Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Robert Madlo

„Out-of-the-Box“ Entwicklungsplattform für Referenzarchitekturen am Beispiel Quasar in Visual Studio

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Robert Madlo

„Out-of-the-Box“ Entwicklungsplattform für Referenzarchitekturen am Beispiel Quasar in Visual Studio

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 19. Dezember 2011

Robert Madlo

Thema der Arbeit

„Out-of-the-Box“ Entwicklungsplattform für Referenzarchitekturen am Beispiel Quasar in Visual Studio

Stichworte

Referenzarchitektur, Quasar, Visual Studio, UML-Designer, Visual Studio Erweiterung, Synchronisierung UML-Diagramm und Code

Kurzzusammenfassung

In großen und komplexen Softwareprojekten ist es notwendig, Entwicklungsumgebungen zu schaffen, die den Entwickler aktiv bei seiner Arbeit unterstützen, bestimmte Prozesse zu automatisieren und von Programmen oder Add-ins zu überwachen. Der Softwareentwickler soll bei seiner Arbeit aktiv von der Entwicklungsumgebung unterstützt werden. Sich wiederholende Arbeitsgänge sollen automatisiert und vom System automatisiert durchgeführt werden. Diese Arbeit befasst sich mit der Entwicklung einer Erweiterung für Visual Studio 2010, die einen Codegenerator für Referenzarchitekturen, hier im speziellen Quasar, auf Basis von T4 (Text Template Transformation Toolkit) bereitstellt.

Robert Madlo

Title of the paper

"‘Out-of-the-Box"’ development platform for reference architectures in example for Quasar in Visual Studio

Keywords

reference architecture, Quasar, Visual Studio, UML-Designer, Visual Studio extension, synchronization UML-diagram and code

Abstract

In large and complex software projects, it is necessary to create development environments which support the developer in its work, to automate certain processes and monitor them by programs or add-ins. The software developer should be actively supported by the development environment in his work. Repetitive operations should be automated and performed by the system. This work deals with the development of an extension for Visual Studio 2010, which provides a code generator for reference architectures, in particular quasar, based on T4 (Text Template Transformation Toolkit).

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Problemstellung

In großen und komplexen Softwareprojekten ist es notwendig, Entwicklungsumgebungen zu schaffen, die den Entwickler aktiv bei seiner Arbeit unterstützen, bestimmte Prozesse zu automatisieren und von Programmen oder Add-ins zu überwachen. Es ist notwendig, das Denken auf Klassenebene hinter sich zu lassen. Um die Komplexität von Problemen dieses Ausmaßes zu analysieren und zu bewerkstelligen, ist es nötig, auf eine höhere Abstraktionsebene zu wechseln und Klassen zu komplexeren Gebilden, im Folgenden Komponenten genannt, zusammenzufassen.

In vielen Softwareprojekten werden sogenannte Referenzarchitekturen genutzt, die entweder firmeninterne Standards definieren oder sich an gängigen Designkonzepten orientieren. In Abschnitt 2.2 „Referenzarchitekturen“ werden die Vorteile von Referenzarchitekturen genauer beschrieben.

An der HAW-Hamburg wird in vielen Projekten und Kursen die Entwicklungsumgebung Visual Studio eingesetzt. Im Rahmen der studentischen Ausbildung soll auch hier die Komplexität von Softwareprojekten erkannt und gemeistert werden. Eine oft eingesetzte Referenzarchitektur ist die Quasar-Standardarchitektur, die in Abschnitt 2.2.1 „Quasar: Qualitäts-Software-Architektur“ vorgestellt wird.

Mit der aktuellen Version von Microsoft Visual Studio 2010 ist es leider nicht möglich, Softwareentwicklung nach den Gesichtspunkten und Richtlinien einer Referenzarchitektur auszulegen, da die Entwicklungsumgebung diese Funktionen nicht zur Verfügung stellt. Hier zeigt Visual Studio gravierende Schwächen.

Die beiden Komponenten von Visual Studio, die davon hauptsächlich betroffen sind, sind zum einen, der UML-Designer und zum anderen der primitive Codegenerator, der in der Basisversion noch gar nicht vorhanden ist. Visual Studio bietet von Haus aus keine Möglichkeit, aus einem UML-Diagramm Code zu erzeugen. Erst mit dem Visual Studio 2010 Feature 2 Add-on und diversen Updates wurde eine sehr eingeschränkte Möglichkeit geschaffen, Codeskeletons aus UML-Diagrammen zu erzeugen.

Ein großes Manko des vorhandenen UML-Designers ist es, dass der Komponenten-Editor keine Verbindung zum vorhandenen Klasseneditor besitzt. Genau das wäre aber nötig, wenn man einzelne Komponenten mit ihrer Innensicht verknüpfen will, um sie anschließend auch sinnvoll zu füllen und zu bearbeiten. Das würde Arbeitszeit sparen und ein hohes Maß an Fehlertoleranz gewähren. Namenskonventionen könnten schon bei der Komponentenerstellung geprüft werden. Eine automatische Generierung des Innenlebens einer Komponente würde dem Designer viel Arbeit ersparen.

Ein weiteres Problem, das zurzeit besteht, ist, dass die vorhandene automatische Codeerzeugung nur auf Klassenbasis basiert. Folgende Codestrukturen können vom Codegenerator umgesetzt werden.

- Klassen
- Interfaces
- Enumerations
- Delegates
- Structs

Eine Zusammenfassung von einzelnen Klassen zu Komponenten ist nicht vorgesehen. Somit ist es mit Visual Studio zurzeit nicht möglich, einen erstellten Entwurf, der nach einer gängigen Referenzarchitektur, z.B. Quasar, entwickelt worden ist, direkt in Codeskeletons umzuwandeln. Die manuelle Umsetzung eines solchen Entwurfes bringt für den Entwickler sehr viele Nachteile mit sich und nur sehr wenige Vorteile. Im Folgenden werden einige Nachteile der manuellen Code-Generierung aufgelistet.

- keine Prüfung von Abhängigkeiten
- keine Prüfung von Zyklen
- keine Prüfung von Namenskonventionen (z.B. nach vorher definierten Richtlinien)
- Namensänderungen geschehen an vielen verschiedenen Orten
- bei der Übertragung können Rechtschreibfehler entstehen
- sehr hoher Arbeitsaufwand (Workload), um die Skeletons und Methodenrümpfe zu erzeugen
- kleine Änderungen der Architektur bewirken große Änderungen im Source-Code
- bei Änderungen im Source-Code muss Konsistenz manuell mit dem UML-Modell hergestellt werden

Als Alternative ist es natürlich möglich, 3rd Party Tools zu verwenden. Aber auch diese Möglichkeit hat Nachteile. Diese Tools stehen beispielsweise nicht direkt unter der Kontrolle eines TFS und der integrierten Quellcodeverwaltung. Man würde aber gern das Ziel verfolgen, eine einzige Entwicklungsplattform zu nutzen und diese um die gewünschten Funktionen zu erweitern. Der Einsatz von externen Werkzeugen außerhalb der eigentlichen Entwicklungsplattform sollte im besten Falle ganz vermieden werden.

Genau in diesem Punkt liegt aber auch die große Stärke von Visual Studio. Durch gut dokumentierte Schnittstellen zu allen internen Funktionen ist es möglich, den UML-Designer anzupassen, die internen Metamodelle der Diagramme abzufragen, zu analysieren und diese in Codeskeletons umzusetzen.

1.2. Motivation

Um nun die Problematik genauer darstellen zu können, wird das folgende einfache Beispiel (Abb. 1.1) verwendet, das im weiteren Verlauf dieser Arbeit immer wieder aufgegriffen wird.

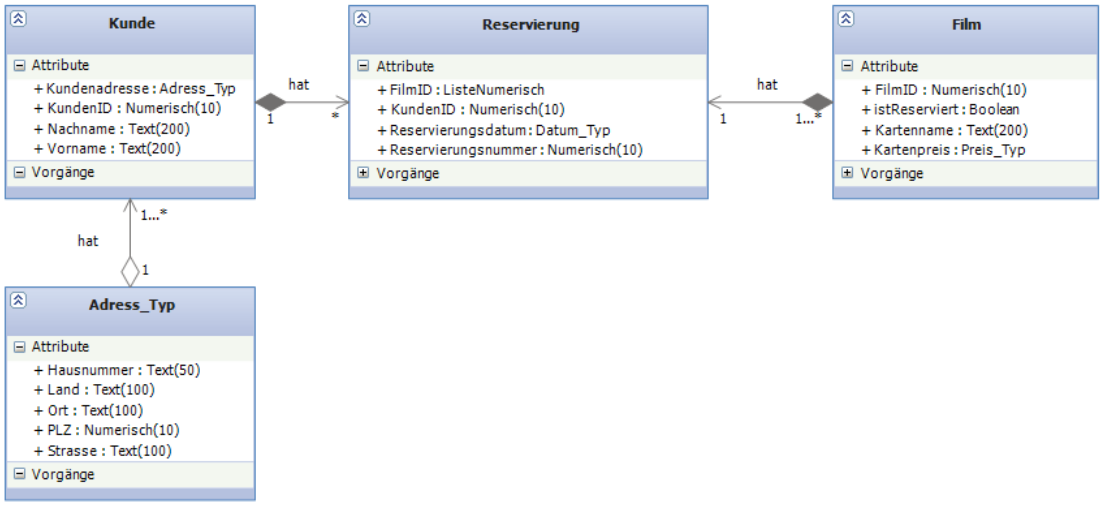


Abbildung 1.1.: Fachliches Datenmodell: Filmreservierung

Das Modell beschreibt ein kleines System mit 4 fachlichen Typen, den Kunden mit ihren Adressdaten und den Filmen als Bestandsdaten, sowie die Reservierung, die von den Kunden getätigt wurden.

In Abschnitt 2.3 „Begleitendes Beispiel“ wird dieses Beispiel noch weiter beschrieben und genauer auf die Problematik der Softwareentwicklung eingegangen.

1.3. Ziele der Arbeit

Diese Arbeit befasst sich mit der Entwicklung einer Erweiterung für Visual Studio 2010, die einen Codegenerator für Referenzarchitekturen, hier im speziellen Quasar, auf Basis von T4 (Text Template Transformation Toolkit) bereitstellt. Es wird eine Erweiterung des UML-Designers vorgenommen, um eine schnelle und effektive Bearbeitung und Erstellung von UML-Diagrammen zu gewährleisten und eine einfache Synchronisierung zwischen UML-Diagramm und Code bereitzustellen.

1.4. Aufbau der Arbeit

a. Analyse (Kapitel 2): In der Analysephase werden zunächst die verschiedenen Schritte des Softwareentwicklungszyklus betrachtet und gezeigt, wo die Schwächen in Zusammenhang mit Visual Studio und dem Umsetzen des Designs nach Gesichtspunkten einer Referenzarchitektur liegen. Danach werden die Vorteile einer solchen Referenzarchitektur erörtert und die Schwächen einer manuellen Umsetzung mit Hilfe eines Beispiels (Abschnitt 2.3 „Begleitendes Beispiel“ - Filmverwaltung) vertieft. Es wird analysiert, welche Anwendungsfälle bei der Entwicklung auftreten können und wie das Add-in den Entwickler unterstützen kann.

b. Entwurf (Kapitel 3): Im Entwurf werden zunächst die Vor- und Nachteile verschiedener Templatemodelle erörtert. Danach werden verschiedene Teilsysteme, wie der Validator oder Code-Generator, genauer untersucht. Anschließend wird der Synchronisationsrahmen für die UML- und Code-Elemente festgelegt. Abschließend wird die fachliche Architektur des Gesamtsystems dargestellt.

c. Implementation (Kapitel 4): Im Abschnitt Implementierung wird auf die Probleme bei der Erstellung dieser Arbeit eingegangen. Es werden beispielhaft einige Implementierungsdetails erläutert.

d. Test & Integration (Kapitel 5): Im Abschnitt Test und Integration werden Testbedingungen, für die im Rahmen dieser Arbeit erstellte Erweiterung, beschrieben. Es wird eine Auswertung manueller Usertests durchgeführt und analysiert. Danach wird genauer auf die Installation der Erweiterung eingegangen.

e. Fazit (Kapitel 6): Abschließend wird auf Probleme bei der Erstellung dieser Arbeit eingegangen. Was kann die hier erstellte Erweiterung, was kann sie nicht? Außerdem

wird ein Ausblick auf mögliche Weiterentwicklungen im Rahmen anderer Projekte gegeben.

2. Analyse

Zunächst muss klar sein, an welchen Stellen der Entwickler unterstützt werden soll und kann. Welche Ziele sollen mit der Erweiterung erreicht werden? Welche Grenzen sind von der Plattform Visual Studio gegeben, die man nicht überwinden kann?

Ausgangspunkt ist die genauere Betrachtung des Softwareentwicklungsprozesses und der Referenzarchitektur Quasar. Dabei werden einige Anwendungsfälle, die bei der Erstellung von Komponenten und Klassen in Visual Studio immer wieder auftreten analysiert, um daraus die Funktionen abzuleiten, die die neue Erweiterung bieten muss. Es wird eine Schwachstellenanalyse durchgeführt, um zu zeigen, wie mit Hilfe der im Rahmen dieser Arbeit erstellten Visual Studio Erweiterungen diese verbessert oder ganz beseitigt werden können.

2.1. Softwareentwicklungsprozess

Der eigentliche Prozess der Softwareentwicklung besteht aus 5 elementaren Teilschritten, egal welches Prozessmodell man zugrunde legt, Wasserfallmodell, V-Modell oder Spiralmodell. Diese 5 Phasen finden sich in jedem dieser Modelle wieder.

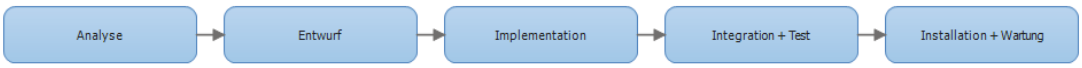


Abbildung 2.1.: Softwareentwicklungsprozess

2.1.1. Analysephase

In der Analysephase wird eine Systemspezifikation erstellt. Das heißt, alle relevanten Anforderungen an das System werden ermittelt und dokumentiert. Kern der Spezifikation sind die funktionalen und nicht-funktionalen Anforderungen an das System sowie eine Skizze des Gesamtsystementwurfs. In der Analysephase werden die verschiedenen Anwendungsfälle ermittelt, analysiert und modelliert. Ein fachliches Datenmodell wird erarbeitet und Geschäftsprozesse werden erstellt.

2.1.2. Entwurfsphase

Die Entwurfsphase dient dazu, die fachliche Architektur des Softwareprojekts festzulegen, Systemoperationen für die verschiedenen Anwendungsfälle und Geschäftsprozesse zu erarbeiten, Komponenten und ihre Schnittstellen zu definieren und deren Innensicht festzulegen. Des Weiteren wird bei Bedarf ein relationales Datenbankmodell erstellt.

2.1.3. Implementationsphase

In dieser Phase wird die entworfene Architektur und die definierten Komponenten in Code übertragen.

2.1.4. Integrations- und Testphase

In dieser Phase werden die erstellten Komponenten des Systems zu einem Gesamtsystem zusammengeführt und getestet.

2.1.5. Installationsphase + Wartung

In der letzten Phase des Prozesses wird das erstellte System dem Kunden übergeben und installiert. Mit Abschluss dieser Phase ist der Softwareentwicklungsprozess abgeschlossen. Was folgt ist die ständige Wartung des Systems.

2.2. Referenzarchitekturen

Eine Referenzarchitektur ist keine generalisierte und eindeutige Lösung für ein bestimmtes Problem im Softwaredesign. Sie ist vielmehr eine grobe Anleitung, wie bestimmte Problemstellungen effektiv gelöst werden können. Sie verbindet allgemeine Architektur-Richtlinien und Konzepte mit den spezifischen Anforderungen an eine bestimmte Softwarelösung zu einem Gesamtkonzept.

Systemstrukturen werden dokumentiert und die wesentlichen Bausteine grob skizziert. Das Zusammenspiel zwischen den einzelnen Systemkomponenten wird beschrieben. Eine Referenzarchitektur besteht also aus einem Referenzmodell und verschiedenen Architekturmitteln. Das heißt aber auch, für verschiedene Probleme gibt es unterschiedliche Referenzarchitekturen.

Der Einsatz einer Referenzarchitektur bringt unter anderem folgende Vorteile:

- Risikominimierung: Durch den Einsatz einer Referenzarchitektur kann z.B. das Risiko minimiert werden, eine nicht tragfähige oder nicht leistungsfähige Architektur selbst zu entwerfen.

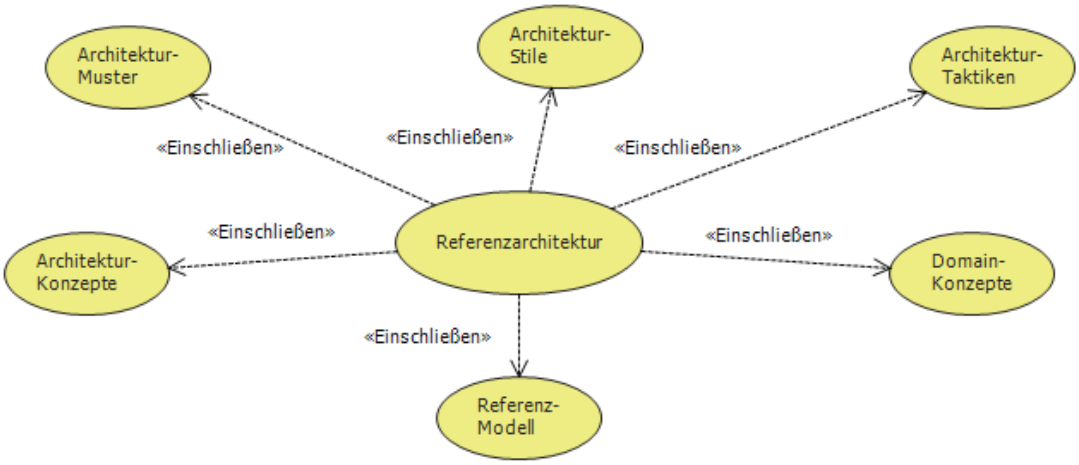


Abbildung 2.2.: Konzept einer Referenzarchitektur (Vogel u. a., 2008, S. 254)

- Kostenreduzierung: Die allgemeinen Kosten für den Architekturentwurf können gesenkt werden, da auf einem bestehenden Entwurf aufbaut wird und dieser an die individuelle Problemstellung anpasst wird.
- Qualitätssteigerung: Die Nutzung einer Referenzarchitektur führt zu einer gesteigerten Qualität des Produktes, da man auf bewährte Mittel, Stile und Designelemente in der Architektur vertraut, die in einer Referenzarchitektur vereinigt sind.
- Zeitreduzierung: Die Designphase wird erheblich verkürzt, damit auch die Kosten und die Gesamtentwicklungszeit. Eine schnelle Markteinführung ist damit möglich.
- Erfahrungsvorsprung: Gängige Referenzarchitekturen unterstehen ständig den kritischen Kontrollen verschiedenster Entwickler weltweit. Somit werden Schwächen schnell erkannt und korrigiert. Referenzarchitekturen können sich im Laufe der Zeit verändern und verbessern. Damit steigt ihre Qualität.

2.2.1. Quasar: Qualitäts-Software-Architektur

In der vorliegenden Arbeit wird als exemplarisches Beispiel die Qualitäts-Software-Architektur (Quasar) der Firma capgemini verwendet. Quasar ist dabei nicht als eine komplette Lösung für den Entwurf einer Software zu verstehen. Sie vereint vielmehr eine Reihe von Design- und Coding Richtlinien, um die Qualität der Software zu steigern.

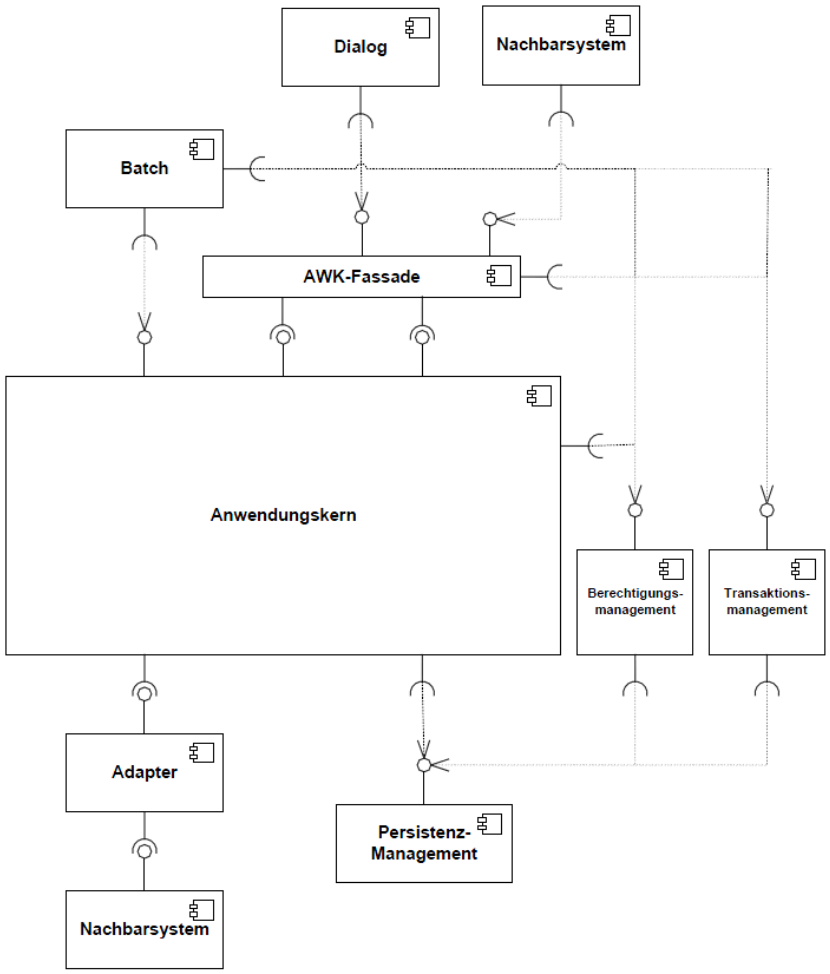


Abbildung 2.3.: Die Quasar-Standardarchitektur

Eines der wesentlichen Grundprinzipien einer Quasar-konformen Anwendungsarchitektur sind die Trennung von Fachlichkeit (A-Software) und Technik (T-Software) und die Bildung von Komponenten, dass heißt, ähnliche Funktionalität wird zusammengefasst.

Technische Dienste der Architektur, z.B. Persistenz, Berechtigungsmanagement, GUI oder Adapter, können durch verschiedene Komponenten implementiert werden.

Im vorliegenden Design werden 3 verschiedene Architekturen betrachtet: die A-,T- und TI-Architektur. Die A-Architektur beschreibt die Architektur der Anwendung. Das heißt, die anwendungsspezifischen Bestandteile des Systems, deren Schnittstellen und Beziehungen (z.B. Kunde, Film, Ausleihe) werden beschrieben. Die Bestandteile werden aus der Spezifikation ermittelt. Sie abstrahiert von technischen Details und von 0-Software. Ihr Aufbau und das

Design ergeben sich aus den Designaspekten von Quasar. In der Quasar-Architektur enthält der Anwendungskern die A-Komponenten.

Die T-Architektur beschreibt die technischen Komponenten eines Systems und wie diese mit dem Anwendungskern interagieren. Hierbei bilden die T-Komponenten eine Art „virtuelle Maschine“ zum Ablauf der A-Komponenten. Exemplarisch sind hier Transaktionen, Persistenz, GUI und Verteilung zu nennen.

Die TI-Architektur beschreibt die technische Infrastruktur. Sie wird in großen Teilen vom Kunden vorgegeben und vor Beginn des Entwicklungsprozessen festgelegt.

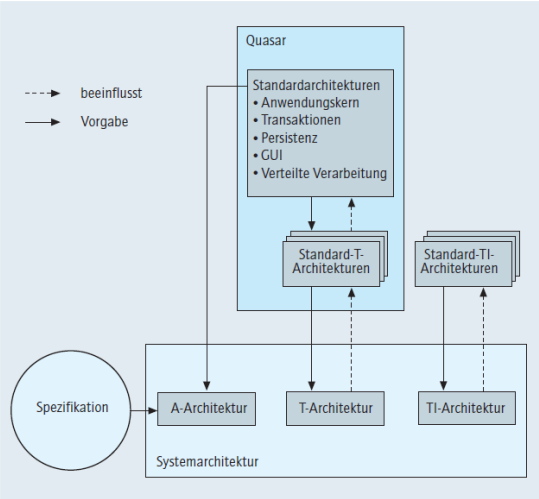


Abbildung 2.4.: Zusammenspiel von Quasar mit verschiedenen Architekturen

Leider ist eine wirklich formalisierte und systematische Umsetzung dieser Konzepte mit dem an der HAW Hamburg eingesetzten Entwicklungstool Visual Studio 2010 nicht möglich. Für die Umsetzung in Softwareprojekten wird eine effizient nutzbare Plattform benötigt.

Zum einen gilt es also, die Lücke zwischen Architekturverständnis und einer formalisierten Umsetzung der Architekturprinzipien zu schließen, zum anderen eine einheitliche „out-of-the-box“-Umgebung für Softwareprojekte bereitzustellen.

Die im Rahmen dieser Arbeit erstellte Erweiterung für Visual Studio versucht nun, einige dieser Lücken zu schließen und somit den Entwicklungszyklus für ein Softwareprojekt zu vereinfachen.

2.3. Begleitendes Beispiel

Aus unserer Spezifikation des Systems, die wie in Abschnitt 2.1.1 „Analysephase“ erstellt wurde, kann folgendes fachliche Datenmodell extrahiert werden.

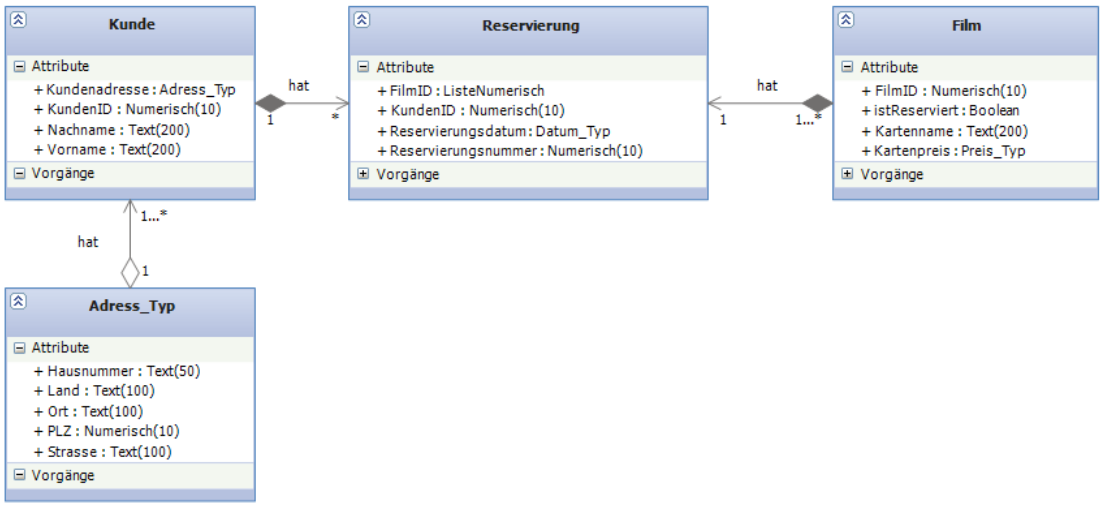


Abbildung 2.5.: Fachliches Datenmodell: Filmreservierung

Im nächsten Schritt des Entwicklungszyklus Abschnitt 2.1.2 „Entwurfsphase“ wird das fachliche Datenmodell in ein Komponentenmodell überführt. Dabei ist zu beachten, dass wenn möglich, fachlich eng zusammenhängende Typen zu einer Komponente gruppiert werden. Exemplarisch wird hier der Adress-Typ und die Entität Kunde in einer gemeinsamen Komponente „Kundenverwaltung“ zusammengefasst. Das System, welches wir jetzt erhalten haben, besteht aus 3 einfachen Komponenten.

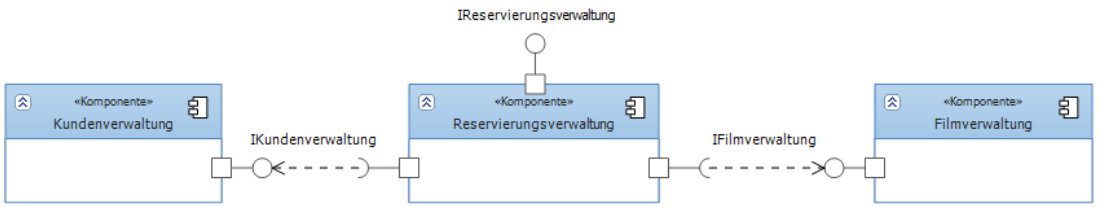


Abbildung 2.6.: Fachliche Architektur

Nun muss der Entwickler zunächst ein neues Projekt in Visual Studio erstellen. Danach für jede einzelne Komponente eine neue Klassenbibliothek anlegen.

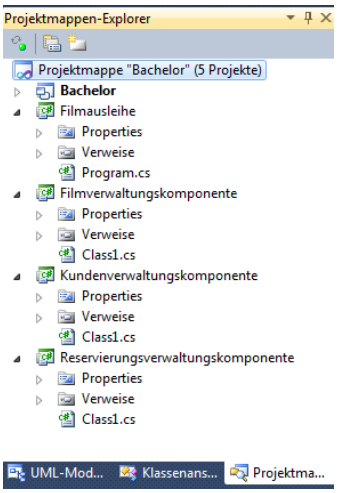


Abbildung 2.7.: Überführung der fachlichen Architektur in Klassenbibliotheken

Es müssen die Komponenteninnensichten modelliert, alle Interfaces implementiert und Codeskeletons erzeugt werden. Dies wird für die Komponente Filmverwaltung exemplarisch durchgeführt.

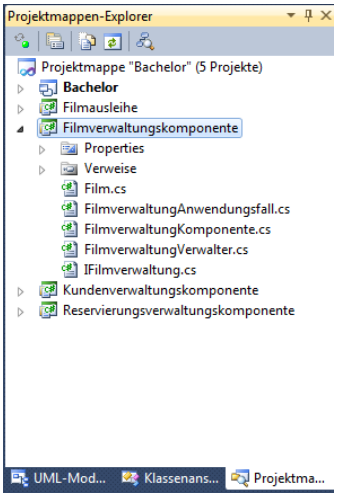


Abbildung 2.8.: Überführung der Innensicht einer Komponente in C#-Code

Die einzelnen Dateien müssen mit Codeskeletons gefüllt werden. Sinnvoll ist es hier mit dem Interface anzufangen, da andere Dateien dieses implementieren. Beispielhaft wurden einige Standardfunktionen integriert. Um den großen Arbeitsaufwand dieser manuellen Codegenerierung zu zeigen, wird dies nun für eine der vier Komponenten durchgeführt.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Filmverwaltungskomponente
{
    public interface IFilmverwaltung
    {
        /// <summary>
        /// Legt einen neuen Film an
        /// </summary>
        Film CreateFilm(string _filmName, int _filmLänge, double
            _filmPreis, bool _istReserviert);

        /// <summary>
        /// Sucht einen Film mit der ID in der Datenbank
        /// </summary>
        Film GetFilmByID(int _filmID);

        /// <summary>
        /// Sucht einen Film mit dem Namen in der Datenbank
        /// </summary>
        Film GetFilmByName(string _filmName);

        /// <summary>
        /// Speichert einen neuen Film in der Datenbank oder ändert einen
             bestehenden
        /// </summary>
        bool SaveFilm(Film _film);

        /// <summary>
        /// Löscht einen Film aus der Datenbank
        /// </summary>
        bool DeleteFilm(Film _film);
    }
}
```

Listing 2.1: Interface IFilmverwaltung

Als Nächstes sollte die Entität der Komponente modelliert werden. Sie kann 1:1 aus dem UML-Diagramm übernommen werden. Des Weiteren wird eine Mapping-Klasse für FluentN-Hibernate generiert.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using FluentNHibernate.Mapping;
6
7  namespace Filmverwaltungskomponente
8  {
9      /// <summary>
10     /// Repräsentiert die Film-Entität
11     /// </summary>
12     public class Film
13     {
14         public virtual int FilmId { get; private set; }
15         public virtual string FilmName { get; set; }
16         public virtual int FilmLänge { get; set; }
17         public virtual double FilmPreis { get; set; }
18         public virtual bool istReserviert { get; set; }
19
20         /// <summary>
21         /// Standardkonstruktor, wird von Hibernate benötigt
22         /// </summary>
23         public Film()
24         {
25         }
26
27         /// <summary>
28         /// Konstruktor um ein Film-Objekt zu erzeugen
29         /// </summary>
30         /// <param name="_name">Filmname</param>
31         /// <param name="_länge">Länge des Filmes in Minuten</param>
32         /// <param name="_preis">Preis des Filmes in Euro</param>
33         /// <param name="_istReserviert">True/False ist der Film
               reserviert</param>
34         public Film(string _name, int _länge, double _preis, bool
               _istReserviert)
35         {
36             this.FilmName = _name;
37             this.FilmLänge = _länge;
38             this.FilmPreis = _preis;
39             this.istReserviert = _istReserviert;
40         }
41
```

```
42          /// <summary>
43          /// To String Methode
44          /// </summary>
45          /// <returns></returns>
46          public override string ToString()
47          {
48              return FilmName + "(" + FilmId + "): " + FilmLänge + "min, "
                    + FilmPreis + "Eur - " + istReserviert;
49          }
50      }
51
52      /// <summary>
53      /// FluentNHibernate Mapping-Klasse für das Film-Objekt
54      /// </summary>
55      public class FilmMap : ClassMap<Film>
56      {
57          public FilmMap()
58          {
59              // mittels "Id" wird ein Id-Feld (eindeutige Nummer) gemappt
60              Id(x => x.FilmId);
61              Map(x => x.FilmName);
62              Map(x => x.FilmLänge);
63              Map(x => x.FilmPreis);
64              Map(x => x.istReserviert);
65          }
66      }
67  }
```

Listing 2.2: Entität Film

Die Filmverwaltungskomponente konfiguriert und initialisiert die Innensicht der Komponente. Sie implementiert das Interface und delegiert die Funktionen und Anfragen an den Anwendungsfall.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Persistence.Interfaces;
6
7  namespace Filmverwaltungskomponente
8  {
9      /// <summary>
```

```
10      /// Die FilmverwaltungKomponente stellt die zentrale Geschäftslogik
            für die
11      /// Filmverwaltung zur Verfügung.
12      /// </summary>
13      public class FilmverwaltungKomponente : IFilmverwaltung
14      {
15          private FilmverwaltungAnwendungsfall filmverwaltungAnwendungsfall
                = null;
16
17          public FilmverwaltungKomponente(IPersistenceManager
                persistenceManager)
18          {
19              // hier konfigurieren wir die Innensicht der Komponente
20              filmverwaltungAnwendungsfall = new
                    FilmverwaltungAnwendungsfall(persistenceManager);
21          }
22
23          public Film CreateFilm(string _filmName, int _filmLänge, double
                _filmPreis, bool _istReserviert)
24          {
25              return filmverwaltungAnwendungsfall.CreateFilm(_filmName,
                    _filmLänge, _filmPreis, _istReserviert);
26          }
27
28          public Film GetFilmByID(int _filmID)
29          {
30              return filmverwaltungAnwendungsfall.GetFilmByID(_filmID);
31          }
32
33          public Film GetFilmByName(string _filmName)
34          {
35              return filmverwaltungAnwendungsfall.GetFilmByName(_filmName);
36          }
37
38          public bool SaveFilm(Film _film)
39          {
40              return filmverwaltungAnwendungsfall.SaveFilm(_film);
41          }
42
43          public bool DeleteFilm(Film _film)
44          {
45              return filmverwaltungAnwendungsfall.DeleteFilm(_film);
46          }
```

```
47      }
48  }
```

Listing 2.3: Filmverwaltungskomponente

Im Anwendungsfall werden für alle Funktionen Basis-Codeskeletons erzeugt, die dann vom Entwickler mit der Geschäftslogik gefüllt werden müssen.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace Filmverwaltungskomponente
7  {
8      /// <summary>
9      /// Der Filmverwaltung-Anwendungsfall stellt die zentrale
           Geschäftslogik für die Filmverwaltung zur Verfügung.
10     /// </summary>
11     ///
12     /// <see cref="IFilmverwaltung"/>
13     internal class FilmverwaltungAnwendungsfall :IFilmverwaltung
14     {
15         private IPersistenceManager persistenceManager;
16
17         internal FilmverwaltungAnwendungsfall(IPersistenceManager
               persistenceManager)
18         {
19             this.persistenceManager = persistenceManager;
20         }
21
22         internal Film CreateFilm(string _filmName, int _filmLänge, double
                _filmPreis, bool _istReserviert)
23         {
24             throw new NotImplementedException();
25         }
26
27         internal Film GetFilmByID(int _filmID)
28         {
29             throw new NotImplementedException();
30         }
31
32         internal Film GetFilmByName(string _filmName)
33         {
34             throw new NotImplementedException();
```

```
35          }
36
37          internal bool SaveFilm(Film _film)
38          {
39              throw new NotImplementedException();
40          }
41
42          internal bool DeleteFilm(Film _film)
43          {
44              throw new NotImplementedException();
45          }
46      }
47  }
```

Listing 2.4: Filmverwaltung Anwendungsfälle

Im Code des Verwalters können zunächst nur rudimentäre Funktionen implementiert werden, da sie sehr spezifisch für jeden Anwendungsfall sind.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Persistence.Interfaces;
6
7
8  namespace Filmverwaltungskomponente
9  {
10     /// <summary>
11     /// Der FilmverwaltungVerwalter stellt Operationen zum Einfügen,
           Ändern und Löschen von Filmen zur Verfügung.
12     /// </summary>
13     internal class FilmverwaltungVerwalter
14     {
15         // Die konkrete Persistenz wird über Konstruktor-Injection von
               außen geliefert.
16         private IPersistenceManager persistenceManager = null;
17
18         public FilmverwaltungVerwalter(IPersistenceManager
               persistenceManager)
19         {
20             this.persistenceManager = persistenceManager;
21         }
22
23         /// <summary>
```

```
24          /// Methode um sich einen Film aus der Datenbank zu holen per ID
25          /// </summary>
26          /// <param name="_id">Film ID</param>
27          /// <returns>Ein Film-Objekt</returns>
28          public Film GetFilmByID(int _id)
29          {
30              return (from Film in persistenceManager.LinqQuery<Film>()
31                      where Film.FilmID == _id
32                      select Film).FirstOrDefault<Film>();
33          }
34
35          /// <summary>
36          /// Methode um sich einen Film aus der Datenbank zu holen per
                Name
37          /// </summary>
38          /// <param name="_name">Film Name</param>
39          /// <returns>Ein Film-Objekt</returns>
40          public Film GetFilmByID(string _name)
41          {
42              return (from Film in persistenceManager.LinqQuery<Film>()
43                      where Film.FilmName == _name
44                      select Film).FirstOrDefault<Film>();
45          }
46
47          /// <summary>
48          /// Methode um einen Film zu ändern oder neu anzulegen
49          /// </summary>
50          /// <param name="_film">Film Objekt</param>
51          public void UpdateFilm(Film _film)
52          {
53              persistenceManager.Save<Film>(_film);
54          }
55
56          /// <summary>
57          /// Methode um einen Film zu löschen
58          /// </summary>
59          /// <param name="_film">Film Objekt</param>
60          public void DeleteFilm(Film _film)
61          {
62              persistenceManager.Delete<Film>(_film);
63          }
64
65      }
```

```
66 }
```

Listing 2.5: Filmverwaltung Verwalter

Aus Gründen der Übersicht werden die Codeskeletons für die Anwendungskernfassade, den Persistenzmanager, den Berechtigungsmanager und die restlichen Komponenten hier nicht aufgeführt. Was man aber generell sagen kann, ist, dass die Codeskeletons für die Komponenten des Anwendungskerns allesamt nach dem gleichen Muster aufgebaut sind, wie die Komponente „**Filmverwaltung**“.

2.4. Anwendungsfälle bei der Entwicklung mit Quasar ohne Einsatz einer Entwicklungshilfe

Um eine Erweiterung für Visual Studio zu erstellen muss im Vorfeld analysiert werden, welche möglichen Funktionen und Prozesse ein Entwickler benötigt und wie häufig diese auftreten. Es muss entschieden werden, ob es nötig ist, all diese Anwendungsfälle umzusetzen. Es ist bestimmt auch möglich, einzelne Anwendungsfälle zu optimieren.

Zunächst einmal werden die Hauptanwendungsfälle betrachtet, wie sie bei der Entwicklung nach den Quasar-Richtlinien vorkommen. Diese Anwendungsfälle werden zunächst ohne Zuhilfenahme der neuen Visual Studio Erweiterung betrachtet, um Schwächen im Prozess aufzuzeigen.

2.4.1. Übertragen eines UML-Komponentendiagramms in Code

Der wohl wichtigste Anwendungsfall ist die Umsetzung des UML-Diagramms in korrekten C#-Code. Wie schon in Abschnitt 2.3 „Begleitendes Beispiel“ genau beschrieben ist dies der Anwendungsfall mit dem größten Potenzial für Fehler. Um eine bessere Übersicht zu bekommen, kann dieser Anwendungsfall aufgesplittet werden. Er besteht zum einem aus dem Anwendungsfall „Übertragen einer neuen Komponente in Code“ und zum anderen „Übertragen einer neuen Klasse in Code“. Im weiteren Verlauf kommen noch die Anwendungsfälle hinzu, die verschiedenen Klassen und Interfaces mit Codeskeletons zu bestücken.

2.4.2. Übertragen einer neuen Komponente in Code

Akteure	Software-Entwickler
Ziel	Übertragen einer Komponente in Code

Auslöser	Es gibt eine neue Anforderung, eine neue Entität, eine neue Fachlichkeit
Vorbedingungen	1. Komponente ist im UML-Diagramm vorhanden 2. Das UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Die UML-Komponente befindet sich als C#-Klassenbibliotek in der Solution
Erfolgsfall	1. Benutzer klickt im Projektmappen-Explorer mit der rechten Maustaste auf die Projektmappe 2. Benutzer wählt „**Hinzufügen -> neues Projekt**“ aus 3. Im Popup wählt Benutzer „**Visual C# -> Windows -> Klassenbibliothek**“ aus 4. Benutzer gibt den Namen der UML-Komponente als neuen Namen für die Klassenbibliothek ein 5. Benutzer drückt Taste „**OK**“ 6. Visual Studio überprüft, ob der eingegebene Name den Namenskonventionen entspricht 7. Wenn die Überprüfung erfolgreich war, wird die neue Klassenbibliothek mit dem ausgewählten Namen angelegt
Erweiterungen	8a. Hinzufügen von Verweisen zu anderen Projekten oder Bibliotheken
Fehlerfälle	6a. Der neue Komponentenname hat nicht zulässige Bezeichner in seinem Namen, wie z.B. "?,ß,&,$,...", welche die Namenskonventionen von C# verletzen 7b. Das System stellt fest, dass das Verzeichnis oder die Klassenbibliothek schon in irgendeiner Form vorhanden ist 7c. System hat keine Schreibrechte und kann die neue Klassenbibliothek nicht anlegen
Häufigkeit	Bei Erstübertragung eines Diagramms in Code 1 mal pro Komponente

Tabelle 2.1.: Anwendungsfall - Komponente in Code überführen

2.4.3. Übertragen einer neuen Klasse in Code

Akteure	Software-Entwickler
Ziel	Übertragen einer Klasse in Code
Auslöser	Es gibt eine neue Anforderung, Refactoring, neue Fachlichkeit
Vorbedingungen	1. Klasse ist im UML-Diagramm vorhanden 2. Relation Klassendiagramm zu Komponente im Komponentendiagramm ist bekannt 3. Das UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Die UML-Klasse befindet sich als C#-Klasse im dazugehörigen Projekt
Erfolgsfall	1. Benutzer klickt im Projektmappen-Explorer mit der rechten Maustaste auf das gewählte Projekt 2. Benutzer wählt „**Hinzufügen -> neues Element**“ aus 3. Im Popup wählt Benutzer „**Visual C# -> Code -> Klasse**“ aus 4. Benutzer gibt den Namen der UML-Klasse als neuen Namen für die Klasse ein 5. Benutzer drückt Taste „**OK**“ 6. Visual Studio überprüft, ob der eingegebene Name den Namenskonventionen entspricht 7. Wenn die Überprüfung erfolgreich war, wird die neue Klasse mit dem ausgewählten Namen angelegt
Erweiterungen	2a. Alternativ kann hier auch ein Interface ausgewählt und neu angelegt werden 3b. Benutzer gibt alternativ den Namen des UML-Interfaces als neuen Namen für das Interface ein
Fehlerfälle	6a. Der neue Komponentenname hat nicht zulässige Bezeichner in seinem Namen, wie z.B. "?,ß,&,$,...", welche die Namenskonventionen von C# verletzen 7b. Das System stellt fest, dass das Verzeichnis oder die Datei schon in irgendeiner Form vorhanden ist 7c. System hat keine Schreibrechte und kann die neue Klasse/Interface nicht anlegen

Häufigkeit	Bei Erstübertragung eines Diagramms in Code 5 + x mal pro Komponente (5 Klassen / Interfaces ist die minimale Anzahl pro Komponente)

Tabelle 2.2.: Anwendungsfall - Klasse in Code überführen

2.4.4. „Quasar-Anwendungsfall“ Datei mit Codeskeleton bestücken

Als exemplarischer Anwendungsfall für das Bestücken der verschiedenen Klassen und Interfaces mit Codeskeletons, dient das Bestücken eines „Quasar-Anwendungsfall“ als Beispiel.

Akteure	Software-Entwickler
Ziel	Befüllen eines „Quasar-Anwendungsfall“ mit Codeskeletons
Auslöser	ein neuer „Quasar-Anwendungsfall“ wurde angelegt
Vorbedingungen	1. „Quasar-Anwendungsfall“ ist im UML-Diagramm vorhanden und hat Relationen 2. Das UML-Diagramm entspricht der Referenzarchitektur 3. C#-Datei für den „Quasar-Anwendungsfall“ ist vorhanden und leer
Nachbedingungen	„Quasar-Anwendungsfall“ Datei ist mit korrekten Codeskeletons gefüllt
Erfolgsfall	1. Benutzer öffnet das Projektitem 2. System zeigt das Projektitem an 3. Benutzer fügt „**using-Direktiven**“ hinzu 4. Benutzer fügt „**Namespace**“ hinzu 5. Benutzer fügt „**Klasse-Header mit Sichtbarkeit und Interface Vererbung**“ hinzu 6. Benutzer fügt „**Instanzvariablen**“ hinzu 7. Benutzer fügt „**Konstruktor mit Parameter und Sichtbarkeit**“ hinzu 8. Benutzer fügt „**Instanziierung der Variablen**“ hinzu 9. Benutzer fügt „**Implementationsskeletons des Interfaces**“ hinzu 10. Benutzer fügt „**Konstruktor mit Parameter und Sichtbarkeit**“ hinzu

Erweiterungen	4a. Alternativ benutzerdefinierte Interfaces oder andere Klassenvererbungen 10b. Definieren einer partiellen Klasse in der Datei
Fehlerfälle	4a. Klassen-Header wird falsch bezeichnet (anders als im UML-Diagramm)
Häufigkeit	Bei Erstübertragung eines Diagramms in Code 1 + x mal pro „Quasar-Anwendungsfall“

Tabelle 2.3.: „Quasar-Anwendungsfall“ Datei mit Codeskeleton bestücken

2.4.5. Überführen eines Projektes vom Code in das UML-Diagramm

Akteure	Software-Entwickler
Ziel	Überführen eines Projektes vom Code in das UML-Diagramm
Auslöser	Es gibt eine neue Anforderung, Refactoring, neue Fachlichkeit
Vorbedingungen	1. Ein UML-Komponentendiagramm ist vorhanden 2. Das UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Eine neue Komponente befindet sich im UML-Komponentendiagramm
Erfolgsfall	1. Benutzer öffnet das Komponentendiagramm im Modellierungsprojekt 2. System zeigt das Komponentendiagramm an 3. Benutzer drückt im Diagramm die rechte Maustaste, Kontextmenü öffnet sich 4. Benutzer wählt „**Hinzufügen -> Komponente**“ aus 5. Benutzer benennt die Komponente um, neuer Name wird der Name der Klassenbibliothek 6. Benutzer gibt den Namen der UML-Klasse als neuen Namen für die Klasse ein 7. Benutzer legt ein Klassendiagramm an, dass die Innensicht der Komponente widerspiegelt 8. Benutzer benennt das neue Klassendiagramm um

Erweiterungen	6a. Benutzer fügt der Komponente ausgehende und eingehende Ports hinzu. Diese repräsentieren Schnittstellen, die implementiert werden müssen
Fehlerfälle	5a. Eine Komponente mit dem Namen ist schon vorhanden. Benutzer muss nun die Komponente und die Klassenbibliothek umbenennen
Häufigkeit	5-50 mal pro Projekt, je nachdem, wie gut der Entwurf des UML-Diagramms ist

Tabelle 2.4.: Überführen eines Projektes vom Code in das UML-Diagramm

2.4.6. Überführen einer Klasse vom Code in das UML-Diagramm

Akteure	Software-Entwickler
Ziel	Überführen einer Klasse vom Code in das UML-Diagramm
Auslöser	Es gibt eine neue Anforderung, Refactoring, neue Fachlichkeit, eine neue Entität
Vorbedingungen	1. Ein UML-Komponentendiagramm ist vorhanden 2. Die zugehörige Komponente befindet sich schon in diesem Diagramm 3. Es existiert ein Klassendiagramm, das zu der Komponente gehört, in der sich die zu überführende Klasse befindet
Nachbedingungen	Eine neue Klasse befindet sich im UML-Klassendiagramm
Erfolgsfall	1. Benutzer öffnet das Klassendiagramm im Modellierungsprojekt 2. System zeigt das Klassendiagramm an 3. Benutzer drückt im Diagramm die rechte Maustaste, Kontextmenü öffnet sich 4. Benutzer wählt „**Hinzufügen -> Klasse**“ aus 5. Benutzer benennt die Klasse um, neuer Name wird der Name der Projektitems 6. Benutzer gibt den Namen der UML-Klasse als neuen Namen für die Klasse ein

Erweiterungen	6a. Benutzer fügt der Klasse Abhängigkeiten und Vererbungen hinzu, welche zuerst in anderen Projektitems abgelesen werden müssen und danach übertragen werden
Fehlerfälle	5a. Eine Klasse mit dem Namen ist schon vorhanden. Benutzer muss nun die Klasse wieder umbenennen
Häufigkeit	20-100 mal pro Projekt, je nachdem, wie gut der Entwurf des UML-Diagramms ist

Tabelle 2.5.: Überführen einer Klasse vom Code in das UML-Diagramm

2.4.7. Überführen einer neuen Funktion von einem UML-Interface in Code

Akteure	Software-Entwickler
Ziel	Überführen einer neuen Funktion von einem UML-Interface in Code
Auslöser	Es gibt eine neue Anforderung, Refactoring, neue Fachlichkeit
Vorbedingungen	1. Ein C#-Projekt ist vorhanden 2. Das C#-Projekt besitzt ein Interface, das in einem vorhergegangenen Anwendungsfall erzeugt und mit Codeskeletons bestückt worden ist
Nachbedingungen	Im C#-Interface befindet sich die neue Funktion
Erfolgsfall	1. Benutzer öffnet das C#-Projektitem (Interface) 2. Benutzer scrollt an das Ende der Datei oder an die Stelle, die er für die neue Funktion als richtig erachtet 3. Benutzer fügt den Methodenrumpf in das Interface ein
Erweiterungen	4a. Benutzer fügt die neue Methode in allen implementierenden Klassen ein
Fehlerfälle	5a. Namenskonventionen werden verletzt, Benutzer muss die Funktion im UML-Diagramm und im C#-Code ändern
Häufigkeit	1-500 mal pro Projekt, je nachdem, wie gut der Entwurf des UML-Diagramms ist

Tabelle 2.6.: Überführen einer neuen Funktion von einem UML-Interface in Code

Als exemplarischer Anwendungsfall für das Ändern eines UML-Elements und das anschließende Übertragen in Code dient der folgende Anwendungsfall. Es wird eine neue Funktion im UML-Interface beschrieben und diese muss nun in den bestehenden Code übertragen werden.

2.4.8. Weitere Anwendungsfälle

Natürlich sind das bei weitem nicht alle Anwendungsfälle, die auftreten können. Aus Gründen der Übersicht werden nur die wichtigsten tabellarisch aufgelistet. Nachfolgend werden weitere Anwendungsfälle aufgeführt, die auftreten können, aber in ihrer Häufigkeit oder Wichtigkeit nicht so relevant sind, wie die Beispiele oder durch exemplarische Anwendungsfälle bereits ausreichend dokumentiert worden sind.

- „Quasar-Entität“ mit Codeskeletons bestücken
- „Quasar-Komponente“ mit Codeskeletons bestücken
- „Quasar-Verwalter“ mit Codeskeletons bestücken
- „Quasar-Interface“ mit Codeskeletons bestücken
- Änderungen in UML-Klassen in bestehenden Code überführen
- Änderungen in C#-Code in UML-Diagramme überführen
- Synchronisierung von UML-Diagramm und C#-Code
- Erstellen der Anwendungskernfassade
- Erstellen der Mappingklassen für NHibernate

2.5. Ableiten von Anforderungen und Funktionen aus den bestehenden Anwendungsfällen

Nachdem die typischen Anwendungsfälle bei der Entwicklung mit der Quasar-Referenzarchitektur analysiert wurden, kann man daraus Anforderungen und Funktionen ableiten, die die Entwicklung vereinfachen. Bestimmte Anwendungsfälle können komplett automatisiert werden. Das System soll den Softwareentwickler unterstützen und Fehler ausschließen. Integrierte Prüfalgorithmen verhindern, dass bestehende Konventionen verletzt werden und die Integrität des Entwurfes und des Quellcodes sicherzustellen. Synchronisationsaufgaben können automatisiert und fehlertolerant gestaltet werden.

Alle Anwendungsfälle zeigen, dass der gesamte Prozess extrem anfällig für Benutzerfehler ist. Namenskonventionen in UML-Diagrammen werden nicht geprüft. Benutzeränderungen an einer Stelle führen zu Änderungen an vielen weiteren Stellen. Die Synchronität von Code und Entwurf ist nur sehr schwer zu pflegen und aufrechtzuerhalten.

Der Benutzer ist manchmal länger mit der Pflege des UML-Entwurfes beschäftigt, wie die Änderung im C#-Code dauert. Der Workload des Benutzers soll minimiert werden, damit der sich mit der Entwicklung beschäftigen kann und nicht mit der „Verwaltung“.

1. Automatische Codegenerierung
   a) automatische Validierung des Komponentendiagramms
   b) automatische Generierung der Klassenbibliotheken, Klassen und Interfaces
   c) automatische Erstellung von Verweisen
2. Synchronisierung von UML-Diagramm mit bestehendem Code
   a) Synchronisation auf Komponentenebene
   b) Synchronisation auf Klassenebene
3. Synchronisierung von bestehendem Code mit UML-Diagramm
   a) Synchronisation auf Projektebene
   b) Synchronisation auf Projektitemebene
4. Erstellung von Projekt- und Projektitemevents
   a) Rename-/Delete-Eventlistener auf Projektebene
   b) Rename-/Delete-Eventlistener auf Projektitemebene
5. Anpassen des UML-Designers
   a) erstellen eines eigenen UML-Profiles
   b) erstellen von Stereotypen für die einzelnen Quasarelemente
   c) erstellen von verschiedenen Shapes für den UML-Designer
   d) verschiedene Exportfunktionen und Funktionen, die die Usability verbessern
6. Anpassen des UML-Komponentendesigners
   a) Namenskonventionen prüfen
   b) Verlinkung zu Klassendiagramm erstellen
   c) Kontextmenüs für Generierung, Validierung und Synchronisation erstellen
   d) Eventlistener für Rename, Add erstellen
7. Anpassen des UML-Klassendesigners
   a) Namenskonventionen prüfen
   b) Verlinkung zu generierten Dateien erstellen
   c) Kontextmenüs für Generierung, Validierung und Synchronisation erstellen
8. Anpassen des Projektmappen-Explorers
   a) Kontextmenüs für Validierung und Synchronisation erstellen

2.6. Anwendungsfälle unter Einsatz der geplanten Visual Studio Erweiterung

Im Folgenden werden einige Anwendungsfälle skizziert, wie sie mithilfe der erstellten Erweiterung ablaufen könnten. Die Erweiterung wird den Benutzer in vielen Schritten unterstützen, ihm Arbeit ganz abnehmen oder erleichtern.

2.6.1. Automatisches Übertragen des UML-Komponentendiagramms in Code

Wie schon bei der manuellen Übertragung des Komponentendiagramms in Code, so wird auch bei der automatischen Übertragung der gesamte Prozess in mehrere Anwendungsfälle untergliedert. Als Erstes den Anwendungsfall der Validierung des UML-Diagramms, anschließend die Generierung der Klassenbibliotheken (automatisches Übertragen einer neuen Komponente in Code) und der Codeskeletons (automatisches Übertragen einer neuen Klasse in Code).

2.6.2. Automatisches Übertragen einer neuen Komponente in Code

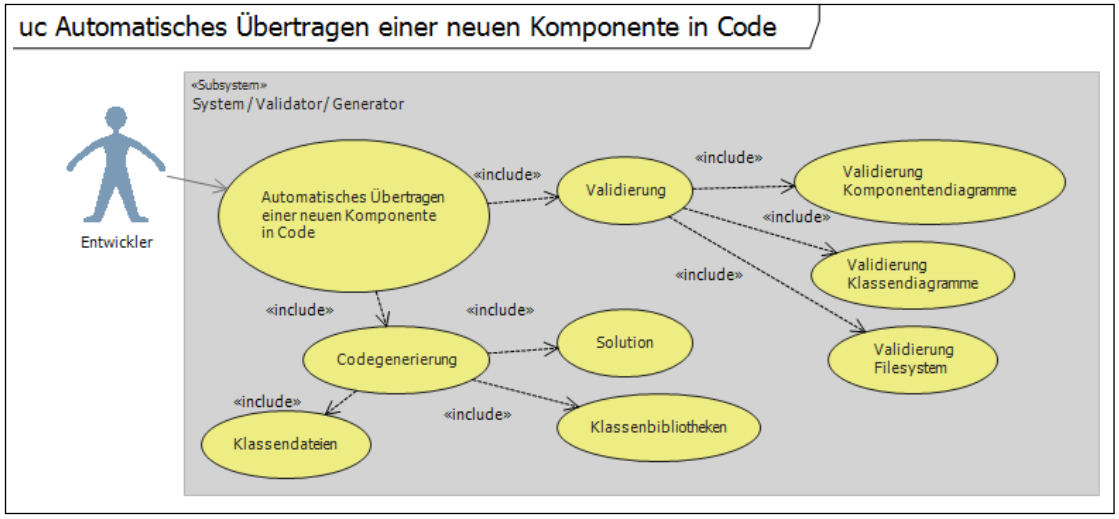


Abbildung 2.9.: Anwendungsfalldiagramm - Automatisches Übertragen einer neuen Komponente in Code

In diesem Anwendungsfall werden viele Anwendungsfälle der manuellen Codeerstellung vereint oder automatisiert aufgerufen. Es beginnt mit der Validierung, die ohne Einsatz dieser Erweiterung komplett manuell und im ermessen des Benutzers liegt. Anschließend folgt die

Generierung der Klassenbibliotheken aus den Komponenten. Nach Erstellung der Komponenten wird der Anwendungsfall (automatisches Übertragen einer neuen Klasse in Code) vom System aufgerufen und durchgeführt. Nach Erstellung der Klassen- und Interface-Dateien werden die Anwendungsfälle zum Füllen dieser Dateien mit Codeskeletons ausgeführt.

Akteure	Software-Entwickler
Ziel	Übertragen einer Komponente in Code
Auslöser	Es gibt eine neue Anforderung, eine neue Entität, eine neue Fachlichkeit
Vorbedingungen	1. Die Komponente wurde noch nicht erstellt 2. Komponente ist im UML-Diagramm vorhanden 3. Das UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Die UML-Komponente befindet sich als C#-Klassenbibliotek in der Solution
Erfolgsfall	1. Benutzer klickt im Hauptmenü auf „**Codegenerator -> Code erzeugen**“ 2. System beginnt mit der Validierung des Komponentendiagramms und der verlinkten Klassendiagramme (Anwendungsfall: Validierung) 3. System meldet dem Benutzer, ob alle Voraussetzungen erfüllt sind 4. Wenn alle Voraussetzungen erfüllt sind, startet das System die Generierung der Klassenbibliotheken 5. System generiert die Klassen- und Interfaceskeletons (Anwendungsfall: Klassenerzeugung)
Erweiterungen	1a. Benutzer kann auch nur bestimmte Komponenten im UML-Diagramm auswählen und für diese die Generierung starten
Fehlerfälle	2a. Validierung schlägt fehl, System zeigt dem Benutzer eine Fehlerliste, die er nun bereinigen muss
Häufigkeit	Bei Erstübertragung eines Diagramms in Code 1 mal pro Komponente

Tabelle 2.7.: Anwendungsfall - Komponente automatisch in Code überführen

2.6.3. Synchronisierung Projektitem mit UML-Diagramm

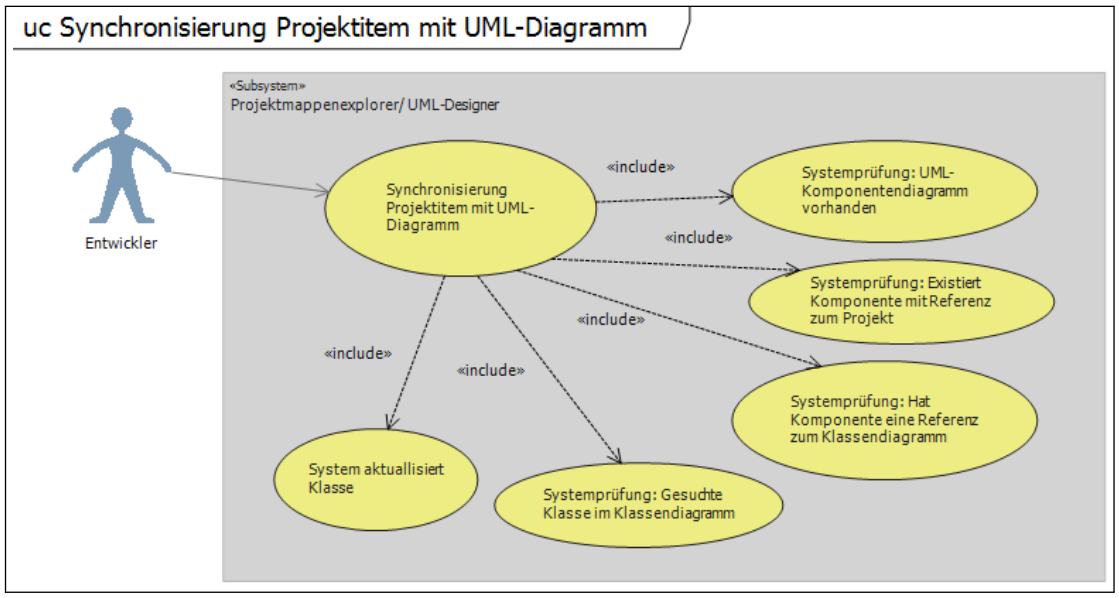


Abbildung 2.10.: Anwendungsfalldiagramm - Synchronisierung Projektitem mit UML-Diagramm

Als exemplarisches Beispiel für die Synchronisierung eines Projektitems mit dem UML-Diagramm, dient der Anwendungsfall Synchronisierung eines „Quasar-Verwalter“ vom Sourcecode mit UML-Diagramm. Alle anderen Projektitems werden auf die gleiche oder ähnliche Art und Weise synchronisiert, abhängig von ihrem Inhalt und der Art der Daten, die in der Datei vorhanden sind.

Akteure	Software-Entwickler
Ziel	Synchronisierung „Quasar-Verwalter“ mit UML-Diagramm
Auslöser	Das UML-Shape wurde versehentlich gelöscht, neue Daten wurden hinzugefügt
Vorbedingungen	1. Ein UML-Klassendiagramm ist vorhanden 2. Ein UML-Komponentendiagramm ist vorhanden 3. Die Komponente, in der sich der „Quasar-Verwalter“ befindet und das Klassendiagramm sind untereinander referenziert

Nachbedingungen	Der „Quasar-Verwalter“ ist mit dem UML-Klassendiagramm synchronisiert
Erfolgsfall	1. Benutzer macht einen Rechtsklick im Projektmappen-Explorer auf den „Quasar-Verwalter“ 2. System zeigt Kontextmenü an 3. Benutzer wählt „**Synchronisieren mit UML-Diagramm**“ aus 4. System prüft, ob Komponentendiagramm vorhanden ist 5. System prüft, ob Komponente mit Referenz zum Projekt vorhanden ist 6. System prüft, ob Komponente eine Referenz zum Klassendiagramm hat 7. System prüft, ob Klasse im Klassendiagramm vorhanden ist 8. System aktualisiert die UML-Klasse
Erweiterungen	7a. System stellt fest, dass der „Quasar-Verwalter“ nicht im UML-Klassendiagramm vorhanden ist 7b. System prüft, ob „Quasar-Verwalter“ im UML-Modell-Explorer vorhanden ist 7c. Wenn vorhanden, fügt das System den „Quasar-Verwalter“ im Klassendiagramm ein 7d. Wenn nicht vorhanden, erzeugt das System ein neues Shape und fügt dieses ein 7e. System prüft, ob alle Vererbungen und Abhängigkeiten vorhanden sind und fügt diese gegebenenfalls neu hinzu
Fehlerfälle	4a. System bricht den Vorgang ab 5b. System bricht den Vorgang ab 6c. System bricht den Vorgang ab
Häufigkeit	10-100 pro Tag

Tabelle 2.8.: Anwendungsfall - Synchronisation „Quasar-Verwalter“ mit UML-Diagramm

2.6.4. Synchronisation UML-Shape mit bestehendem Sourcecode

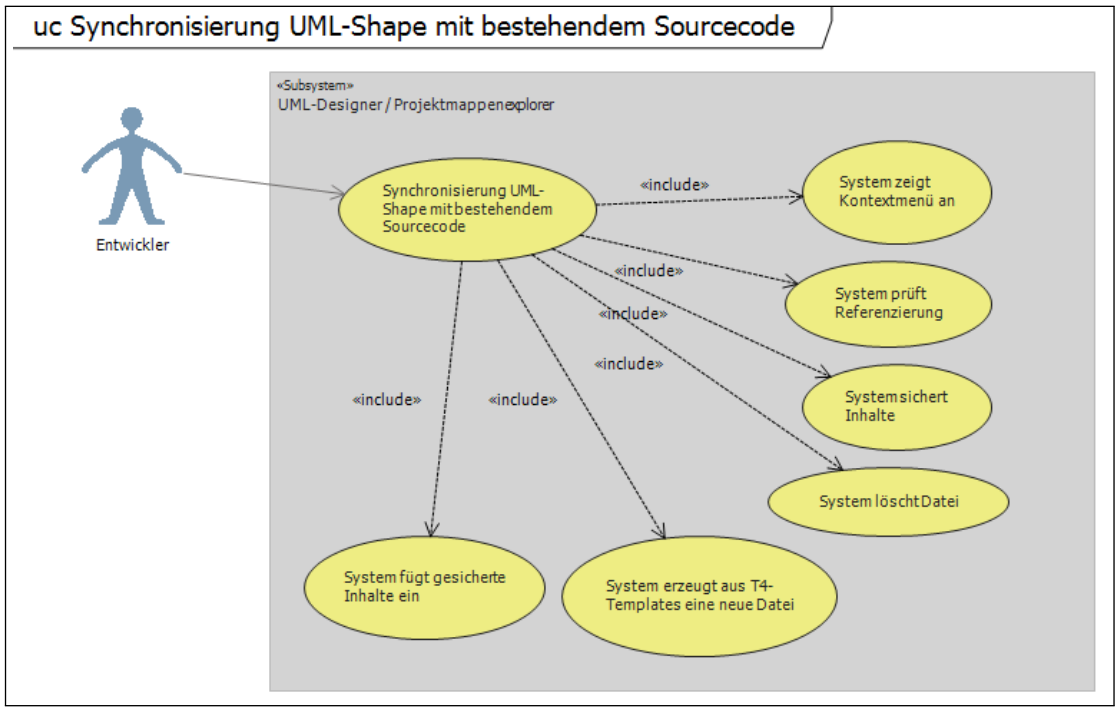


Abbildung 2.11.: Anwendungsfalldiagramm - Synchronisierung UML-Shape mit bestehendem Sourcecode

Als Beispiel für die Synchronisierung eines UML-Shapes mit bestehendem Sourcecode dient der Anwendungsfall Synchronisation „Quasar-Anwendungsfall“ von UML-Diagramm nach C#-Code. Alle anderen Shapes werden auf die gleiche oder ähnliche Art und Weise synchronisiert, abhängig von ihrem Inhalt und der zu sichernden Daten.

Akteure	Software-Entwickler
Ziel	Synchronisation „Quasar-Anwendungsfall“ mit C#-Code
Auslöser	Im UML-Diagramm wurden neue Klassen eingefügt, die Sourcecode Datei ist Out-of-Date
Vorbedingungen	1. Ein UML-Klassendiagramm ist vorhanden 2. Ein UML-Shape „Quasar-Anwendungsfall“ ist vorhanden 3. Eine Anwendungsfall-Datei existiert, auf die das UML-Shape referenziert

Nachbedingungen	Der „Quasar-Anwendungsfall“ ist mit dem C#-Code synchronisiert
Erfolgsfall	1. Benutzer macht einen Rechtsklick auf das Shape „Quasar-Anwendungsfall“ im Klassendiagramm 2. System zeigt Kontextmenü an 3. Benutzer wählt „**Synchronisieren mit Sourcecode**“ aus 4. System prüft, ob Referenzierung vorhanden ist 5. System beginnt mit der Synchronisierung 6. System sichert zuerst alle benutzerspezifischen Implementierungen und Inhalte, um den Datenverlust zu verhindern 7. System löscht die alte Datei 8. System erzeugt aus T4-Templates eine neue Datei 9. System fügt benutzerspezifische Inhalte wieder in die neue Datei ein
Erweiterungen	4a. System stellt fest, dass die Datei zu „Quasar-Anwendungsfall“ nicht vorhanden ist 4b. System legt eine neue Datei mit Codeskeletons an 4c. System muss keine Synchronisation durchführen
Fehlerfälle	4a. System geht wie in den Erweiterungen beschrieben vor 6b. System stellt fest, dass der Benutzer sich nicht an die vorgeschriebene Codestruktur gehalten hat und bricht den Vorgang ab 8c. System hat keine Schreibrechte und bricht den Vorgang ab
Häufigkeit	10-100 pro Tag

Tabelle 2.9.: Anwendungsfall - Synchronisation „Quasar-Anwendungsfall“ mit Sourcecode

2.6.5. Rename Events

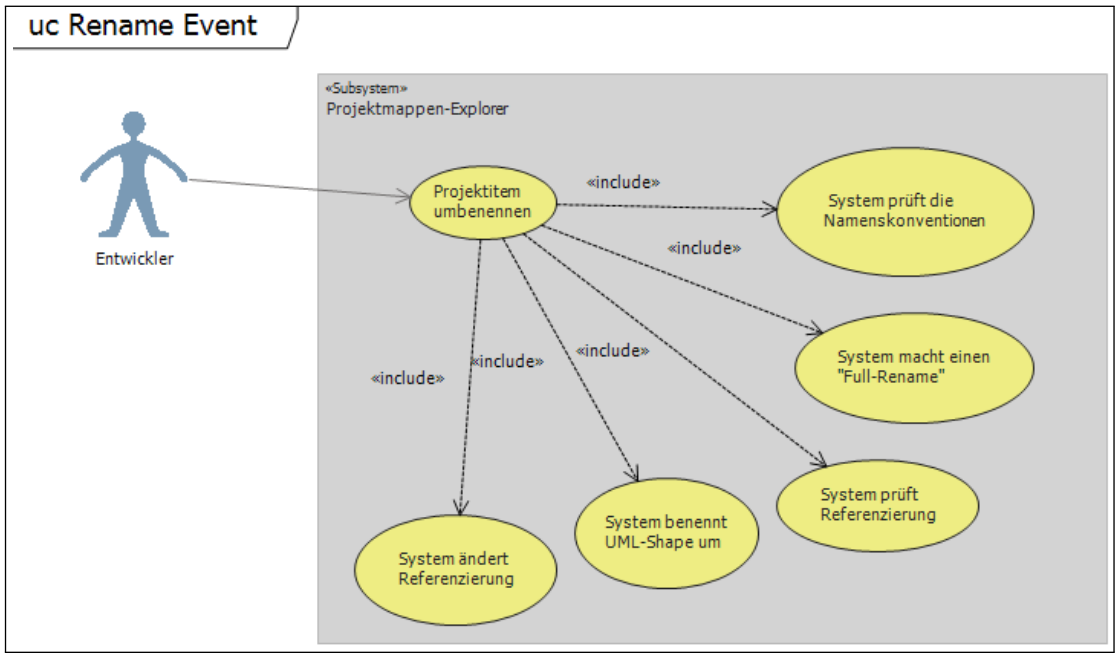


Abbildung 2.12.: Anwendungsfalldiagramm - Projektitem Rename Event

Um bestimmte Projektitems referenzieren zu können, ist es nötig, auf Rename-Events zu reagieren. Da der Dateiname eines Projektitems die einzige Möglichkeit ist, eine Referenz aufzubauen, muss hier besonderes Augenmerk darauf gelegt werden. Projektitems haben im Gegensatz zu Projekten keine GUID, an denen man sie eindeutig wiedererkennen kann. Weitere Ausführungen zum Thema Referenzierung werden im Entwurf in Abschnitt 3.3 „Referenzierung“ getätigt.

Akteure	Software-Entwickler
Ziel	Umbenennen eines „Quasar-Verwalter“ im Sourcecode
Auslöser	Zuständigkeit des Verwalters hat sich geändert, neue Namenskonvention, neue Anforderung, neue Fachlichkeit
Vorbedingungen	1. Ein UML-Klassendiagramm ist vorhanden 2. Ein UML-Shape „Quasar-Verwalter“ ist vorhanden 3. Eine C#-Datei „Quasar-Verwalter“ ist vorhanden 4. UML-Shape referenziert auf die C#-Datei

Nachbedingungen	Der „Quasar-Verwalter“ ist mit allen verweisenden C#-Dateien und dem UML-Shape synchronisiert
Erfolgsfall	1. Benutzer benennt „Quasar-Verwalter“ im Projektmappen-Explorer um 2. System prüft den Namen, ob er den Namenskonventionen entspricht 3. System beginnt mit der Synchronisierung 4. System macht einen „Full-Rename“, dass heißt, alle referenzierenden Objekte im Sourcecode werden mit umbenannt, ähnlich wie bei der Refaktorisierung mit „F2“ 5. System prüft, ob Referenzierung zu einem UML-Shape vorhanden ist 6. System benennt das UML-Shape um 7. System ändert die Referenzierung im UML-Shape zu der umbenannten Datei
Erweiterungen	keine
Fehlerfälle	2a. System stellt fest, dass der neue Name eine Namenskonvention verletzt und macht einen Rollback 6b. System stellt fest, dass keine UML-Shape gefunden werden kann und beendet den Vorgang an dieser Stelle
Häufigkeit	10-100 pro Tag

Tabelle 2.10.: Anwendungsfall - Synchronisation „Quasar-Anwendungsfall“ Rename-Event

2.6.6. Delete Events

Um die Integrität der Referenzierungen zu gewährleisten, muss bei Löschung von Sourcecode Dateien darauf reagiert werden. Es muss geprüft werden, ob ein UML-Shape auf dieses Projekt oder das Projektitem referenziert hat. Wenn dies der Fall ist, muss diese Referenzierung gelöscht werden, da sie sonst ein sogenannter „Dead-Link“ ist.

Akteure	Software-Entwickler
Ziel	Löschen eines „Quasar-Verwalter“ im Sourcecode
Auslöser	Zuständigkeit des Verwalters hat sich geändert, er soll komplett neu erzeugt werden

Vorbedingungen	1. Eine C#-Datei „Quasar-Verwalter“ ist vorhanden
Nachbedingungen	Der „Quasar-Verwalter“ ist gelöscht
Erfolgsfall	1. Benutzer löscht „Quasar-Verwalter“ im Projektmappen-Explorer 2. System beginnt mit der Synchronisierung 3. System prüft, ob Referenzierung zu einem oder mehreren UML-Shapes vorhanden ist 4. System löscht die Referenzierung in den UML-Shapes
Erweiterungen	keine
Fehlerfälle	3a. System stellt fest, dass kein UML-Shapes gefunden werden kann und beendet den Vorgang an dieser Stelle
Häufigkeit	10-100 pro Tag

Tabelle 2.11.: Anwendungsfall - Synchronisation „Quasar-Anwendungsfall“ Delete-Event

2.6.7. Anlegen einer neuen Komponente im UML-Diagramm

Akteure	Software-Entwickler
Ziel	Anlegen einer neuen Komponente im UML-Diagramm
Auslöser	Eine neue Anforderung wird ermittelt oder ein neues Komponentendiagramm wird erstellt
Vorbedingungen	1. Ein UML-Diagramm ist entweder vorhanden oder muss angelegt werden 2. Das UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Ein neues UML-Diagramm befindet sich im UML-Diagramm, ein neues Klassendiagramm für die Komponente wird erstellt

Erfolgsfall	1. Entwickler öffnet ein UML-Komponentendiagramm 2. System zeigt das Komponentendiagramm an 3. Entwickler wählt das Tool „**Komponente**“ in der Toolbox aus 4. Entwickler fügt neue Komponente per Drag & Drop dem Diagramm hinzu 5. System erstellt ein neues UML-Klassendiagramm für diese Komponente und verknüpft die beiden 6. System legt alle notwendigen Klassen für die Quasar-Referenzarchitektur an 7. Entwickler benennt die neu erzeugte Komponente um 8. System prüft den angegebenen Namen und gibt wenn nötig einen Fehler aus
Erweiterungen	3a. Entwickler wählt über Rechtsklick „**Hinzufügen -> Komponente**“ aus
Fehlerfälle	5a. System hat keine Schreibrechte und kann das neue Klassendiagramm nicht anlegen 8b. Der neue Komponentenname hat nicht zulässige Bezeichner in seinem Namen, wie z.B. "?,ß,&,$,...", welche die Namenskonventionen von C# verletzen
Häufigkeit	Bei Neuerstellung eines Diagramms 1 mal pro Komponente

Tabelle 2.12.: Anwendungsfall - Neue Komponente im UML-Diagramm hinzufügen

2.6.8. Hinzufügen einer neuen Klasse in einer Komponente im UML-Diagramm

Akteure	Software-Entwickler
Ziel	Hinzufügen einer neuen Klasse in einer Komponente im UML-Diagramm
Auslöser	Eine neue Klasse wird benötigt, da sich die Anforderungen geändert haben oder eine leere Komponente erstellt wurde
Vorbedingungen	1. Das vorhandene UML-Diagramm entspricht der Referenzarchitektur
Nachbedingungen	Ein neue Klasse befindet sich im UML-Diagramm

Erfolgsfall	1. Entwickler öffnet ein UML-Klassendiagramm 2. System zeigt das Klassendiagramm an 3. Entwickler wählt das Tool „**Klasse - <Stereotype>**“ in der Toolbox aus 4. Entwickler fügt neue Klasse per Drag & Drop dem Diagramm hinzu 5. Entwickler benennt die neu erzeugte Klasse um 6. System prüft den angegebenen Namen und gibt wenn nötig einen Fehler aus
Erweiterungen	3a. Entwickler wählt über Rechtsklick „**Hinzufügen -> Klasse**“ aus 3b. Entwickler wählt in den Eigenschaften den Stereotype für diese Klasse aus
Fehlerfälle	5a. Der neue Klassenname hat nicht zulässige Bezeichner in seinem Namen, wie z.B. "?,ß,&,$,...", welche die Namenskonventionen von C# verletzen.
Häufigkeit	Mehrere Male pro Tag möglich, je nach Größe des Software-Projektes

Tabelle 2.13.: Anwendungsfall - Neue Klasse im UML-Diagramm hinzufügen

2.6.9. Weitere Anwendungsfälle

Einige weitere Anwendungsfälle, die denkbar und möglich sind:
- Anlegen einer neuen Klasse im Quellcode
- Anlegen einer neuen Funktion im Quellcode
- Anlegen eines neuen Interfaces samt Abhängigkeit im Quellcode
- Löschen einer Komponente/Klasse/Interface/Funktion im Quellcode
- Löschen einer Komponente/Klasse/Interface im UML-Diagramm

Die Frage, die sich bei der Fülle der Anwendungsfälle stellt, ist, welche Synchronität man zwischen UML-Diagramm und Sourcecode pflegen will und kann.

3. Entwurf

Im Entwurf werden zunächst verschiedene Aspekte zum Thema Deployment, Template-Design und Referenzierung betrachtet, um zu verstehen, wie diese funktionieren, sich in das Projekt einpassen und welche für die spezielle Umsetzung in diesem Fall geeignet sind. Danach werden die Hauptbestandteile Validierung, Code-Generierung und Synchronisation genauer analysiert. Anschließend wird die fachliche Architektur des Gesamtsystems erstellt und Events und Funktionen festgelegt.

3.1. Package Deployment

Bevor mit dem Entwurf des Systems begonnen wird, müssen die Rahmenbedingungen abgrenzt werden, wie später die Visual Studio Erweiterung verpackt und installiert werden soll. Hierbei werden zunächst die Anforderungen, die in Abhängigkeit mit der Visual Studio Erweiterung stehen, betrachtet. Zum einen kann man einige direkt aus Kapitel 2 ableiten, andere ergeben sich aus den allgemeinen Anforderungen, die vor Beginn der Arbeit aufgestellt worden sind.

- erstellen diverser UML-Designer Erweiterungen und Eventlistener, welche normalerweise MEF-Komponenten sind
- erstellen von Projekt- und Projektitem Templates, als Vorlagen für Quasar-Projekte
- erstellen und einbinden von benutzerdefinierten Shapes in der Toolbox Control
- erstellen und einbinden eigener UML-Profile
- erstellen und installieren diverser benutzerdefinierter Dateien und Daten
- erstellen und einbinden von verschiedenen Kontextmenüs

Unablässig für das Projekt ist es, das der bestehende UML-Designer angepasst werden kann und muss. Mit dem .NET 4.0 Framework wurde von Microsoft eine Möglichkeit geschaffen, ihren UML-Designer zu erweitern. Diese Erweiterungen werden in sogenannte MEF-Projekte (Managed Extensibility Framework) verpackt, die dann installiert und in das System eingebunden werden (Microsoft (e)).

Erweiterungstyp	VSIX	MSI	VSI
Projekt Template	✓	✗	✓
Projektitem Template	✓	✗	✓
Assembly	✓	✓	✗
MEF Komponente	✓	✓	✗
VSPackage	✓	✓	✗
Toolbox Control	✓	✓	✓
Macro	✗	✓	✓
Add-in	✗	✓	✓
Code Schnipsel	✗	✗	✓
Benutzerdefinierte Erweiterung	✓	✓	✗

Tabelle 3.1.: Visual Studio - Extension Deployment (Microsoft (h))

Des Weiteren wird Visual Studio selber angepasst. Um die Usability zu erhöhen, ist es nötig, verschiedene neue Menüs und Kontextmenüs zu erstellen, um auf die bereitgestellten Funktionen schnell und effektiv zugreifen zu können. Hierzu wird in den meisten Fällen ein VSPackage genutzt.

Da nun aber VSI-Pakete weder MEF-Projekte noch VSPackage hosten und deployen können, fallen diese bei unserer Auswahl nun schon weg. Des Weiteren werden benutzerdefinierte Projektvorlagen mit in die Erweiterung integriert, um dem Anwender später vorgefertigte Schablonen zur Verfügung stellen zu können. Dies ist mit einem VSI-Paket nicht möglich. Somit scheidet auch diese Möglichkeit aus den weiteren Betrachtungen aus.

Letztendlich bleibt nur noch eine Erweiterungspaketform, das „VSIX-Paket“, übrig. Es erfüllt alle Funktionen, die laut Spezifikation benötigt werden. Leider ist es nicht möglich Makros zu deployen, was im weiteren Verlauf vielleicht hilfreich gewesen wäre. Ein weiterer kleiner Nachteil von VSIX-Paketen besteht darin, dass es zwingend notwendig ist, das Visual Studio SDK auf dem Rechner installiert sein muss, um ein „VSIX-Paket“ erfolgreich zu installieren.

Da für die Erweiterung das Visual Studio SDK und noch einige andere Updates benötigt werden, damit es lauffähig ist, ist dieser Nachteil hinfällig.

3.2. Codetemplates - T4 vs. CodeDOM

Sowohl T4 als auch CodeDOM sind Codegenerierungswerkzeuge, die Microsoft in seinen Visual Studio Versionen ab 2008 zur Verfügung stellt. Für den allgemeinen Einsatz eines Templatesystems bei der Codegenerierung spricht natürlich zum einen die Wiederverwendbarkeit

an mehreren Stellen eines Projektes, zum anderen auch eine gute Wartbarkeit. Der zusätzlich erzeugte Overhead wird durch diese Punkte mehr als wettgemacht.

T4 (Text Template Transformation Toolkit) generiert den Code im Wesentlichen aus einer textuellen Vorlage mit Hilfe von String-Verkettungen. Dabei werden keine Objekte erzeugt. Die Ausgabe des Codes geschieht in einer Textdatei. CodeDOM hingegen generiert den Code direkt mit Hilfe des Objektgraphen. Für jedes zu erzeugende Element wird ein neuer Zweig im Objektbaum angelegt. T4 ermöglicht es, einen großen Teil des Ausgabe-Codes in Template-Blöcke zu legen, wie zum Beispiel Kommentare oder Headerinformationen die sich nicht verändern. Das bietet eine leichte Lesbarkeit und eine gute Editierbarkeit. Nachteil ist, dass man für verschiedene Sprachen auch verschiedene Templates anlegen muss. CodeDOM Generatoren sind oft sehr verschachtelt und schwer zu lesen. Aber sie ermöglichen es, mit einer Template-Datei Codeausgaben für mehrere Sprachen zu erzeugen.

3.2.1. Funktion von T4

Die Visual Studio T4-Engine führt 2 Schritte aus, um aus einem Template einen Output zu generieren. In Visual Studio enthält ein T4-Template eine Mischung von Textblöcken und Steuerelementlogik. Die Steuerelementlogik wird in Form von Fragmenten des Programmcodes in Visual C# oder Visual Basic geschrieben. Die generierte Datei kann Text einer beliebigen Art sein, z. B. eine Webseite, eine Ressourcendatei oder Programmquellcode in **e**iner beliebigen Sprache.

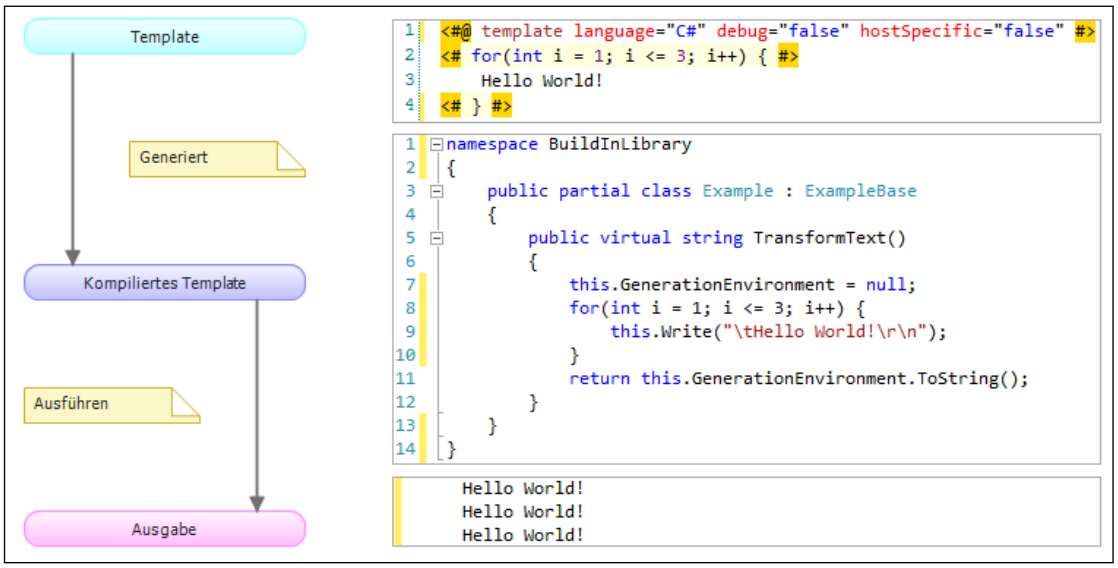


Abbildung 3.1.: Funktionsweise von T4

Während des ersten Schrittes wird das Template von der T4-Engine kompiliert. Sie analysiert die Verarbeitungsanweisungen, Text- und Code-Blöcke, Includes und Preprozessoranweisungen und erzeugt eine konkrete Text-Transformations-Klasse und erstellt für sie eine .NET-Assembly. Im zweiten Schritt erzeugt die T4-Engine eine Instanz dieser Klasse und weist wenn benötigt externe Variablen zu. Danach wird die TransformText-Methode aufgerufen und die generierte Zeichenfolge wird in die Ausgabedatei gespeichert.

3.2.2. Vorteile und Nachteile von T4

Für den Einsatz von T4 spricht seine Einfachheit und Benutzerfreundlichkeit. Die erstellten Templates sind klar strukturiert und für fast jeden lesbar. Das liegt an einer ASP ähnlichen sehr rudimentären Syntax. Da bei der Anwendungsentwicklung in der Regel nur der Code für eine Sprache generiert werden muss, ist es für diesen Zweck das bevorzugte Werkzeug. Erweiterungen und Korrekturen in bestehenden Templates lassen sich einfach und ohne großes Verständnis des gesamten Codes durchführen. In verschiedenen Online-Bibliotheken gibt es schon für sehr viele Problemstellungen fertige Templates, die ggf. nur noch leicht modifiziert werden müssen.

Aber es gibt auch Nachteile. Da die T4-Engine in ihrer bestehenden Integration in Visual Studio relativ neu ist, gibt es hier einen großen Verbesserungsbedarf in der Usability. Der Editor in Visual Studio bietet fast kein Syntax-Highlighting, kein IntelliSense für eigene Klassen. Die

Autovervollständigung funktioniert nur sehr eingeschränkt. Um effektiv arbeiten zu können, muss man auf meist kostenpflichtige Plugins zurückgreifen.

3.2.3. Vorteile und Nachteile von CodeDOM

Eine klare Stärke von CodeDOM ist die Möglichkeit, aus einem Template Code für mehrere Sprachen und Systeme zu erstellen. Es wäre für einen Entwickler schwer und zeitaufwendig mehrere Templates zu erstellen, zu pflegen und zu warten. Hier würden zusätzliche Kosten entstehen. Um ein plattformunabhängiges Werkzeug zu erstellen, wäre es sinnvoll, CodeDOM zu wählen. Da aber schon zu Beginn der Entwicklung festgelegt wurde, die geplante Erweiterung nur für Visual Studio zu schreiben und der erzeugte Code C# sein wird, war es nicht nötig ein CodeDOM Template zu wählen.

Ein Nachteil von CodeDOM ist der eher kryptische Aufbau. Um ein solches Template verstehen zu können, muss man sich etwas Zeit nehmen, da es ist weniger intuitiv als T4 ist. Da CodeDOM aber direkt im CodeTree arbeitet, bietet es natürlich auch sehr viele Vorteile. Als Beispiel wären, „typeof“ Prüfungen, das Benutzen von Objektreferenzen und viele andere zu nennen, die auf Textebene nicht funktionieren.

```
1  <#@ template language="C#" #>
2  <#+
3  public class ClassTemplate : Template
4  {
5      public ModelClass modelClass { get; set; }
6
7      public override string TransformText()
8      {
9  #>
10 namespace <#= TransformationContext.DefaultNamespace #>
11 {
12     using System;
13     using System.Collections.Generic;
14
15     public class <#= this.modelClass.Name #>
16     {
17 <#+
18         foreach (ModelAttribute attribute in this.modelClass.Attributes)
19         {
20 #>
21         public <#= attribute.Type #> <#= attribute.Name #> { get; set; }
22 <#+
```

```
        }
#>
    }
}
<#+
        return this.GenerationEnvironment.ToString();
    }
}
#>
```

Listing 3.1: T4 Template zum Schreiben von Instanzvariablen einer Klasse (Sych)

```
using System.CodeDom;

public class ClassTemplate : Template
{
    protected virtual void RenderProperty(ModelAttribute attribute)
    {
        CodeMemberProperty property = new CodeMemberProperty();
        property.Attributes = MemberAttributes.Public | MemberAttributes.
            Final;
        property.Type = new CodeTypeReference(attribute.Type);
        property.Name = this.LanguageProvider.CreateEscapedIdentifier(
            attribute.Name);

        property.GetStatements.Add(
          new CodeMethodReturnStatement(
            new CodeFieldReferenceExpression(
              new CodeThisReferenceExpression(),
              this.FieldName(attribute.Name))));

        property.SetStatements.Add(
          new CodeAssignStatement(
            new CodeFieldReferenceExpression(
              new CodeThisReferenceExpression(),
              this.FieldName(attribute.Name)),
            new CodePropertySetValueReferenceExpression()));

        this.LanguageProvider.GenerateCodeFromMember(
          property, this.generationWriter, null);
    }
    // ...
}
```

Listing 3.2: CodeDOM Template zum Schreiben von Instanzvariablen (Sych)

An diesen beiden sehr einfachen Beispielen kann man schon erkennen, wie komplex die Handhabung des CodeDOM Template Systems ist. Es bleibt festzustellen, dass CodeDOM, das sehr viel mächtigere Tool von beiden ist. Für unsere Zwecke und die gewünschten Ziele bietet es aber zu viel Funktionalität und ist darum ungeeignet.

3.2.4. Fazit

Nach Betrachtung der Vor- und Nachteile der beiden Template Technologien und der Rahmenbedingungen des Projektes wird für die vorliegende Arbeit der Einsatz eines T4-Templates gewählt. Unser Zielsystem und die Sprache (C#) sollen sich laut Anforderung nicht verändern. Die Templates sollen wartungsfreundlich und gut verständlich sein. All das spricht für den Einsatz von T4.

3.3. Referenzierung

Die erste Frage die sich stellt ist, wie man zwischen einem UML-Shape, einem Projekt, einem Projektitem oder einer beliebigen Datei eine Verbindung (Referenz) erstellen kann. Dazu bietet sich die Funktionalität des IElement-Interfaces an, das alle UML-Shapes implementieren kann. Es bietet die Möglichkeit, im Shape eine oder mehrere Referenzen zu speichern und diese später wieder abzurufen.

```
1  /// <summary>
2  /// Methode um eine Referenz in einem IElement zu speichern
3  /// </summary>
4  /// <param name="element">Das Element, in dem die Referenz gespeichert
       werden soll</param>
5  /// <param name="name">Name der Referenz, dient als Identifier</param>
6  /// <param name="value">Der Wert der Referenz</param>
7  /// <param name="allowMultiple">Mehere Referenzen mit gleichen Identifier
       erlaubt</param>
8  /// <returns>Eine Referenz</returns>
9  public static IReference AddReference(
10   this IElement element,
11   string name,
12   string value,
13   bool allowMultiple
14 )
```

```
15
16 /// <summary>
17 /// Methode um eine Liste von Referenzen eines IElement abzurufen
18 /// </summary>
19 /// <param name="element">Das Element, in dem die Referenz abgerufen
       werden soll</param>
20 /// <param name="name">Name der Referenz, dient als Identifier</param>
21 /// <returns>Eine Liste mit Referenzen</returns>
22 public static IEnumerable<IReference> GetReferences(
23   this IElement element,
24   string name
25 )
26
27 /// <summary>
28 /// Methode um alle Referenzen in einem IElement zu löschen
29 /// </summary>
30 /// <param name="element">Das Element, in dem die Referenz gelöscht
       werden soll</param>
31 /// <param name="name">Name der Referenz, dient als Identifier</param>
32 public static void DeleteAllReference(
33   this IElement element,
34   string name
35 )
```

Listing 3.3: Referenzierung mit dem IElement-Interface

Mit Hilfe dieser 3 Funktionen kann man nun Referenzen zu anderen Objekten in einem Modellierungsprojekt herstellen und verwalten, indem zum Beispiel der Pfad zu einer Datei als Referenz gespeichert wird, um ihn später bei Bedarf wieder abzurufen. Projekte unter Visual Studio bieten noch eine weitere Möglichkeit, sie zu identifizieren, ihre GUID (Globally Unique Identifier). In Visual Studio haben nur Solutions und Projekte eine GUID, Projektitems und Files nicht. Somit können diese nur mit Hilfe ihres Namens identifiziert werden, was bedeutet, dass hier bei der Synchronisation besonderes Augenmerk darauf gelegt werden muss, um keine Referenzen zu verlieren und tote Links zu erhalten.

Jetzt, wo es eine Möglichkeit gibt, UML-Shapes mit beliebigen Objekten zu referenzieren, stellt sich eine weitere Frage. Kann man diese Referenzierung auch in die entgegengesetzte Richtung erhalten? Kann man in einem Projekt oder einem Projektitem Referenzen zu beliebigen Objekten speichern? Hier lautet die Antwort nein. Die Referenzierung in einer Visual Studio Erweiterung ist eine unidirektionale Referenzierung. Um von einem Projekt auf die referenzierende Komponente im UML-Diagramm zu schließen, muss man alle Komponenten

abfragen, ob sie eine Referenz auf das aufrufende Projekt besitzen.

Es ist klar, dass hier ein zusätzlicher Workload für das System entsteht, das aber ist sicherer als z.B. nur nach einem gleichen Namen zu suchen, um auf eine Referenz zu schließen.

3.4. Validierung

Bei einer automatischen Codegenerierung ist es besonders wichtig, dass der Entwurf, der in Code umgesetzt werden soll, fehlerfrei ist. Das garantiert eine hohe Codequalität und eine solide Arbeitsgrundlage für alle folgenden Arbeitsschritte. Eine Validierung ist notwendig, um externe Benutzereingriffe zu registrieren. Das könnte zum Beispiel eine Veränderung einer Projektdatei außerhalb von Visual Studio sein oder auch Arbeiten mit deaktivierter Quasar-Erweiterung.

Um all diese Einflüsse zu negieren, werden bei jeder Codegenerierung die UML-Diagramme und alle enthaltenen Elemente validiert.

3.4.1. Validierung auf 3 Ebenen

Es stellt sich nun die Frage, was validiert und wie nach Fehlern gesucht werden soll? Das System stellt eine Validierung auf 3 Ebenen zur Verfügung. Die erste Ebene ist die Komponentenebene. Hier werden alle „UML-Component“ Elemente und deren Beziehungen untersucht.

Validierung auf Komponentenebene

- **„Namenskonvention“:** Prüfung, ob der eingegebene Name den C#-Konventionen für Projektnamen entspricht. Das muss sichergestellt werden, da im UML-Designer auch Sonderzeichen oder sonstige reservierte Namen erlaubt sind
- **„Namenslänge“:** Ist der eingegebene Name einer Komponente zu lang, verletzt er damit die maximale Länge von C#-Projektnamen
- **„Referenz“:** Prüfung der Referenzierung, ist die vorhandene Referenzierung gültig oder ist sie ein toter Link
- **„Port und Abhängigkeiten“:** Besitzt die Komponente angebotene oder eingehende Ports, Prüfung auf Zyklen bei Abhängigkeiten zwischen verschiedenen Komponenten

Validierung auf Klassenebene

Nach der erfolgreichen Validierung auf Komponentenebene wird die darunterliegende Klassenebene geprüft. Hierzu folgt der Validierungsalgorithmus den gespeicherten Referenzen in den Komponenten zu den Klassendiagrammen und prüft deren Inhalt.

- **„Vollständigkeit“:** Ist von jedem Quasarelement eines oder mehrere vorhanden. Das heißt, es muss genau eine „Quasar-Komponente“ vorhanden sein. Es muss mindestens ein „Quasar-Anwendungsfall“, ein „Quasar-Verwalter“ und eine „Quasar-Entität“ vorhanden sein. Außerdem ein „Quasar-Interface“, wenn es den dazugehörigen angebotenen Port im UML-Diagramm gibt
- **„Abhängigkeiten“:** Sind alle Abhängigkeiten modelliert, bestehen Interface-Vererbungen
- **„Namenskonvention“:** Prüfung ob der eingegebene Name den C#-Konventionen für Projektitemnamen entspricht. Das muss sichergestellt werden, da im UML-Designer auch Sonderzeichen oder sonstige reservierte Namen erlaubt sind

Validierung des Dateisystems

Die Prüfung des UML-Entwurfs ist jetzt abgeschlossen. Nun ist es aber dennoch wichtig das Dateisystem zu prüfen, um sicherzustellen, dass keine der zu erzeugenden Objekte schon vorhanden ist. Dies ist die letzte Ebenen auf der der Entwurf validiert wird.

- **„Prüfung der Pfade“:** Prüfung, ob Projektpfad noch nicht vorhanden ist
- **„Prüfung der Dateien“:** Prüfung, ob Dateien noch nicht vorhanden sind

Wenn ein Validierungsschritt fehlgeschlagen ist, wird dem Benutzer eine Fehlermeldung ausgegeben, und eine Liste mit den vorhandenen Fehlern wird angezeigt. Sie müssen zuerst beseitigt werden, bevor er fortfahren kann.

3.4.2. Entwurf Validierung

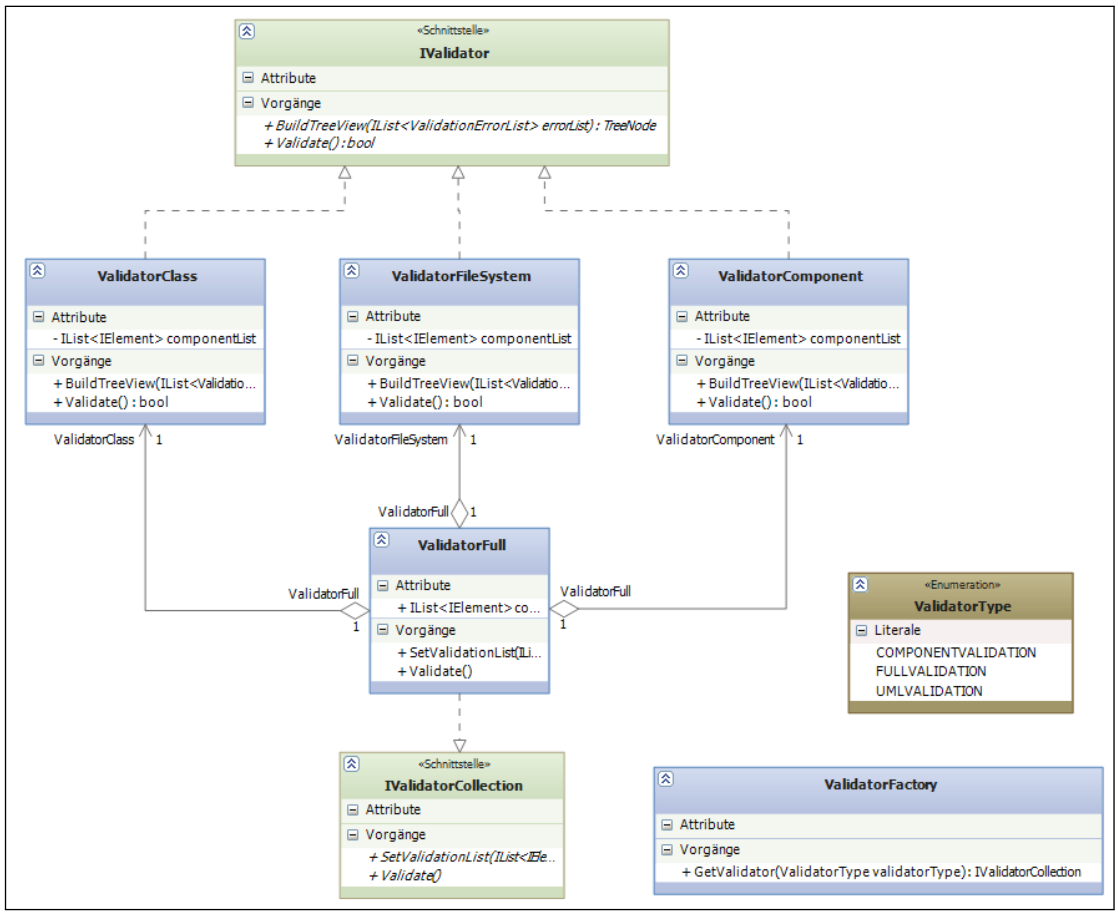


Abbildung 3.2.: UML-Entwurf Validierung

Die Validatoren für die 3 verschiedenen Ebenen werden durch die Klassen „ValidatorComponent, ValidatorClass und ValidatorFileSystem“ repräsentiert. Sie alle implementieren das Interface „IValidator“, um verschiedene Funktionen sicherzustellen. Des Weiteren gibt es eine FaktoryKlasse, die für verschiedene Fälle eine Sammlung von Validatoren zurückgibt. So wird es ermöglicht, weitere Validierungsfälle zu erstellen, bei denen unterschiedliche Validatoren zum Einsatz kommen. Im obigen Beispiel den „ValidatorFull“, der eine vollständige Prüfung des Entwurfes durchführt. Er beinhaltet 3 einzelne Validatoren, die er aufruft und auf Fehler, die auftreten, reagiert. Die Validierung in den einzelnen Ebenen wird mit Hilfe des

Command-Pattern realisiert. Genauer wird darauf in Abschnitt 4.2 „Validator-Commands als Command-Pattern“ eingegangen.

3.5. Generierung

Auch der Code-Generator wird, ähnlich wie die Validierung, in mehrere Schritte unterteilt. Erstens den Komponentengenerator, dieser setzt UML-Components in Code um. Das bedeutet, dass ein neues Projekt (Klassenbibliothek) erzeugt und in die Solution eingebunden wird. Gegebenenfalls werden noch einige Verweise wie zum Beispiel ein OR-Mapper, ein Logging-framework oder sonstige Bibliotheken mit eingebunden.

Der zweite Teil der Generierung ist die Erstellung der eigentlichen Codeskeletons. Hierbei ist es zunächst sinnvoll, für die verschiedenen Quasar-Elemente jeweils einen eigenen Generator zu erstellen, um auf die verschiedenen Anforderungen bei jedem einzelnen Element reagieren zu können.

Prüfungen von Vorbedingungen müssen in den Generatorklassen nicht mehr getätigt werden. Vor jedem Aufruf des Generators wird eine komplette Validierung durchgeführt, die erst vollständig und fehlerfrei durchlaufen werden muss.

3.5.1. Entwurf Generierung

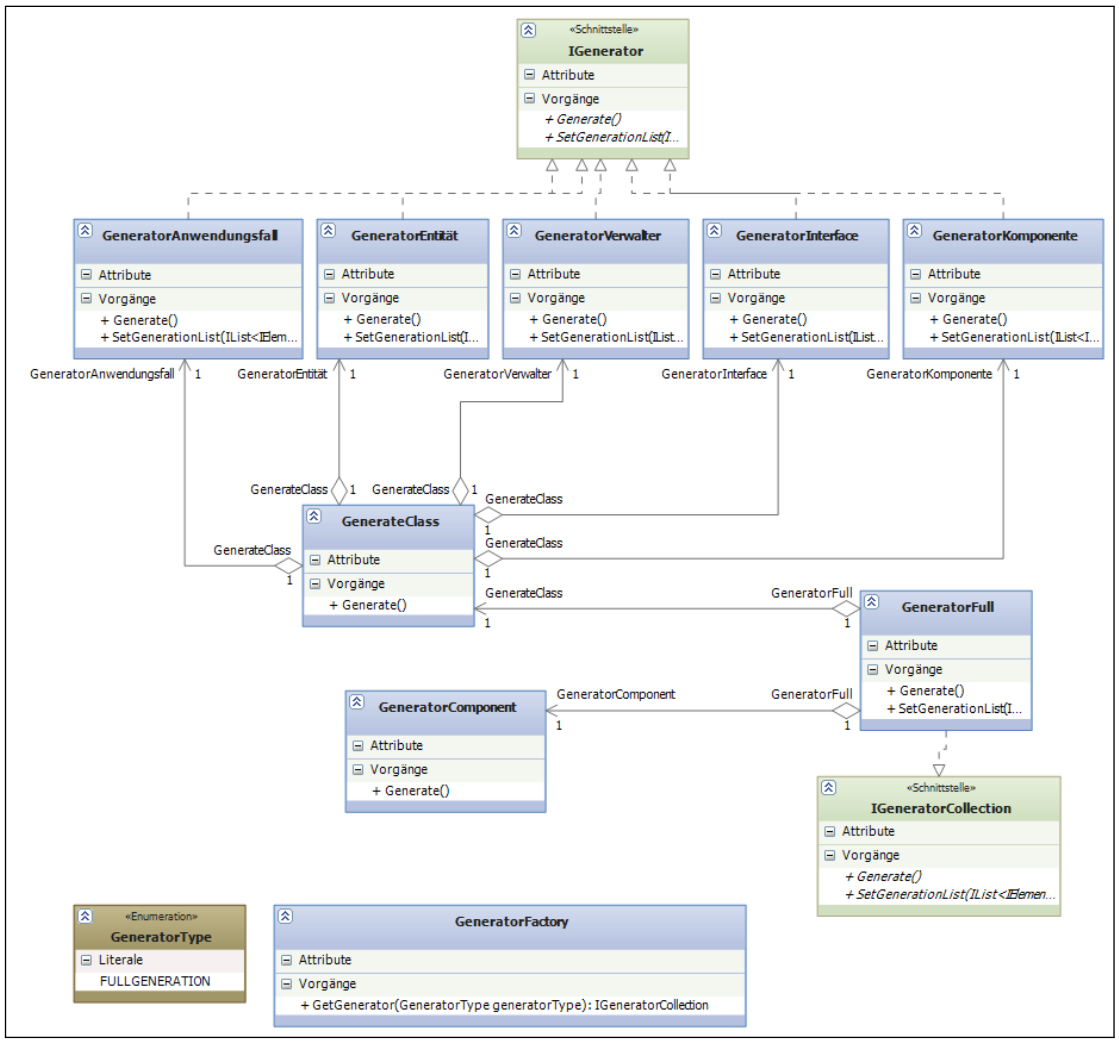


Abbildung 3.3.: UML-Entwurf Generator

Der Code-Generator besteht anfänglich aus der Factoryklasse. Sie erzeugt einen neuen Generator. Im UML-Beispiel wird ein „FullGenerator“ erzeugt. Dieser beinhaltet einen Generator für Komponenten und einen Generator für Klassen. Der Komponentengenerator erzeugt wie schon oben erwähnt die Projekte, der Klassengenerator die Codeskeletons der einzelnen Quasar-Elemente. Der Klassengenerator beinhaltet die einzelnen Instanzen für die verschiedenen Quasartypen, die bei Bedarf aufgerufen werden.

Die Implementierung eines Klassengenerators besteht wiederum aus dem Code-Generator und der dazugehörigen Template-Datei. Der genaue Aufbau und die Implementation wird in Abschnitt 4.4 „Code Generator - Quasar Anwendungsfall“ ausführlich dargestellt.

3.6. Synchronisation

Bei der automatischen Pflege von Entwurf und Implementation stellt sich die Frage, in welchem Rahmen die Synchronisation automatisiert vor sich gehen soll und welche Schritte manuell vom Benutzer durchgeführt werden müssen? Hierbei ist zum einen die Benutzerfreundlichkeit ein wichtiger Aspekt, aber auch die Performance des Systems und der entstehende Workload sind zu berücksichtigen.

Die Synchronisation zwischen dem UML-Modell und dem Sourcecode ist auf verschiedene Ebenen unterteilt. Es gibt hier zum einen die Komponentenebene, die mit der Klassenbibliothek synchronisiert werden muss. Zum anderen Klassen und Interfaces, die mit den Projektitems einer Klassenbibliothek synchronisiert werden müssen. Weiterhin müssen die UML-Diagramme untereinander gepflegt werden.

Es gibt zwei verschiedene Ansätze, wie man die Synchronisationsrichtungen realisieren kann. Es stehen die unidirektionale Synchronisation (in eine Richtung) und die bidirektionale Synchronisation (in beide Richtungen gleichzeitig) zur Verfügung. Ich habe mich für die unidirektionale Synchronisation entschieden. Der Grund ist, dass bei der unidirektionalen Synchronisation klar feststeht, welche Elemente und Codeteile gesichert und verglichen werden sollen. Dies ist bei der bidirektionalen Synchronisation nicht der Fall. Es muss entschieden werden, welche Seite zuerst an der Reihe ist. Was geschieht, wenn Fehler auftreten und ein Rollback notwendig ist? Außerdem ist die Gefahr eines Zyklus vorhanden, wenn immer wieder auf die Änderungen auf der einen oder anderen Seite reagiert wird.

3.6.1. Synchronisation UML-Komponente mit Klassenbibliothek und Klassendiagramm

Zunächst wird festgelegt, in welchem Rahmen eine automatische Synchronisation ausgelöst wird und in welchen Fällen eine manuelle Synchronisation von Nöten ist und warum. Aus diesen Erkenntnissen kann man später Event-Handler und Funktionen ableiten, die zu implementieren sind. Dabei beginnt man auf Komponentenebene bzw. auf Projektebene im Sourcecode.

Hinzufügen einer UML-Komponente

Da im Normalfall eine Komponente im Entwurf eine Innensicht hat, ist es für den Benutzer sehr hilfreich, wenn diese automatisch generiert wird. Ich halte es für sehr wichtig, dass beim Anlegen einer Komponente im UML-Diagramm, ein dazugehöriges UML-Klassendiagramm erstellt wird, das mit den grundlegenden Bestandteilen eines Quasar-Entwurfes gefüllt und in der Komponente eine Referenz auf dieses Diagramm gespeichert wird. Eine automatische Generierung einer Klassenbibliothek halte ich hier für nicht sinnvoll, da der Benutzer sich noch im Entwurfsstadium befindet.

Es wird in dieser Phase oft vorkommen, dass Komponenten aus dem Diagramm entfernt oder geändert werden, was zu übermäßigem Workload führen würde. Die Synchronisation einer neuen Komponente muss also manuell durchgeführt werden, wenn schon andere Projekte in der Solution vorhanden sind oder beim ersten Erstellen mit dem Codegenerator. Hierfür wird dem Benutzer natürlich eine Routine zur Verfügung gestellt, die er nur noch manuell auslösen muss.

Umbenennen einer UML-Komponente

Beim Umbenennen einer UML-Komponente gilt es zwei Fälle zu betrachten. Zum einen den Fall einer Komponente, die noch keine Referenz auf eine Klassenbibliothek besitzt, zum anderen den Fall, dass eine solche Referenz vorhanden ist. Wenn keine Referenz vorhanden ist, sollte das System prüfen, ob schon eine Komponente mit dem neuen Namen vorhanden ist und ggf. den Vorgang abbrechen.

Des Weiteren wird, wenn vorhanden, das referenzierende Klassendiagramm auch mit umbenannt. Wenn nun auch noch eine GUID-Referenz auf eine existierende Klassenbibliothek vorhanden ist, so wird diese auch mit umbenannt. Hierbei ist zu beachten, dass der Verzeichnisname nicht umbenannt wird, sondern nur die Projektdatei.

Löschen einer UML-Komponente

Beim Löschen einer Komponente wird eine automatische Synchronisation für nicht sinnvoll erachtet. Beim endgültigen Entfernen einer Komponente wird nur das dazugehörige Klassendiagramm gelöscht. Die dazugehörigen Sourcecodefragmente bleiben erhalten. Hier könnte man natürlich sagen, wenn eine Komponente endgültig gelöscht wird, will man sie auch später nicht mehr nutzen. Ich habe mich trotzdem so entschieden, hier nur eine manuelle Löschung zuzulassen.

Hinzufügen eines Projekts im Sourcecode

Wenn man ein neues Projekt in den Sourcecode einfügt, ist es nicht zwingend notwendig, dieses in den UML-Entwurf zu übernehmen. Es könnte sich z.B. um ein Projekt handeln, dass nur Hilfsklassen oder sonstigen Code beinhaltet, der nicht direkt in einem Quasar-Entwurf dokumentiert werden muss. Auch hierfür wird dem Benutzer eine Routine zur Verfügung gestellt, die er manuell auslösen kann, um das gesamte Projekt und alle dazugehörigen Projektitems zu übertragen.

Umbenennen eines Projekts im Sourcecode

Bei der Umbenennung eines Projektes habe ich mich für eine automatische Synchronisierung entschlossen. Es wird geprüft, ob ein Projekt mit gleichem Namen vorhanden ist. Anschließend wird die Komponente gesucht, die auf dieses Projekt referenziert. Dieses wird dann umbenannt. Sollte keine Referenzierung vorhanden sein, wird nur das Projekt umbenannt.

Löschen eines Projekts im Sourcecode

Beim Löschen eines Projekts wird nur die Referenz im UML-Diagramm gelöscht und nicht das UML-Shape. Außerdem werden alle Referenzen der Projektitems gelöscht. Es könnte zum Beispiel der Fall auftreten, dass der Benutzer Fehler in diesem Projekt hat, es löschen will und von Grund auf neu erzeugen möchte. Darum bleibt das Shape erhalten und muss notfalls manuell gelöscht werden.

Ändern einer UML-Komponente/Projekts im Sourcecode

Wenn eine UML-Komponente eine Referenzierung besitzt, sollte der Benutzer die Möglichkeit haben, diese auch mit dem bestehenden Sourcecode und umgekehrt zu synchronisieren. Hierbei muss der Benutzer manuell eine Routine auslösen, in der erst auf Komponentenebene, sprich Verweise und sonstiges, eine Synchronisierung durchgeführt wird und anschließend auf Klassenebene, welche im weiteren Verlauf noch erörtert wird.

Zusammenfassung

UML-Komponente erstellen	$\xrightarrow{manuell}$ + alle Klassen	Projekt
UML-Komponente	$\xleftarrow{manuell}$	**Projekt anlegen**
UML-Komponente umbenennen	$\xrightarrow[\text{mit Klassendiagramm}]{automatisch}$	Projekt
UML-Komponente	$\xleftarrow[\text{mit Klassendiagramm}]{automatisch}$	**Projekt umbenennen**
UML-Komponente löschen	$\xrightarrow{manuell}$	Projekt
UML-Komponente	$\xleftarrow[\text{Referenz löschen}]{automatisch}$	**Projekt löschen**
UML-Komponente ändern	$\xrightarrow{manuell}$ + alle Klassen	Projekt
UML-Komponente	$\xleftarrow[\text{+ alle Projektitems}]{manuell}$	**Projekt löschen**

Tabelle 3.2.: Synchronisationsverhalten von UML-Komponenten und Projekten

3.6.2. Synchronisation UML-Klassendiagramm mit Projektitems

In der zweiten Synchronisationsebene befindet man sich auf Klassenebene. Hier geht es darum, die einzelnen Projektitems und Klassen untereinander zu synchronisieren.

Hinzufügen einer UML-Klasse/Interface

Ähnlich wie bei der Komponente habe ich mich auch hier entschlossen, nur eine manuelle Synchronisation dem Benutzer zur Verfügung zu stellen. Klassen und Interfaces werden in der Entwurfsphase so oft geändert, umbenannt oder gelöscht, dass hier ein zu großer Overhead entstehen würde, wenn man jedesmal eine neue Klasse direkt in Sourcecode übertragen will. Es gibt für die Synchronisation wieder eine Routine, die der Benutzer auslösen kann. Nach der manuellen Synchronisation besitzt die UML-Klasse/Interface eine Referenz auf die erzeugte Datei.

Umbenennen einer UML-Klasse/Interface

Analog zum Umbenennen einer Komponente gibt es hier wieder 2 Fälle zu betrachten. Vom Synchronisationsverhalten sind beide analog zu betrachten. Darum werden sie hier nicht weiter ausgeführt.

Löschen einer UML-Klasse/Interface

Wie bei der UML-Komponente, habe ich auch hier keine automatische Synchronisation vorgesehen. Der Benutzer muss, wenn nötig, das vorhandene Projektitem und die UML-Klasse von Hand löschen.

Die Synchronisation beim Hinzufügen, Umbenennen und Löschen eines Projektitems im Sourcecode geschieht hier analog wie bei einem Projekt.

Ändern einer UML-Klasse/Interface / Projektitems

Der wohl komplexeste Synchronisationsschritt ist die Erfassung einer Änderung im Inhalt einer Klasse. Dabei reicht es nicht aus nur den Namen oder eine Verknüpfung zu aktualisieren, sondern der Inhalt der Sourcecode-Datei oder des UML-Shapes muss betrachtet und verglichen werden. Wenn notwendig, muss das System Codeteile sichern und wieder einspielen. Da dies ein höchst komplexer Vorgang ist, der nicht bei jeder Veränderung ausgelöst werden soll, sondern erst nach Abschluss, wird auch hier dem Benutzer eine Routine zur Verfügung gestellt, die er manuell auslösen kann.

Zusammenfassung

UML-Klasse erstellen	$\xrightarrow{manuell}$	Projektitem
UML-Komponente	$\xleftarrow{manuell}$	**Projektitem anlegen**
UML-Klasse umbenennen	$\xrightarrow{automatisch}$	Projektitem
UML-Komponente	$\xleftarrow{automatisch}$	**Projektitem umbenennen**
UML-Klasse löschen	$\xrightarrow{manuell}$	Projekt

UML-Komponente	$\xleftarrow[\text{Referenz löschen}]{automatisch}$	**Projektitem löschen**
UML-Klasse ändern	$\xrightarrow[\text{alle Inhalte}]{manuell}$	Projektitem
UML-Komponente	$\xleftarrow[\text{alle Inhalte}]{manuell}$	**Projektitem löschen**

Tabelle 3.3.: Synchronisationsverhalten von UML-Klassen/Interfaces und Projektitems

3.6.3. Welche Inhalte müssen synchronisiert werden

Die Synchronisierung einer UML-Klasse oder eines UML-Interfaces ist bei dem gesamten Vorgang der schwierigste Teil. Bei vielen automatischen Codegeneratoren geht bei der Änderung des Modells die erzeugte Sourcecode-Datei verloren. Bei den integrierten Generatoren in Visual Studio ist es nicht anders. Sei es bei T4 oder anderen Generatoren, es wird immer darauf verzichtet, Code des Benutzers zu sichern.

Durch die in dieser Arbeit erstellte Erweiterung soll dem Benutzer eine effektive Handhabung ermöglicht werden. Es muss sichergestellt werden, dass möglichst jede Änderung in den Codeskeletons oder dem UML-Designer ohne Verluste ineinander übertragen werden kann. Es stellt sich natürlich die Frage, welche Teile aus den einzelnen Projektitems überhaupt gesichert werden sollen?

- Quasar-Interface
  - In den Codeskeletons eines Quasar-Interfaces steckt keinerlei Implementation oder Logik. Es ist eine reine Beschreibung der Schnittstelle.
  - Synchronisierung Codesekeleton -> UML-Shape: Alle vorhandenen Funktionen im UML-Shape werden gelöscht und durch die im Codeskeleton vorhandenen Funktionen ersetzt.
  - Synchronisierung UML-Shape -> Codesekeleton: Da keine Implementierung vorhanden ist, wird das bestehende Interface gelöscht und anschließend neu erzeugt.
- Quasar-Komponente
  - In den Codeskeletons einer Quasar-Komponente steckt keine Implementierung oder Logik. Die verschiedenen Funktionen werden nur an die entsprechenden Anwendungsfälle delegiert. Es existieren verschiedene Instanzvariablen, die definiert und initialisiert werden. Außerdem kann der Konstruktor angepasst worden sein.
  - Synchronisierung Codesekeleton -> UML-Shape: Es werden die Abhängigkeiten und Vererbungen geprüft und wenn nötig erneuert oder aktualisiert.

  - Synchronisierung UML-Shape -> Codesekeleton: Bestehende Instanzvariablen, der Konstruktor und der Inhalt des Konstruktors werden gesichert. Anschließend wird der neue Codeskeleton erzeugt und ein Abgleich mit den gesicherten Codeelementen durchgeführt. Wenn Differenzen auftreten sollten, werden diese behoben, indem fehlende Codeelemente eingefügt werden.
- Quasar-Verwalter
  - In den Codeskeletons eines Quasar-Verwalters steckt benutzerspezifische Implementierung. Hier werden die verschiedenen Funktionen implementiert, um eine Entität später verwalten zu können. Im Vorfeld einer Synchronisation ist nicht abzuschätzen, wieviel Code der Benutzer in diese Datei eingefügt hat. Es bietet sich darum hier an, eine partielle Klasse zu definieren, die bei der Synchronisation gerettet wird.
  - Synchronisierung Codesekeleton -> UML-Shape: Es werden die Abhängigkeiten und Vererbungen geprüft und wenn nötig erneuert oder aktualisiert.
  - Synchronisierung UML-Shape -> Codesekeleton: Inhalt der partiellen Klasse wird gesichert. Die alte Datei wird gelöscht und danacht wieder neu erzeugt. Der gesicherte Inhalt wird eingefügt.
- Quasar-Entität
  - In den Codeskeletons einer Quasar-Entität unterliegt fast der gesamte Code den Änderungen des Benutzers. Es ist sinnvoll einen solchen Codeskeleton zu unterteilen. Es gibt zum einen die Instanzvariablen, als Weiteres die Mapping-Klasse und außerdem noch Teile benutzerspezifischen Codes.
  - Synchronisierung Codesekeleton -> UML-Shape: Bestehende Instanzvariablen werden mit den vorhandenen Instanzvariablen abgeglichen. Es werden Return-Value, Mappingtyp und weitere Bestandteile verglichen und wenn nötig angepasst.
  - Synchronisierung UML-Shape -> Codesekeleton: Zuerst werden die Instanzvariablen gesichert. Anschließend die Mappingklasse und eine partielle Klasse, in der der Benutzer seine eigenen Codeelemente unterbringen kann. Die alte Datei wird gelöscht und neu erzeugt. Nun wird geprüft, ob sich die gesicherte Mappingklasse von der neu erzeugten unterscheidet. Unterschiede werden übernommen. Anschließend wird der Inhalt der partiellen Klasse wieder eingefügt.
- Quasar-Anwendungsfall
  - In den Codeskeletons eines Quasar-Anwendungsfalles befindet sich die eigentliche Geschäftslogik. Hier werden alle Funktionen, die in einem Interface definiert worden sind, implementiert. Es ist wichtig, dass diese Implementierungen gesichert werden.

- Synchronisierung Codesekeleton -> UML-Shape: Es werden die Abhängigkeiten und Vererbungen geprüft und wenn notwendig erneuert oder aktualisiert.
- Synchronisierung UML-Shape -> Codesekeleton: Die eigentliche Implementierung wird in einer partiellen Klasse durchgeführt, die zuerst gesichert wird. Anschließend wird die Datei gelöscht und wieder neu erzeugt. Der Inhalt der partiellen Klasse wird in die Datei eingefügt.

3.6.4. Aufbau der Dateien bei der Synchronisierung

Als Beispiel für den Aufbau und das Aussehen einer solchen Datei nach der Synchronisation wird das Codeskeleton einer Quasar-Anwendungsfall-Datei benutzt.

```
1  //------------------------------------------------------------------------
2  // <auto-generated>
3  //     This code was generated by a tool.
4  //     Changes to this file will be lost if the code is regenerated.
5  //
6  //       WICHTIG !!!!
7  //   Die Klassenstruktur darf nicht verändert werden.
8  //   Dass heisst:
9  //   1.) User Klasse
10 //   2.) Base Klasse
11 //
12 // </auto-generated>
13 //------------------------------------------------------------------------
14 using System;
15 using System.Collections.Generic;
16 using System.Linq;
17 using System.Text;
18
19 namespace AnwendungskernAlpha
20 {
21
22     /// <summary>
23     /// Der FilmverwaltungAnwendungsfall stellt die zentrale
           Geschäftslogik
24     /// der Komponente zur Verfügung
25     ///
26     /// Die Funktionen der Basisklasse aus den Interfaces
27     /// können hier implementiert werden.
28     /// Diese Änderungen werden auch bei der Syncronisation übernommen.
29     /// Änderungen in der Basisklasse gehen verloren !!!
```

```
30      /// </summary>
31      ///
32      /// <see cref="IFilmverwaltung"/>
33      internal class FilmverwaltungAnwendungsfall :
            FilmverwaltungAnwendungsfallBase
34      {
35          private FilmverwaltungVerwalter filmverwaltungVerwalter = null;
36
37          internal FilmverwaltungAnwendungsfall(IPersistenceManager
                persistenceManager)
38          {
39              filmverwaltungVerwalter = new FilmverwaltungVerwalter(
                    persistenceManager);
40
41          }
42
43        #region Implementierung der BasisKlasse
44        /*
45         * Beispiel:
46         *
47         * In der Basisklasse FilmverwaltungAnwendungsfallBase befindet
                sich folgende Funktion:
48         * public virtual void Operation1()
49         * {
50         *      throw new NotImplementedException();
51         * }
52         *
53         * nun können wir diese hier überladen und die Logik implementieren
                .
54         * public override void Operation1()
55         * {
56         *      do something.........
57         * }
58         *
59         */
60        public override FilmEntität GetFilmByID(int FilmID)
61        {
62            filmverwaltungVerwalter.GetFilmByID(FilmID);
63        }
64
65        #endregion
66      }
67
```

```
68      /// <summary>
69      /// Base Klasse, bitte hier nichts ändern !!!
70      /// Änderungen gehen verloren !!!
71      /// </summary>
72      internal class FilmverwaltungAnwendungsfallBase: IFilmverwaltung
73      {
74        #region Implementation of IFilmverwaltung methods
75
76        /// <summary>
77        /// <see cref="IFilmverwaltung.cs"/>
78        ///
79        /// </summary>
80        public virtual FilmEntität GetFilmByID(int FilmID)
81        {
82            throw new NotImplementedException();
83        }
84
85        /// <summary>
86        /// <see cref="IFilmverwaltung.cs"/>
87        ///
88        /// </summary>
89        public virtual FilmEntität GetFilmByName(string FilmName)
90        {
91            throw new NotImplementedException();
92        }
93
94        #endregion
95      }
96  }
```

Listing 3.4: Aufbau eines Codeskeletons einer Quasar-Anwendungsfall-Datei

In der Klasse „FilmverwaltungAnwendungsfallBase“ wird das Interface „IFilmverwaltung“ implementiert und die Funktionsrümpfe werden erzeugt. Beim Aufruf werfen sie eine „NotImplementedException“. Die Klasse „FilmverwaltungAnwendungsfall“ erbt die Basisklasse. Hier kann der Benutzer die einzelnen Funktionen implementieren. Mittels „override“ werden die Funktionen der Basisklasse überschrieben. Beim Aufruf wird die überschriebene Funktion aufgerufen. Sollte für eine Funktion noch keine konkrete Implementierung vorliegen, wirft die Basisklasse eine „NotImplementedException“. Hier sieht man, wie wichtig es ist, Inhalte, die der Benutzer erstellt hat, zu sichern, und sich im Vorfeld eine gute Codestruktur zu überlegen.

3.6.5. Entwurf Synchronisierung

Der Entwurf besteht zunächst aus einer Factoryklasse, die einen der 3 Synchronisator-Collections („SyncronatorFile“, „SyncronatorProject“, „SyncronatorFull“) erzeugt und zurückgibt. Je nachdem in welchem Kontext man sich befindet wird entweder ein „SyncronatorFull“, ein „SyncronatorFile“ oder ein „SyncronatorProject“ erzeugt. Die Aggregationen des „SyncronatorFull“ sind im Diagramm aus Gründen der Übersichtlichkeit nicht dargestellt. Er beinhaltet aber alle 6 einzelnen Synchronisatoren. In den Synchronisatoren, die von „ISyncronator“ erben, ist die eigentliche Logik implementiert.

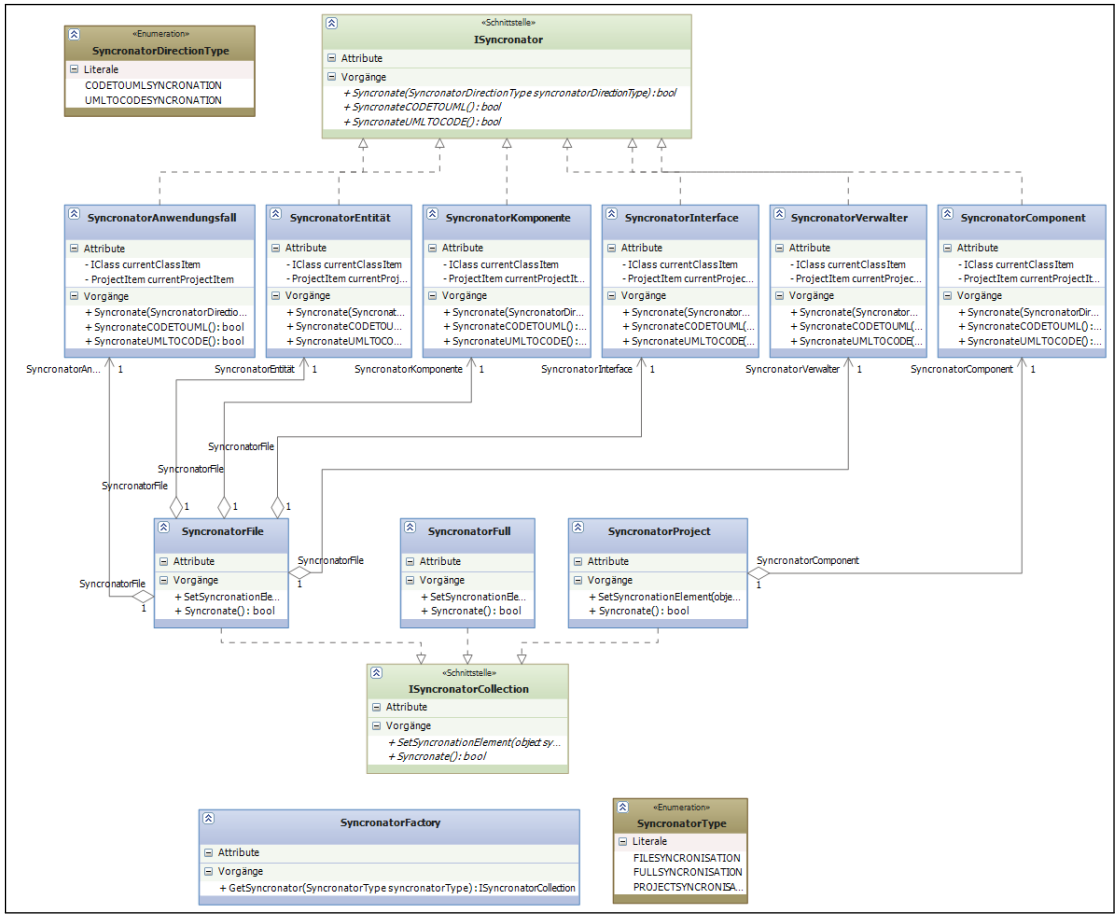


Abbildung 3.4.: UML-Entwurf Synchronisation

3.7. Fachliche Architektur des Systems

3.7.1. MEF-Pakete

Bevor man sich Gedanken über das Design der MEF-Pakete macht, sollte man wissen, dass es 3 verschiedene UML-Designer Erweiterungen gibt, die man in einem MEF-Paket unterbringen kann. Es ist natürlich auch möglich, mehrere dieser Erweiterungen in einem Paket zu verpacken.

- **Command Extension:** Die Command Extension bietet dem Benutzer die Möglichkeit, in einem Kontextmenü eigene Kommandos unterzubringen. Es wird die Möglichkeit zur Verfügung gestellt, jedes einzelne Kommando zu deaktivieren oder den gezeigten Text zu ändern. Es ist nicht möglich Icons, Nmonics oder Shotcuts zu definieren. Eine Verschachtelung auf mehrere Ebenen ist auch nicht vorgesehen (Microsoft (b)).
- **Gesture Extension:** Die Gesture Extension bietet dem Benutzter die Möglichkeit, auf 2 verschiedene Events zu reagieren. Zum einen auf das „Drag & Drop“ Event zum anderen auf ein „DoubleClick“ Event. Das Element, mit dem der Benutzer gerade interagiert steht im Eventkontext zur Verfügung. Dem Benutzer stehen Kollisionsabfragen, verschiedene Mousebuttonzustände und der Diagrammkontext zur Verfügung (Microsoft (a)).
- **Model Validation Extension:** Die Model Validation Extension dient dazu, Events und Rules zu registrieren und einen Container für ihre Ausführung zur Verfügung zu stellen. Es gibt verschiedene Optionen, wann ein Event oder eine Rule ausgelöst werden kann. Zum Beispiel beim Öffnen eines Diagramms, beim Schließen oder beim Speichern (Microsoft (c)).

Mittels Annoations wird Visual Studio mitgeteilt, um welche Erweiterung es sich handelt und in welchem Diagrammkontext diese auszuführen ist.

```
1 [Export(typeof(ICommandExtension))]
2 [ComponentDesignerExtension]
```

Listing 3.5: Beispiel Annoations bei UML-Designer Extensions

UML-Komponentendesigner Erweiterung

Im Entwurf für die Erweiterungen des Komponentendesigners habe ich für alle gängigen Erweiterungen Codeskeletons geschaffen und sie beispielhaft gefüllt, damit in späteren Entwicklungsschritten auch andere Entwickler schnell den Einstieg in die Materie bekommen.

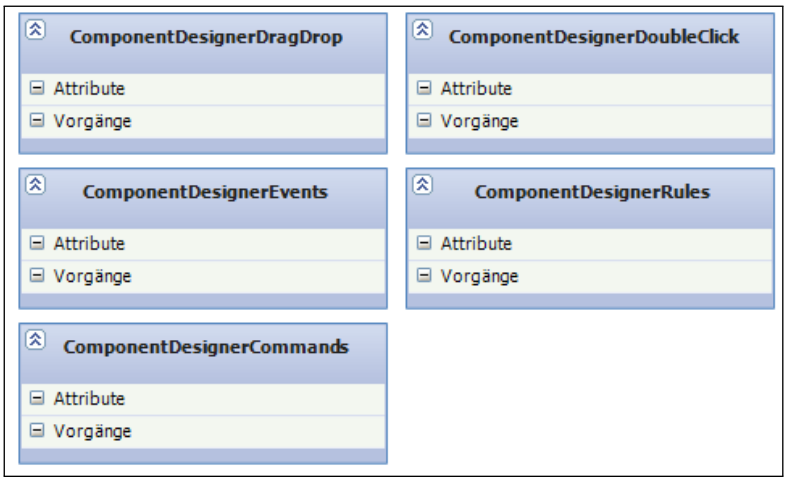


Abbildung 3.5.: UML-Entwurf ComponentDesigner MEF-Paket

Ich habe mich bei diesem Projekt gegen Command Extensions entschieden, weil sie zum einen keine Möglichkeit bieten Icons zu setzen und die verschiedenen Commands zu sortieren. Sie werden einfach nach der Reihenfolge ihrer Paketnamen aufgelistet. Zum anderen kann man sie nicht in verschiedene Submenüs verpacken, wodurch es für den Benutzer sehr unübersichtlich wird. Stattdessen habe ich die Kontexterweiterung mit einem VSPackage realisiert. Somit ist die „ComponentDesignerCommands“ eine Klasse, die nur als Beispiel für andere Entwickler im Projekt verbleibt.

In den „ComponentDesignerEvents“ und „ComponentDesignerRules“ Klassen werden die verschiedenen Events verpackt. Zum Beispiel Rename-Events, Events zum Reagieren auf das Hinzufügen eines Ports oder das Hinzufügen einer Komponente zu registrieren. Bei diesen Klassen handelt es sich bei beiden um Model Validation Extension.

Die Klassen „ComponentDesignerDoubleClick“ und „ComponentDesignerDragDrop“ sind Gesture Extensions. Hier wird zum Beispiel auf den Doppelklick eines Benutzers auf ein Component-Shape reagiert . Wenn dieses Shape eine Referenz auf ein anderes Objekt hat, wird versucht, dieses in Visual Studio zu öffnen.

UML-Klassendesigner Erweiterung

Bei den Erweiterungen für den Klassendesigner gestaltet sich das Bild etwas einfacher. Hier gibt es zum einen die Klasse „ColorClassesByStereotype“. Bei ihr handelt es sich um eine

Model Validation Extension. Sie dient dazu, Änderungen des Stereotypes zu registrieren und mit einem Farbwechsel des UML-Shapes darauf zu reagieren.

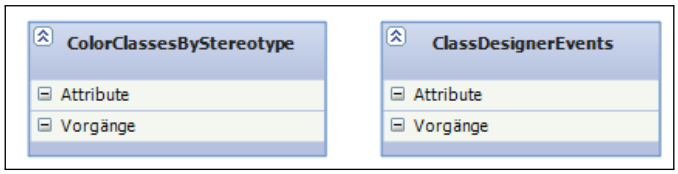


Abbildung 3.6.: UML-Entwurf ClassDesigner MEF-Paket

In der „ComponentDesignerEvents“ Klasse werden Events, zum Beispiel das Reagieren auf ein „Rename“, verpackt. Namensprüfungen können somit realisiert werden. Hierbei handelt es sich auch um eine Model Validation Extension.

3.7.2. VSPackage

Das VSPackage dient dazu, Visual Studio zu erweitern. Da ich mich dazu entschieden habe, Kontexterweiterungen und Menüs nicht als Command Extension zu gestalten, ist eine andere Möglichkeit das VSPackage. Mit dem integrierten grafischen Designer ist es möglich, schnell und effektiv eine Lösung zu erarbeiten.

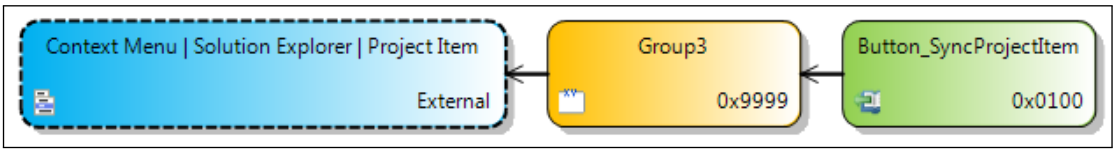


Abbildung 3.7.: Kontextmenüerweiterung Solutionexplorer - Projektitems

Mittels eines Hook-in Points, im Diagramm als „External“ gekennzeichnet, ist es möglich, in das in diesem Punkt spezifizierte Menü oder Kontextmenü eine Erweiterung einzubringen. Anschließend werden Icons und Shortcuts zugewiesen und die verschiedenen Buttons, die Commands entsprechen, mit Code gefüllt.

In einem VSPackage ist es möglich, verschiedene Sichtbarkeiten zu realisieren. Vorgefertigte Einstellungen, wie zum Beispiel ein „NoSolution“-Constraint, erleichtern hier dem Entwickler die Arbeit.

Beispielhaft wird hier eine Kontextmenüerweiterung (Abb. 3.7) und die Erweiterung des Hauptmenüs aufgeführt (Abb. 3.8). Alle anderen erstellten Erweiterungen sind in Analogie zu diesen beiden Beispielen entstanden.

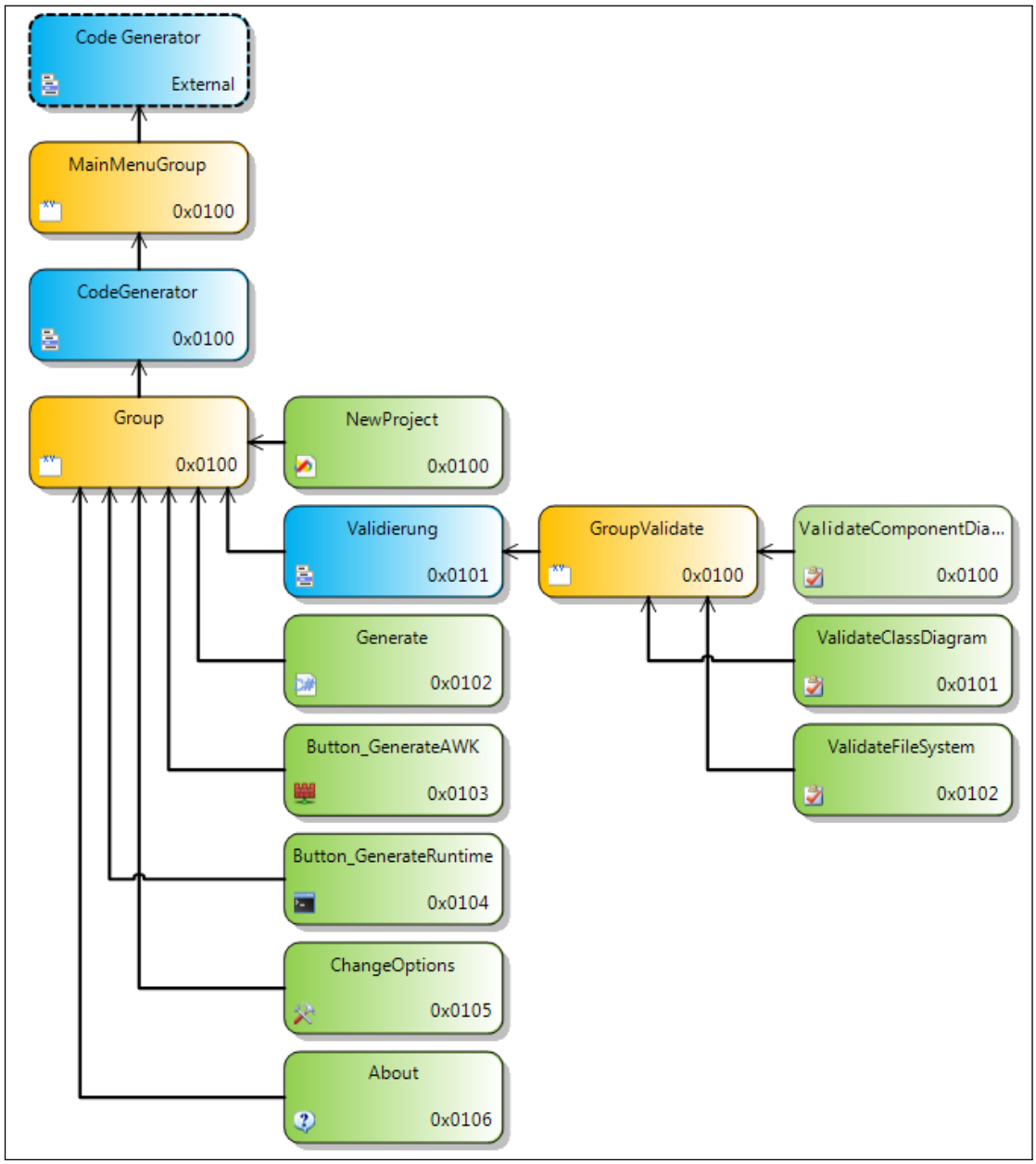


Abbildung 3.8.: Menüerweiterung - Hauptmenü

3.7.3. Gesamtsystem

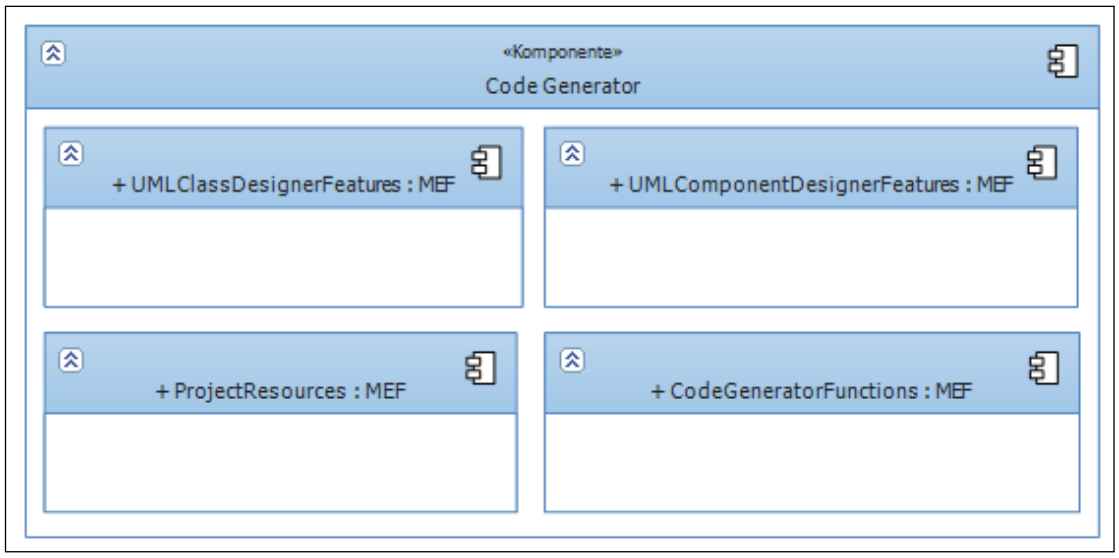


Abbildung 3.9.: UML-Entwurf Gesamtsystem

Nachdem alle Teilsysteme analysiert und beschrieben worden sind, kann man sie zu einem Gesamtsystem zusammensetzen. Das gesamte System besteht aus einer großen Komponente „Code Generator“. Diese Komponente ist ein VSIX-Paket, die dazu dient, die MEF-Pakete und alle Inhalte zu verpacken und zu deployen. In diesem Paket befindet sich auch das VSPackage, wie in Abschnitt 3.7.2 „VSPackage“ beschrieben, um Visual Studio Erweiterungen bereitzustellen. Außerdem beinhaltet es 4 weitere MEF-Pakete.

Die beiden in Abschnitt 3.7.1 „MEF-Pakete“ beschriebenen UML-Designer Erweiterungspakete „UMLClassDesignerFeatures“ und „UMLComponentDesignerFeatures“. Sowie das Paket „ProjectResources“, das verschiedene Bilder, Icons und Resourcendateien bereitstellt.

Das letzte MEF-Paket ist „CodeGeneratorFunctions“. In diesem Paket werden alle weiteren Funktionen implementiert: der Validator (Abschnitt 3.4 „Validierung“), der CodeGenerator (Abschnitt 3.5 „Generierung“), die Synchronisatoren (Abschnitt 3.6 „Synchronisation“), die verschiedenen Commands, Interfaces, T4 Templates und Hilfsklassen. Viele dieser einzelnen Funktionen wurden im Laufe diesen Kapitels bereits beschrieben, darum wird an dieser Stelle darauf verzichtet noch einmal genauer auf sie einzugehen.

4. Implementation

In diesem Kapitel geht es um die Implementation der Visual Studio Erweiterung. Anders wie der Titel es vermuten lässt, wird es hier aber keine komplette Auflistung des Sourcecodes geben, sondern vielmehr wird auf die Schwierigkeiten bei der Implementierung eingegangen. Beispielhaft werden einige besondere Implementationen vorgestellt und erläutert.

Im Vorfeld kann man sagen, dass die Implementierung, also die Umsetzung des Entwurfes in C#-Code, bei diesem Projekt nicht das eigentliche Problem darstellt. Das Hauptproblem, das immer wieder aufgetreten ist, war eine unvollständige, nicht vorhandene oder falsche Dokumentation der Microsoft Visual Studio API.

Bei der Implementierung vieler, augenscheinlich einfacher Anwendungsfälle, sind Probleme aufgetreten, die im Vorfeld nicht absehbar waren. Allgemeine Exceptions, welche von Visual Studio geworfen worden sind, machten das Debuggen und Auffinden solcher Fehler sehr mühsam.

4.1. Problembeispiel: Projektitem hinzufügen

Das folgende kleine Beispiel wird die Problematik verdeutlichen und zeigen, mit welchen Problemen die gesamte Implementierungsphase behaftet war. Der nachfolgende Codeschnipsel zeigt, wie man programmiertechnisch ein Projekt erzeugen und dieses anschließend zu einer Solution hinzufügen kann.

```
1 Project currentProject = ((Solution2)Dte.Solution).AddFromTemplate(
      templatePath, projectPath, components.Name, false);
```

Listing 4.1: Projekt erzeugen und hinzufügen

Laut der Microsoft Dokumentation (Microsoft (f)) soll die angebotene Schnittstelle das hinzugefügte Projekt zurückgeben. Diese Dokumentation ist aber eindeutig falsch.

```
Project AddFromTemplate(
        string FileName,
        string Destination,
        string ProjectName,
        bool Exclusive
)
```

Listing 4.2: AddFromTemplate - Funktionsdefinition

Es wird nur eine Null-Referenz zurückgegeben. Der Grund hierfür ist nicht ersichtlich, in früheren Versionen von Visual Studio wurde das erzeugte Projekt immer korrekt zurückgegeben. Da diese Frage immer wieder im Microsoft-SDK-Entwicklerforum aufgekommen ist, hat Microsoft jetzt eine Note verfasst, um die Entwickler auf diese Abnormalität hinzuweisen. Dieser Hinweis war noch nicht vorhanden, als dieses Projekt in der Implementierungsphase war.

„For Visual Basic and Visual C# projects: The returned Project object has a value of Nothing or null. You can find the created Project object by iterating through the DTE.Solution.Projects collection by using the ProjectName parameter to identify the newly created project."

(Microsoft)

Wenn man auf dieses Projekt im Anschluss zugreifen möchte, muss man es natürlich erst wieder suchen. Hierfür gibt es 2 Möglichkeiten, wie dies erreicht werden kann. Erstens kann man sich iterativ durch alle Projekte bewegen und das neue Projekt mittels seines Namens identifizieren. Die zweite Möglichkeit ist, man implementiert einen Event-Handler, der auf ein Add-Event eines Projekts reagiert. Mit steigender Zahl der Projekte, die durchsucht werden müssen, wird die Suchzeit immer größer. Somit ist der Einsatz eines Event-Handlers durchaus berechtigt. Im vorliegenden Projekt wird die Identifizierung durch Einsatz der letzteren Variante realisiert, da im Vorfeld nicht klar war, wie viel neue Projekte von UML in Code umgesetzt werden sollen.

```
private Project currentProject;

// Event handler - neues Project hinzugefügt
ProjectsEvents csharpProjectEvents = Dte.Events.GetObject("
    CSharpProjectsEvents") as ProjectsEvents;
if (csharpProjectEvents != null)
{
```

```
7           csharpProjectEvents.ItemAdded += new
                _dispProjectsEvents_ItemAddedEventHandler(CSharpProjectAdded)
                ;
8   }
9
10  private void CSharpProjectAdded(EnvDTE.Project newProject)
11  {
12          currentProject = newProject;
13  }
```

Listing 4.3: Eventhandler Projekt erzeugen

Durch Versäumnisse in der Dokumentation, unsaubere Schnittstellen und manchmal nicht ganz nachvollziehbare API-Funktionen kam es bei der Entwicklung leider immer wieder zu Verzögerungen und „unsauberen Code“. Es mussten für viele Probleme Work-Arounds erstellt werden, um Funktionalität, Fehlertoleranz und Robustheit zu gewährleisten.

4.2. Validator-Commands als Command-Pattern

Da die Validierung eines UML-Diagramms nichts weiter als eine Aneinanderreihung verschiedener Commands ist, die abgearbeitet werden, wird hier das „Command-Pattern“ eingesetzt.

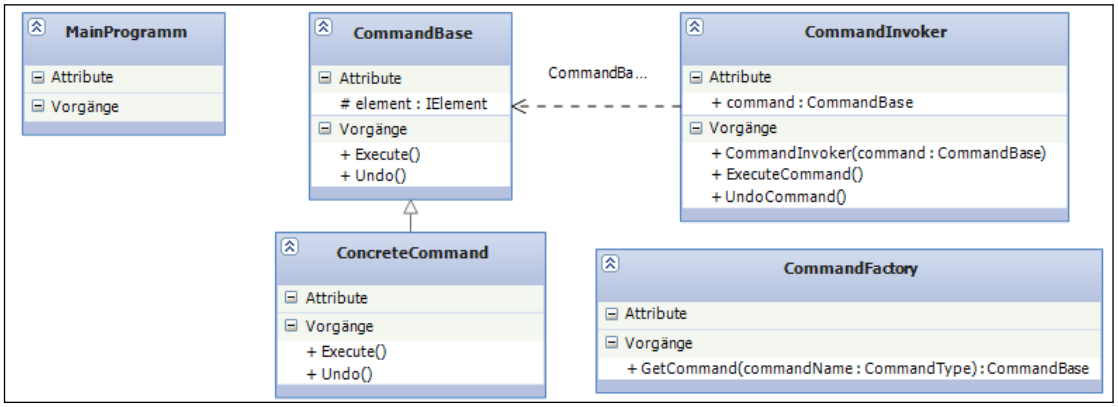


Abbildung 4.1.: Command-Pattern Implementation in C#

Beim klassischen Entwurf des Command-Pattern gibt es zusätzlich noch eine Receiver-Klasse. Diese Funktion übernimmt in diesem Entwurf die konkreten Implementationen des Commands. Zusätzlich wurde noch eine Factory-Klasse implementiert, die ein bestimmtes Command zurückgeben kann. Sie ist nicht Teil des Command-Pattern. Um die Implementierung

etwas zu verdeutlichen, werden einige Code-Ausschnitte gezeigt. Da es bei der Implementation der Validierungskommandos kein Undo-Fall geben wird, wurde auf diesen Teil des Entwurfes verzichtet.

```
public abstract class CommandBase : GenerateHelper
{
        protected IElement element;

        public abstract ValidationError Execute(IElement element);

}
```

Listing 4.4: CommandBase.cs

Die abstrakte Klasse „CommandBase“ wird dazu genutzt, eine gemeinsame Basis zu schaffen. Dies hätte aber auch in einer Schnittstelle passieren können.

```
class CommandInvoker
{
        private CommandBase command { get; set; }

        public CommandInvoker(CommandBase command)
        {
                this.command = command;
        }
        public ValidationError ExecuteCommand(IElement element)
        {
                return this.command.Execute(element);
        }
}
```

Listing 4.5: CommandInvoker.cs

Die „CommandInvoker“ Klasse kapselt den Aufruf vom Objekt. Sie verwendet den Execute-Befehl, um das Kommando abzusetzen.

```
public class ValidatePathCommand : CommandBase
{
        // Constructor
        public ValidatePath()
        {
        }
        public override ValidationError Execute(IElement element)
        {
                string projectPath = Path.GetFullPath(Path.
                    GetDirectoryName(Dte.Solution.FullName));
```

```
10                  if (Directory.Exists(@projectPath + "\\" + ((IComponent)
                        element).Name))
11                          return new ValidationError(ValidationErrorTyp.
                                ERROR, ValidationErrorDescription.
                                COMPONENTERRORDIREXIST, element, @projectPath
                                 + "\\" + ((IComponent)element).Name);
12                  else
13                          return new ValidationError(ValidationErrorTyp.OK)
                                ;
14          }
15  }
```

Listing 4.6: ValidatePathCommand.cs

Eine konkrete Implementation eines Commands benutzt die CommandBase, um die abstrakten Methoden explizit zu implementieren. Als Beispiel dient hier die Prüfung, ob ein Verzeichnis vorhanden ist.

```
1   public enum CommandType
2   {
3           VALIDATENAMECONVENTIONCOMMAND,
4           VALIDATEPATHCOMMAND
5   }
6   public static class CommandFactory
7   {
8           public static CommandBase GetCommand(CommandType command)
9           {
10                  switch (command)
11                  {
12                          case CommandType.VALIDATENAMECONVENTIONCOMMAND:
13                                  return new ValidateNameConventionCommand
                                        ();
14                          case CommandType.VALIDATEPATHCOMMAND:
15                                  return new ValidatePathCommand();
16                          default:
17                                  return null;
18                  }
19          }
20  }
```

Listing 4.7: CommandFactory.cs

Die CommandFactory ist nicht Teil des Command-Pattern. Mit ihr ist es aber möglich, sich schnell und einfach ein neues Command zu erstellen.

4.3. Events und Rules

Einen wichtigen Bestandteil der Implementation stellen Events und Rules dar. Mit ihnen ist es zum Beispiel möglich, den UML-Designer zu erweitern oder auf Handlungen des Benutzers zu reagieren. Sie machen das ganze System interaktiver. Sie ermöglichen es, Automatismen zu gestalten und Befehlsabläufe zu steuern und zusammenzufassen.

4.3.1. Events vs. Rules bei MEF-Erweiterungen

Das Visual Studio Modelling SDK bietet zwei grundsätzlich unterschiedliche Verfahren zum Reagieren auf Veränderungen im Modellstore.

- Ein "**Event-Handler**" ist eine Methode, die nach jeder Änderung des Modells aufgerufen wird. Das Event wird auch ausgelöst, wenn ein Undo- oder Redo-Befehl ausgeführt oder durch eine Transaktion ein Rollback verursacht wird. Hier gilt es vor Abarbeitung des eigentlichen Events zu prüfen, ob sich der Modellstore in einem solchen Zustand befindet. Event-Handler werden normalerweise verwendet, um Änderungen an Komponenten außerhalb des Modells, wie Benutzeroberflächen, Dateien oder Datenbanken, weiterzuleiten (Microsoft (d)).
- Auch eine "**Rule**" wird aufgerufen, wenn eine Änderung im Modell eintritt. Aber im Gegensatz zu Events wird eine Regel nicht ausgelöst, wenn ein Undo- oder Redo-Befehl ausgeführt wird oder durch eine Transaktion ein Rollback erfolgt. Typischerweise werden Regeln verwendet, um Änderungen im Modell zu propagieren oder um die Kohärenz zwischen zwei Teilen des Modells zu erhalten. In diesen Fällen wird eine gespeicherte Kopie des Modells verwendet, um im Falle eines Rollbacks alle Modell-Elemente auf ihre vorherigen Werte zurückzusetzen (Microsoft (g)). Alle Regeln werden in einem transaktionalen Kontext ausgeführt. Das heißt aber auch, das es hier Einschränkungen gibt, welche Funktionalität implementiert werden kann. Das Öffnen von Dateien und Diagrammen ist in diesem Kontext zum Beispiel nicht möglich.

Um Regeln und Events zu definieren, muss man zunächst den Namen der implementierenden Klasse ermitteln, die man überwachen will. Diese Klassen sind in der Datei „Microsoft.VisualStudio.Modeling.Uml.dll" definiert. Um die verschiedenen Schnittstellen zu ermitteln, die man überwachen will, ist es das Einfachste, den Debugger zu starten und die Aufrufhierarchie zu beobachten, die bei verschiedenen Ereignissen erzeugt wird. Für viele Klassen ist es aber sehr offensichtlich, welche Schnittstellen zu überwachen sind, wie zum Beispiel IClass oder IStereotype.

4.3.2. Event: Stereotype hinzufügen und UML-Klasse umfärben

Als ein Beispiel für die Implementierung eines Events wird hier ein einfaches Beispiel aufgeführt. Um eine bessere Usability für den Klassendesigner zu erreichen, wurde eine Umfärbung der einzelnen Quasarelemente implementiert. Das heißt, nach der Auswahl des Stereotypes wird die Farbe je nach Auswahl verändert.

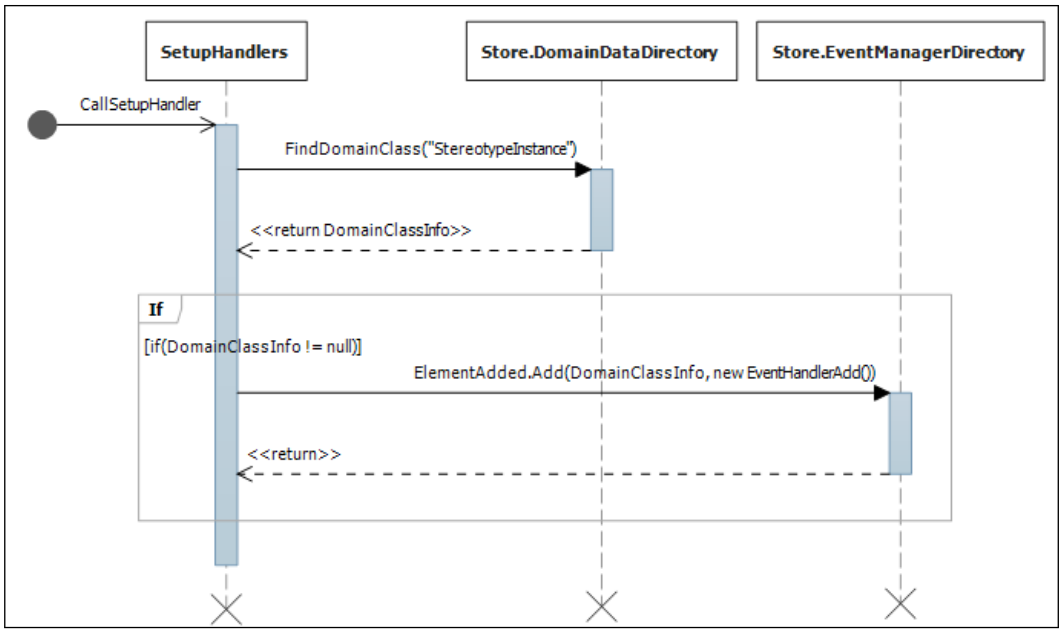
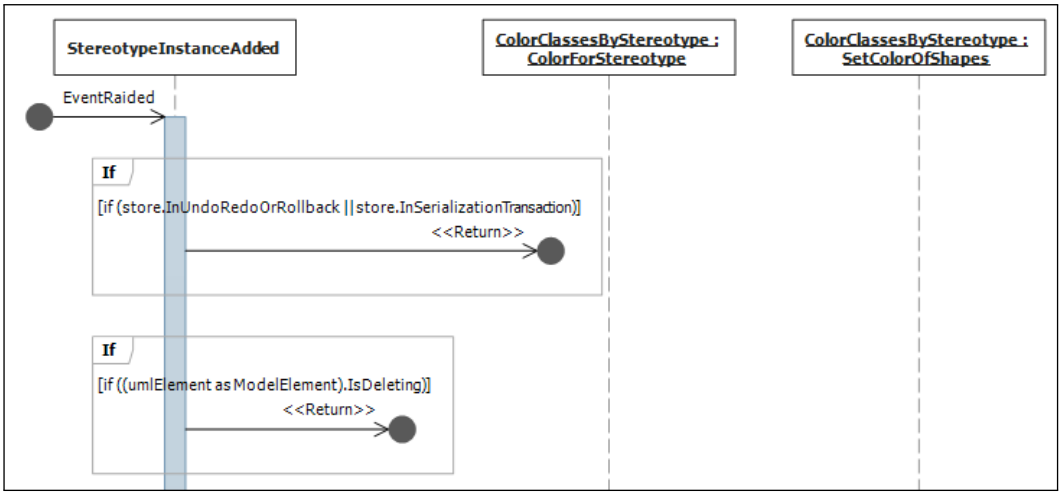


Abbildung 4.2.: Sequenzdiagramm - Eventregistrierung



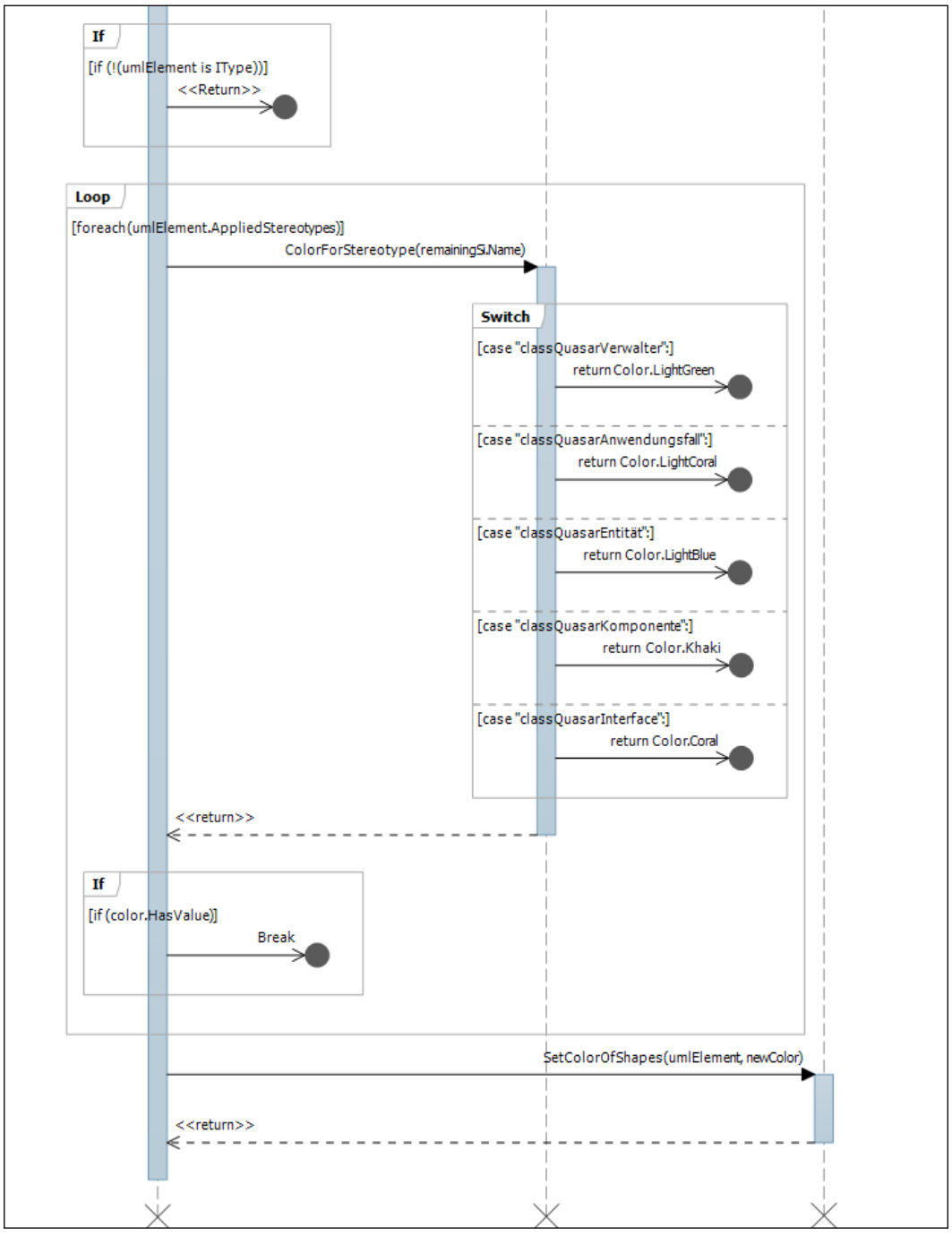


Abbildung 4.3.: Sequenzdiagramm - Event Stereotype hinzugefügt

Die Sequenzdiagramme zeigen den Programmablauf eines Events. Die Registrierung des Event-Handlers geschieht beim Öffnen des entsprechenden Diagramms.

```
[Export(typeof(System.Action<ValidationContext, object>))]
[ValidationMethod(ValidationCategories.Open)]
private void SetupHandlers(ValidationContext vcontext, IModel model)
{
        store = (model as ModelElement).Store;
        DomainClassInfo siClass = store.DomainDataDirectory.
            FindDomainClass("Microsoft.VisualStudio.Uml.Classes.
            StereotypeInstance");
        store.EventManagerDirectory.ElementAdded.Add(siClass, new
            EventHandler<ElementAddedEventArgs>(StereotypeInstanceAdded))
            ;
}
```

Listing 4.8: Registrierung eines Events

Zunächst muss wie bei allen Events oder Rules, die zu überwachende Domainklasse aus dem entsprechenden Dictionary ausgelesen werden. An diese DomainClassInfo wird nun das Event gehängt. In diesem Fall wird die Implementierung „StereotypeInstance“ gesucht. Für alle Instanzen dieser Klasse in diesem Diagramm wird nun dieser Event-Handler registriert.

```
private void StereotypeInstanceAdded(object sender, ElementAddedEventArgs
    e)
{
        if (store.InUndoRedoOrRollback || store.
            InSerializationTransaction) return;
        IStereotypeInstance si = e.ModelElement as IStereotypeInstance;
        IElement umlElement = si.Element;

        if (!(umlElement is IType)) return;
        Color? color = ColorForStereotype(si.Name);

        if (color.HasValue)
        {
                SetColorOfShapes(si.Element, color.Value);
        }
}
```

Listing 4.9: Event-Handler Stereotype hinzugefügt

Beim Hinzufügen eines Stereotype wird die Methode „StereotypeInstanceAdded“ aufgerufen. Da Events auch bei Undo- und Redo-Operationen oder eines Rollbacks aufgerufen werden, muss zunächst sichergestellt werden, dass sich das System nicht in solch einem Zustand befindet. Wenn doch, wird die Operation abgebrochen. Anschließend wird das Sender-Objekt

auf ein Stereotype gecastet und eine Funktion zur Bestimmung des Typs aufgerufen. Zum Schluss wird diese Farbe noch im entsprechenden UML-Shape gesetzt.

```
1  private static Color? ColorForStereotype(string name)
2  {
3          switch (name)
4          {
5                  case "classQuasarVerwalter": return Color.LightGreen;
6                  case "classQuasarAnwendungsfall": return Color.LightCoral
                       ;
7                  case "classQuasarEntität": return Color.LightBlue;
8                  case "classQuasarKomponente": return Color.Khaki;
9                  case "classQuasarInterface": return Color.Coral;
10         }
11         return null;
12 }
```

Listing 4.10: Methode um die Farbe nach gewählten Stereotype zu setzen

Die Methode „ColorForStereotype“ prüft anhand des Namens des Stereotypes mit einer einfachen Switch-Case-Anweisung, um welchen Typ es sich handelt. Danach wird eine zugeordnete Farbe zurückgegeben.

```
1  private static void SetColorOfShapes(IElement element, Color color)
2  {
3          foreach (IShape shape in element.Shapes())
4          {
5          shape.Color = color;
6          }
7  }
```

Listing 4.11: Methode um die Farbe eines UML-Shapes zu setzen

Die Methode „SetColorOfShapes“ setzt die Farbe für ein UML-Shape und alle enthaltenden Elemente.

4.4. Code Generator - Quasar Anwendungsfall

Einer der wichtigsten Implementierungsschritte war die Erstellung des Code Generators. Beispielhaft wird hier die grundlegende Implementierung des Code Generators für den Quasar Anwendungsfall dargestellt. Er besteht zum einen aus der aufrufenden und initialisierenden Klasse und zum anderen aus dem dazugehörigen Template.

```
public class GeneratorAnwendungsfall : GenerateHelper, IGeneratorFile
{
        public GeneratorAnwendungsfall()
        {
        }

        public bool Generate(string path, string component, IElement
            element, out string completePathFileOut)
        {
                if (element == null)
                {
                        completePathFileOut = null;
                        return false;
                }
                // Cast to IClass
                IClass currentElement = (IClass)element;
                if (currentElement == null) return false;
                // Template Instanz
                TemplateAnwendungsfall template = new
                    TemplateAnwendungsfall();
                // Session öffnen
                template.Session = new Microsoft.VisualStudio.
                    TextTemplating.TextTemplatingSession();
                // Parameter zuweisen
                template.Session["_namespace"] = ProjectResources.
                    Properties.Settings.Default.UserNamespace;
                template.Session["_currentElement"] = currentElement;
                // Parameter initialisieren
                template.Initialize();
                // File erstellen
                string result = template.TransformText();
                // Schreiben auf Platte
                completePathFileOut = path + "\\" + component + "\\" +
                    currentElement.Name + ".cs";
                if (File.Exists(completePathFileOut))
                        File.Delete(completePathFileOut);
                File.WriteAllText(completePathFileOut, result);
                return true;
        }
}
```

Listing 4.12: GeneratorAnwendungsfall.cs

Im Generator wird zuerst geprüft, ob ein IElement übergeben worden ist. Anschließend wird es zu einer IClass gecastet und geprüft ob es auch eine IClass ist, ggf. wird die Operation abgebrochen (Zeile 9-17). Danach wird eine Instanz der Templateklasse „TemplateAnwendungsfall“ erstellt (Zeile 19) und eine Template Session eröffnet. Dies ermöglicht es Parameter an ein Template zu übergeben, was in Zeile 23-24 geschieht. Nach der Übergabe werden diese Parameter im Template initialisiert (Zeile 26).

Das fertig initialisierte Template kann nun gerendert werden. Der zurückgelieferte Textstring wird abschließend in einer Datei gespeichert (Zeile 28-33).

```
1  <#@ template language="C#" inherits="BuildInFunctionsClasses" #>
2  <#@ import namespace="System.Linq"#>
3  <#@ import namespace="System.Collections.Generic"#>
4  <#@ import namespace="Microsoft.VisualStudio.Uml.Classes"#>
5  <#@ parameter name="_namespace" type="System.String" #>
6
7  using System;
8  using System.Collections.Generic;
9  using System.Linq;
10 using System.Text;
11
12 namespace <#= _namespace #>
13 {
14
15 <# GenerateHeaderAnwendungsfall(); #>
16         internal class <#= CurrentElement.Name #> : <#= CurrentElement.
               Name #>Base
17         {
18 <# WriteInstanzVariablesKomponente(GetAllClassInDependency("
       classQuasarVerwalter")); #>
19
20                 internal <#= CurrentElement.Name #>(IPersistenceManager
                       persistenceManager)
21                 {
22 <# WriteInstanzVariablesKomponenteConfig(GetAllClassInDependency("
       classQuasarVerwalter")); #>
23                 }
24
25                 #region Implementierung der BasisKlasse
26                 #endregion
27         }
28
```

```
29          /// <summary>
30          /// Base Klasse, bitte hier nichts ändern !!!
31          /// Änderungen gehen verloren !!!
32          /// </summary>
33          internal class <#= CurrentElement.Name #>Base<#=
                GetInterfaceString() #>
34          {
35  <# GetInterfaceMethods(); #>
36          }
37  }
```

Listing 4.13: TemplateAnwendungsfall.tt

Dieser Ausschnitt der Templateklasse für den Quasar Anwendungsfall zeigt, mit welchen einfachen Schritten eine solche Vorlage erstellt werden kann. Vererbungen (Zeile 1), Imports (Zeile 2-4) und Parameterübergaben von außerhalb (Zeile 5) sind alle mit wenigen Zeilen Code realisierbar. Funktionsaufrufe von vererbten Klassen stellen eine sehr übersichtliche Gestaltung der Haupttemplateklasse sicher.

4.5. Synchronisator - Quasar Anwendungsfall

Die meiste Zeit der Implementierungsarbeit wurde in die Erstellung der Synchronisatoren verwendet. Die Schwierigkeiten hierbei lagen zum einen in weitreichenden Entscheidungen über das Design der Templatevorlage und zum anderen im Explorieren eines vorhandenen Codeelementes. Für Letzteres bietet der CodeDOM eine sehr gute Möglichkeit vorhandenen Code auszulesen, zu analysieren und weiterzuverarbeiten. Hier gibt es keine direkte Möglichkeit UML, T4-Template und CodeDOM zu synchronisieren. Es wird eine Wrapper-Klasse benötigt, um Funktionen und Attribute zu transformieren. Das ist der einzige Punkt, warum sich rückblickend der Einsatz eines CodeDOM-Templates ausgezahlt hätte.

Beispielhaft für alle Synchronisationsfunktionen wird der Synchronisator - Quasar Anwendungsfall mit einigen Ausschnitten beschrieben, um markante Punkte aufzuzeigen.

```
1 public class SyncronatorAnwendungsfall : GenerateHelper, ISyncronator
2 {
3         public SyncronatorAnwendungsfall(object synronateElement,
              IComponent parentComp)
4         {
5                 if (synronateElement is ProjectItem)
6                         this.currentProjectItem = synronateElement as
                              ProjectItem;
```

```
7                 else if (synronateElement is IClass)
8                         this.currentClassItem = synronateElement as
                              IClass;
9                 this.parentComp = parentComp;
10        }
11
12        public bool Syncronate(SyncronatorDirectionType
              syncronatorDirectionType)
13        {
14                if (syncronatorDirectionType == SyncronatorDirectionType.
                      CODETOUMLSYNCRONATION)
15                        return SyncronateCODETOUML();
16                else if (syncronatorDirectionType ==
                      SyncronatorDirectionType.UMLTOCODESYNCRONATION)
17                        return SyncronateUMLTOCODE();
18                else
19                        return false;
20        }
21
22        public bool SyncronateUMLTOCODE()
23        {
24                //......
25                if (HasReference(currentClassItem))
26                {
27                        oldRef = GetSimpleDecodedReference(
                              GetReferenceValues(currentClassItem).
                              FirstOrDefault());
28                        // Delete old ProjectItem
29                        if (oldRef != null)
30                        {
31                                ProjectItem oldProjectItem = Dte.Solution
                                      .FindProjectItem(oldRef);
32                                if (oldProjectItem != null)
33                                {
34                                        // rescue user impl Code
35                                        CodeClass codeElementClass =
                                              ProjectItemHelper.GetClass(
                                              oldProjectItem);
36                                        if (codeElementClass != null)
37                                        {
38                                                // Content
39                                                EditPoint2 editPointStart
                                                      = (EditPoint2)
```

```
                                                    codeElementClass.
                                                    GetStartPoint(
                                                    vsCMPart.vsCMPartBody
                                                    ).CreateEditPoint();
40                                                EditPoint2 editPointEnd =
                                                     (EditPoint2)
                                                    codeElementClass.
                                                    GetEndPoint(vsCMPart.
                                                    vsCMPartBody).
                                                    CreateEditPoint();
41                                                codeUserImplRescue =
                                                    editPointStart.
                                                    GetText(editPointEnd)
                                                    ;
42                                          }
43                                          oldProjectItem.Delete();
44                                  }
45                          }
46                  }
47
48                  IGeneratorFile fileGenerator = new
                        GeneratorAnwendungsfall();
49                  fileGenerator.Generate(projectPath, parentComponent.Name,
                         currentClassItem, out newFile);
50
51                  //......
52
53                  if ((newProjectItem != null) && (codeUserImplRescue !=
                        null))
54                  {
55                          CodeClass codeElementClass = ProjectItemHelper.
                                GetClass(newProjectItem);
56                          if (codeElementClass != null)
57                          {
58                                  // Content
59                                  EditPoint2 editPointStart = (EditPoint2)
                                        codeElementClass.GetStartPoint(
                                        vsCMPart.vsCMPartBody).
                                        CreateEditPoint();
60                                  EditPoint2 editPointEnd = (EditPoint2)
                                        codeElementClass.GetEndPoint(vsCMPart
                                        .vsCMPartBody).CreateEditPoint();
61                                  // Delete Old Content
```

```
62                                  editPointStart.Delete(editPointEnd);
63                                  // Insert Rescued Content
64                                  editPointStart.Insert(codeUserImplRescue)
                                        ;
65                              }
66                              // Format
67                              EditPoint2 editPointFormatStart = (EditPoint2)
                                    codeElementClass.GetStartPoint(vsCMPart.
                                    vsCMPartBody).CreateEditPoint();
68                              EditPoint2 editPointFormatEnd = (EditPoint2)
                                    codeElementClass.GetEndPoint(vsCMPart.
                                    vsCMPartBody).CreateEditPoint();
69                              editPointFormatStart.SmartFormat(
                                    editPointFormatEnd);
70                      }
71                      return true;
72              }
73              public bool SyncronateCODETOUML()
74          {
75              //......
76          }
77      }
```

Listing 4.14: SyncronatorAnwendungsfall.cs

Wie man erkennen kann, wurden beide Synchronisationsrichtungen in einer Klasse realisiert. Je nachdem welcher Objekttyp im Konstruktor (Zeile 3) übergeben wird, wird beim Aufruf der Syncronate (Zeile 12) entschieden, in welche Richtung die Synchronisation ablaufen wird. Das Hauptaugenmerk bei der Synchronisation eines UML-Diagramms mit bestehendem Code ist das Sichern und Retten benutzerspezifischer Implementierungen.

In Zeile 35 wird mithilfe einer Explorerklasse der CodeDOM erforscht und die erste Klasse, die im Projektitem gefunden wird, zurückgegeben. Laut Template-Definition befinden sich dort die benutzerspezifischen Implementierungen. In Zeile 39 und 40 werden EditPoints am Start und am Ende des Funktionskörpers erstellt. Anschließend wird der Code innerhalb dieser Punkte gesichert und das alte Projektitem wird gelöscht. Danach wird mit dem Anwendungsfallgenerator ein neues Projektitem erstellt und eingebunden.

Das neue Projektitem wird exploriert und die Stelle wird gesucht, an der der gerettete Code eingefügt werden kann (Zeile 55). Unnötiger Code wird gelöscht und anschließend

wird der Code eingefügt (Zeile 59-64). Zum Abschluss wird der gesamte Text formatiert, um Einrückungen und Optik zu garantieren (Zeile 67-69).

5. Test & Integration

Am Ende der Entwicklungsphase eines Softwareprojektes steht der intensive Test der erstellten Software. Ziel ist es immer, automatisierte Unit-Tests zu erstellen, um Schwachpunkte der Software aufzudecken, Fehler zu finden und eine fehlerfreie und einwandfreie Funktion zu gewährleisten.

Aber genau das ist das Problem bei der Erstellung dieser Erweiterung für Visual Studio. Der Umfang und der Programmieraufwand wären meiner Ansicht nach zu hoch, als dass es sich lohnen würde, automatisierte Unit-Tests zu erstellen. Als Beispiel nehmen wir den Anwendungsfall - „Synchronisation UML-Shape(Anwendungsfall) mit bestehendem Sourcecode“. Der Benutzer wählt zuerst eine Komponente im Komponentendiagramm aus und öffnet das dazugehörige Klassendiagramm. Dann selektiert er die Klasse xyz-Anwendungsfall, macht einen Rechtsklick und wählt „Synchronisieren“ aus. Das System sucht anhand der Referenz die passende Datei und der Inhalt der partiellen Klasse wird gesichert, in der der Benutzer seine Implementierung durchgeführt hat. Danach wird die Datei gelöscht und eine neue erzeugt.

Ein Unit-Test müsste nun zuerst einen Anwendungsfall in einem Klassendiagramm suchen und selektieren. Dazu muss das Klassendiagramm erst geöffnet werden. Dann sucht er die dazugehörige Datei, die referenziert worden ist. Das heißt, es muss in der Solution ein Projekt hinzugefügt werden, in der der Unit-Test Projektitems erzeugen und testen kann, was ich nicht für sinnvoll halte. Der Unit-Test ruft nun die Synchronisationsfunktionen auf. Anschließend muss der Unit-Test prüfen, ob eine neue Datei vorhanden ist, was ein trivialer Fall ist. Der Unit-Test prüft danach, ob sich genau der Inhalt, der sich vorher in der Datei befunden hat, sich auch jetzt wieder darin befindet.

Aus diesen Gründen habe ich mich entschlossen, manuelle Testreihen durchzuführen und die verschiedenen Anwendungsfälle von Benutzern testen zu lassen.

5.1. Unit-Tests

Im Projekt wurde beispielhaft ein Unit-Test durchgeführt, um zu zeigen, wie groß der Aufwand wäre, für alle Anwendungsfälle die dazugehörigen Unit-Tests zu kreieren. Der weiter oben in Kapitel 5 beschriebene Anwendungsfall - „Synchronisation UML-Shape(Anwendungsfall) mit bestehendem Sourcecode“ wurde dafür gewählt.

Precondition : Komponentendiagramm existiert, Klassendiagramm existiert,
Komponente hat eine Referenz auf das Klassendiagramm, UML-Shape
Anwendungsfall hat eine Referenz auf ein Projekt
Postcondition : Unidirektionale Synchronisation wurde durchgeführt
Data : Name des Anwendungsfalls, der synchronisiert werden soll
Result : Bestätigung, dass die Synchronisation stattgefunden hat

```
initialization;
shapeName ← GetInput();
currentShape ← FindCurrentShape(shapeName);
if currentShape ==null then
    errorMSG;
    break;
end
currentComponent ← FindComponent(currentShape);
if currentComponent ==null then
    errorMSG;
    break;
end
if TestStereotype(currentShape,„Anwendungsfall“) ==false then
    errorMSG;
    break;
end
currentProject ← FindProject(currentComponent);
if currentProject ==null then
    errorMSG;
    break;
end
OpenDiagram(currentComponent);
/* Save Projectitem for later comparisim                    */
savedProjectItem ← ReadProjectItem(currentShape);
selection ← SelectShape(currentShape);
result ← SynchronizeUMLtoCode(selection);
```

```
/* Read syncronized Shape                                              */
newProjectItem ← ReadProjectItem(currentShape);
testResult ← CompareCodeFragments(savedProjectItem, newProjectItem)
if testResult == true then
    successMSG;
end
else
    errorMSG;
end
return;
```

Algorithmus 1: Unit-Test: „Synchronisation UML-Shape mit Code“

```
get IModelingProject;
get IModelStore from IModelingProject;
get currentShape from IModelStore;
return currentShape;
```

Funktion 2 : FindCurrentShape(string shapeName)

Um diesen Unit-Test ausführen zu können, muss ein Komponentendiagramm erzeugt und befüllt werden. Anschließend wird eine entsprechende Komponenteninnensicht generiert und der Anwendungsfall, den es zu prüfen gilt, wird benannt. Ein C#-Projekt muss generiert und eine Referenz zu der entsprechenden Komponente erzeugt werden. Nun ist das System für den Unit-Test bereit. Der Implementierungsumfang eines solchen Unit-Tests hätte ungefähr den gleichen Aufwand, wie das Erstellen der eigentlichen Klasse für die Synchronisierung. Darum ist es in meinen Augen hier nicht sinnvoll, ein solches Arbeitsvolumen zu investieren.

5.2. Manuelle Usertests

Um die Analyse der Schwächen bei der manuellen Entwicklung einer Software nach Quasargesichtspunkten unter Visual Studio zu bestätigen, habe ich den Usertest in 2 Schritte unterteilt. Zuerst mussten alle Probanden das Beispiel aus Abschnitt 2.3 „Begleitendes Beispiel“ manuell umsetzen. Des Weiteren sollten sie eine neue Klassenbibliothek „Rechnung“ erzeugen und anschließend die Synchronität zwischen Entwurf und Implementation herstellen.

Damit alle Benutzer den gleichen Wissensstand für diesen Test hatten, habe ich eine kurze Einführung in das Thema Quasar gegeben und die wichtigsten Entwicklungsschritte erläutert.

5.2.1. Auswertung der Entwicklung ohne Nutzung der Visual Studio Erweiterung

Jeder der Probanden hatte 30 Minuten Zeit, um den Entwurf des Beispiels aus Abschnitt 2.3 „Begleitendes Beispiel“ umzusetzen. Anschließend sollten sie kurz ihre Erfahrungen stichpunktartig niederschreiben, Schwächen und Fehler benennen, die aufgetreten sind. Des Weiteren konnten die Probanden neue Anforderungen für zukünftige Projekte vorschlagen. Aus den gesammelten Fehlern und Schwachstellen habe ich anschließend eine Top10-Liste erstellt.

1. sehr hoher Workload
2. Rechtschreibfehler
3. Fehler bei Copy&Paste-Vorgängen
4. kein einheitlicher, vom System vorgeschriebener Aufbau der Dateien
5. sich oft wiederholende Arbeitsabläufe
6. Refaktorisierung führt zu doppelter Arbeitsbelastung
7. manuelle Synchronisation kann zu Fehlern führen, wenn sie nicht zeitnah getätigt wird
8. der Benutzer kann vergessen ein Objekt zu synchronisieren
9. es wird sehr unübersichtlich bei großen Projekten
10. Pflege von UML-Diagrammen ist sehr anstrengend

Anzumerken ist, dass alle Probanden die vollen 30 Minuten ausschöpfen mussten, um die ihnen gestellte Aufgabe abzuschließen und den Code eingehend auf Fehler zu prüfen.

Im Nachhinein wäre es natürlich sinnvoller gewesen, diesen Test vor Beginn der Entwicklung durchzuführen. Dies ließ sich aus Zeitgründen der Testpersonen in den Semesterferien nicht realisieren. Aber der Test zeigt, dass alle Benutzer die Schwächen, die in Kapitel 2 aufgezeigt wurden, bestätigen konnten.

Außerdem sollten die Probanden jeweils 3 Vorschläge/Anforderungen an eine Entwicklungsumgebung äußern, die sie für besonders wichtig hielten.

- automatische Codegenerierung aus einem UML-Diagramm heraus
- Synchronisierung von UML-Diagramm mit Sourcecode
- Fehlererkennung und -vermeidung
- Zusammenfassung von Arbeitsschritten
- Validierung von Entwurf und Sourcecode

5.2.2. Auswertung der Entwicklung unter Nutzung der Visual Studio Erweiterung

Nach Abschluss des ersten Testdurchlaufes sollten die Probanden nun unter Einsatz der von mir erstellten Visual Studio Erweiterung das gleiche Szenario noch einmal durchspielen. Es wurde wieder ein Zeitlimit von 30 Minuten festgelegt. Anschließend sollten die Probanden die Vor- und Nachteile der neuen Entwicklungsumgebung wiedergeben.

Der erste Punkt, der sofort hervortrat, war die enorme Zeitersparnis. Alle Probanden konnten den Versuch nach maximal 5 Minuten abschließen, einige geübte Softwareentwickler sogar schon nach 3 Minuten.

Einige der genannten Vorteile:

- hohe Zeitersparnis
- keine monotonen sich wiederholenden Copy&Paste-Aktionen
- Fehlererkennung und Validierung
- viele Automatismen
- Synchronisierung und Pflege von Entwurf und Implementierung

Nachteilig wurde eigentlich nur ein einziger Punkt genannt, dass war die unidirektionale Synchronisierung. Einige der Benutzer hätten es noch effektiver gefunden, wenn die Synchronisation bidirektional gewesen wäre. So wie es in Abschnitt 3.6 „Synchronisation“ beschrieben wurde, habe ich den Probanden das Problem mit der bidirektionalen Synchronisation dargelegt, um zu zeigen, dass nicht grundlos diese Form der Implementierung gewählt worden ist.

5.2.3. Manueller Test verschiedener Anwendungsfälle

Um repräsentative Testergebnisse zu bekommen, sollten die Probanden im letzten Schritt eine Liste von Anwendungsfällen abarbeiten, um eventuelle Fehler in der Implementierung aufzudecken oder Verbesserungsvorschläge zu unterbreiten. Hierzu wurden für diesen Test einige zusätzliche Funktionen freigeschaltet, wie zum Beispiel das Aufrufen spezieller Validierungsschritte, um explizit nach Fehlern suchen zu können. Die komplette Tabelle befindet sich in Anhang B.

Anwendungsfall	**Zeit**	**Fehler**	**Anmerkungen**
...			
Umbenennen eines Projektitems	>1 min	—	—

...

Tabelle 5.1.: Ausschnitt Tabelle mit Testaufgaben für manuelle Usertests

Wenn Fehler auftraten, wurden diese behoben. Auch einige Verbesserungsvorschläge wurden unterbreitet, wie z.B. das Einführen einer Bildexport-Funktion oder einer Funktion, um UML-Shapes automatisch anordnen zu können.

5.3. Installation

Um die Visual Studio Erweiterung erfolgreich installieren zu können müssen einige Rahmenbedingungen geschaffen werden. Zu beachten ist, dass die Erweiterung nur unter Visual Studio Professional 2010 und höherwertigen Editionen, sprich Premium und Ultimate, läuft. Die Express-Version wird nicht unterstützt.
Es müssen, wenn nötig, noch einige Zusatzpakete installiert werden. Die benötigten Dateien werden mit der Erweiterung bereitgestellt.

1. Visual Studio SDK (VsSDK_sfx.exe)
2. Visualization and Modeling Feature Pack (Visualization and Modeling Feature Pack Runtime.vsix)
3. Visualization and Modeling SDK (vs_vmsdk.exe)
4. Visual Studio Patch KB2403277 (VS10-KB2403277-x86.exe)
5. Visualization and Modeling Feature Pack 2 (en_visual_studio_2010_fp2_x86_604352.msi)

Bei der Installationsroutine für die hier erstellte Erweiterung wurden oben genannte Pakete auch als Precondition für die Installation vorausgesetzt. Leider gibt der Visual Studio Installer nur sehr kryptische Fehlermeldungen über die benötigten Pakete aus. Darum wurden sie in obiger Aufzählung noch einmal ausführlich aufgeführt.

Damit ist alles für die Installation der Quasar-Entwicklungsplattform (CodeGenerator.vsix) bereit. Nach der Installation ist Visual Studio und die installierte Erweiterung nutzbar.

5.4. Benutzung

Eine detaillierte Anleitung mit dem Beispiel aus Abschnitt 2.3 „Begleitendes Beispiel“ befindet sich in Anhang C.

6. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Erweiterung für Visual Studio entwickelt, die den Entwickler bei der Gestaltung und Umsetzung einer nach Quasar-Richtlinien entworfenen Software unterstützen soll. Der realisierte Prototyp bietet dem Softwareentwickler eine „Out-of-the-Box“ Entwicklungsumgebung, die es ihm ermöglicht, den gesamten Implementierungsvorgang in einem einzigen Vorgang zu bündeln.

Durch die Automatisierung von Codegenerierung und Synchronisation, die Validierung der Entwürfe in den UML-Diagrammen, ist es mit dieser Erweiterung möglich, die Produktivität eines Entwicklers entscheidend zu steigern und den anfallenden Workload für diese Aufgaben zu senken. Lästige, monotone und sich ständig wiederholende Aufgaben entfallen ganz oder wurden auf ein Minimum reduziert.

Der gesamte Entwicklungszyklus ist weniger fehleranfällig. Somit steigen Qualität und Wartbarkeit der mit dieser Erweiterung erzeugten Software.

6.1. Probleme und Einschränkungen

Wie in vielen anderen Softwareprojekten gab es auch hier einige Probleme und Kompromisse, die man eingehen musste. An vielen Stellen ist man bei der Entwicklung an die Grenzen der API von Visual Studio gestoßen. Einige Funktionen, die man sich gewünscht hätte, konnten nicht vollständig oder manchmal nur auf sehr kompliziertem Wege realisiert werden.
Ein Beispiel dafür ist das Auslesen und Verändern des Inhaltes eines UML-Diagramms bei der Codegenerierung.

```
1 // Modellierungsprojekt aus vorhandenen Projekten auswählen
2 Project modellingProject = ((Solution2)Dte.Solution).Projects.OfType<
      Microsoft.VisualStudio.TeamArchitect.ModelingProject.
      ModelProjectAutomationObject>().First();
3 IModelingProject modellingProjectCast = modellingProject as
      IModelingProject;
```

```
IModelStore modelStore = modellingProjectCast.Store;
using (IModelingProjectReader projectReader = ModelingProject.
    LoadReadOnly(modellingProject.FullName))
{
  IDiagram diagram = projectReader.LoadDiagram(projectItem.Name);
  ..... do something
  foreach (IShape<IElement> currentElement in diagram.GetChildShapes<
      IElement>())
  {
    IElement currentElementSearch = modelStore.FindElement(currentElement
        .GetElement().GetId());
    if (currentElementSearch != null)
      ... do something
  }
}
```

Listing 6.1: Auslesen und Verändern des Inhaltes eines UML-Diagramms

Es ist schon an der Namensgebung zu erkennen, dass es sich bei der vorhandenen Möglichkeit, auf ein bestimmtes Diagramm zuzugreifen, leider nur um einen Reader handelt. Änderungen im Diagramm oder einem UML-Shape sind somit ausgeschlossen und werden mit einer Exception von Visual Studio beantwortet. Will man nun Änderungen in einem UML-Shape durchführen, muss man dieses erst wieder aufwendig im Modelstore suchen, referenzieren und anschließend verändern. Das ist nur ein Beispiel dafür, dass die Visual Studio API an vielen Stellen noch Verbesserungsbedarf hat. Auch das Fehlen von einigen Event-Handlern führt dazu, das bei einigen Implementierungsschritten ein gewisser Überhang an Code entstanden ist, der normalerweise in Before-/und After-Events implementiert und ausgeführt werden müsste.

Von den Anforderungen, die im Vorfeld an diese Visual Studio Erweiterung gestellt wurden, konnten alle bis auf eine Forderung umgesetzt werden. Eine Mehrfachvererbung innerhalb einer Interfacekette ist zur Zeit nicht möglich. Das heißt, eine Konstellation wie in der linken Abbildung von Abb. 6.1 kann leider nicht umgesetzt werden.

Durch eine rekursive Funktion, die die „Interface-Realisations“ in einem UML-Diagramm durchläuft, könnte diese Funktion in den bestehenden Code-Generator integriert werden.

Ein Änderungswunsch, die zeitgleiche bidirektionale Synchronisation, der bei den Usertests vorgeschlagen wurde, konnte nicht umgesetzt werden. Wie ich aber schon in Abschnitt 3.6 „Synchronisation“ dargelegt habe, wurde diese Funktion aus oben genannten Gründen nicht

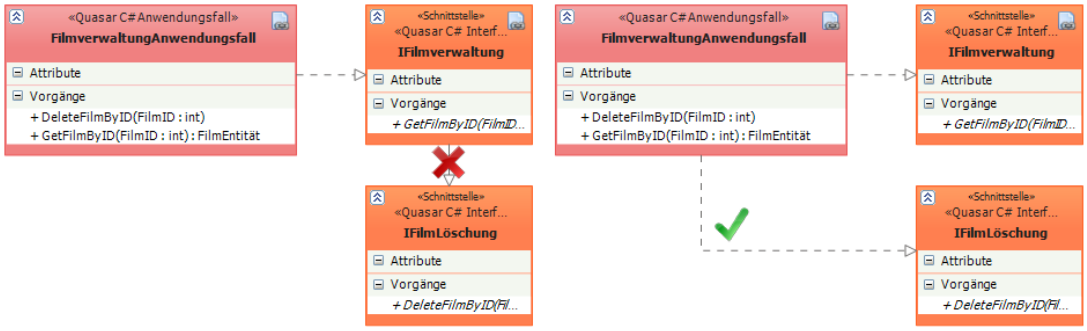


Abbildung 6.1.: Mehrfachvererbung innerhalb von Interfaces

implementiert. Ich habe mich bei der Synchronisation auf eine unidirektionale Synchronisation beschränkt.

6.2. Ausblick

Bei der in dieser Arbeit entwickelten Visual Studio Erweiterung handelt es sich um einen Prototyp. Um diesen für einen produktiven Einsatz vorzubereiten, müssen im Vorfeld weitere Untersuchungen und gegebenenfalls Weiterentwicklungen einiger Teilkomponenten stattfinden. Vom derzeitigen Standpunkt aus bieten sich mehrere Alternativen zur weiteren Entwicklung dieser Visual Studio Erweiterung an.

- **Reverse-Engineering:** Eine interessante Option stellt die Möglichkeit dar, bestehenden Code in einem Arbeitsschritt einem komplett Reverse-Engineering zu unterziehen. Das bedeutet, dass alle zugehörigen UML-Diagramme erzeugt und befüllt werden. Diese Funktion wäre für die Wartung eines Fremdsystems, das unter Quasargesichtspunkten entworfen wurde, sehr nützlich.
- **Persistenzschicht:** Auch bei der Codegenerierung sind verschiedene Erweiterungen denkbar. Zurzeit ist nur der Einsatz des OR-Mappers NHibernate und dem dazugehörigen Fluentinterface angedacht. Hier ist es aber in Zukunft durchaus denkbar, flexibler auf den Benutzer einzugehen und ihm eine größere Auswahl an Datenbankverbindungen und Persistenzschichten anzubieten.
- **Unit-Tests:** Im Bereich der Unit-Tests gibt es auch Potenzial für Erweiterungen. Es wäre in späteren Entwicklungsschritten möglich, dass automatisch generierte Unit-Tests für verschiedene Operationen, sei es auf Datenbankebene oder der Test von verschiedenen

Anwendungsfällen, erstellt werden. Das würde dem Entwickler eine weitere Zeitersparnis bringen.

- **Bugtracking:** Im Zusammenhang mit Bugtracking ist es denkbar, auftretende Fehler bei der Validierung, der Generierung oder der Synchronisation als Fehlerreports zu exportieren. Anschließend könnten noch dazugehörige Workitems erstellt werden.
- **TFS (Team Foundation Server):** Eine mögliche Weiterentwicklung für eine komplette Fehlererfassung wäre, alle erfassten Fehler und dazugehörigen Workitems mit einem TFS (Team Foundation Server) zu synchronisieren. Das würde einem größeren Entwicklerteam ermöglichen, den kompletten Überblick über das Projekt und alle auflaufenden Fehler und ausstehenden Arbeitsschritte zu geben.
- **Dokumentation:** Für eine lückenlose Dokumentation wäre es sinnvoll, Exportschnittstellen zur Verfügung zu stellen. Dem Entwickler kann somit ermöglicht werden, beispielsweise Schnittstellenbeschreibungen, Komponenteninnensichten oder die fachliche Architektur mit dem Entwurf zu synchronisieren. C# bietet hier einige Bibliotheken für PDF oder Word Export. Auch eine Anbindung an ein firmeneigenes Dokumentationssystem ist denkbar.
- **Visual Studio 2011:** Mit der neuen Version von Visual Studio 2011 werden neue Funktionen der API hinzugefügt. Leider konnte Microsoft keine vollständige Kompatibilität zwischen MEF-Komponenten bezogenen Schnittstellen, besonders in Bezug auf Events und Rules, garantieren. Somit gilt es hier, Integrationstests durchzuführen, bevor man die hier erstellte Erweiterung unter kommenden Versionen von Visual Studio benutzen kann.
- **Usability-Analyse:** Im Rahmen dieser Arbeit wurde keine Usability-Analyse in dem an der HAW-Hamburg vorhandenen Labor durchgeführt. Es wurden lediglich die subjektiven Wünsche einiger Testprobanden ermittelt und versucht sie umzusetzen. Im weiteren Verlauf einer Weiterentwicklung ist zu empfehlen, eine Usability-Analyse durchzuführen und Schwachstellen der Erweiterung zu ermitteln und zu beheben.

Zusammenfassend kann man sagen, die im Rahmen dieser Arbeit entwickelte Erweiterung für Visual Studio erfüllt alle Anforderungen, Erwartungen und Vorstellungen, die gefordert waren. Trotz sehr intensiver Vorbetrachtung und Analyse sind im Verlauf der Bearbeitung des Themas viele neue Gesichtspunkte und Aspekte zum Vorschein getreten, die im Vorfeld nicht vorhersehbar waren und die in zukünftigen Entwicklungsprozessen umgesetzt werden sollten.

A. Abkürzungsverzeichnis

Abkürzung	**Langform**
API	**A**pplication **P**rogramming **I**nterface
ASP	**A**ctive **S**erver **P**age
AWK	**An**wendungs**k**ern
BMP	**B**it**Ma**p
CodeDOM	**Code D**ocument **O**bject **M**odel
GUI	**G**raphical **U**ser **I**nterface
GUID	**G**lobally **U**nique **Id**entifier
JPEG	**J**oint **P**hotographic **E**xpert **G**roup
MEF	**M**anaged **E**xtensibility **F**ramework
MSI	**M**icro**s**oft **I**nstaller
PNG	**P**ortable **N**etwork **G**raphics
Quasar	**Q**ualitäts-**S**oftware-**A**rchitektur
SDK	**S**oftware **D**evelopment **K**it
T4	**T**ext **T**emplate **T**ransformation **T**oolkit
TFS	**T**eam **F**oundation **S**erver
UML	**U**nified **M**odeling **L**anguage
VSI	**V**isual **S**tudio Content **I**nstaller
VSIX	**V**isual **S**tudio **E**xtension
VSPackage	**V**isual **S**tudio **P**ackage

B. Tabelle Anwendungsfälle Usertests

Anwendungsfall	Zeit	Fehler	Anmerkung
Einstellungen			
Eingabe eines unzulässigen Namespaces			
Eingabe eines zulässigen Namespaces			
Eingabe eines unzulässigen Namespaces AWKF			
Eingabe eines zulässigen Namespaces AWKF			
Eingabe eines unzulässigen Namespaces RTE			
Eingabe eines zulässigen Namespaces RTE			
Framework ändern			
Hinzufügen von verschiedenen Libraries			
Neue Solution anlegen			
Eingabe eines unzulässigen Namens			
Projektpfad ändern			
Vorgang abbrechen			
Eingabe eines zulässigen Namens			
Projekt anlegen			
Leeres Komponentendiagramm öffnen			
UML-Komponentenentwurf			
Hinzufügen einer UML-Komponente			
Eingabe eines unzulässigen Namens			
Eingabe eines zulässigen Namens			
Umbenennen einer UML-Komponente			
Eingabe eines unzulässigen Namens			
Eingabe eines zulässigen Namens			
Öffnen des Klassendiagramms (Doppelklick)			
Hinzufügen eines Ports (Interface)			

Verbinden zweiter Ports von Komponenten
Löschen einer UML-Komponente
Dateiverlinkung
Ändern einer Dateiverlinkung
Löschen einer Dateiverlinkung
Hinzufügen einer Dateiverlinkung
Öffnen des Klassendiagramms (Doppelklick)
UML-Komponenteninnensicht Entwurf
Inhalt eines neu angelegten Komponente prüfen
Hinzufügen einer UML-Klasse (Kontextmenü)
Hinzufügen eines UML-Interfaces (Kontextmenü)
Hinzufügen einer UML-Klasse (Toolbox)
Hinzufügen eines UML-Interfaces (Toolbox)
Änderung des Stereotypes
Umbenennen einer UML-Klasse/Interface
Eingabe eines unzulässigen Namens
Eingabe eines zulässigen Namens
Löschen einer UML-Klasse/Interface
Hinzufügen Vorgang in einem Interface
Hinzufügen eines Attributes in einer Entität
Validierung Komponentendiagramm
Validierung aller Komponenten
Validierung einer Komponente
Löschen einer Dateiverlinkung
Validierung einer Fehlerausgabe
Validierung Klassendiagramm
Validierung aller Klassendiagramme
Validierung eines Klassendiagrammes
Löschen eines Stereotypes
Validierung einer Fehlerausgabe

Hinzufügen mehrerer Stereotypes
Validierung einer Fehlerausgabe
Löschen einer Abhängigkeit (Referenz)
Validierung einer Fehlerausgabe
Löschen einer „Quasar-Komponente“
Validierung einer Fehlerausgabe
Löschen eines „Quasar-Anwendungsfalls“
Validierung einer Fehlerausgabe
Löschen eines „Quasar-Verwalters“
Validierung einer Fehlerausgabe
Löschen einer „Quasar-Entität“
Validierung einer Fehlerausgabe
Hinzufügen mehrerer „Quasar-Komponenten“
Validierung einer Fehlerausgabe
Validierung Filesystem
Validierung aller Komponenten
Validierung einer Komponente
Verzeichnis anlegen (Komponentenname)
Validierung einer Fehlerausgabe
Datei anlegen (Element Klassendiagramm)
Validierung einer Fehlerausgabe
Codegenerierung
Prüfen ob Validator gestartet wird
Generierung aller Komponenten
Generierung einer Komponente
Generierung Quasar-Anwendungsfall
Generierung Quasar-Interface
Generierung Quasar-Komponente
Generierung Quasar-Verwalter
Generierung Quasar-Entität
Generierung Anwendungskernfassade
Generierung einer Anwendungskernfassade
Prüfen ob Button weiter verfügbar ist

Validierung Inhalt Anwendungskernfassade
Generierung Runtime Environment
Generierung einer Laufzeitumgebung
Prüfen ob Button weiter verfügbar ist
Validierung Inhalt der Laufzeitumgebung
Synchronisierung UML -> Code
Synchronisierung Komponente
Synchronisierung Quasar-Anwendungsfall
Synchronisierung Quasar-Interface
Synchronisierung Quasar-Komponente
Synchronisierung Quasar-Verwalter
Synchronisierung Quasar-Entität
Erzeugen diverser Fehler (Dateilink löschen,
Inhalt+Struktur der Dateien grob verändern....)
Validierung einer Fehlerausgabe
Synchronisierung Code -> UML
Synchronisierung Komponente
Synchronisierung Quasar-Anwendungsfall
Synchronisierung Quasar-Interface
Synchronisierung Quasar-Komponente
Synchronisierung Quasar-Verwalter
Synchronisierung Quasar-Entität
Erzeugen diverser Fehler (Dateilink löschen,
Inhalt+Struktur der Dateien grob verändern....)
Validierung einer Fehlerausgabe
Projektmappenexplorer - Projektebene
Hinzufügen eines Projekts
Umbenennen eines Projekts
Eingabe eines unzulässigen Namens
Eingabe eines zulässigen Namens
Löschen eines Projekts

Validierung - Projekt kann neu generiert werden
Projektmappenexplorer - Projektitemebene
Hinzufügen eines Projektitems
Umbenennen eines Projektitems
Eingabe eines unzulässigen Namens
Eingabe eines zulässigen Namens
Löschen eines Projektitems
Validierung Projektitem kann neu generiert werden
Validierung Dateiverlinkung ist nicht vorhanden

Tabelle B.1.: Tabelle mit Testaufgaben für manuelle Usertests

C. Bedienungsanleitung

C.1. Installation

Um die Visual Studio Erweiterung erfolgreich installieren zu können müssen einige Rahmenbedingungen geschaffen werden. Zu beachten ist, dass die Erweiterung nur unter Visual Studio Professional und höherwertigen Editionen, sprich Premium und Ultimate, läuft. Die Express-Version wird nicht unterstützt.
Es müssen, wenn nötig, noch einige Zusatzpakete installiert werden. Die benötigten Dateien werden mit der Erweiterung bereitgestellt.

1. Visual Studio SDK (VsSDK_sfx.exe)
2. Visualization and Modeling Feature Pack (Visualization and Modeling Feature Pack Runtime.vsix)
3. Visualization and Modeling SDK (vs_vmsdk.exe)
4. Visual Studio Patch KB2403277 (VS10-KB2403277-x86.exe)
5. Visualization and Modeling Feature Pack 2 (en_visual_studio_2010_fp2_x86_604352.msi)

Damit ist alles für die Installation der Quasar-Entwicklungsplattform (CodeGenerator.vsix) bereit. Nach der Installation ist Visual Studio und die installierte Erweiterung nutzbar.

C.2. Die ersten Schritte

Nach der Installation der Quasar-Entwicklungsplattform muss diese noch konfiguriert und eingerichtet werden. Hierzu öffnen sie die Einstellungen unter **„CodeGenerator -> Einstellungen Ändern“**. Hier haben sie auf zwei verschiedenen Reitern Einstellungsmöglichkeiten für die Entwicklungsplattform.

C.2.1. Code Generator Einstellungen

In diesem Bereich können sie die verschiedenen Namespaces ihres Systems anlegen. Mit der Reset-Taste werden die Einstellungen auf die im System hinterlegten Default-Werte zurückgesetzt. Des weiteren können sie hier das .NET-Zielframework festlegen, mit denen alle neuen Projekte und Projektitems erzeugt werden. **Bitte beachten sie, dass das angegebene Framework zu den benutzten Libraries kompatibel sein muss!**

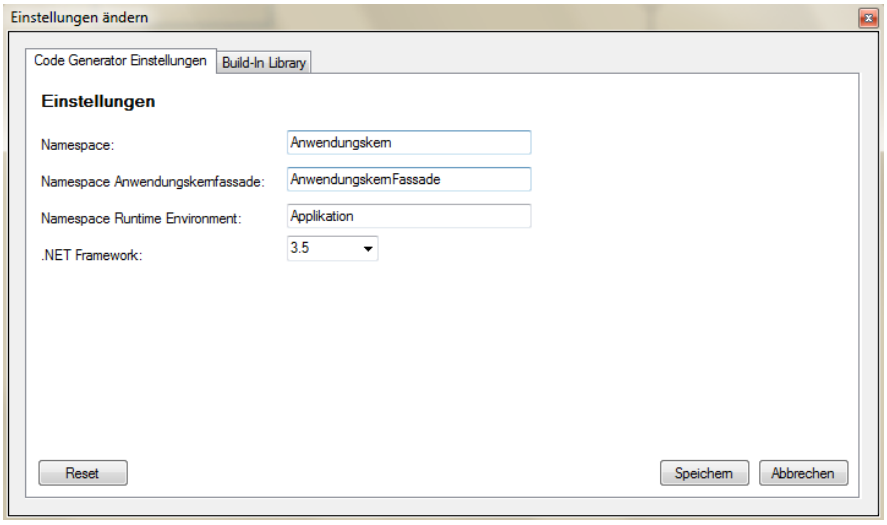


Abbildung C.1.: Einstellungen - Namespaces

C.2.2. Build-In Library

Im Bereich Build-In Library können sie verschiedene Libraries auswählen, die bei der Codeerzeugung automatisch mit eingebunden werden.

- **Autofac:** Google Autofac ist ein Dependency Injection Framework. Es dient dazu, die einzelnem Komponente automatisch zu initialisieren und zu instanziieren (Google Autofac).
- **Log4Net:** Apache Log4Net ist ein Logging-Framework. Mit diesem Framework ist es mit wenigen Handgriffen möglich, Fehler und Systemmeldungen effektiv zu registrieren und zu archivieren. Es stehen mehrere Logger zur Verfügung (Apache Log4Net).
- **NHibernate:** NHibernate ist ein O/R-Mapper. Er dient dazu, Objekte in Datenbankrelationen zu mappen (NHibernate Forge).
- **FluentNHibernate:** FluentNHibernate dient dazu, mittels FluentInterface sehr schnell und einfach Mappingklassen zu erstellen (FluentNHibernate).

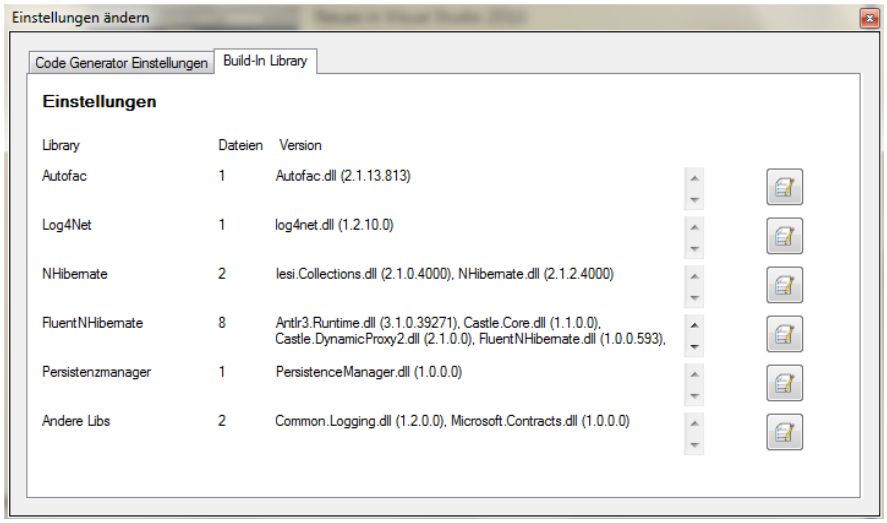


Abbildung C.2.: Einstellungen - Libraries

- **Persistenzmanager:** Ein an der HAW-Hamburg, von Professor Sarstedt, erstellter Persistenzmanager. **Wichtig: Der Persistenzmanager muss zwingend mit den hier benutzen NHibernate und FluentNHibernate Library-Versionen kompiliert worden sein. Stellen sie dies bitte sicher.**
- **Andere Libs:** Hier können sie verschiedene andere Libraries einbinden die verwendet werden sollen.

Bei der Einrichtung der Libraries ist zu beachten, dass die verschiedenen Bereiche unbedingt eingehalten werden müssen. Nach der Einrichtung ist die Quasar-Entwicklungsplattform für den Einsatz bereit.

C.3. Neue Solution anlegen

Der erste Schritt ist das Anlegen einer neuen Solution. Hierzu klicken sie auf **„CodeGenerator -> Neues Projekt anlegen“**.

Geben sie den Namen und den Pfad an, wo es gespeichert werden soll. Bei der Namenseingabe wird geprüft, ob der Name zulässig ist. Für Namen von Projekten und Projektmappen gilt folgendes:

- Sie dürfen die folgende Zeichen nicht enthalten: / ? : & * "<>| # %
- Sie dürfen keine Unicode-Steuerzeichen enthalten
- Sie dürfen keine Ersatzzeichen enthalten

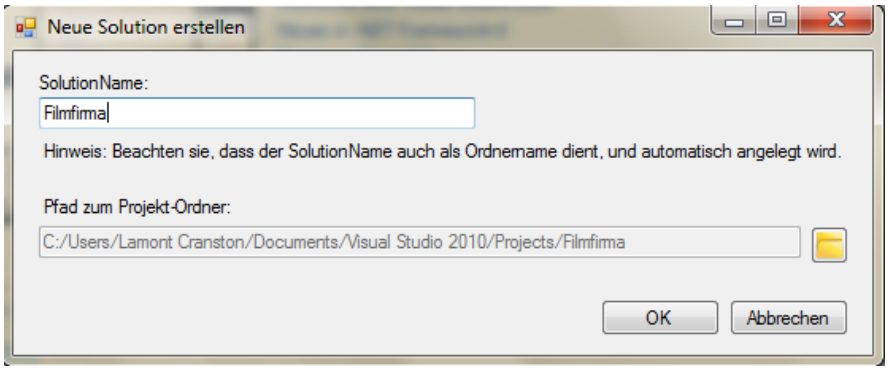


Abbildung C.3.: Neue Solution anlegen

- Sie dürfen keine für das System reservierten Namen darstellen, einschließlich „CON“, „AUX“, „PRN“, „COM1“ oder „LPT2“
- Sie dürfen nicht „.“ oder „..“ sein
- Sie dürfen nicht leer sein
- Sie dürfen nicht länger als 39 Zeichen sein

Beim Anlegen des neuen Projektes wird automatisch ein neues Modellierungsprojekt und ein Komponenendiagramm erstellt.

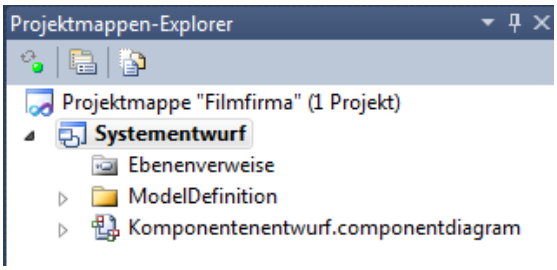


Abbildung C.4.: Neues Modellierungsprojekt

C.4. UML-Komponenten erstellen

Jetzt können sie mit der Erstellung des Entwurfes beginnen. Um eine Komponente zu erstellen machen sie entweder Rechtsklick **„Hinzufügen -> Komponente“**, oder ziehen sie per Drag & Drop ein Komponenten-Shape aus der Toolbox in das Diagramm. Anschließend erscheint ein Popup, wo sie den neuen Namen der Komponente eingeben müssen. Für Namen von Projekten und Projektitems gilt folgendes:

- Sie dürfen die folgende Zeichen nicht enthalten: / ? : & * "<>| # %
- Sie dürfen keine Unicode-Steuerzeichen enthalten
- Sie dürfen keine Ersatzzeichen enthalten
- Sie dürfen keine für das System reservierten Namen darstellen, einschließlich „CON“, „AUX“, „PRN“, „COM1“ oder „LPT2“
- Sie dürfen nicht „.“ oder „..“ sein
- Sie dürfen nicht leer sein
- Sie dürfen nicht länger als 60 Zeichen sein

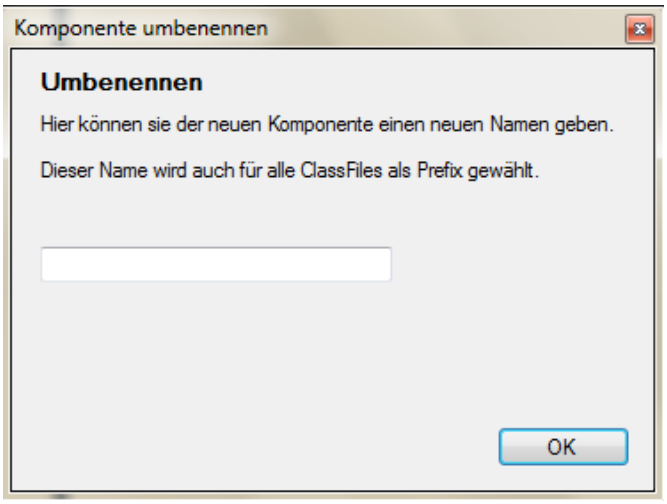


Abbildung C.5.: Neue Komponente umbenennen

Jetzt wird die Komponente vom System erstellt. Außerdem wird ein Klassendiagramm erzeugt, welches mit der Komponente über eine Referenz verbunden ist. Der Inhalt des Klassendiagramms wird automatisch vom System erstellt.

Mittels Doppelklick auf eine Komponente kann das dazugehörige Klassendiagramm geöffnet werden. Erkennbar ist eine Referenzierung an einem Ketten-Symbol in der Komponente.

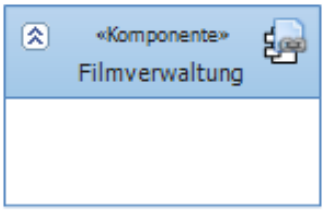


Abbildung C.6.: Verlinkte Komponente

C.5. UML-Komponente Ports hinzufügen

Damit das System erkennen kann, welche Komponenten miteinander interagieren, müssen die Komponenten mit Ports versehen werden. Fügen sie den einzelnen Komponenten ausgehende und eingehende Ports hinzu. Wenn einer Komponente ein ausgehender Port hinzugefügt worden ist, wird automatisch in der Komponenteninnensicht ein Interface erzeugt.

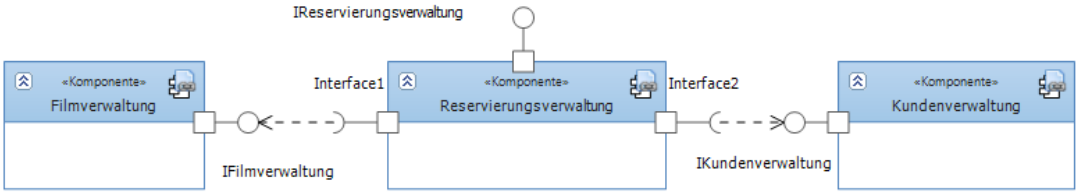


Abbildung C.7.: Verlinkte Komponente

C.6. UML-Komponenteninnensicht erstellen

Nun können die Komponenteninnensichten erstellt werden. Wie man schnell feststellt, befinden sich schon viele Elemente in der Innensicht. Lediglich Interfaces müssen noch mit einer Vererbungslinie verlinkt werden. Jetzt kann man die einzelnen Elemente füllen.

Beim Hinzufügen neuer Elemente ist es wichtig, dass immer der korrekte Stereotype ausgewählt wird. **Wichtig: Beim Umbenennen eines Elementes in der Innensicht ist darauf zu achten, dass es den korrekten Präfix besitzt. Zum Beispiel muss ein „Quasar-Anwendungsfall“ immer auf „(A/a)nwendungsfall“ enden.**

C.6.1. Interface befüllen

Im UML-Shape für ein Interface ist es notwendig, verschiedene Vorgänge zu implementieren. Wir nehmen hier als Beispiel die Methode „AddFilm“ im Interface IFilmverwaltung.

- **Returntype:** Zuerst setzten wir den Returntype. Er ist entweder, wie hier im Beispiel eine FilmEntität, aber auch ein int, string, eine IList<> oder jeder andere beliebige Rückgabetyp. Hier kann man auch eine Returntype-Description festlegen z.B. (Gibt eine neue Entität Film zurück). Diese taucht in der Interfacebeschreibung auf.
- **Parameter:** Danach werden die Parameter festgelgt. Dies sind die Werte, mit denen die Funktion aufgerufen wird. Auch hier ist es wichtig, den Namen und den Typ des Parameters anzugeben.

- **Description:** In der Beschreibung können wir eine komplette Beschreibung des Interfaces durchführen. Diese wird auch im Code angezeigt.
- **Vorbedingungen:** Hier können verschiedene Vorbedingungen definiert werden, die in der Interfacebeschreibung angezeigt werden. Zum Beispiel wäre denkbar (Filmname != null; Filmlänge != null);
- **Nachbedingung:** Es können verschiedene Nachbedingungen definiert werden, zum Beispiel wäre denkbar (Eine neue FilmEntität befindet sich in der Datenbank).

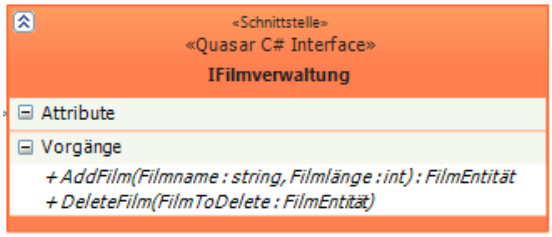


Abbildung C.8.: UML-Entwurf IFilmverwaltung

Das Interface ist jetzt vollständig modelliert und kann so verwendet werden.

C.6.2. Entität befüllen

Nun muss noch die Entität befüllt werden. Hier werden verschiedene Attribute festgelegt. Wir nehmen als Beispiel das Attribut „FilmName“ in der FilmEntität.

- **Type:** Hier müssen sie einen Type angeben. Es können primitive Datentype, wie string oder auch int angegeben werde. Es sind aber auch alle anderen Datentypen zulässig.
- **Stereotype:** Sie müssen einen Stereotype angeben. Dies ist wichtig für das Erstellen der Mappingklasse.
- **Description:** Hier kann man außerdem noch eine Beschreibung angeben.

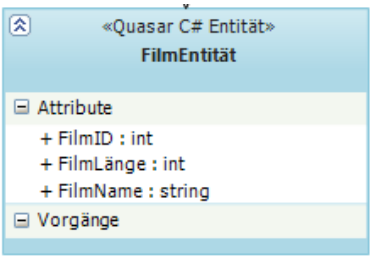


Abbildung C.9.: UML-Entwurf FilmEntität

Die FilmEntität ist jetzt komplett und kann vom System verwendet werden.

C.6.3. Gesamtentwurf

Der gesamte UML-Entwurf ist nun vollständig. Diese Prozedur muss für alle übrigen Komponenteninnensichten durchgeführt werden.

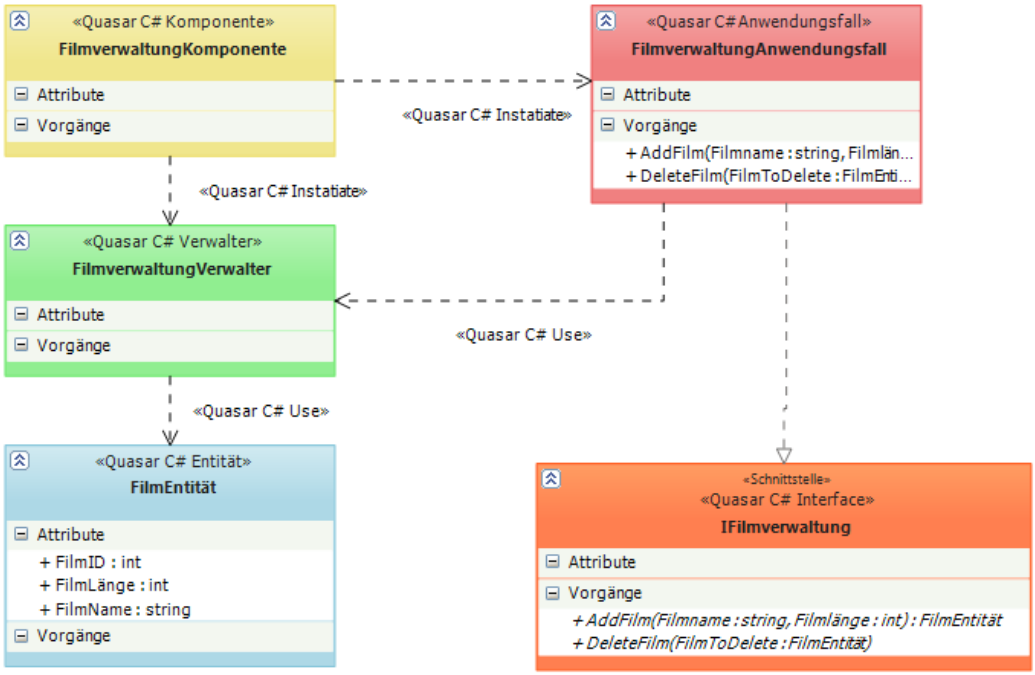


Abbildung C.10.: UML-Entwurf FilmKomponente

Der Entwurf ist nun vollständig und er kann automatisch in Code überführt werden.

C.7. Validierung

Es ist möglich, seinen Entwurf zu validieren. Diese Validierung wird vor jeder automatischen Codeerstellung durchgeführt. Sie können sie aber auch manuell starten. Klicken sie im Menü **„CodeGenerator -> Validierung -> ...“**. Hier können sie das Dateisystem, das Komponenten- oder Klassendiagramm validieren. Wenn ein Fehler auftreten sollte, wird dies gemeldet. Hier wird die Validierung exemplarisch für die Klassendiagramme durchgeführt.

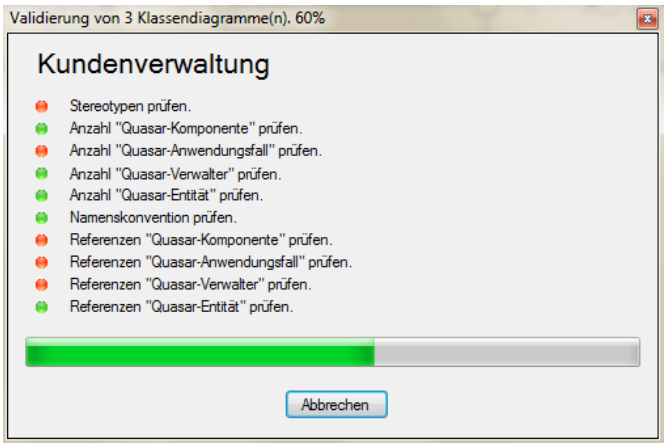


Abbildung C.11.: Validierung

In der Ausgabe des Fehlerbaums kann man nun beginnen, die Fehler genau zu analysieren.

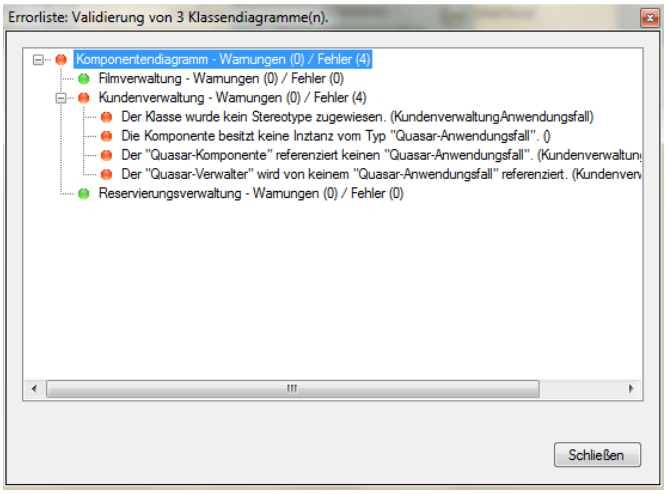


Abbildung C.12.: Fehlerausgabe Validierung

Der Fehler trat im Entwurf der Kundenverwaltung auf. Wie man erkennen kann, wurde der Klasse KundenverwaltungAnwendungsfall kein Stereotype zugewiesen. Wird dieser Fehler behoben, gibt die Validierung keinen Fehler mehr aus. Die anderen aufgelaufenen Fehler resultieren alle aus diesem einen.

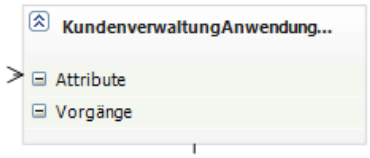


Abbildung C.13.: Fehlerhafter UML-Entwurf KundenverwaltungAnwendungsfall

C.8. Codegenerierung

C.8.1. Codeskeletons

Nach der erfolgreichen Validierung ist das System bereit Codeskeletons zu erzeugen. Sie können entweder alle Komponenten in Code umsetzen, indem sie im Menü **„CodeGenerator -> Code Skeletons generieren“** klicken. Oder sie können auch einzelne Komponenten erzeugen indem sie Rechtsklick auf die Komponente machen und **„Synchronisierung -> Generiere neuen C#Projekt Code“** klicken.

Das System durchläuft die Validierung und erzeugt den neuen Code.

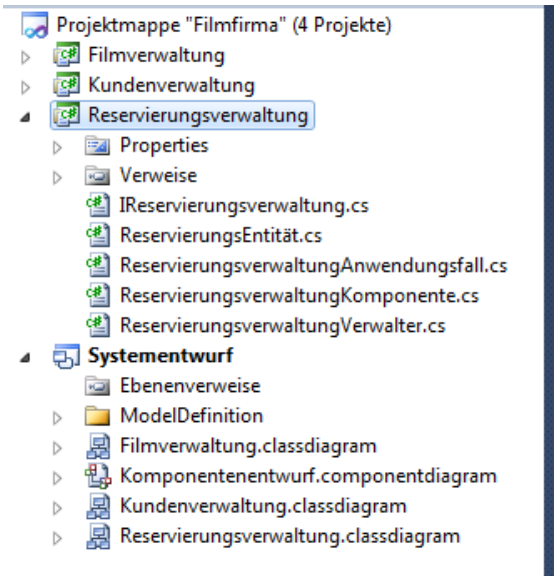


Abbildung C.14.: Neue erzeugte Projekte und Codeskeletons

C.8.2. Anwendungskernfassade

Sie können nun für ihr System eine Anwendungskernfassade erstellen. Sie bündelt alle Funktionen, die in den Interfaces implementiert worden sind. Sie können die AWK-Fassade auch nach ihren Wünschen anpassen. Zum Erstellen klicken sie im Menü **„CodeGenerator -> Generiere AWK-Fassade“**.

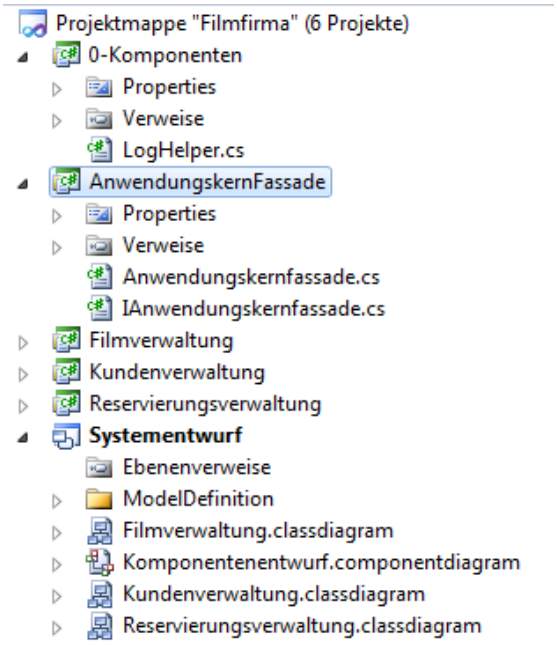


Abbildung C.15.: Neue Anwendungskernfassade

Außerdem wird eine Komponente „0-Komponenten erzeugt“, in der Helferklassen und nichtfachliche Implementationen abgelegt werden können.

C.8.3. Runtime Environment

Als letztes können sie eine Runtime Environment generieren. Dies ist eine Konsolenanwendung, die ihr Programm betriebsbereit macht. Es werden mit Hilfe des Dependency Injection Frameworks die Komponenten konfiguriert und erzeugt. Das FluentMapping wird erstellt und ausgeführt. Zum Erstellen klicken sie im Menü **„CodeGenerator -> Generiere Laufzeitumgebung“**.

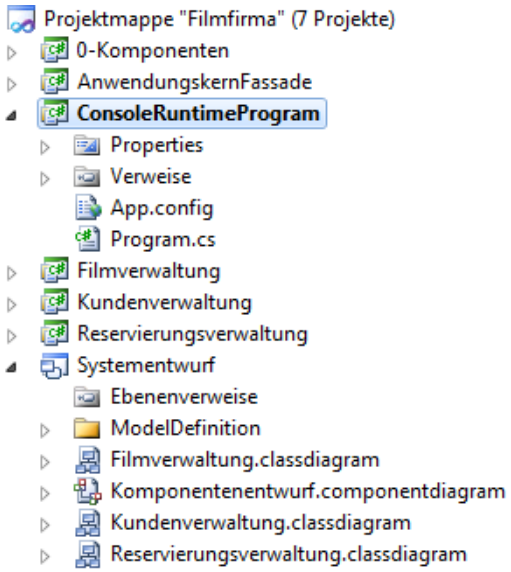


Abbildung C.16.: Neue Runtime Environment

Diese neuerzeugte Konsolenanwendung wird automatisch als Startprojekt festgelegt. Alle Codeskeletons sind erzeugt worden und sie können sie mit Code befüllen.

C.9. Synchronisation

Um die Synchronität zwischen UML-Entwurf und Code sicherzustellen, gibt es verschiedene Möglichkeiten, die Synchronisation durchzuführen. In den Kontextmenüs der UML-Shapes, der C#-Projekte und der C#-Projektitems befinden sich Einträge, mit denen die Synchronisation ausgelöst wird.

Hierbei ist zu beachten, das bestimmte Codeteile gesichert werden, andere werden überschrieben. In jedem Dokument wird genau erläutert, welche Teile betroffen sind. Bitte halten sie sich strickt an diese Anweisungen, denn sonst kann es zum Verlust ihres Codes kommen.

C.10. Weitere Funktionen

Es werden zusätzlich noch einige weitere Funktionen bereitgestellt, die über das Kontextmenü des UML-Designers abgerufen werden können.

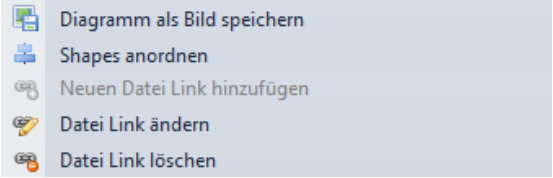


Abbildung C.17.: Weitere Funtionalität

C.10.1. Dateireferenzierung

Sie haben die Möglichkeit, vorhandene Dateireferenzierungen zu ändern oder zu löschen. Wenn keine Referenzierung vorhanden ist, können sie eine neue Referenzierung hinzufügen.

C.10.2. Bildexport

Sie haben die Möglichkeit, ihr erstelltes Diagramm direkt als Bild zu exportieren. Es stehen Formate wie „jpeg“, „png“ oder „bmp“ zur Verfügung.

C.10.3. Shapes anordnen

Es ist möglich mehrere Shapes anzuordnen. Hierzu müssen mindestens zwei Shapes ausgewählt sein. Es kann entweder eine horizontale oder eine vertikale Ausrichtung vorgenommen werden.

D. Inhalt Begleit-DVD

1. Projekt Sourcecode
2. benötigte Libraries und Packages
3. kompiliertes VSIX-Paket
4. Starterpaket einiger benötigter Libraries
5. Bachelorarbeit
6. Bedienungsanleitung
7. Programmbeispiel aus der Bedienungsanleitung

Literaturverzeichnis

[Microsoft a] Microsoft: *Definieren eines Handlers für Ablagevorgänge und Doppelklicks in einem Modellierungsdiagramm.* – URL http://msdn.microsoft.com/de-de/library/ee534033.aspx. – Zugriffsdatum: 15.11.2011

[Microsoft b] Microsoft: *Definieren eines Menübefehls in einem Modellierungsdiagramm.* – URL http://msdn.microsoft.com/de-de/library/ee329481.aspx. – Zugriffsdatum: 15.11.2011

[Microsoft c] Microsoft: *Definieren von Validierungseinschränkungen für UML-Modelle.* – URL http://msdn.microsoft.com/de-de/library/ee329482.aspx. – Zugriffsdatum: 15.11.2011

[Microsoft d] Microsoft: *Event Handlers - Propagate Changes Outside the Model.* – URL http://msdn.microsoft.com/library/bb126250.aspx. – Zugriffsdatum: 15.11.2011

[Microsoft e] Microsoft: *Managed Extensibility Framework.* – URL http://msdn.microsoft.com/de-de/library/dd460648.aspx. – Zugriffsdatum: 08.11.2011

[Microsoft f] Microsoft: *Project AddFromTemplate-Methode.* – URL http://msdn.microsoft.com/en-us/library/behck2xd(v=VS.100).aspx. – Zugriffsdatum: 15.11.2011

[Microsoft g] Microsoft: *Rules - Propagate Changes Within the Model.* – URL http://msdn.microsoft.com/library/bb126258.aspx. – Zugriffsdatum: 15.11.2011

[Microsoft h] Microsoft: *Visual Studio Extension Deployment.* – URL http://msdn.microsoft.com/de-de/library/dd393694.aspx. – Zugriffsdatum: 08.11.2011

[Sych] Sych, Oleg: *T4 and CodeDOM - Better Together.* – URL http://www.olegsych.com/2009/09/t4-and-codedom-better-together. – Zugriffsdatum: 11.11.2011

[Vogel u. a. 2008] Vogel, Oliver ; Arnold, Ingo ; Chughtai, Arif ; Ihler, Edmund ; Kehrer, Timo ; Mehlig, Uwe ; Zdun, Uwe: *Software-Architektur: Grundlagen - Konzepte - Praxis.* 2. Auflage. Spektrum Akademischer Verlag, Oktober 2008. – ISBN 3827419336

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19. Dezember 2011 Robert Madlo