

Leichtweiß-Institut für Wasserbau
Abteilung Hydromechanik und Küsteningenieurwesen
Technische Universität Carolo Wilhelmina zu Braunschweig
Prof. Dr.-Ing. Hocine Oumeraci

Entwurf und Implementierung einer nachhaltigen Softwarelösung zur Umsetzung der Empfehlungen für die Ausführungen von Küstenschutzwerken

Janis Zisenis

Matrikelnummer 4104786

24. Oktober 2016

Betreuer:

Dr.-Ing. Jan Linxweiler (iRMB)*

Kai Tegethoff, M. Eng. (LWI)

*Institut für rechnergestützte Modellierung im Bauingenieurwesen

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
Listingsverzeichnis	vii
Abkürzungsverzeichnis	ix
1 Einleitung	1
2 Grundlagen des Software Design	3
2.1 Die Software und ihre Komplexität	4
2.2 Konzepte zur Strukturierung von komplexen Systemen	5
2.3 Ziele und Metriken des objektorientierten Designs	6
2.4 Strukturierung der Komplexität durch die Objektorientierung	7
2.5 Prinzipien des objektorientierten Designs	8
2.6 Entwurfsmuster	13
2.6.1 Fabrik-Muster – Factory Pattern	13
2.6.2 Besucher-Muster – Visitor Pattern	15
2.6.3 Azyklisches Besucher-Muster – Acyclic Visitor Pattern	16
2.6.4 Befehl-Muster – Command Pattern	19
2.6.5 Beobachter-Muster – Observer Pattern	20
2.7 Architekturmuster	23
2.7.1 Model-View-Presenter (MVP) Muster – Model-View-Presenter Pattern	23

2.7.2	Passive-View-Muster – Passive View Pattern	24
2.8	Zusammenfassung	26
3	Anforderungen an die Applikation CECLOUD	27
3.1	Anforderungen aus der Problemdomäne	28
3.1.1	Organisation der Berechnungstheorien	28
3.1.2	Erweiterung der Applikation durch neue Theorien	29
3.1.3	Benutzerfreundlichkeit – Intuitive Bedienung	30
3.1.4	Plattformunabhängige Benutzung von CECLOUD	30
3.2	Softwarequalität	31
3.3	Bewerten der Qualitätsfaktoren für die Umsetzung von CECLOUD	33
3.4	Zusammenfassung	35
4	Konzept, Architektur und Technologie	36
4.1	Das Konzept der Applikation CECLOUD	37
4.2	Die Basisarchitektur	40
4.3	Verwendete Technologie	41
5	Entwurf des Modells	43
5.1	Die Berechnungshierarchie als Klassenstruktur	44
5.2	Die Projekthierarchie	46
5.3	Erzeugung der Hierarchieknoten	46
5.4	Verbinden passender Knoten	48
5.5	Bearbeitung der Knoteneigenschaften	51
5.6	Einführen einer Bearbeitungshistorie	55
6	Entwurf der Präsentation	58
6.1	Umsetzung der Basisapplikation	59
6.2	Darstellung und Manipulation der Berechnungshierarchien	64
6.2.1	Darstellung der HierarchyNodes innerhalb der Baumstruktur	64
6.2.2	Darstellung der Baumstruktur in dem Solution Explorer	67
6.2.3	Aktualisierung der Baumstruktur	68
6.2.4	Einfügen von Knoten in die Baumstruktur	69
6.2.5	Verbinden von Knoten in der Baumstruktur	71

6.3	Darstellung und Manipulation der Knoteneigenschaften	73
6.3.1	Darstellung der Knoteneigenschaften	74
6.3.2	Verändern der Knoteneigenschaften	78
6.4	Undo – Redo	79
6.5	Auswertung der Berechnungshierarchien	80
6.5.1	Erzeugung der Auswertungsfenster	80
6.5.2	Anzeigen der Auswertungsfenster über die Menüleiste	83
6.5.3	Aktualisierung der Auswertungsfenster	84
7	Erweiterung der Applikation CECLOUD	85
7.1	Die Beständigkeit der Applikationsbereiche	86
7.2	Erweiterung des Modells um neue Knotentypen	87
7.2.1	Die Berechnungshierarchie als Klassenstruktur	87
7.2.2	Erweiterung des Besucher-Musters zur Verbindung von Knoten	89
7.2.3	Erweiterung der Editor-Hierarchie	93
7.3	Erweiterung der Präsentation um neue Knotentypen	95
7.3.1	Darstellung neuer Knotentypen in der Baumstruktur	95
7.3.2	Darstellung der Eigenschaften neuer Knotentypen	96
7.3.3	Erweiterung um neue Auswertungsarten	96
8	Zusammenfassung	98
	Literaturverzeichnis	102

Abbildungsverzeichnis

2.1	Missachtung des SRP in Anlehnung an Martin u. Martin (2008).	9
2.2	Umsetzung des SRP in Anlehnung an Martin u. Martin (2008).	10
2.3	Missachtung des OCP (Martin u. Martin, 2008).	10
2.4	Umsetzung des OCP in Anlehnung an Martin u. Martin (2008).	11
2.5	Applikation erzeugt konkrete Implementierungen in Anlehnung an Martin u. Martin (2008)).	14
2.6	Factory Pattern in Anlehnung an Martin u. Martin (2008).	14
2.7	Visitor Pattern in Anlehnung an Martin u. Martin (2008).	15
2.8	Abhängigkeit der ausgeführten Operation zur Klassendefinition der Geometrie und des Besuchers.	17
2.9	Acyclic Visitor Pattern in Anlehnung an Martin u. Martin (2008).	18
2.10	Umsetzung des Command Patterns.	19
2.11	Individuelle Visualisierungen der Shapes durch unterschiedliche Views in Anlehnung an Gamma u. Riehle (2004).	21
2.12	Mögliches System zur Darstellung der Liste in SomeApplication.	21
2.13	Observer Pattern in Anlehnung an Martin u. Martin (2008).	22
2.14	Model-View-Presenter-Architektur in Anlehnung an Potel (1996).	24
2.15	Umsetzung des Passive-View-Musters in Anlehnung an Fowler (2010).	25
3.1	Hierarchie der Berechnungseinheiten zur Berechnung der Druckkräfte nach Morison.	29

3.2	Erweiterung der Applikation um eine neue Berechnungshierarchie.	30
3.3	Wechselwirkungen zwischen externen Qualitätsfaktoren nach McConnell (2009).	33
4.1	Konzeptuelle Ansicht der grafischen Benutzeroberfläche.	37
4.2	Konzeptuelle Ansicht nach dem Hinzufügen von Knoten zur Baumstruktur.	38
4.3	Konzeptuelle Darstellung der Berechnungshierarchien.	39
4.4	Konzeptuelle Darstellung der Knoteneigenschaften.	39
4.5	Architektur der Applikation CECLOUD.	40
5.1	Umsetzung der Berechnungshierarchie aus Abbildung 3.1 als Klassenstruktur.	44
5.2	Speichern der HierarchyNodes in der ProjectHierarchy während der Laufzeit der Applikation.	46
5.3	Anwendung des Factory Patterns zur Erzeugung von HierarchyNodes.	47
5.4	Binding-Mechanismus unter Verwendung des acyclic Visitor Patterns.	48
5.5	Die Umsetzung der BinderFactory zur Erzeugung von Bindern.	49
5.6	Umsetzung der Editor-Hierarchie zur Veränderung der Knoteneigenschaften.	52
5.7	Umsetzung des Command Patterns mit AddHierarchyNodeCommand, BindingCommand und EditCommand.	55
6.1	Hauptanwendungsfenster der Applikation CECLOUD.	59
6.2	Umsetzung der Basisapplikation – Änderung der Sichtbarkeiten der Explorer-Ansichten durch den ViewMenuPresenter.	60
6.3	Umsetzung der Basisapplikation – Auslösen der Aktualisierung des ViewMenus.	62
6.4	Parallele Klassenhierarchien der HierarchyNodes und HierarchicItems.	65
6.5	System zur Darstellung der Baumstruktur im Solution Explorer.	67
6.6	Aktualisierungsmechanismus der Baumstruktur im Solution Explorer.	68
6.7	Umsetzung des AddNodeMenus zur Erstellung von Knoten in der Applikation.	69
6.8	Observer-Mechanismus zwischen ProjectHierarchy, SolutionExplorerPresenter und MyQTreeModel.	70
6.9	Referenzieren des ausgewählten HierarchyNodes durch den SolutionExplorerPresenter.	71
6.10	Speichern des ausgewählten HierarchyNodes in dem SolutionExplorerPresenter.	72
6.11	System zur Darstellung der Eigenschaften in dem Properties Explorer.	74
6.12	Parallele Klassenhierarchien der HierarchyNodes und der TableRootBuilder.	75

6.13	Erzeugung und Anzeige der knotenspezifischen Eigenschaftstabelle.	77
6.14	Klassenbeziehungen zur Manipulation der Knoteneigenschaften.	78
6.15	Umsetzung des EditMenus zur Verwendung der Bearbeitungshistorie.	79
6.16	Erzeugung der Views und der Presenter durch die ViewFactory.	81
6.17	Aktualisierung des EvaluationMenus durch die Observierung des Solution- ExplorerPresenters.	83
7.1	Darstellung des abstrakten Basiskerns der Applikation CECLOUD.	86
7.2	Umsetzung der Berechnungshierarchie aus Abbildung 3.2 als Klassenstruktur. .	88
7.3	Erweiterung der HierarchyNode-Hierarchie.	89
7.4	Erweiterung des Binding-Mechanismus.	90
7.5	Erweiterung der Editor-Klassenhierarchie.	93
7.6	Erweiterung der HierarchicItem-Klassenhierarchie.	95
7.7	Erweiterung der TableRootBuilder-Klassenhierarchie.	96
7.8	Erweiterung der View-Klassenhierarchie.	97
8.1	Unterteilung der Applikationsstruktur	99

Listingsverzeichnis

5.1	Pseudocode zur Erzeugung einer Berechnungshierarchie nach Abbildung 5.1. . .	45
5.2	Quelltext der HierarchyNodeFactoryImplementation nach Abbildung 5.3. . .	47
5.3	Typumwandlung innerhalb der BinderFactory zu den konkreten Typen der HierarchyNode-Hierarchie.	50
5.4	Quelltext zur Erzeugung einer Berechnungshierarchie zur Berechnung der Linearen Wellentheorie nach Airy und Laplace unter Verwendung der HierarchyNodeFactory, der BinderFactory und des Binding-Mechanismus nach Abbildung 5.4.	51
5.5	Quelltext der Klasse CoordinateSystemEditor nach Abbildung 5.6.	53
5.6	Veränderung der Berechnungsparameter eines CoordinateSystem-Objekts un- ter Verwendung der Editor-Klassenhierarchie und der EditorFactory.	54
5.7	Quelltext der Klasse AddHierarchyNodeCommand.	56
6.1	Quelltext der main-Methode zur Umsetzung einer ausführbaren Basisapplikation. .	63
6.2	Quelltext der Klasse WaveDynamicItem.	66
6.3	Quelltext der Klasse TableRootBuilder.	76
6.4	Quelltext der Klasse CoordinateSystemRootBuilders.	76
6.5	Ausschnitt der Fabrik-Methode der MyQViewFactoryImplementation.	82
7.1	Modifikation der HierarchyNodeFactoryImplementation.	89
7.2	Umsetzung der „accept“-Methode des SedimentTransports.	91

7.3	Umsetzung der Klasse SedimentBinder.	92
7.4	Umsetzung der Klasse WaveDynamicBinder.	92
7.5	Umsetzung der Klasse SedimentEditor.	94
7.6	Umsetzung der Klasse SedimentTransportEditor.	94

Abkürzungsverzeichnis

DIP	Dependency Inversion Principle
GUI	Graphical User Interface
ISP	Interface Segregation Principle
KI	Küsteningenieurwesen
MVP	Model-View-Presenter
LSP	Liskov Substitution Principle
LWT	Lineare Wellentheorie
OCP	Open Closed Principle
OOD	objektorientiertes Design
OOP	objektorientierte Programmierung
SRP	Single Responsibility Principle

KAPITEL 1

Einleitung

Durch den heutigen Stand der Technik im Bereich der Digitalisierung und Automatisierung sowie durch die gute Verfügbarkeit an leistungsfähigen Computern nimmt die Anzahl an Softwarelösungen rasch zu. Dabei handelt es sich um Software, die an bestimmte Rahmenbedingungen gebunden ist. Diese werden durch äußere Faktoren, wie zum Beispiel Marktänderung, Kundenwünsche oder wissenschaftlichen Fortschritt bestimmt. Kontinuierlich wechselnde Anforderungen rufen ebenfalls Änderungen in der Spezifikation hervor, die Anpassungen an der Software nach sich ziehen. Aus diesem Grund muss nachhaltige Software analog zu Konzepten entwickelt werden, die es erlauben Erweiterungen und Modifikationen des Verhaltens flexibel einzuarbeiten, ohne erhebliche Umstrukturierungen der Software hervorzurufen. Wird das System durch diese Merkmale charakterisiert, handelt es sich um einen qualitativ hochwertigen Softwareentwurf.

Vor einigen Jahren entstand im Leichtweiß Institut für Wasserbau in Braunschweig die Idee, die Empfehlungen für die Ausführung von Küstenschutzwerken in Software umzusetzen, um fachspezifische Formeln und Problemstellungen schneller und automatisiert auswerten zu können. Zu diesen Berechnungen gehören unter anderem die Auswertung der Linearen Wellentheorie sowie die Berechnung von Kräften aus Wellen- und Wasserströmungen auf Bauwerke und die Gewässersohle. Bei der Umsetzung der Formeln ist zu beachten, dass die mathematischen und physikalischen Beziehungen miteinander gekoppelt sind. Aus diesem Grund ist die

Applikation so zu gestalten, dass verschiedene Einzeltheorien kombinierbar sind, sodass Ergebnisse einer Berechnung auch als Eingangswerte für eine andere dienen können. Da es sich im Küsteningenieurwesen um einen dynamischen Bereich handelt, in dem kontinuierlich neue Berechnungsansätze verfolgt werden, ist es von besonderer Bedeutung die Applikation flexibel und erweiterbar zu gestalten. Dadurch wird sichergestellt, dass die Applikation jederzeit an den neuesten Stand der Forschung angepasst werden kann.

Innerhalb dieser Arbeit wird eine mögliche Umsetzung der Applikation CECLOUD¹ gezeigt. Die tatsächliche Implementierung kann in einigen Abschnitten abweichen. Das Ziel besteht darin, die problembezogenen Anforderungen aus dem Bereich Küsteningenieurwesen in einem System umzusetzen, das gleichzeitig die Eigenschaften eines hochwertigen Softwareentwurfs aufweist. Für das Verständnis der vorliegenden Arbeit werden grundlegende Programmierkenntnisse vorausgesetzt.

Gliederung

Zu Beginn gibt das zweite Kapitel einen Überblick über die Prinzipien und Methoden des objektorientierten Entwurfs. Dabei wird auf generelle Konzepte zur Strukturierung von Komplexität eingegangen. Aus diesen Konzepten ergeben sich die anschließend erläuterten Werkzeuge der objektorientierten Programmierung. Darunter befinden sich unter anderem die SOLID-Prinzipien sowie einige Entwurfs- und Architekturmuster. Im dritten Kapitel werden die Anforderungen und Spezifikationen der Applikation CECLOUD aufgezeigt. In diesem Zuge wird sowohl die aufzuweisende Funktionalität, als auch die allgemeine Qualität von Software erörtert. Kapitel 4 bis 6 befassen sich mit der Implementierung der Applikation, basierend auf den zuvor gewonnenen Kenntnissen. Im letzten Kapitel wird schrittweise die Erweiterung der Software um eine neue Berechnungstheorie durchgeführt. Als Beispiel dient dazu die Berechnungsformel des Sedimenttransports.

¹CECLOUD: „coastal engineering – collection of university research and design tools“

KAPITEL 2

Grundlagen des Software Design

Die Entstehung hochwertiger Software geschieht idealerweise in mehreren Teilprozessen. Der wichtigste ist dabei nach McConnell (2009) die Konstruktionsphase, da sie maßgeblich die Qualität der Software beeinflusst. Dabei handelt es sich um die tatsächliche Umsetzung des Programms in Quelltext. Jeder weitere Teilprozess der Implementierung dient allein dazu, die Qualität der Konstruktionsphase und somit auch die Qualität des Endproduktes zu verbessern (McConnell, 2009). Hochwertige Software kennzeichnet sich unter anderem auch durch eine gute Struktur des umgesetzten Systems. Der Designprozess fokussiert die Strukturierung des Quelltextes und trägt dadurch erheblich zur Qualität der Software bei.

Dieses Kapitel erläutert die Grundlagen, die zum Entwurf der Applikation CECLOUD verwendet werden. Darunter befinden sich die Konzepte und Prinzipien des objektorientierten Designs (OOD), die sich aus generellen Konzepten zur Strukturierung komplexer Systeme ableiten und zur Umsetzung eines hochwertigen Systems führen. Des Weiteren werden Entwurfs- und Architekturmuster erläutert, die auf den Prinzipien des OOD basieren.

2.1 Die Software und ihre Komplexität

Bei dem Software Design handelt es sich nach McConnell (2009) um einen nicht-deterministischen Prozess, durch den Software iterativ verbessert wird (Budgen, 2003). Ziel ist es dadurch eine Lösung zu entwickeln, die hinsichtlich ihrer Anforderungen optimiert ist. Die größte Schwierigkeit besteht dabei in der Komplexität von Software, die laut Dijkstra (1972) inhärent ist. Zu den Ursachen der Komplexität gehören nach Booch (2007):

- **Die Komplexität der Problemdomäne.** Software kann nicht weniger komplex sein als das reale System, das sie widerspiegelt. Die Problemdomäne ist geprägt von Strukturen und Beziehungen, deren Umsetzung bedeutsam für das zu lösende Problem ist. Eine Vernachlässigung einzelner Aspekte der Problemdomäne würde das System im Kern ändern (Brooks, 1987, McConnell, 2009). Diese Art der Komplexität beschreibt Brooks (1987) als eine essentielle Eigenschaft. Eine weitere Form ist die willkürliche Komplexität, welche aus Faktoren resultiert, die von außen auf die Problemdomäne einwirken. Darunter befinden sich unter anderem Anforderungen der Benutzer an die Funktionsweise des Programms.
- **Die Schwierigkeit den Entwicklungsprozess handzuhaben.** Software übersteigt durch ihren Umfang oftmals die Kapazität eines einzelnen Entwicklers. Aus diesem Grund werden Softwarelösungen meistens in Teams realisiert. Je größer dabei die Anzahl an teilnehmenden Personen, desto komplexer wird die Organisation der Gruppe. Dabei gilt als größte Herausforderung ein einheitliches und integriertes Design zu erreichen (Booch, 2007).
- **Die mögliche Flexibilität von Software.** Der Entwurf von Software bietet aufgrund ihrer hohen Flexibilität eine Vielzahl an Entscheidungsmöglichkeiten, die zur Umsetzung der Spezifikationen gewählt werden können. Die Schwierigkeit besteht für den Entwickler sich nicht nur innerhalb des Entscheidungsspielraums zu bewegen, sondern genau die Implementierung zu wählen, welche die Anforderungen am besten erfüllen (Booch, 2007).
- **Die Charakterisierung des Verhaltens diskreter Systeme.** Im Gegensatz zu kontinuierlichen Funktionen haben diskrete Systeme eine begrenzte Anzahl an möglichen Zuständen. Dabei hängt die Dimension der Zustände maßgeblich von der Größe des betrachteten Systems ab. Auch bei Software handelt es sich um eine diskrete Struktur, die nur innerhalb eines gewissen Gültigkeitsbereichs funktionsfähig ist. Die Software ist so zu entwerfen, dass sie sich zu jeder Zeit in einem gültigen Zustand befindet (Booch, 2007).

2.2 Konzepte zur Strukturierung von komplexen Systemen

Unabhängig von der Softwareentwicklung bestehen allgemeine Konzepte zur Organisation komplexer Strukturen. Diese erlauben dem Menschen eine Komplexität zu beherrschen, die seinen intellektuellen Grad übersteigt. Unter diesen Methoden befinden sich die Dekomposition, die Abstraktion und die Hierarchie.

Dekomposition

Zur Bewältigung der Komplexität findet nach Booch (2007) die Dekomposition Anwendung. Das Gesamtsystem wird dabei in mehrere Subsysteme unterteilt. Dadurch entstehen Bestandteile deren Zusammenwirken die Gesamtkomplexität der Problemdomäne widerspiegeln. Die Dekomposition wird auf die einzelnen Bestandteile solange rekursiv angewendet, bis die kleinsten Elemente durch ihre Einfachheit der menschlichen Kognition genügen. Durch die Dekomposition kann nach Parnas (1972) direkt die inhärente Komplexität von Software angesprochen werden.

Abstraktion

Wie aus einem Experiment von Miller (1956) hervorgeht, kann sich ein Mensch nur auf ca. sieben verschiedene Informationen konzentrieren. Diese Anzahl kann während des Entwurfs von Software schnell überschritten werden. Da nicht alle Informationen der gesamten Komplexität in jedem Kontext von Bedeutung sind, können unwichtige Bestandteile außer Betracht gelassen werden (Booch, 2007). Nach Wulf et al. (1981) hat der Mensch die Technik der Abstraktion entwickelt, um einen zu hohen Informationsgehalt an den momentanen Kontext anzupassen. Dabei entstehen einzelne Abstraktionsniveaus, die unabhängig voneinander verständlich sind.

Hierarchie

Die Organisation in hierarchischen Strukturen trägt zum besseren Verständnis der Komplexität bei. Dabei sind nach Booch (2007) mehrere Arten von Hierarchien vertreten. Durch Anwendung der Dekomposition und der Abstraktion entstehen strukturelle Hierarchien sowie Verhaltenshierarchien, durch welche Kommunikations- und Verhaltensstrukturen verdeutlicht werden. Jedes Hierarchielevel spiegelt dabei ein Abstraktionsniveau wider.

2.3 Ziele und Metriken des objektorientierten Designs

Ein häufig verwendeter Ansatz zur Implementierung von Software ist die objektorientierte Programmierung (OOP) (Booch, 2007). Die Grundidee dieses Ansatzes ist die Modellierung von realen Systemen zu Programmabläufen mithilfe von Klassen und Objekten. Dabei basiert jedes Objekt auf einer Klassendefinition, in der Daten und Verhalten in Form von Attributen und Methoden festgelegt werden. Alle Objekte verhalten sich im Programmkontext gemäß der Spezifikation ihrer Klassendefinitionen. Es entstehen kleine logische Einheiten, die eine bestimmte Aufgabe der Software übernehmen. Durch Beziehungen zwischen Objekten können diese miteinander interagieren, wodurch komplexe Strukturen, auch Module genannt, erzeugt werden können (Booch, 2007).

»Object-oriented design is about managing dependencies«

Metz (2013)

Um die Interaktion zwischen Objekten zu ermöglichen, müssen sie die Spezifikationen des jeweiligen Kommunikationspartners kennen, sie hängen von ihnen ab. Durch Veränderungen der Spezifikation eines Objektes werden abhängige Objekte gezwungen sich dem neuen Verhalten anzupassen. Dadurch ergibt sich eine Kaskade von Veränderungen in entgegengesetzter Richtung der Abhängigkeiten (Martin, 2000). Um die Software wartbar und flexibel zu gestalten, ist es unbedingt notwendig die Kette von Veränderungen im Quelltext so kurz wie möglich zu halten.

Die Beurteilung der Qualität von Software hinsichtlich der Abhängigkeiten erfolgt über die Metriken Kohäsion und Kopplung von Yourdon u. Constantine (1979).

Kopplung. Hierbei handelt es sich um ein Maß, das die Stärke der Abhängigkeit zweier Module beschreibt. Je geringer dabei die Verbindungen zwischen zwei Modulen, desto loser ist die Kopplung.

Kohäsion. Die Kohäsion ist ein Maß für die Zusammengehörigkeit der Daten innerhalb eines Moduls.

Das Ziel des OOD liegt darin, die Abhängigkeiten zwischen Modulen bzw. Objekten zu minimieren, sodass eine möglichst lose Kopplung mit gleichzeitig hoher Kohäsion der Software entsteht (Metz, 2013).

2.4 Strukturierung der Komplexität durch die Objektorientierung

Die inhärente Komplexität der Software übersteigt nach Booch (2007) die intellektuelle Kapazität des Menschen. Daher werden auch in der Softwareentwicklung die in Abschnitt 2.2 erläuterten Konzepte herangezogen. Durch die Wahl der Objektorientierung können die Methoden dieses Ansatzes verwendet werden, die auf den zuvor beschriebenen allgemeinen Konzepten basieren und gleichzeitig die Kohäsion und die Kopplung berücksichtigen.

Zu den Methoden der OOD zur Strukturierung der Softwarekomplexität gehören die Abstraktion, die Datenkapselung, die Modularisierung sowie die Hierarchie.

Abstraktion

Wie in Abschnitt 2.2 beschrieben kann durch die Abstraktion der Informationsgehalt an den momentanen Kontext angepasst werden, indem unwichtige Details vernachlässigt werden. Dies wird in der OOP durch Schnittstellen und Vererbung realisiert. Dabei wird das zu abstrahierende Objekt im Hinblick seiner Funktionalität betrachtet, die es nach außen aufweisen muss. Die Anwendung der Abstraktion in der OOP ruft Schnittstellen für Objekte und Module hervor, die nach Booch (2007) nur das essentielle Verhalten für den jeweiligen Kontext umfassen. Die äußere Wahrnehmung wird von der internen Implementierung getrennt, wodurch eine lose Kopplung zu abhängigen Modulen entsteht.

Datenkapselung

Im Gegensatz zu der Abstraktion fokussiert die Datenkapselung die innere Repräsentation. Nach diesem Konzept sollte die interne Struktur von Objekten und Modulen verborgen sein, um die Komplexität des Gesamtsystems zu verringern (McConnell, 2009). Basieren Bestandteile der Software nur auf dem äußeren Verhalten von anderen Bestandteilen, vermindert sich die Abhängigkeiten zwischen ihnen (Booch, 2007). Dadurch kann die innere Implementierung von Objekten und Modulen zuverlässig verändert werden, ohne dass die Modifikation nach außen durchdringt. Des Weiteren hilft die Datenkapselung dabei die Komplexität des Entwurfsprozesses zu verringern, indem sie dem Entwickler die Möglichkeit gibt die interne und die externe Repräsentation getrennt voneinander zu entwerfen.

Modularisierung

Durch die Dekomposition lässt sich die Gesamtkomplexität eines Systems in mehrere Komponenten geringerer Komplexität unterteilen. Dabei entstehen Elemente deren Zusammenwirken das Gesamtsystem ergeben (Dijkstra, 1979). Inhaltlich verwandte Abstraktionen werden durch Module zusammengefasst und strukturiert. Dadurch steigt die Kohäsion des Systems. Die Modularisierung vereinfacht den Entwurf von Software, da durch sie autonome Subsysteme entstehen. Dabei ist die Kenntnis über das Verhalten und die Struktur anderer Module nicht nötig. Durch die Modularisierung kann die Software als Gefüge verschiedener Einzelteile betrachtet werden, die unabhängig voneinander ausgetauscht und modifiziert werden können (Booch, 2007).

Hierarchie

Wie in Abschnitt 2.2 beschrieben, verbessert die Verwendung von hierarchischen Strukturen das Verständnis der Komplexität. Die zuvor erwähnte strukturelle Hierarchie wird durch Aggregation von Objekten umgesetzt, die Verhaltenshierarchie durch die Verwendung von Vererbung (Booch, 2007).

2.5 Prinzipien des objektorientierten Designs

Die kontinuierliche Anwendung der in Abschnitt 2.4 beschriebenen Methoden hat aus gesammelten Erfahrungen eine Reihe von Entwurfsprinzipien hervorgebracht, welche die Softwareentwickler in der Ausarbeitung des Designs unterstützen.

Wie Brooks (1987) in seinem Paper *No Silver Bullet* erwähnt, gibt es in der Softwareentwicklung keinen allgemeingültigen Weg, der zu einem System mit perfektem Design führt. Auch die Umsetzung dieser Prinzipien ist deshalb kein Wundermittel gegen schlechtes Arrangement der objektorientierten Elemente. Sie sind als Richtlinien zu betrachten, die bei der Entscheidungsfindung zu berücksichtigen sind.

Zu den bekanntesten Prinzipien des OOD gehören die *SOLID Principles* (Martin u. Martin, 2008). Dabei steht die Abkürzung *SOLID* für:

- *Single Responsibility Principle*
- *Open-Closed Principle*

- *Liskov Substitution Principle*
- *Interface Segregation Principle*
- *Dependency Inversion Principle*

Single Responsibility Principle (SRP)

Wie in Abschnitt 2.3 beschrieben, führen Änderungen in Klassen zu Anpassungen in abhängigen Klassen. Um ein stabiles Design zu erzeugen ist es von Bedeutung die Auswirkungen der Modifikation möglichst lokal zu halten. Nach Martin u. Martin (2008) kann das durch Einhalten des SRP erreicht werden. Das SRP fordert dazu auf, Klassen zu entwerfen, die nur **eine** Verantwortung übernehmen. Abbildung 2.1 stellt ein System dar, das gegen das SRP verstößt.

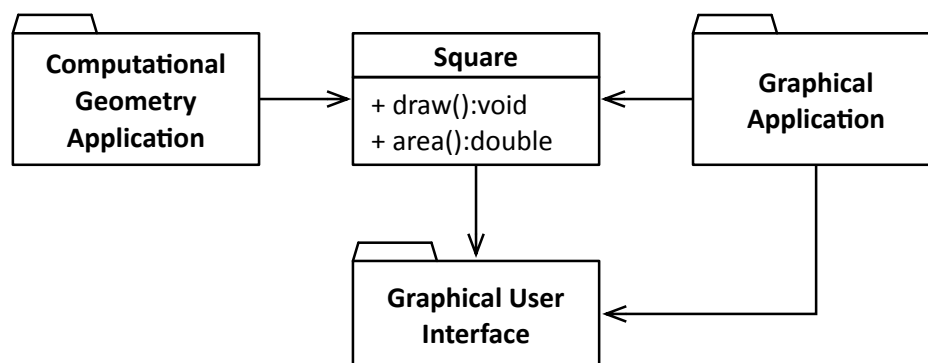


Abbildung 2.1: Missachtung des SRP in Anlehnung an Martin u. Martin (2008).

Die Klasse `Square` trägt zwei Verantwortungen in sich. Sie hat zum einen die Funktionalität sich auf dem `Graphical User Interface (GUI)`¹ zu zeichnen („draw“), zum anderen kann sie ihren Flächeninhalt berechnen und ausgeben („area“). Die Zeichenoperation gehört inhaltlich zu dem Modul `Graphical Application`, die Rechenoperation zu `Computational Geometry Application`. Beide Module sind abhängig von `Square`. Werden in dieser Klasse Änderungen vorgenommen, müssen sie sich anpassen. Dadurch werden inhaltlich nicht verwandte Bereiche gekoppelt. Der Verstoß gegen das SRP erzeugt fragiles Design. Kleine Änderungen im Quelltext können eine Vielzahl an Anpassungen hervorrufen, die über die gesamte Applikation verstreut sind.

Abbildung 2.2 zeigt die Entkopplung der beiden Module. Damit `Square` nach dem SRP nicht mehr als eine Verantwortung beinhaltet, wird sie in die Klassen `Geometric Square` und

¹deutsch: grafische Benutzeroberfläche

Graphic Square unterteilt. Die Rechenoperation wird von Geometric Square übernommen, die Zeichenoperation von Graphic Square. Beide Klassen besitzen nun nicht mehr als einen Grund sich zu ändern. Erfordert z.B. die grafische Seite der Applikation eine Veränderung der Zeichenoperation, können die Anpassungen auf den grafischen Bereich eingegrenzt werden.

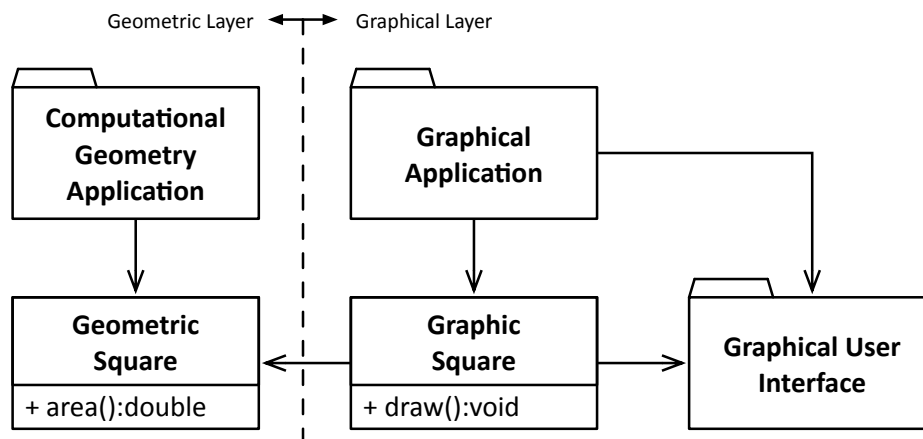


Abbildung 2.2: Umsetzung des SRP in Anlehnung an Martin u. Martin (2008).

Open Closed Principle (OCP)

Nach dem OCP sollen Module entworfen werden, die nicht modifizierbar sind, jedoch gleichzeitig eine Erweiterung erlauben. Neues Verhalten soll durch das Hinzufügen von Quelltext entstehen, nicht durch die Veränderung des bereits Bestehenden (Meyer, 2009). Die Umsetzung des OCP erfolgt durch die Verwendung der Abstraktion im objektorientierten Sinn (Martin u. Martin, 2008). Abbildung 2.3 zeigt die Umsetzung eines Moduls unter Missachtung des OCP.

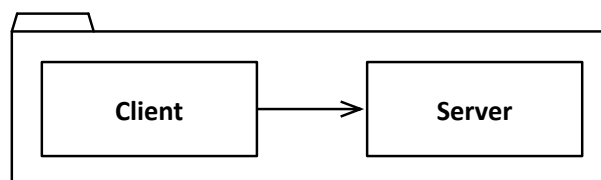


Abbildung 2.3: Missachtung des OCP (Martin u. Martin, 2008).

Bei dem **Client** handelt es sich um das Objekt, das Operationen des **Servers** auslöst. Dabei ist die Abhängigkeit des **Clients** in Richtung einer konkreten Klasse gerichtet. Ohne die Methoden der Klasse **Server** zu modifizieren, ist es nicht möglich das Verhalten des Moduls zu

erweitern. Eine Veränderung des Servers würde eine Anpassung des Clients nach sich ziehen. Das Modul ist somit weder offen für Erweiterungen, noch geschlossen für Veränderungen. Um die Vorgaben des OCP einzuhalten, kann das Modul wie in Abbildung 2.4 umgesetzt werden.

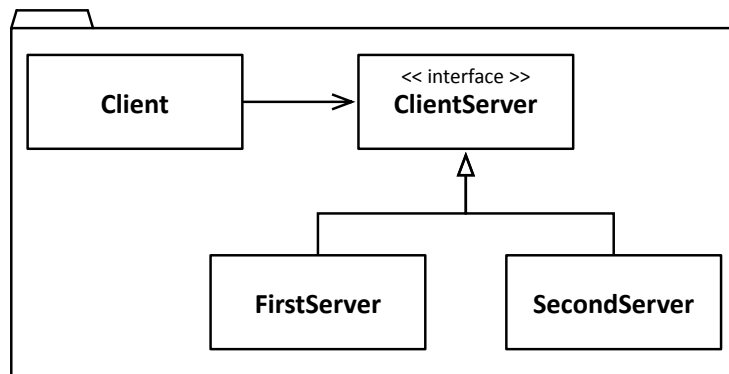


Abbildung 2.4: Umsetzung des OCP in Anlehnung an Martin u. Martin (2008).

Die Klasse `Client` definiert dabei eine abstrakte Schnittstelle `ClientServer`. Hier legt der `Client` das Verhalten fest, das er von einem möglichen Kommunikationspartner erwartet. Die Vererbung kann das nötige Verhalten an konkrete Klassen weitergeben.

Durch Polymorphie kann jedes Derivat der Schnittstelle `ClientServer` von dem `Client` benutzt werden. Die Implementierung des Verhaltens wird hinter der Schnittstelle verborgen und kann in den Subklassen variieren (Martin u. Martin, 2008). Für eine Erweiterung des Moduls werden neue Derivate von `ClientServer` erstellt, die das Verhalten auf ihre individuelle Weise implementieren. Hierfür ist keine Änderung des bereits vorhandenen Quelltextes notwendig. Das Modul ist offen für Erweiterungen und geschlossen für Modifikationen.

Polymorphie. Durch Vererbung entstehen Klassenhierarchien, die durch eine Basisklasse repräsentiert werden. Dabei wird ihr Verhalten an Subklassen vererbt. Die Polymorphie erlaubt es Objekte des Subtyps anstelle des Basistyps zu verwenden. Jedes Derivat kann als Typ der Basisklasse betrachtet werden. In Abbildung 2.4 können z.B. Objekte des Typs `FirstServer` und `SecondServer` ebenfalls als Objekt des Typs `ClientServer` betrachtet werden (Martin u. Martin, 2008).

Liskov Substitution Principle (LSP)

Nach Martin u. Martin (2008) ist die Polymorphie einer der mächtigsten Mechanismen der OOP. Sie erlaubt Flexibilität der Software durch dynamische Variation der Objekttypen innerhalb von Klassenhierarchien. Dies wird durch die Vererbung von Verhalten ermöglicht. Zugleich trennt die Verwendung von Schnittstellen das öffentliche Verhalten von der inneren Implementierung. In Abbildung 2.4 legt der `Client` das Verhalten fest, das er von einem möglichen `ClientServer` erwartet. Die Implementierung der Methoden geschieht in den konkreten Klassen `FirstServer` und `SecondServer`. Das LSP fordert die Umsetzung des Verhaltens entsprechend der Erwartungen des `Clients`, damit ein gefahrloser Austausch der Subtypen stattfinden kann. Eine dynamische Typvariation darf den Programmablauf des `Clients` nicht beeinflussen (Liskov, 1987).

Interface Segregation Principle (ISP)

Die Umsetzung der Datenkapselung geschieht mithilfe von abstrakten Schnittstellen. Um die interne Struktur von Objekten bzw. Modulen zu verbergen, werden *Interfaces* erstellt, über die ein Aufrufer Zugang zu Methoden von Objekten erhält, deren konkrete Typen nicht bekannt sind. Werden Methoden eines Objekts von unterschiedlichen Aufrufern ausgelöst, ist es nach dem ISP notwendig viele individuelle Schnittstellen zu gestalten, die den Anforderungen des jeweiligen Aufrufers angepasst sind. Wird stattdessen nur ein *Interface* erzeugt, in dem die verschiedenen Verhaltensweisen akkumuliert werden, entsteht analog zum SRP eine Kopplung der unterschiedlichen Aufrufer. Sie hängen dadurch von Verhalten ab, das sie nicht benötigen (Martin u. Martin, 2008).

Dependency Inversion Principle (DIP)

Das DIP von Martin u. Martin (2008) fordert die Umkehrung der Abhängigkeiten zwischen Objekten. In Abbildung 2.3 hängt der `Client` direkt vom dem `Server` ab. Handelt es sich bei dem `Server` um einen volatilen Bereich der Applikation, ist der `Client` einem hohen Anpassungsbedarf ausgesetzt. Nach Martin u. Martin (2008) ist die Abhängigkeit zu konkreten Klassen zu vermeiden. Stattdessen sollen sie auf Abstraktionen basieren. Wird das System wie in Abbildung 2.4 dargestellt entworfen, kann der volatile Bestandteil hinter der Abstraktion verborgen werden. Die Definition des erwarteten Verhaltens durch den `Client` in der

Schnittstelle `ClientServer` gibt den konkreten Klassen `FirstServer` und `SecondServer` Erwartungen vor, denen sie sich anpassen müssen. Die Abhängigkeit wurde in Richtung des Clients umgekehrt.

2.6 Entwurfsmuster

Der Entwurf von Software nach den Prinzipien aus Abschnitt 2.5 zeigte in der Vergangenheit bei strukturell ähnlichen Problemen immer wiederkehrende Anordnungen von Klassen. Diese häufig auftretenden Strukturen konnten isoliert und spezifischen Problemen zugeordnet werden. Dadurch ist eine Vielzahl von Entwurfsmustern entstanden, die als Schablonen für die Umsetzung verschiedener Mechanismen dienen. Sie beschreiben die Anordnung sowie die Kommunikationsstrukturen der Klassen als eine Kernlösung von strukturell ähnlichen Problemen, losgelöst von dem inhaltlichen Kontext in der Applikation (Buschmann, F. et al., 1996, Gamma u. Riehle, 2004). Durch die Verwendung und Kombination von Entwurfsmustern entsteht Software, die den zuvor beschriebenen Prinzipien entspricht (Martin u. Martin, 2008).

2.6.1 Fabrik-Muster – Factory Pattern

Gemäß des DIP sind Abhängigkeiten zu Abstraktionen zu bevorzugen. Hängen Module oder Klassen von volatilen Implementierungen ab, kann es zu einem hohen Anpassungsbedarf kommen. Basieren sie auf Abstraktionen haben Änderungen in den konkreten Implementierungen keine Auswirkung auf das Modul. Die Abstraktionen werden dann zur Laufzeit unter Anwendung der Polymorphie durch Objekte der Subtypen ersetzt. Durch die Verwendung der Abstraktion wird ebenfalls das OCP umgesetzt. Bevor Objekte der konkreten Klassen verwendet werden können, müssen diese durch den Konstruktor erzeugt werden. Dabei spielt der Ort der Konstruktion eine bedeutende Rolle, da das Erstellen eines Objektes zu einer starken Abhängigkeit führt (Martin u. Martin, 2008). Abbildung 2.5 zeigt ein System in dem `SomeApplication` die konkreten Klassen `Square`, `Circle` und `Triangle` verwendet. Zur Erfüllung des DIP definiert `SomeApplication` eine abstrakte Schnittstelle `Shape`, von der die konkreten Klassen abgeleitet werden. `Square`, `Circle` und `Triangle` können benutzt werden, ohne von ihnen abzuhängen. Die Erzeugung der Objekte übernimmt die Klasse `SomeApplication`, wodurch wiederum die Abhängigkeiten zu den Derivaten der Schnittstelle eingeführt werden. Dadurch werden sowohl das DIP, als auch das OCP verletzt (Martin u. Martin, 2008).

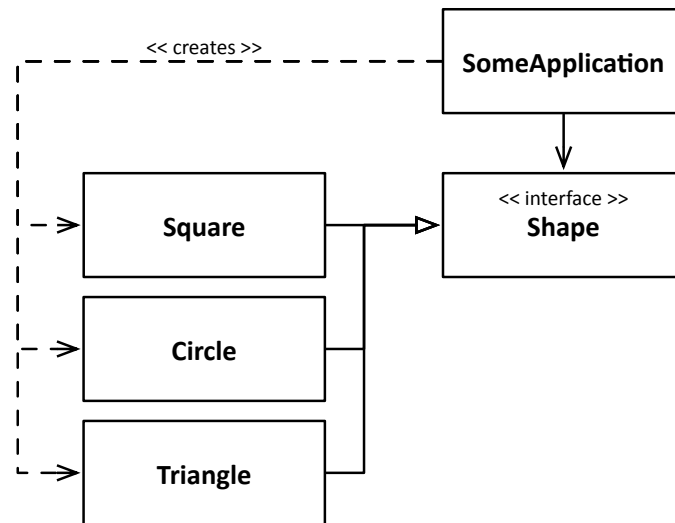


Abbildung 2.5: Applikation erzeugt konkrete Implementierungen in Anlehnung an Martin u. Martin (2008)).

Zur Wahrung des DIP kann das System wie in Abbildung 2.6 implementiert werden. Die Erzeugung der Objekte wird in die Klasse **ShapeFactory Implementation** ausgelagert. Damit auch an dieser Stelle **SomeApplication** nicht auf konkreten Klassen basiert, wird die Schnittstelle **ShapeFactory** eingeführt. Die Konstruktion eines Objektes kann **SomeApplication** an die **ShapeFactory Implementation** delegieren. Dazu werden die Methoden „makeSquare“, „makeCircle“ und „makeTriangle“ benutzt. Die Fabrik erstellt ein Objekt von dem geforderten Typ und händigt es als **Shape** aus. Dadurch können die Typinformationen vor der Klasse

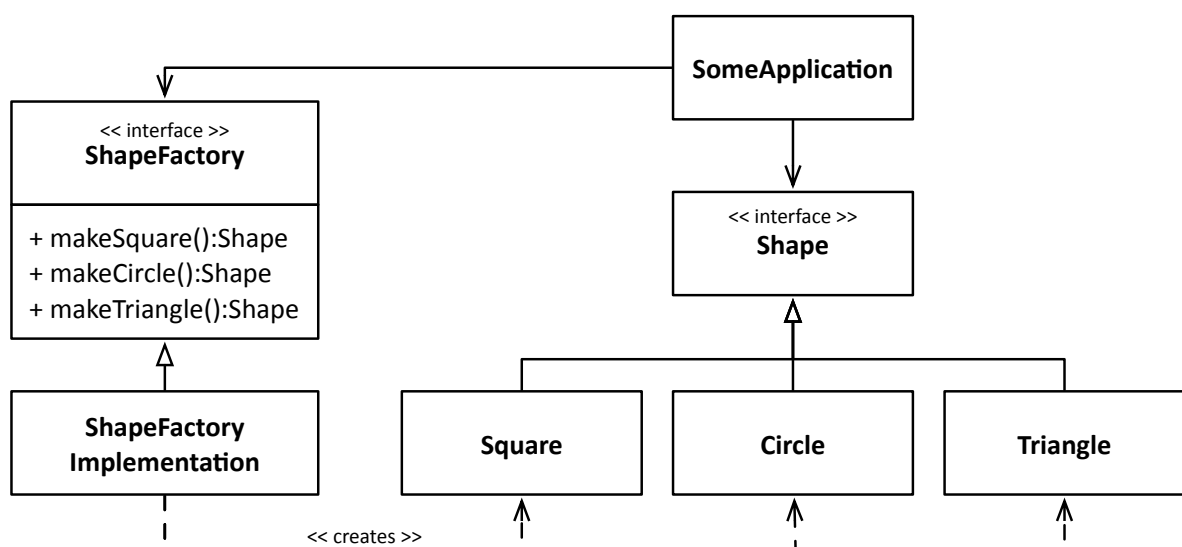


Abbildung 2.6: Factory Pattern in Anlehnung an Martin u. Martin (2008).

SomeApplication verborgen werden. Die Abhängigkeit beschränkt sich dadurch nur auf die unterschiedlichen Methodennamen der Fabrik. Sie wird auf ein Minimum reduziert (Gamma u. Riehle, 2004, Martin u. Martin, 2008).

2.6.2 Besucher-Muster – Visitor Pattern

Änderungen in der Spezifikation können Modifikationen der Software hervorrufen. Nach dem OCP ist ein Design zu wählen, dass eine Erweiterung des Programms erlaubt, ohne vorhandenen Quelltext zu bearbeiten.

Zur Erweiterung einer Klassenhierarchie hinsichtlich ihrer Funktionalität kann die Basisklasse um neue Methoden ergänzt werden. Die in Abbildung 2.6 dargestellten Derivate der abstrakten Schnittstelle Shape sollen beispielsweise um die Verschiebungsoperation „move“ erweitert werden. Damit SomeApplication diese Funktionalität benutzen kann, ist eine Erweiterung der Schnittstelle Shape notwendig. Durch die Abhängigkeit zu der abstrakten Schnittstelle der Klassenhierarchie muss sich SomeApplication anpassen. Das OCP wird verletzt.

Die Anwendung des Visitor Patterns nach Abbildung 2.7 erlaubt eine Erweiterung der Funktionalität von Klassenhierarchien ohne deren Änderung (Gamma u. Riehle, 2004, Martin u. Martin, 2008). Dadurch kann die abstrakte Schnittstelle Shape entsprechend des OCP geschlossen werden. Zur Implementierung dieses Musters wird eine weitere Klassenhierarchie eingeführt. Die Derivate der Basisklasse ShapeVisitor kapseln die Funktionalität, durch welche die Hier-

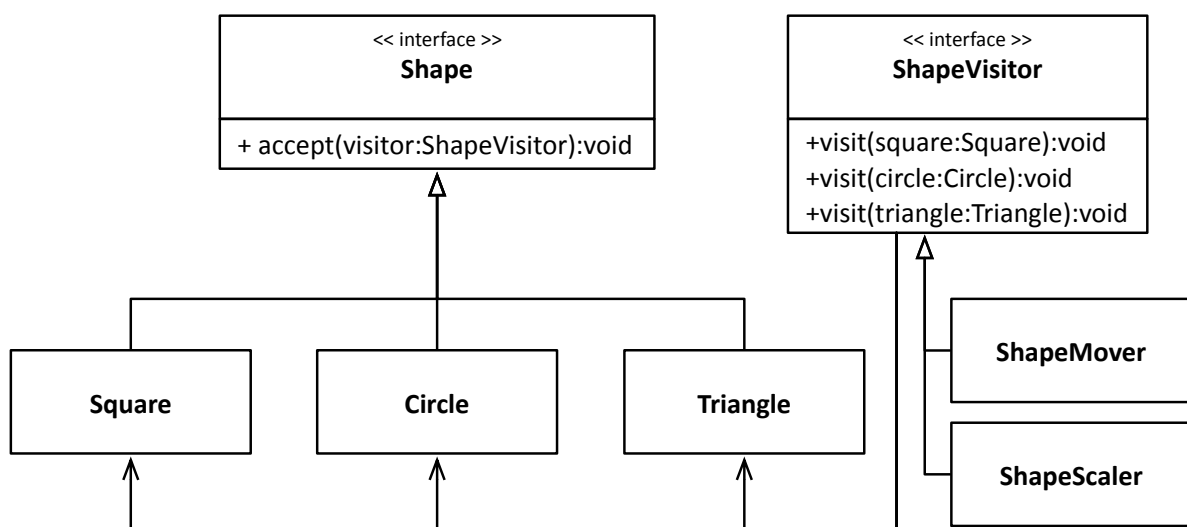


Abbildung 2.7: Visitor Pattern in Anlehnung an Martin u. Martin (2008).

archie erweitert werden soll, die sich hinter Shape befindet. So ist der ShapeMover für die Verschiebung, der ShapeScaler für die Skalierung der verschiedenen Geometrien zuständig. Operationen der Besucherhierarchie werden durch SomeApplication ausgelöst, indem sie die polymorphe „accept“-Methode der Geometrien aufruft. Dabei übergibt sie als Parameter ein Objekt des Typs ShapeVisitor. Entsprechend ihrer Klassendefinition ruft die angesprochene Geometrie die überladene „visit“-Methode des übergebenen ShapeVisitors auf, die sich ebenfalls polymorph verhält. Verbirgt sich hinter dem Besucher ein ShapeMover, wird die Geometrie verschoben, handelt es sich um einen ShapeScaler, wird eine Skalierungsoperation ausgeführt.

Bei dem Aufruf der „accept“-Methode handelt es sich nach Gamma u. Riehle (2004) um einen sogenannten *double dispatch*-Mechanismus², da das Ergebnis dieser Operationen von den beiden konkreten Subtypen der verschiedenen Klassenhierarchien abhängt. Die auszuführende Operation ist zum Zeitpunkt des Auslösens innerhalb der Klasse SomeApplication noch nicht bekannt. Durch die Verknüpfung dieser beiden polymorphen Methoden entsteht die zweidimensionale Verhaltensmatrix nach Abbildung 2.8 (Martin u. Martin, 2008).

2.6.3 Azyklisches Besucher-Muster – Acyclic Visitor Pattern

Die Anwendung des allgemeinen Visitor Patterns aus Abschnitt 2.6.2 verursacht eine zirkuläre Abhängigkeit der in Abbildung 2.7 dargestellten Komponenten des Entwurfsmusters. Die Basisklasse Shape hängt von der Schnittstelle der Besucherhierarchie ab. Diese basiert wiederum auf den verschiedenen Derivaten von Shape. Es entsteht eine enge Kopplung. Jede Erweiterung der Shape-Hierarchie zieht eine Modifikation der gesamten Besucherhierarchie nach sich. Dadurch wird es erschwert neue Derivate der Klasse Shape hinzuzufügen. Handelt es sich bei der ursprünglichen Klassenhierarchie um einen volatilen Bestandteil der Software, kann das Azyklische Besucher-Muster nach Abbildung 2.9 verwendet werden (Martin u. Martin, 2008).

Die Nutzung des acyclic Visitor Pattern verhindert die zirkuläre Abhängigkeit, in dem es die Basisklasse der Besucherhierarchie als degenerierte Form verwendet. Bei ShapeVisitor handelt es sich deswegen um eine leere Klasse ohne Eigenschaften und Verhalten. Dadurch kann die Abhängigkeit der Besucher zu den konkreten Klassen Square, Circle und Triangle vermieden werden. Des Weiteren beinhaltet dieses Entwurfsmuster spezielle Besucherschnittstellen für jede Subklasse von Shape. Die konkreten Besucher erben zusätzlich von diesen Schnittstel-

²nach Martin u. Martin (2008) auch *dual dispatch*-Mechanismus genannt

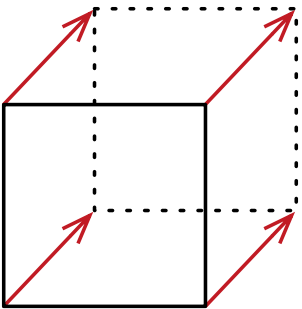
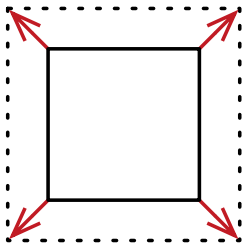
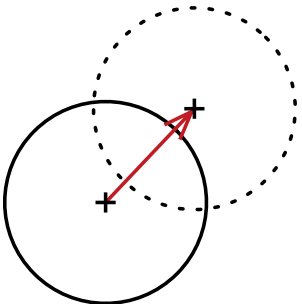
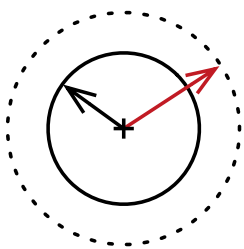
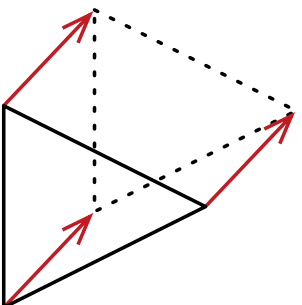
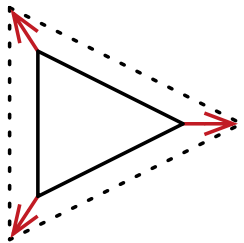
Ausgeführte Operation in Abhängigkeit der Geometrie (unten) und des Besuchers (rechts)	Klassendefinition „ShapeMover“	Klassendefinition „ShapeScaler“
Klassendefinition „Square“		
Klassendefinition „Circle“		
Klassendefinition „Triangle“		

Abbildung 2.8: Abhängigkeit der ausgeführten Operation zur Klassendefinition der Geometrie und des Besuchers.

len und implementieren somit das Verhalten, die jeweilige Geometrie besuchen zu können. Die Struktur des Entwurfsmusters lässt eine lose Kopplung zwischen den Geometrien und den Besuchern entstehen. Dadurch können neue Derivate von Shape hinzugefügt werden, ohne dass Anpassungen in der Besucherhierarchie notwendig sind (Martin u. Martin, 2008). Das Auslösen der Operation geschieht wie in Abschnitt 2.6.2 durch die Klasse `SomeApplication`. Sie ruft

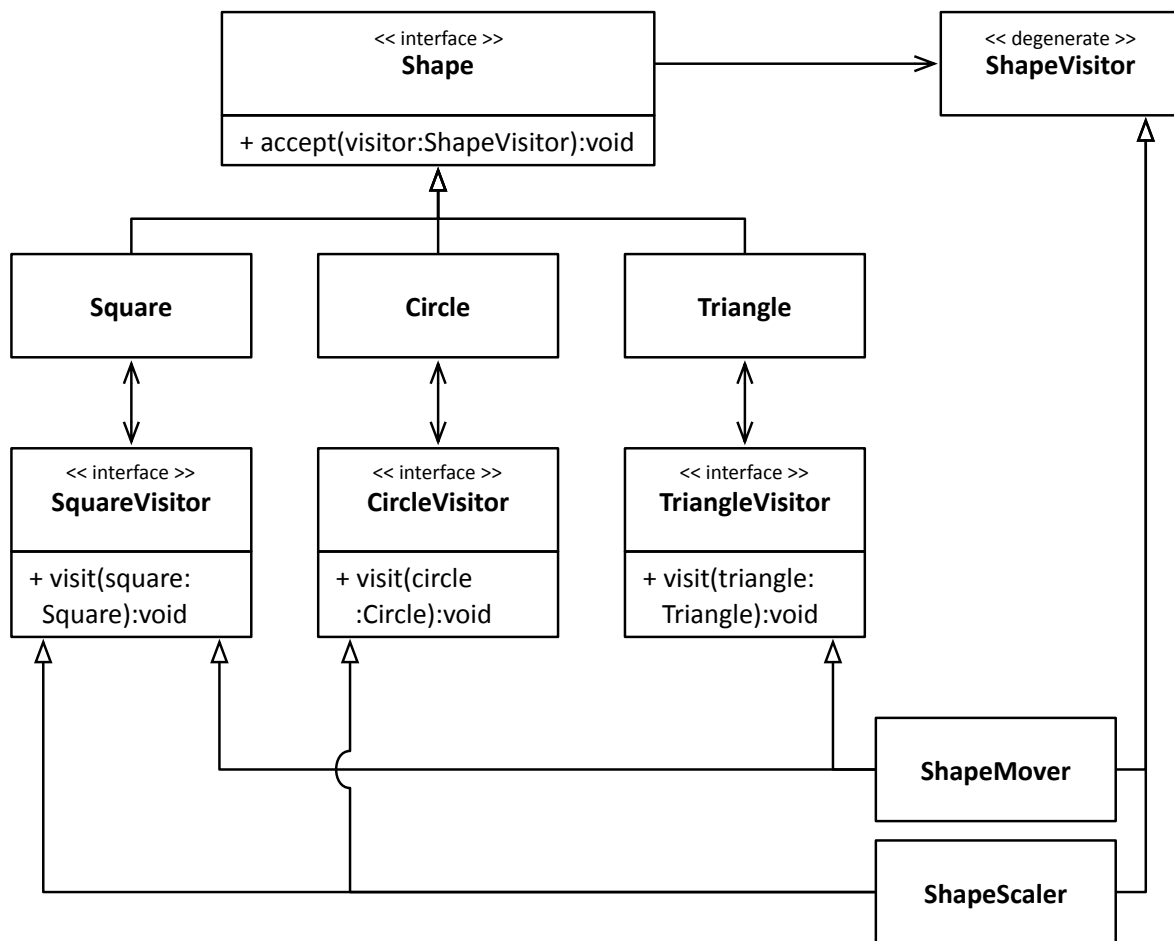


Abbildung 2.9: Acyclic Visitor Pattern in Anlehnung an Martin u. Martin (2008).

die „accept“-Methode der Geometrien auf und übergibt ein Objekt des Typs ShapeVisitor. Da diese in degenerierter Form vorliegt, kann die „visit“-Methode nicht ausgelöst werden. Zunächst muss dafür eine Typumwandlung zu der speziellen Besucherschnittstelle stattfinden. Im Gegensatz zu dem allgemeinen Visitor Pattern muss nicht jeder konkrete Besucher für jede Subklasse der zu besuchenden Hierarchie eine passende Funktionalität aufweisen. Durch die individuelle Ableitung von den typspezifischen Besucherschnittstellen können Operationen für bestimmte Derivate ausgelassen werden. In Abbildung 2.9 verfügt der ShapeMover nur über die Verschiebungsoperationen für Square und Triangle. Eine Typumwandlung zu SquareVisitor und TriangleVisitor wird erfolgreich durchgeführt. Demnach können Objekte beider Typen die „visit“-Methode aufrufen. Wird der ShapeMover an einen Circle übergeben, schlägt die Typumwandlung fehl. Dieser kann die „visit“-Methode nicht auslösen, es wird keine Handlung ausgeführt. Da nicht jede Kombination aus konkretem Visitor und zu

besuchendem Derivat die Ausführung einer Operation bewirkt, wird aus Abbildung 2.8 eine dünnbesetzte Verhaltensmatrix³ (Martin u. Martin, 2008).

2.6.4 Befehl-Muster – Command Pattern

Interaktionen des Benutzers mit der grafischen Oberfläche rufen Datenänderungen innerhalb des Programms hervor. Um nach eventuellen Fehlern des Benutzers den vorherigen Zustand wiederherzustellen, müssen die Aktionen gespeichert und rückwärts ausgeführt werden. Das Command Pattern ermöglicht das Anlegen einer Historie von Handlungen des Benutzers. Dafür werden Operationen in parametrisierter Form in Command-Objekten gekapselt und in der CommandQueue abgelegt. Die Commands können durch ihre öffentlichen Methoden „execute“, „undo“ und „redo“ ausgeführt, rückgängig gemacht und erneut ausgeführt werden (Gamma u. Riehle, 2004). Das Befehl-Muster kann analog zu Abbildung 2.10 umgesetzt werden.

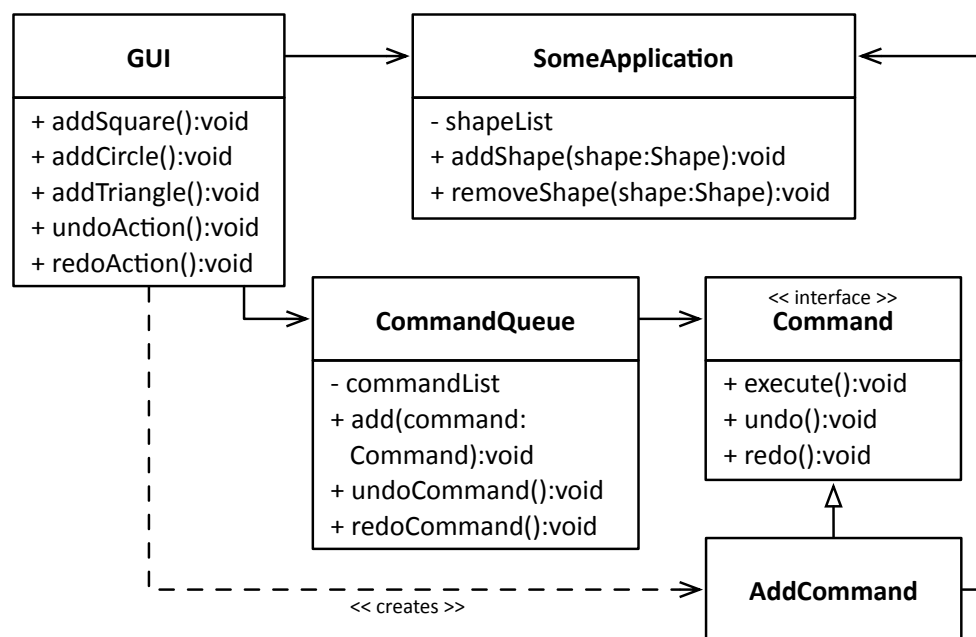


Abbildung 2.10: Umsetzung des Command Patterns.

Die Klasse **SomeApplication** enthält eine Liste aller Shapes, die durch den Benutzer während der Laufzeit erzeugt werden. Durch die Verwendung der Methoden „addShape“ und „removeShape“ können Geometrien hinzugefügt oder entfernt werden. Die Bearbeitung der Liste kann der Benutzer über die Methoden der GUI veranlassen. Durch Klicken auf bestimmte Befehls-

³engl.: sparse matrix

flächen in der grafischen Benutzeroberfläche werden die Methoden der GUI ausgelöst. Wird eine der „add“-Methoden aufgerufen, erzeugt die GUI ein passendes Geometrie-Objekt und kann es über die „addShape“-Methode zu der Liste in `SomeApplication` hinzufügen. Die rückwärtige Ausführung dieser Handlung würde den Aufruf der „removeShape“-Methode mit der zuvor hinzugefügten Shape bedeuten. Diese Funktionalität kann in einem Command gekapselt werden. Dem `AddCommand` wird im Konstruktor eine Instanz der Klasse `SomeApplication` sowie ein Geometrie-Objekt übergeben, die er als Instanzvariablen referenziert. Innerhalb der „execute“-Methode wird „addShape“ aufgerufen, sodass die Geometrie in die Liste eingefügt wird. Die „undo“-Methode bewirkt das Entfernen der Shape. Die GUI erstellt einen Command, dem sie die Parameter für das Ausführen der Handlung übergibt. Anschließend wird der Befehl in der `CommandQueue` gespeichert und ausgelöst. Die `CommandQueue` beinhaltet eine Liste, in der dynamisch Befehle hinzugefügt werden können. Dadurch wird eine Historie der Aktionen des Benutzers erstellt. Über die Methode „undoAction“ der GUI kann der Benutzer veranlassen, dass die `CommandQueue` den zuletzt ausgeführten Befehl rückgängig macht. Durch „redoAction“ der GUI führt die `CommandQueue` den zuletzt widerrufenen Command erneut aus. Durch Ableitung von der abstrakten Schnittstelle `Command` können neuartige Befehlsklassen zu der `CommandQueue` hinzugefügt werden, die einen anderen Algorithmus als der `AddCommand` kapseln. Zu jeder Handlung des Benutzers, die in der Befehlsliste gespeichert werden soll, können individuelle Subklassen erzeugt werden. Zur Bearbeitung einzelner Formen kann z.B. der `EditShapeCommand` eingeführt werden, der als Parameter die zu bearbeitende Shape und einen `ShapeVisitor` nach Abbildung 2.9 übergeben wird. Die „execute“-Methode des `EditShapeCommands` löst den *double dispatch*-Mechanismus des Visitor Patterns aus.

2.6.5 Beobachter-Muster – Observer Pattern

Die Intention des objektorientierten Entwurfs ist die Unterteilung eines komplexen Systems in mehrere zusammenwirkende Bestandteile. Auf niedrigster Abstraktionsebene werden diese durch Klassen repräsentiert. Damit ein Programm zu jedem Zeitpunkt seinen korrekten Zustand präsentieren kann, gilt als eine der prägnantesten Schwierigkeiten konsistente Zustände der verschiedenen Objekte herzustellen (Gamma u. Riehle, 2004). Abbildung 2.11 zeigt ein Objekt der Klasse `SomeApplication`. Es enthält eine Liste von Shapes, in der sich zwei verschiedene Instanzen der Klassen `Circle` und `Square` sowie eine Instanz der Klasse `Triangle` befinden.

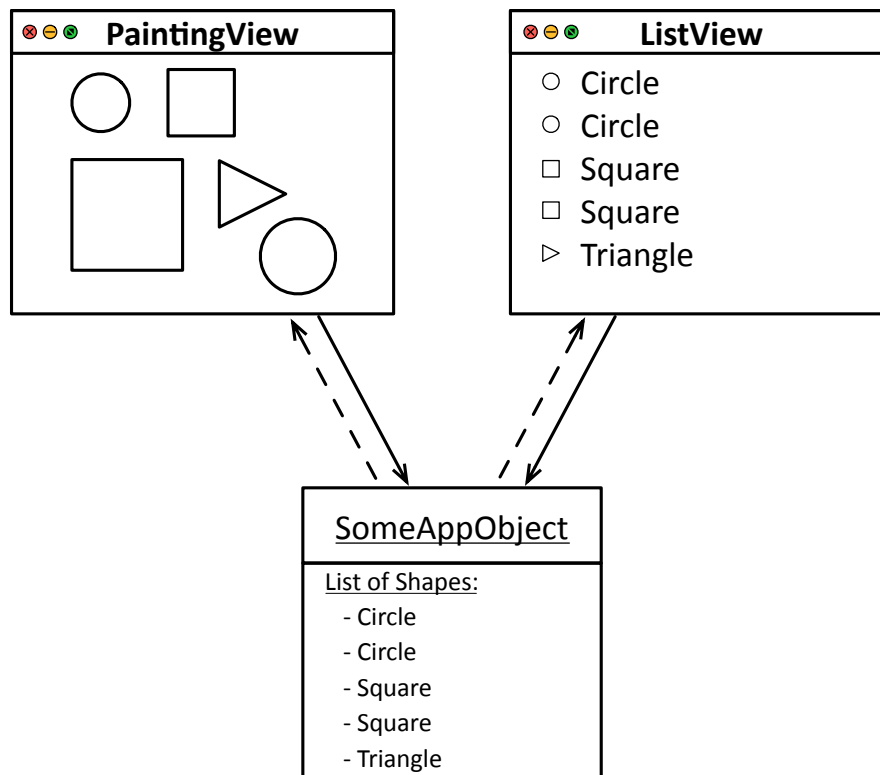


Abbildung 2.11: Individuelle Visualisierungen der Shapes durch unterschiedliche Views in Anlehnung an Gamma u. Riehle (2004).

Basierend auf dieser Liste können unterschiedliche Darstellungen erfolgen. Die **PaintingView** zeichnet alle Shapes entsprechend ihrer konkreten Typen sowie der Ausprägungen ihrer Attribute. Die **ListView** erstellt eine Auflistung. Bei Änderungen der Liste in **SomeAppObject** müssen sich beide Darstellungen aktualisieren. Diese Funktionalität kann durch die Implementierung des Systems analog zu Abbildung 2.12 erreicht werden.

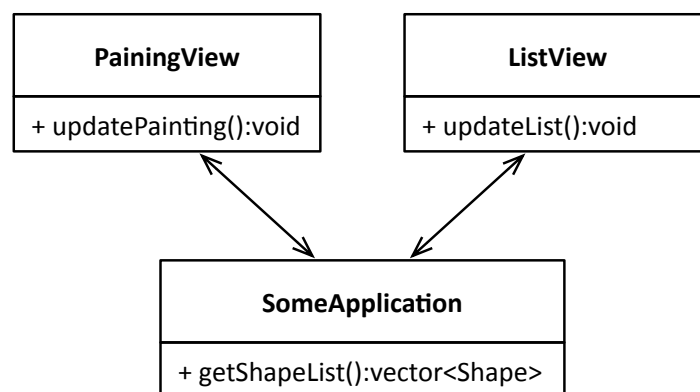


Abbildung 2.12: Mögliches System zur Darstellung der Liste in **SomeApplication**.

Durch die bidirektionalen Beziehungen zwischen `SomeApplication` und den verschiedenen Views besteht gegenseitiger Zugriff auf die öffentlichen Schnittstellen der Klassen. Ändert sich die Liste in `SomeApplication`, werden die spezifischen „update“-Methoden der Views aufgerufen. Die Viewklassen greifen anschließend über die „getShapes“-Methode auf die Liste in `SomeApplication` zu und lösen ihren individuellen Aktualisierungsmechanismus aus.

Durch die Beziehung der Klasse `SomeApplication` zu den konkreten Klassen `PaintingView` und `ListView` entsteht eine enge Kopplung. Dadurch verstößt das in Abbildung 2.12 dargestellte System gegen das OCP, weil die Erweiterung um eine neuartige Viewklasse die Modifikation von `SomeApplication` erfordert.

Das in Abbildung 2.13 gezeigte Observer Pattern beinhaltet eine lose Kopplung zwischen Domänenlogik und Benutzerschnittstelle. Dazu werden die abstrakten Schnittstellen `Subject` und `Observer` eingeführt. Durch die Ableitung der Klasse `SomeApplication` von `Subject`, erhält sie die Fähigkeit Objekte vom Typ `Observer` zu referenzieren und diese bei Bedarf zu benachrichtigen. Die Viewklassen erben von der Schnittstelle `Observer`, wodurch sie sich bei `SomeApplication` registrieren können. Ändert sich die Liste in `SomeApplication`, wird die vererbte „notify“-Methode aufgerufen, innerhalb der durch die Liste von `Observers` iteriert und für jedes `Observer`-Objekt die „update“-Methode aufgerufen wird.

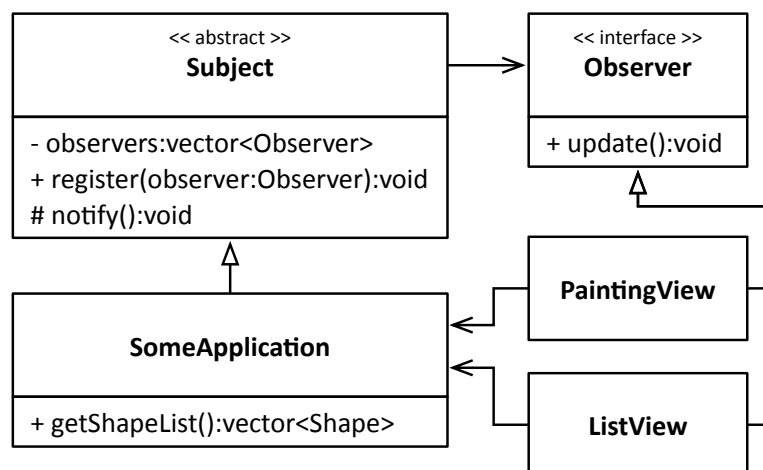


Abbildung 2.13: Observer Pattern in Anlehnung an Martin u. Martin (2008).

2.7 Architekturmuster

Die in Abschnitt 2.6 erläuterten Entwurfsmuster beinhalten vorgefertigte Lösungen für strukturell wiederkehrende Probleme auf Ebene der Klassen innerhalb von Modulen. Betrachtet man die Applikation auf höherem Abstraktionsgrad können Architekturmuster angewendet werden. Vergleichbar mit den Entwurfsmustern werden durch architektonische Muster Vorschläge für die Strukturierung der Hauptbestandteile einer Applikation unterbreitet. Dabei handelt es sich um Softwarearchitekturen, die sich in bestimmten Anwendungsfällen bewährt und verbreitet haben. Eines der bekanntesten Architekturmuster ist die Model-View-Presenter (MVP) Architektur.

2.7.1 Model-View-Presenter (MVP) Muster – Model-View-Presenter Pattern

Zur einfachen Bedienung von Applikationen werden zumeist grafische Benutzeroberflächen eingesetzt, welche die innerhalb des Programms verwendeten Daten präsentieren. Bereiche der Software, deren Aufgabenfeld die Darstellung auf dem Bildschirm beinhaltet, sind Bestandteil der View. Durch eine große Anzahl an verschiedenen Präsentationsmöglichkeiten sowie durch sich kontinuierlich ändernde Bedürfnisse der Benutzer handelt es sich hierbei um eine sehr volatile Komponente der Software. Im Allgemeinen unterliegt die View einem größeren Änderungsbedarf als Module, die die Logik der Problemdomäne enthalten (Martin u. Martin, 2008). Diese Module werden im Zusammenhang des MVP dem Model zugeordnet. Um Auswirkungen auf das Model zu vermeiden, die aus der Modifikation der View resultieren, ist es von Bedeutung den Quelltext der unterschiedlichen Bereiche zu trennen. Nach Fowler (2010) wird dieses Vorgehen als *Seperated Presentation* bezeichnet. Durch die Barriere, die zwischen den Aufgabenbereichen erzeugt wird, entstehen die Module Model und View. Die Interaktion der Module kann auf zwei Arten erfolgen. Zum einen können Eingaben des Benutzers auf der grafischen Oberfläche Veränderungen der in den Modelklassen befindlichen Daten bewirken. Zum anderen kann die Veränderung des Models eine Aktualisierung der Darstellung auslösen. In der klassischen MVP-Architektur übernehmen die Presenterklassen die Modifikation der Modelklassen. Die Synchronisierung der Darstellung geschieht durch Observer-Mechanismen. Die Viewklassen sind als Beobachter bei den Modelklassen registriert und aktualisieren sich, sobald Änderungen im Model bemerkbar werden. Nach Potel (1996) ergibt sich durch die Umsetzung des MVP-Muster eine Applikationsstruktur nach Abbildung 2.14.

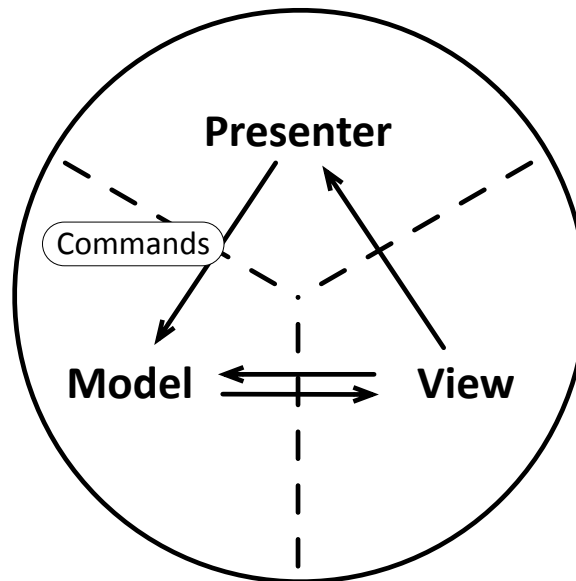


Abbildung 2.14: Model-View-Presenter-Architektur in Anlehnung an Potel (1996).

Bei den Viewklassen handelt es sich um Applikationsfenster, in denen sich mehrere Steuerelemente befinden. Zu jedem Fenster existiert eine zugehörige Presenterklasse. Die Eingaben des Benutzers lösen Ereignisse der Steuerelemente aus, die von der View aufgefangen werden. Anschließend werden diese an den Presenter weitergereicht, der über die Weiterverarbeitung der erhaltenen Informationen entscheidet. Beinhaltet das ausgelöste Ereignis die Änderung des Models, kann der Presenter diese veranlassen. Um die Modelklassen entsprechend der Benutzereingaben zu modifizieren, kann z.B. das Command Pattern verwendet werden (siehe Abschnitt 2.6.4). Die Änderung des Models induziert den Observer-Mechanismus, wodurch sie in der View bemerkt wird. Die Darstellung kann sich basierend auf dem neuen Zustand aktualisieren.

2.7.2 Passive-View-Muster – Passive View Pattern

Das Passive-View-Muster ist eine Variante der MVP-Architektur, bei der das Verhalten der Viewklassen auf ein Minimum reduziert wird. Dabei wird dem Presenter nicht nur wie üblich die Aufgabe zugeteilt, die vom Benutzer ausgelösten Ereignisse zu verarbeiten. Im Gegensatz zu der allgemeinen Form des MVP-Musters ist der Presenter zusätzlich für die Aktualisierung der View verantwortlich, die aus Änderungen im Model resultieren. In den Viewklassen verbleibt somit nur der Präsentationscode. In der testgetriebenen Entwicklung wird dieses Muster

angewendet, um möglichst viel Präsentationslogik im Kontext des Presenters testen zu können. Das Testen der Viewklassen selbst kann durch ihre geringe Komplexität vernachlässigt werden. Im Umfang dieser Arbeit wird nicht weiter auf die testgetriebene Entwicklung eingegangen. Hier wird dieses Muster dafür verwendet die Klassen der View vollständig von dem Model zu entkoppeln (Fowler, 2010). Es ergibt sich eine Klassenstruktur nach Abbildung 2.15. Die abstrakte Schnittstelle View ermöglicht einen flexiblen Austausch der konkreten Subklassen. Hierbei kann ebenfalls ein anderes Visualisierungsframework gewählt werden. Durch das passive Verhalten der Viewklassen ist bei der Implementierung neuer Derivate lediglich der individuelle Präsentationscode umzusetzen.

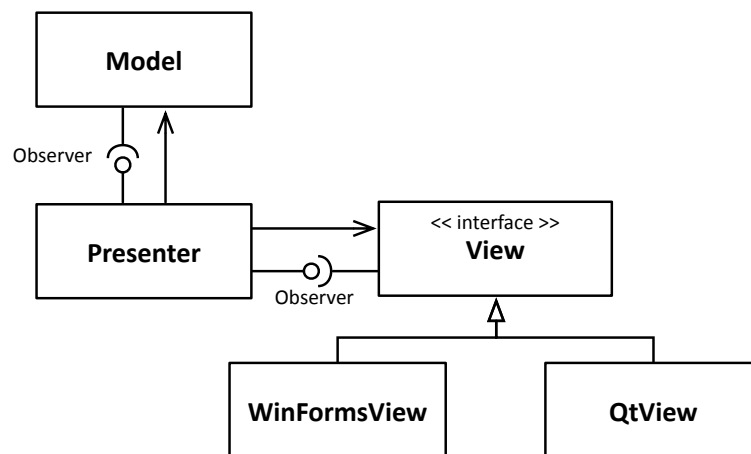


Abbildung 2.15: Umsetzung des Passive-View-Musters in Anlehnung an Fowler (2010).

2.8 Zusammenfassung

Gegenstand dieses Kapitels sind die Grundlagen des Softwaredesigns. Zunächst wird dabei auf die Komplexität eingegangen, die der Entwurf von Software mit sich bringt. Die inhärente Komplexität setzt sich nach Booch (2007) aus den Faktoren *Komplexität der Problemdomäne*, *die Schwierigkeit den Entwicklungsprozess handzuhaben*, *die mögliche Flexibilität von Software* und *die Charakterisierung des Verhaltens diskreter Systeme* zusammen. Anschließend werden Basiskonzepte zur Strukturierung komplexer Systeme erläutert. Dabei handelt es sich um die generellen Konzepte Dekomposition, Abstraktion und Hierarchie, die nicht nur in der Softwareentwicklung Anwendung finden. Nachfolgend werden die Schwierigkeiten der OOP aufgezeigt, die sich in den Beziehungen zwischen Objekten, Modulen oder Komponenten widerspiegeln. Daraus lässt sich als Ziel ableiten, Systeme mit hoher Kohäsion und geringer Kopplung umzusetzen. Für die Implementierung von Systemen entsprechend dieser Metriken stehen dem Entwickler die Konzepte des OOD zur Verfügung. Darunter befinden sich die Abstraktion im objektorientierten Sinn, die Datenkapselung, die Modularisierung sowie die Hierarchie. Basierend auf den genannten Konzepten entstanden verschiedene Entwurfsprinzipien zur Strukturierung von objektorientierten Systemen. Im diesen Zusammenhang werden die *SOLID-Principles* erläutert. Die kontinuierliche Anwendung der Entwurfsprinzipien brachte bei häufig auftretenden und strukturell ähnlichen Entwurfsproblemen wiederkehrende Muster hervor. Im Bezug auf die Abstraktionsebene der Software können unterschiedliche Arten von Mustern identifiziert werden. Auf niedriger Ebene befinden sich die Entwurfsmuster, die Klassenstrukturen beschreiben. Die auf hoher Abstraktionsebene angesiedelten Architekturmuster beschreiben das Arrangement der Hauptbestandteile der Software. Dazu gehören übergeordnete Module und Komponenten. Durch die Anwendung bzw. die Kombination der verschiedenen Muster kann Software entwickelt werden, die den Prinzipien des OOD entspricht und somit die Ziele der OOP erfüllt. Zu den erläuterten Entwurfsmustern gehören das Fabrik-Muster, das Besucher-Muster, das azyklische Besucher-Muster, das Befehl-Muster sowie das Besucher-Muster. Die beschriebenen Architekturmuster sind das MVP-Muster und das Passive-View-Muster.

KAPITEL 3

Anforderungen an die Applikation CECLOUD

Software unterliegt hinsichtlich ihres Anwendungszwecks bestimmten Anforderungen, denen sie gerecht werden muss, um über eine hohe Qualität zu verfügen. Neben den Aspekten der Funktionalität gelten die internen und externen Qualitätsfaktoren nach McCall (1977) als ein wesentliches Merkmal guter Software. Aufgrund der gegenseitigen Beeinflussung der Faktoren untereinander, ist die vollständige Umsetzung der einzelnen Merkmale nur bedingt möglich. Daher ist eine Bewertung der einzelnen Faktoren in Bezug auf ihre Bedeutung für die umzusetzende Software vorzunehmen.

In diesem Kapitel werden die funktionalen Anforderungen an die Applikation CECLOUD thematisiert. Darunter befinden sich die Organisation von Berechnungstheorien, die Erweiterung der Applikation durch neue Theorien, eine intuitive Benutzeroberfläche sowie eine plattformunabhängige Benutzung. Anschließend werden die Faktoren für die Bewertung der Softwarequalität erläutert und für die Umsetzung dieser Software gewichtet.

3.1 Anforderungen aus der Problemdomäne

Mit CECLOUD soll eine Anwendung entstehen, die zur automatisierten Berechnung und Visualisierung von mathematischen und physikalischen Gleichungen im Fachbereich Küsteningenieurwesen (KI) verwendet werden kann. Als Beispiel für die Berechnungstheorien werden hier die Lineare Wellentheorie (LWT) nach Airy und Laplace und die Berechnung der Wellenkraft auf Pfahlbauwerke nach Morison verwendet.

Lineare Wellentheorie. Die LWT wird im KI dazu verwendet deterministische Aussagen über die Ausbreitung von Wasserwellen und deren Dynamik zu treffen. In die Berechnung gehen dafür die Parameter des vorherrschenden Wellenklimas und Informationen über den betrachteten Zustand ein. Das Wellenklima wird charakterisiert durch die Parameter Wellenlänge L , Wellenperiode T , Wellenhöhe H und Wassertiefe h . Der betrachtete Zustand (i. F. Koordinatensystem genannt) drückt die räumliche und zeitliche Distanz x und t zum Ausgangszustand sowie die betrachtete Wassertiefe z aus.

Druckkraftberechnung nach Morison. Trifft eine Welle auf ein Pfahlbauwerk resultieren daraus Druckkräfte auf den Pfahl, die über die Wassertiefe veränderlich sind. Die Berechnung der Druckverteilung hängt von der Dynamik der Welle ab, welche durch die LWT beschrieben wird. Zur Ermittlung der Druckverteilung müssen deswegen die Ergebnisse der LWT bekannt sein. Des Weiteren hängt die Berechnung nach Morison von dem Pfahldurchmesser sowie von der Viskosität und Dichte des Wassers ab.

3.1.1 Organisation der Berechnungstheorien

Bei der Auswertung der Berechnungsvorschriften ist zu beachten, dass Theorien voneinander abhängen können, da einzelne Berechnungseinheiten die Ergebnisse anderer benötigen. Dadurch entstehen hierarchische Beziehungen. Abbildung 3.1 zeigt die sich bildende Hierarchie, die sich aus den zuvor beschriebenen mathematischen und physikalischen Gleichungen ergibt. Sie verdeutlicht den Datenfluss während der Auswertung innerhalb der Hierarchie. Zur Berechnung der Morisonkraft müssen nicht nur die Parameter bekannt sein durch die das Wasser und der Pfahl repräsentiert werden, sondern auch die Ergebnisse der LWT. Diese ergeben sich aus den Parametern der Elemente Wellenklima und Koordinatensystem. Das hat zur Folge, dass die Auswertung von der niedrigsten Hierarchiestufe an aufsteigend stattfinden muss. Als erstes wird auf Grundlage der Parameter des Wellenklimas und des Koordinatensystems die

LWT ausgewertet. Anschließend gehen diese Ergebnisse zusammen mit den Parametern des Pfahls und des Wassers in die Berechnung nach Morison ein. CECLOUD soll dem Benutzer die Möglichkeit geben Berechnungstheorien in einer solchen hierarchischen Struktur zu organisieren und auszuwerten. Dafür sollen sowohl die Hierarchie der Elemente, als auch die Ergebnisse der Auswertung visualisiert werden.

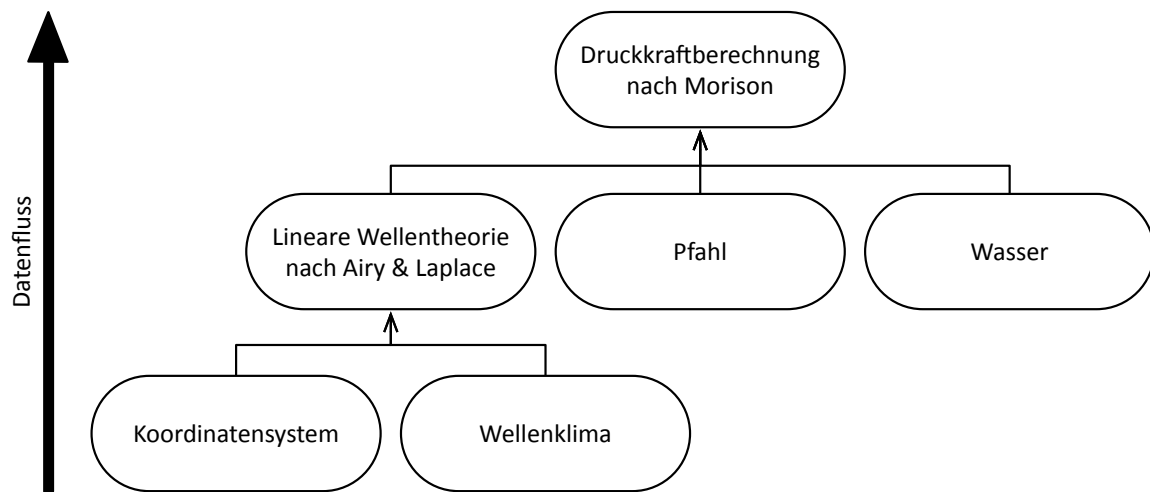


Abbildung 3.1: Hierarchie der Berechnungseinheiten zur Berechnung der Druckkräfte nach Morison.

3.1.2 Erweiterung der Applikation durch neue Theorien

Neben der LWT und der Druckkraftberechnung nach Morison beinhaltet der Fachbereich KI viele weitere Berechnungstheorien, die schrittweise in die Applikation aufgenommen werden sollen. Aus diesem Grund ist CECLOUD so umzusetzen, dass eine Erweiterung durch neue Elemente unterstützt wird, ohne dabei die Basisstruktur der Software ändern zu müssen. Die Erweiterung der Applikation wird im Rahmen dieser Arbeit durch die Integration der Berechnung des Sedimenttransports gezeigt.

Sedimenttransport. Durch die Bewegungen der Welle entstehen Strömungen, die das Sediment der Gewässersohle verlagern. Berechnungen in diesem Kontext gehören zu der Berechnungstheorie des Sedimenttransports. Dieser ist abhängig von den Ergebnissen der LWT und erweitert sie um die Parameter des Wassers sowie des betroffenen Sediments. Das Wasser wird dabei durch seine Viskosität und seine Dichte charakterisiert, das Sediment durch seine Dichte.

Abbildung 3.2 zeigt die Berechnungshierarchie zur Auswertung des Sedimenttransports. Der Datenfluss während der Berechnung erfolgt analog zu Kapitel 3.1.1.

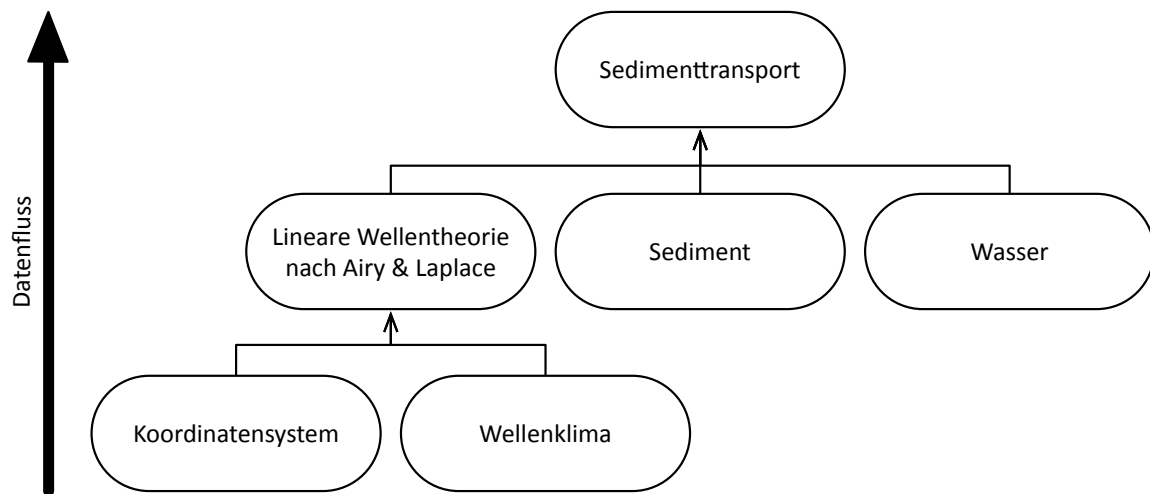


Abbildung 3.2: Erweiterung der Applikation um eine neue Berechnungshierarchie.

3.1.3 Benutzerfreundlichkeit – Intuitive Bedienung

Die Schwierigkeit bei dem Entwurf von Software ist die Erarbeitung eines Konzepts, das sich in den gewöhnlichen Arbeitsablauf des Benutzers nahtlos einfügt. Durch die Anwendung der Applikation wird die Bearbeitungsgeschwindigkeit gefördert. Der Schlüssel dazu ist ein Bedienkonzept, das intuitiv verstanden und leicht erlernt werden kann.

CECLOUD soll über ein solches Bedienkonzept verfügen, dass dem Benutzer einen „einfachen“ Umgang ermöglicht, sodass eine Fehlbedienung vermieden wird. Gleichzeitig wird dadurch die positive Erfahrung erhöht, wodurch die Zufriedenheit des Anwenders steigt.

3.1.4 Plattformunabhängige Benutzung von CECLOUD

Durch Verwendung bestimmter Techniken bei der Umsetzung der Software können Restriktionen hinsichtlich der Benutzung von Software auf verschiedenen Endgeräten mit unterschiedlichen Betriebssystemen entstehen. Um die Zielgruppe möglicher Benutzer von CECLOUD nicht zu beschränken muss gewährleistet sein, dass die Applikation unabhängig von der Umgebung funktionsfähig ist. Die Wahl des Betriebssystems darf sich dabei nicht negativ auf den Gebrauch der Applikation auswirken. Die Software ist so zu entwerfen, dass sie die Hürde unterschiedlicher Plattformen überwinden kann und davon unabhängig funktionsfähig bleibt.

3.2 Softwarequalität

Ziel dieser Arbeit ist es, mit der Applikation CECLOUD hochwertige Software zu entwickeln. Um den Zustand der Software zu bewerten, bedarf es bestimmter Metriken die Aussagen über die Qualität treffen können. Zu diesem Zweck werden die externen und internen Qualitätsfaktoren nach McCall (1977) verwendet. Dabei handelt es sich um Schlüsseigenschaften, die Software aufweisen muss, um einen hohen Qualitätsstandard zu erreichen.

Externe Qualitätsfaktoren

Durch die Unterscheidung in externe und interne Merkmale lässt sich die Perspektive auf die Software erfassen. Die externen Qualitätsfaktoren berücksichtigen dabei die Sicht des Anwenders. Um die Qualität für den Benutzer zu gewährleisten muss die Software nach McConnell (2009) folgende Kriterien erfüllen:

- **Korrektheit.** Der Grad, zu dem ein System frei von Fehlern in Bezug auf Spezifikation, Entwurf und Implementierung ist.
- **Benutzerfreundlichkeit.** Das Maß an Leichtigkeit, mit dem ein Benutzer die Bedienung der Software erlernen und nutzen kann.
- **Effizienz.** Beschreibt den Ausnutzungsgrad des Systems. Es sollte minimale Anforderungen an Ressourcen wie z.B. Speicher und Ausführungszeit aufweisen.
- **Zuverlässigkeit.** Beschreibt die Fähigkeit eines Systems unter definierten Bedingungen mit nur geringer Ausfallrate funktionsfähig zu sein.
- **Integrität.** Das Maß, mit dem ein System vor unautorisierten Zugriffen auf Daten geschützt ist. Darunter befinden sich sowohl beschränkte Zugriffsrechte für Benutzer als auch die korrekte Datenverarbeitung innerhalb des Programms.
- **Anpassungsfähigkeit.** Beschreibt das Maß an Flexibilität eines Systems, mit der es ohne Veränderungen an andere Umgebungsbedingungen und Applikationen angepasst werden kann.
- **Genauigkeit.** Der Grad, zu dem ein System frei von Fehlern im Hinblick auf quantitative Aussagen ist.
- **Robustheit.** Beschreibt die Fähigkeit eines Systems ohne Versagen Fehlbedienungen des Benutzers und schwierige Umgebungsbedingungen zu tolerieren.

Interne Qualitätsfaktoren

Die Sicht des Entwicklers auf die Software wird durch die internen Qualitätsfaktoren erfasst. Nach McConnell (2009) muss die Software für eine hohe interne Qualität folgende Eigenschaften aufweisen:

- **Wartungsfreundlichkeit.** Die Leichtigkeit, mit der Software modifiziert werden kann. Durch Veränderungen können Fehler korrigiert, die Performance gesteigert oder neue Funktionalität hinzugefügt werden.
- **Flexibilität.** Die Möglichkeit ein System an Umgebungsbedingungen anzupassen, für die es nicht entworfen wurde.
- **Portierbarkeit.** Die Leichtigkeit, mit der ein System an Umgebungsbedingungen angepasst werden kann, für die es nicht entworfen wurde.
- **Wiederverwendbarkeit.** Die Möglichkeit Bestandteile der Software in anderen Systemen wiederzuverwenden.
- **Lesbarkeit.** Die Leichtigkeit, mit welcher der Quelltext gelesen und verstanden werden kann, besonders auf Detailebene.
- **Testbarkeit.** Beschreibt die Möglichkeit ein System im Bezug auf seine Anforderungen in Einzelteilen oder im Gesamten testen zu können.
- **Verständlichkeit.** Die Leichtigkeit, mit der ein System sowohl auf hoher Abstraktionsebene als auch auf Detailebene verstanden werden kann.

Wechselwirkung der Qualitätsfaktoren

Sind alle Qualitätsmerkmale vollständig umgesetzt ergibt sich eine Software von höchster Qualität. Da sich die Kriterien untereinander beeinflussen, ist dies jedoch nicht möglich. Abbildung 3.3 zeigt die Wechselwirkungen zwischen den externen Qualitätsfaktoren. Die Förderung eines Faktors kann dabei die Umsetzung eines anderen reduzieren. Maximiert man zum Beispiel die Genauigkeit der Software kann es zu Effizienzeinbußen kommen. Auch positive Beeinflussung der Faktoren ist möglich, wie der Zusammenhang zwischen der Zuverlässigkeit und der Korrektheit zeigt. Hier bedeutet das Ausprägen des einen gleichzeitig das Fördern des anderen Merkmals. Eine der Hauptaufgaben des Entwicklers ist es in Abhängigkeit der Anforderungen an die Software die Relevanz der einzelnen Qualitätsfaktoren zu bewerten (McCall, 1977). Nach

McConnell (2009) zeichnet das Erarbeiten einer optimalen Lösung der Softwareentwicklung als Ingenieurdisziplin aus.

Auswirkung des unteren Faktors (Zeilen) auf den rechten Faktor (Spalten)	Korrektheit	Benutzerfreundlichkeit	Effizienz	Zuverlässigkeit	Integrität	Kompatibilität	Genauigkeit	Robustheit
Korrektheit	↑		↑	↑			↑	↓
Benutzerfreundlichkeit		↑				↑	↑	
Effizienz	↓		↑	↓	↓	↓	↓	
Zuverlässigkeit	↑			↑	↑		↑	↓
Integrität			↓	↑	↑			
Kompatibilität					↓	↑		↑
Genauigkeit	↑		↓	↑		↓	↑	↓
Robustheit	↓	↑	↓	↓	↓	↑	↓	↑

↑ fördert

↓ reduziert

Abbildung 3.3: Wechselwirkungen zwischen externen Qualitätsfaktoren nach McConnell (2009).

3.3 Bewerten der Qualitätsfaktoren für die Umsetzung von CECLOUD

Software wird allein zu dem Zweck entwickelt, den Benutzer zufrieden zu stellen. Aus diesem Grund wird unabhängig von der Problemdomäne die Realisierung hoher externer Qualität priorisiert. Die Umsetzung der internen Qualitätsfaktoren sind nach Meyer (2009) der Schlüssel zur Realisierung der externen Faktoren. Innerhalb der externen Qualität können die einzelnen Faktoren hinsichtlich ihrer allgemeingültigen Wichtigkeit eingestuft werden. Nach Budgen

(2003) gilt »*fitness for purpose*«¹ als die wichtigste Voraussetzung für hochwertige Software. Gleich wie elegant oder effizient ein System funktioniert, es wird keine hohe Qualität aufweisen ohne die ursprüngliche Aufgabe zu erfüllen, für die es entworfen wurde. Aus diesem Grund sind die externen Qualitätsfaktoren, die sich mit dieser Thematik befassen, von höchster Bedeutung. Nach Meyer (2009) sind deswegen die wichtigsten Faktoren die Korrektheit und die Genauigkeit. Die Merkmale Zuverlässigkeit und Robustheit können aufgrund ihrer thematischen Ähnlichkeit dicht hinter der Korrektheit angesiedelt werden. Die Forderung nach einer einfachen Bedienung der Anwendung durch den Benutzer verleiht der Benutzerfreundlichkeit eine wichtige Rolle. Weiterhin ergibt sich aus der plattformunabhängigen Benutzung das Bedürfnis einer hohen Kompatibilität. Die Effizienz ist in diesem System weniger relevant, da die Berechnungen im Bereich des KI im Vergleich zu komplexen numerischen Simulationen einen geringen Aufwand benötigen. Auch bei weniger effizienten Strukturen, kann eine hinreichend schnelle Auswertung der mathematischen Gleichungen durch den Computer erfolgen. Die Integrität ist von untergeordneter Bedeutung, da innerhalb der Applikation keine Daten erhoben bzw. bearbeitet werden, die eine Sicherheitsbarriere erfordern.

Die äußere Anwendungsqualität wird maßgeblich von der internen Umsetzung der Software beeinflusst. Infolgedessen muss der Entwickler Techniken anwenden, welche die interne Qualität garantieren (Meyer, 2009). Aus der Anforderung die Applikation erweiterbar zu gestalten, ergibt sich als wichtigster interner Faktor die Flexibilität. Auch die Wartungsfreundlichkeit ist für die Implementierung hochwertiger Software essenziell. Sie stellt sicher, dass Änderungen und Anpassungen mit geringem Risiko des Versagens der Applikation eingearbeitet werden können. Die Lesbarkeit, die Verständlichkeit und die Wiederverwendbarkeit unterstützen die Wartungsfreundlichkeit und sind somit von gleicher Bedeutung. Die Kompatibilität der externen Qualität kann durch das innere Merkmal der Portierbarkeit umgesetzt werden.

¹deutsch: die Eignung der Software den vorgesehenen Zweck zu erfüllen

3.4 Zusammenfassung

In diesem Kapitel werden die allgemeinen Anforderungen an das System CECLOUD beschrieben. Des Weiteren ist ein Überblick über Metriken gegeben, durch welche die Qualität der Software bewertet werden kann. Diese werden letztlich anhand der Anforderungen hinsichtlich ihrer Relevanz für die Umsetzung der Applikation gewichtet.

Der Zweck der Software besteht darin mathematische und physikalische Gleichungen des Bereichs KI berechnen zu können. Die unterschiedlichen Berechnungstheorien können dabei voneinander abhängen. Dadurch entsteht eine Berechnungshierarchie. Die Applikation CECLOUD soll es dem Benutzer ermöglichen diese Hierarchie zu erstellen, zu bearbeiten und auszuwerten. Gleichzeitig soll die Applikation durch neue Berechnungstheorien risikofrei erweitert werden können. Weitere Anforderungen sind eine hohe Benutzerfreundlichkeit und eine plattformunabhängige Benutzung.

Die Qualität der Software hängt maßgeblich von der Umsetzung der internen und externen Qualitätsfaktoren ab. Da sich die einzelnen Qualitätsfaktoren untereinander beeinflussen, muss eine möglichst optimale Balance erarbeitet werden. Als die bedeutsamsten externen Merkmale für die Umsetzung von CECLOUD stellen sich die Korrektheit, die Zuverlässigkeit und die Robustheit heraus. Danach gelten als weitere entscheidende Faktoren die Benutzerfreundlichkeit und die Kompatibilität. Die Effizienz, die Genauigkeit sowie die Integrität sind in diesem System von geringer Relevanz.

Nach Meyer (2009) lässt sich eine hohe externe Qualität durch den Entwurf der Software nach den internen Qualitätsfaktoren erreichen. Bei den wichtigsten internen Faktoren handelt es sich um die Flexibilität sowie die Wartungsfreundlichkeit. Die Umsetzung der Wartungsfreundlichkeit wird von den Faktoren Lesbarkeit, Verständlichkeit und Wiederverwendbarkeit unterstützt. Die externe Kompatibilität kann durch die interne Portabilität erreicht werden.

KAPITEL 4

Konzept, Architektur und Technologie

Bei dem Entwurf von Software handelt es sich um eine komplexe Aufgabe. Die grobe Strukturierung der zu entwerfenden Applikation unterstützt den Entwickler dabei den Überblick zu behalten und sich nicht in Details zu verlieren.

Als Leitfaden für das Design der Applikation wird in diesem Kapitel das Anwendungskonzept von CECloud beschrieben. Dazu gehören die Bereiche der grafischen Oberfläche sowie die grundlegenden Funktionalitäten. Weiterhin wird die Basisarchitektur von CECloud gewählt und beschrieben. Sie unterteilt die Software aus Entwicklersicht in ihre Hauptbestandteile. Dadurch entsteht ein Gerüst für die Software, an dem sich der Designprozess orientiert. Ein weiterer Bestandteil dieses Kapitels ist die Technologie, die zur Umsetzung der Applikation verwendet wurde.

4.1 Das Konzept der Applikation CECLOUD

Wie in Abschnitt 3.1.1 beschrieben, soll CECLOUD dem Benutzer die Möglichkeit bieten, Berechnungshierarchien zu erstellen, zu verwalten und auszuwerten. Abbildung 4.1 zeigt die hauptsächlichen Visualisierungssegmente, die in der grafischen Benutzeroberfläche enthalten sind. In dem Bereich des Solution Explorers befindet sich eine Baumstruktur, die zur Darstellung der Berechnungshierarchien verwendet wird. Sie zeigt zu jedem Knoten den vom Benutzer gewählten Namen sowie den Typen an. Innerhalb des Properties Explorers wird eine Tabelle verwendet, die knotenspezifische Eigenschaften darstellt. Der Inhalt der Eigenschaftstabelle hängt dabei von dem Knoten ab, der in der Baumstruktur ausgewählt ist. Nach dem erfolgreichen Erstellen einer Berechnungshierarchie kann diese ausgewertet und ihre Ergebnisse visualisiert werden. Dafür steht in der Applikation der linke Bereich zur Verfügung.

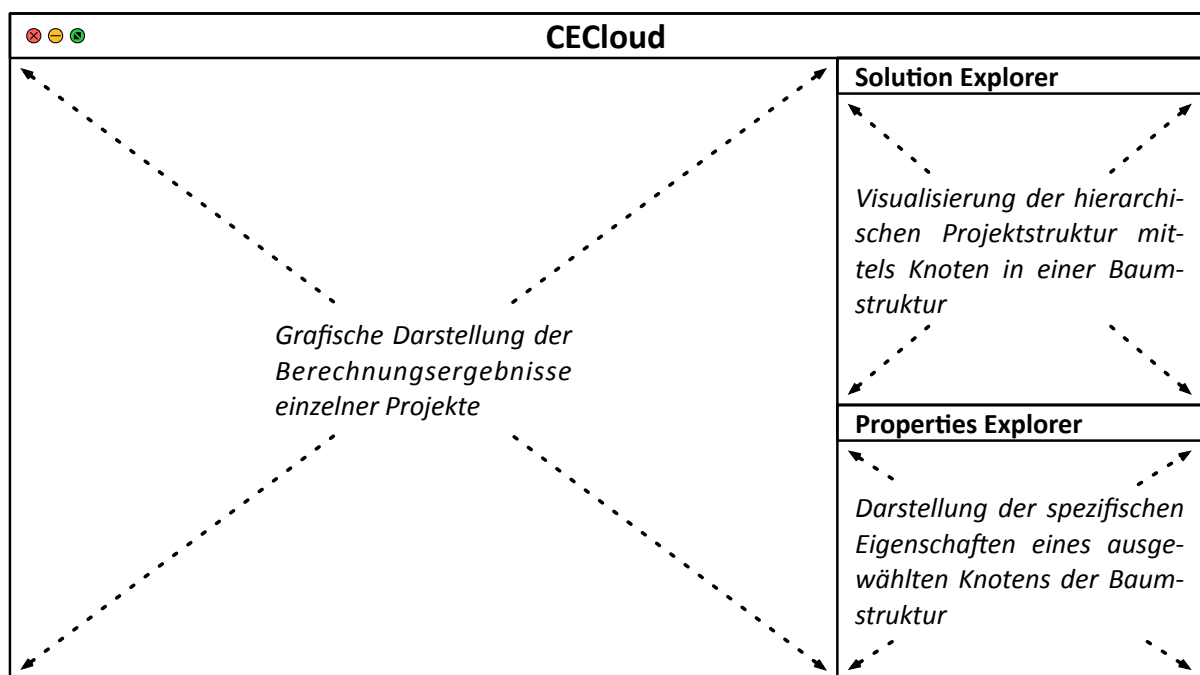


Abbildung 4.1: Konzeptuelle Ansicht der grafischen Benutzeroberfläche.

Die Hauptfunktionalität der Applikation spiegelt sich in den Punkten *Erstellen von Berechnungshierarchien*, *Veränderung der Berechnungsparameter* sowie in der *Visualisierung der Berechnungsergebnisse* wider.

Erstellen von Berechnungshierarchien

Über Menübefehlsflächen kann der Nutzer Knoten zu der Baumstruktur hinzufügen. Soll eine Berechnungshierarchie nach Abbildung 3.1 modelliert werden, müssen zunächst alle benötigten Knoten erstellt werden. Da diese noch in keiner Beziehung zueinander stehen, befinden sie sich alle auf oberster Ebene der Baumstruktur, es entsteht eine Liste. Abbildung 4.3 zeigt eine mögliche Liste, die bei der Modellierung einer Hierarchie zur Berechnung der Druckkräfte nach Morison im Solution Explorer entstehen kann. Die „Druckkraftberechnung nach Morison“ wird in diesem Beispiel durch den Typ „Morison“ repräsentiert, die „Lineare Wellentheorie nach Airy & Laplace“ durch „Lineare Wellentheorie“.

Solution Explorer	
Name	Typ
Morison_1234	Morison
Wellentheorie_xy	Lineare Wellentheorie
meinPfahl	Pfahl
Nordseewasser	Wasser
KSystem_Spundwand	Koordinatensystem
TideWelle_Jan-1999	Wellenklima

Abbildung 4.2: Konzeptuelle Ansicht nach dem Hinzufügen von Knoten zur Baumstruktur.

Um die vorgesehene hierarchische Struktur zu erreichen, können Knoten als Kinder an andere gebunden werden. Abbildung 4.3 stellt die Baumstruktur nach dem Vorgang des Bindens dar. Das *Morison_1234* benötigt zur Auswertung Knoten vom Typ Lineare Wellentheorie, Pfahl und Wasser. *Wellentheorie_xy*, *meinPfahl* und *Nordseewasser* werden deswegen als Kinder bei dem *Morison_1234* registriert. Damit die *Wellentheorie_xy* ihre Ereignisse korrekt ermitteln und diese an die Druckkraftberechnung nach Morison weiterreichen kann, müssen auch ihr Kindknoten zugeordnet werden. Dafür kann der Benutzer das *KSystem_Spundwand* und die *TideWelle_Jan-1999* an die *Wellentheorie_xy* binden.

Zu beachten ist, dass Knoten nach dem Binden als Kind nicht aus der obersten Ebene der Baumstruktur entfernt werden, wodurch ihre Wiederverwendbarkeit erreicht wird. Kinder sind dadurch unabhängig von der Existenz ihrer übergeordneten Knoten. Wird eine Berechnungshierarchie entfernt, können die verwaisten Kindknoten Bestandteil neuer Berechnungen werden. Des Weiteren erlaubt es dieses System einzelne Knoten mit geringem Aufwand flexibel auszutauschen. Dadurch wird eine einfache Durchführung von Parameterstudien ermöglicht.

Berechnungshierarchie zur Auswertung der Druckkraftberechnung nach Morison	Solution Explorer	
	Name	Typ
	Morison_1234	Morison
	Wellentheorie_xy	Lineare Wellentheorie
	KSystem_Spundwand	Koordinatensystem
	TideWelle_Jan-1999	Wellenklima
	meinPfahl	Pfahl
	Nordseewasser	Wasser
	Wellentheorie_xy	Lineare Wellentheorie
	KSystem_Spundwand	Koordinatensystem
Berechnungshierarchie zur Auswertung der Linearen Wellentheorie	TideWelle_Jan-1999	Wellenklima
	meinPfahl	Pfahl
	Nordseewasser	Wasser
	KSystem_Spundwand	Koordinatensystem
	TideWelle_Jan-1999	Wellenklima
	meinPfahl	Pfahl
	Nordseewasser	Wasser

Abbildung 4.3: Konzeptuelle Darstellung der Berechnungshierarchien.

Verändern der Knoteneigenschaften

Die Berechnungsparameter können durch Änderung der Knoteneigenschaften eingebracht werden. Die Modifikation der Parameter geschieht über die Tabellendarstellung im Properties Explorer. Abbildung 4.4 zeigt eine mögliche Ansicht, wenn in der Baumstruktur aus Abbildung 4.3 der Knoten *KSystem_Spundwand* ausgewählt wird. Durch Klick in die rechte Spalte kann der Nutzer Zellen der Tabelle in den Bearbeitungsmodus versetzen und editieren. Dadurch können Parameter modifiziert werden.

Properties Explorer	
Eigenschaftsname	Wert
Typ	Koordinatensystem
Name	KSystem_Spundwand
x [m]	<input type="text" value="7.00"/>
t [s]	-5.30
z [m]	-15.00

Abbildung 4.4: Konzeptuelle Darstellung der Knoteneigenschaften.

Visualisierung der Berechnungsergebnisse

Jede vollständige Berechnungshierarchie bietet die Möglichkeit der Auswertung an. Dazu werden die Ergebnisse der Berechnung im linken Bereich nach Abbildung 4.1 dargestellt. Für die Visualisierung der Ergebnisse stehen grafische Elemente wie Diagramme, Animationen,

Text, etc. zur Verfügung. Jeder Knoten kann unterschiedliche Arten von Visualisierungen anbieten. Die Berechnungshierarchie, die in Abbildung 4.3 durch den Knoten *Wellentheorie_xy* repräsentiert wird, enthält die Berechnungsvorschriften der Linearen Wellentheorie nach Airy & Laplace. Darunter befinden sich unter anderem die Ausprägung des Wasserspiegels sowie die Dynamik des Wassers unterhalb des Wasserspiegels. Trotz ihrer thematischen Ähnlichkeit differieren die Visualisierungen. Der Benutzer kann eine dieser Darstellungen auswählen und anzeigen lassen.

4.2 Die Basisarchitektur

Die Anforderung an die Applikation unabhängig von der Plattform funktionsfähig zu bleiben, wird durch die Verwendung des Visualisierungsframework *Qt* erreicht. Dadurch kann die Software auf verschiedenen Systemen ausgeführt werden, ohne sie anpassen zu müssen. Alternativen zu der *cross-plattform*-Lösung mit *Qt* bieten die nativen Frameworks *WinForms* für Windows und *Cocoa* für macOS. Um später den Austausch der Visualisierung zugunsten einer nativen Darstellung zu ermöglichen, wird die in Abschnitt 2.7 beschriebene Passive-View-Variante der MVP-Architektur gewählt. Es entsteht eine Applikationsstruktur nach Abbildung 4.5.

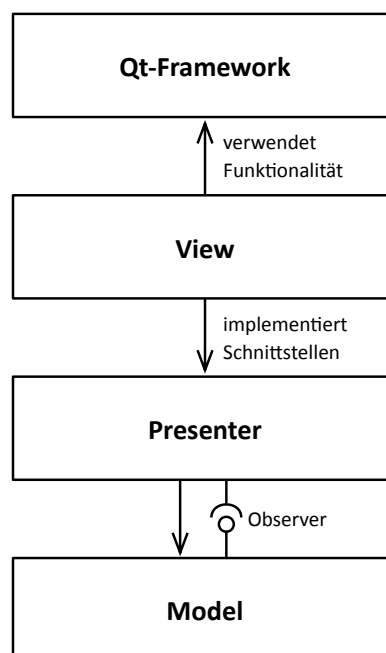


Abbildung 4.5: Architektur der Applikation CECLOUD.

Model – Das Modell

Die Logik der Problemdomäne ist Gegenstand des Modells. Dazu gehören das Erstellen und Bearbeiten der Elemente der Baumstruktur, die Organisation dieser in den Berechnungshierarchien sowie die Berechnung der Ergebnisse.

View

Die View übernimmt die Darstellung auf dem Bildschirm unter Ausnutzung der Funktionalität des gewählten Visualisierungsframeworks *Qt*. Dazu gehören die Darstellung der Elemente in der Baumstruktur, die Anzeige der individuellen Eigenschaften im Properties Explorer und die Visualisierung der berechneten Ergebnisse. Entsprechend des Passive-View-Musters ist die Logik der Viewklassen beschränkt. Sie ist lediglich auf das Layout der Applikationsfenster, sowie die korrekte Darstellung der vom Presenter übergebenen Ergebnisse ausgelegt. Benutzereingaben, welche die Modifikation des Modells erfordern, werden an die Presenter-Schicht weitergegeben.

Presenter

Die Schnittstelle zwischen dem Modell und der View wird durch die Presenter-Schicht gebildet. Sie ist einerseits für die Aktualisierung der Darstellung, andererseits für die Verarbeitung von Benutzereingaben aus der View zuständig. Zur Darstellung werden die Ergebnisse und Daten aus dem Modell an die View delegiert. Die Veränderung des Modells erfolgt durch die Presenterklassen. Sie können das Hinzufügen von Elementen in die Baumstruktur, die Verbindung der Elemente zu hierarchischen Strukturen und die Veränderung ihrer Eigenschaften auslösen. Zu der Presenter-Schicht gehören die konkreten Presenterklassen sowie die abstrakten Schnittstellen für die konkreten Klassen der View.

4.3 Verwendete Technologie

In diesem Abschnitt werden die wichtigsten für die Implementierung der Applikation verwendeten Technologien zusammengestellt. Die Umsetzung erfolgt mit der Programmiersprache C++, CMake, Qt sowie QCustomplot.

Die Programmiersprache C++

Entwickelt wurde C++ in den 1980er Jahren. Sie basiert auf der Programmiersprache C und erweitert diese um die Möglichkeit des objektorientierten Programmieransatzes. Daher unterstützt sie die in Abschnitt 2.4 erläuterten Schlüsselemente der OOP. Üblicherweise setzt sich ein Programm aus einer Vielzahl an Dateien zusammen, in denen sich der Quelltext befindet. Um daraus ein ausführbares Programm zu erzeugen, muss der Quelltext durch einen Compiler übersetzt werden. Jeder Compiler erzeugt das ausführbare Programm speziell für eine *hardware-software*-Kombination. Durch die Verwendung verschiedener Compiler kann das System plattformunabhängig ausgeführt und ausgeliefert werden. Die Wahl dieser Programmiersprache gewährleistet somit die zuvor geforderte Unabhängigkeit bezüglich verschiedener Betriebssysteme (Stroustrup, 2015).

CMake

Um die Applikation auf verschiedenen Plattformen ausführen zu können, muss sie entsprechend kompiliert werden. Für jede *hardware-software*-Kombination werden dafür unterschiedliche Konfigurationen benötigt. CMake ermöglicht die Konfiguration unabhängig von der Plattform und von dem Compiler zu gestalten. Dadurch kann die Software flexibel für unterschiedliche Plattformen erstellt und ausgeführt werden.

Qt – Ein Framework für grafische Darstellungen

Für den Entwurf einer grafischen Benutzeroberfläche kann auf die Funktionalität bereits bestehender Frameworks zurückgegriffen werden. Die Visualisierung der Applikation CECLOUD erfolgt mit dem Framework Qt. Es stellt visuelle Elemente zur Verfügung, aus deren Kombination eine GUI erstellt werden kann. Des Weiteren kann Qt durch entsprechende Compiler auf verschiedenen Plattformen verwendet werden und unterstützt somit die Unabhängigkeit der Applikation von dem Betriebssystem. Darüber hinaus bietet Qt grafische Elemente, deren Umsetzung der nativer Frameworks detailgetreu nachempfunden ist.

QCustomPlot

Zur Visualisierung der Berechnungsergebnisse werden innerhalb der Applikation Diagramme verwendet. Diese basieren auf der Klassendefinition QCustomPlot, bei der es sich um eine *open source* Erweiterung des Qt-Frameworks handelt.

KAPITEL 5

Entwurf des Modells

In diesem Abschnitt wird der Entwurf des Modells beschrieben und erläutert. Zunächst wird geschildert wie die Berechnungshierarchien in Form von Klassen und Objekten abgebildet werden können. Basierend auf dieser Klassenstruktur wird die Funktionalität des Modells schrittweise erweitert. Zuerst wird eine Projekthierarchie entworfen, in der die Knoten der Berechnungshierarchien abgespeichert werden können. Als nächstes wird die Erzeugung der Berechnungselemente thematisiert. Im Anschluss wird der Mechanismus dargestellt, der zum Verbinden von Knoten verwendet wird, um die in Abbildung 4.3 gezeigte hierarchische Struktur der Knoten zu erzeugen. Danach wird auf die Funktionalität eingegangen, durch die die Eigenschaften der Knoten verändert werden können. Zuletzt wird die Umsetzung des Command Patterns in der Applikation CECloud gezeigt, die es erlaubt, ausgeführte Aktionen zu widerrufen oder gegebenenfalls erneut auszuführen.

5.1 Die Berechnungshierarchie als Klassenstruktur

Wie in Abbildung 3.1 und 3.2 dargestellt, können die Berechnungsformeln in einer hierarchischen Struktur organisiert werden. Dabei stellen die untergeordneten Knoten Informationen bereit, die für die Berechnung übergeordneter Elemente notwendig sind. Abbildung 5.1 zeigt die in Abbildung 4.1 erläuterte Hierarchie zur Berechnung der Druckkräfte nach Morison in Form einer Klassenstruktur.

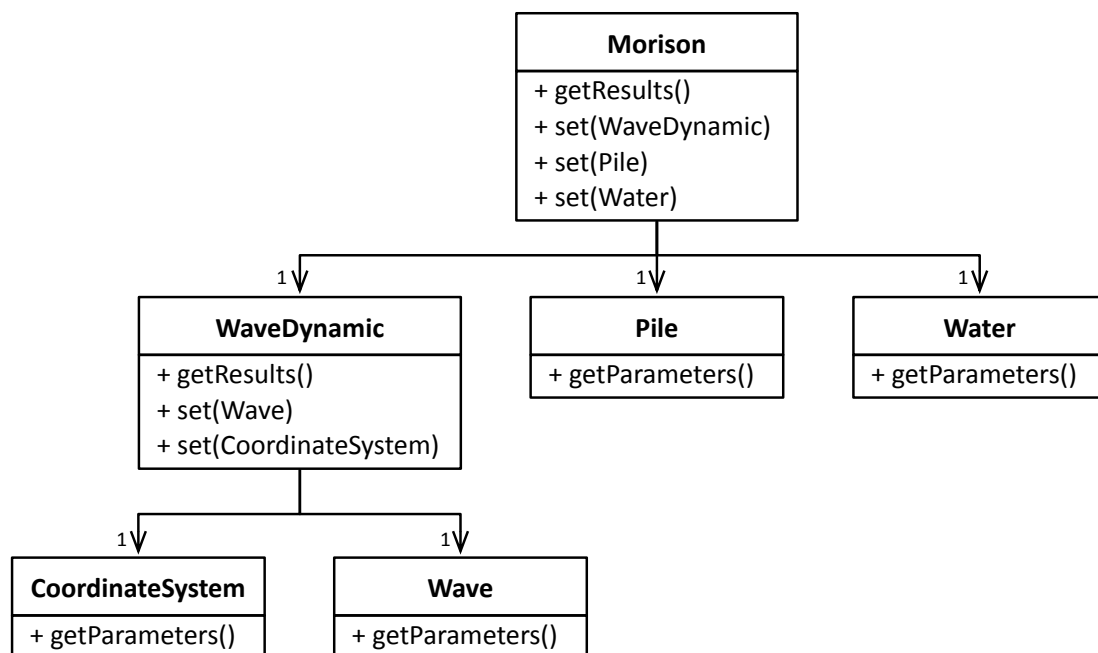


Abbildung 5.1: Umsetzung der Berechnungshierarchie aus Abbildung 3.1 als Klassenstruktur.

Es ist zu beachten, dass einige Klassendefinitionen nur für die Zusammenfassung und Bereitstellung thematisch ähnlicher Daten zuständig sind und somit keine Berechnungsanweisungen beinhalten. Übergeordnete Knoten erfragen unter Verwendung der „getter“-Methoden die gespeicherten Parameter und lassen diese in ihre Berechnung einfließen. Die „getter“-Methoden werden hier schematisch durch die „getParameters“-Methoden repräsentiert. Tatsächlich existiert zu jedem Berechnungsparameter eine spezielle „getter“-Methode. Die „getResults“-Methoden symbolisieren die Berechnungsoperationen von Knoten, die mathematische Formeln beinhalten. Kinder werden mithilfe von „setter“-Methoden in die übergeordneten Knoten gesetzt.

Listing 5.1 zeigt einen Ausschnitt von Pseudocode zur Erstellung einer Berechnungshierarchie nach Abbildung 5.1.

```
1 //Erzeugen eines Objekts vom Typ Morison
  Morison morison = new Morison()
3
4 //Erzeugen und Setzen der Kindknoten für Morison
5 Pile pile = new Pile(parameters)
  Water water = new Water(parameters)
7 WaveDynamic waveDynamic = new WaveDynamic()
8
9 morison.set(pile)
  morison.set(water)
11 morison.set(waveDynamic)
12
13 //Erzeugen und Setzen der Kindknoten für WaveDynamic
  Wave wave = new Wave(parameters)
15 CoordinateSystem coordinateSystem = new CoordinateSystem(parameters)
16
17 waveDynamic.set(wave)
  waveDynamic.set(coordinateSystem)
19
  Write(morison.getResults())
```

Listing 5.1: Pseudocode zur Erzeugung einer Berechnungshierarchie nach Abbildung 5.1.

In diesem Beispiel wird ein Objekt vom Typ `Morison` erstellt, das im letzten Schritt ausgewertet werden soll. Damit die Berechnung des `Morisons` einwandfrei stattfinden kann, müssen ihm Objekte der Typen `WaveDynamic`, `Pile` und `Water` zugewiesen werden. Zunächst werden dafür Instanzen der drei Klassen erstellt und über die „setter“-Methoden zu dem `Morison` hinzugefügt. Zur Berechnung werden die Ergebnisse des Objekts vom Typ `WaveDynamic` benötigt. Hierfür bedarf es Instanzen der Klassen `CoordinateSystem` und `Wave`, die der `WaveDynamic` über die „setter“-Methoden gesetzt werden. Jede Position der Berechnungshierarchie ist nun besetzt, die Auswertung der mathematischen Formeln des `Morisons` kann fehlerfrei durchgeführt werden.

Der Aufruf der „`getResults`“-Methode in Zeile 20 läutet die Berechnung des `Morisons` ein. Dazu werden alle benötigten Informationen der untergeordneten Knoten zusammengetragen. Als erstes stößt das `Morison`-Objekt durch die „`getResults`“-Methode die Berechnung der `WaveDynamic` an. Diese ermittelt über die „`getParameters`“-Methoden die Parameter der Instanzen vom Typ `CoordinateSystem` und `Wave`. Nach der Berechnung werden die Ergebnisse

an das Morison überliefert. Nun können die Parameter der Objekte Pile und Water erfragt und die Berechnung ausgeführt werden. Das Ergebnis wird an die „Write“-Methode übergeben, die es auf der Konsole ausgibt.

5.2 Die Projekthierarchie

Damit die Elemente der Berechnungshierarchie dauerhaft in der Applikation zur Verfügung stehen, werden sie während der Laufzeit des Programms in einem Objekt der Klasse ProjectHierarchy abgelegt und gespeichert. Die Struktur des Systems wird in Abbildung 5.2 gezeigt.

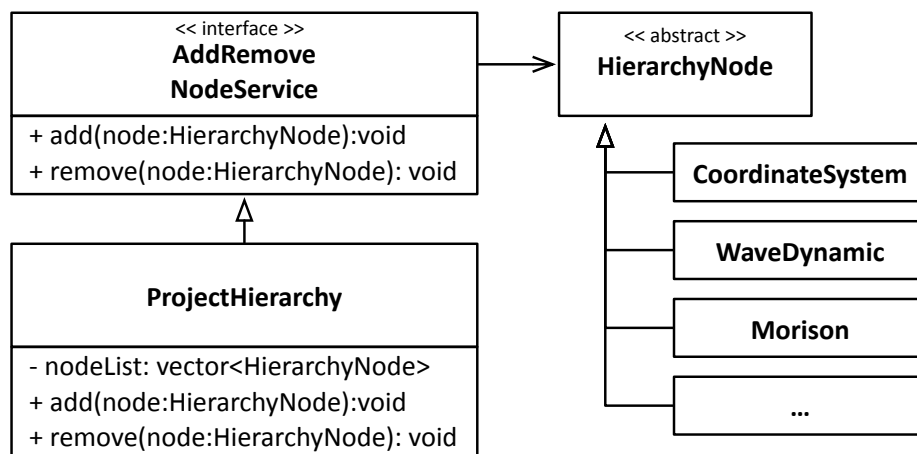


Abbildung 5.2: Speichern der HierarchyNodes in der ProjectHierarchy während der Laufzeit der Applikation.

Die Möglichkeit der Erweiterung der Software um neuartige Berechnungselemente kann durch das Einführen der abstrakten Schnittstelle HierarchyNode gewährleistet werden. Jede neue Subklasse der Schnittstelle kann durch die „add“-Methode in die ProjectHierarchy aufgenommen und mit der „remove“-Methode entfernt werden. Damit andere Klassen die Funktionalität der ProjectHierarchy verwenden können, ohne das DIP zu verletzen, stellt sie die Schnittstelle AddRemoveNodeService zur Verfügung.

5.3 Erzeugung der Hierarchieknoten

Wie in Abschnitt 2.6.1 beschrieben, ist der Aufruf von Konstruktoren ein Verstoß gegen das DIP. Aus diesem Grund findet das Factory Pattern analog zu Abbildung 5.3 Anwendung.

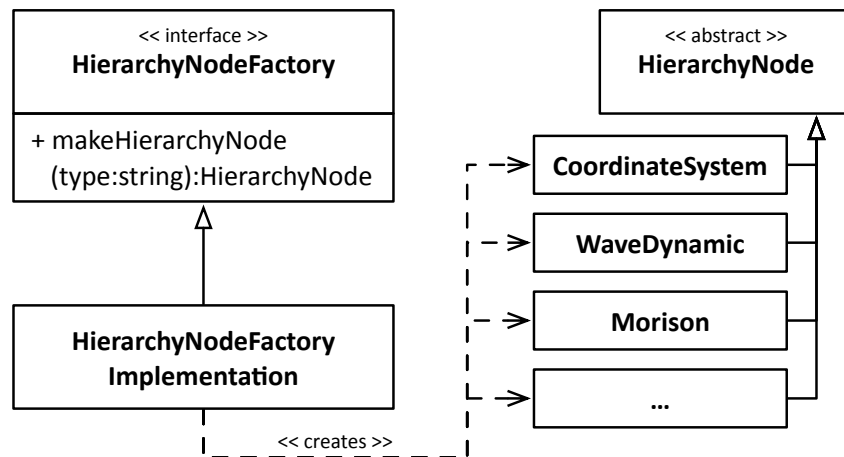


Abbildung 5.3: Anwendung des Factory Patterns zur Erzeugung von HierarchyNodes.

Die Fabrik beinhaltet eine Methode zur Erzeugung der verschiedenen Subtypen. Der HierarchyNodeFactoryImplementation wird ein string mitgegeben, anhand dessen sie die zu erzeugende Subklasse identifizieren kann. Listing 5.2 zeigt die Umsetzung der Fabrik.

```

1 class HierarchyNodeFactoryImplementation : public HierarchyNodeFactory{
2     public:
3     HierarchyNode* makeHierarchyNode(string type){
4         if(type == "Coordinate System") return new CoordinateSystem();
5         if(type == "Wave Dynamic") return new WaveDynamic();
6         if(type == "Morison") return new Morison();
7         if(type == "...") return new ...();
8
9         else throw UnknownTypeException();
10    }
11 }

```

Listing 5.2: Quelltext der HierarchyNodeFactoryImplementation nach Abbildung 5.3.

Innerhalb der „makeHierarchyNode“-Methode werden die übergebenen strings überprüft und entsprechend ein neues Objekt zurückgegeben. Kann der string keiner Klasse zugeordnet werden, löst die Fabrik eine UnknownTypeException aus, die den Programmablauf aufgrund des aufgetretenen Fehlers unterbricht.

5.4 Verbinden passender Knoten

Das Fabrik-Muster aus Abbildung 5.3 führt dazu, dass die Typinformationen der Hierarchy-Nodes verloren gehen, sobald diese die HierarchyNodeFactoryImplementation verlassen. Für den Aufruf der „setter“-Methoden zur Verbindung der Knoten aus Abbildung 5.1 sind die konkreten Typen der Objekte notwendig. Zur Rückgewinnung der Typinformationen wird das acyclic Visitor Pattern analog zu Abbildung 5.4 umgesetzt.

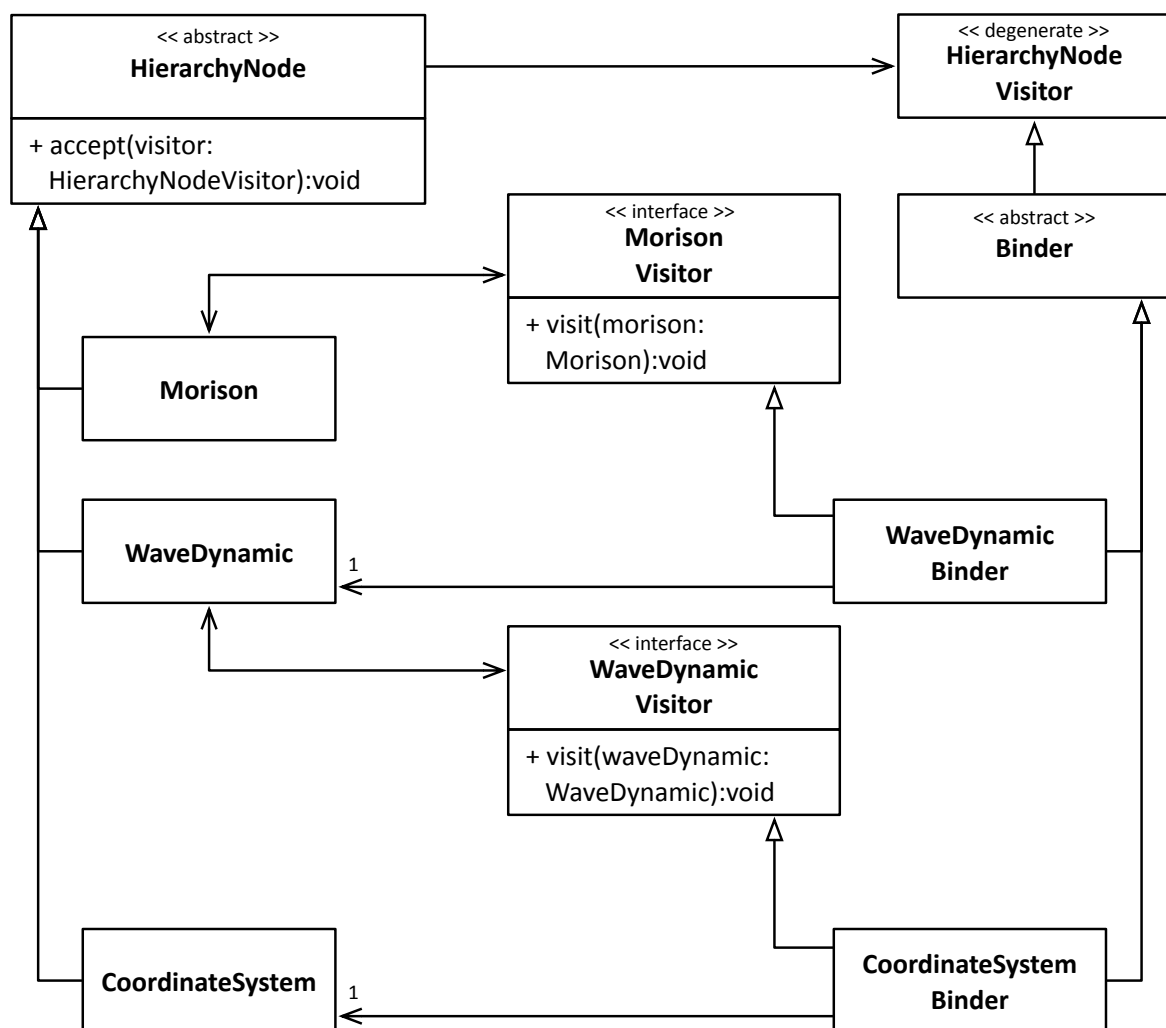


Abbildung 5.4: Binding-Mechanismus unter Verwendung des acyclic Visitor Patterns.

Wie in Abschnitt 2.6.3 beschrieben, können Klassenhierarchien durch die Verwendung des acyclic Visitor Patterns erweitert werden, ohne selbst modifiziert zu werden. Zu diesem Zweck wird eine weitere Klassenhierarchie eingeführt, welche die Besucherklassen beinhaltet. Innerhalb

der Besucher wird die zu erweiternde Funktionalität gekapselt. Durch den *double-dispatch*-Mechanismus können die Besucher die konkreten Typen der *HierarchyNodes* feststellen und die korrekten „setter“-Methoden aufrufen. Jede Klasse, die als übergeordneter Knoten Kinder aufnehmen kann, definiert eine typspezielle Besucherschnittstelle, in der das Verhalten festgelegt wird, dass zum Besuchen der Knoten benötigt wird. Für die Klasse *WaveDynamic* wird die abstrakte Schnittstelle *WaveDynamicVisitor* eingeführt, für die *Morison*-Klasse *MorisonVisitor*. Die Besucherhierarchie beinhaltet zu jeder Knotenklasse, die als Kind gebunden werden kann, einen Binder, der für den Aufruf der „setter“-Methoden zuständig ist. Um einen Knoten des Typs *CoordinateSystem* an ein *WaveDynamic* zu binden, wird ein *CoordinateSystemBinder* erstellt, dem das zu bindende *CoordinateSystem* im Konstruktor übergeben wird. Anschließend wird durch den Aufruf der „accept“-Methode des *WaveDynamics* der *double-dispatch*-Mechanismus ausgelöst. Der degenerierte *HierarchyNodeVisitor* wird in einen *WaveDynamicVisitor* umgewandelt. Gelingt die Typumwandlung, wird dieser durch die „visit“-Methode zum Besuch eingeladen. Dadurch wird der Typ des *WaveDynamics* zurückgewonnen und der Besucher kann die entsprechende „setter“-Methode aufrufen, um das *CoordinateSystem* zu setzen. Zur Erzeugung der Binder-Objekte wird erneut das Fabrik-Muster angewendet. Die Umsetzung der *BinderFactory* wird anhand von Abbildung 5.5 und Listing 5.3 verdeutlicht.

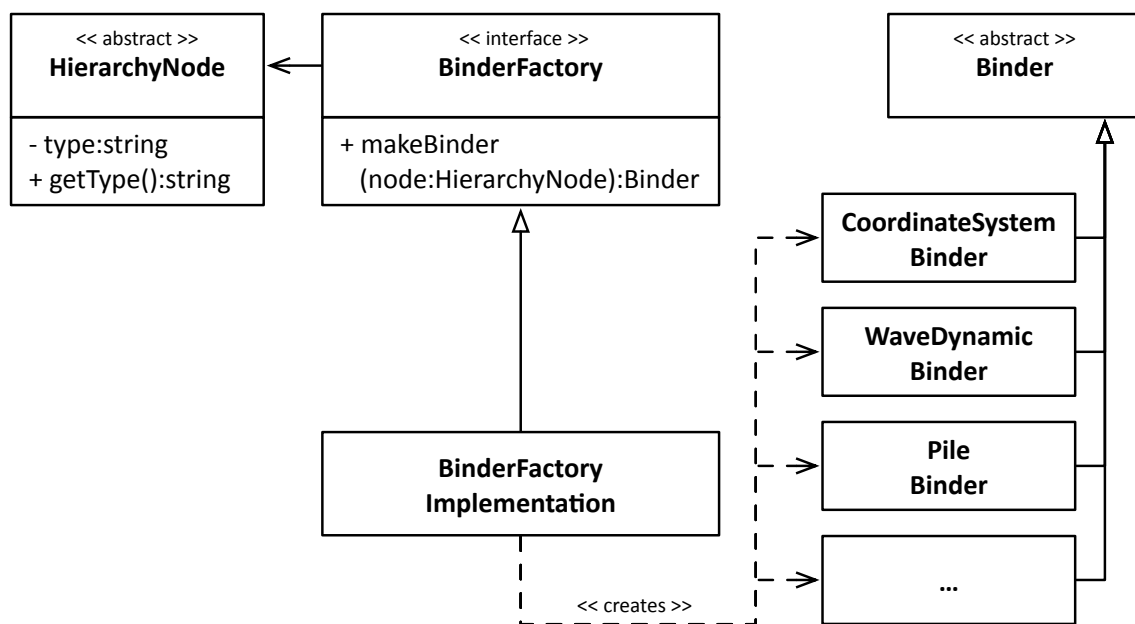


Abbildung 5.5: Die Umsetzung der *BinderFactory* zur Erzeugung von Bindern.

```

class BinderFactoryImplementation : public BinderFactory {
2   public:
    Binder* makeBinder(HierarchyNode* node){
4       if(node->getType() == "Coordinate System")
           return new CoordinateSystemBinder(castToCoordinateSystem(node));

6       if(node->getType() == "Wave Dynamic")
           return new WaveDynamicBinder(castToWaveDynamic(node));

8       if (node->getType() == "...")
           return new ...Binder(castTo...(node));
10      }

12     private:
        CoordinateSystem* castToCoordinateSystem(HierarchyNode* node){
14            return (CoordinateSystem*) node;
        }
        WaveDynamic* castToWaveDynamic(HierarchyNode* node){
16            return (WaveDynamic*) node;
        }
        ...* castTo...(HierarchyNode* node){
18            return (...*) node;
        }
20    }
22 }
24 }

```

Listing 5.3: Typumwandlung innerhalb der BinderFactory zu den konkreten Typen der HierarchyNode-Hierarchie.

Für das Erstellen der Binder wird der HierarchyNode benötigt, der als Kind gebunden werden soll. Dieser wird der „makeBinder“-Methode übergeben. Entsprechend des konkreten Typs des Knotens erzeugt die Fabrik ein passendes Binder-Objekt. Zunächst wird die „getType“-Methode verwendet um den konkreten Typ des Knotens zu ermitteln. Zurückgeliefert wird ein string anhand der die Subklasse des HierarchyNodes identifiziert werden kann. Anschließend wandelt die Fabrik den Knoten in den Subtyp um. Im letzten Schritt wird das entsprechende Objekt erzeugt und als Binder ausgeliefert.

Durch die Verwendung der Fabriken für die HierarchyNodes und die Binder sowie die Umsetzung des generischen Binding-Mechanismus entsteht eine Applikation, die hinsichtlich ihrer Berechnungselemente gemäß des OCP erweiterbar ist.

Listing 5.4 zeigt wie auf Basis der abstrakten Schnittstellen eine Berechnungshierarchie erzeugt werden kann. Als Beispiel dient dazu die Organisation der Elemente zur Berechnung der LWT nach Airy und Laplace. Die Abhängigkeiten zu den konkreten HierarchyNodes werden auf einen string minimiert. Änderungen dieser Klassen betreffen nicht die Funktionalität des Codes.

```
//Erzeugen der Fabriken
2 HierarchyNodeFactory* nodeFactory = new HierarchyNodeFactory();
BinderFactory* binderFactory = new BinderFactory();
4
//Erzeugen der HierarchyNodes
6 HierarchyNode* waveDynamic =
    nodeFactory->makeHierarchyNode("Wave Dynamic");
8 HierarchyNode* coordinateSystem =
    nodeFactory->makeHierarchyNode("Coordinate System");
10 HierarchyNode* wave = nodeFactory->makeHierarchyNode("Wave");
12
//Erzeugen der Binder
Binder* coordinateSystemBinder =
14     binderFactory->makeBinder(coordinateSystem);
Binder* waveBinder = binderFactory->makeBinder(wave);
16
//Verbinden der Knoten
18 waveDynamic->accept(coordinateSystemBinder);
waveDynamic->accept(waveBinder);
```

Listing 5.4: Quelltext zur Erzeugung einer Berechnungshierarchie zur Berechnung der Linearen Wellentheorie nach Airy und Laplace unter Verwendung der HierarchyNodeFactory, der BinderFactory und des Binding-Mechanismus nach Abbildung 5.4.

5.5 Bearbeitung der Knoteneigenschaften

Die HierarchyNodes werden innerhalb der Fabrik durch einen Standardkonstruktor erstellt, dem keine Parameter zur Erzeugung übergeben werden. Die Initialisierung erfolgt mit Standardwerten. Damit der Benutzer eine Berechnung ausführen kann, muss CECloud ihm die Möglichkeit geben die Berechnungsparameter ändern zu können. Für jeden Parameter weist eine Subklasse von HierarchyNode eine „setter“-Methode auf, die zur Modifikation verwen-

det wird. Die Erzeugung der Knoten durch die Fabrik führt zum Verlust der konkreten Typen, wodurch die „setter“-Methoden nicht mehr verwendet werden können. Zur Bearbeitung der Knoteneigenschaften werden die in Klassen der Editor-Hierarchie aus Abbildung 5.6 benutzt.

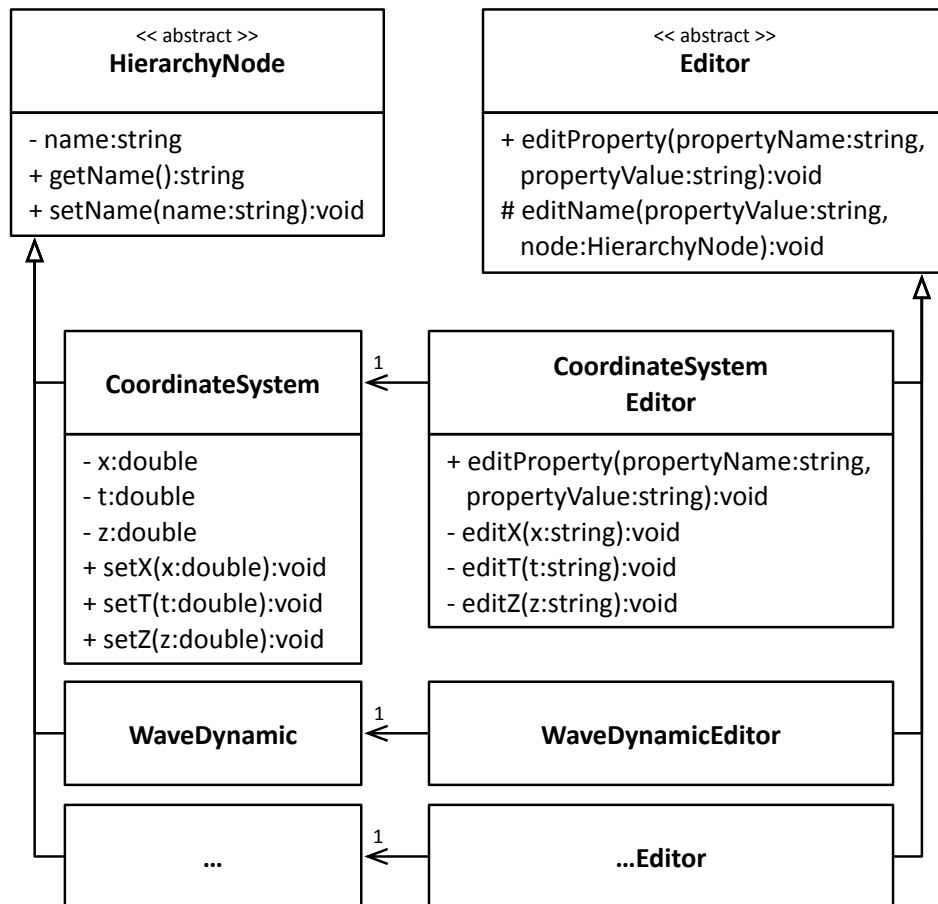


Abbildung 5.6: Umsetzung der Editor-Hierarchie zur Veränderung der Knoteneigenschaften.

Für jeden Knotentyp wird die parallele Editor-Hierarchie um ein Derivat erweitert. Die konkreten Editoren hängen von den jeweiligen Knotentypen ab und können somit die typspeziellen „setter“-Methoden aufrufen, um die Parameter zu verändern. Da auch für diese Hierarchie zur Erzeugung das Fabrik-Muster verwendet wird, ist der Applikation nur der abstrakte Typ Editor bekannt. Daher können nicht die typspeziellen „edit“-Methoden aufgerufen werden. Um eine Veränderung der Knoteneigenschaften zu veranlassen werden der Methode „editProperty“ strings übergeben. Dabei handelt es sich um ein Paar bestehend aus dem Namen der zu verändernden Eigenschaft und dem neuen Wert. Anhand des Namens kann der Editor die korrekte „setter“-Methode auswählen. Für den Aufruf der Methode wird der string, der den neuen Wert der Eigenschaft enthält, in das benötigte Format umgewandelt und übergeben.

Ein Knoten vom Subtyp `CoordinateSystem` wird mit dem Wert „0.00“ für seine Eigenschaften `x`, `t` und `z` erstellt. Zur Veränderung dieser Parameter enthält er die Methoden „setX“, „setT“ und „setZ“. Verlässt das `CoordinateSystem` die `HierarchyNodeFactoryImplementation`, können durch den Verlust des konkreten Typs diese Methoden nicht mehr aufgerufen werden. Listing 5.5 zeigt den Quelltext zur Umsetzung des `CoordinateSystemEditors`, der für die Änderung der Eigenschaften des `CoordinateSystems` zuständig ist.

```
1 class CoordinateSystemEditor : Editor{
   private:
3   CoordinateSystem* coordinateSystem;

   public:
5   CoordinateSystemEditor( CoordinateSystem* node) :
7       coordinateSystem( node) {}

9   virtual void editProperty( string propertyName, string propertyValue){
       if( propertyName == "Name")
11       editName( coordinateSystem, propertyValue); //aus Basisklasse
       if( propertyName == "x") editX( propertyValue);
13       if( propertyName == "t") editT( propertyValue);
       if( propertyName == "z") editZ( propertyValue);

15       else throw UnknownPropertyException();
17   }

   private:
19   void editX( string x){
21       coordinateSystem->setX( std:: stod( x));
       }
23   void editT( string t){
       coordinateSystem->setT( std:: stod( t));
25   }
       void editZ( string z){
27       coordinateSystem->setZ( std:: stod( z));
       }
29 }
```

Listing 5.5: Quelltext der Klasse `CoordinateSystemEditor` nach Abbildung 5.6.

Wird der Methode „editProperty“ als Eigenschaftsname einer der Identifikatoren „x“, „t“ oder „z“ übergeben, ruft der Editor die entsprechende private „edit“-Methode auf und übergibt den neuen Eigenschaftswert als string. Innerhalb der privaten Methoden wird dann der neue Wert in das passende Format umgewandelt, damit die „setter“-Methode aufgerufen werden kann. Zur Umwandlung des strings zu einem double wird hier die Funktion „std::stod“ der Standardbibliothek verwendet. Kann der übergebene string keiner Eigenschaft zugeordnet werden, löst der `CoordinateSystemEditor` eine `UnknownPropertyException` aus. Die Erzeugung der Editor-Objekte übernimmt eine `EditorFactory`, die analog zur `BinderFactory` nach Abbildung 5.5 und Listing 5.3 umgesetzt wird.

Listing 5.6 zeigt beispielhaft das Editieren eines Knotens vom Typ `CoordinateSystem` unter Verwendung der Editor-Klassen und der `EditorFactoryImplementation`.

```
1 //Erzeugung der Fabriken
HierarchyNodeFactoryImplementation* nodeFactory =
3     new HierarchyNodeFactoryImplementation();

5 EditorFactoryImplementation* editorFactory =
    new EditorFactoryImplementation();
7
//Erzeugen des CoordinateSystems
9 HierarchyNode* node =
    nodeFactory->makeHierarchyNode("Coordinate System");
11
//Erzeugen des CoordinateSystemEditors
13 Editor* editor = editorFactory->makeEditor(node);

15 //Verändern der Eigenschaften
editor->editProperty("x", "7.00");
17 editor->editProperty("t", "5.30");
editor->editProperty("z", "-15.00");
19 editor->editProperty("Name", "KSystem_Spundwand");
```

Listing 5.6: Veränderung der Berechnungsparameter eines `CoordinateSystem`-Objekts unter Verwendung der Editor-Klassenhierarchie und der `EditorFactory`.

5.6 Einführen einer Bearbeitungshistorie

Die Funktionalität einer Bearbeitungshistorie wird in CECLOUD analog zu Abschnitt 2.6.4 durch die Verwendung des Befehl-Musters erreicht. Das umgesetzte System wird anhand der Klassenstruktur in Abbildung 5.7 erläutert.

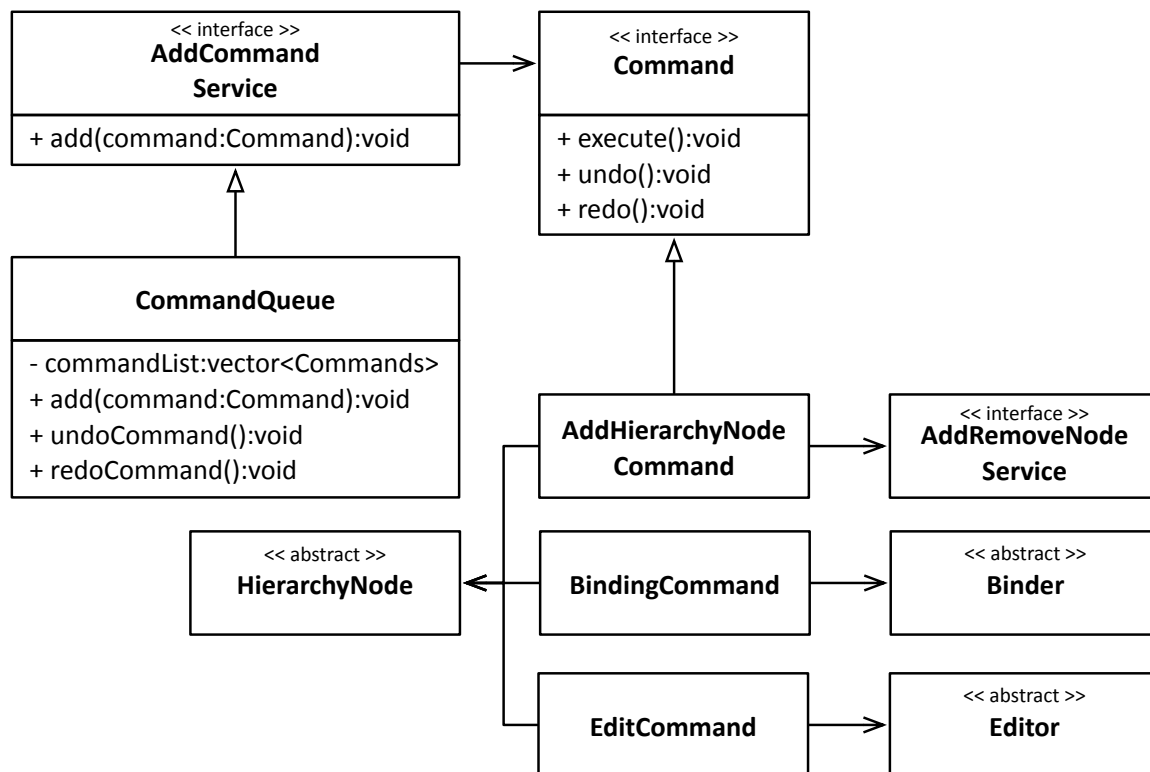


Abbildung 5.7: Umsetzung des Command Patterns mit **AddHierarchyNodeCommand**, **BindingCommand** und **EditCommand**.

Für die unterschiedlichen Änderungsarten der Modelldaten werden selbständige Command-Klassen angelegt, die parametrisierte Handlungen vorwärts und rückwärts ausführen können. Nach der Erzeugung wird das Befehlsobjekt der **CommandQueue** hinzugefügt. Für das Speichern von Befehlen stellt die **CommandQueue** die Schnittstelle **AddCommandService** zur Verfügung. Innerhalb der „add“-Methode wird der aufgenommene Befehl ausgeführt. Bei Aufruf der „execute“-Methode übernimmt der **AddHierarchyNodeCommand** das Hinzufügen von **HierarchyNodes** zum **AddNodeRemoveService**, der **BindingCommand** löst den *double-dispatch*-Mechanismus des acyclic Visitor Patterns aus und der **EditCommand** ruft die „editProperty“-Methode der **Editors** auf. Die „undoCommand“-Methode löst die „undo“-Methode des zuletzt ausgeführten Befehls aus, wodurch dieser widerrufen wird. So können

nacheinander alle gespeicherten Handlungen in umgekehrter Reihenfolge rückwärtig ausgeführt werden. Durch Aufruf der „redoCommand“-Methode wird die zuletzt widerrufenen Änderung des Modells erneut durchgeführt.

Exemplarisch für die Umsetzung der verschiedenen Befehlsklassen zeigt Listing 5.7 den Quelltext der Klasse AddHierarchyNodeCommand.

```
1 class AddHierarchyNodeCommand : public Command{
2     private:
3         HierarchyNode* node;
4         AddRemoveNodeService* nodeService
5
6     public:
7         AddHierarchyNodeCommand( HierarchyNode* node ,
8                                 AddRemoveNodeService* nodeService ) :
9             node(node) ,
10            nodeService(nodeService) {}
11
12     virtual void execute(){
13         nodeService->add(node);
14     }
15
16     virtual void undo(){
17         nodeService->remove(node);
18     }
19
20     virtual void redo(){
21         execute();
22     }
23 }
```

Listing 5.7: Quelltext der Klasse AddHierarchyNodeCommand.

Im Konstruktor werden dem Command die Parameter für das Ausführen der gewünschten Handlung übergeben. Diese speichert das erzeugte Befehlsobjekt als Instanzvariablen ab. Wird der Befehl ausgeführt, fügt er dem AddRemoveNodeService unter Verwendung der „add“-Methode den übergebenen Knoten hinzu. „undo“ entfernt den HierarchyNode mit Hilfe der „remove“-Methode. „redo“ enthält die Logik für die erneute Ausführung des Commands.

Die Erzeugung der Command-Objekte erfolgt durch die Umsetzung des Factory Patterns. Die

CommandFactory hat für jeden Befehlstypen eine Fabrikmethode, der die Parameter für die verschiedenen Commands übergeben werden. Nach der Erzeugung händigt sie es als Objekt vom Typ Command aus.

KAPITEL 6

Entwurf der Präsentation

Die Präsentation setzt sich aus der Presenter-Schicht und der View zusammen. Der Entwurf beider Aufgabenbereiche wird in diesem Kapitel erläutert. Zunächst wird die Implementierung einer Basisapplikation gezeigt, welche die Bereiche der konzeptuellen Darstellung aus Abschnitt 4.1 enthält. Die Applikation wird anschließend schrittweise um Funktionalität erweitert, die zur Darstellung und zur Manipulation der Modelldaten dient. In einem ersten Schritt wird die Visualisierung der Baumstruktur integriert. Dazu gehört zum einen der Aktualisierungsprozess der Baumstruktur auf Grundlage der Berechnungshierarchien, zum anderen das Einfügen und das Verbinden von Knoten. Als nächstes wird die Umsetzung der Tabellenansicht im Properties Explorer beschrieben. Im Einzelnen wird dabei auf die Anzeige und die Veränderung der Knoteneigenschaften eingegangen. Anschließend wird erklärt wie das Befehl-Muster über die grafische Benutzeroberfläche ausgelöst werden kann. In einem letzten Schritt wird die Erzeugung, die Darstellung sowie die Aktualisierung der grafischen Bestandteile beschrieben, welche die Ergebnisse der Berechnungshierarchien visualisieren.

6.1 Umsetzung der Basisapplikation

Ausgangspunkt des Entwurfs der Präsentation ist die Basisapplikation, welche die in Abschnitt 4.1 beschriebenen Bereiche enthält. Darunter befinden sich der Solution Explorer zur Visualisierung der Berechnungshierarchien, der Properties Explorer zur Anzeige der knotenspezifischen Eigenschaften sowie der linke Bereich, in dem die Berechnungsergebnisse präsentiert werden. Abbildung 6.1 zeigt das Hauptanwendungsfenster der Applikation CECLOUD.

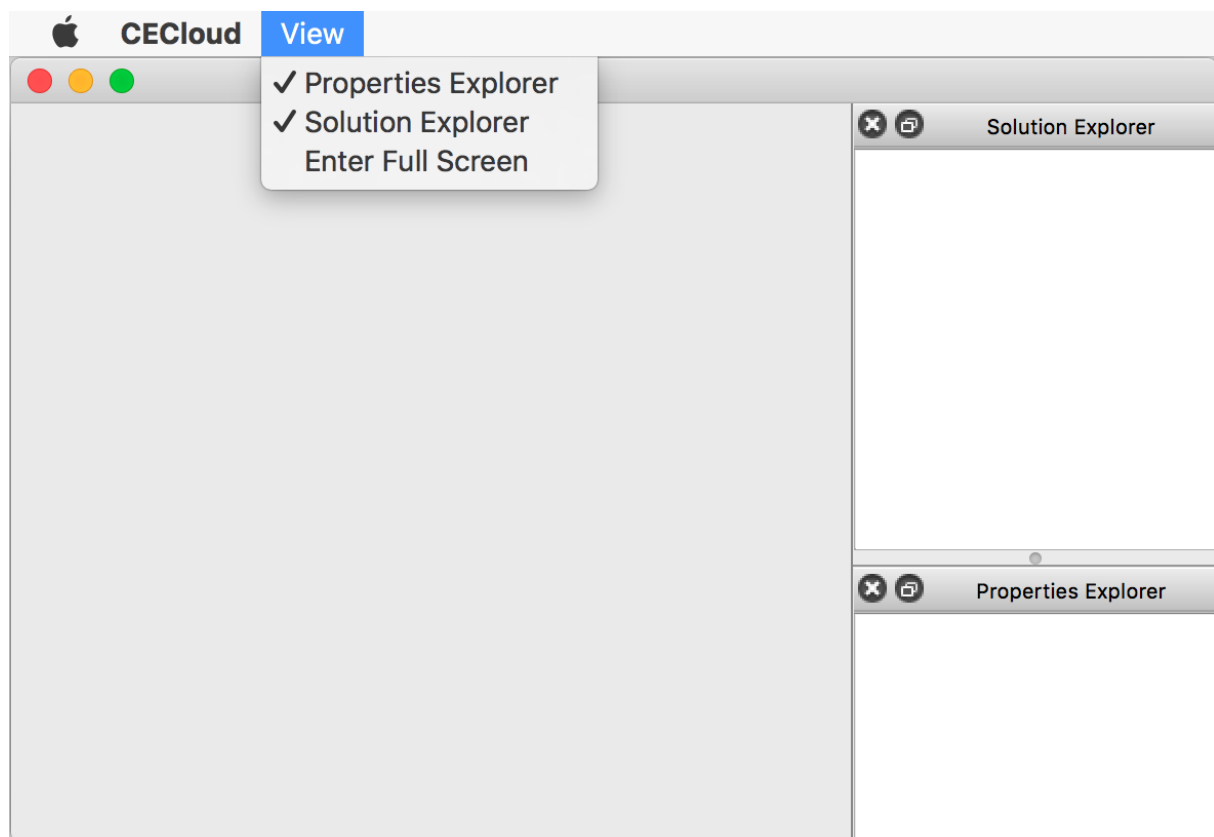


Abbildung 6.1: Hauptanwendungsfenster der Applikation CECLOUD.

Um den Raum für die Darstellung der Ergebnisse zu maximieren, soll dem Benutzer ermöglicht werden, die Explorer-Ansichten temporär zu verbergen. Dazu steht ihm in der Menüleiste das Menü „View“ zur Verfügung. Es enthält analog zu den Explorer-Ansichten die Einträge „Solution Explorer“ und „Properties Explorer“. Werden die Ansichten angezeigt, sind die Menüeinträge mit einem vorstehenden Haken gekennzeichnet. Diese erlöschen, sobald der entsprechende Explorer verborgen wird. Mit Klick auf die Einträge des Menüs kann die Sichtbarkeit der beiden Ansichten geändert werden. Basierend auf dem neuen Zustand der Explorer-Ansichten aktua-

lisiert sich dann das ViewMenu. Abbildung 6.2 zeigt eine mögliche Klassenstruktur zur Umsetzung der Basisapplikation.

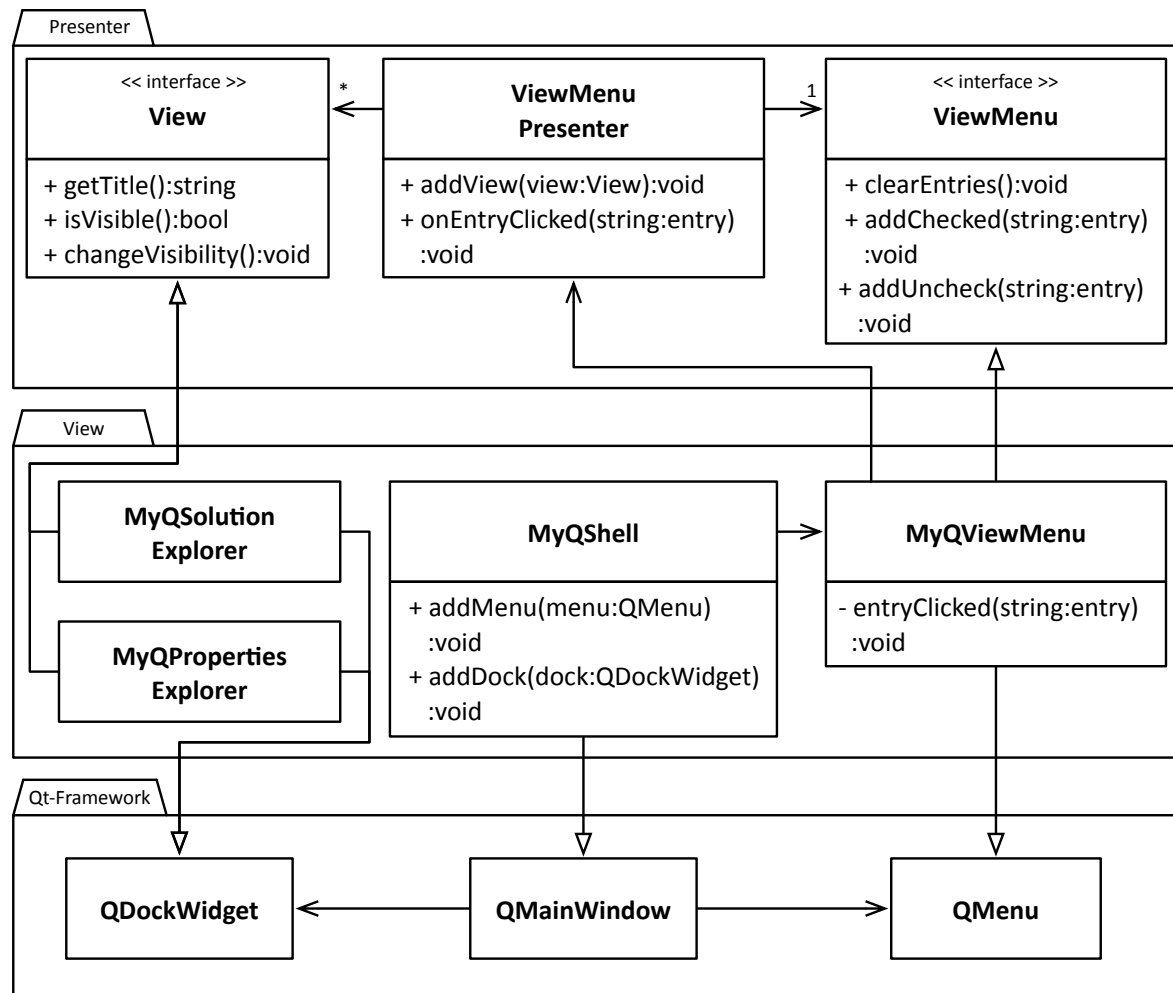


Abbildung 6.2: Umsetzung der Basisapplikation – Änderung der Sichtbarkeiten der Explorer-Ansichten durch den ViewMenuPresenter.

Die grafische Benutzeroberfläche besteht aus dem Hauptanwendungsfenster, den Explorer-Ansichten und dem „View“-Menu, die durch die Viewklassen `MyQShell`, `MyQSolutionExplorer`, `MyQPropertiesExplorer` und `MyQViewMenu` repräsentiert werden. Um die Funktionalität des Qt-Frameworks benutzen zu können, werden diese Klassen von den Qt-Klassen `QMainWindow`, `QDockWidget` und `QMenu` abgeleitet. Dadurch können die Viewklassen sowohl im Kontext des Frameworks zur Darstellung verwendet, als auch um applikationsspezifische Funktionalität erweitert werden.

Die `MyQShell` kann Instanzen der Klassen `MyQSolutionExplorer`, `MyQProperties`-

Explorer und MyQViewMenu über den Aufruf der „add“-Methoden aufnehmen. Sie übernimmt die Darstellung der grafischen Elemente unter Verwendung der Qt-Funktionalität. Die Kommunikation zwischen MyQViewMenu und den Explorer-Ansichten zum Ein- und Ausblenden ist Aufgabe des ViewMenuPresenters. Um die Unabhängigkeit der Presenterklasse von dem Qt-Framework zu erreichen, entstehen die Schnittstellen ViewMenu und View. Objekte der Klassen MyQSolutionExplorer und MyQPropertiesExplorer werden dem Presenter als Instanz vom Typ View hinzugefügt. Dieser speichert sie in einer Liste und fügt in das ViewMenu einen neuen Menüeintrag ein. Dazu ermittelt er über die „getTitle“-Methode den Titel der View. Objekte vom Typ MyQSolutionExplorer geben den string „Solution Explorer“ zurück. Instanzen der Klasse MyQPropertiesExplorer enthalten den Titel „Properties Explorer“. Je nach aktueller Sichtbarkeit der übergebenen View fügt der Presenter einen Menüeintrag ein, der entweder mit Haken gekennzeichnet ist oder nicht. Für die Information, ob die View sichtbar ist, steht die „isVisible“-Methode zur Verfügung.

Ein Klick auf die Menüeinträge löst in der Viewklasse MyQViewMenu die „entryClicked“-Methode aus. Entsprechend des Passive-View-Musters wird dieses Ereignis an den ViewMenuPresenter weitergegeben. Dies geschieht durch den Aufruf der „onEntryClicked“-Methode. Der Presenter iteriert durch seine Liste und vergleicht den geklickten Eintrag mit den Titeln der View-Objekte. Findet er einen passenden Titel, fordert er die zugehörige View auf ihre Sichtbarkeit zu ändern. Anschließend aktualisiert der Presenter das Menü. Dazu löscht er zunächst alle Einträge durch Aufruf der „clear“-Methode. Danach iteriert er durch die Liste der View-Objekte und fügt Einträge entsprechend der Titel und der Sichtbarkeiten in das Menü ein.

Bei der Umsetzung der Basisapplikation analog zu Abbildung 6.2 wird die Aktualisierung des Menüs von dem ViewMenuPresenter direkt nach dem Ändern der Sichtbarkeit der Explorer induziert. Dadurch kann der Aktualisierungsprozess nur ausgelöst werden, wenn die Änderungen aus dem ViewMenu herrühren. Die Ableitung der Klassen MyQSolutionExplorer und MyQPropertiesExplorer von QDockWidget erlaubt es dem Benutzer den Solution Explorer und den Properties Explorer auszublenden, ohne die Einträge des Menüs zu verwenden. Dafür kann das „X“ in der oberen linken Ecke der Explorer-Ansichten geklickt werden. Um die Einträge des Menüs mit dem Zustand der Explorer konsistent zu halten, müssen diese nach ihrem Ausblenden den ViewMenuPresenter benachrichtigen, der anschließend die Aktualisierung des ViewMenus vornimmt. Diese Funktionalität wird durch die Erweiterung des Systems nach Abbildung 6.3 erreicht.

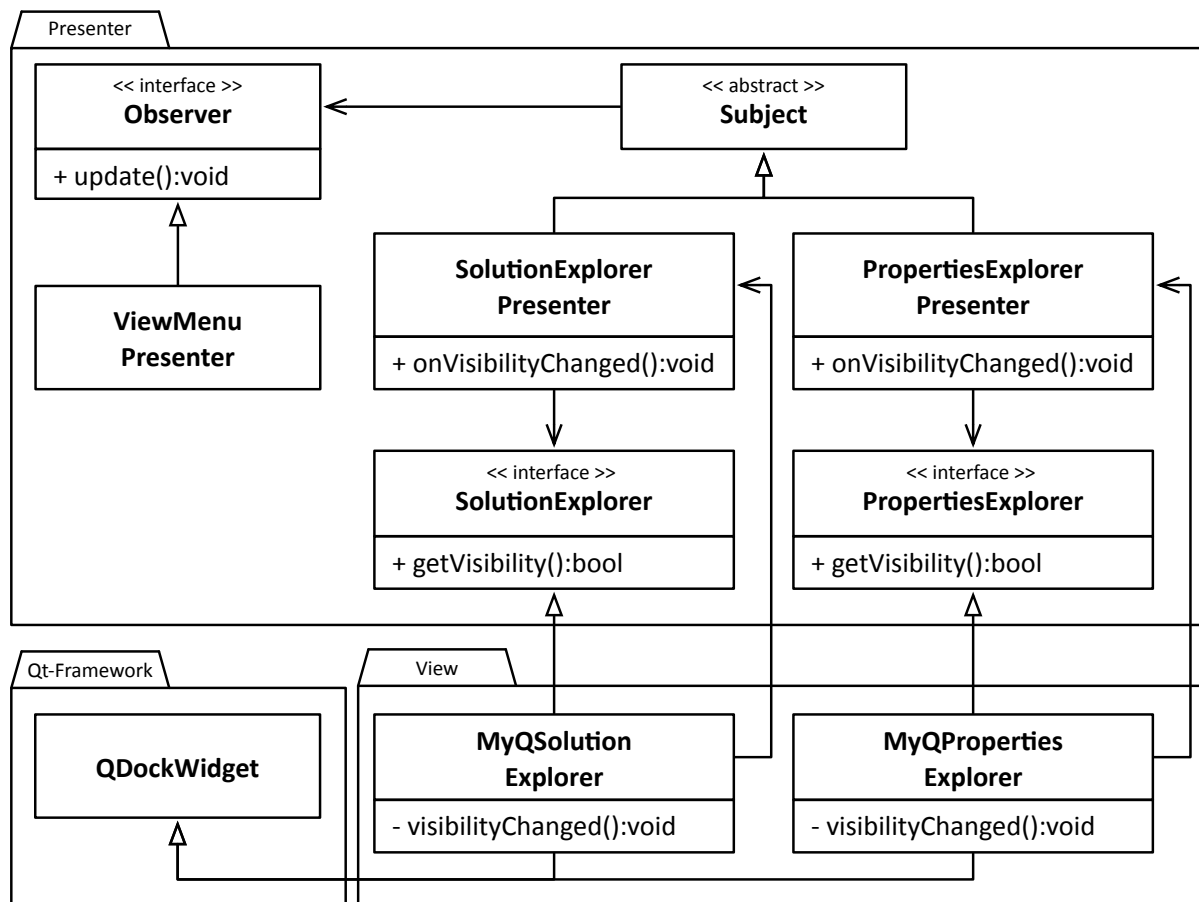


Abbildung 6.3: Umsetzung der Basisapplikation – Auslösen der Aktualisierung des ViewMenus.

Die Veränderung der Sichtbarkeiten löst in den Viewklassen die Methode „visibilityChanged“ aus. Um ein passives Verhalten der Viewklassen zu erzeugen, benachrichtigen diese ihre Presenter durch den Aufruf der „onVisibilityChanged“-Methode. Damit das ViewMenu an den neuen Zustand der View-Objekte angepasst werden kann, müssen die Presenter der Explorer-Ansichten den Aktualisierungsprozess des ViewMenuPresenters in Gang setzen. Um eine lose Kopplung zu erhalten wird dafür das Beobachter-Muster nach Abschnitt 2.6.5 angewendet. Der ViewMenuPresenter wird von der Schnittstelle Observer abgeleitet. Die Presenter der Explorer-Ansichten erben von der abstrakten Klasse Subject. Dadurch erhalten sie die Fähigkeit die „update“-Methode des ViewMenuPresenters aufzurufen, in der die Aktualisierung des ViewMenus stattfindet.

Um eine ausführbare Anwendung zu erhalten, welche die zuvor beschriebene Funktionalität aufweist, müssen Objekte der Klassen in der „main“-Methode erstellt und arrangiert werden. Die Umsetzung der „main“-Methode der Basisapplikation wird anhand von Listing 6.1 gezeigt.


```
1 int main(int argc, char **argv){
    QApplication a(argc, argv);

3
    //Erzeugung Shell
5    MyQShell* shell = new MyQShell();
    shell->showMaximized();

7
    //Erzeugung ViewMenu
9    MyQViewMenu* viewMenu = new MyQViewMenu();
    ViewMenuPresenter* viewMenuPresenter = new ViewMenuPresenter(viewMenu);
11    viewMenu->setPresenter(viewMenuPresenter);

13    //Erzeugung SolutionExplorer
    MyQSolutionExplorer* solutionExplorer = new MyQSolutionExplorer();
15    SolutionExplorerPresenter* solutionExplorerPresenter = new
        SolutionExplorerPresenter(solutionExplorer);
    solutionExplorer->setPresenter(solutionExplorerPresenter);

17
    //Erzeugung PropertiesExplorer
19    MyQPropertiesExplorer* propertiesExplorer = new MyQPropertiesExplorer();
    PropertiesExplorerPresenter* propertiesExplorerPresenter = new
        PropertiesExplorerPresenter(propertiesExplorer);
21    propertiesExplorer->setPresenter(propertiesExplorerPresenter);

23    //Hinzufügen zu Shell
    shell->addMenu(viewMenu);
25    shell->addDockWidget(solutionExplorer);
    shell->addDockWidget(propertiesExplorer);

27
    //Hinzufügen zum ViewMenuPresenter
29    viewMenuPresenter->addView(solutionExplorer);
    solutionExplorerPresenter->register(viewMenuPresenter);
31    viewMenuPresenter->addView(propertiesExplorer);
    propertiesExplorerPresenter->register(viewMenuPresenter);

33
    return a.exec();
35 }
```

Listing 6.1: Quelltext der main-Methode zur Umsetzung einer ausführbaren Basisapplikation.

Zunächst wird eine Instanz der Klasse `QApplication` erzeugt, durch welche der *flow of control* der Qt-Objekte geregelt wird. Anschließend wird ein Objekt der Klasse `MyQShell` erzeugt und angezeigt. Im nächsten Schritt werden die grafischen Elemente `MyQViewMenu`, `MyQSolutionExplorer` und `MyQPropertiesExplorer` mit ihren entsprechenden Presenterklassen erzeugt. Über die „setPresenter“-Methode werden den Objekten ihre Presenter übergeben. Damit das Hauptanwendungsfenster die grafischen Elemente darstellt, werden sie über die „add“-Methoden hinzugefügt. Um die Einträge des Menüs mit den Sichtbarkeiten der Views zu synchronisieren werden abschließend zum einen die View-Objekte dem `ViewMenuPresenter` hinzugefügt, zum anderen der `ViewMenuPresenter` bei den restlichen Presentern als Observer registriert.

6.2 Darstellung und Manipulation der Berechnungshierarchien

In diesem Abschnitt wird die Funktionalität der Applikation `CECloud` zur Darstellung und Manipulation der Berechnungshierarchien erläutert. Die Visualisierung der auf den `HierarchyNodes` basierenden Hierarchie erfolgt in Form einer Baumstruktur innerhalb des Solution Explorers. Zunächst wird erklärt wie die einzelnen `HierarchyNodes` durch `HierarchicItems` dargestellt werden. Anschließend wird beschrieben, wie sich die Baumstruktur aus den `HierarchicItems` zusammensetzt, sodass die gesamte Hierarchie angezeigt werden kann. Danach wird darauf eingegangen, wie sich die Baumstruktur an Änderungen der Modelldaten anpasst. Letztlich wird gezeigt wie der Benutzer unter Verwendung von Menüs in der Menüleiste das Einfügen und das Verbinden von Knoten veranlassen kann.

6.2.1 Darstellung der `HierarchyNodes` innerhalb der Baumstruktur

Die Baumstruktur im Solution Explorer dient zur Darstellung der sich im Modell befindlichen `ProjectHierarchy`, die aus `HierarchyNodes` besteht. Bei den `HierarchyNodes` handelt es sich um Modellobjekte, welche die Logik zur Berechnung beinhalten. Wie die Daten der Berechnungshierarchien auf dem Bildschirm präsentiert werden, ist innerhalb der Modelklassen unbekannt. Um das SRP nicht zu verletzen, befindet sich die Präsentationslogik in den `HierarchicItems`. Zu jeder Subklasse von `HierarchyNode` existiert ebenfalls ein Derivat in der Item-Hierarchie. Jedes `HierarchicItem`-Objekt ist für die Darstellung eines `HierarchyNode`-Objekts zuständig. Es ist bei diesem als Observer registriert und kann da-

durch Änderungen der Daten im Modell erfassen und neu darstellen. Abbildung 6.4 zeigt die parallelen Klassenstrukturen.

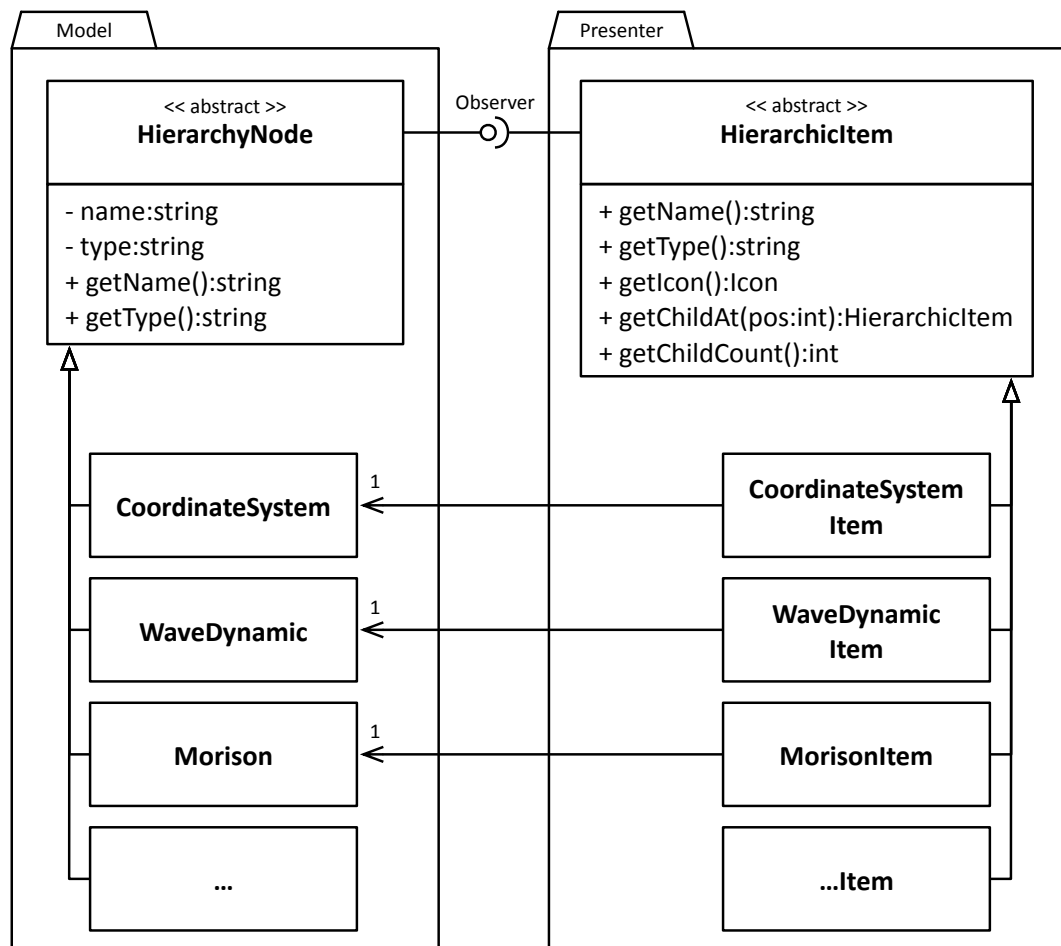


Abbildung 6.4: Parallele Klassenhierarchien der HierarchyNodes und HierarchicItems.

Die HierarchicItems können als grafische Aufbereitung der HierarchyNodes interpretiert werden. Sie sind auf die Knotentypen zugeschnitten und stellen die Daten der Modell-Objekte für die View entsprechend ihres Subtyps in der richtigen Reihenfolge bzw. Ordnung bereit. Ein Knoten vom Typ WaveDynamic beinhaltet Instanzvariablen vom Typ CoordinateSystem und Wave. In welcher Reihenfolge die Kindknoten in der Baumstruktur abgebildet werden, ist nicht bekannt. Diese Aufgabe wird von dem WaveDynamicItem übernommen. Des Weiteren umfassen die Item-Klassen typspezifische Icons für die Darstellung in der MyQTreeView.

Listing 6.2 zeigt exemplarisch die Umsetzung der HierarchicItems anhand des WaveDynamicItems.

```

1 class WaveDynamicItem : public HierarchicItem{
    private:
3     WaveDynamic* waveDynamic;
    HierarchicItem* coordinateSystemItem;
5     HierarchicItem* waveltem;
    Icon icon;
7
    public:
9     int getChildCount(){
        return 2; //genau zwei Kinder -> CoordinateSystemItem und Waveltem
11    }
    HierarchicItem* getChildAt(int position){
13        //spiegelt die Positionen der Kinder in der Baumstruktur wider
        if(position == 0) return coordinateSystemItem;
15        if(position == 1) return waveltem;

17        throw OutOfBoundsException();
    }
19    string getName(){
        return waveDynamic->getName();
21    }
    string getType(){
23        return waveDynamic->getType();
    }
25    Icon getIcon(){
        return icon;
27    }
}

```

Listing 6.2: Quelltext der Klasse WaveDynamicItem.

Damit es auf die Daten des Modell-Objektes zugreifen kann, besitzt es eine Instanz der Klasse WaveDynamic. Anstatt Referenzen zu den Kindknoten der WaveDynamic enthält das WaveDynamicItem HierarchicItems in denen sich die Kindknoten befinden. Die Darstellungsreihenfolge der Kinder in der Baumstruktur wird durch die „getChildAt“-Methode festgelegt. Der Name und der Typ der WaveDynamic wird über die Methoden „getName“ und „getType“ bereitgestellt.

6.2.2 Darstellung der Baumstruktur in dem Solution Explorer

Die zur Visualisierung der Berechnungshierarchie gewählte Baumstruktur basiert auf den Klassen `QTreeView` und `QAbstractItemModel` des Frameworks Qt. Die Darstellung der Baumstruktur wird dabei von der `QTreeView` übernommen. Das `QAbstractItemModel` stellt die Daten für die Visualisierung zur Verfügung. In der Applikation `CECloud` werden die abgeleiteten Klassen `MyQTreeView` und `MyQTreeModel` verwendet. Die innerhalb der Explorer-Ansicht dargestellte hierarchische Struktur setzt sich aus `HierarchicItems` zusammen. Repräsentiert wird sie durch das Wurzelement des `MyQTreeModels` vom Typ `TreeRoot`. Abbildung 6.5 zeigt die Klassenbeziehungen zur Darstellung der Baumstruktur.

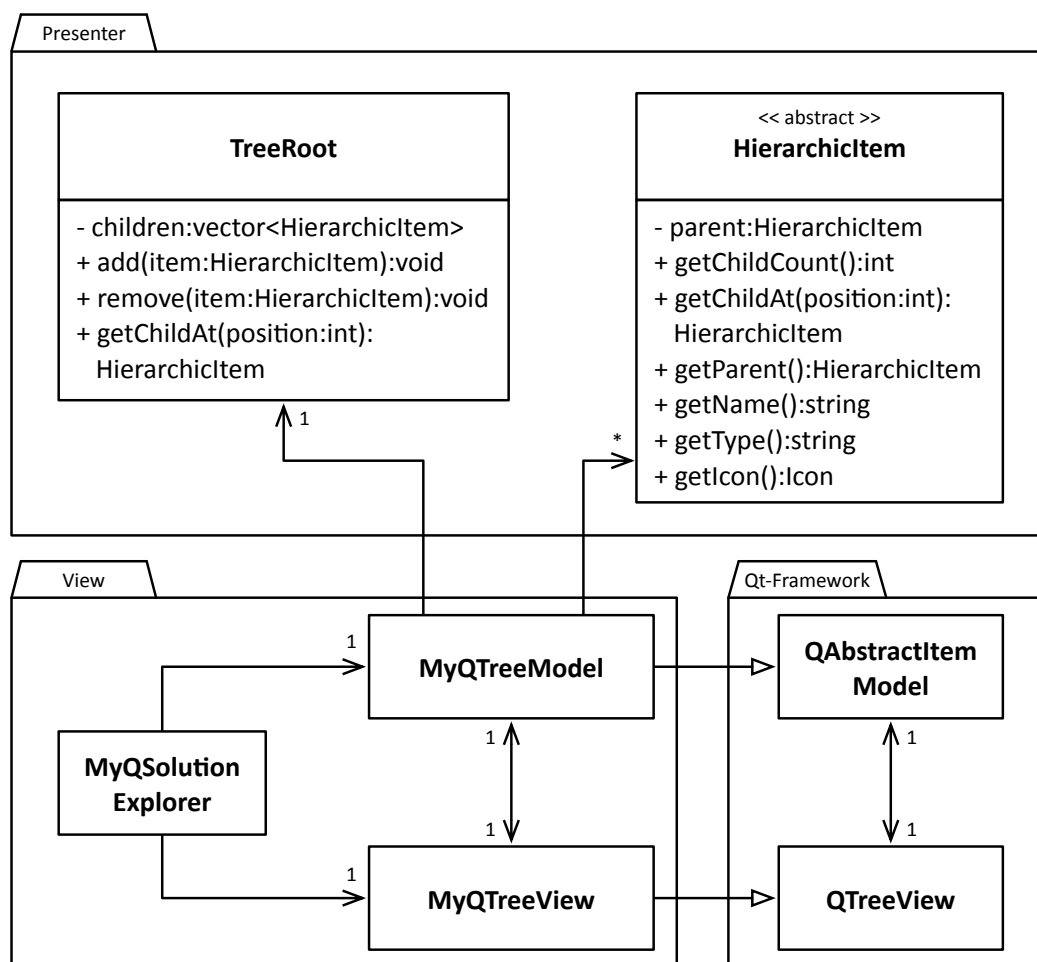


Abbildung 6.5: System zur Darstellung der Baumstruktur im Solution Explorer.

Die Fähigkeit der Items Instanzen der eigenen Klassendefinition als Kinder aufnehmen zu können, lässt unterschiedliche Hierarchieebenen entstehen. Zur Darstellung der Baumstruk-

tur wird ausgehend von der Wurzel durch die Level der Hierarchie iteriert. Zu jedem der `HierarchicItems` werden Name, Typ und Icon bereitgestellt. Die `MyQTreeView` kann dann die Daten an die entsprechenden Positionen der Explorer-Ansicht zeichnen. Die Daten der `HierarchicItems` können über die Methoden „getName“, „getType“ und „getIcon“ ermittelt werden. Um sich durch die Hierarchie zu bewegen, dienen die Methoden „getChildCount“, „getChildAt“ und „getParent“. Die Wurzel des `MyQTreeModels` wird nicht in der Baumstruktur dargestellt.

6.2.3 Aktualisierung der Baumstruktur

Bei Änderungen in der hierarchischen Struktur, die durch das Wurzelement des `MyQTreeModel`s repräsentiert wird, muss die Baumstruktur aktualisiert werden. Dies geschieht unter Verwendung eines Observer-Mechanismus innerhalb der Hierarchie nach Abbildung 6.6.

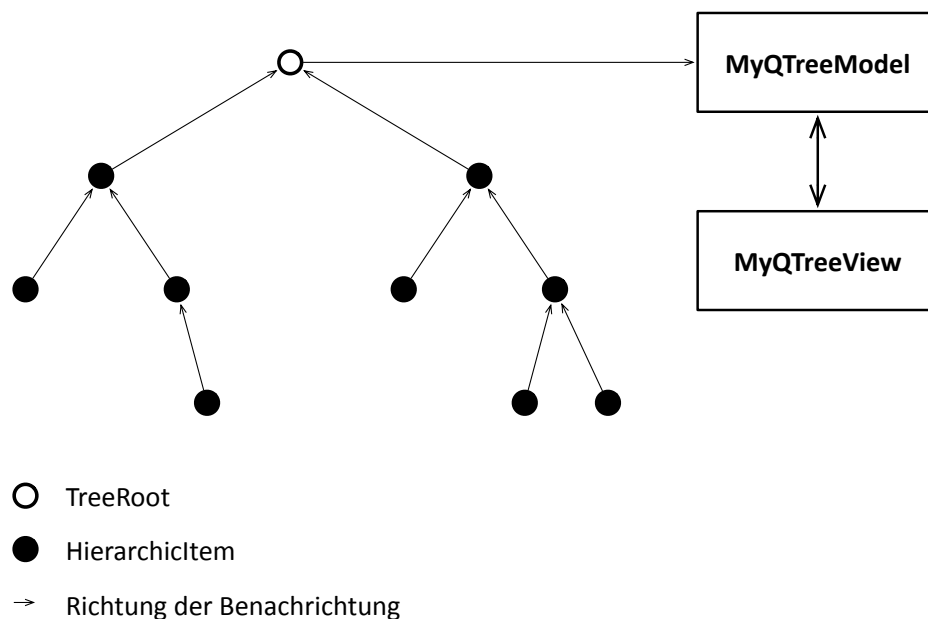


Abbildung 6.6: Aktualisierungsmechanismus der Baumstruktur im Solution Explorer.

Bei jedem `HierarchicItem`, das in die Struktur eingefügt wird, registriert sich das übergeordnete Element als Beobachter. Löst ein Item der Hierarchie den Aktualisierungsmechanismus aus, wird dadurch das Väterelement benachrichtigt. Von dort aus wird die Nachricht in Richtung des `TreeRoots` weitertransportiert. Gelangt sie in der Wurzel der Hierarchie an, informiert diese das `MyQTreeModel` darüber, dass die Baumstruktur einen veralteten Zustand

darstellt. Das `MyQTreeModel` löst einen Qt-internen Mechanismus aus, der die `MyQTreeView` zur Aktualisierung der Baumstruktur anregt.

6.2.4 Einfügen von Knoten in die Baumstruktur

Über Menüs in der Menüleiste des Hauptanwendungsfensters wird dem Benutzer die Möglichkeit gegeben die Daten des Modells zu manipulieren. Innerhalb des Menüs „Data“ befindet sich das Untermenü „New Element“, welches die unterschiedlichen Typen der `HierarchyNodes` auflistet. Mit Klick auf einen der Einträge wird der Baumstruktur ein entsprechender Eintrag hinzugefügt. Abbildung 6.7 zeigt die Klassenstruktur zur Umsetzung des „New Element“-Menüs.

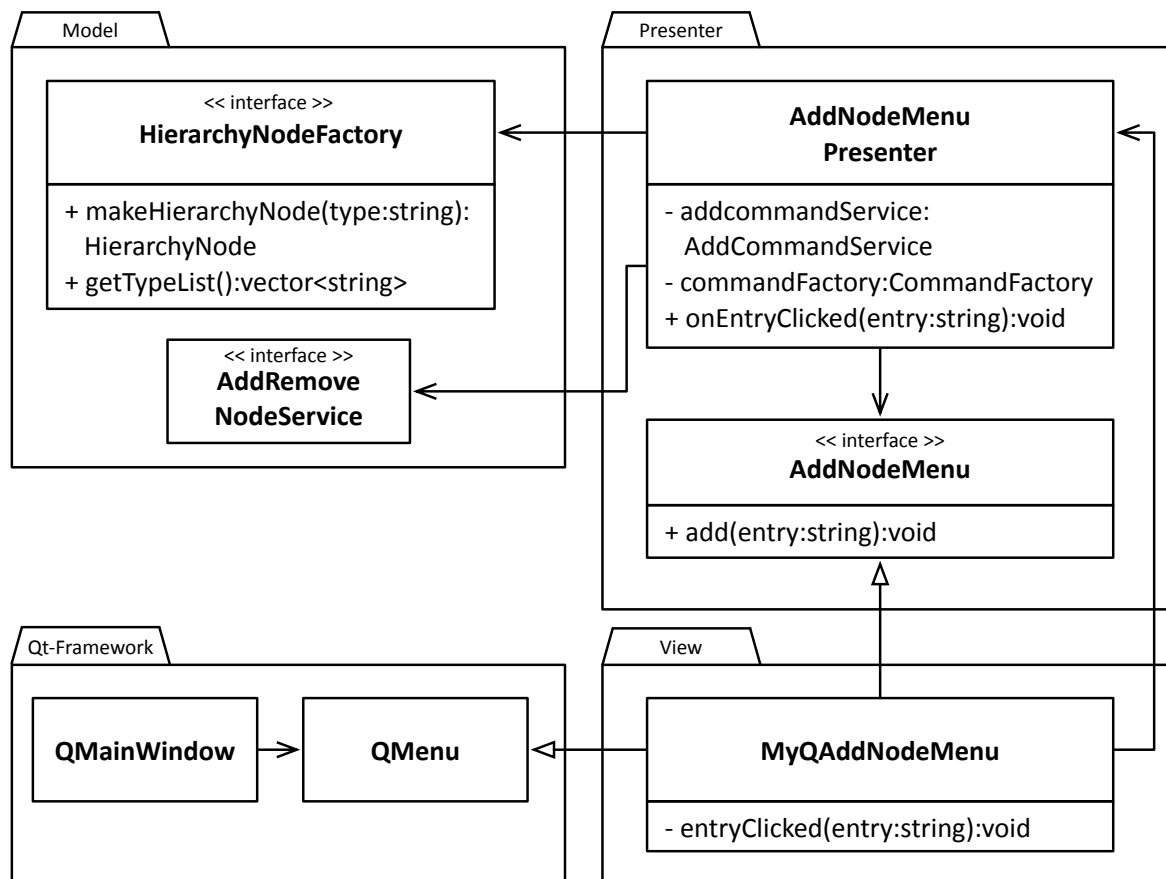


Abbildung 6.7: Umsetzung des AddNodeMenus zur Erstellung von Knoten in der Applikation.

Das Untermenü zum Hinzufügen von Einträgen in die Baumstruktur wird durch das `MyQAddNodeMenu` repräsentiert. Entsprechend des Passive-View-Musters wird die Klasse `AddNodeMenuPresenter` sowie die Schnittstelle `AddNodeMenu` eingeführt. Die Auf-

listung der Knotentypen ermittelt der Presenter über die „getTypeList“-Methode der HierarchyNodeFactory. Für jeden Typen dieser Liste fügt er dem MyQAddNodeMenu über die Schnittstelle AddNodeMenu einen Eintrag hinzu. Klickt der Benutzer auf einen der Menüeinträge wird die Methode „entryClicked“ der Viewklasse ausgelöst. Diese benachrichtigt den Presenter durch Aufruf der „onEntryClicked“-Methode und übergibt ihm den geklickten Eintrag. Mit dem erhaltenen string beauftragt der Presenter die HierarchyNodeFactory. Diese erzeugt einen neuen HierarchyNode des entsprechenden Subtyps und liefert ihn zurück. Das Hinzufügen des Knotens zur ProjectHierarchy erfolgt über das Command-Pattern. Unter Verwendung einer CommandFactory wird ein AddNodeCommand erzeugt und in der CommandQueue ausgeführt. Dadurch wird der HierarchyNode in die ProjectHierarchy eingefügt. Nach der Änderung der Modelldaten muss eine Aktualisierung der Baumstruktur erfolgen. Dies geschieht durch die Klassenstruktur nach Abbildung 6.8.

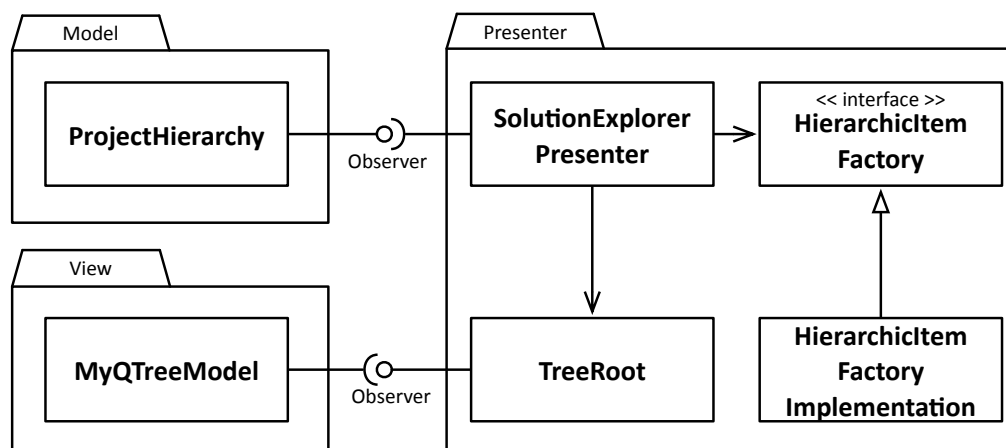


Abbildung 6.8: Observer-Mechanismus zwischen ProjectHierarchy, SolutionExplorerPresenter und MyQTreeModel.

Sobald Knoten hinzugefügt oder entfernt werden, erhält der SolutionExplorerPresenter über einen Observer-Mechanismus den entsprechenden HierarchyNode. Wurde der Knoten hinzugefügt, beauftragt er die HierarchyItemFactory ein passendes HierarchyItem zu erzeugen, das er anschließend dem TreeRoot hinzufügt. Wurde der Knoten aus der ProjectHierarchy entfernt, wird ebenfalls das HierarchyItem aus der Wurzel entfernt, das den Knoten referenziert. Das MyQTreeModel ist bei dem Wurzelement als Observer registriert und erhält somit Informationen über die veränderte hierarchische Struktur. Es löst die Aktualisierung der MyQTreeView aus, wodurch sich diese neu zeichnet.

6.2.5 Verbinden von Knoten in der Baumstruktur

Durch das Hinzufügen von `HierarchyNodes` zu der `ProjectHierarchy` entstehen in der Baumstruktur Items auf oberster Hierarchieebene. Um weitere Level der Hierarchie zu erzeugen, können Knoten miteinander verbunden werden. Dazu steht in dem „Data“-Menü das Untermenü „Bind To“ zur Verfügung. Um `HierarchyNodes` zu koppeln, muss der Benutzer in der Baumstruktur das Item auswählen, dessen Knoten er als Kind binden möchte. Das Menü zeigt alle existierenden `HierarchyNodes` an, bei denen der Knoten des ausgewählten Items als Kind gebunden werden kann. Durch Klick auf einen der Menüeinträge wird das Binden der Knoten veranlasst. Dabei nimmt der Knoten, den der Benutzer in dem Menü ausgewählt hat, den ausgewählten Knoten der Baumstruktur als Kind auf. In dem Solution Explorer wird dadurch ein neues Item sichtbar, das dem des Elternknotens untergeordnet ist.

Damit das „Bind To“-Menü eine korrekte Auflistung der möglichen Elternknoten zusammenstellen kann, muss jederzeit der ausgewählte `HierarchyNode` der Baumstruktur bekannt sein. Dieser kann analog zu Abbildung 6.9 ermittelt werden.

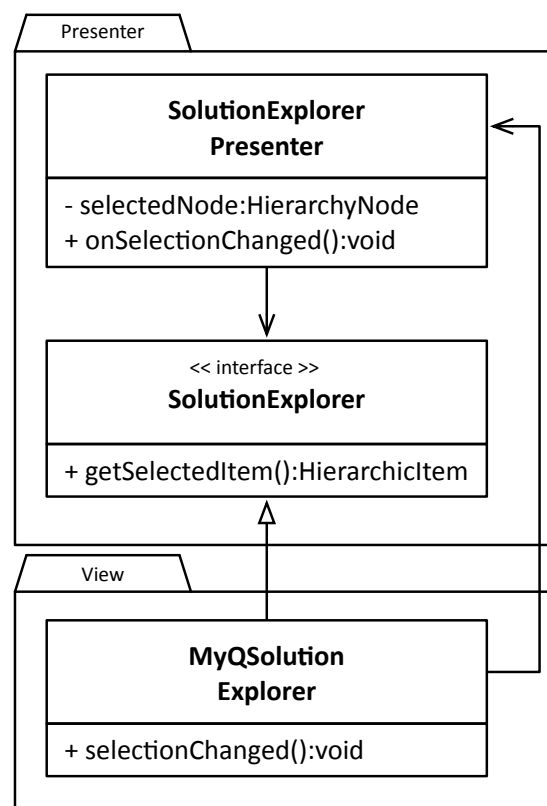


Abbildung 6.9: Referenzieren des ausgewählten `HierarchyNodes` durch den `SolutionExplorer-Presenter`.

Das Auswählen eines Items in der Baumstruktur löst die Methode „selectionChanged“ des `MyQSolutionExplorers` aus. Dadurch wird die „onSelectionChanged“-Methode des `SolutionExplorerPresenters` aufgerufen. Dieser erfragt mit Hilfe der „getSelectedItem“-Methode des `SolutionExplorerPresenters` aufgerufen. Dieser erfragt mit Hilfe der „getSelectedItem“-Methode das ausgewählte `HierarchyItem`. Anschließend extrahiert er den in dem Item befindlichen `HierarchyNode` und speichert ihn in einer Instanzvariablen ab. Die Zusammenstellung der Menüeinträge erfolgt nach Abbildung 6.10.

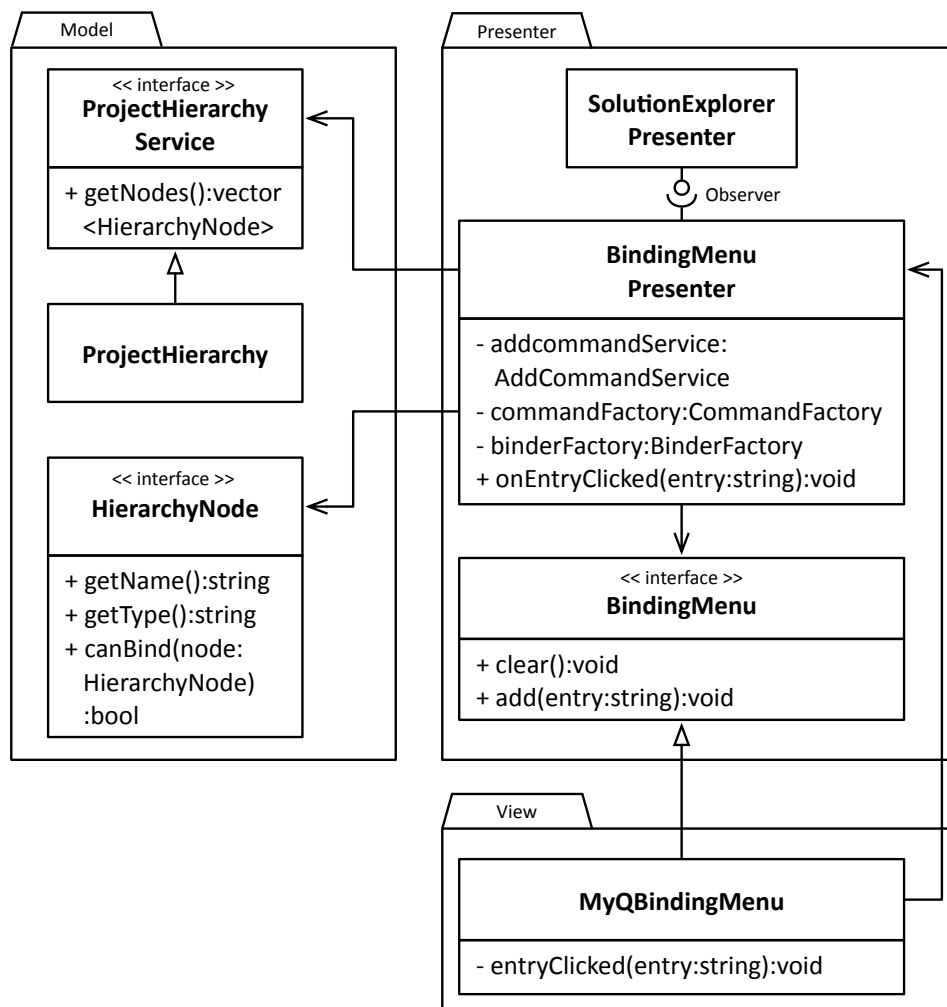


Abbildung 6.10: Speichern des ausgewählten `HierarchyNodes` in dem `SolutionExplorerPresenter`.

Durch einen Observer-Mechanismus wird der `BindingMenuPresenter` über eine neue Auswahl informiert. Um in das `MyQBindingMenu` die richtigen Einträge einzufügen, muss die Presenterklasse eine Liste von Knoten ermitteln, die den ausgewählten `HierarchyNode` aufnehmen können. Dazu benutzt er die Liste aller existierenden Knoten, die er über die Schnittstel-

le `ProjectHierarchyService` erhält. Unter Verwendung der „`canBind`“-Methode fragt der Presenter alle Elemente der Liste, ob sie den ausgewählten Knoten als Kind binden können. Ist dies der Fall, fügt er dem Menü einen Eintrag hinzu, der den Namen des Knotens aus der Liste enthält.

Ändert sich erneut die Auswahl in der Baumstruktur, wiederholt sich der Vorgang. Damit die Menüeinträge für die vorherige Auswahl aus dem Menü gelöscht werden, ruft der Presenter zunächst die „`clear`“-Methode auf.

Durch Klick auf einen Menüeintrag löst der Benutzer die Methode „`entryClicked`“ des `MyQ-BindingMenus` aus. Dadurch wird die Methode „`onEntryClicked`“ des Presenters aufgerufen. Unter Verwendung der `BinderFactory` veranlasst der Presenter die Erzeugung eines passenden Binders für den ausgewählten Knoten der Baumstruktur. Anschließend wird durch eine `CommandFactory` ein `BindingCommand` erstellt, dem als Parameter der Binder und der im Menü ausgewählte Elternknoten mitgegeben werden. Das Ausführen des Commands in der `CommandQueue` löst den *double dispatch*-Mechanismus des acyclic Visitor Patterns aus. Dadurch werden die `HierarchyNodes` gebunden.

Das Binden eines Kindes löst in dem Elternknoten den Observer-Mechanismus aus. Dadurch wird das entsprechende `HierarchicItem` der Baumstruktur benachrichtigt. Um die Berechnungshierarchie korrekt darzustellen, muss das benachrichtigte Item für den gebundenen Knoten ein `HierarchicItem` erstellen und es als Kind hinzufügen. Zur Erzeugung verwendet es die `HierarchicItemFactory`, die es als Instanzvariable hält. Nach Einfügen des erzeugten Items in die hierarchische Struktur, werden die in Abschnitt 6.2.3 beschriebenen Beziehungen verwendet, um eine Aktualisierung der Baumstruktur hervorzurufen.

6.3 Darstellung und Manipulation der Knoteneigenschaften

In diesem Abschnitt werden die Klassenbeziehungen zur Umsetzung der Eigenschaftstabelle im Properties Explorer beschrieben. Zuerst werden dabei die Zusammenstellung und die Anzeige der knotenspezifischen Daten thematisiert und gezeigt wie sich der Properties Explorer in Bezug auf den ausgewählten Knoten in der Baumstruktur aktualisiert. Im Anschluss wird die Umsetzung der Klassen zur Veränderung von Knoteneigenschaften erklärt.

6.3.1 Darstellung der Knoteneigenschaften

Im Properties Explorer werden zu jedem Zeitpunkt die Eigenschaften des ausgewählten Knotens angezeigt. Die Darstellung erfolgt in einer Tabelle, die auf der Qt-Klasse `QTableView` basiert. Für die Applikation CECLOUD wird die abgeleitete Klasse `MyQTableView` verwendet. Ähnlich wie bei der Baumstruktur ist die `MyQTableView` für die Visualisierung von Daten zuständig, die durch ein Objekt der Klasse `QAbstractItemModel` bereitgestellt werden. Für die Implementierung der Eigenschaftstabelle wird die Klasse `MyQTableModel` von `QAbstractItemModel` abgeleitet. Zur Darstellung werden die Daten analog zur Baumstruktur ausgehend von einem Wurzelement des `MyQTableModels` auf dem Bildschirm gezeichnet. Die Daten der Tabelle werden in `PropertiesItems` organisiert. Sie beinhalten den Namen sowie den aktuellen Wert der Eigenschaft in Form eines strings. Es ergeben sich Beziehungen der Klassen nach Abbildung 6.11.

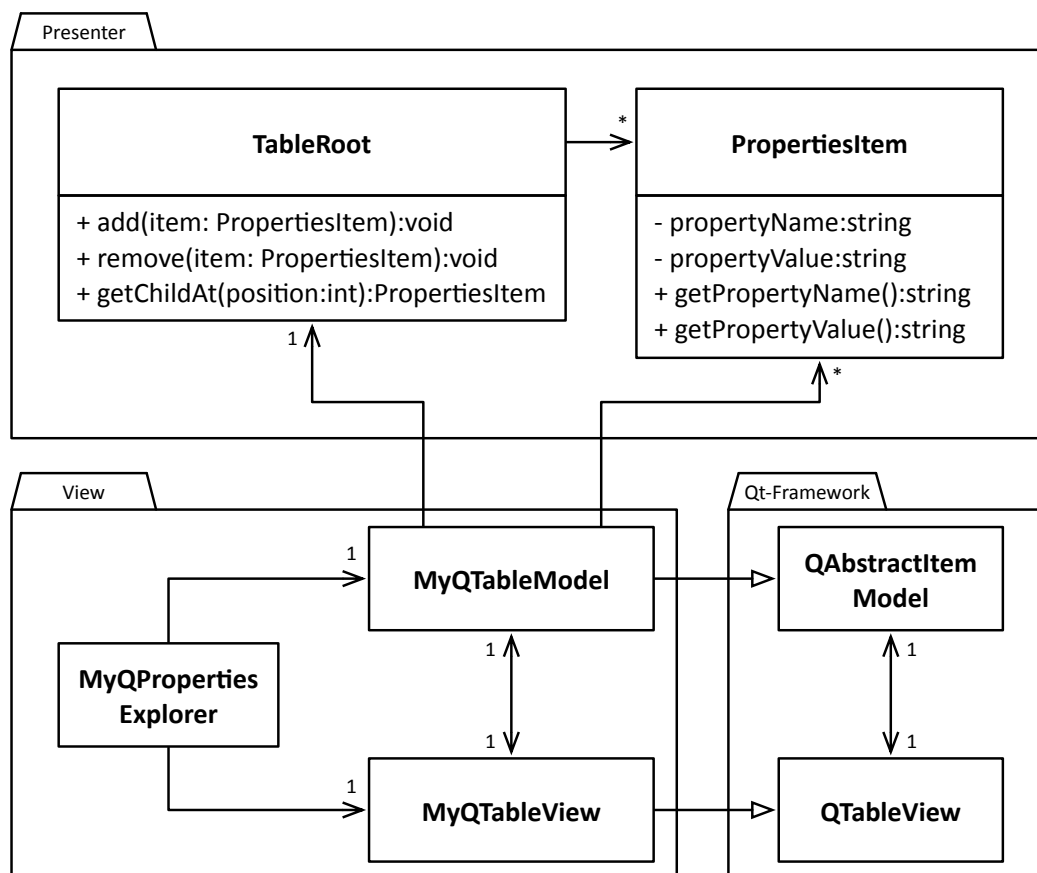


Abbildung 6.11: System zur Darstellung der Eigenschaften in dem Properties Explorer.

Hinter dem `TableRoot` verbirgt sich die Datenstruktur der Tabelle. Dieser enthält eine Liste von `PropertiesItems`, in welcher die Daten der Knoten enthalten sind. Da die unterschiedlichen Subtypen von `HierarchyNode` verschiedene Eigenschaften besitzen, müssen Wurzelemente erzeugt werden, die entsprechend des Knotentyps aufgebaut sind. Das Erstellen der individuellen `TableRoots` wird von Klassen der `TableRootBuilder`-Hierarchie übernommen. Hinter der abstrakten Schnittstelle verbirgt sich eine parallele Klassenhierarchie, die für jeden Knotentypen ein Derivat aufweist. Abbildung 6.12 zeigt die parallelen Klassensstrukturen.

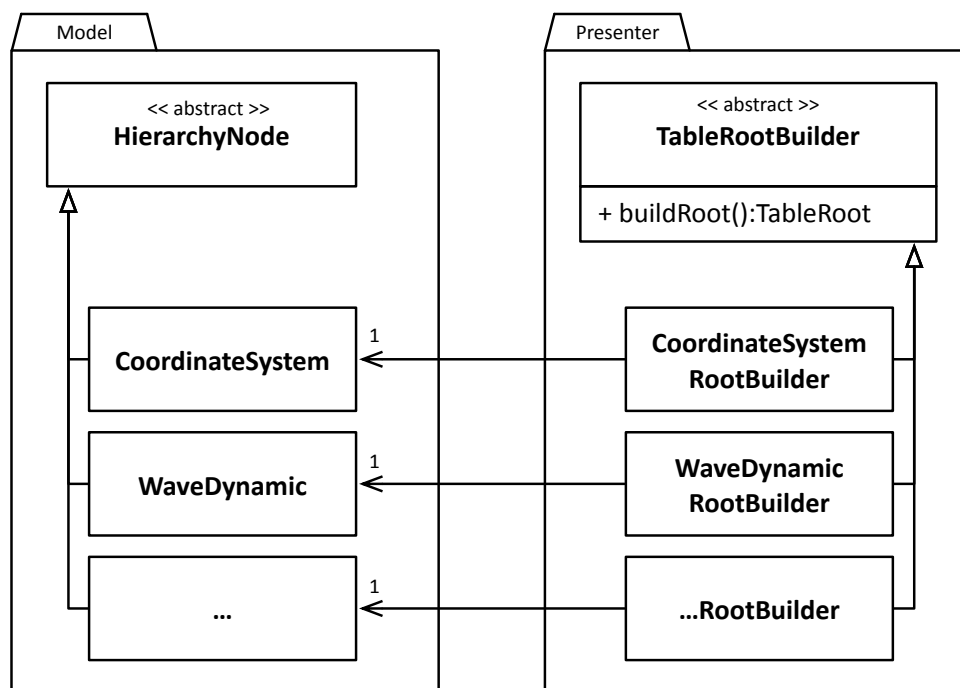


Abbildung 6.12: Parallele Klassenhierarchien der `HierarchyNodes` und der `TableRootBuilder`.

Listing 6.3 und 6.4 verdeutlichen die Umsetzung der Klassenhierarchie zur Erzeugung der entsprechenden `TableRoots`. In der Methode „`buildRoot`“ werden dem Wurzelement zunächst die allgemeinen Informationen des Knotens hinzugefügt. Dazu gehören der Name und der Typ. Da diese Operationen bei jedem `HierarchyNode` durchgeführt werden, befindet sich die Logik dafür zur Vermeidung von Redundanzen in der abstrakten Klasse `TableRootBuilder`. Anschließend fügt der `CoordinateSystemRootBuilder` die individuellen Knoteneigenschaften unter Verwendung der privaten Methoden „`addX`“, „`addT`“ und „`addZ`“ ein. Für die Erzeugung der `PropertiesItems` werden der Name und der Wert der Eigenschaft als `string` benötigt. Die Konvertierung erfolgt mit Hilfe der Operation „`std::to_string`“.

```
class TableRootBuilder{
2   protected:
    PropertiesItemFactory* itemFactory;
4   TableRoot* tableRoot;

6   public:
    virtual TableRoot* buildRoot() = 0
8   protected:
    PropertiesItem* getNewItem(string propertyName, string propertyValue){
10    return itemFactory->makePropertiesItem(propertyName, propertyValue);
    }
12   virtual void addNodeInfo(HierarchyNode* node){
        tableRoot->add(itemFactory->makePropertiesItem("Name", node->getName()));
14    tableRoot->add(itemFactory->makePropertiesItem("Type", node->getType()));
    }
16 }
```

Listing 6.3: Quelltext der Klasse TableRootBuilder.

```
class CoordinateSystemRootBuilders : public TableRootBuilder{
2   private:
    CoordinateSystem* coordSystem;
4
6   public:
    TableRoot* buildRoot(){
        addNodeInfo(coordSystem); //aus abstrakter Basisklasse
8        addX();
        addT();
10       addZ();
        return tableRoot;
12    }
    private: //analog auch die Methoden addT, addZ,...
14   virtual void addX(){
        string xValue = std::to_string(coordSystem->getX());
16        PropertiesItem* item = itemFactory->makePropertiesItem("x", xValue);
        tableRoot->add(item);
18    }
}
```

Listing 6.4: Quelltext der Klasse CoordinateSystemRootBuilders.

Damit nach einer Änderung der Auswahl in der Baumstruktur eine entsprechende Eigenschaftstabelle im Properties Explorer angezeigt werden kann, registriert sich der PropertiesExplorerPresenter als Beobachter bei dem SolutionExplorerPresenter. Durch den Observer-Mechanismus wird er über die veränderte Auswahl informiert. Mit dem ausgewählten HierarchyNode beauftragt er die TableRootBuilderFactory einen entsprechenden TableRootBuilder zu erzeugen. Unter Verwendung der „buildRoot“-Methode veranlasst er die Zusammenstellung der Eigenschaftsdaten des ausgewählten HierarchyNodes, die durch das zurückgelieferte Wurzelement repräsentiert werden. Anschließend wird das erhaltene TableRoot über die „setter“-Methode des MyQPropertiesExplorer an die View übergeben. Von dort aus gelangt es als neue Wurzel in das MyQTableModel. Der Austausch des Wurzelements im MyQTableModel löst den Qt-internen Aktualisierungsprozess der MyQTableView aus. Dadurch wird die Eigenschaftstabelle auf Grundlage des neuen TableRoots gezeichnet. Abbildung 6.13 zeigt die Klassenbeziehungen zur Umsetzung der Aktualisierung des Properties Explorers nach Verändern der Auswahl im Solution Explorer.

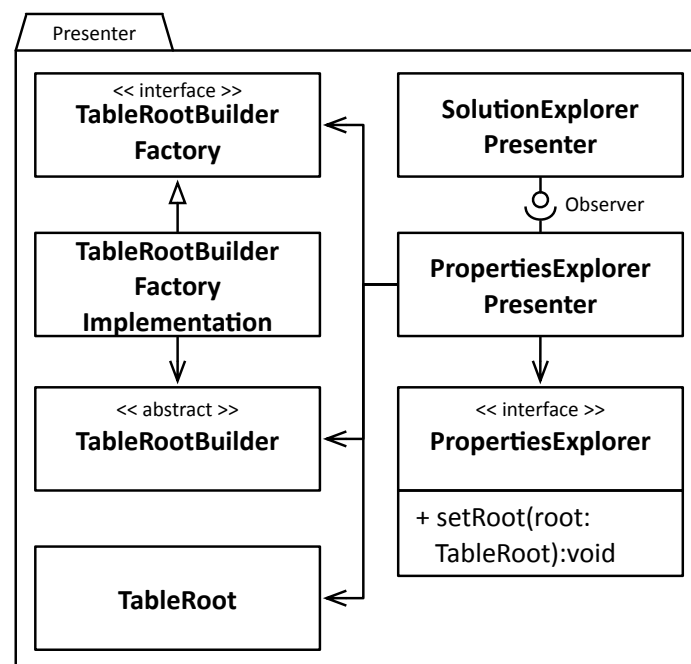


Abbildung 6.13: Erzeugung und Anzeige der knotenspezifischen Eigenschaftstabelle.

6.3.2 Verändern der Knoteneigenschaften

Neben der Anzeige der knotenspezifischen Eigenschaften bietet der Properties Explorer dem Benutzer die Möglichkeit, die Parameter der Knoten zu manipulieren. Jede Zeile der Tabelle besteht aus zwei Einträgen. In der linken Spalte befinden sich die Namen, in der rechten die Werte der Eigenschaften. Ein Klick in die rechte Spalte der Tabelle löst für den geklickten Eintrag den *Editor*-Modus aus. Es erscheint ein Textfeld, in dem der Benutzer den neuen Wert der Eigenschaft eingeben kann. Durch Bestätigen mit der *Enter*-Taste wird die Eingabe des Benutzers angenommen und die Methode „propertyEdited“ des `MyQPropertiesExplorers` aufgerufen. Dabei wird als Eigenschaftswert die Eingabe des Benutzers übergeben. Als Eigenschaftsname dient der Eintrag der Tabelle, der sich in der selben Zeile in der linken Spalte befindet. Zur Weiterverarbeitung der Benutzereingaben wird das Datenpaar durch den Aufruf der „onPropertyEdited“-Methode an den `PropertiesExplorerPresenter` übermittelt. Mit dem ausgewählten `HierarchyNode` veranlasst dieser die `EditorFactory` einen passenden Editor zu erzeugen. Unter Verwendung der `CommandFactory` erstellt der Presenter anschließend einen `EditCommand`, dem er als Parameter das Datenpaar und den Editor übergibt. Danach fügt er den Befehl der `CommandQueue` über die Schnittstelle `AddCommandService` hinzu. Durch die Ausführung des Befehls in der `CommandQueue` wird das Verändern der Eigenschaft ausgelöst. Abbildung 6.14 zeigt die Klassenbeziehungen durch die eine Manipulation der Berechnungsparameter erreicht wird.

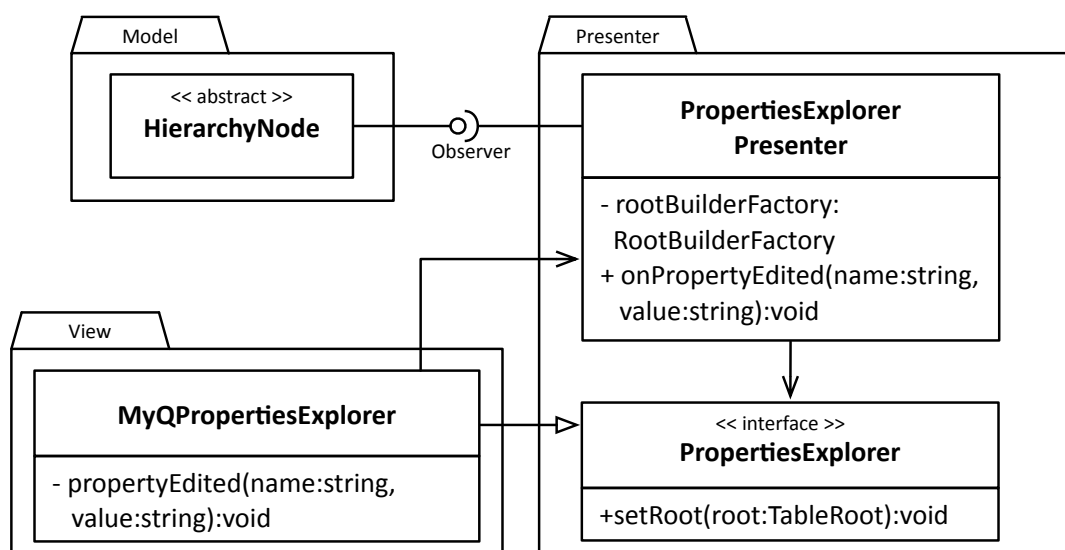


Abbildung 6.14: Klassenbeziehungen zur Manipulation der Knoteneigenschaften.

Aufgrund der Änderung der Knoteneigenschaften ist eine Aktualisierung der Tabelle notwendig. Damit der `PropertiesExplorerPresenter` Modifikationen der Knoteneigenschaften erfassen kann, registriert er sich bei dem ausgewählten `HierarchyNode` als Beobachter. Verändern sich die Daten des Knotens, löst dieser den Observer-Mechanismus aus, wodurch die Aktualisierung der Tabelle in Gang gesetzt wird. Dazu werden die in Abbildung 6.13 dargestellten Klassenbeziehungen verwendet.

6.4 Undo – Redo

Die Verwendung des Command Patterns erlaubt es dem Benutzer Handlungen, die als Befehl in die `CommandQueue` eingefügt wurden, zu widerrufen bzw. widerrufenen Befehle erneut auszuführen. Dafür steht in der Menüleiste das „Edit“-Menü zur Verfügung, welches die Einträge „Undo“ und „Redo“ beinhaltet. Das Auslösen der Befehle kann nach Abbildung 6.15 erfolgen.

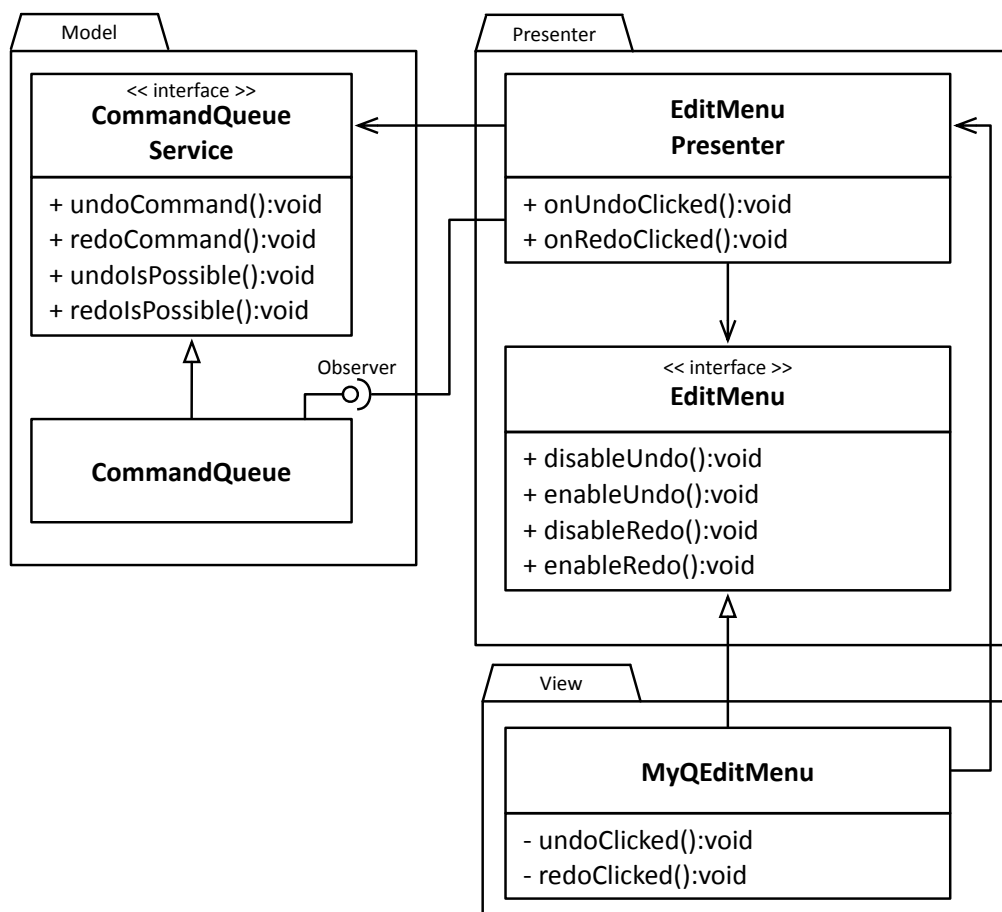


Abbildung 6.15: Umsetzung des EditMenus zur Verwendung der Bearbeitungshistorie.

Durch Klick auf die Einträge im Menü werden die privaten Methoden des `MyQEditMenus` ausgelöst. Über die Methoden „`onUndoClicked`“ und „`onRedoClicked`“ wird der `EditMenuPresenter` informiert. Er verwendet die Methoden „`undoCommand`“ und „`redoCommand`“ der `CommandQueue` über die Schnittstelle `CommandQueueService`, um entsprechend des geklickten Eintrags einen Befehl zu widerrufen oder erneut auszuführen.

Es ist zu beachten, dass die beiden Funktionen nicht zu jedem Zeitpunkt durchgeführt werden können. Befinden sich in der `CommandQueue` keine Befehle oder nur Befehle, die widerrufen wurden, muss die „Undo“-Funktion deaktiviert werden. Gleiches gilt für die „Redo“-Funktion, wenn keine Handlung existiert, die zuvor mittels „Undo“ rückgängig gemacht wurde. Mithilfe der Methoden „`undolsPossible`“ und „`redolsPossible`“ kann der Presenter evaluieren, welchen Eintrag des Menüs er aktivieren oder deaktivieren muss. Dafür stehen die Methoden „`enable`“ und „`disable`“ der Schnittstelle `EditMenu` zur Verfügung. Die Aktualisierung des Menüs erfolgt über das Observer-Muster. Die `CommandQueue` benachrichtigt den `EditMenuPresenter`, sobald sich ihr Zustands ändert. Dieser kann anschließend die Einträge des Menüs an den Zustand der `CommandQueue` anpassen.

6.5 Auswertung der Berechnungshierarchien

Zu jedem Knotentyp, der Berechnungsanweisungen beinhaltet, existieren Klassen, welche die Darstellung der Ergebnisse auf dem Bildschirm übernehmen. Dabei handelt es sich um Auswertungsfenster, die in den linken Bereich des Hauptanwendungsfensters integriert werden. In diesem Abschnitt wird die Erzeugung und die Anzeige der Auswertungsfenster sowie deren Aktualisierung nach Änderung der Modelldaten erläutert.

6.5.1 Erzeugung der Auswertungsfenster

Die Auswertungsfenster sind von der Qt-Klasse `QDockWidget` abgeleitet. Dadurch können sie unter Ausnutzung der Qt-Funktionalität in dem Hauptanwendungsfenster, dass durch die `MyQShell` umgesetzt wird, dargestellt werden. Jede Klassendefinition, die ein Auswertungsfenster repräsentiert, ist für eine Art der Visualisierung von Ergebnissen zuständig. Dabei kann es vorkommen, dass zu einem Knotentypen mehrere Darstellungsarten vorhanden sind. Mit der LWT nach Airy und Laplace können basierend auf den Berechnungsparametern z.B. das Wellenprofil, die Wellenkinematik, der Wellendruck und die Wellenenergie berechnet werden.

Aus diesem Grund liegen für den HierarchyNode vom Subtyp WaveDynamic die Auswertungsfenster MyQWaveProfileView, MyQWaveKinematicsView, MyQWavePressureView und MyQWaveEnergyView vor. Die Erzeugung der Auswertungsfenster erfolgt unter Anwendung des Fabrik-Musters nach Abbildung 6.16.

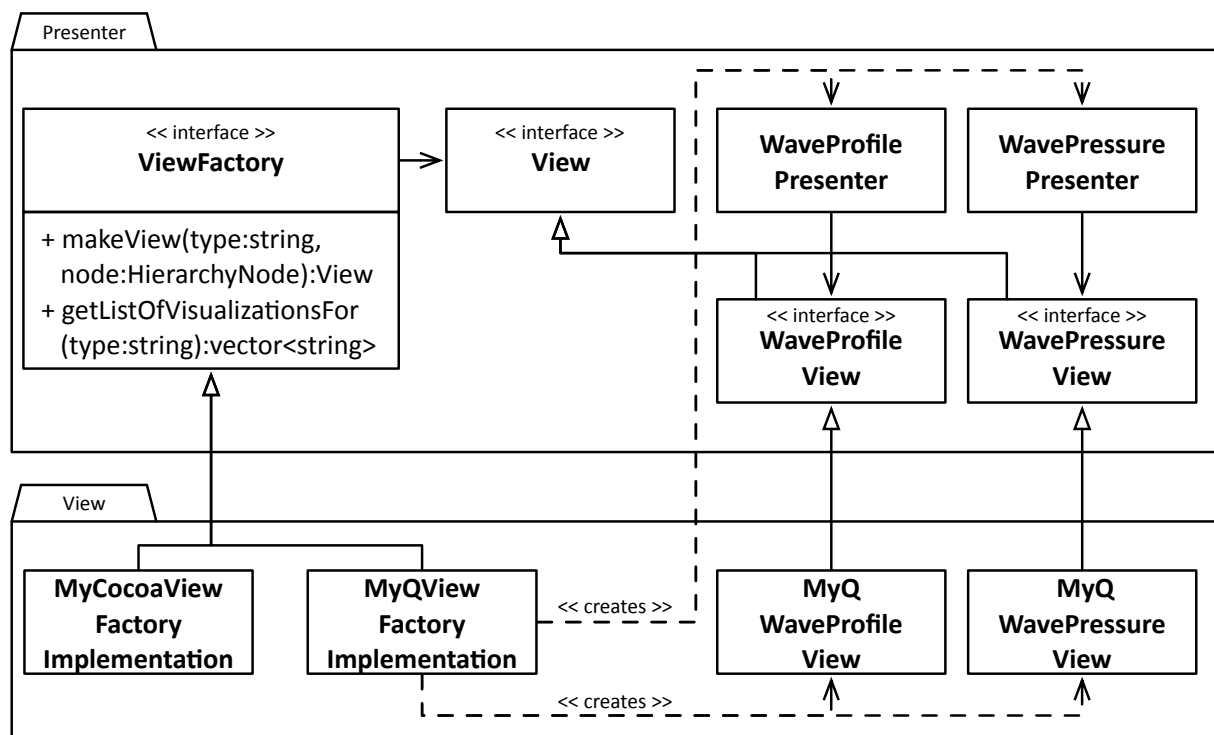


Abbildung 6.16: Erzeugung der Views und der Presenter durch die ViewFactory.

Für die Erzeugung eines Auswertungsfensters benötigt die MyQViewFactoryImplementation den auszuwertenden HierarchyNode. Damit sie aus den verschiedenen Darstellungsarten eines Knotentyps die richtige erstellen kann, benötigt sie des Weiteren den Typ der gewünschten Visualisierung. Wird ihr z.B. der string „Wave Profile“ übergeben, gibt sie ein View-Objekt vom Subtyp MyQWaveProfileView zurück.

Welche Darstellungstypen die ViewFactory für die einzelnen Derivate von HierarchyNode erstellen kann, lässt sich über die „getListOfVisualizationsFor“-Methode ermitteln. Der Fabrik wird ein Knotentyp in Form eines strings übergeben. Diese stellt eine Liste aller möglichen Auswertungstypen zusammen, die sie zur Darstellung erzeugen kann.

Gemäß des Passive-View-Musters wird zu jedem der Auswertungsfenster eine Presenterklasse benötigt, welche die Schnittstelle zu den Modellklassen bildet. Damit die Presenterklassen unabhängig von dem verwendeten Darstellungsframework sind, werden Schnittstellen

eingeführt. Im Fall der `MyQWaveProfileView` entstehen somit die Klassendefinitionen `WaveProfilePresenter` sowie `WaveProfileView`. Innerhalb der Fabrik-Methode werden sowohl die Auswertungsfenster als auch die Presenter-Objekte erzeugt. Listing 6.5 zeigt anhand eines Ausschnitts der Fabrik-Methode der `MyQViewFactoryImplementation` die Erzeugung der Auswertungsfenster sowie der Presenter.

```
1 class MyQViewFactoryImplementation : public ViewFactory{
  //...
3 public:
  View* makeView(string type, HierarchyNode* node){
5    //...
    if(type == "Wave Profile"){
7      WaveProfileView* view = new WaveProfileView();
      WaveProfilePresenter* presenter =
9        new WaveProfilePresenter((WaveDynamic*)node, view);
      view->setPresenter(presenter);
11
      return view;
13    }
    //...
15    else throw UnknownViewTypeException();
    }
17 }
```

Listing 6.5: Ausschnitt der Fabrik-Methode der `MyQViewFactoryImplementation`.

Zunächst wird eine Instanz der Viewklasse erstellt. Anschließend wird der passende Presenter erzeugt, dem der auszuwertende `HierarchyNode` und die View als Parameter übergeben werden. Danach wird der Presenter unter Verwendung einer „setter“-Methode bei dem View-Objekt gesetzt. Zuletzt gibt die Fabrik die View zurück. Das Presenter-Objekt kann nun die Berechnungsergebnisse des auszuwertenden Knotens ermitteln und zur Darstellung an die Viewklasse delegieren.

Um einen Austausch des Visualisierungsframeworks zu ermöglichen, muss der Presenter-Schicht die Verwendung einer Qt-unabhängigen Fabrik ermöglicht werden. Dafür wird die Schnittstelle `ViewFactory` eingeführt. Soll statt Qt das Framework Cocoa verwendet werden, kann die `MyQViewFactoryImplementation` durch die `MyCocoaViewFactoryImplementation` ersetzt, ohne dass sich Änderungen in der Presenter-Schicht ergeben.

6.5.2 Anzeigen der Auswertungsfenster über die Menüleiste

Damit die Auswertungsfenster in der Anwendung sichtbar sind, müssen sie dem Hauptanwendungsfenster hinzugefügt werden. Dazu steht dem Benutzer in dem „Data“-Menü das Untermenü „Evaluation“ zur Verfügung. In Abhängigkeit zu dem ausgewählten Knotentyp in der Baumstruktur des Solution Explorers enthält das „Evaluation“-Menü Einträge mit den Typen der unterschiedlichen Darstellungsarten. Durch Klick auf einen der Einträge wird das entsprechende Auswertungsfenster erstellt und in der Anwendung angezeigt. Diese Funktionalität kann durch die Klassenstruktur nach Abbildung 6.17 erreicht werden.

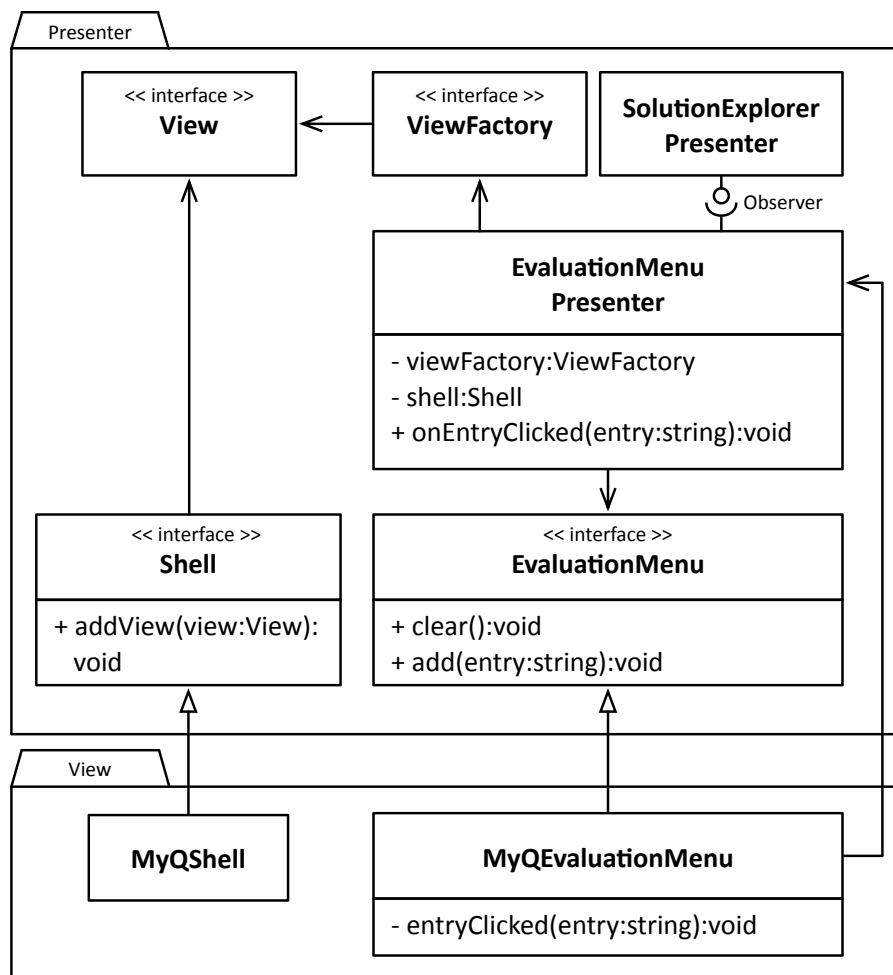


Abbildung 6.17: Aktualisierung des EvaluationMenus durch die Observierung des SolutionExplorerPresenters.

Ändert sich der ausgewählte Knoten in dem Solution Explorer, wird der EvaluationMenuPresenter zur Aktualisierung des MyQEvaluationMenus angeregt. Zunächst löscht

der Presenter alle vorherigen Einträge des Menüs unter Verwendung der „clear“-Methode. Anschließend ermittelt er über die „getListFor“-Methode der ViewFactory eine Liste aller Darstellungstypen für den ausgewählten Knoten. Der ViewFactory wird dafür der Typ des HierarchyNodes übergeben. Im letzten Schritt fügt der Presenter für jeden string der erhaltenen Liste einen Eintrag in das Menü ein.

Klickt der Benutzer auf einen der Einträge wird die private Methode „entryClicked“ des Menüs aufgerufen. Das MyQEvaluationMenu informiert den Presenter, welcher die Erzeugung des Auswertungsfensters veranlasst. Der Presenter löst die Fabrik-Methode der ViewFactory aus und übergibt den ausgewählten Knoten sowie den geklickten Eintrag. Nach der Erstellung durch die Fabrik fügt er das View-Objekt in die MyQShell ein. Nun wird das Auswertungsfenster in der Anwendung angezeigt. Damit der EvaluationMenuPresenter nicht von der Qt-basierten Klasse MyQShell abhängt, wird die Schnittstelle Shell zwischengeschaltet.

6.5.3 Aktualisierung der Auswertungsfenster

Der auszuwertende HierarchyNode repräsentiert eine Berechnungshierarchie, die sich aus Knoten auf mehreren Ebenen zusammensetzt. In die Berechnung fließen die Parameter aller HierarchyNodes ein, die sich in dieser Hierarchie befinden. Nach der Änderung eines Parameters muss die Auswertung erneut erfolgen. Um auf Änderungen der Daten reagieren zu können, wird der Presenter des Auswertungsfensters bei dem auszuwertenden Knoten als Beobachter registriert.

Zur Aktualisierung der Auswertung wird eine ähnliche Struktur verwendet wie bei der Baumstruktur nach Abschnitt 6.2.3. Die HierarchyNodes sind bei ihren Kindern als Beobachter registriert. Sobald sich Änderungen ergeben wird der Observer-Mechanismus ausgelöst. Die Benachrichtigung wird zu den übergeordneten Knoten weitertransportiert, bis sie in dem auszuwertenden Knoten angelangt ist. Dieser benachrichtigt den Presenter des Auswertungsfensters, wodurch die Berechnung erneut ausgeführt wird.

KAPITEL 7

Erweiterung der Applikation CECloud

In diesem Kapitel wird die Erweiterung CECloud um neue Knotenstrukturen gezeigt. Als Beispiel dient dafür die in Abbildung 3.2 erläuterte Theorie zur Berechnung des Sedimenttransports (s. Seite 30).

Zunächst wird die Applikation im Bezug auf ihre Beständigkeit in die zwei Bestandteile *abstrakter Basiskern* und *volatiler Applikationsbereich* untergliedert. Im Anschluss wird Schritt für Schritt erläutert welche Änderungen im volatilen Bereich der Applikation stattfinden müssen, um neue Knotenstrukturen in die Applikation zu integrieren. Dabei wird als erstes gezeigt welche Klassen im Modell hinzugefügt oder modifiziert werden müssen. Danach werden die Änderungen in der Präsentationsschicht erklärt, durch welche die neuen Knotenstrukturen sowie deren Auswertungen in der Applikation visualisiert werden können.

7.1 Die Beständigkeit der Applikationsbereiche

Die Anwendung der Prinzipien und Muster des OOD lässt Software entstehen, die hinsichtlich ihrer Beständigkeit in zwei Abschnitte untergliedert werden kann (McConnell, 2009). Der *Basiskern* einer Applikation beruht auf Abstraktionen. Konkrete Klassen und Implementierungen werden verborgen, wodurch Abhängigkeiten zur Detailebene vermieden werden. Die Schnittstellen werden zur Laufzeit durch Objekte konkreter Subklassen repräsentiert, deren Typen im Kontext des Kerns nicht bekannt sind. Somit gewährleistet der abstrakte Basiskern die applikationsspezifische Funktionalität, losgelöst von der Implementierung konkreter Klassen. Der *volatile Bestandteil* einer Applikation beinhaltet die Detailebene. Hier werden die konkreten Subtypen und deren Operationen verwendet. Die Modifikation einer Subklasse oder die Erweiterung einer Hierarchie um ein neues Derivat wirken sich somit unmittelbar auf den volatilen Bestandteil aus. Durch die Verwendung von abstrakten Schnittstellen ist der Basiskern vor Anpassungen geschützt, die aus Veränderungen in dem volatilen Bereich resultieren. Durch den Entwurfsprozess kann mit Hilfe der Methoden und Prinzipien des OOD wesentlich Einfluss auf die Größe der beiden Bereiche genommen werden. Für flexible Systeme ist der Basiskern möglichst groß zu gestalten, sodass Erweiterungen der Software unkompliziert sind.

Der Basiskern der Applikation CECLOUD

Auch CECLOUD kann in diese beiden Bestandteile untergliedert werden. Abbildung 7.1 zeigt den Basiskern der Applikation zusammen mit seinen wichtigsten Schnittstellen.

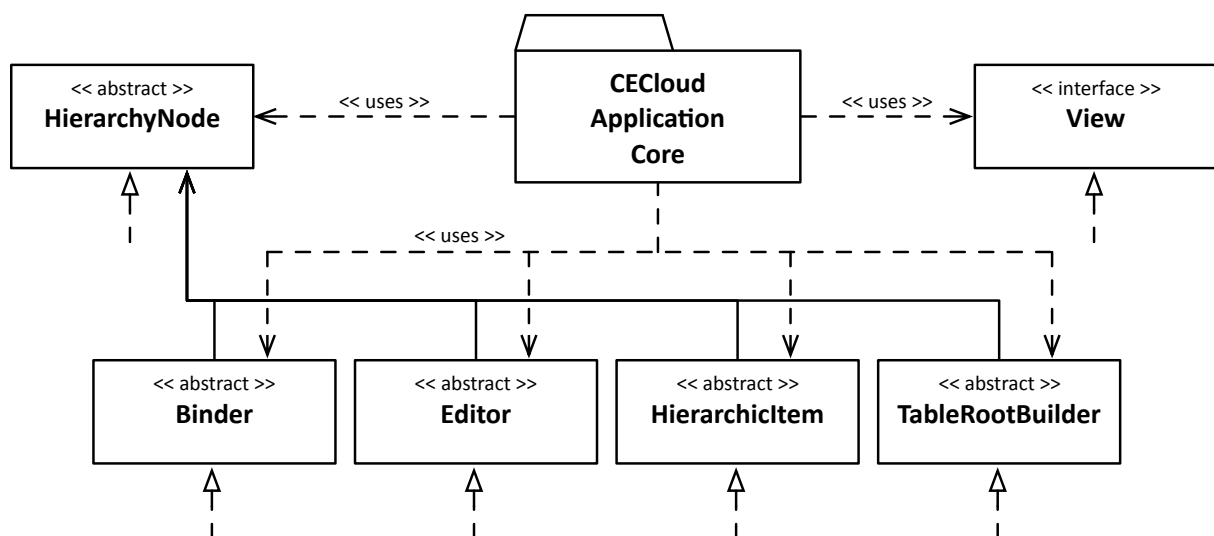


Abbildung 7.1: Darstellung des abstrakten Basiskerns der Applikation CECLOUD.

Hinter CECloud Application Core verbirgt sich eine komplexe Klassenstruktur, welche die gesamte Funktionalität der Applikation zur Verfügung stellt. Darunter befinden sich die Strukturen zum Verbinden und Verändern von Knoten, die Anzeige und Aktualisierung der Explorer-Ansichten sowie die Darstellung der Auswertungen und der Menüflächen im Hauptanwendungsfenster. Die Funktionalität basiert dabei auf den abstrakten Schnittstellen `HierarchyNode`, `Binder`, `Editor`, `HierarchicItem`, `TableRootBuilder` und `View`. Zur Erzeugung von Objekten existiert zu jeder der Schnittstellen eine Fabrik, die wiederum durch Abstraktionen vor dem Kern der Applikation verborgen werden.

Der volatile Bestandteil der Applikation CECloud

Der volatile Bereich besteht aus den Derivaten der abstrakten Schnittstellen. Im Kontext der Applikation sind die Subtypen der parallelen Klassenhierarchien, die Fabriken für deren Erzeugung sowie die View- und Presenterklassen zu deren Visualisierung von großer Bedeutung. Durch Verwendung von Polymorphie kann durch sie eine Erweiterung der Applikation erreicht werden, ohne dass dadurch Anpassungen im Kern stattfinden müssen. Für das Einarbeiten neuer Berechnungstheorien müssen demnach nur Änderungen in dem volatilen Bereich der Applikation erfolgen.

7.2 Erweiterung des Modells um neue Knotentypen

Gegenstand dieses Abschnitts ist die Erweiterung des Modells um die Strukturen zur Berechnung des Sedimenttransports. Zunächst wird das hierarchische Klassengefüge umgesetzt, durch das die Berechnungshierarchie beschrieben werden kann. Im Anschluss wird gezeigt, welche Schritte notwendig sind, damit der Binding-Mechanismus und die Veränderung der Knoteneigenschaften durch Editor-Klassen für die neuen Klassendefinitionen ermöglicht wird.

7.2.1 Die Berechnungshierarchie als Klassenstruktur

Die Berechnung des Sedimenttransports nach Abbildung 3.2 hängt von mehreren Variablen ab. Darunter befinden sich neben den Ergebnissen der LWT auch die Parameter des Sediments und des Wassers. Das Sediment wird dabei durch seine Dichte charakterisiert. Als Klassenstruktur wird die Berechnungshierarchie analog zu Abbildung 7.2 umgesetzt.

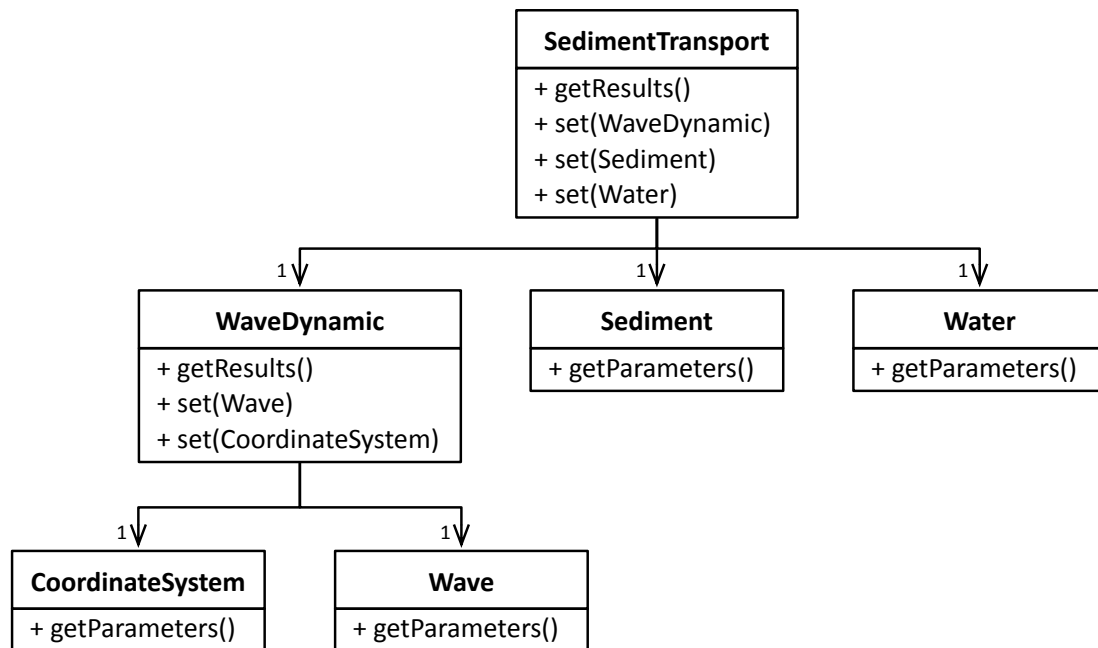


Abbildung 7.2: Umsetzung der Berechnungshierarchie aus Abbildung 3.2 als Klassenstruktur.

Bei `CoordinateSystem`, `Wave`, `WaveDynamic` und `Water` kann auf bereits implementierte Klassen zurückgegriffen werden. Die Umsetzung des Systems erfordert die Erweiterung der Applikation um die Klassen `Sediment` und `SedimentTransport`. Die Berechnung erfolgt über eine Formel, die in der Klasse `SedimentTransport` implementiert ist. Durch „getter“-Methoden können auf die Parameter der Kindknoten zugegriffen werden, die anschließend in die Berechnung einfließen.

Damit Objekte vom Typ der neuen Klassen sich im Kontext der Applikation verwenden lassen, ist die Ableitung von der abstrakten Schnittstelle `HierarchyNode` erforderlich. Des Weiteren muss die `HierarchyNodeFactoryImplementation` um die Erzeugung der neuen Subtypen von `HierarchyNode` erweitert werden. Es ergibt sich eine Klassenstruktur nach Abbildung 7.3.

Beim Ausführen des Programms wird zunächst innerhalb des Kerns der Applikation die GUI initialisiert. Dazu gehört ebenfalls das `AddNodeMenu`, in dem sich alle Subtypen der Schnittstelle `HierarchyNode` befinden, die durch die Fabrik erzeugt werden können. Wird die Liste, welche die „getTypeList“-Methode zurückgibt, um die Subtypen der neuen Klassen erweitert, werden sie während der Initialisierung in das Menü aufgenommen. Klickt der Benutzer auf die Einträge des Menüs werden diese an die Fabrik über die „makeHierarchyNode“-Methode weitergereicht. Zur Erzeugung der neuen Derivate ist die Fabrik nach Listing 7.1 zu erweitern.

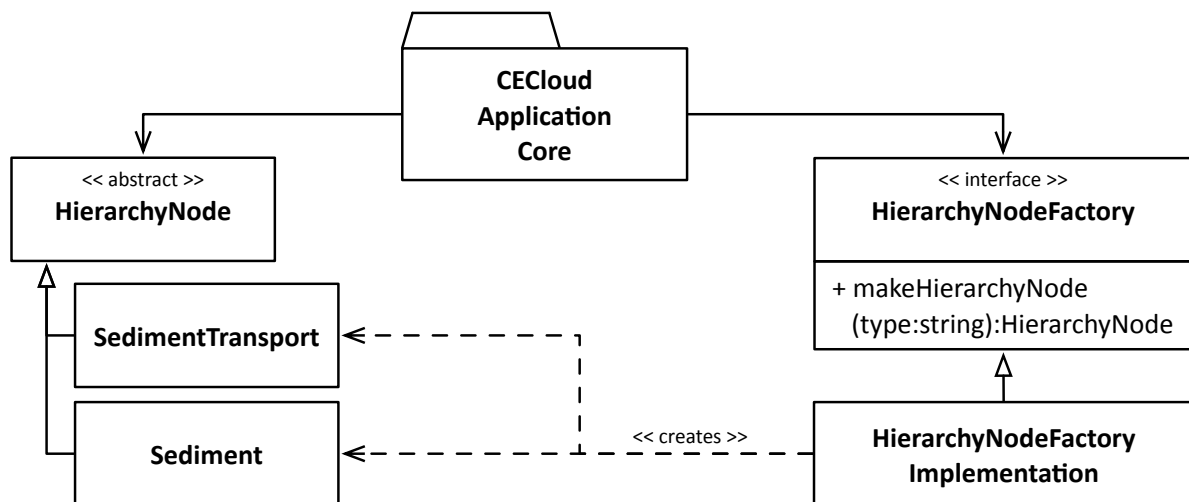


Abbildung 7.3: Erweiterung der HierarchyNode-Hierarchie.

```

1 class HierarchyNodeFactoryImplementation : public HierarchyNodeFactory{
2     public:
3         //Konstruktor...
4         virtual HierarchyNode* makeHierarchyNode(string type){
5             //...
6             if((type == "Sediment") return new Sediment();
7             if((type == "SedimentTransport") return new SedimentTransport());
8             //...
9         }
10        virtual vector<string> getTypeList(){
11            vector<string> types;
12            //...
13            types.push_back("Sediment");
14            types.push_back("SedimentTransport");
15
16            return types;
17        }
18    }
19 }

```

Listing 7.1: Modifikation der `HierarchyNodeFactoryImplementation`.

7.2.2 Erweiterung des Besucher-Musters zur Verbindung von Knoten

Die Auswertung des SedimentTransports kann nur erfolgen, wenn jede Position der Berechnungshierarchie besetzt ist. Dafür müssen Objekte der Typen Sediment, WaveDynamic

und Water bei dem SedimentTransport als Kindknoten gesetzt werden. Die Umsetzung dieser Funktionalität kann durch die Erweiterung des azyklischen Besucher-Musters analog zu Abbildung 7.4 erreicht werden.

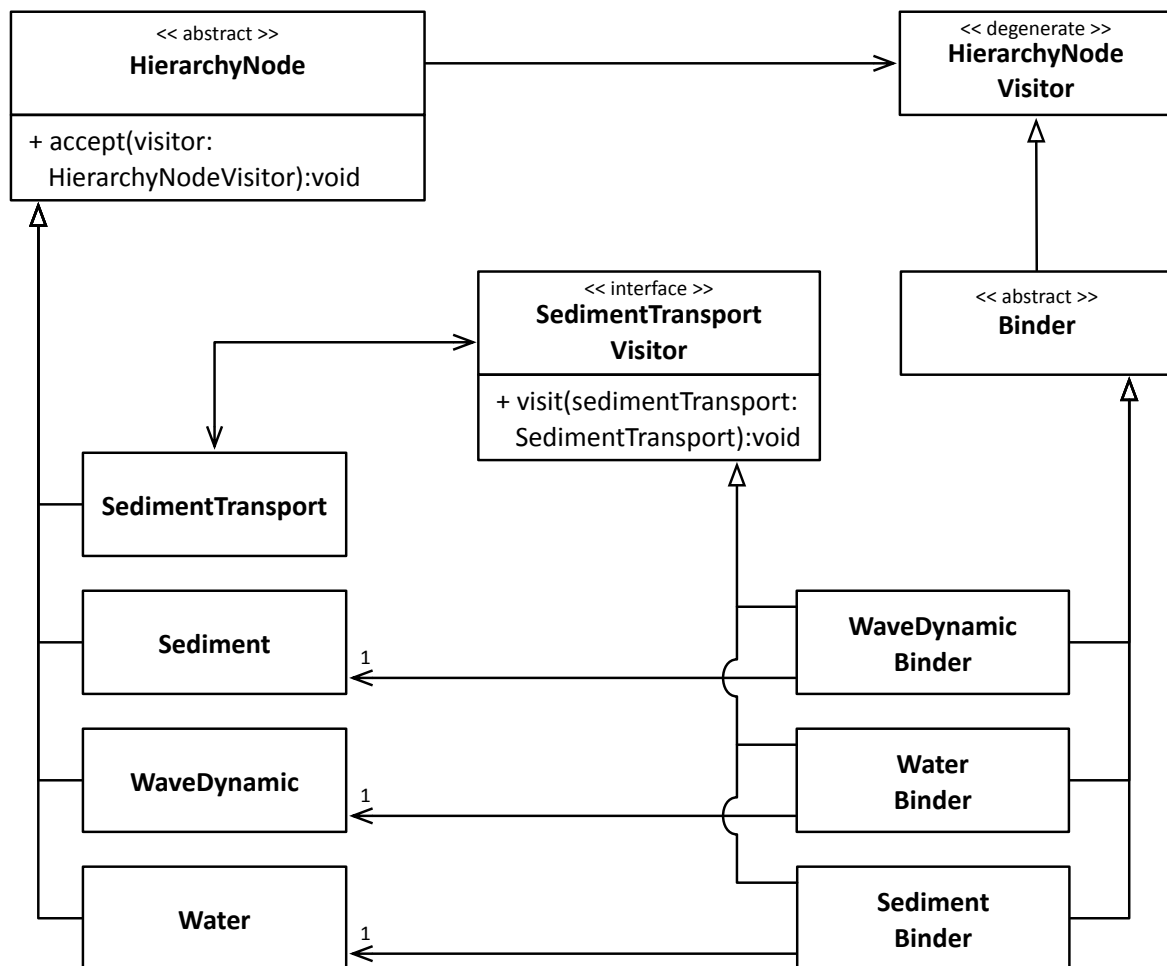


Abbildung 7.4: Erweiterung des Binding-Mechanismus.

Das Binden von Kindknoten übernehmen typspezielle Binder-Klassen. Um ein Sediment als Kind bei einem übergeordneten Knoten zu setzen, ist eine Erweiterung der Binder-Hierarchie um den SedimentBinder notwendig. Dadurch ergibt sich auch eine Anpassung in der BinderFactoryImplementation.

Bei dem SedimentTransport handelt es sich um einen Typ der Kindknoten aufnehmen kann. Damit dieser die Fähigkeit erhält die entsprechenden Binder als Besucher über die „visit“-Methode einzuladen, wird zunächst die Schnittstelle SedimentTransportVisitor umgesetzt.

Soll ein Subtyp von `HierarchyNode` an ein Objekt der Klasse `SedimentTransport` gebunden werden, muss der Binder dieses Derivats von der Schnittstelle abgeleitet werden. Aus diesem Grund wird eine Vererbungsbeziehung der Klassen `SedimentBinder`, `WaveDynamicBinder` und `WaterBinder` zu der abstrakten Besucherschnittstelle des `SedimentTransports` eingeführt.

Mit Aufruf der „accept“-Methode wird der *double dispatch*-Mechanismus des azyklischen Besucher-Musters ausgelöst. Das `SedimentTransport` unternimmt den Versuch den übergebenen `HierarchyNodeVisitor` in einen `SedimentTransportVisitor` umzuwandeln. Gelingt die Typumwandlung, wird die „visit“-Methode des Besuchers aufgerufen.

Den Binding-Mechanismus kann der Benutzer durch das `BindingMenu` veranlassen. Es enthält die Namen aller Knoten, die den in der Baumstruktur ausgewählten `HierarchyNode` als Kind binden können. Sobald sich die Auswahl in der Baumstruktur ändert, aktualisiert der `BindingMenuPresenter` die Einträge des `BindingMenu` wie in Abschnitt 6.2.5 beschrieben. Für diesen Mechanismus müssen die neuen Derivate von `HierarchyNode` die „canBind“-Methode so implementieren, dass sie für alle möglichen Knotentypen, die als Kind aufgenommen werden können, den Wert „true“ zurückgibt. Ist der übergebene Knoten von einem anderen Typ, muss der Wert „false“ zurückgegeben werden. Listing 7.2 zeigt die Umsetzung der Klasse `SedimentTransport`.

```
class SedimentTransport : public HierarchyNode{
2 //Konstruktor & Eigenschaften...
public:
4 virtual void accept(HierarchyNodeVisitor* visitor){
    if((SedimentTransportVisitor*) visitor)
6        (SedimentTransportVisitor*) visitor.visit(this);
    }
8
    virtual bool canBind(HierarchyNode *node){
10        if(node->getType() == "Sediment") return true;
        if(node->getType() == "Water") return true;
12        if(node->getType() == "WaveDynamic") return true;
        return false;
14    }
}
```

Listing 7.2: Umsetzung der „accept“-Methode des `SedimentTransports`.

Die Implementierung der Binder-Klassen wird anhand von Listing 7.3 und 7.4 gezeigt. Dabei dient der `SedimentBinder` als Beispiel für die Umsetzung eines neuen Subtyps. Anhand des `WaveDynamicBinders` wird verdeutlicht, wie ein bereits bestehender Binder modifiziert werden kann.

```
1 class SedimentBinder : public Binder, public SedimentTransportVisitor{
    private:
3     Sediment* sediment;

    public:
5     //Konstruktor
7     virtual void visit(SedimentTransport* sedimentTransport){
        sedimentTransport->setSediment(sediment);
9     }
}
```

Listing 7.3: Umsetzung der Klasse `SedimentBinder`.

```
class WaveDynamicBinder : public Binder,
2                          public MorisonVisitor,
                          public SedimentTransportVisitor{
4     private:
        WaveDynamic* waveDynamic;
6     public:
        //Konstruktor
8     virtual void visit(Morison* morison){
        morison->setWaveDynamic(waveDynamic);
10    }
    virtual void visit(SedimentTransport* sedimentTransport){
12        sedimentTransport->setWaveDynamic(waveDynamic);
    }
14 }
```

Listing 7.4: Umsetzung der Klasse `WaveDynamicBinder`.

Bei dem `WaveDynamicBinder` wird eine zusätzliche Vererbung zu der Besucherschnittstelle des `SedimentTransports` eingeführt. Dadurch wird die Implementierung der „visit“-Methode erforderlich, die es ihm erlaubt ein `SedimentTransport` zu besuchen (Z. 11-13). Bereits bestehender Quelltext dieser Klasse muss dafür nicht modifiziert werden.

7.2.3 Erweiterung der Editor-Hierarchie

Die Veränderung der Knoteneigenschaften erfolgt über Klassen der Editor-Hierarchie. Für eine Erweiterung des Systems um die beiden Subtypen `Sediment` und `SedimentTransport` ist ebenfalls eine Erweiterung um die Klassen `SedimentEditor` und `SedimentTransportEditor` notwendig. Zur Erzeugung der Editor-Klassen muss eine Anpassung der `EditorFactoryImplementation` stattfinden. Es ergibt sich ein System analog zu Abbildung 7.5.

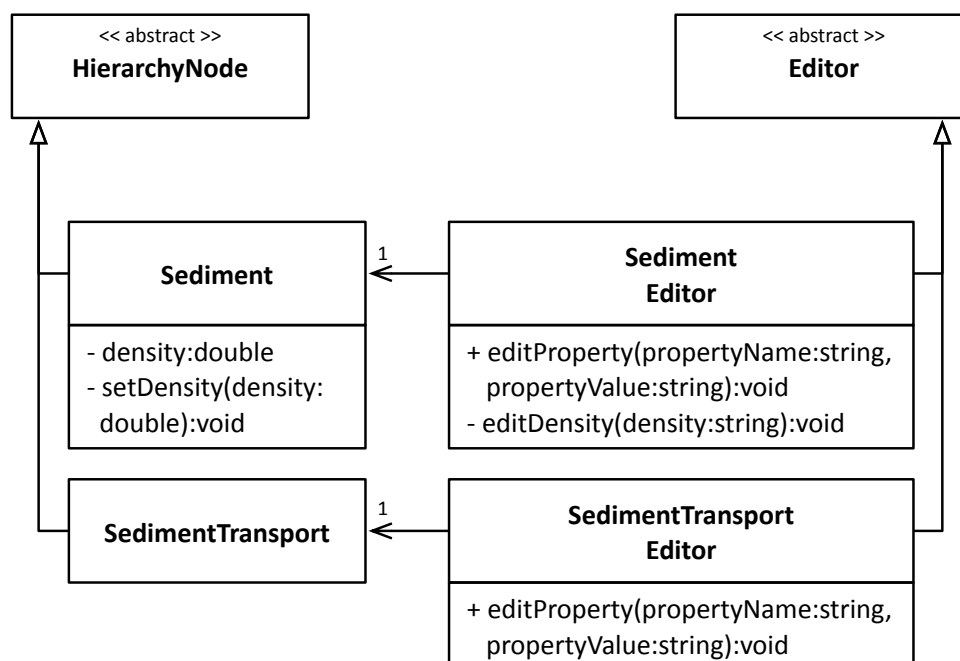


Abbildung 7.5: Erweiterung der Editor-Klassenhierarchie.

Das `Sediment` ist durch seinen Parameter „density“ gekennzeichnet. Zu dessen Modifikation stellt es die Methode „setDensity“ bereit. Wird dem `SedimentEditor` als Eigenschaftsname der string „Density“ übergeben, wird die private Methode „editDensity“ aufgerufen. Mit Hilfe der Operation „std::stod“ wird der string, der den neuen Eigenschaftswert enthält, zu einem double konvertiert. Anschließend wird die „setter“-Methode des `Sediments` aufgerufen.

Das `SedimentTransport` enthält bis auf seine Kindknoten keine Parameter. Da die Kindknoten innerhalb des Properties Explorers nicht verändert werden können, spielen sie für den `SedimentTransportEditor` keine Rolle. Aus diesem Grund beinhaltet er keine privaten Methoden zur Modifikation des `SedimentTransports`. Listing 7.5 und 7.6 zeigen die Umsetzung der Editor-Klassen.

```
class SedimentEditor : Editor{
2   private:
    Sediment* sediment;
4
    public:
6   SedimentEditor(Sediment* node) : sediment(node) {}

8   virtual void editProperty(string propertyName, string propertyValue){
        if(propertyName == "Name")
10      editName(sediment, propertyValue); //aus Basisklasse
        if(propertyName == "Density") editDensity(propertyValue);
12
        else throw UnknownPropertyException();
14   }

16   private:
    void editDensity(string density){
18      sediment->setDensity(std::stod(x));
        }
20 }
```

Listing 7.5: Umsetzung der Klasse SedimentEditor.

```
class SedimentTransportEditor : Editor{
2   private:
    SedimentTransport* sedimentTransport;
4
    public:
6   SedimentTransportEditor(SedimentTransport* node) :
        sedimentTransport(node) {}

8   virtual void editProperty(string propertyName, string propertyValue){
10      if(propertyName == "Name")
        editName(sedimentTransport, propertyValue); //aus Basisklasse
12
        else throw UnknownPropertyException();
14   }
}
```

Listing 7.6: Umsetzung der Klasse SedimentTransportEditor.

7.3 Erweiterung der Präsentation um neue Knotentypen

In diesem Abschnitt wird gezeigt, wie neue Knotenstrukturen nach der Erweiterung des Modells innerhalb der Applikation visualisiert werden können. Im Einzelnen wird dabei auf die Darstellung der Knoten in der Baumstruktur, die Anzeige der Knoteneigenschaften im Properties Explorer sowie die Erweiterung der Applikation um neue Arten der Auswertung eingegangen.

7.3.1 Darstellung neuer Knotentypen in der Baumstruktur

Die Darstellung der HierarchyNodes wird von Objekten der HierarchicItem-Hierarchie übernommen. Damit neue Knotentypen in der Baumstruktur des Solution Explorers angezeigt werden können, ist deswegen eine Erweiterung der Applikation um die Klassen SedimentItem und SedimentTransportItem erforderlich. Abbildung 7.6 zeigt die Erweiterung der Software.

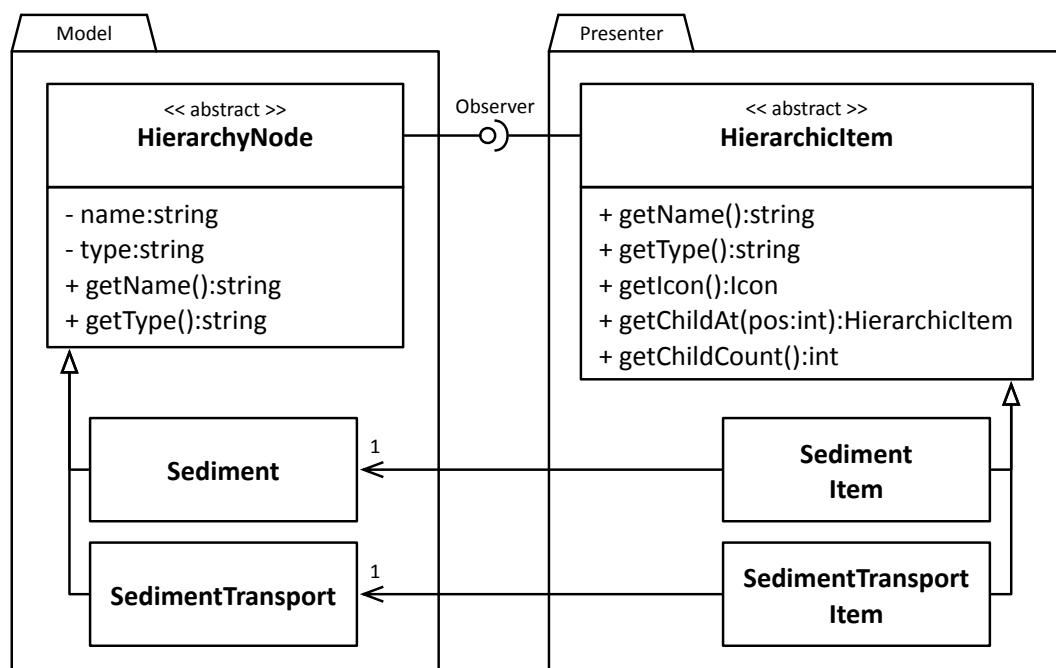


Abbildung 7.6: Erweiterung der HierarchicItem-Klassenhierarchie.

Beide Klassen sind von der abstrakten Schnittstelle HierarchicItem abgeleitet. Um Instanzen dieser Typen während der Laufzeit erzeugen zu können, muss die HierarchicItemFactory-Implementation erweitert werden.

Die Umsetzung der HierarchicItem-Klassen erfolgt analog zu dem WaveDynamicItem aus Listing 6.2 (s. Seite 66).

7.3.2 Darstellung der Eigenschaften neuer Knotentypen

Die Anzeige der knotenspezifischen Eigenschaften im Properties Explorer basiert auf einer Verkettung von `PropertiesItems`, welche durch ein Wurzelement des Typs `TableRoot` repräsentiert wird. Die Erzeugung dieser Wurzelemente wird von den Derivaten der `TableRootBuilder`-Hierarchie übernommen. Zur Zusammenstellung der Daten für die Eigenschaftstabelle im Properties Explorer, basierend auf den Knotentypen `Sediment` und `SedimentTransport`, ist eine Erweiterung der Applikation um die Klassen `SedimentRootBuilder` und `SedimentTransportRootBuilder` erforderlich. Diese werden, wie aus Abbildung 7.7 ersichtlich, von der abstrakten Schnittstelle `TableRootBuilder` abgeleitet. Daraus ergibt sich Anpassungsbedarf in der `TableRootBuilderFactoryImplementation`. Die Umsetzung der Klassen erfolgt analog zu dem `CoordinateSystemRootBuilder` aus Listing 6.4 (s. Seite 76).

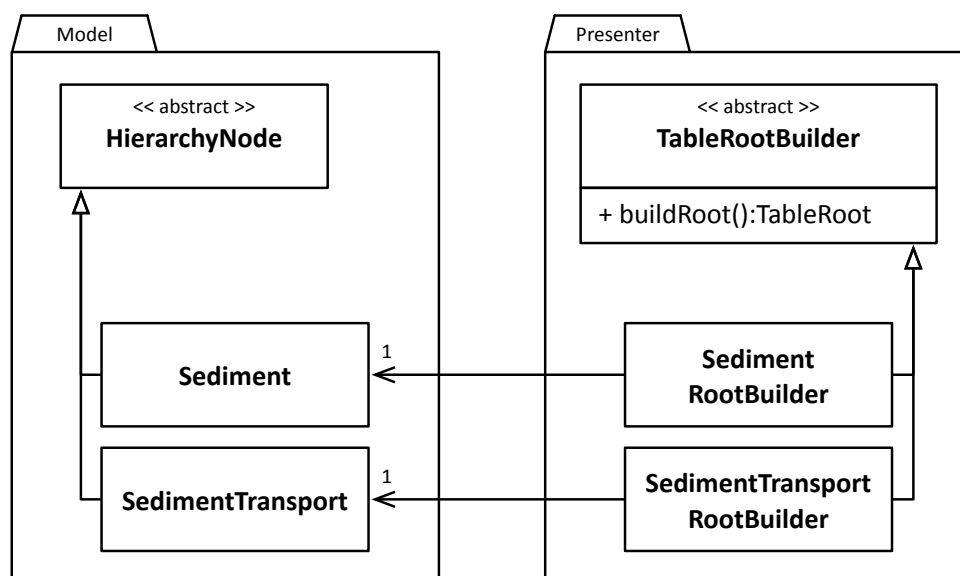


Abbildung 7.7: Erweiterung der `TableRootBuilder`-Klassenhierarchie.

7.3.3 Erweiterung um neue Auswertungsarten

Die Auswertung der `HierarchyNodes` basiert auf der abstrakten Schnittstelle `View`. Um der Applikation neue Auswertungsarten hinzuzufügen, muss die `View`-Hierarchie um neue Derivate erweitert werden. Damit die Konformität zu dem Passive-View-Muster eingehalten wird, ist zusätzlich die Einführung einer passenden Presenterklasse erforderlich. Folgend wird die

Applikation exemplarisch um die Auswertungsart „SomeEvaluation“ erweitert, welche die Berechnungsergebnisse der Klasse SedimentTransport visualisiert. Es entsteht ein System nach Abbildung 7.8.

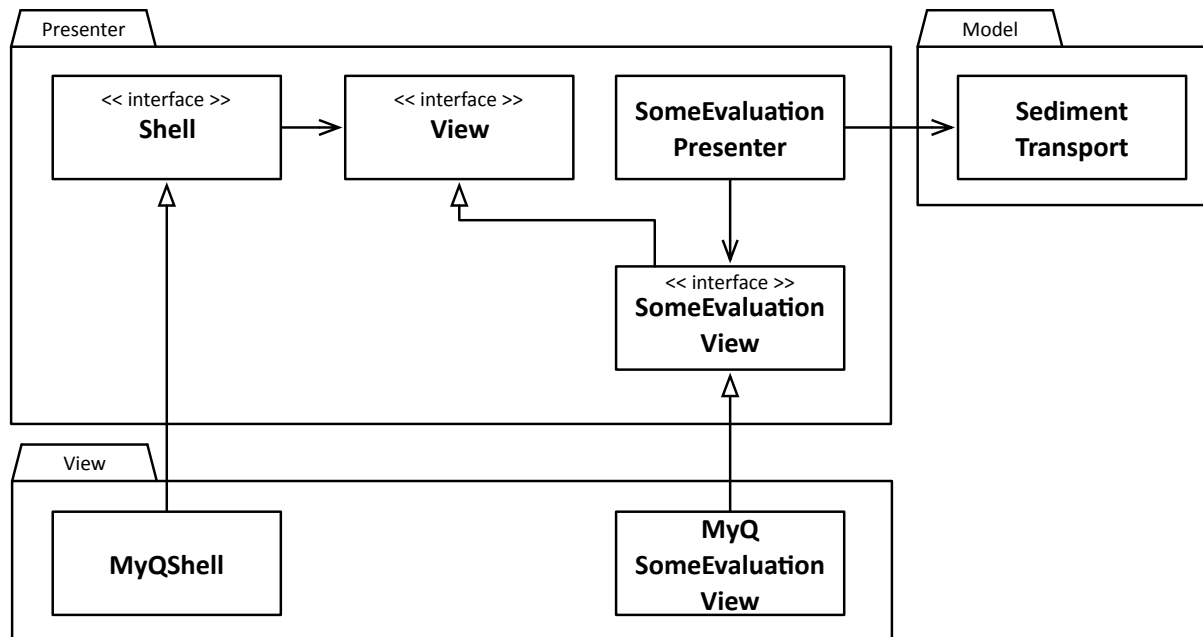


Abbildung 7.8: Erweiterung der View-Klassenhierarchie.

Für jeden ausgewählten Knoten in der Baumstruktur zeigt das EvaluationMenu die möglichen Auswertungsarten an. Ändert sich die Auswahl aktualisiert der EvaluationMenuPresenter die Einträge des Menüs entsprechend des Typs des ausgewählten Knotens. Dafür wird die „getTypeList“-Methode der ViewFactoryImplementation aus Abbildung 6.16 (s. Seite 81) verwendet. Sie gibt entsprechend des erhaltenen Knotentyps eine Liste von strings mit den möglichen Auswertungsarten zurück. Damit das Menü den Eintrag „SomeEvaluation“ enthält, muss die Liste, die für den Typ „SedimentTransport“ zurückgegeben wird, einen string mit diesem Inhalt umfassen. Zugleich muss innerhalb der Fabrik-Methode „makeView“ die Erzeugung der Objekte vom Typ MyQSomeEvaluationView und SomeEvaluationMenuPresenter mit diesem string verknüpft werden.

KAPITEL 8

Zusammenfassung

Zusammenfassung

In dieser Arbeit wird eine mögliche Implementierung der Applikation CECLOUD vorgestellt, deren Funktionalität die automatisierte Auswertung und Visualisierung von Berechnungstheorien im Küsteningenieurwesen beinhaltet. Dabei wird gezeigt wie qualitativ hochwertige Software unter Anwendung der Methoden und Prinzipien des objektorientierten Designs entworfen werden kann.

Zu Beginn wird auf die Grundlagen des Softwareentwurfs eingegangen. Danach werden die Methoden und Prinzipien des objektorientierten Designs erläutert. Darunter befinden sich die SOLID-Prinzipien sowie einige Entwurfs- und Architekturmuster. Als nächstes folgt die Beschreibung der Anforderungen an die Applikation. Neben der Funktionalität die Berechnungstheorien innerhalb der Anwendung organisieren und auswerten zu können, sollen weitere Merkmale umgesetzt werden. Es wird gefordert, dass die Applikation erweiterbar gestaltet wird, sodass neue Berechnungstheorien integriert werden können. Darüber hinaus soll sie über eine intuitive Benutzeroberfläche verfügen. Außerdem ist auf die plattformunabhängige Ausführbarkeit der Applikation zu achten. Die Qualität des Entwurfs kann an den nachfolgend erläuterten internen und externen Faktoren überprüft werden, die zur Umsetzung hochwertiger

Softwaresysteme herangezogen werden. Aus den zuvor beschriebenen Anforderungen ergeben sich für die Applikation CECLOUD die Flexibilität, die Benutzerfreundlichkeit und die Kompatibilität als wichtige Faktoren. Basierend auf den bisher gewonnenen Kenntnissen erfolgt dann die Implementierung der Anwendung. Die Hauptbestandteile der Applikation sind entsprechend der gewählten Passive-View-Architektur die Bereiche Modell, View und Presenter. Die Umsetzung der Software geschieht in zwei Abschnitten. Zunächst wird der Modellbereich entworfen, in dem die Logik der Problemdomäne umgesetzt wird. Im Anschluss wird das System zur Präsentation der Anwendung auf dem Bildschirm gezeigt. Dazu gehören die Bereiche View und Presenter. Zum Schluss findet exemplarisch eine schrittweise Erweiterung der Applikation um die Berechnungstheorie des Sedimenttransports statt. Dadurch kann gezeigt werden, dass die geforderte Erweiterbarkeit der Applikation gegeben ist.

Fazit

Durch die Anwendung der Prinzipien des objektorientierten Designs kann eine Applikation entwickelt werden, deren Struktur hinsichtlich ihrer Beständigkeit in zwei Bereiche unterteilt wird. Dazu gehört zum einen ein beständiger Applikationskern, zum anderen ein volatiler Bestandteil. Der Kern beinhaltet dabei die Grundfunktionalität der Applikation. Er basiert auf Abstraktionen, die durch Beziehungen untereinander ein abstraktes System erzeugen, in welchem die umzusetzenden Themenbereiche des Küsteningenieurwesens gänzlich unbekannt sind. Mit dem volatilen Bestandteil wird die problembezogene Thematik hinzugefügt. Abbildung 8.1 zeigt das Zusammenspiel der beiden Applikationsbereiche.

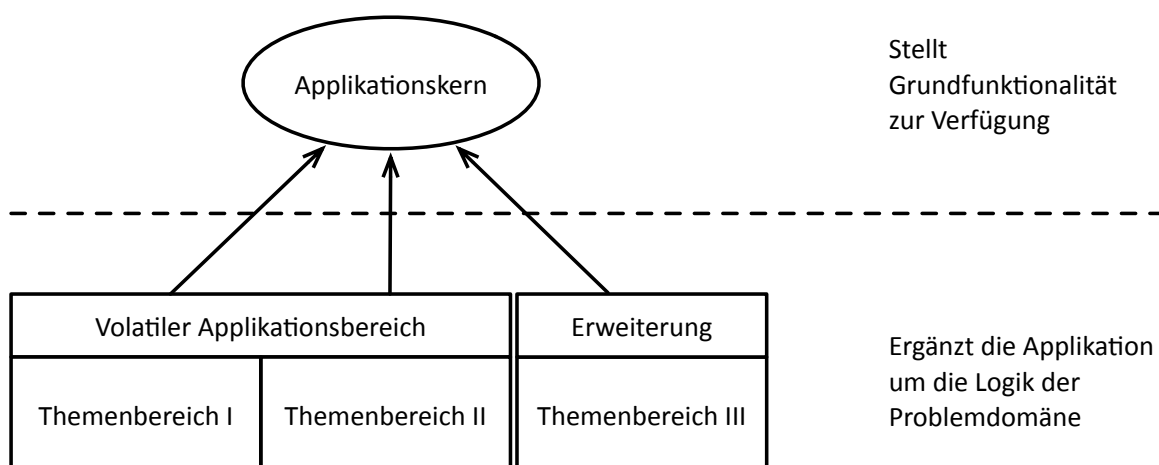


Abbildung 8.1: Unterteilung der Applikationsstruktur

Die volatilen Bereiche der Problemdomäne basieren auf dem abstrakten Applikationskern. Sie verwenden seine Schnittstellen, um Zugang zu der von ihm bereitgestellten Basisfunktionalität zu erlangen. Durch diese Struktur ist der Kern der Applikation von der konkreten Detailebene getrennt. Alle Änderungen die getätigt werden, um Funktionalität hinzuzufügen oder zu modifizieren, geschehen auf der Ebene des volatilen Bereichs. Die grundlegende Struktur der Software ist davon nicht betroffen. Im Vergleich zu starren Softwaresystemen, in denen die einzelnen Bereiche eng miteinander gekoppelt sind, erlaubt die Applikation CECloud Erweiterungen und Modifikationen der Funktionalität mit wenig Aufwand vorzunehmen. Aus diesem Grund erfüllt CECloud die geforderte Erweiterbarkeit, sie verfügt somit über eine hohe Flexibilität.

Die Umsetzung einer intuitiven Benutzeroberfläche, bedarf in der Regel mehrerer Iterationsschritte. Es treten immer wieder Änderungen in der Präsentationsschicht der Anwendung auf, bis eine möglichst intuitive Bedienung erreicht wird. Die Verwendung des Passive-View-Musters erzeugt eine lose Kopplung zwischen der Logik und der Präsentation auf dem Bildschirm. Dies erlaubt eine flexible Änderung der Darstellung, ohne dass sich daraus Anpassungen in den Bereichen Modell und Presenter ergeben. Dadurch kann die Applikation mit einer hohen Benutzerfreundlichkeit umgesetzt werden.

Des Weiteren wird durch die Anwendung des Passive-View-Musters eine hohe Kompatibilität der Software erreicht. Für den Entwurf von grafischen Benutzeroberflächen wird in der Regel auf die Funktionalität von Visualisierungsframeworks zurückgegriffen. Diese sind meistens an ein Betriebssystem gebunden. Das Passive-View-Muster bietet die Möglichkeit die Klassen der View auszutauschen. Dadurch kann das verwendete Visualisierungsframework problemlos variiert werden.

Ausblick

Eine wichtige Funktionalität von Software ist das permanente Abspeichern der bearbeiteten Daten im Dateisystem des Computers. Die im Rahmen dieser Arbeit entstandene Applikation umfasst lediglich die Modellierung und Auswertung von Berechnungstheorien. Das Speichern oder Einlesen von Projektdateien ist noch nicht möglich. Zukünftig könnte die Software um eine Komponente erweitert werden, die die Persistenz ermöglicht. Dafür könnte das Memento-Muster (Gamma u. Riehle, 2004) in der von Linxweiler (2011) gezeigten Variante verwendet werden.

Zusätzlich sollte die Applikation über ein benutzerfreundliches Hilfesystem verfügen, das einerseits Hinweise zur Bedienung des Programms gibt, andererseits aber auch die zur Berechnung verwendeten mathematischen Beziehungen erläutert. Dadurch wird die Auswertung für den Benutzer transparent.

Durch die lose Kopplung zwischen Grundfunktionalität und problembezogener Thematik ist CECloud nicht an den Fachbereich des Küsteningenieurwesens gebunden. Zukünftig könnte die Applikation um weitere Ingenieurdisziplinen erweitert werden, sodass eine Sammlung von fachübergreifenden Berechnungen und Nachweisen entsteht.

Literaturverzeichnis

- [Booch 2007] BOOCH, Grady: *Object-oriented analysis and design with applications*. 3. ed., 1. printing. Upper Saddle River, NJ : Addison-Wesley, 2007 (The Addison-Wesley object technology series). <http://proquest.tech.safaribooksonline.de/9780201895513>. – ISBN 9780201895513
- [Brooks 1987] BROOKS: No Silver Bullet Essence and Accidents of Software Engineering. In: *Computer* 20 (1987), Nr. 4, S. 10–19. <http://dx.doi.org/10.1109/MC.1987.1663532>. – DOI 10.1109/MC.1987.1663532. – ISSN 0018–9162
- [Budgen 2003] BUDGEN, D.: *Software design*. 2nd ed. Harlow : Pearson Addison-Wesley, 2003 (International computer science series). – ISBN 9780201722192
- [Buschmann, F. et al. 1996] BUSCHMANN, F. ; SCHMIDT, DOUGLAS C. PATTERNS FOR CONCURRENT AND NETWORKED OBJECT ; KIRCHER, MICHAEL. PATTERNS FOR RESOURCE MANAGEMENT ; BUSCHMANN, FRANK. PATTERN LANGUAGE FOR DISTRIBUTED COMPUTING ; BUSCHMANN, FRANK. ON PATTERNS AND PATTERN LANGUAGES: *Pattern-oriented software architecture*. Chichester : Wiley, 1996- (Wiley series in software design patterns). – ISBN 9780471958697
- [Dijkstra 1972] DIJKSTRA, Edsger W.: The humble programmer. In: *Commun. ACM* 15 (1972), Nr. 10, S. 859–866

- [Dijkstra 1979] DIJKSTRA, Edsger W.: Programming considered as a human activity. In: YOURDON, Edward N. (Hrsg.): *Classics in software engineering*. Upper Saddle River, NJ, USA : Yourdon Press, 1979, S. 1–9
- [Fowler 2010] FOWLER, Martin: *Development of Further Patterns of Enterprise Application Architecture*. "Website", 2010. – Available online at <http://www.martinfowler.com/eaDev/index.html>, visited on 2010-02-15
- [Gamma u. Riehle 2004] GAMMA, Erich ; RIEHLE, Dirk: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl., [Neuauf]. München and Boston [u.a.] : Addison-Wesley, 2004 (Programmer's choice). – ISBN 3–8273–2199–9
- [Linxweiler 2011] LINXWEILER, Jan: *Ein integrierter Softwareansatz zur interaktiven Exploration und Steuerung von Strömungssimulationen auf Many-Core-Architekturen*, Technischen Universität Carolo-Wilhelmina zu Braunschweig, Diss., 2011
- [Liskov 1987] LISKOV, Barbara: Data Abstraction and Hierarchy. In: *ACM SIGPLAN Notices* 23 (1987), 3, Nr. 5, S. 17–34
- [Martin 2000] MARTIN, Robert C.: *Design Principles and Design Patterns*. Object Mentor, 2000
- [Martin u. Martin 2008] MARTIN, Robert C. ; MARTIN, Micah: *Agile principles, patterns, and practices in C#*. 4. print. Upper Saddle River, NJ : Prentice-Hall, 2008 (Robert C. Martin series). – ISBN 9780131857254
- [McCall 1977] MCCALL, J.: *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisition Manager*. Bd. 1-3. General Electric, 1977
- [McConnell 2009] MCCONNELL, Steve: *Code Complete*. 2nd ed. Sebastopol : O'Reilly Media Inc, 2009. – ISBN 9780735619678
- [Metz 2013] METZ, Sandi: *Practical object-oriented design in Ruby: An agile primer*. Upper Saddle River, NJ : Addison-Wesley/Pearson, 2013 (Always learning). – ISBN 0321721330
- [Meyer 2009] MEYER, Bertrand: *Object-oriented software construction*. 2. ed., 15. print. Upper Saddle River, NJ : Prentice Hall PTR, 2009. – ISBN 9780136291558

- [Miller 1956] MILLER, George A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. In: *The Psychological Review* 63 (1956), March, Nr. 2, S. 81–97
- [Parnas 1972] PARNAS, David L.: On the Criteria To Be Used in Decomposing Systems into Modules. In: *Communications of the ACM* 15 (1972), December, Nr. 12, S. 1053–1058
- [Potel 1996] POTEL, Mike: MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java, Taligent Inc. 1996. – Forschungsbericht
- [Stroustrup 2015] STROUSTRUP, Bjarne: *Die C++-Programmiersprache: Aktuell zum C++11-Standard*. München : Hanser, 2015. – ISBN 978–3446439610
- [Wulf et al. 1981] WULF, W. ; LONDON, R. ; SHAW, M.: Abstraction and verification in Alphard: Introduction to language and methodology. In: SHAW, M. (Hrsg.): *ALPHARD: Form and Content*. New York : Springer-Verlag, 1981
- [Yourdon u. Constantine 1979] YOURDON, Edward ; CONSTANTINE, Larry L.: *Structured design: Fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ : Prentice-Hall, 1979. – ISBN 0–13–854471–9

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorstehende Studienarbeit selbstständig angefertigt und die benutzten Hilfsmittel sowie die befragten Personen und Institutionen vollständig angegeben habe.

Braunschweig, den 24. Oktober 2016

Janis Zisenis