereits entwickeltes Model genutzt werden. Dieses bestehende Model wurde von der TKI mbh bereits entwickelt. Das Model beinhaltet die Datenmodell-Objekte gebündelt in einem GenXStructure-Objekt und einen GenXParser.

4.3.1. GenXParser

Der GenXParser ist verantwortlich für das Einlesen der Genx-Datei und das Erstellen der Datenmodell-Objekte und der GenXStructure. Dafür bietet der GenXParser die Funktion **GetStructureFromGenX(string genxContent)** an. Diese nimmt den Inhalt der Genx-Datei als **string** entgegen und gibt das daraus entstehende **GenXStructure**-Objekt zurück.

4.3.2. GenXStructure-Klasse und Datenmodell-Objekte

Für die interne Darstellung eines Datenmodells ist die GenXStructure-Klasse verantwortlich. Diese Klasse beinhaltet ein Dictionary, für jede Objektart, in denen die Datenmodell-objekte gespeichert sind. Die Dictionaries haben einen Schlüssel vom Typ String, der vom Namen des Datenmodellobjektes stammt. Der Wert des Dictionaries ist das Datenmodell-objekt selbst. Zum Beispiel wird eine Feature-Klasse mit dem Name **Person** als Schlüssel-Wert-Paar wie folgt gespeichert: ("Person", Person-Feature-Klasse-Objekt)

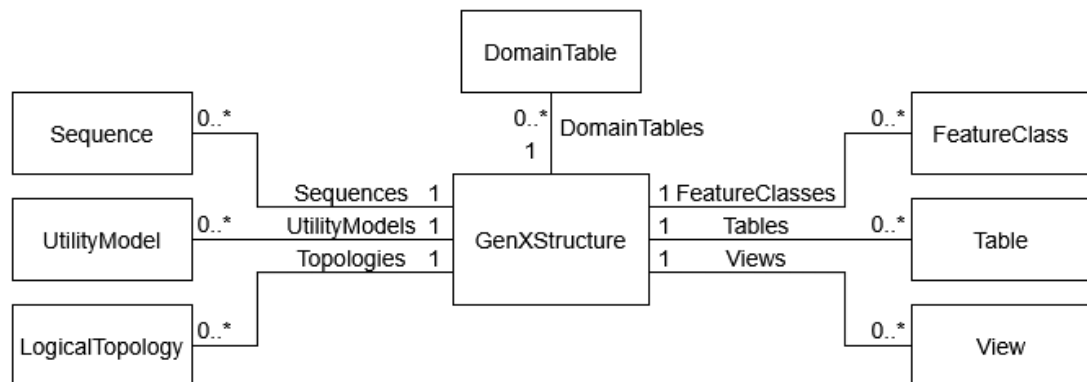


Abbildung 4.6.: vereinfachtes Klassendiagramm von GenXStructure und Datenmodellobjekten

Das von der TKI mbh entwickelte Model ist jedoch unvollständig und deckt nicht alle AutoCAD Datenmodellobjekte ab. Es werden nur die Datenmodellobjekte abgebildet, die für die TKI selbst von Relevanz sind. Das sind „Feature-Klassen“, „Tabellen“, „Ansichten“, „Domänen“, „Sequenzen“, „Versorgungsmodelle“ und „logische Topologien“. Diese werden in der GenXStructure abgespeichert (Siehe Abbildung 4.6)

4.3.3. Probleme des bestehenden Models

Das bestehende Model weist einige Probleme auf, die während der Entwicklung noch gelöst werden müssen.

Fehlende Themen Klasse

Das bestehende Model besitzt keine Klasse für das Datenmodellobjekt Thema, dadurch fehlt ein wichtiger Bestandteil zur Strukturierung eines Datenmodells.

Verschachtelte Objekte

In einem Datenmodell befinden sich Feature-Klassen und Ansichten in einem Thema. Dies sorgt für eine saubere Trennung der Objekte. Für eine genauere Unterteilung können Themen weitere Themen enthalten, ebenso können Feature-Klassen weitere Feature-Klassen und Ansichten beinhalten. Diese Verschachtelung wird in dem TKI Model unvollständig berücksichtigt. In der FeatureClass-Klasse wird nur der Name der nächsthöheren Eltern-Feature-Klasse und des nächsthöchsten Themas gespeichert. Das führt dazu, dass die Information über verschachtelte Themen und Feature-Klassen verloren geht.

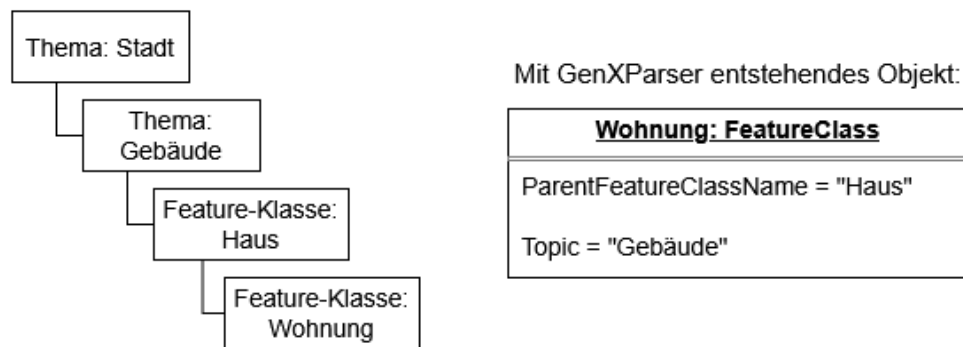


Abbildung 4.7.: Links: Beispiel-Datenmodell mit verschachtelten Objekten, Rechts: Ergebnis der GenXParser der Wohnung Feature-Klasse

In Abbildung 4.7 wird links ein Beispiel-Datenmodell gezeigt, das aus zwei verschachtelten Themen und zwei verschachtelten Feature-Klassen besteht. Wenn dieses Datenmodell mit dem GenXParser eingelesen wird, entsteht ein GenXStructure-Objekt das zwei Feature-Klassen enthält. Das Problem mit den Feature-Klassen-Objekten ist, dass das Feld **Topic**, das den Namen des nächsthöchsten Themas enthält, mit dem Wert **Gebäude** befüllt wird. Die Feature-Klassen-Objekte verlieren die Information, dass sie sich auch in dem Thema **Stadt** befinden (Siehe Abbildung 4.7 rechts). Ähnliches würde auch geschehen, wenn die Feature-Klasse **Wohnung** eine weitere Kind-Feature-Klasse hätte. Diese würde den Namen der **Wohnung**-Feature-Klasse abspeichern, jedoch nicht den der **Haus**-Feature-Klasse.

Ähnliches passiert auch bei der View-Klasse. Hier fehlt die Beziehung zu Feature-Klasse und Thema vollständig und wird nicht gespeichert.

Unzureichende Zugriffsmodifikatoren

Viele der Klassen in dem bestehenden Model haben den Zugriffsmodifikator **internal**. Dieser Zugriffsmodifikator beschränkt den Zugriff auf die eigene Assembly [5].

Da der neue Struktureditor ein eigenes Projekt sein soll, führt das zu Problemen, wenn auf solche Klassen zugegriffen werden soll oder ein Objekt einer solche Klasse erstellt werden soll.

Beispielsweise gibt es in dem bestehenden Model die Klasse **TableField**, die mit dem Zugriffsmodifikator **internal** definiert wurde (Siehe Listing 4.1). Diese Klasse kann in dem bestehenden Model-Projekt problemlos verwendet werden. Für den Struktureditor ist der Zugriff auf diese Klasse nicht möglich. Die Klasse kann nicht instanziiert werden und es kann nicht auf die Klasse zugegriffen werden.

Listing 4.1: Klasse mit dem Zugriffsmodifikator Internal

```
1 internal class TableField
2 {
3     public TableField(...) {...}
4 }
```

4.3.4. Lösungskonzepte der Probleme

Lösungskonzept für die fehlende Themen Klasse

Die fehlende Thema-Klasse kann nachträglich erstellt werden. Diese wird analog zu den bereits entwickelten Datenmodellobjekten ebenfalls in einer GenXStructure-Instanz abgespeichert. Um die Struktur der Themen korrekt abzuspeichern benötigen die Themen-Objekte eine Eigenschaft, die die Position des Themen-Objektes im Datenmodell widerspiegelt.

Lösungskonzept für verschachtelte Objekte

Um die Verschachtelung einer Feature-Klasse und einer View abzuspeichern, muss in den Klassen nicht nur das nächsthöchste Eltern-Objekt und Thema abgespeichert werden. Es müssen extra Informationen über die genaue Position abgespeichert werden. ein weiteres Feld hinzugefügt werden, das die Relation zu den Themen und den Eltern-Objekten abspeichert.

Lösungskonzept für unzureichende Zugriffsmodifikatoren

Eine Lösung für die unzureichenden Zugriffsmodifikatoren ist, diese zu erweitern, sodass auf diese auch außerhalb der Assembly zugegriffen werden kann.

5. Umsetzung

5.1. Benutzeroberfläche

5.1.1. Container Struktur

Für die Benutzeroberfläche wird eine Kombination aus Grid- und StackPanel-Objekten benutzt, die innerhalb eines `<Window>`-Elements definiert sind. Das `<Window>`-Element erzeugt ein Anwendungsfenster des Windows-Betriebssystems. Das Anwendungsfenster bietet Windows-Funktionalitäten wie „Minimieren“, „Maximieren“ oder „Schließen“. Das Grid-Objekt erlaubt das Platzieren der UI-Elemente in einem definierten Raster. In dieser Anwendung wird ein Grid mit zwei Spalten und drei Zeilen definiert. Siehe Abbildung 5.1.

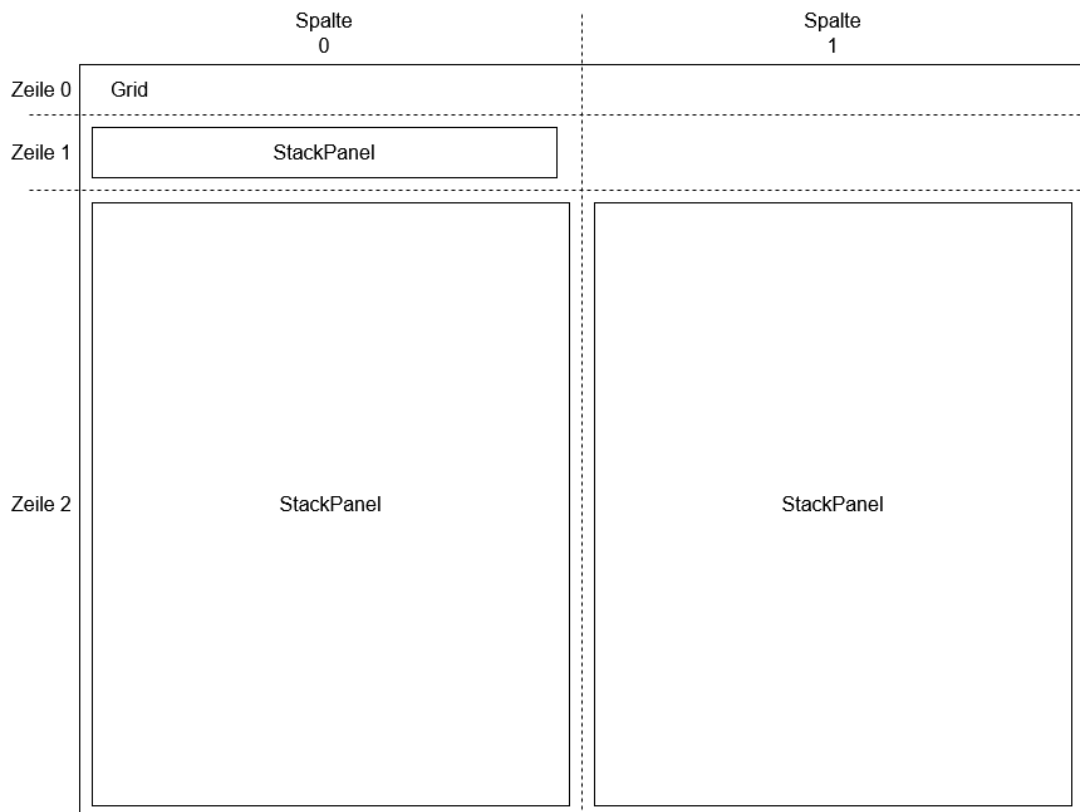


Abbildung 5.1.: Skizze des Grids und der StackPanel

In Zeile 1, Spalte 0, Zeile 2, Spalte 0 und Zeile 2, Spalte 1 werden StackPanel definiert, ein Objekt, welches Inhalte horizontal oder vertikal anordnet. In diesem Fall werden alle StackPanel vertikal ausgerichtet.

Listing 5.1: Auszug der Xaml-Definition des Grids und der StackPanels

```

1 <Window x:Class="Datenmodell_Editor"
2     <!-- XML-Namespace-Definitionen -->
3     Title="Datenmodell Editor"
4     Height="600"
5     Width="800">
6     <Grid>
7         <Grid.RowDefinitions>
8             <RowDefinition Height="Auto"/>
9             <RowDefinition Height="Auto"/>
10            <RowDefinition Height="*/>
11        </Grid.RowDefinitions>
12        <Grid.ColumnDefinitions>
13            <ColumnDefinition Width="*/>
14            <ColumnDefinition Width="*/>
15        </Grid.ColumnDefinitions>
16
17        <StackPanel Orientation="Vertical"
18            Margin="5, 5, 2.5, 2.5"
19            Grid.Row="1" Grid.Column="0">
20            <!-- Inhalt der Versions-
21                anzeige ausgelassen -->
22        </StackPanel>
23
24        <StackPanel Orientation="Vertical"
25            Margin="5, 2.5, 2.5, 5"
26            Grid.Row="2" Grid.Column="0">
27            <!-- Inhalt der Datenmodell-
28                darstellung ausgelassen -->
29        </StackPanel>
30
31        <StackPanel Orientation="Vertical"
32            Margin="2.5, 2.5, 5, 5"
33            Grid.Row="2" Grid.Column="1">
34            <!-- Inhalt des Eigenschaften-
35                bereichs ausgelassen -->
36        </StackPanel>
37    </Grid>
38 </Window>

```

Für die Definition des Grids, werden `<RowDefinition>`- und `<ColumnDefinition>`-Elemente in `<Grid.RowDefinitions>` und `<Grid.ColumnDefinitions>` definiert. Den RowDefinition-Elementen werden mit dem `Height`-Attribut eine Höhe definiert, analog dazu werden die ColumnDefinition-Elemente mit dem `Width`-Attribut definiert. Die `Width`- und `Height`-Attribute erlauben verschiedene Arten der Größenangabe. Der Wert `Auto` passt die Größe an den Inhalt der Zeile oder Spalte an, das heißt die Zeile oder Spalte wird so klein wie möglich ohne die Größe des Inhalts zu ändern. Der Wert `*`, gleichbedeutend mit `1*`, erlaubt eine gewichtete proportionale Größe des verbleibenden Platzes, das heißt zwei ColumnDefinitions mit `Width="*"` sind gleich groß zueinander. Bei unterschiedlichen Größenangaben, wie zum Beispiel Spalte 0 mit `Width="2*"` und

Spalte 1 mit `Width="1★"`, ist Spalte 0 doppelt so breit wie Spalte 1. [6]

Für eine bessere Übersichtlichkeit wird für die `StackPanel`-Elemente ein `Margin`-Attribut festgelegt. Eine `Margin` ist der äußere Abstand eines UI-Elements zu anderen UI-Elementen. Für den Struktureditor wurde ein allgemeiner Abstand von 5 gewählt. Bei den `Margins` zwischen anderen UI-Elementen wurde darauf geachtet, dass der Abstand zwischen den beiden UI-Elementen 5 beträgt. Das heißt auf der Seite zwischen UI-Elementen haben beide `Margins` einen Wert von 2,5.

Mit dem `Grid.Row`- und `Grid.Column`-Attribut werden die `StackPanel` in die vorgesehenen `Grid`-Zellen platziert. Die Spalten und Zeilen eines `Grid` sind null-basiert und beginnt in der oberen linken Ecke.

5.1.2. UI-Elemente

Menüleiste

Die Menüleiste wird mit einem `<Menu>`-Element erstellt. Dieses wird in Zeile 0, Spalte 0 und mit einem `ColumnSpan` von 2 definiert. Das `Grid.ColumnSpan`-Attribut erlaubt es einem UI-Element sich über mehrere Spalten zu strecken. Um die Menüleiste zu befüllen, werden innerhalb des `<Menu>`-Elements `<MenuItem>`-Elemente definiert.

Das `<MenuItem>`-Element bietet viele Attribute, um den Menüpunkt zu gestalten. In diesem Fall sind `Header`, `InputGestureText` und `Command` relevant. Das `Header`-Attribut definiert den Text, der in dem Menüpunkt primär angezeigt wird. Das `InputGestureText`-Element definiert einen Text am Ende eines Menüpunkts, der die Tastenkombination anzeigt, mit der man den Menüpunkt alternativ aufrufen kann, jedoch handelt es sich hierbei um den reinen Text. Die Tastenkombination wird an einer anderen Stelle in der Xaml-Code definiert, das wird im folgenden Abschnitt näher behandelt. Das `Command`-Element bindet ein `ICommand`-Objekt des ViewModels, der Command ausgeführt wird, wenn der Menüpunkt angeklickt wird.

Das `<MenuItem>`-Element erlaubt auch eine Verschachtelung weiterer `<MenuItem>`-Elemente, dies erlaubt eine saubere Trennung und Kapselung der Menüpunkte. In diesem Fall werden die Menüpunkte `Neu`, `Öffnen`, `Speichern` und `Speichern unter` im Menüpunkt `Datei` gesammelt.

Um Menüpunkte innerhalb eines `<MenuItem>`-Elements nochmals zu unterteilen wird ein `<Seperator/>`-Element definiert. Dieses fügt eine dünne horizontale Linie ein, die eine Trennung der Funktionalität veranschaulichen soll

Listing 5.2: Xaml-Definition der Menüleiste

```

1
2 <Menu Grid.Row="0"
3     Grid.Column="0"
4     Grid.ColumnSpan="2"
5     Height="24">
6     <MenuItem Header="Datei">
7         <MenuItem Header="Neu" InputGestureText="Strg+N"
8             Command="{Binding FileNewCommand}"/>
9         <MenuItem Header="Öffnen" InputGestureText="Strg+O"
10            Command="{Binding FileOpenCommand}"/>
11         <Separator />
12         <MenuItem Header="Speichern" InputGestureText="Strg+S"
13             Command="{Binding FileSaveCommand}"/>
14         <MenuItem Header="Speichern unter"
15             InputGestureText="Strg+Shift+S"
16             Command="{Binding FileSaveAsCommand}"/>
17     </MenuItem>
18 </Menu>

```

Tastenkombinationen

Für die Definitionen der Tastenkombinationen bietet das `<Window>`-Element die Property `<Window.InputBindings>`, in der `<KeyBinding>`-Elemente definiert werden können. In den `<KeyBinding>`-Elementen werden die Tastenkombinationen und Commands verknüpft, dafür wird ein `Key`, ein oder mehr `Modifiers` und ein `Command` gesetzt. Das `Key`-Attribut spezifiziert die Tastaturtaste die gedrückt werden soll. Das `Modifiers`-Attribut spezifiziert, welche Hilfstasten in Kombination mit der Tastaturtaste gedrückt werden sollen, um den `Command` auszuführen. Ein wichtiger Punkt bei `<KeyBinding>`-Elementen ist, dass diese direkt an einen Command gebunden sind und nicht an einen Menüpunkt. Die Verbindung zwischen einem Menüpunkt und einer Tastenkombination ist der gleiche gebundene Command.

Listing 5.3: Xaml-Definition der Tastenkombinationen

```

1 <Window.InputBindings>
2     <KeyBinding Key="N" Modifiers="Ctrl"
3         Command="{Binding FileNewCommand}"/>
4     <KeyBinding Key="O" Modifiers="Ctrl"
5         Command="{Binding FileOpenCommand}"/>
6     <KeyBinding Key="S" Modifiers="Ctrl"
7         Command="{Binding FileSaveCommand}"/>
8     <KeyBinding Key="S" Modifiers="Ctrl+Shift"
9         Command="{Binding FileSaveAsCommand}"/>
10 </Window.InputBindings>

```

Versionsanzeige

Die Versionsanzeige wird in Zeile 1, Spalte 0 des Grids definiert. Hier werden in einem `<StackPanel>` ein `<TextBlock>` und ein weiteres `<Grid>` definiert. Der `<TextBlock>` zeigt den Text **Version** an. In dem `<Grid>` werden 3 Spalten definiert. Spalte 0 und Spalte 2 haben eine Breite von **Auto** und Spalte 1 hat eine Breite von *****. Dies sorgt dafür, dass UI-Elemente in Spalte 0 sich am linken Rand und UI-Elemente in Spalte 2 am rechten Rand befinden. In Spalte 0 wird ein weiterer `<TextBlock>` definiert, dessen Text an die `VersionText`-Property des ViewModels gebunden ist. In Spalte 2 wird ein `<Button>` definiert, der den Text **Neue Version** anzeigt.

Listing 5.4: Xaml-Definition der Versionsanzeige

```

1 <StackPanel Orientation="Vertical"
2     Grid.Row="1" Grid.Column="0"
3     Margin="5, 5, 2.5, 2.5">
4     <TextBlock Text="Version"/>
5     <Grid>
6         <Grid.ColumnDefinitions>
7             <ColumnDefinition Width="Auto"/>
8             <ColumnDefinition Width="*/>
9             <ColumnDefinition Width="Auto"/>
10        </Grid.ColumnDefinitions>
11
12        <TextBlock Text="{Binding VersionText}"
13            FontSize="13" Margin="5,3,0,0"
14            Grid.Column="0"/>
15        <Button Grid.Column="2" Content="Neue Version" />
16    </Grid>
17 </StackPanel>

```

Datenmodellldarstellung

Die Datenmodellldarstellung wird in Zeile 2, Spalte 0 des Grids definiert. Das `<StackPanel>` wird mit einem `<TextBlock>` und einem `<TreeView>` Element befüllt.

Der `<TextBlock>` zeigt den Text **Datenmodell-Explorer** an. Das `<TreeView>`-Element zeigt das geladene Datenmodell als hierarchische Struktur (Siehe Anforderung FA-8 auf Seite 15). Dieses Element ist besonders geeignet, weil es eine einfache Möglichkeit bietet Objekte in einer hierarchischen Struktur mit einklappbaren Knoten anzuzeigen. Die Knoten eines `<TreeView>`-Elements werden dynamisch mit einer Kombination aus `<HierarchicalDataTemplate>`-Elementen und dem `ItemContainerStyle` erstellt.

Das `<HierarchicalDataTemplate>`-Element bestimmt den Inhalt eines Knotens, anhand des Typs des Quellobjekts. Hier werden Knoten des Typs `TreeNodeBase` mit einem `<TextBlock>`, der den Namen des Knotens angibt, definiert. Mit dem `ItemContainerStyle` wird definiert, wie sich ein erstellter Knoten verhält. In diesem Fall werden die Properties `IsSelected`, `IsExpanded` und `ContextMenu` definiert. `IsSelected` ist ein Property mit booleschem Wert, das angibt, ob der Knoten ausgewählt wurde. `IsExpanded` gibt an, ob die Kindknoten eines Knotens ausgeklappt sind, dies ist ebenfalls ein boolescher Wert. Das `ContextMenu`-Property definiert das Kontextmenü eines Knotens. Dies wird im Abschnitt 5.3.6 auf Seite 38 näher erklärt

Listing 5.5: Xaml-Definition der Datenmodelldarstellung

```

1 <StackPanel Orientation="Vertical"
2     Grid.Row="2" Grid.Column="0"
3     Margin="5, 2.5, 2.5, 5">
4     <TextBlock Text="Datenmodell-Explorer"/>
5     <TreeView Height="400"
6         ItemsSource=
7             "{Binding CurrentStructureViewModelList}">
8         <TreeView.Resources>
9             <HierarchicalDataTemplate
10                 DataType="{x:Type genxViewModels:TreeNodeBase}"
11                 ItemsSource="{Binding Children}">
12                 <TextBlock Text="{Binding NodeCaption}"/>
13             </HierarchicalDataTemplate>
14         </TreeView.Resources>
15
16         <TreeView.ItemContainerStyle>
17             <Style TargetType="TreeViewItem">
18                 <Setter Property="IsSelected"
19                     Value="{Binding IsSelected, Mode=TwoWay}"/>
20                 <Setter Property="IsExpanded"
21                     Value="{Binding IsExpanded, Mode=TwoWay}"/>
22                 <Setter Property="ContextMenu">
23                     <Setter.Value>
24                         <ContextMenu ItemsSource="{Binding MenuItems}">
25                             <ContextMenu.ItemContainerStyle>
26                                 <Style TargetType="MenuItem">
27                                     <Setter Property="Header"
28                                         Value="{Binding Header}"/>
29                                     <Setter Property="Command"
30                                         Value="{Binding Command}"/>
31                                 </Style>
32                             </ContextMenu.ItemContainerStyle>
33                         </ContextMenu>
34                     </Setter.Value>
35                 </Setter>
36             </Style>
37         </TreeView.ItemContainerStyle>
38     </TreeView>
39 </StackPanel>

```

Eigenschaftsbereich

Der Eigenschaftsbereich wird in Zeile 2 und Spalte 1 definiert. Er besteht aus einem `<TextBlock>`-Element, das den Text **Eigenschaften** anzeigt, und einem `<DataGrid>`-Element, welches sich eignet Daten in tabellarischer Form anzuzeigen. In `<DataGrid>` wird mit `ItemsSource` definiert aus welcher Property die Datensätze stammen und mit `AutoGenerateColumns="False"` wird verhindert, dass die Spalten der Tabelle automatisch erstellt werden. Dies wird verhindert, weil die Spalten sonst nicht breit genug erstellt werden und die Spaltenüberschrift **Attribute** und **Werte** sein sollen. Die Spalten werden mit dem `<DataGridTextColumn>`-Element definiert, damit Textwerte in der Spalte angezeigt werden. Die zwei Spalten werden mit einer Breite von `1*` definiert, damit beide Spalten eine gleiche Breite haben. Die Überschrift der ersten Spalte wird mit `Header="Attribute"` definiert und bezieht seine Werte aus dem `Key`-Property des Quellobjektes. Die zweite Spalte wird mit `Header="Werte"` definiert und bezieht seine Werte aus dem `Value`-Property des Quellobjektes.

Listing 5.6: Xaml-Definition des Eigenschaftsbereichs

```

1 <StackPanel Grid.Row="2" Grid.Column="1"
2     Margin="2.5, 2.5, 5, 5">
3     <TextBlock Text="Eigenschaften"/>
4     <DataGrid ItemsSource="{Binding DisplayedNodeData}"
5         AutoGenerateColumns="False">
6         <DataGrid.Columns>
7             <DataGridTextColumn Header="Attribute" Width="1*"
8                 Binding="{Binding Key}"/>
9             <DataGridTextColumn Header="Werte" Width="1*"
10                 Binding="{Binding Value}"/>
11         </DataGrid.Columns>
12     </DataGrid>
13 </StackPanel>

```

5.1.3. Verbinden der View mit dem ViewModel

Um zwischen der View und dem ViewModel eine Verbindung zu schaffen, muss der `DataContext` der View gesetzt werden. Dies passiert bei der Erstellung der View in der CodeBehind-Datei. Hier wird, im Konstruktor, nach der Ausführung der `InitializeComponent`-Methode der `DataContext` auf eine neue Instanz der `MainViewModel`-Klasse gesetzt. Dies erlaubt es der View auf die öffentlichen Properties des `MainViewModel` zu zugreifen und diese mit Data Bindings zu binden.

Listing 5.7: Quelltext der CodeBehind-Datei der `MainWindow`-View

```

1 public partial class MainWindow : Window
2 {
3     public MainWindow()
4     {
5         InitializeComponent();
6         DataContext = new MainViewModel();
7     }
8 }

```

5.2. Umgang mit den Problemen des bestehenden Models aus Kapitel 4.3.3

5.2.1. Lösung für die fehlende Themen Klasse

Um die Themen eines Datenmodells darzustellen wurde eine neue Klasse **Topic** zum Model hinzugefügt. Diese Klasse enthält den Namen des Themas, die Operation die auf das Thema angewandt wird, eine eventuelle Caption und eventuelle **XmlAttributeTags**. Um die Position des Themas im Datenmodell zu definieren, wird eine Liste aus Strings, die **ParentTopicList** definiert, welche den Namen der Eltern-Themen, der Reihe nach vom höchsten Thema bis zum Eltern-Thema des Thema-Objektes, abspeichert. Diese Lösung ähnelt einem Dateipfad, in dem Dateien anhand ihrer Eltern-Objekte eindeutig positioniert werden können.

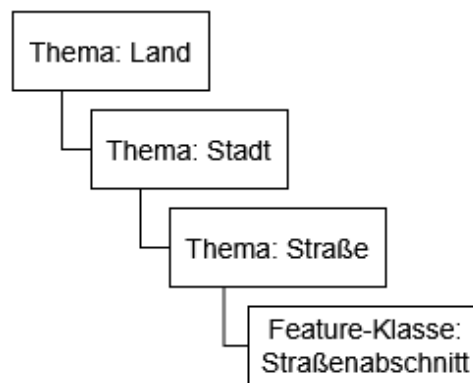


Abbildung 5.2.: Beispiel-Datenmodell mit drei verschachtelten Themen

In Abbildung 5.2 wird ein Beispiel-Datenmodell betrachtet, in dem drei Themen ineinander verschachtelt sind. Das oberste Thema ist „Land“, darauf folgt das Thema „Stadt“ und das unterste Thema ist „Straße“. Nach dem dieses Datenmodell analysiert wird, würden drei **Topic**-Objekte entstehen, jeweils eins für „Land“, „Stadt“ und „Straße“. Die **ParentTopicList**-Liste des Land Themas würde leer sein, weil es das oberste Thema-Element ist. Die **ParentTopicList**-Liste des Stadt Themas würde den Text "Land" enthalten, weil es ein Unterthema des Land Themas ist. Die **ParentTopicList**-Liste des Straßen Themas würde die Texte "Land" gefolgt von Stadt enthalten. Durch diese Speicherung der Eltern-Themen kann jedes Thema eindeutig positioniert werden.

Die Positionierung eines Themas wurde auch mit einer reinen String-Lösung versucht, bei der die Namen der Eltern-Themen durch Sonderzeichen wie „-“, „/“ oder „\“ getrennt wurden, ähnlich zu einem Dateipfad. Diese Lösung schlug jedoch fehl, weil während der Entwicklung kein simples Sonderzeichen gefunden wurde, welches nicht selbst im Namen eines Themas auftauchen konnte.

Listing 5.8: Quelltext der Model-Klasse Topic

```
1 public class Topic(string name, Operation? operation)
2 {
3     public string Name { get; set; } = name;
4     public Operation? Operation { get; set; } = operation;
5     public ResourceTag? Caption { get; set; }
6     public XmlAttributeTags? Tags { get; set; }
7     public List<string> ParentTopicList { get; set; }
8
9     internal void UpdateTopic(Topic newTopic)
10    {
11        this.Name = newTopic.Name;
12        this.Operation = newTopic.Operation;
13        this.Caption = newTopic.Caption;
14    }
15 }
```

5.2.2. Lösung für verschachtelte Objekte

Ähnlich zu den verschachtelten Themen wurden in der **FeatureClass**-Klasse und der **View**-Klasse Sammlungen eingefügt, die die Namen der Eltern-Objekte abspeichern. Analog zu der **Topic**-Klasse wurde eine Liste **TopicList** hinzugefügt, die die Namen der Themen abspeichert, in dem sich die Feature-Klasse oder die Ansicht befindet. Für die Speicherung der Verschachtelung der Feature-Klassen oder Ansichten wird ein Dictionary **ParentObjectNamesTypesDict** hinzugefügt, dessen Schlüssel der Name des Eltern-Objektes ist und dessen Wert der Typ des Eltern-Objektes ist. Betrachtet man die Wohnung-Feature-Klasse im Beispiel-Datenmodell in Abbildung 4.7 auf Seite 22, so wird im **ParentObjectNamesTypesDict** nun das Schlüssel-Wert-Paar ("Haus", **FeatureClass**) gespeichert. Hätte die Wohnung-Feature-Klasse eine untergeordnete Ansicht, so wären im **ParentObjectNamesTypesDict** die Schlüssel-Wert-Paare ("Haus", **FeatureClass**) und ("Wohnung", **FeatureClass**) gespeichert.

Der Grund für die zusätzliche Speicherung des Typs wird in Abschnitt 5.3.4 *Erstellung der Feature-Klassen und Ansichten* auf Seite 36 erläutert.

5.2.3. Lösung für unzureichende Zugriffsmodifikatoren

Um aus dem neuen Struktureditor-Projekt auf einige Klassen des Model zu greifen zu können müssen die Klassen-Zugriffsmodifikatoren erweitert werden. Die Lösung dafür ist simpel und die betroffenen Klassen werden vom Zugriffsmodifikator **internal** zu **public** geändert. Dies erlaubt den Zugriff auf die Klasse auch außerhalb des ursprünglichen Model-Projekts.

5.3. ViewModel

5.3.1. Haupt-ViewModel

Das Haupt-ViewModel für den neuen Struktureditor ist die Klasse `MainViewModel`. Dieses ViewModel ist die zentrale Steuerung des Programms und zuständig für die Bereitstellung der Daten für die Data Bindings. Diese Klasse stellt der View die Struktur für die Datenmodellldarstellung, die Daten für den Eigenschaftenbereich und die Commands für die Interaktion zur Verfügung. Das `MainViewModel` bietet einen Konstruktor, der einen `GenXParser` und einen `TreeNodeSelectionService` instanziiert, und zwei Methoden, die `setFileStream`-Methode, welche die Datenmodellstruktur aus einer Genx-Datei erstellt und die `UpdateDisplayedNodeData`-Methode, die die Daten des Eigenschaftenbereichs aktualisiert, wenn sich der ausgewählte Knoten ändert und dieser gültige Daten besitzt.

5.3.2. Angleichen des ViewModels an das Model

Um jede datenmodellrelevante Klasse des Models möglichst genau darzustellen, wurde für jede Model-Klasse eine eigene ViewModel-Klasse erstellt. Diese ViewModel-Klassen werden im Folgenden kollektiv unter dem Begriff „GenXElement-ViewModels“ zusammengefasst. Jede Instanz einer GenXElement-ViewModel-Klasse enthält eine Referenz auf das ursprüngliche Model-Objekt. Diese Spezifizierung an die einzelnen Model-Klassen erlaubt eine genauere Anpassung der ViewModel-Klasse an die Model-Klasse. Zum Beispiel enthält die ViewModel-Klasse `TopicViewModel` eine private Referenz auf das ursprüngliche Model-Objekt vom Typ `Topic`.

5.3.3. Zusätzliches ViewModel für die Darstellung der Sammelknoten eines Datenmodells

In der visuellen Darstellung eines Datenmodells werden Knoten gezeigt, welche nicht in der Genx-Datei des Datenmodells vorkommen. Es handelt sich um reine visuelle Kategorien. Dies sind beispielsweise die Knoten „Themen“, „Domänen“ oder „Objektklassen“.

Für diese Art Sammelknoten wurde ein zusätzliches ViewModel erstellt. Die `ObjectCollectionViewModel`-Klasse bildet diese Ordner-ähnlichen Strukturen ab. Die `ObjectCollectionViewModel`-Klasse speichert den Typen der untergeordneten Objekte, eine `ObservableCollection` und eine `CaptionPrefix` ab.

Der Typ bestimmt, welche Objekte innerhalb einer Instanz gespeichert werden können, es bestimmt auch die `CaptionPrefix`. Die `ObservableCollection` enthält die untergeordneten Objekte. Der `CaptionPrefix` ist ein string, der den Text des angezeigten Knotens bestimmt. Dies kann beispielsweise „Themen“, „Domänen“ oder „Objektklassen“ sein.

Die Klasse bietet drei Konstruktoren. Alle Konstruktoren erwarten einen angegebenen Typen, um den Typen der untergeordneten Elemente zu bestimmen. Ein Konstruktor instanziiert ein `ObjectCollectionViewModel`-Objekt mit einer leeren `ObservableCollection`. Dies wird dafür benutzt, um einen leeren Sammelordner darzustellen. Die zwei weiteren Konstruktor nehmen auch vorgefertigte Sammlungen entgegen, die die `ObservableCollection` gleich mit untergeordneten Elementen befüllt.

5.3.4. Erstellen der GenXElement-ViewModels

Die Erstellung der Datenmodelldarstellung findet hauptsächlich in der `setFileStream`-Methode des `MainViewModel` und im Konstruktor der `GenXStructureViewModel`-Klasse statt. In der `setFileStream`-Methode wird ein Dateipfad einer Genx-Datei erwartet. Diese Datei wird mit einem `StreamReader`-Objekt eingelesen. Der Inhalt der Datei wird dem `GenXParser`-Objekt übergeben, der daraus ein `GenXStructure`-Objekt mit den, im Datenmodell enthaltenen, Objekten erstellt. Dieses `GenXStructure`-Objekt wird als Parameter dem Konstruktor der `GenXStructureViewModel`-Klasse übergeben.

Im `GenXStructureViewModel`-Konstruktor wird die die Datenmodellstruktur der Reihe nach aufgebaut. Zuerst werden die Themen erstellt, anschließend die Feature-Klassen und Ansichten, danach die Domänen gefolgt von den Tabellen, logische Topologien und zuletzt die Sequenzen

Erstellung der Themen

Der Aufbau der Datenmodellstruktur beginnt mit der Erstellung der Themen. Dies wird in der `buildTopicTree`-Methode des `GenXStructureViewModel`-Objekts gemacht.

Initialisierung des Hilfsobjekte Zu Beginn wird ein `SortedDictionary`, mit einem `int`-Schlüssel und einem `List`-Wert, erstellt. Diese wird dazu benutzt, um während der Erstellung der `TopicViewModel`-Objekte, die Tiefe jener Objekte festzuhalten.

Erstellung der TopicViewModel Die Erstellung der `TopicViewModel`-Objekte findet in einer For-Each-Schleife statt, die über die Liste der `Topic`-Objekte iteriert, die in `Topics` im `GenXStructure`-Objekt gespeichert sind. Diese `TopicViewModel`-Objekte werden dann im `SortedDictionary` einer Liste mit dem Schlüssel des Tiefenwerts hinzugefügt. Ein Thema, das ganz oben in der Themenhierarchie steht, wird im `SortedDictionary` bei der Liste mit dem Schlüssel „0“ gespeichert. Ein Thema mit 2 Eltern-Themen wird in der Liste mit dem Schlüssel „2“ gespeichert.

Listing 5.9: Quelltextausschnitt der `TopicViewModel`-Erstellung

```

1 foreach (Topic topic in _genXStructure.Topics.Values)
2 {
3     TopicViewModel topicViewModel =
4         new TopicViewModel(topic, this);
5
6     int layerIndex = topicViewModel.ParentTopicList.Count;
7     List<TopicViewModel>? result;
8     if (!layerTopicDict.TryGetValue(layerIndex, out result))
9     {
10         // Erstellung einer Liste, falls diese nicht existiert
11         layerTopicDict.Add(layerIndex, new List<TopicViewModel>());
12     }
13     layerTopicDict[layerIndex].Add(topicViewModel);
14 }
```

Korrektes Anordnen der TopicViewModel Um die hierarchische Struktur der Themen-Objekte korrekt aufzubauen wird über das `SortedDictionary`-Objekt iteriert. Durch

die Nutzung eines `SortedDictionary` wird, bei 0 anfangend aufzählend iteriert. Dadurch wird der Reihe nach eine komplette Themenebene erstellt, bevor die Kind-Themen eingefügt werden. Dies sorgt dafür, dass sichergestellt wird, dass ein Kind-Thema einem Eltern-Thema hinzugefügt werden kann. Es werden also zuerst alle `TopicViewModel` in die Struktur eingefügt, die keine Eltern-Themen haben, danach wird die nächste Ebene eingefügt mit den `TopicViewModel`-Objekten, die genau ein Eltern-Thema haben, und so weiter, bis die tiefste Ebene in die Struktur eingefügt wurde.

Listing 5.10: Quelltextausschnitt der `TopicViewModel` Anordnung

```

1 foreach (KeyValuePair<int, List<TopicViewModel>> kvp
2         in layerTopicDict)
3 {
4     foreach (TopicViewModel topicViewModel in kvp.Value)
5     {
6         if (kvp.Key == 0)
7         {
8             _collectionTopics.AddChild(topicViewModel);
9         }
10        else
11        {
12            TopicViewModel? parentViewModel;
13            if (findTopicViewModelByPath(
14                topicViewModel.ParentTopicList,
15                out parentViewModel) &&
16                parentViewModel is not null)
17            {
18                parentViewModel.AddChild(topicViewModel);
19            }
20        }
21    }
22 }
```

TopicViewModel Suchmethode Um während des Aufbaus der Struktur die korrekte Stelle zum Einfügen zu finden, wird die `findTopicViewModelByPath`-Methode verwendet, die die `ParentTopicList` eines `Topic`-Objektes annimmt und einen booleschen Wert und das eventuell gefundene Eltern-`TopicViewModel` zurückgibt.

Die `findTopicViewModelByPath` Methode sucht Ebene für Ebene rekursiv mithilfe der `ParentTopicList` nach dem `TopicViewModel` dessen Name dem letzten Element der `ParentTopicList` gleicht und dessen Position mit der `ParentTopicList` übereinstimmt. Dies geschieht, indem mit jedem gefundenem `TopicViewModel` das erste Element der `ParentTopicList` übersprungen wird und die restliche `ParentTopicList` beim nächsten Such-Aufruf weitergegeben wird.

Wurde am Ende der Suche das richtige `TopicViewModel` gefunden, wird von der `findTopicViewModelByPath`-Methode `True` zurückgegeben. Das gefundene `TopicViewModel` wird über den `out`-Parameter zurückgegeben.

Zur Veranschaulichung wird ein Beispiel-`TopicViewModel` betrachtet. Diese hat die `ParentTopicList` ("Land", "Bundesland", "Stadt"). Bei der ersten Suche wird in der obersten Ebene nach einem `TopicViewModel` mit dem Namen `Land` gesucht, wenn diese gefunden wurde, wird die Suche in dem `Land-TopicViewModel` mit der Liste ("Bundesland", "Stadt") weitergeführt. Wenn das `Bundesland-`

`TopicViewModel` gefunden wurde, wird die Suche mit der Liste ("Stadt") weitergeführt. Wenn das `Stadt-TopicViewModel` gefunden wurde bricht die Rekursion ab, weil die `ParentTopicList` nun leer ist. Das `Stadt-TopicViewModel` wird mit `return` die Rekursionsebene hochgereicht und das Eltern-Thema wurde gefunden.

Erstellung der Feature-Klassen und Ansichten

Die ViewModel für die Feature-Klassen und Ansichten werden gleichzeitig erstellt, weil diese zwei Objektarten Kindobjekt von sich selbst und einander sein können. Die Erstellung und die Anordnung der `FeatureClassViewModel`- und `ViewViewModel`-Objekte folgen dem gleichen Prinzip wie die Erstellung der Themen-ViewModels

Generalisierung der Feature-Klasse und der Ansicht Damit sich beide ViewModel-Klassen bei der Anordnung gleich zu einander verhalten, wird eine abstrakte Klasse definiert, von der die zwei ViewModel-Klassen erben. Die abstrakte Klasse `TopicChild` speichert den Typen der Unterklasse ab, damit die `FeatureClassViewModel`- und `ViewViewModel`-Instanzen im `TopicChild` unterscheidbar sind. Diese Klasse bietet zwei Methoden, die `FindTopicChild`-Methode mit der ein untergeordnetes `TopicChild`, anhand seines Namens und Typs, gefunden werden kann und die `AddFeatureClassOrViewChild`-Methode die ein weiteres `TopicChild`-Objekt als untergeordnetes Objekt hinzufügen kann.

Initialisierung der Hilfsobjekte Ähnlich zur Erstellung der `TopicViewModel` wird hier ebenfalls ein `SortedDictionary` benutzt, um die Objekte der Ebenentiefe nach zu sortieren. Hier ist die Ebenentiefe jedoch nicht die gesamte Tiefe vom Datenmodell-Ursprung, sondern der Abstand vom nächsthöchsten Thema, oder die Anzahl der übergeordneten Feature-Klassen oder Ansichten

Erstellung der `FeatureClassViewModel`- und `ViewViewModel`-Objekte In zwei getrennten For-Each-Schleifen wird über die `FeatureClasses` und die `Views` der `GenXStructure` iteriert und die passenden ViewModels erstellt. Diese `FeatureClassViewModel`- und `ViewViewModel`-Objekte werden, anhand ihrer Ebenentiefe, den Listen der `SortedDictionary` hinzugefügt.

Anordnung der `FeatureClassViewModel`- und `ViewViewModel`-Objekte Die Anordnung der `FeatureClassViewModel`- und `ViewViewModel`-Objekte verläuft ebenfalls ähnlich zur Anordnung der `TopicViewModel`-Elemente. Es wird bei 0 anfangend über `SortedDictionary` iteriert,

`TopicChild` Suchmethode Anders als bei der Suche der Einfügestelle für die `TopicViewModel`-Elemente, besteht dieser Schritt bei den `TopicChild`-Klassen aus zwei Schritten. Zuerst wird über die `TopicList` das passende `TopicViewModel` gesucht, anschließend wird mit der `ParentObjectsNamesTypesDict` das passende `TopicChild` gesucht, um bei diesem ein weiteres Objekt unterzuordnen.

Der Grund für die Nutzung eines Dictionaries, welches den Objektnamen und den Typen des Objektes speichert, ist Spezifizierung des Typs eines Objektes. Würden die Namen der Eltern-Objekte in einer Liste gespeichert werden, wie die Namen der Themen, steht es am Ende offen, ob es sich bei einem bestimmten Objekt um eine Feature-Klasse oder einer Ansicht handelt. Mit der Angabe des Typs kann im ViewModel genauer nach einem bestimmten ViewModel-Objekt gesucht werden.

Die Suche nach dem passenden **TopicChild**-Eltern-Objekt verläuft in der **FindParentViewModel**-Methode. Diese Methode nimmt ein **TopicChild**-Objekt, dessen Eltern-Objekt gefunden werden soll, an und gibt einen booleschen Wert und das eventuell gefundene **TopicChild**-Eltern-Objekt zurück. Zuerst wird mit der **findTopicViewModelByPath**-Methode das nächsthöchste **TopicViewModel**-Objekt. Auf dem gefundenen Objekt wird einmal die Methode **SearchChild** aufgerufen, die anhand eines Namens und Objekt-Typs ein bestimmtes untergeordnetes Objekt sucht. In diesem Fall wird nach dem obersten **TopicChild** gesucht, dem das neue **TopicChild** zugeordnet werden soll.

Wenn das oberste **TopicChild** gefunden wurde, wird mit einer For-Each-Schleife über die **ParentObjectNamesTypesDict** iteriert, wobei das erste Schlüssel-Wert-Paar übersprungen wird, weil im vorherigen Schritt schon nach diesem gesucht wurde. In der Schleife wird die **FindTopicChild**-Methode aufgerufen, die ein untergeordnetes **TopicChild**-Objekt mit einem bestimmten Namen und Typen sucht.

Am Ende wird das zuletzt gefundene **TopicChild**-Objekt zurückgegeben. Ist dieses das direkte Eltern-Objekt des einzufügenden **TopicChild**-Objektes, so gibt die **FindParentViewModel**-Methode **True** zurück.

Erstellung der Domänen, Tabellen, logischen Topologien und Sequenzen

Die ViewModel-Erstellung der Domänen, Tabellen, logischen Topologien und Sequenzen verläuft sehr ähnlich zueinander. Es wird mit einer **Select**-Methode über die Elemente der jeweiligen Sammlung des **GenXStructure**-Objektes iteriert. Diese Elemente werden mit einem Lambda-Ausdruck in die jeweiligen ViewModel-Klassen umgewandelt und mit **ToList** in einer weiteren Liste zusammengefasst. Diese wird in einen **ObjectCollectionViewModel**-Konstruktor weitergegeben, damit die Sammelstruktur mit Element befüllt wird.

5.3.5. Anzeigen der Objekte in der Datenmodellldarstellung

Um die Darstellung der ViewModel-Klassen zu vereinheitlichen, wird mit einer Vererbung gearbeitet. Die Basisklasse, von der alle angeglichenen ViewModel-Klassen erben, ist die **TreeNodeBase**-Klasse. Diese abstrakte Klasse definiert die nötigen Elemente, damit ein Objekt korrekt in der Datenmodellldarstellung angezeigt wird.

Listing 5.11: Verkürzter Quelltext der TreeViewNodeBase-Klasse mit darstellungsrelevanten Properties

```

1 public abstract class TreeViewNodeBase
2 {
3     // Konstruktor ausgelassen
4
5     public abstract string Name { get; }
6     public abstract string NodeCaption { get; }
7     public abstract IDictionary<string, string> TableData
8         { get; }
9
10    private bool _isSelected;
11    public bool IsSelected
12    {
13        // Get und Set ausgelassen
14    }
15
16    public ObservableCollection<MenuItemViewModel> MenuItems
17        { get; } = new ObservableCollection<MenuItemViewModel>();
18
19    // EventHandler ausgelassen
20 }

```

Die **Name**-Property gibt den direkten Namen des Model-Objektes zurück. Die **NodeCaption**-Property gibt die Beschriftung eines Knotens zurück, so wie er in der hierarchischen Struktur erscheinen soll. In den meisten Fällen ist dies eine Kombination aus dem Namen und der Caption eines Model-Objektes. Die **IsSelected**-Property ist für die Bestimmung des ausgewählten Objektes wichtig und wird in Abschnitt 5.3.8 näher behandelt. Die **MenuItems**-Property ist vom Typ **ObservableCollection**. Dies sind Sammlungen, die, wenn sie an eine View gebunden sind, automatisch aktualisieren, wenn sich die Sammlung verändert. Dies geschieht, zum Beispiel, wenn der Sammlung ein Element hinzugefügt wird oder ein Element entfernt wird. Die **MenuItems**-Property enthält Elemente die, für die Erstellung des Kontextmenüs des Knotens wichtig sind.

5.3.6. Erstellung des Kontextmenüs eines Knotens

Die Erstellung des Kontextmenüs eines Knotens verläuft über das ViewModel **MenuItemViewModel**, welches in der **ObservableCollection** **MenuItems** gespeichert wird. Das ViewModel besteht aus zwei Properties, einem **string**-Property **Header** und einem **ICommand**-Property **Command**. Das **Header**-Property bestimmt den Text, der in einem Menüpunkt angezeigt wird. Die **Command**-Property bestimmt den Command der ausgeführt werden soll, wenn der Menüpunkt aktiviert wird.

Listing 5.12: Quelltext der MenuItemViewModel-Klasse

```

1 public class MenuItemViewModel
2 {
3     public string Header { get; set; }
4     public ICommand Command { get; set; }
5
6     public MenuItemViewModel(string header, ICommand command)
7     {
8         Header = header;
9         Command = command;
10    }
11 }

```

Die `MenuItemViewModel`-Instanzen werden im Konstruktor eines `GenXElement-ViewModels` erstellt und der `ObservableCollection MenuItem`s mit der Methode `add` hinzugefügt. Diese `MenuItem`s-Sammlung wird als `ItemsSource` des `Context-Menu-Elements` gebunden. Mit dem `ItemContainerStyle`-Property wird bestimmt wie sich der Inhalt des `ContextMenu-Elements` verhält. In diesem Fall wird mit `<Style TargetType="MenuItem">` bestimmt, dass `MenuItem`-Elemente bearbeitet werden sollen. Mit dem `<Setter>`-Element wird der `Header` des `MenuItem`-Elements mit dem `Header` der `MenuItemViewModel`-Instanz, aus der `ItemsSource`, gebunden. Analog dazu der `Command` des `MenuItem`-Elements mit dem `Command` der `MenuItem-ViewModel`-Instanz. Dies geschieht für alle Elemente in `MenuItems`.

Listing 5.13: Ausschnitt der Xaml-Definition für die Kontextmenü-Erstellung eines Knotens

```

1 <Setter Property="ContextMenu">
2     <Setter.Value>
3         <ContextMenu ItemsSource="{Binding MenuItems}">
4             <ContextMenu.ItemContainerStyle>
5                 <Style TargetType="MenuItem">
6                     <Setter Property="Header"
7                         Value="{Binding Header}"/>
8                     <Setter Property="Command"
9                         Value="{Binding Command}"/>
10                </Style>
11            </ContextMenu.ItemContainerStyle>
12        </ContextMenu>
13    </Setter.Value>
14 </Setter>

```

5.3.7. Commands

Die `Commands` des `Struktureditor` erlauben das Interagieren mit dem Programm. Diese `Commands` implementieren das `ICommand`-Interface. Um dies bei mehreren unterschiedlichen `Command`-Klassen zu vereinheitlichen und Code-Dopplungen zu vermeiden, wird eine abstrakte Basisklasse `CommandBase` erstellt, von der die spezifischen `Command`-Klassen erben. Die Basisklasse definiert die `CanExecute`-Methode, die `OnCanExecute-Changed`-Methode, das `CanExecuteChanged`-Event und die abstrakte Methode `Execute`. Die `CanExecute`-Methode bestimmt, ob ein `Command` ausführbar ist. In diesen

Fällen gibt die Methode immer **True** zurück, heißt der Command kann ausgeführt werden. Die **CanExecute**-Methode wird als eine virtuelle Methode gekennzeichnet, für den Fall, dass es in späteren Weiterentwicklung des Struktureditors eine Methode hinzugefügt wird, die aktiviert oder deaktiviert werden kann. Falls sich dies jedoch ändern sollte, sollte die **OnCanExecuteChanged**-Methode ausgeführt werden. Dies sorgt dafür, dass Methoden ausgeführt werden, die das Event **CanExecuteChanged** beobachten.

Listing 5.14: Quelltext der abstrakten Klasse CommandBase

```

1 public abstract class CommandBase : ICommand
2 {
3     public virtual bool CanExecute(object? parameter)
4     {
5         return true;
6     }
7
8     public abstract void Execute(object? parameter);
9
10    public event EventHandler? CanExecuteChanged;
11    protected void OnCanExecuteChanged()
12    {
13        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
14    }
15 }
```

Implementierung einer Command-Klasse

Zur Verdeutlichung der Funktionsweise einer Command-Klasse wird die Klasse **FileOpenCommand** betrachtet. Dieser Command ist zuständig für die Auswahl einer Genx-Datei und das Setzen der Datei in das Haupt-ViewModel. Die **FileOpenCommand**-Klasse enthält eine Referenz auf das Haupt-ViewModel, die beim Instanzieren übergeben und gesetzt wird. In der **Execute**-Methode wird ein **OpenFileDialog**-Objekt erstellt. Die **OpenFileDialog**-Klasse ist eine vom Windows-Betriebssystem bereitgestellte Klasse, die den Datei-Explorer öffnet, damit der Benutzer eine Datei auswählen kann. Diesem Dialog kann ein Filter zugewiesen werden, damit im Datei-Explorer nur Dateien angezeigt werden, die die relevante Dateiendung besitzen. In diesem Fall werden ein Filter für **.genx**-Dateien und ein Filter für alle Dateien ***.*** gesetzt. Mit der **ShowDialog**-Methode des **OpenFileDialog**-Objekts wird der Datei-Explorer geöffnet, damit der Nutzer eine Datei auswählen kann. Die Methode **ShowDialog** gibt einen Nullable-Boolean zurück, hat der Nutzer den Dialog mit „OK“ beendet wird **True** zurückgegeben, andernfalls wird **False** zurückgegeben. Wenn die **ShowDialog**-Methode **True** zurückgibt, wird der Dateipfad der ausgewählten Datei an die **setFileStream**-Methode des Haupt-ViewModels weitergegeben. Die Funktionsweise des HauptViewModels wird in Abschnitt 5.3.1 näher behandelt.

Listing 5.15: Quelltext der Klasse FileOpenCommand

```

1 internal class FileOpenCommand(MainViewModel viewModel)
2     : CommandBase
3 {
4     private readonly MainViewModel _viewModel = viewModel;
5
6     public override void Execute(object? parameter)
```

```

7  {
8      OpenFileDialog dialog = new OpenFileDialog();
9      dialog.Filter = "GENX Files|*.genx|All Files|*.*";
10
11     bool? result = dialog.ShowDialog();
12     if (result == true)
13     {
14         _viewModel.setFileStream(dialog.FileName);
15     }
16 }
17 }

```

Instanziierung und DataBinding eines Commands

Damit ein Command-Klasse an die View gebunden werden kann, muss diese Instanziiert werden. Dies geschieht in der `FileOpenCommand`-Property. Hier wird bei einem `Get`-Aufruf eine neue Instanz der `FileOpenCommand`-Klasse erstellt und zurückgegeben. Diese Property wird in der View mit `Command="Binding FileOpenCommand"` gebunden. Dies geschieht für das `FileOpenCommand`-Property an zwei Stellen, der Xaml-Definition für das `<KeyBinding>`-Element mit der Tastenkombination `Strg+O` und der Definition des `<MenuItem>`-Elements `Öffnen`.

Listing 5.16: Quelltextausschnitt der `MainViewModel`-Klasse, Definition der `FileOpenCommand`-Property

```

1  // weiterer Quelltext der MainViewModel-Klasse ausgelassen
2  public ICommand FileOpenCommand
3  {
4      get
5      {
6          return new FileOpenCommand(this);
7      }
8  }
9  // weiterer Quelltext der MainViewModel-Klasse ausgelassen

```

Listing 5.17: Xaml-Definition des `Öffnen`-MenuItems

```

1  <MenuItem Header="Öffnen" InputGestureText="Strg+O"
2      Command="{Binding FileOpenCommand}" />

```

Listing 5.18: Xaml-Definition des `Öffnen`-MenuItems

```

1  <KeyBinding Key="O" Modifiers="Ctrl"
2      Command="{Binding FileOpenCommand}" />

```

5.3.8. Auswahl eines Knotens

Die Auswahl eines Knotens wird über einen Service gesteuert, der mit allen `GenXElementViewModels` verbunden ist. Die Auswahl eines Knotens erfolgt über eine Kombination aus Data Bindings und Events. Durch das Data Binding der `IsSelected`-Property im `TwoWay`-Modus ist das Setzen der gebundenen Property von der View möglich.

Listing 5.19: Xaml-Definition des `IsSelected` Data Bindings

```

1 <Setter Property="IsSelected"
2   Value="{Binding IsSelected, Mode=TwoWay}"/>

```

Diese Property wird mit der `IsSelected`-Property aus der `TreeViewNodeBase`-Klasse gebunden. Um das Setzen der `IsSelected`-Property besser zu steuern, basiert die öffentliche Property auf einem privaten Backing Field `_isSelected`.

Die Property mit einem Backing Field ist eine häufig erscheinende Struktur, die erhöhte Kontrolle über das Lesen und Setzen der Property bietet. Das Backing Field ist ein privates Feld mit dem gleichen Typen wie das Property, das den eigentlichen Zustand der Property speichert. Die öffentliche Property kann im `get` und `set` steuern, wie mit dem Backing Field interagiert werden soll.

In diesem Fall erweitert die Property das `set`-Verhalten. Im `set` wird zuerst geprüft, ob der sich der neue Wert vom alten Wert unterscheidet, wenn dies nicht der Fall ist wird das `set` mit `return` unterbrochen. Unterscheiden sich die Werte wird der neue Wert dem Backing Field zugewiesen und es wird ein Event ausgelöst, dass sich die `IsSelected`-Property geändert hat. Wenn der neue Wert `True` ist, der Knoten also ausgewählt wurde, wird das ausgewählte Objekt der `SelectedNode`-Property des `SelectionService` zugewiesen.

Listing 5.20: Auszug aus `TreeViewNodeBase`: die Property-Definition von `IsSelected`

```

1 private bool _isSelected;
2 public bool IsSelected
3 {
4     get => _isSelected;
5     set
6     {
7         if (_isSelected == value) return;
8         _isSelected = value;
9         OnPropertyChanged();
10        if (value)
11        {
12            SelectionService.SelectedNode = this;
13        }
14    }
15 }

```

In der `TreeNodeSelectionService`-Klasse wird ebenfalls eine Property mit Backing Field definiert. Die Property ist `SelectedNode` vom Typ `TreeViewNodeBase`. In dieser Property wird das `GenXElement-ViewModel`-Objekt gespeichert, das in der Datenmodellldarstellung ausgewählt wurde. Beim Setzen der Property wird zuerst geprüft, ob das schon ausgewählte Objekt und das neu ausgewählte Objekt gleich sind. Ist dies der Fall wird das `set` mit einem `return` beendet. Sind die Objekte unterschiedlich wird das neue Objekt dem Backing Field `_selectedNode` zugewiesen. Danach werden die zwei Events `SelectedNodeChanged` und `PropertyChanged` aufgerufen, damit sich andere Klassen, die die `TreeNodeSelectionService`-Instanz beobachten, aktualisieren gegebenenfalls aktualisieren können.

Listing 5.21: Quelltext der `TreeNodeSelectionService`-Klasse

```

1 public class TreeNodeSelectionService : INotifyPropertyChanged
2 {
3     private TreeViewNodeBase _selectedNode;
4     public TreeViewNodeBase SelectedNode
5     {
6         get => _selectedNode;
7         set
8         {
9             if (_selectedNode == value) return;
10            _selectedNode = value;
11
12            SelectedNodeChanged?.Invoke(this, value);
13            PropertyChanged?.Invoke(this,
14                new PropertyChangedEventArgs(nameof(SelectedNode)));
15        }
16    }
17
18    public event EventHandler<TreeViewNodeBase>?
19        SelectedNodeChanged;
20    public event PropertyChangedEventHandler? PropertyChanged;
21 }

```

5.3.9. Anzeige der Eigenschaften eines Knotens

Das Anzeigen der Eigenschaften eines Knotens wird im `MainViewModel` definiert. Die Knoteneigenschaften werden in der Property `DisplayedNodeData` gespeichert. Diese ist eine `ObservableCollection`, die Elemente des Typs `TableRowViewModel` enthält.

Die `TableRowViewModel`-Klasse stellt einen Datensatz dar, der in dem `<DataGrid>` des Eigenschaftenbereichs angezeigt wird. Die Klasse besteht aus zwei Properties `Key` und `Value`. Diese werden in den `<DataGridTextColumn>`-Elementen gebunden (Siehe Abschnitt 5.1.2 *Eigenschaftenbereich* auf Seite 30).

Die Aktualisierung der `DisplayedNodeData`-Collection übernimmt die `UpdateDisplayedNodeData`-Methode, die bei der Erstellung des `MainViewModel` als Event Handler dem `SelectedNodeChanged`-Event zugewiesen wird. Es wird zuerst überprüft, ob die `TableData`-Property des ausgewählten Knoten Elemente enthält. Ist diese leer geschieht nichts weiteres, enthält diese jedoch Elemente, so wird die `DisplayedNodeData`-Collection mit `Clear()` entleert und mit einer Iteration über `TableData` mit neuen `TableRowViewModel`-Objekten befüllt.

Listing 5.22: Quelltext der UpdateDisplayedNodeData-Methode

```

1 private void UpdateDisplayedNodeData(object sender,
2                                     TreeNodeBase newNode)
3 {
4     if (SelectedNode.TableData.Count > 0)
5     {
6         DisplayedNodeData.Clear();
7
8         foreach (KeyValuePair<string, string> kvp
9                 in SelectedNode.TableData)
10        {
11            DisplayedNodeData.Add(
12                new TableRowViewModel(kvp.Key, kvp.Value)
13            );
14        }
15    }
16 }

```

TableData ist eine Property die in **TreeNodeBase** als ein **IDictionary** definiert wird, das einen **string**-Schlüssel und einen **string**-Wert abspeichert. In den Unterklassen, die Eigenschaften haben, die im Eigenschaftenbereich angezeigt werden sollen, wird ein Backing Field **_tableData** als **Dictionary** definiert und mit der Methode **Add** befüllt. Der Schlüssel des hinzugefügten Datensatzes ist der Name der Eigenschaft und der Wert des Datensatzes ist der Wert der Eigenschaft.

Listing 5.23: TableData-Erstellung in der DomainTableViewModel-Klasse

```

1 _tableData = new Dictionary<string, string>();
2 _tableData.Add("Caption", _domainTable.Caption is null ? "" :
3                 _domainTable.Caption.ToString());
4 _tableData.Add("DomainEntries", "");
5 _tableData.Add("Name", _domainTable.Name);
6 _tableData.Add("Operation", _domainTable.Operation.ToString());
7 _tableData.Add("Owner", "");
8 updateTableData();

```

Für das Aktualisieren der **_tableData** gibt es die **updateTableData**-Methode, die die Datensätze die sich verändern können aktualisiert. In der **DomainTableViewModel**-Klasse kann sich der **DomainEntries** Datensatz verändern, in dem ein neues **DomainEntryViewModel** hinzugefügt wird. Der Wert des Datensatzes ist ein zusammengesetzter String, der die Namen der untergeordneten **DomainEntryViewModels** aufzählt und durch Kommata trennt. Bei einer Aktualisierung wird der String neu gebildet, heißt das hinzugefügte Objekt wird auch aufgezählt.

5.3.10. Datenmodellobjekte erstellen

Das Erstellen eines neuen Datenmodellobjektes verläuft über Commands. Für jedes Datenmodellobjekt wird ein Command definiert, das für die Erstellung der Datenmodellobjektes zuständig ist. Dies erlaubt eine präzisere Erstellung der neuen Objekte. Diese Commands sollen im folgenden als „Create-Command“ bezeichnet werden. Die Create-Commands enthalten private Felder, die das Objekt speichern, zu dem das neue Objekt hinzugefügt werden soll.

Veranschaulichung anhand der **CreateTopicCommand**-Klasse

Die **CreateTopicCommand**-Klasse ist zuständig für das Erstellen eines Themenobjekts. Im Struktureditor gibt es zwei Stellen, an denen ein Themenobjekt erstellt werden kann. Es kann im Kontextmenü eines anderen Themas erstellt werden oder im Kontextmenü des Sammelknotens „Themen“. Um diese zwei Stellen in der **CreateTopicCommand**-Klasse korrekt zu behandeln, werden zwei private Felder definiert, die mit den jeweiligen Objekt der jeweiligen Stelle befüllt werden.

Die **CreateTopicCommand**-Instanz des obersten Sammelknotens wird mit der Referenz auf das **ObjectCollectionViewModel**-Objekt befüllt. Eine **CreateTopicCommand**-Instanz eines **TopicViewModel**-Objektes wird mit der Referenz auf jenes **TopicViewModel**-Objekt erstellt. Diese Unterscheidung ist wichtig, um in der **Execute**-Methode diese zwei Stellen unterschiedlich zu behandeln.

Execute-Methode In der **Execute**-Methode wird ein neues **Topic**-Objekt erstellt, diesem wird ein Name mit einer zufälligen Ziffer und die **Add**-Operation gegeben. Basierend auf diesem Objekt wird ein neues **TopicViewModel**-Objekt erstellt, um das **Topic**-Objekt visuell darzustellen.

Im nächsten Schritt wird unterschieden an welcher Stelle das neue Objekt eingefügt werden soll. Wird das neue Objekt über ein anderes **TopicViewModel**-Objekt erstellt, wird die **ParentTopicList** des neuen Objekts gesetzt und das neue **TopicViewModel** wird dem übergeordneten **TopicViewModel**-Objekt mit **AddChild** zu gewiesen. Wird das neue Objekt jedoch über den obersten „Themen“-Sammelknoten erstellt, bedeutet dies, dass das neue Objekt keine übergeordneten Themen hat. Die **ParentTopicList** bleibt leer und das neue Objekt wird direkt in die **ObjectCollectionViewModel**-Instanz eingefügt.

Instanziierung der **Create-Command**-Objekte

Die Instanziierung eines **Create-Command**-Objekts geschieht im Konstruktor der **GenX-Element-ViewModel**-Klasse. Diese **Command**-Instanzen werden anschließend in einem **MenuItemViewModel** der **MenuItems**-Sammlung hinzugefügt, die in **TreeNodeBase** definiert ist.

Listing 5.24: Auszug der **TopicViewModel**-Konstruktors: Erstellung einer **CreateTopicCommand**-Instanz

```

1 CreateTopicCommand createTopicCommand =
2   new CreateTopicCommand(this, _genXStructureViewModel);
3 _topicCollection.MenuItems.Add(
4   new MenuItemViewModel("Thema erstellen", createTopicCommand)
5 );
6 MenuItems.Add(
7   new MenuItemViewModel("Thema erstellen", createTopicCommand)
8 );
```

Da **TopicViewModel** selbst einen Themen-Sammelknoten besitzt, wird der **Command**, zur Erstellung eines weiteren Themas, auch den **MenuItems** des **ObjectCollectionViewModel** hinzugefügt. In der **AddChild**-Methode wird dann das neu erstellte **TopicViewModel**-Objekt dem **ObjectCollectionViewModel** hinzugefügt.

Aktualisierung der Datenmodelldarstellung

Die Aktualisierung der Datenmodelldarstellung passiert automatisch, in dem das **PropertyChanged**-Event mit der **Children**-Property ausgelöst wird. Diese Änderung wird vom View erkannt und diese kann sich aktualisieren

Listing 5.25: Quelltext der CreateTopicCommand-Klasse

```

1 public class CreateTopicCommand : CommandBase {
2     private TopicViewModel _topicViewModel;
3     private ObjectCollectionViewModel _objectCollectionViewModel;
4     private GenXStructureViewModel _genXStructureViewModel;
5
6     public CreateTopicCommand(
7         TopicViewModel topicViewModel,
8         GenXStructureViewModel genXStructureViewModel )
9     {
10         _topicViewModel = topicViewModel;
11         _genXStructureViewModel = genXStructureViewModel;
12     }
13     public CreateTopicCommand(
14         ObjectCollectionViewModel objectCollectionViewModel,
15         GenXStructureViewModel genXStructureViewModel )
16     {
17         _objectCollectionViewModel = objectCollectionViewModel;
18         _genXStructureViewModel = genXStructureViewModel;
19     }
20
21     public override void Execute(object? parameter)
22     {
23         Random rng = new Random();
24         Topic newTopic = new Topic(
25             "TOPIC_" + rng.Next(100000, 1000000),
26             Operation.Add
27         );
28
29         newTopic.ParentTopicList = new List<string>();
30         TopicViewModel newTopicViewModel =
31             new TopicViewModel(newTopic, _genXStructureViewModel);
32
33         if(_topicViewModel is not null) {
34             List<string> parentTopicList =
35                 new List<string>(_topicViewModel.ParentTopicList);
36             parentTopicList.Add(newTopic.Name);
37             newTopic.ParentTopicList = parentTopicList;
38             _topicViewModel.AddChild(newTopicViewModel);
39         } else {
40             _objectCollectionViewModel.AddChild(newTopicViewModel);
41         }
42         _genXStructureViewModel.Add(newTopic);
43     }
44 }

```

6. Ergebnis

6.1. Überblick der Funktionen des Struktureditors

Der im Rahmen dieser Arbeit entwickelte Struktureditor-Prototyp ist ein erster Ansatz für einen unternehmenseigenen Struktureditor. Der Fokus lag hierbei auf der Nachbildung der Grundfunktionen des bestehenden Struktureditors, wie das Öffnen einer Genx-Datei, die visuelle Darstellung des Datenmodells und dessen Eigenschaften sowie das erstellen neuer Objekte. Weitere Grundfunktionen, wie das Speichern und Erstellen eines Datenmodells, das Bearbeiten der Eigenschaften oder das Löschen bestehender Objekte, konnten aufgrund des begrenzten zeitlichen Umfangs nicht umgesetzt werden.

Funktionskategorie	Anforderung	Umsetzungsgrad
Dateiverarbeitung	FA-1 Genx-Datei öffnen	vollständig
Dateiverarbeitung	FA-2 Genx-Datei speichern	nicht umgesetzt
Dateiverarbeitung	FA-3 neues Datenmodell erstellen	nicht umgesetzt
Datenmodell-Bearbeitung	FA-4 neue Objekte erstellen	teilweise umgesetzt
Datenmodell-Bearbeitung	FA-5 bestehende Objekte löschen	nicht umgesetzt
Datenmodell-Bearbeitung	FA-6 Eigenschaften bearbeiten	nicht umgesetzt
Datenmodell-Bearbeitung	FA-7 neues Strukturupdate erstellen	nicht umgesetzt
Darstellung	FA-8 Datenmodell als Baum-Struktur anzeigen	teilweise umgesetzt
Darstellung	FA-9 Objekt-Eigenschaften anzeigen	vollständig
Darstellung	FA-10 Änderungen anzeigen	teilweise umgesetzt

Tabelle 6.1.: Übersicht der Anforderungen und deren Umsetzungsstatus

6.2. Dateiverarbeitung

6.2.1. Öffnen einer Genx-Datei (FA-1)

Diese Anforderung wird vollständig vom Struktureditor umgesetzt. Um eine Datei auszuwählen, kann die Auswahl über den Menüpunkt „Öffnen“ oder alternativ der Tastenkombination **Strg+O** gestartet werden. Dies öffnet ein Windows-Dialog-Fenster, das es erlaubt eine Genx-Datei auszuwählen. Wurde eine Datei ausgewählt und der Knopf „Öffnen“ gedrückt wird diese Datei eingelesen und auf der Benutzeroberfläche wird das Datenmodell, dieser Datei angezeigt.

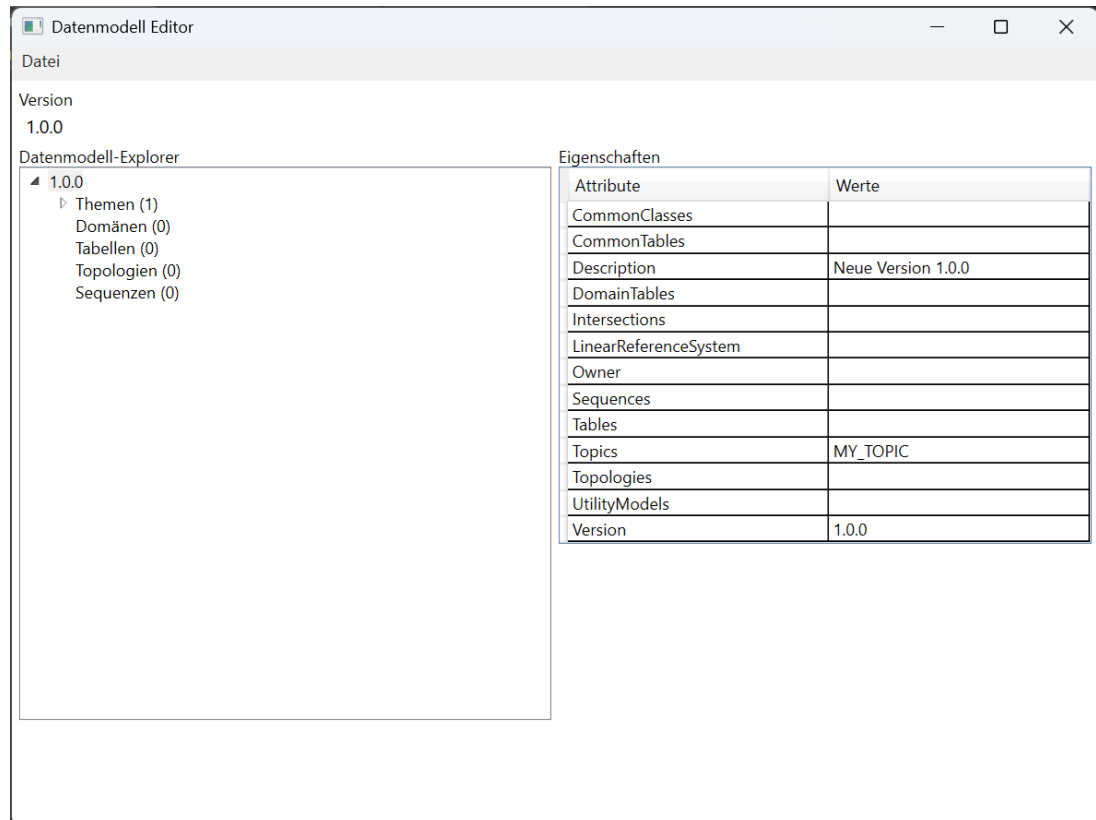


Abbildung 6.1.: Struktureditor nach dem Öffnen einer Genx-Datei

6.2.2. Speichern einer Genx-Datei (FA-2)

Die Anforderung FA-2 das Speichern einer Genx-Datei wurde im Rahmen dieser Arbeit nicht umgesetzt.

6.2.3. Erstellen eines neuen Datenmodells (FA-3)

Die Anforderung FA-3 das Erstellen eines neuen Datenmodells wurde im Rahmen dieser Arbeit nicht umgesetzt.

6.3. Datenmodell-Bearbeitung

6.3.1. Erstellen neuer Datenmodellobjekte (FA-4)

Diese Anforderung wurde teilweise umgesetzt. Diese Funktion wurde umgesetzt für Themen, Feature-Klassen, Ansichten, Attribute und Indexe. Um dem Datenmodell eines dieser Objekte hinzuzufügen, muss das Kontextmenü des Knotens, dem das neue Objekt untergeordnet sein soll, geöffnet werden. Im Kontextmenü werden die gültigen Optionen aufgezählt. Wird eine Erstellungsoption ausgewählt wird ein neues Objekt mit einem Namen mit zufälliger Zahl und Standardwerten erstellt und an die korrekte Stelle eingefügt. In Abbildung 6.2 wird das Kontextmenü des Themen-Sammelknotens gezeigt. Da dieser Knoten nur Themen-Objekte als gültige Unterobjekte erlaubt, wird nur die Erstellungsoption eines Themas angezeigt. In Abbildung 6.3 wird wiederum das Kontextmenü eines Thema-Knotens gezeigt. Im Kontextmenü werden drei Erstellungsoptionen angeboten, weil Themen nur weitere Themen, Feature-Klassen und Ansichten als gültige Unterobjekte erlauben.

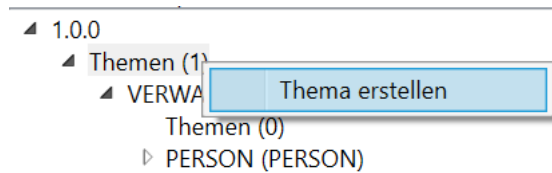


Abbildung 6.2.: Kontextmenü des Themen-Sammelknotens

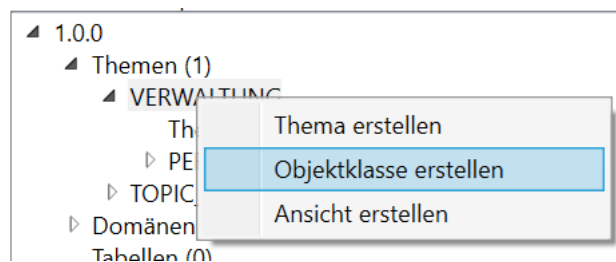


Abbildung 6.3.: Kontextmenü eines Thema-Knotens

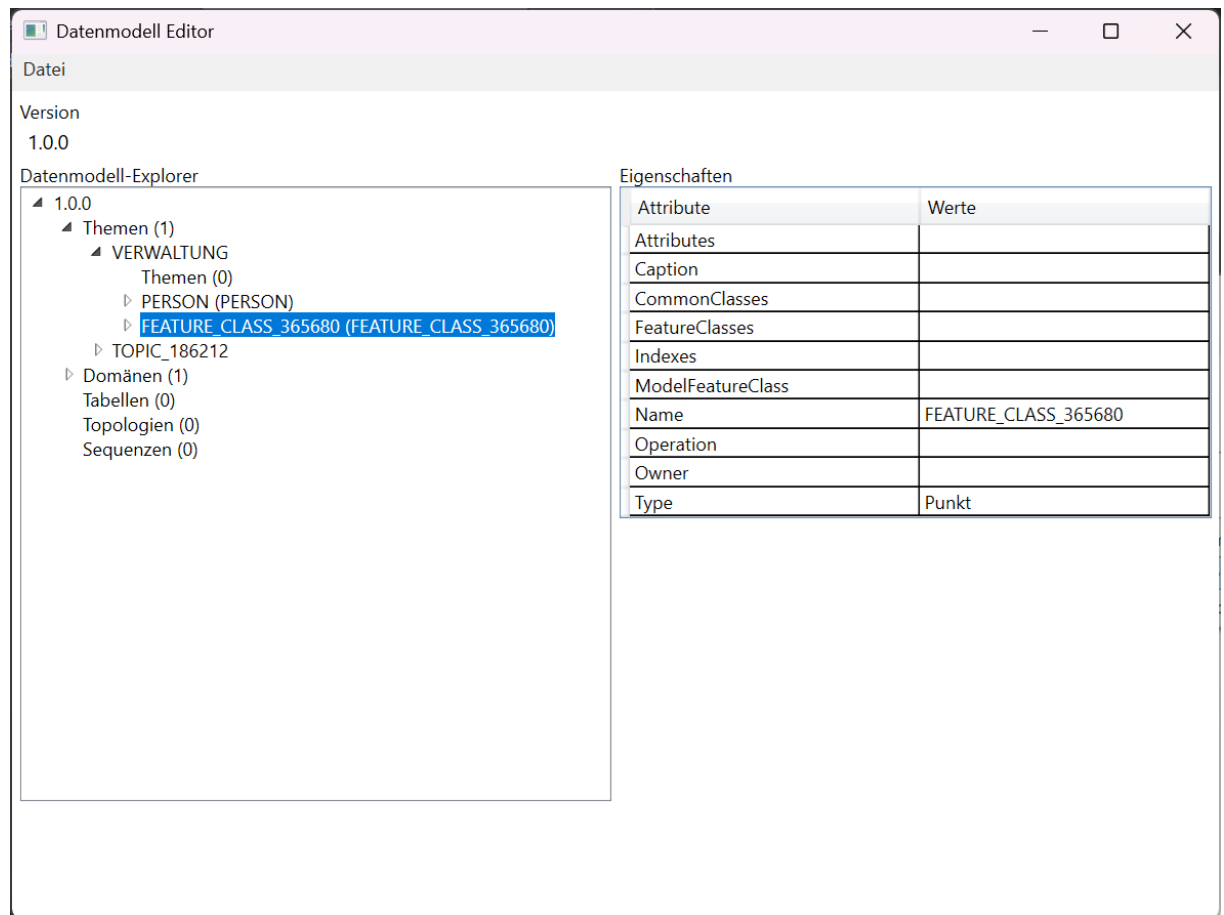


Abbildung 6.4.: Struktureditor nach dem erstellen eines neuen Objekts

6.3.2. Löschen bestehender Objekte (FA-5)

Die Anforderung FA-5 das Löschen bestehender Datenmodellobjekte wurde im Rahmen dieser Arbeit nicht umgesetzt.

6.3.3. Bearbeiten von Objekteigenschaften (FA-6)

Die Anforderung FA-6 das Bearbeiten der Objekteigenschaften wurde im Rahmen dieser Arbeit nicht umgesetzt.

6.3.4. Erstellen eines neuen Strukturupdate (FA-7)

Die Anforderung FA-7 das Erstellen eines neuen Strukturupdate wurde im Rahmen dieser Arbeit nicht umgesetzt.

6.4. Darstellung

6.4.1. Datenmodell als hierarchische Struktur (FA-8)

Die Anforderung FA-8, dass ein Datenmodell in einer hierarchischen Struktur angezeigt werden soll, wurde im Rahmen dieser Arbeit teilweise umgesetzt. Die Darstellung in einer hierarchischen Struktur gilt für alle Datenmodellobjekte die der GenXParser abbilden kann. Nicht abgebildet werden Allgemeine Klassen, Allgemeine Tabellen, Flächentopologien, Verschnitte und Lineare Referenzsysteme.

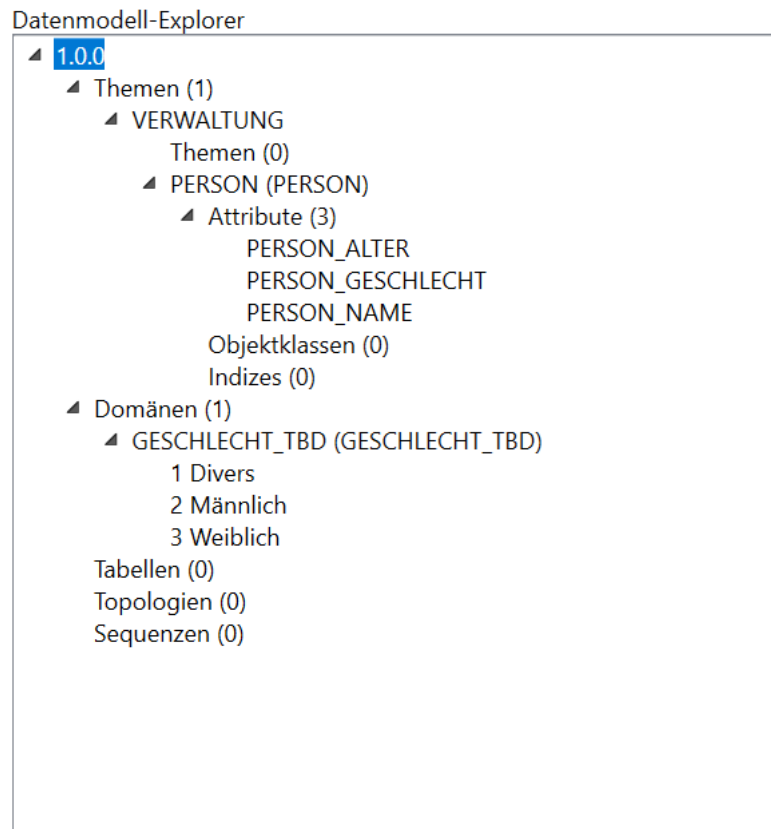


Abbildung 6.5.: Darstellung eines Datenmodells in hierarchischer Struktur

6.4.2. Objekteigenschaften anzeigen (FA-9)

Die Anforderung FA-9, dass die Eigenschaften eines ausgewählten Objektes angezeigt werden sollen, wurde vollständig umgesetzt. Wenn ein Knoten in der Datenmodelldarstellung ausgewählt wird und dieser Eigenschaften enthält, werden diese Eigenschaften im Eigenschaftenbereich in tabellarischer Form angezeigt. Diese Tabelle ändert sich und wird neu befüllt, wenn ein neuer Knoten ausgewählt wird.

Datenmodell-Explorer	Eigenschaften																						
1.0.0 - Themen (1) - VERWALTUNG - Themen (0) PERSON (PERSON) - Attribute (3) - PERSON_ALTER - PERSON_GESCHLECHT - PERSON_NAME - Objektklassen (0) - Indizes (0) - Domänen (1) - GESCHLECHT_TBD (GESCHLECHT_TBD)	<table> <tr> <th>Attribute</th><th>Werte</th></tr> <tr> <td>Attributes</td><td>PERSON_ALTER; PERSON_GESCHLECHT</td></tr> <tr> <td>Caption</td><td></td></tr> <tr> <td>CommonClasses</td><td></td></tr> <tr> <td>FeatureClasses</td><td></td></tr> <tr> <td>Indexes</td><td></td></tr> <tr> <td>ModelFeatureClass</td><td></td></tr> <tr> <td>Name</td><td>PERSON</td></tr> <tr> <td>Operation</td><td></td></tr> <tr> <td>Owner</td><td></td></tr> <tr> <td>Type</td><td>Punkt</td></tr> </table>	Attribute	Werte	Attributes	PERSON_ALTER; PERSON_GESCHLECHT	Caption		CommonClasses		FeatureClasses		Indexes		ModelFeatureClass		Name	PERSON	Operation		Owner		Type	Punkt
Attribute	Werte																						
Attributes	PERSON_ALTER; PERSON_GESCHLECHT																						
Caption																							
CommonClasses																							
FeatureClasses																							
Indexes																							
ModelFeatureClass																							
Name	PERSON																						
Operation																							
Owner																							
Type	Punkt																						

Abbildung 6.6.: Die Eigenschaften eines ausgewählten Knotens (Person Feature-Klasse) werden im Eigenschaftenbereich angezeigt

6.4.3. Anzeigen einer Änderung (FA-10)

Die Anforderung FA-10, dass eine Änderung in der Benutzeroberfläche angezeigt werden soll, wurde im Rahmen dieser Arbeit teilweise umgesetzt. Die Änderung, dass ein neues Objekt in das Datenmodell eingefügt wird, wird angezeigt, in dem das neue Objekt nach dem Einfügen ein Teil der hierarchischen Struktur wird. Diese Änderung wird auf der Benutzeroberfläche durch einen neuen Knoten angezeigt (Siehe Abbildung 6.4 auf Seite 50). Weitere Änderungsoptionen, wie das Löschen eines Knotens, wurden im Rahmen dieser Arbeit nicht umgesetzt und können deswegen nicht angezeigt werden.

6.5. Zusammenfassung des entwickelten Struktureditors

Der entwickelte Struktureditor-Prototyp erlaubt es dem Benutzer eine Genx-Datei zu öffnen, sich den Aufbau des Datenmodells anzuschauen und die Eigenschaften der Datenmodellobjekte anzeigen zu lassen. Funktionen, wie das Speichern einer Genx-Datei oder das Verändern der Objekteigenschaften, wurden nicht umgesetzt.

7. Fazit

7.1. Was wurde erreicht?

Das Ziel dieser Arbeit war die Konzeption und Entwicklung eines Struktureditor-Prototyps, der Genx-Dateien Laden, Visualisieren, Bearbeiten und Speichern kann.

Im Rahmen dieser Arbeit ist ein Programm-Prototyp entstanden, der eine Genx-Datei einlesen und visuell darstellen kann. Es können die Eigenschaften der Datenmodellobjekte eingesehen werden und es können eine Auswahl neuer Datenmodellobjekte erstellt werden.

7.2. Herausforderungen

Während des Prozesses der Konzeption und Entwicklung des Prototyps kamen häufig unerwartete Probleme auf.

7.2.1. Unzureichende Analyse

Eine unzureichende Analyse des bestehenden Struktureditors und Datenmodell sorgte dafür, dass während der Entwicklung häufig auf ein falsches Verständnis aufgebaut wurde, das im Nachhinein beseitigt werden musste. Beispielsweise entstand anfänglich das Verständnis, dass Feature-Klassen nur Attribute und untergeordnete Feature-Klassen enthalten können. Während der Entwicklung kam jedoch auf, dass eine Feature-Klasse auch Ansichten enthalten kann und Ansichten wiederum weitere Feature-Klassen. Dieses Missverständnis sorgte für eine komplette neu Entwicklung des Algorithmus der diese Objekte erstellt und in die hierarchische Struktur einfügt.

7.2.2. Komplexes Model

Die Komplexität der Funktionsweise und die Vollständigkeit des Models wurden bei ersten Analyse nur oberflächlich verstanden. Dies führte bei der Entwicklung des ViewModels zu reichlich Fehlern und nachträglichen Anpassungen des Models.

A. Model-View-ViewModel-Beispiel

Listing A.1: Model-Klasse: Person

```
1 public class Person
2 {
3     private int _age;
4     private string _name;
5
6     public Person(string name, int age)
7     {
8         _name = name;
9         _age = age;
10    }
11
12    public void IncreaseAge()
13    {
14        _age += 1;
15    }
16
17    public string Name { get => _name; }
18    public int Age { get => _age; }
19 }
```

Listing A.2: View: Xaml-Definition der Benutzeroberfläche

```
1 <Window x:Class="Easiest_Example_Program.View.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3     presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/blend
6     /2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8     compatibility/2006"
9     xmlns:local="clr-namespace:Easiest_Example_Program"
10    xmlns:vm="clr-namespace:Easiest_Example_Program.ViewModel"
11    mc:Ignorable="d"
12    Title="MainWindow" Height="200" Width="200">
13    <StackPanel Orientation="Vertical" >
14        <TextBlock Text="{Binding PersonDataString}"/>
15        <Button Command="{Binding IncreaseAgeCommand}"
16            Content="Alter erhöhen"/>
17    </StackPanel>
18 </Window>
```

Listing A.3: View: CodeBehind von MainWindow.xaml

```
1 public partial class MainWindow : Window
2 {
3     public MainWindow()
4     {
5         DataContext = new MainViewModel();
6         InitializeComponent();
7     }
8 }
```

Listing A.4: ViewModel-Klasse: MainViewModel

```
1 public class MainViewModel : INotifyPropertyChanged
2 {
3     private Person _myPerson;
4     public event PropertyChangedEventHandler? PropertyChanged;
5
6     public MainViewModel()
7     {
8         _myPerson = new Person("Max Mustermann", 25);
9     }
10
11     public void IncreaseAge()
12     {
13         _myPerson.IncreaseAge();
14         OnPropertyChanged(nameof(PersonDataString));
15     }
16
17     public string PersonDataString
18     {
19         get
20         {
21             string personData = "Name: ";
22             personData += _myPerson.Name;
23             personData += "\nAlter: ";
24             personData += _myPerson.Age;
25             return personData;
26         }
27     }
28
29     public ICommand IncreaseAgeCommand
30     {
31         get => new IncreaseAgeCommand(this);
32     }
33
34     private void OnPropertyChanged(string propertyName)
35     {
36         PropertyChanged?.Invoke(
37             this,
38             new PropertyChangedEventArgs(propertyName)
39         );
40     }
41 }
```

Listing A.5: Commands-Klasse: IncreaseAgeCommand

```
1 public class IncreaseAgeCommand : ICommand
2 {
3     private MainViewModel _mainViewModel;
4     public event EventHandler? CanExecuteChanged;
5
6     public IncreaseAgeCommand(MainViewModel mainViewModel)
7     {
8         _mainViewModel = mainViewModel;
9     }
10
11     public bool CanExecute(object? parameter)
12     {
13         return true;
14     }
15
16     public void Execute(object? parameter)
17     {
18         _mainViewModel.IncreaseAge();
19     }
20 }
```

Literatur

- [1] Zen Admin. *AutoCAD MAP 3D - What is meant by a feature?* 2025. URL: <https://ukcommunity.arkance.world/hc/en-us/articles/21549519453074-AutoCAD-MAP-3D-What-is-meant-by-a-feature> (besucht am 05.02.2026).
- [2] Autodesk. *Feature Data Objects (FDO) Developer's Guide*. 2008. URL: https://download.osgeo.org/fdo/3.4.0/docs/FDG_FD0DevGuide.pdf (besucht am 12.01.2026).
- [3] Autodesk. *Was ist AutoCAD?* URL: <https://www.autodesk.com/de/products/autocad/overview> (besucht am 05.02.2026).
- [4] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. 2005. URL: <https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps> (besucht am 26.01.2026).
- [5] Microsoft. *Access Modifiers - C#*. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers> (besucht am 02.02.2026).
- [6] Microsoft. *GridUnitType Enum*. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.windows.gridunittype> (besucht am 19.01.2026).



Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage.



Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe.



Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden.



Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet.



Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel“ mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt.



Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt.



Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.



Chemnitz, 5.2.2026
Ort, Datum

Long Nguyen
Unterschrift