

Technische Universität Carolo-Wilhelmina zu Braunschweig
Fakultät für Architektur, Bauingenieurwesen und Umweltwissenschaften

Studienarbeit



Janis Zisenis

4104786

Anwendung und Evaluation der testgetriebenen Entwicklung am Beispiel eines plattformübergreifenden Frameworks zur Realisierung nativer Benutzeroberflächen

Institut für rechnergestützte Modellierung im Bauingenieurwesen

Erstprüfer: Prof. Dr.-Ing. habil. Manfred Krafczyk

Zweitprüfer: Prof. Dr. rer. nat. Martin Geier

Betreuer: Dr.-Ing. Jan Linxweiler

Braunschweig 2018

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Listingsverzeichnis	xi
Abkürzungsverzeichnis	xvii
1 Einleitung	1
2 Grundlagen der testgetriebenen Entwicklung	5
2.1 Konzept der testgetriebenen Entwicklung	7
2.2 Vorgehensweise der testgetriebenen Entwicklung	8
2.3 Unit Testing	11
2.3.1 Definition des Unit Tests	12
2.3.2 Aufbau eines Unit Tests	13
2.3.3 Eigenschaften eines guten Unit Tests	14
2.4 Testen in Isolation	16
2.4.1 Indirekter Input/Output	17
2.4.2 Test Doubles	19

2.5	Effekte der testgetriebenen Entwicklung	22
2.5.1	Verifizierung des Systemverhaltens	22
2.5.2	Objektorientiertes Design	23
2.5.3	Dokumentation	24
2.5.4	Entlastung des Entwicklers	25
2.6	GoogleMock	25
2.7	Zusammenfassung	29
3	Konzeption des Entwurfs von CrossNative	31
3.1	Allgemeiner Aufbau grafischer Oberflächen mit verschiedenen GUI-Frameworks	33
3.1.1	Aufbau einer GUI unter Verwendung von Qt	34
3.1.2	Aufbau der GUI unter Verwendung von Cocoa	35
3.2	Aufbau applikationsspezifischer Oberflächen	37
3.3	Schwierigkeiten bei dem Entwurf von grafischen Oberflächen	40
3.3.1	Komplexität der grafischen Oberfläche	40
3.3.2	Unterschiede in den Funktionsweisen der Frameworks	41
3.3.3	Die GUI – Der volatilste Bestandteil der Software	41
3.3.4	Adressieren der Entwurfsschwierigkeiten	42
3.4	Positionskriterien	47
3.5	Speicherverwaltung durch intelligente Zeiger	48
3.6	Zusammenfassung	50
4	Testgetriebene Entwicklung des Kompositionsalgorithmus	53
4.1	Zyklus 1 – Erstmaliges Verknüpfen von Views	54
4.2	Zyklus 2 – Unterlassen des Verknüpfens	63
4.3	Zyklus 3 – Die dritte Ebene der Objekthierarchie	71
4.4	Zyklus 4 – Unterlassen des Verknüpfens auf dritter Ebene	74
4.5	Zyklus 5 – FirstView als Elternknoten von ThirdView	75
4.6	Zyklus 6 – Unterlassen des Verknüpfens von FirstView und ThirdView	77
4.7	Zyklus 7 – Komplexes Verknüpfen	79
4.8	Zusammenfassung	90

5 Enstehung des Framework CrossNative	93
5.1 Anwendungsbeispiel	94
5.2 Basis der Objekthierarchie	97
5.3 Zusammensetzbare Knoten der Hierarchie	99
5.4 Vermeidung von Redundanzen durch den Acceptor	107
5.5 Verknüpfen der Views durch den besuchenden Composer	111
5.6 Identifizieren der Elternknoten durch den besuchenden Matcher	114
5.7 Verallgemeinerung der Tags	122
5.8 Zusammenfassung	125
6 Entwurf des Gittergenerators – Anwendung von CrossNative	129
6.1 Konzept der grafischen Oberfläche	130
6.2 CrossNative als hierarchisches Modell	133
6.3 Implementierung des Solution Explorers	137
6.4 Implementierung des Properties Explorers	139
6.5 Implementierung der Knotenvisualisierungen	141
6.6 Implementierung der Menüsteuerung	142
6.6.1 Implementierung der Menüeinträge	143
6.6.2 Implementierung der Menüs	144
6.6.3 Implementierung der Menüleiste	145
6.7 Implementierung des Hauptanwendungsfensters	146
6.8 Komposition der GUI unter Verwendung von CrossNative	149
6.9 Flexibilität der GUI durch CrossNative	154
6.10 Zusammenfassung	157
7 Zusammenfassung & Ausblick	159
Literatur	163

Abbildungsverzeichnis

2.1	Die Phasen der testgetriebenen Entwicklung nach Beck (2003) in Anlehnung an Freeman et al. (2009).	9
2.2	Abhangigkeit des Testcodes zu dem Produktionscode in Anlehnung an Martin et al. (2008).	16
2.3	Ablauf eines Tests mit <i>value-based</i> oder <i>state-based</i> Endergebnis und DOC in Anlehnung an Meszaros (2006).	17
2.4	Ablauf eines Tests mit <i>interaction-based</i> Endergebnis und DOC in Anlehnung an Meszaros (2006).	18
2.5	Entkopplung des Tests von der DOC des SUT in Anlehnung an Martin et al. (2008).	20
3.1	Objekthierarchie einer grafischen Oberfache unter Verwendung des Frameworks Qt.	35
3.2	Objekthierarchie einer grafischen Oberfache unter Verwendung des Frameworks Cocoa.	37
3.3	Die Objekthierarchie einer applikationsspezifischen GUI.	38
3.4	Mogliche grafische Darstellung der Objekthierarchie aus Abbildung 3.3. .	39
3.5	Schichtenarchitektur einer Applikation in Anlehnung an Brown et al. (2003). .	41

3.6	Composite Pattern in Anlehnung an Gamma et al. (2004).	43
3.7	Entkopplung des Kompositionsalgorithmus von den View-Implementierungen.	45
3.8	Austauschbarkeit des Identifizierungsmechanismus durch die Klassenhierarchie der Kriterien.	48
4.1	Grafische Darstellung des ersten Testfalls.	54
4.2	Das Klassensystem nach der Implementierung der ersten Testmethode. .	60
4.3	Grafische Darstellung des zweiten Testfalls.	64
5.1	Frameworkspezifische Objekthierarchie des Anwendungsbeispiels.	94
5.2	Applikationsspezifische Objekthierarchie des Anwendungsbeispiels.	94
5.3	Verknüpfen der Menüleiste mit dem Menü.	96
5.4	Umsetzung des <i>Dependency Inversion Principles</i> und des <i>Open/Closed Principles</i> im Klassensystem der Views.	97
5.5	CNComponent als Schnittstelle des <i>Composite Patterns</i> in Anlehnung an Gamma et al. (2004).	98
5.6	Verwendung der Komposition für die Umsetzung verschiedener View-Knoten.	101
5.7	Verwendung der <i>Acyclic Visitor Patterns</i> in Anlehnung an Martin et al. (2008) zur Verknüpfung der Views.	103
5.8	Verbergen der <i>Templates</i> mit Hilfe des <i>Interfaces CNAcceptor</i>	110
5.9	Verwendung des <i>Interfaces CNComposer</i> durch die Klasse CNComposable . .	112
5.10	Klassenbeziehungen für das Laden eines CNComponents in die CNDynamicHierarchy	115
5.11	Klassensystem zur <i>Tag</i> -basierten Identifikation der Klasse CocoaMenu . .	119
5.12	Klassenbeziehungen des CNVisitingMatchers	121
5.13	Klassenbeziehungen zur <i>Tag</i> -basierten Identifizierung der Views.	122
5.14	Verwenden des einheitlichen <i>Interfaces CNTagged</i> zur Identifizierung von Views.	124
5.15	Das Framework CrossNative im Überblick.	128

6.1	Interne Modellstruktur zur Generierung eines Gitters.	130
6.2	Exemplarische Darstellung einer Modellstruktur zur Generierung eines Gitters.	131
6.3	Exemplarische Darstellung einer Eigenschaften der STL-Datei <i>STLFile_1</i> aus Abbildung 6.1 in dem <i>Properties Explorer</i>	132
6.4	Mögliche Aufteilung der Ansichten in dem Hauptanwendungsfenster. . . .	132
6.5	Klassenbeziehungen des <i>GridGenerators</i>	133
6.6	Klassenbeziehungen der <i>Simulation</i>	134
6.7	Zusammensetzen des <i>GridGenerators</i> unter Verwendung des azyklischen Besuchermusters.	135
6.8	Die API der Klasse CNHierarchy	136
6.9	Klassenbeziehungen des SolutionExplorers	138
6.10	Beobachten der Modellhierarchie durch den SolutionExplorer	138
6.11	Propagieren des ausgewählten CNIndex durch den SolutionExplorer . .	139
6.12	Beobachten der aktuellen Auswahl durch den PropertiesExplorer	140
6.13	Aktualisierung der tabellarischen Ansicht des PropertiesExplorers . . .	141
6.14	Visualisierung eines Gitters mit Hilfe des QVTKWidgets	142
6.15	Klassenbeziehungen der View-Klasse MenuEntry	144
6.16	Klassensystem der Klasse Menu	145
6.17	Klassenbeziehungen der Klasse MenuBar	146
6.18	Konzeptionelle Darstellung der <i>Shell</i>	147
6.19	Frameworkspezifische Objekthierarchie der View-Klasse Shell	148
6.20	Klassensystem der View-Klasse Shell	149
6.21	Klassensystem zur Komposition der Shell unter Verwendung von Cross-Native.	150
6.22	Die modulare Unterteilung des <i>Presentation Layers</i>	154
6.23	Kategorische Identifizierung des ShellComposingVisitors	156

Listingsverzeichnis

2.1	Beispiel eines <i>Unit Tests</i> unter Verwendung des Frameworks GoogleMock in Anlehnung an Osheroove (2014)	13
2.2	main-Methode zur Ausführung der Tests unter Verwendung des Frame- works GoogleMock.	25
2.3	Definition einer Testmethode mit dem TEST-Makro.	26
2.4	Implementierung einer <i>Test Fixture Class</i> und Definition eines Test mit dem TEST_F-Makro.	27
2.5	Anwendung der von GoogleMock bereitgestellten Matcher in dem Makro EXPECT_THAT	28
3.1	Komposition der exemplarischen Objekthierarchie nach Abbildung 3.3 oh- ne Verwendung von CrossNative.	44
3.2	Komposition der exemplarischen Objekthierarchie nach Abbildung 3.3 mit CrossNative.	46
3.3	Genereller Aufbau einer Klasse unter Verwendung von <code>shared_ptr</code>	49
4.1	Zyklus 1 – Rot: Implementierung des ersten Testmethode.	55

4.2	Zyklus 1 – Rot: Implementierung des <i>Test Fixtures</i> <code>CNDynamicViewHierarchyTest</code> .	56
4.3	Zyklus 1 – Rot: Erweiterung des <i>Test Fixtures</i> um die Helfer-Methode <code>expectReceiverAddedView</code> .	57
4.4	Zyklus 1 – Rot: Verwendung der Helfer-Methode aus Listing 4.3 in dem ersten Testfall.	57
4.5	Zyklus 1 – Rot: Implementierung der Klasse <code>CNDynamicHierarchy</code> (<code>CNDynamicViewHierarchy.h</code>).	58
4.6	Zyklus 1 – Rot: Implementierung der Klasse <code>CNDynamicHierarchy</code> (<code>CNDynamicViewHierarchy.cpp</code>).	58
4.7	Zyklus 1 – Rot: Implementierung des <i>Interfaces CNView</i> .	59
4.8	Zyklus 1 – Rot: Implementierung des <i>Test Doubles CNViewSpy</i> .	59
4.9	Zyklus 1 – Rot: Implementierung des <i>Test Doubles CNViewDummy</i> .	60
4.10	Zyklus 1 – Grün: Erweiterung der Klasse <code>CNDynamicViewHierarchy</code> um das Attribut <code>firstView</code> .	61
4.11	Zyklus 1 – Grün: Implementierung der <code>load</code> -Methode entsprechend des ersten Tests.	61
4.12	Zyklus 1 – Grün: Erweiterung des <i>Interfaces CNView</i> um die <code>add</code> -Methode.	62
4.13	Zyklus 1 – Grün: Erweiterung des <i>CNViewSpys</i> um die Implementierung der <code>add</code> -Methode zur Verifizierung des indirekten Outputs.	63
4.14	Zyklus 2 – Rot: Test 2, Variante 1 – Indirekter Input durch das <i>Interface CNView</i> .	65
4.15	Zyklus 2 – Rot: Erweiterung des <i>Test Fixtures</i> um die Helfer-Methode <code>expectReceiverDidNotAddView</code> .	66
4.16	Zyklus 2 – Rot: Test 2, Variante 2 – <code>load</code> -Methode mit direktem Input durch die Testmethode.	66
4.17	Zyklus 2 – Rot: Angepasste API der Klasse <code>CNDynamicViewHierarchy</code> zur Ausführung des Tests (<code>CNDynamicViewHierarchy.h</code>).	69
4.18	Zyklus 2 – Rot: Angepasste API der Klasse <code>CNDynamicViewHierarchy</code> zur Ausführung des Tests (<code>CNDynamicViewHierarchy.cpp</code>).	69

4.19 Zyklus 2 – Rot: Anpassen des ersten Tests an die veränderte API der Klasse <code>CNDynamicViewHierarchy</code>	69
4.20 Zyklus 2 – Grün: Erweiterung der <code>load</code> -Methode um den Einfluss der boolschen Variablen <code>matches</code> (<code>CNDynamicViewHierarchy.cpp</code>).	70
4.21 Zyklus 2 – Refactoring: Redundanzvermeidung durch Fabrik-Methoden für die Instanziierung der <i>Test Doubles</i> <code>CNViewDummy</code> und <code>CNViewSpy</code>	71
4.22 Zyklus 3 – Rot: Implementierung des dritten Testfalls.	72
4.23 Zyklus 3 – Grün: Erweiterung der Klasse <code>CNDynamicViewHierarchy</code> um das Attribut <code>secondView</code>	72
4.24 Zyklus 3 – Grün: Erweiterung der <code>load</code> -Methode entsprechend des dritten Testfalls.	73
4.25 Zyklus 3 – Refactoring: Erweiterung des <i>Test Fixtures</i> um die Helfer-Methode <code>getReceiverAddedViewErrorMessage</code>	73
4.26 Zyklus 4 – Rot: Implementierung des dritten Testfalls.	74
4.27 Zyklus 4 – Grün: Erweiterung der <code>load</code> -Methode um den Einfluss der boolschen Variable <code>matches</code> auf zweiter Ebene der Hierarchie.	75
4.28 Zyklus 5 – Rot: Implementierung des fünften Testfalls.	76
4.29 Zyklus 5 – Grün: Erweiterung der <code>load</code> -Methode um die Verknüpfung zwischen der ersten und der dritten View.	77
4.30 Zyklus 6 – Rot: Implementierung des sechsten Testfalls.	78
4.31 Zyklus 6 – Grün: Modifizierung der <code>load</code> -Methode, um die Verknüpfung zwischen der ersten und der dritten View zu verhindern.	79
4.32 Zyklus 7 – Rot: Implementierung des siebten Testfalls.	80
4.33 Zyklus 7 – Rot: Modifikation der Signatur der <code>load</code> -Methode (<code>CNDynamicHierarchyView.h</code>).	81
4.34 Zyklus 7 – Grün: Erweiterung der <code>load</code> -Methode um den Einfluss des <code>CNMatchers</code> in dem dritten Methodenaufruf.	82
4.35 Zyklus 7 – Grün: Erweiterung des <i>Interfaces</i> <code>CNMatcher</code> um die rein virtuelle Methode <code>matches</code>	82
4.36 Zyklus 7 – Grün: Erweiterung des <code>CNFakeMatchers</code> um die Methode <code>matches</code>	83

4.37 Zyklus 7 – <i>Refactoring</i> : Erweiterung des <i>Test Fixtures</i> um eine Fabrik-Methode zur Instanziierung und Konfiguration der Klasse <code>CNMatcherStub</code> .	84
4.38 Zyklus 7 – <i>Refactoring</i> : Anpassen der Tests an die <code>CNMatcher</code>	85
4.39 Zyklus 7 – <i>Refactoring</i> : Umstellung des Algorithmus auf die Verwendung der <code>CNMatcher</code>	85
4.40 Zyklus 7 – <i>Refactoring</i> : Verwendung eines <code>vectors</code> zum Speichern von Views (<code>CNDynamicViewHierarchy.cpp</code>).	86
4.41 Zyklus 7 – <i>Refactoring</i> : Vorziehen des Aufrufs der <code>push_back</code> -Methode der Klasse <code>vector</code>	87
4.42 Zyklus 7 – <i>Refactoring</i> : Umschließen der Aufrufe der <code>load</code> -Methode durch <code>for</code> -Schleifen mit statischen Zählvariablen.	88
4.43 Zyklus 7 – <i>Refactoring</i> : Ersetzen der statischen Zählvariablen durch die Größe des <code>vectors</code>	89
4.44 Zyklus 7 – <i>Refactoring</i> : Eliminieren des <code>if-else</code> -Blocks.	89
 5.1 Implementierung der Klasse <code>CocoaMenubar</code> durch Ableiten von <code>CNComponent</code>	99
5.2 Implementierung der Klasse <code>CocoaMenu</code> durch Ableiten von <code>CNComponent</code>	100
5.3 Implementierung der <code>accept</code> -Methode der Klasse <code>CocoaMenu</code>	104
5.4 Implementierung der Klasse <code>CocoaMenubarComposingVisitor</code>	105
5.5 Implementierung der <code>accept</code> -Methode der Klasse <code>CocoaMenuComponent</code>	105
5.6 Implementierung der <code>add</code> -Methode der Klasse <code>CocoaMenuComponent</code>	106
5.7 Umsetzung der <code>accept</code> -Methode unter Verwendung eines <code>CNAcceptors</code>	107
5.8 Implementierung der <i>Template</i> -basierten Klasse <code>CNAcceptor</code>	108
5.9 Instanziierung des <code>CNAcceptors</code> in dem Konstruktor der Klasse <code>CocoaMenu</code>	109
5.10 Implementierung der <i>Template</i> -basierten Klasse <code>CNAcceptorImp</code>	110
5.11 Verwendung des <code>CNComposers</code> in der Klasse <code>CNComposable</code>	113
5.12 Implementierung der Klasse <code>CNVisitingComposer</code>	113
5.13 Erstellen eines Knotens zur Repräsentation einer zusammensetzbaren Menüleiste im Kontext der Hierarchie.	114
5.14 Implementierung der Klasse <code>CNNullMatcher</code>	116

5.15 Implementierung der Klasse <code>CNTypematcher</code>	117
5.16 Implementierung der Klasse <code>CocoaMenuIdentifier</code>	117
5.17 Implementierung der Klasse <code>CocoaMenuTagMatcher</code>	119
5.18 Implementierung der Klasse <code>CNVisitingMatcher</code>	121
5.19 Implementierung der Klasse <code>CocoaTagIdentifier</code>	123
5.20 Verwendung mehrerer <code>CNAcceptors</code> in der Klasse <code>CocoaMenu</code>	125
6.1 Implementierung der Klasse <code>ShellComposingVisitor</code>	151
6.2 Implementierung der Klasse <code>ShellTypeIdentifier</code>	152
6.3 Implementierung der Klasse <code>ShellTypeIdentifier</code>	153

Abkürzungsverzeichnis

API Application Programming Interface

DOC Dependent-On Component

GUI Graphical User Interface

SUT System Under Test

STL Stereolithography

TDD Test-Driven Development

KAPITEL 1

Einleitung

Der ständige Wandel in den Spezifikationen von Software ruft das Bedürfnis hervor, flexible und wandlungsfähige Systeme umzusetzen. Dabei erschwert die Komplexität der Problemdomäne zusätzlich den Entwurfsprozess. Zur Unterstützung während der Umsetzung der Software existieren Entwurfsregeln wie zum Beispiel die Metriken Kohäsion und Kopplung sowie die Prinzipien des objektorientierten Designs, an denen sich Entwickler orientieren können, um hohe Flexibilität und Anpassungsfähigkeit zu erreichen. In der konventionellen Softwareentwicklung werden diese Entwurfsregeln verwendet, um während einer ausgiebigen Planungsphase vorausschauend ein Design zu entwerfen, das sowohl die gegenwärtig geforderten Spezifikationen berücksichtigt als auch zukünftig erwartete Funktionalität ermöglicht. Da es nicht immer möglich ist, aus dem gegenwärtigen Wissensstand spätere Anforderungen an die Software zuverlässig zu antizipieren, entstehen häufig fehlerhafte Designstrukturen. Zum Beispiel kann das umgesetzte Design aufgrund falscher Annahmen die Erweiterung des Systems um neue Funktionalität zu einem späteren Zeitpunkt hemmen. Zugleich wird auch das gegenwärtige Design der Software unnötig verkompliziert und das System verliert an Verständlichkeit. Je wei-

1. Einleitung

ter in der Planungsphase versucht wird, sich an zukünftige Anforderungen anzupassen, desto größer ist die auch die Wahrscheinlichkeit für ein fehlerhaftes Design.

Im Gegensatz zu der konventionellen Softwareentwicklung stehen agile Ansätze wie zum Beispiel *Extreme Programming* oder *Scrum*, in denen das System iterativ und inkrementell vorangetrieben wird. Durch ihre kurzen Entwurfszyklen unterstützen sie dabei, zukünftige Eventualitäten zu vernachlässigen und das Design des Produkts auf die momentanen Anforderungen anzupassen. Jeder Entwurfszyklus bringt Feedback hervor, mit dessen Hilfe die Qualität des Produkts gemessen und verbessert werden kann. Durch das agile Vorgehen ist das System in jedem Entwurfszyklus gezwungen, sich an neue Randbedingungen anzupassen. Infolgedessen entsteht durch diese Methoden inhärent flexible und anpassungsfähige Software.

Eine agile Praktik für die Implementierung von Quelltext bietet die testgetriebene Entwicklung nach Beck (2003). Sie stützt sich auf die vorgezogene Implementierung von *Unit Tests*, durch die jeder Entwurfszyklus Feedback hinsichtlich der Fehlerfreiheit und des Designs der Software auf Ebene des Quelltexts erzeugt. Dieses Vorgehen unterstützt auch das Einhalten der Entwurfsregeln und bringt deshalb qualitativ hochwertige Softwareprodukte hervor. Ein weiterer Vorteil ist die vollständige Dokumentation, die sich als Nebenprodukt der testgetriebenen Entwicklung ergibt, wodurch einerseits das Arbeiten im Team erleichtert wird, andererseits das Ausscheiden einzelner Entwickler kompensiert werden kann. Trotz dieser Vorteile findet die testgetriebene Entwicklung in der Praxis bisher nur sehr selten Anwendung.

Diese Arbeit zeigt die Anwendung der testgetriebenen Entwicklung. Sie veranschaulicht die Vorgehensweise sowie die positive Wirkung dieser Praktik anhand eines ausreichend komplexen Anwendungsbeispiels, um ihre Effektivität zu verdeutlichen und sie für die Verwendung in der Praxis zu empfehlen. Das Anwendungsbeispiel beinhaltet die Implementierung des Frameworks CrossNative, das den Entwurf von flexiblen und wandlungsfähigen grafischen Oberflächen ermöglicht. Es unterstützt ebenfalls den Austausch des verwendeten GUI-Frameworks und erlaubt somit Applikationen, die auf unterschiedlichen Betriebssystemen über native Oberflächen verfügen. Daraus wird der Name CrossNative ableitet.

Zu Beginn dieser Arbeit werden die Grundlagen der testgetriebenen Entwicklung nach Beck (2003) erläutert. Hierbei werden neben der Vorgehensweise auch die positiven Einflüsse dieser Entwurfsmethode auf die Qualität der Software sowie den Entwurfsprozess dargelegt. Anschließend erfolgt die Konzeption von CrossNative, in der die gewünschte Funktionsweise des Frameworks erarbeitet wird. Danach wird die testgetriebene Entwicklung angewendet, um CrossNative entsprechend der zuvor definierten Anforderungen umzusetzen. Dabei wird die Vorgehensweise dieser Entwurfsmethode dokumentiert, ihre Wirkung diskutiert und anhand von Abbildungen und Quelltext veranschaulicht. Zunächst wird der Fokus dabei auf eine einzelne zu testende Komponente gelegt. Als Beispiel dafür wird der Kompositionsalgorithmus von CrossNative verwendet, der grafischen Oberflächen aus einzelnen Bestandteilen zusammensetzt. Nachfolgend wird die Betrachtung auf das gesamte Framework ausgeweitet, um den Einfluss der testgetriebenen Entwicklung auf die Entstehung komplexer Klassensysteme zu verdeutlichen. Abschließend erfolgt die Anwendung von CrossNative zur Demonstration der Flexibilität und der leichten Anwendbarkeit der entstandenen Software. Dafür wird eine grafische Oberfläche für die Generierung und Visualisierung von diskreten Berechnungsgittern als Grundlage für Strömungssimulationen umgesetzt. Die Generierung der Gitter basiert dabei auf der Implementierung von Peters (2016).

Für das Verständnis dieser Arbeit werden grundlegende Kenntnisse der objektorientierten Programmierung sowie der Prinzipien und Paradigmen des objektorientierten Softwaredesigns vorausgesetzt.

KAPITEL 2

Grundlagen der testgetriebenen Entwicklung

Software kann auf unterschiedlichen Abstraktionsebenen betrachtet werden: Die Software als Ganzes, die Hauptbestandteile der Architektur, Module, Klassen sowie Methoden und Attribute. Unabhängig von der betrachteten Ebene wird die Software immer durch den zu Grunde liegenden Quelltext repräsentiert, dessen Qualität die Gesamtqualität der Software bestimmt. Daher gilt in der Softwareentwicklung neben der Fehlerfreiheit, die von erfolgreichen Softwareprojekten vorausgesetzt wird, die Umsetzung von hochwertigem Code (*Clean Code*, Martin, 2009) als wichtigste Forderung. Darunter verstehen viele renommierte Softwareentwickler¹ Quelltext, der sich durch hohe Lesbarkeit, Verständlichkeit und Redundanzfreiheit auszeichnet. Grund dafür ist der ständige Änderungsbedarf, dem Software unterliegt. Die Erweiterung bzw. die Modifikation eines Systems erfordert es, neue Strukturen in die vorhandene Software zu integrieren. Dieser Prozess erleichtert sich, wenn die umgebende Software durch seine Einfachheit und Direktheit leicht verständlich ist. Des Weiteren birgt jede Veränderung die Gefahr bereits implementierte Funktionalität unbemerkt zu beschädigen. Eine Erweiterung oder Modifikation kann daher zu einem fehlerhaften Zustand des Systems führen.

¹Booch, G.; Cunningham, W.; Feathers, M.; Jeffries, R.; Thomas, D.; Stroustrup, B. (Martin, 2009)

2. Grundlagen der testgetriebenen Entwicklung

»*Clean code that works, in Ron Jeffries' pithy phrase, is the goal of Test-Driven Development (TDD).*«
(Beck, 2003)

Nach Ron Jeffries Aussage adressiert der testgetriebene Entwicklungsansatz (Beck, 2003) die Umsetzung von sowohl qualitativ hochwertigem als auch fehlerfreiem Code. Die meisten der Heuristiken und Prinzipien des objektorientierten Entwurfs lassen sich auf diese fundamentale Forderung nach hochwertigem Quelltext zurückführen (Martin, 2009). Die bekanntesten darunter sind die Metriken Kohäsion und Kopplung (Yourdon et al., 1979) sowie die SOLID-Prinzipien (Martin et al., 2008) des objektorientierten Designs. Das Vorgehen und die Mechanismen dieser Methode führen zur Einhaltung der Heuristiken und Prinzipien, wodurch sich die hohe Qualität auf Ebene des Quelltextes ergibt (Beck, 2003; Oshero, 2014). Die Fehlerfreiheit des Systems wird durch ein Sicherheitsnetz von automatisierten Tests gewährleistet. Somit lässt *Test-Driven Development* (TDD) ein System entstehen, das mit geringem Risiko leicht modifiziert und erweitert werden kann.

Das folgende Kapitel beschreibt die Grundlagen des testgetriebenen Entwicklungsansatzes. Es beginnt mit der Erläuterung des Konzepts von TDD und hebt die Wichtigkeit des Feedbacks während des Entwurfsprozesses hervor. Anschließend wird der dreiphasige Entwicklungskreislauf geschildert. Des Weiteren setzt sich dieses Kapitel mit den Grundlagen des *Unit Testings* auseinander. In diesem Zusammenhang werden die Definition sowie der Aufbau und die Eigenschaften eines guten *Unit Tests* vorgestellt. In einem nächsten Schritt wird die Isolation der Testfälle durch die Verwendung von *Test Doubles* diskutiert. Abschließend wird die Funktionsweise des verwendeten *Testing Frameworks* GoogleMock vorgestellt.

2.1 Konzept der testgetriebenen Entwicklung

Für die Gewährleistung der Funktionalität von Software werden Tests formuliert, die Feedback über die Fehlerfreiheit des Systems geben. Nach Meszaros (2006) erfolgt der Vorgang des Testens in der traditionellen Softwareentwicklung im Anschluss an die Produktion. Das Feedback wird daher erst sehr spät in den Entwicklungsprozess eingebracht. Die nachträgliche Beseitigung eines Defekts kann unter Umständen eine teilweise Neukonzeptionierung des Systems bedeuten und ist daher mit hohem Aufwand verbunden. Infolgedessen vermindert sich der Wert des Feedbacks, da es nur als Maß der Fehlerfreiheit von Software dient und nicht aktiv in den Entwicklungsprozess eingreift (Meszaros, 2006). Damit das Feedback in den Entwurf der Software einfließt, muss es dem Entwickler schon während der Implementierung kontinuierlich bereitgestellt werden. Je häufiger es dabei generiert wird, desto schneller kann der Entwickler auf Defekte und Fehlkonzeptionen reagieren (Freeman et al., 2009; Meszaros, 2006). Die konstante Bereitstellung von Feedback benötigt einen sich wiederholenden Ablauf von Aktivitäten, der bei jedem Durchlauf neue Rückmeldungen erzeugt. In der Softwareentwicklung wird das durch eine inkrementelle Erweiterung des Systems erreicht. Anstatt die gesamte Software erst nach Fertigstellung auf Korrektheit zu überprüfen, sieht der inkrementelle Ansatz die Implementierung und das Testen einzelner Funktionalitäten vor.

Eine besondere Art des Testens liefert der in TDD verwendete *Test-First*-Ansatz. Nach Beck (2003) werden demnach die Tests der Software vor dem Produktionscode formuliert. Das iterative Schreiben eines vorgezogenen Tests gibt dem Entwickler die Möglichkeit, die konkrete Spezifikation der zu erweiternden Funktionalität herauszufinden. Jede Iteration spiegelt dabei bestimmte Designentscheidungen wider (Martin et al., 2008). Durch das Feedback des Tests wird der Entwickler in seinen Entscheidungen unterstützt. Tests, die nach dem *Test-First*-Ansatz definiert werden, greifen aktiv in den Entwurfsprozess ein und treiben die Entwicklung des Produktionscodes an. Wichtige Designentscheidungen werden daher schon vor der Umsetzung des Produktionscodes getroffen. Aufgrund der geänderten Reihenfolge befasst sich die Implementierung einer Testmethode mehr mit dem Design der Software als mit der Verifizierung des Systemverhaltens (Martin et al., 2008).

»TDD is an awareness of the gap between decision and feedback during programming, and techniques to control that gap.«
(Beck, 2003)

Das Konzept von TDD ist die Verbesserung des Entwurfsprozesses durch das Bereitstellen von schnellem und häufigem Feedback. Die inkrementelle Entwicklung der Software sorgt für Produktionscode, der feingranular durch Tests verifiziert wird. Das häufige Ausführen aller vorhandenen Tests ermöglicht es dem Entwickler Regressionen, die durch eine Modifikation der Software entstehen, schnell zu entdecken und zu beseitigen (Oshevirov, 2014). Die Fehler in dem Produktionscode lassen sich anhand der Testergebnisse erkennen, die dem Entwickler zur leichteren Auswertung in visueller Form präsentiert werden. Dabei spiegelt Grün das Bestehen und Rot das Fehlschlagen eines Tests wider. Zusätzlich geben Tests durch den *Test-First*-Ansatz frühes Feedback über das Design der Software. Die Kombination der inkrementellen Entwicklung mit dem *Test-First*-Ansatz, erzeugt jederzeit ausreichend Feedback hinsichtlich der Fehlerfreiheit und des Designs der Software. Dadurch können Entscheidungen und Modifikationen unmittelbar evaluiert und somit das Vertrauen in die Tests sowie in den Produktionscode gesteigert werden. Die Anwendung von TDD erleichtert es dem Entwickler deshalb das von Ron Jeffries geforderte Ziel (siehe Seite 6) zu erreichen (Beck, 2003).

2.2 Vorgehensweise der testgetriebenen Entwicklung

Nach Beck (2003) geht die testgetriebene Entwicklung aus der Anwendung der nachfolgenden Regeln hervor:

- Implementiere nur neuen Produktionscode, wenn ein automatisierter Test existiert, der fehlschlägt.
- Entferne Redundanzen.

Aus den Regeln lassen sich die drei in Abbildung 2.1 dargestellten Phasen Rot, Grün und Refactoring der testgetriebenen Entwicklung ableiten. Das inkrementelle Wesen von

TDD formt aus diesen Phasen einen Kreislauf, der sich während des Entwurfs im Abstand von einigen Minuten wiederholt. Ein Durchlauf entspricht dabei der Erweiterung des Systems um eine neue Funktionalität (Beck, 2003). Zugleich entsteht mit jeder neuen Abfolge der drei Phasen eine Testmethode, die die erweiterte Funktionalität des Produktionscodes sicherstellt.

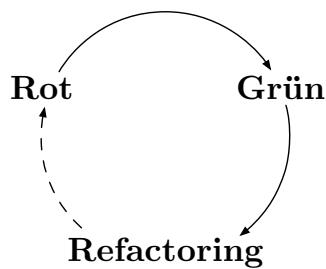


Abbildung 2.1: Die Phasen der testgetriebenen Entwicklung nach Beck (2003) in Anlehnung an Freeman et al. (2009).

Rot. Die erste Phase des Kreislaufs befasst sich mit der Implementierung eines fehlgeschlagenen Tests, der die zu erweiternde Funktionalität beschreibt (Beck, 2003). Die Aufgabe des Entwicklers beläuft sich dabei auf die Konzeption der neuen Funktionalität. Im Einzelnen gehören dazu das Auslösen der Operation, die Identifikation von beteiligten Elementen und das Gestalten von Schnittstellen zur Kommunikation (Beck, 2003). Der Entwickler implementiert den Test so wie er sich die Funktionsweise der neuen Funktionalität vorstellt. Dabei kann er auch Typen und Methoden verwenden, die in dem Produktionscode noch nicht existieren (Freeman et al., 2009; Osherove, 2014). Für den Moment der Implementierung ignoriert er, dass der Test unter Umständen nicht kompiliert. Damit der Test ausführbar wird, implementiert der Entwickler anschließend die von dem Test geforderte Infrastruktur, die die verwendeten Schnittstellen, Klassen und Methoden umfasst (Freeman et al., 2009). Danach erfolgt die Ausführung der Tests, um das Fehlschlagen der neuen Testmethode zu überprüfen. Dieser Schritt ist notwendig, um die Korrektheit des Tests selbst sicherzustellen. Schlägt der Test nicht auf die erwartete Weise fehl oder besteht sogar, lässt das auf eine fehlerhafte Implementierung schließen (Freeman et al., 2009; Osherove, 2014). Im Rahmen dieser Arbeit wird die erstmalige Ausführung eines Test zur Überprüfung seiner Fehlerfreiheit als *Failure Run* bezeichnet.

2. Grundlagen der testgetriebenen Entwicklung

Grün. In der zweiten Phase des testgetriebenen Ansatzes muss der Entwickler den zuvor definierten Test auf dem schnellstmöglichen Weg zum Bestehen bringen. Dafür ergänzt er die in Phase Rot geschaffene Infrastruktur mit dem entsprechenden Verhalten. Die Erweiterung des Produktionscodes darf dabei nur so gering und simpel wie möglich ausfallen. Nach Beck (2003) dominiert die Forderung das Ergebnis des Tests schnellstens von Rot auf Grün umschlagen zu lassen jegliche Qualitätsanforderungen an den Quelltext und entschuldigt zunächst ein schlechtes Design. Im Vordergrund dieser Phase steht nicht die Sauberkeit des Produktionscodes, sondern das fehlerfreie Verhalten der Software gemäß den durch die Tests beschriebenen Spezifikationen (Beck, 2003).

Refactoring². Im letzten Schritt des Kreislaufs wird der Produktionscode zugunsten seiner Qualität modifiziert ohne dabei das Ergebnis des Tests, respektive die Funktionalität, zu verändern. Durch Umstrukturierungen und das Entfernen von Redundanzen verbessert sich das Design des Quelltexts, da sich sowohl die Lesbarkeit als auch die Wartbarkeit erhöht (Beck, 2003; Oshero, 2014). Sind keine Veränderungen des Quelltextes notwendig, kann diese Phase des Zyklus entfallen.

Jede der einzelnen Phasen beinhaltet eine wichtige Aufgabe während des Entwurfszyklus. Nur zusammen und in der korrekten Reihenfolge erzielt TDD die gewünschten Auswirkungen auf den Entwicklungsprozess. Der fehlschlagende Test, der in Phase Rot entsteht, erzeugt nach Beck (2003) Fortschritt, da durch ihn die zu erweiternde Funktionalität als konkrete Spezifikation vorliegt. Aus einer vagen Vermutung der Funktionsweise entwickelt sich die klare Absicht des Designers. Das Fehlschlagen des Tests verdeutlicht, dass sich das System nicht mehr konform der nun geänderten Spezifikation verhält. Diesen Defekt gilt es zu beheben. Nach Ron Jeffries ist dafür hochwertiger und fehlerfreier Produktionscode umzusetzen. Diese Forderung wird in den zwei folgenden Phasen nach dem Prinzip *Divide-and-Conquer*³ erfüllt. Zunächst löst Phase Grün das »*that works*« der Aussage, während das anschließende Refactoring den funktionierenden Quelltext in »*clean code*« umwandelt (Beck, 2003).

²Refactoring bezeichnet die Modifikation von bereits implementiertem Code zur Verbesserung seines Designs. Während des Refactorings wird das Verhalten des Codes nicht verändert (Fowler, 1999).

³Divide-And-Conquer beschreibt einen Ansatz zur Beherrschung von komplexen Aufgaben durch Dekomposition des Gesamtproblems in mehrere kleine Teilprobleme (Booch, 2007).

2.3 Unit Testing

In der Softwareentwicklung wird zwischen den drei Testarten *Acceptance Tests*, *Integration Tests* und *Unit Tests* unterschieden, die das System nach Freeman et al. (2009) auf unterschiedlichen Ebenen testen. Maßgeblich unterscheiden sich die Testarten anhand der Größe des zu testenden Systems, das auch als *System Under Test* (SUT) bezeichnet wird. Während *Acceptance Tests* die Funktion eines gesamten *Features* verifizieren, umfassen nach Osheroove (2014) *Integration Tests* und *Unit Tests* nur einzelne Einheiten der Software. Bei *Acceptance Tests* handelt es sich deswegen um *End-to-End-Tests* (Freeman et al., 2009), die das System als *Blackbox* verwenden. *Integration Tests* und *Unit Tests* hingegen geben Aufschluss über die interne Struktur der Software (Martin et al., 2008), die die Funktionsweise eines größeren Systems (z.B. *Feature*) beschreibt. Aufgrund der Größe des getesteten Systems entsteht bei *Acceptance Tests* weniger und seltener Feedback als bei *Integration Tests* und *Unit Tests*.

In TDD sollen Tests auch Feedback über die Fehlerfreiheit des Systems geben. Zuverlässiges und schnelles Feedback benötigen sowohl konsistente Testergebnisse als auch eine schnelle Ausführbarkeit. Für die Ausführung des zu testenden Verhaltens benötigt das SUT unter Umständen externe Abhängigkeiten, über die der Test keine Kontrolle besitzt. Die Ergebnisse des Tests hängen daher nicht mehr nur von dem SUT selbst, sondern auch von dem Verhalten anderer Systemkomponenten ab. Dadurch können die Konsistenz und die Geschwindigkeit der Ausführung nachteilig beeinflusst werden. Verwendet ein Test beispielsweise die echte Systemzeit zur Überprüfung seiner Annahmen, verliert er seine Konsistenz aufgrund der sich ständig ändernden Zeit. Weiterhin verlängern z.B. Datenbanken die Ausführungsgeschwindigkeit der Tests, da ihre Lese- und Schreiboperationen zeitintensiv sind.

Integration Tests benutzen die externen Abhängigkeiten, um das SUT im Zusammenhang mit anderen Systemkomponenten zu überprüfen. Aufgrund ihrer mangelnden Konsistenz und ihrer langsamen Ausführung sind sie für TDD nicht geeignet. Der Unterschied zwischen *Integration Tests* und *Unit Tests* liegt in der isolierten Betrachtung des SUT, durch die der Einfluss anderer Systemkomponenten vermieden wird (Osheroove, 2014). Abschnitt 2.4 beschreibt die Schwierigkeiten der Verwendung von externen

Abhängigkeiten in Tests näher und zeigt, wie die Isolation der Testfälle erreicht werden kann. Da sich diese Arbeit mit der testgetriebenen Entwicklung unter Verwendung von *Unit Test* befasst, werden *Acceptance Tests* und *Integration Tests* nicht näher erläutert.

2.3.1 Definition des Unit Tests

Es gelten die von Oshero (2014) formulierten Definitionen im Zusammenhang mit *Unit Tests*.

Definition 2.3.1. Unit Test: Ein *Unit Test* ist automatisierter Quelltext, der die zu testende *Unit of Work* (siehe Definition 2.3.2) auslöst, um anschließend Annahmen über ein einzelnes Endergebnis (siehe Definition 2.3.3) zu überprüfen. Er ist leicht zu formulieren und schnell ausführbar. *Unit Tests* sind vertrauenswürdig, lesbar und wartbar. Sie sind konsistent in ihren Ergebnissen, solange sich der durch sie abgedeckte Produktionscode nicht verändert.

Definition 2.3.2. Unit of Work: Eine *Unit of Work* bezeichnet die Gesamtheit aller Operationen, die durch den Aufruf einer öffentlichen Methode des getesteten Systems ausgelöst werden sowie ein einzelnes Endergebnis dieses Systems. Sie kann einzelne Methoden bis hin zu mehreren Klassen überspannen.

Definition 2.3.3. Endergebnis: Ein Endergebnis kann wahrgenommen werden ohne den internen Zustand des getesteten Systems zu betrachten. Dazu benötigt man allein das öffentliche *Application Programming Interface* (API) und das Verhalten des Systems. Endergebnisse liegen in den drei nachfolgend beschriebenen Typen vor:

- **value-based** – Die aufgerufene Methode des getesteten Systems gibt einen Wert zurück.
- **state-based** – Der Aufruf der Methode erzeugt eine auffällige Veränderung des getesteten Systems bezüglich seines Zustands oder Verhaltens, die allein über die öffentliche API erfasst werden kann.
- **interaction-based** – Die *Unit of Work* löst Verhalten eines *third-party*-Systems aus, über das der Test keine Kontrolle hat und das entweder keinen Wert zurück gibt oder dessen Rückgabewert von dem getesteten System ignoriert wird.

2.3.2 Aufbau eines Unit Tests

Listing 2.1 zeigt beispielhaft die Implementierung eines *Unit Tests* unter Verwendung der Programmiersprache C++ und des Frameworks GoogleMock (siehe Abschnitt 2.6). Bei dem getesteten System handelt es sich um die Klasse **SimpleParser**. Sie summiert in ihrer Methode `parseAndSum` die einzelnen durch Komma getrennten Zahlen in einer übergebenen Zeichenkette (`string`) auf und gibt sie als ganze Zahl (`int`) zurück. Der dargestellte Testfall definiert das Verhalten des SUT, für den Fall, dass ein leerer `string` übergeben wird. Anstatt eine Summe zu bilden, liefert der **SimpleParser** den Wert 0. Der Aufbau eines Tests gliedert sich, wie in Listing 2.1 gezeigt, in die drei Abschnitte *Arrange*, *Act* und *Assert* (Beck, 2003).

```

1 TEST(SimpleParserTests, ParseAndSum_EmptyString_ShouldReturnZero) {
2     //Arrange
3     SimpleParser sut = SimpleParser();
4     std::string empty;
5
6     //Act
7     int actual = sut.parseAndSum(empty);
8
9     //Assert
10    int expected = 0;
11    EXPECT_EQ(expected, actual);
12 }
```

Listing 2.1: Beispiel eines *Unit Tests* unter Verwendung des Frameworks GoogleMock in Anlehnung an Osheroove (2014).

Arrange. Im ersten Abschnitt des Tests findet die Instanziierung des SUT sowie aller benötigten Objekte statt, die die *Unit of Work* für die Ausführung benötigt. Das SUT wird in einen von dem Test definierten Ausgangszustand versetzt.

Act. Das Auslösen der *Unit of Work* in der Testmethode bezeichnet man als das *Act* des Tests. Hierbei handelt es sich nach Definition 2.3.2 um den Aufruf einer öffentlichen Methode des SUT.

Assert. Abgeschlossen wird jeder Test mit der Überprüfung der Endergebnisse der *Unit of Work* im *Assert* des Tests.

2.3.3 Eigenschaften eines guten Unit Tests

Das Vorgehen des testgetriebenen Entwicklungsansatzes allein ist keine Gewährleistung für hohe Qualität des Endprodukts. Um hochwertigen und fehlerfreien Produktionscode sicherzustellen, müssen die *Unit Tests* gewisse Eigenschaften aufweisen. Wie in Definition 2.3.1 beschrieben, zeichnen sich gute *Unit Tests* nach Osherove (2014) durch Vertrauenswürdigkeit, Wartbarkeit und Lesbarkeit aus.

Vertrauenswürdigkeit

Die Vertrauenswürdigkeit hinsichtlich der Fehlerfreiheit der Software hängt von der Aussagekraft der Tests ab. Ein wichtiger Faktor für aussagekräftige Tests ist die zuvor beschriebene Konsistenz der *Unit Tests*, die durch die isolierte Betrachtung des SUT erreicht wird. Bei mehrfacher Ausführung darf das Resultat eines Tests nicht variieren, wenn nicht auch der durch ihn abgedeckte Produktionscode eine Änderung erfährt (Osherove, 2014). Weder die Reihenfolge der Ausführung noch die Ergebnisse anderer Tests dürfen Auswirkungen auf das Endresultat hervorrufen. Daher müssen Tests unabhängig voneinander implementiert werden (Beck, 2003).

Nach der Implementierung eines Tests kommt der Entwickler häufig nur noch mit dem Namen der Testmethode in Kontakt. Seine Aussagekraft ist daher von hoher Bedeutung. Allein anhand des Namens sollte der Entwickler erkennen können, welchen Zweck der Test erfüllt. Tests, die zu viele Annahmen bzw. Endergebnisse zugleich überprüfen, gehen mehreren Interessen nach. Ein aussagekräftiger Name wird daher schwer zu finden sein. Schlägt ein solcher Test fehl, liegt dem Entwickler lediglich die Information vor, dass sich das System fehlerhaft verhält. Der Name des Tests liefert jedoch nicht den Grund des Versagens. Um aussagekräftige Namen für die Testmethoden finden zu können, sollte jeder Test nicht mehr als ein Endergebnis nach Definition 2.3.3 überprüfen (Osherove, 2014). Des Weiteren senkt zu viel Logik wie z.B. `switch`- oder `if/else`-Anweisungen

sowie Schleifen innerhalb einer Testmethode das Vertrauen in den Test, da das Fehlerpotential ansteigt (Osheroove, 2014).

Wartbarkeit

Wie in Abbildung 2.2 dargestellt, löst der Quelltext des Tests den Produktionscode des SUT aus. Der Testcode hängt somit von dem Produktionscode ab. Da sich der Produktionscode während der Entwicklung der Software kontinuierlich verändert, befindet sich auch der Testcode in ständigem Wandel. Wird bei der Implementierung wenig Wert auf die Wartbarkeit gelegt, können kleine Änderungen im Produktionscode hohen Anpassungsbedarf in den Tests auslösen und dadurch den Zeitplan des Projekts deutlich beeinflussen. Zu hoher Aufwand für die Änderung von Tests kann daher dazu führen, dass der Entwickler die Wartung der Tests einstellt. Infolgedessen verlieren diese Tests an Aussagekraft über die Fehlerfreiheit des Systems und somit auch an Vertrauenswürdigkeit (Osheroove, 2014). Für eine hohe Wartbarkeit sind die Abhängigkeiten zu dem Produktionscode nach Osheroove (2014) so gering wie möglich zu halten. Das Auflösen der *Unit of Work* und das Überprüfen der Endergebnisse geschieht nach Definition 2.3.2 und 2.3.3 nur über die öffentliche API des SUT. Welche Implementierungsdetails zum Bestehen des Tests führen, ist nicht von Bedeutung. Durch das Testen von privaten und geschützten Methoden entsteht eine Überspezifikation, die die interne Struktur des SUT vorgibt. Auch wenn eine Änderung der internen Implementierung das Verhalten des Systems nicht variiert, müssen Tests aufgrund ihrer hohen Abhängigkeiten zu privatem Verhalten angepasst werden. Daher gilt es das Testen der Implementierungsdetails zu vermeiden (Osheroove, 2014). Auch Abhängigkeiten der Tests untereinander wirken sich negativ auf die Wartbarkeit aus. Bedingen sich Tests untereinander, kann die Modifikation einer Testmethode die Anpassung einer anderen erzwingen. Deswegen sollte auch aus Sicht der Wartbarkeit die Isolation der Testfälle sichergestellt werden. Weiterhin sind Redundanzen zu entfernen, da sie einen hohen Wartungsaufwand bedeuten und Potential für Fehler bieten. Sich wiederholende Anweisungen können z.B. in einer Helper-Methode gekapselt werden (Osheroove, 2014).

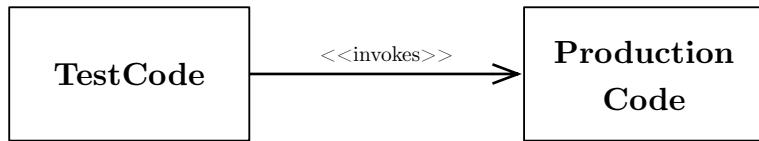


Abbildung 2.2: Abhangigkeit des Testcodes zu dem Produktionscode in Anlehnung an Martin et al. (2008).

Lesbarkeit

Neben der Verifizierung des Systemverhaltens dienen Tests nach Martin et al. (2008) Zusatzlich als Dokumentation der Software. Sie enthalten beispielhafte Anwendungen des Produktionscodes und zeigen dem Leser somit die Funktionsweise des Systems. Eine hohe Lesbarkeit fordert das Verstandnis des Entwicklers fur den Inhalt der Tests und lsst eine anschauliche Dokumentation entstehen. Nach Osherove (2014) wirkt sich die Lesbarkeit auch auf die Vertrauenswurdigkeit der Tests aus. Ist der Zweck eines Tests aufgrund seiner unleserlichen Schreibweise nur schwer zu erkennen, sinkt seine Aussagekraft. Infolgedessen vermindert sich das Vertrauen des Entwicklers in den Test. Gleichzeitig nimmt auch die Wartbarkeit ab, wenn die Implementierung der Testmethode nur schwer verandlich ist. Fur eine hohe Wartbarkeit und Vertrauenswurdigkeit der Tests sowie eine verandliche Dokumentation der Software ist daher auf eine hohe Lesbarkeit zu achten. Fur die Implementierung bedeutet das eine aussagekrftige Benennung der verwendeten Variablen und Methoden sowie eine klare Trennung der in Abschnitt 2.3.2 beschriebenen Bestandteile der Tests (Osherove, 2014).

2.4 Testen in Isolation

Nach Abschnitt 2.3 ist die isolierte Betrachtung des SUT fur gute *Unit Tests* von hoher Bedeutung, da sie sowohl die Vertrauenswurdigkeit als auch die Wartbarkeit fordert. Sie erfordert die Entkopplung der Tests von den Abhangigkeiten des SUT. Hangt das SUT wie der `SimpleParser` in Listing 2.1 nicht von weiteren Klassen ab, ergibt sich die Isolation automatisch. In solchen Fallen ist das Formulieren von Tests meistens unkompliziert. Sind jedoch weitere Klassen involviert, treten einige Schwierigkeiten auf.

Um das Verhalten des SUT sicherzustellen, stehen dem Entwickler zwei Möglichkeiten zur Verfügung. Entweder testet er das SUT im Zusammenhang mit allen Abhängigkeiten oder er strebt die Isolation des Systems an (Meszaros, 2006).

Dieser Abschnitt zeigt die Schwierigkeiten auf, die durch die Verwendung externer Komponenten in Testmethoden entstehen und erläutert Techniken zur Isolation des getesteten Systems. Komponenten, von denen das SUT abhängt, werden im Folgenden als *Dependent-On Component* (DOC) bezeichnet.

2.4.1 Indirekter Input/Output

Nachdem das SUT, wie in Abbildung 2.3 gezeigt, durch die Testmethode ausgelöst wurde, kollaboriert es während der *Unit of Work* mit seiner DOC, indem es ihre Methoden aufruft. Rückgabewerte oder mögliche *Exceptions* fließen in das Verhalten des SUT ein. Dieser Input wird nicht direkt von der Testmethode geleistet. Deswegen bezeichnet man ihn als indirekten Input. Nach der Ausführung der *Unit of Work* verifiziert die Testmethode das korrekte Verhalten des SUT. Der Einfluss des indirekten Inputs wirkt sich auf das Endergebnis der *Unit of Work* aus. Daher kann die korrekte Interaktion anhand des Endergebnisses festgestellt werden. Bei *value-based* oder *state-based* Endergebnissen geschieht das entweder über den zurückgegeben Wert der ausgelösten Methode oder über die öffentliche API des getesteten Systems. In beiden Fällen erzeugt das SUT, wie in Abbildung 2.3 dargestellt, Output, der direkt von der Testmethode erfasst werden kann (Meszaros, 2006).

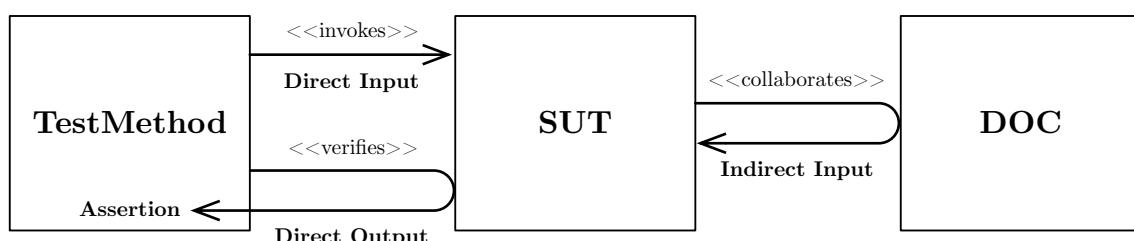


Abbildung 2.3: Ablauf eines Tests mit *value-based* oder *state-based* Endergebnis und DOC in Anlehnung an Meszaros (2006).

2. Grundlagen der testgetriebenen Entwicklung

Der indirekte Input kontrolliert durch die zurückgegebenen Werte oder ausgelösten *Exceptions* den auszuführenden Pfad der *Unit of Work*. Da die Tests das Verhalten der DOC nicht kontrollieren können, ist der indirekte Input nicht vorhersehbar (z.B. Systemzeit, siehe Abschnitt 2.3). Infolgedessen decken Tests nicht immer alle Szenarien ab. Vor allem Pfade, die auftretende Fehler verarbeiten, sind schwer auszulösen. Damit die DOC fehlerhafte Werte zurückgibt oder *Exceptions* auslöst, muss sie zuvor in einen ungültigen Zustand versetzt werden. Bleiben die schwer bzw. unmöglich auszulösenden Pfade ungetestet, führt das zu nicht verifiziertem Produktionscode (Meszaros, 2006).

Fließen die Rückgabewerte der aufgerufenen Methoden der DOC Abbildung 2.4 entsprechend nicht in das Verhalten des SUT ein, schwindet gleichzeitig die Möglichkeit die Interaktion durch ein übergeordnetes Endergebnis zu verifizieren. Es werden lediglich Seiteneffekte ausgelöst, die der Test nicht durch das Verhalten des SUT wahrnehmen kann. Dieser Fall tritt nach Osherove (2014) auf, wenn die Methoden der DOC den Rückgabetyp `void` besitzen oder das getestete System den Rückgabewert ignoriert. Das Auslösen der Seiteneffekte bezeichnet man im Zusammenhang mit *Unit Tests* als indirekten Output. Um die Interaktion des SUT mit der DOC sicherzustellen, wird der Aufruf der Methode selbst zu einem Endergebnis – *interaction-based*. Die Überprüfung müsste die DOC wie in Abbildung 2.4 entweder selbst übernehmen oder Methoden zur Verfügung stellen, die der Testmethode Informationen über den Methodenaufruf liefern. Dabei handelt es sich um testspezifisches Verhalten, das eine DOC nicht enthält. Werden Klassen im Zusammenhang getestet, ist der indirekte Output daher nicht verifizierbar. Daraus ergibt sich ungetestetes Verhalten der Software (Meszaros, 2006).

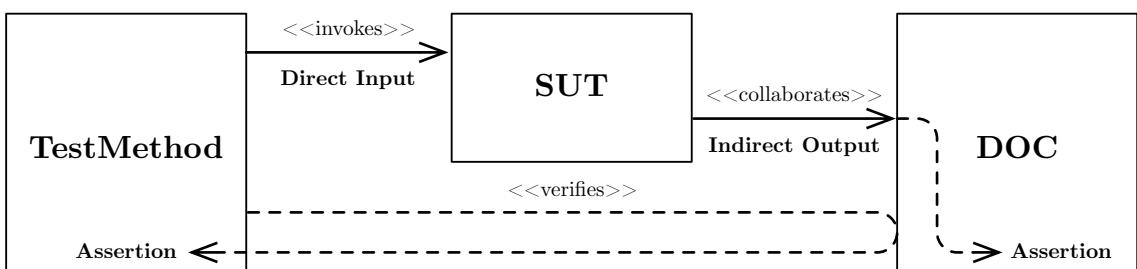


Abbildung 2.4: Ablauf eines Tests mit *interaction-based* Endergebnis und DOC in Anlehnung an Meszaros (2006).

2.4.2 Test Doubles

Die Herausforderung des Testens liegt darin, die vollständige Kontrolle über das Verhalten der externen Abhängigkeit zu erlangen, sodass die Tests sowohl den indirekten Input steuern als auch den indirekten Output verifizieren können. Dafür wird die DOC durch ein *Test Double* ersetzt. Bei einem *Test Double* handelt es sich um ein testspezifisches Objekt, das während des Tests den Platz der DOC einnimmt und ihr Verhalten simuliert. Durch testspezifische Methoden wird dem Test erlaubt, das Verhalten des *Test Doubles* zu kontrollieren. Beispielsweise kann er das *Test Double* mit einer Reihe an Werten konfigurieren, die bei dem Aufruf seiner Methoden zurückgegeben werden (siehe *Test Stub*). Dadurch steuert der Test den indirekten Input und ermöglicht es, jeden Pfad der *Unit of Work* zuverlässig auszulösen und zu testen. Des Weiteren können *Test Doubles* entweder Informationen über aufgerufene Methoden speichern und sie dem Test zur Verifizierung des indirekten Outputs zur Verfügung stellen (siehe *Test Spy*) oder den indirekten Output selbst verifizieren (siehe *Mock Object*) (Meszaros, 2006).

Um einen Austausch der echten Abhängigkeit zu ermöglichen, wird nach Martin et al. (2008) ein *Interface* zwischen dem SUT und der DOC eingefügt. Die Verwendung von Polymorphie erlaubt es dem SUT das *Test Double* zu benutzen, als wäre es eine Instanz der echten Abhängigkeit. Es entsteht ein System analog zu Abbildung 2.5. Anstatt die DOC für die Ausführung des Testfalls zu verwenden, injiziert der Test ein *Test Double* in das SUT. Das Verhalten der DOC wirkt sich somit nicht mehr auf den Testfall aus und das Testen erfolgt in vollständiger Isolation. Dadurch wird eine Entkopplung des Tests erreicht (Martin et al., 2008).

Test Doubles werden hinsichtlich ihres Zwecks nach Meszaros (2006) in die fünf folgenden Arten unterschieden:

- *Dummy Object*
- *Test Stub*
- *Test Spy*
- *Mock Object*
- *Fake Object*

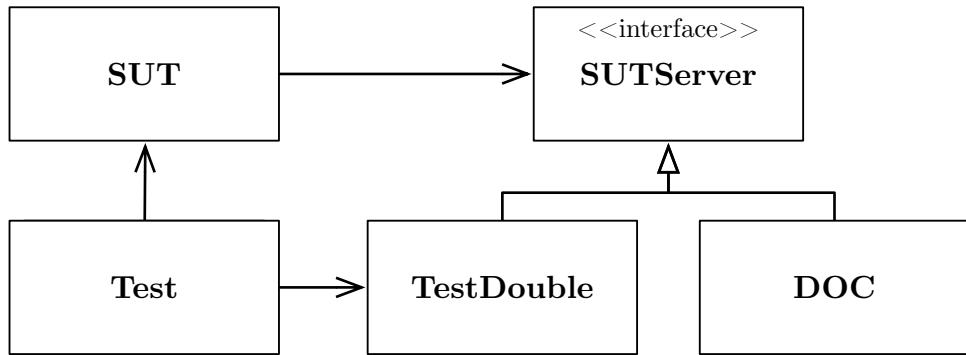


Abbildung 2.5: Entkopplung des Tests von der DOC des SUT in Anlehnung an Martin et al. (2008).

Dummy Object

Bei *Dummy Objects* handelt es sich um eine degenerierte Form der *Test Doubles*. Ihr Verhalten ist für das Endergebnis des Tests irrelevant, da ihre Methoden in der *Unit of Work* nicht aufgerufen werden. Sie existieren nur aufgrund der Infrastruktur des Produktionscodes und besitzen kein eigenes Verhalten. Die Methoden des *Dummy Objects* geben Werte zurück, die in den entsprechenden Datentypen den Nullwert repräsentieren (z.B. `0`, `false`, `nullptr`, ...).

Test Stub

Test Stubs dienen als Kontrollpunkt für indirekten Input. Sie werden weiterhin in *Responders* und *Saboteurs* unterschieden. Bei *Responders* handelt es sich um die gebräuchliche Form der *Test Stubs*. Über *setter*-Methoden erlauben sie dem Test die Rückgabewerte ihrer Methoden zu definieren. Der Test kann die *Responders* mit gültigen oder ungültigen Rückgabewerten ausstatten, die dann als indirekter Input in das Verhalten des SUT eingehen. Ein *Saboteur* hingegen löst *Exceptions* bzw. Fehler aus, um das SUT ungewöhnlichem indirekten Input auszusetzen.

Test Spy

Tests Spies dienen als Beobachtungspunkt für indirekten Output. Sie erweitern die Funktionalität des *Test Stubs*, um die Fähigkeit Methodenaufrufe zusammen mit den übergebenen Parametern aufzuzeichnen. Bei *Test Spies* erfolgt die Verifizierung des indirekten Outputs in der Testmethode nach dem Auslösen der *Unit of Work*. Diese Art des Testens wird nach Meszaros (2006) als *Procedural Behavior Verification* bezeichnet. Über *getter*-Methoden erhält der Test die gespeicherten Informationen und vergleicht sie anschließend mit erwarteten Methodenaufrufen.

Mock Object

Wie *Test Spies* dienen auch *Mock Objects* als Beobachtungspunkt für indirekten Output und geben wie *Test Stubs* zuvor definierte Werte zurück. Sie unterscheiden sich von *Test Spies*, indem sie ihre Methodenaufrufe während der *Unit of Work* mit erwarteten Methodenaufrufen selbst vergleichen, anstatt die Verifizierung der Testmethode zu überlassen. Hierbei handelt es sich nach Meszaros (2006) um *Expected Behavior Specification*. Dafür injiziert der Test dem *Mock Object* eine Reihe an erwarteten Methodenaufrufen, bevor das SUT ausgelöst wird. Durch die Verwendung gleicher *Mock Objects* in unterschiedlichen Tests kann die Logik zur Überprüfung wiederverwendet werden.

Fake Object

Im Gegensatz zu den anderen Arten der *Test Doubles* werden *Fake Objects* weder zur Steuerung des indirekten Inputs von dem Test kontrolliert noch zur Verifizierung des indirekten Outputs beobachtet. Sie ersetzen die DOC inklusive ihres Verhaltens. Die *Fake Objects* und die echten Abhängigkeiten besitzen identische Funktionalität, unterscheiden sich jedoch in der Implementierung. *Fake Objects* werden unter anderem verwendet, wenn die DOC noch nicht implementiert, zu langsam, oder nicht in der Testumgebung vorhanden ist.

2.5 Effekte der testgetriebenen Entwicklung

Wie in Abschnitt 2.1 und 2.3 beschrieben, kombiniert TDD die inkrementelle Entwicklungsmethode mit der Vorgehensweise des *Unit Testings* nach dem *Test-First*-Ansatz. Das Zusammenwirken dieser drei Aspekte löst Mechanismen aus, die die nachfolgend beschriebenen Effekte hervorrufen.

2.5.1 Verifizierung des Systemverhaltens

Im Mittelpunkt der testgetriebenen Entwicklung stehen automatisierte *Unit Tests*, die den Produktionscode hinsichtlich seiner Funktionalität überprüfen. Die Verifizierung des Systemverhaltens ist deshalb nach Martin (2009) der wohl offensichtlichste Effekt von TDD. Die Vorgabe des *Test-First*-Ansatzes (Beck, 2003) die Tests der Software vor der Implementierung des Produktionscodes zu definieren, verstärkt diesen Effekt weiter. Das Vorgehen von TDD nach Abschnitt 2.2 verbietet die Implementierung von Produktionscode, wenn nicht ein fehlschlagender Test vorliegt. Der implementierte Produktionscode darf keinen weiteren Zweck erfüllen, als den fehlschlagenden Test zu erfüllen. Dadurch entsteht eine vollständige Testabdeckung der Systemfunktionalität. Das Fehlerpotential minimiert sich und die Dichte an Defekten nimmt ab. Auch der inkrementelle Ansatz wirkt sich positiv auf die Verifizierung des Systemverhaltens aus. Die Funktionalitäten werden unabhängig voneinander getestet und implementiert. Zusammen mit der geringen Größe der getesteten Systeme in den *Unit Tests* ergibt sich eine feingranulare Teststruktur. Da jede Funktionalität durch ihre eigenen Tests verifiziert wird, gibt das Fehlschlagen eines Tests nicht nur Aufschluss über ein Fehlverhalten, sondern hilft auch bei der Lokalisierung des Defekts. Zeitintensives Debugging zur Fehlersuche wird damit vermieden (Oshero, 2014).

Die schnelle Ausführbarkeit der vorhandenen Test ermöglicht es dem Entwickler, jederzeit das Systemverhalten zu verifizieren. Das visuelle Feedback zeigt auf einen Blick, ob sich die Software verhält wie gewünscht. Dadurch sinkt die Angst vor Regressionen, die durch unachtsame Modifikationen der Software entstehen und der Entwickler wird ermutigt, Veränderungen in dem Produktionscode vorzunehmen.

2.5.2 Objektorientiertes Design

Damit die Funktionalität der Software leicht durch Tests überprüft werden kann, ist eine hohe Testbarkeit des Systems vorausgesetzt. Darunter versteht sich ein Design, das es ermöglicht die einzelnen Funktionalitäten, wie in Abschnitt 2.4.2 gezeigt, isoliert zu betrachten. Für die Entkopplung der Tests wird nach Abbildung 2.5 ein *Interface* zwischen dem SUT und den externen Abhängigkeiten eingefügt. Infolgedessen kann ein Test echte Abhängigkeiten durch testspezifische Objekte ersetzen und so die Interaktion sicherstellen. Als Nebenprodukt entkoppelt das *Interface* auch das SUT von den externen Abhängigkeiten (*Serendipitous Decoupling*, Martin et al., 2008). Die Umsetzung einer hohen Testbarkeit im Produktionscode trifft deswegen auch die Ziele des objektorientierten Designs. Aufgrund der Abstraktion werden nach Osheroove (2014) unter anderem das *Open/Closed Principle*⁴ und das *Dependency Inversion Principle*⁵ eingehalten. Das Testen von Software führt daher zu hoch kohäsiven und lose gekoppelten Systemen⁶ (Beck, 2003). Auch das *Interface Segregation Principle*⁷ wird eingehalten, da die durch die Testfälle erzwungenen *Interfaces* zumeist auf die jeweilige *Unit of Work* angepasst sind. Das SUT hängt deshalb nur von Verhalten anderer Komponenten ab, das es für seine eigene Funktionalität benötigt.

Mögliche Designfehler im Produktionscode erschweren das Testen der Software. Tests, die wie in der traditionellen Softwareentwicklung nachträglich formuliert werden, decken diese Fehler erst nach der Produktion des gesamten Systems auf. Die Implementierung von nachträglichen Tests ist deswegen meistens mit Schwierigkeiten verbunden. Aufgrund des nicht testbaren Designs besteht nach Meszaros (2006) nur eine geringe Wahrscheinlichkeit Tests für bereits bestehende Funktionalität zu formulieren ohne den Produktionscode zu verändern. Das kann zu einem hohen Anpassungsbedarf des ge-

⁴Dem Open/Closed Principle entsprechend sollen Softwareentitäten sowohl offen für Erweiterungen als auch geschlossen für Modifikationen sein (Meyer, 2009).

⁵Das Dependency Inversion Principle gibt vor Abhängigkeiten zu Implementierungsdetails zu vermeiden und stattdessen auf Abstraktionen zu basieren (Martin et al., 2008).

⁶Kohäsion und Kopplung beschreiben das Ausmaß von intermodularen Abhängigkeiten sowie der Zusammengehörigkeit von Daten, die innerhalb eines Moduls enthalten sind (Yourdon et al., 1979).

⁷Das Interface Segregation Principle gibt vor Schnittstellen so schmal wie möglich zu gestalten, um Clients der Schnittstelle vor Abhängigkeiten zu nicht verwendetem Verhalten zu schützen (Martin et al., 2008).

2. Grundlagen der testgetriebenen Entwicklung

samten Systems führen und einen hohen Aufwand bedeuten. Die Implementierung von Quelltext entlang eines zuvor definierten Tests hingegen bringt inhärent testbare Software hervor und lässt nur wenig Platz für fehlerhaftes Design (Meszaros, 2006).

Darüber hinaus unterstützen vorgezogene Tests den Entwickler in seinen Designentscheidungen. Die Art und Weise der Implementierung eines Tests impliziert das Design und die Kommunikationsstrukturen des Produktionscodes und gibt Aufschluss über beteiligte Objekte und Methodenaufrufe (Martin et al., 2008). Je einfacher dabei die Struktur des Tests, desto klarer und simpler wird auch das Design der Software. Zusätzlich wird der Entwickler nach Martin et al. (2008) durch das Verfassen von Tests in die Sichtweise eines Benutzers bzw. Aufrufers versetzt. Dadurch ist er gezwungen, sich nicht nur mit der Umsetzung der Funktionalität zu befassen sondern auch mit dem Entwurf der API des getesteten Systems.

Die Wirkung von TDD bringt Systeme hervor, die durch flexibles, verständliches und wartbares Design gekennzeichnet sind. Ein weniger offensichtlicher, aber dennoch viel wichtigerer Effekt als die Validierung des Systemverhaltens spiegelt sich deshalb nach Martin et al. (2008) in den Auswirkungen von TDD auf das Design wider.

2.5.3 Dokumentation

Eine weiterer Effekt liegt in der Dokumentation der Software, die durch die testgetriebene Entwicklung entsteht. Speziell in großen Systemen sind dem Entwickler häufig nicht die Funktionsweisen aller Mechanismen gegenwärtig. Um die Funktionalität dennoch leicht verwenden zu können, dienen Tests als ausführbares Beispiel der Anwendung. Durch beteiligte Objekte, Methodenaufrufe und das zu überprüfende Endergebnis zeigen sie dem Leser die Grenzen des getesteten Systems sowie die Spezifikationen der Funktionalität auf und drücken somit die Absicht des Entwicklers aus (Martin et al., 2008; Oshero, 2014). Verständliche und aussagekräftige Tests erleichtern die Entwicklung im Team, da Designentscheidungen und Absichten des Entwicklers durch die Dokumentation in den Tests geteilt und diskutiert werden können. Im Vergleich zu *Acceptance Tests* umfassen *Unit Tests* nur kleine Einheiten des Systems. Daher gibt die Dokumentation dem gesamten Team lückenlos Aufschluss über die interne Funktionsweise der Software.

2.5.4 Entlastung des Entwicklers

In der traditionellen Softwareentwicklung müssen die Aspekte der Fehlerfreiheit, des Designs und der Dokumentation durch den Entwickler allein beherrscht werden. In komplexen Systemen kann das dazu führen, dass er an die Grenzen seiner kognitiven Fähigkeiten stößt. TDD dient als Hilfestellung und entlastet den Entwickler während des Entwurfsprozesses. Unter der Anwendung dieser Methode beschränkt sich seine Aufgabe auf die möglichst einfache und klare Formulierung seiner Absichten in Tests. Die Anforderungen an die Qualität der Software werden durch die Mechanismen des testgetriebenen Ansatzes erfüllt. Diese Art der Programmierung bezeichnen Martin et al. (2008) als *Intentional Programming*.

2.6 GoogleMock

Zur Automatisierung von Quelltext zu Testzwecken wird in dieser Arbeit das Framework GoogleMock⁸ verwendet, das zur Familie der *xUnit Testing Frameworks*⁹ gehört. Es erlaubt das Verfassen von Testmethoden in der Sprache C++, die in der `main`-Methode eines Testprojektes analog zu Listing 2.2 automatisch erkannt und ausgeführt werden. Die Testmethoden müssen sich dafür nicht in derselben Datei befinden wie die `main`-Methode, solange die Datei der Testmethode Bestandteil des ausführenden Projekts ist.

```
#include "gmock/gmock.h"
2 int main( int argc , char **argv ) {
    ::testing::InitGoogleTest(&argc , argv );
4    return RUN_ALL_TESTS();
}
```

Listing 2.2: `main`-Methode zur Ausführung der Tests unter Verwendung des Frameworks GoogleMock.

⁸<https://github.com/google/googletest>

⁹Die Familie der xUnit Testing Frameworks umfasst alle Testing Frameworks, die auf den Konzepten von SUnit basieren. Bei SUnit handelt es sich um ein Testing Framework, das 1998 von Kent Beck in der Sprache Smalltalk entwickelt wurde.

2. Grundlagen der testgetriebenen Entwicklung

Wie jede Methode setzt sich auch die Testmethode in GoogleMock aus Kopf und Rumpf zusammen. Anders jedoch als bei normalen Methoden enthält der Methodenkopf nicht die üblichen Bestandteile wie Sichtbarkeit, Rückgabetyp und Liste der Übergabeparameter. Unter Verwendung von GoogleMock wird der Kopf, wie in Listing 2.3 gezeigt, durch das Makro `TEST` eingeleitet, dem ein Testfall und ein Testname als Parameter übergeben werden. Die einzigartige Kombination der Übergabeparameter sorgt für eine eindeutige Identifikation der Tests.

```
1 TEST(testcase, testname) {
2     // ... test body ...
3 }
```

Listing 2.3: Definition einer Testmethode mit dem `TEST`-Makro.

Benötigen mehrere Tests die gleichen Daten oder beinhalten gleiche Logik, bietet GoogleMock zur Vermeidung von Redundanzen die Implementierung von *Test Fixture Classes* (auch *Test Fixture* genannt) analog zu Listing 2.4 an. Um zu kennzeichnen, dass die Testmethode einer *Test Fixture Class* angehört, wird sie durch das Makro `TEST_F` eingeleitet. Der übergebene Testfall muss dafür mit dem Namen der *Test Fixture Class* übereinstimmen. In den Testmethoden sind alle geschützten und öffentlichen Methoden und Attribute der *Test Fixture Class* verfügbar. Redundante Logik kann deshalb in Helfer-Methoden gekapselt und während des Tests aufgerufen werden. Die Methoden `SetUp` und `TearDown` entstammen der Vererbung zu der von GoogleMock bereitgestellten Schnittstelle `Test`. Sie werden durch das Framework vor bzw. nach dem Ausführen der Testmethode aufgerufen. Für Logik und Daten, die sich im *Arrange* mehrerer Testmethoden wiederholen, wird die `setUp`-Methode der Basisklasse überschrieben, sodass sie zu Beginn eines jeden Tests den gewünschten Ausgangszustand des SUT herstellt. Die `TearDown`-Methode wird am Ende der Tests ausgelöst und ist für die Bereinigung des Speichers zuständig. Sie gibt den in der `setUp`-Methode allokierten Speicher frei. Für die Ausführung der Testmethoden instanziert GoogleMock die *Test Fixture Class* und löst die Tests aus. Damit sich mögliche Seiteneffekte eines Tests nicht auf die nachfolgenden Tests auswirken, wird jede Testmethode mit einer frischen

Instanz der *Test Fixture Class* ausgeführt. Diese Art der Verwendung der *Test Fixture Classes* wird nach Meszaros (2006) als *Transient Fresh Fixture* bezeichnet.

```

1 #include "gmock/gmock.h"
2 class TestClass : public testing::Test {
3 protected:
4     virtual void SetUp() override {
5         //some common setup code
6     }
7
8     virtual void TearDown() override {
9         //clean up memory allocated in setup
10    }
11
12    virtual void helperMethod(int number) {
13        //shared functionality across tests
14    }
15    int data; //shared data across tests
16 }
17
18 TEST_F(TestClass, testname) {
19     // ...
20     helperMethod(data);
21     // ...
22 }
```

Listing 2.4: Implementierung einer *Test Fixture Class* und Definition eines Test mit dem `TEST_F`-Makro.

Da GoogleMock den korrekten Aufruf der Methoden `SetUp` und `TearDown` übernimmt, sind Aufrufe in der Testmethode nicht nötig. Das Überschreiben dieser Methoden verbirgt daher einen wichtigen Bestandteil der Tests vor dem Leser. Den einzigen Hinweis für den Aufruf dieser Methoden liefert das Makro `TEST_F`, das Aufschluss darüber gibt, dass es sich bei dem Testfall um eine *Test Fixture Class* handelt. Dadurch wird die Verständlichkeit und die Lesbarkeit gestört. Aus diesem Grund rät Osheroove (2014) von der Verwendung dieser Methoden ab.

2. Grundlagen der testgetriebenen Entwicklung

Für die Auswertung der Tests bietet GoogleMock weitere Makros an. Sie stellen das tatsächliche Endergebnis einer *Unit of Work* erwarteten Werten oder Konditionen gegenüber. Basierend auf diesem Vergleich schlägt der Test entweder fehl oder besteht. In GoogleMock werden die Makros zur Überprüfung in zwei Arten unterteilt. Makros, die mit `ASSERT_` beginnen, erzeugen beim Fehlschlagen einen fatalen Fehler, der die Ausführung des aktuellen Tests abbricht. Weitere Annahmen des selben Tests werden somit nicht mehr überprüft. Im Gegensatz dazu führt das Fehlschlagen eines Makros, das mit `EXPECT_` beginnt, nicht zu einem Abbruch des Tests. Dadurch können mehrere fehlschlagende Annahmen überprüft werden. Zu jedem `ASSERT_` existiert in GoogleMock auch ein korrespondierendes `EXPECT_`. Die geläufigsten unter den Makros sind `EXPECT_THAT`, `EXPECT_TRUE` und `EXPECT_FALSE`. Während `EXPECT_THAT` zwei übergebene Parameter vergleicht, verifizieren die anderen beiden Makros den Wert eines boolschen Ausdrucks. Die Art und Weise des Vergleichs in dem Makro `EXPECT_THAT` wird über sogenannte Matcher bestimmt, mit denen der erwartete Wert während des Aufrufs versiehen wird. Listing 2.5 zeigt beispielhaft die Überprüfung von zwei ganzen Zahlen auf Gleichheit durch die Verwendung des Matchers `Eq`. Eine vollständige Auflistung aller Makros und Matcher ist der Dokumentation des Frameworks¹⁰ zu entnehmen.

Des Weiteren enthält GoogleMock die Funktionalität eines Isolationsframeworks und erlaubt eine leichte Implementierung von *Test Doubles* nach Abschnitt 2.4.2. In dieser Arbeit werden die *Test Doubles* vom Entwickler selbst formuliert. Aus diesem Grund wird diese Funktionalität von GoogleMock nicht näher erläutert.

```
TEST(MatcherDemonstration , MatchingForEquality) {  
    int i = 1;  
    int j = 2;  
  
    EXPECT_THAT(i , testing::Eq(j));  
}
```

Listing 2.5: Anwendung der von GoogleMock bereitgestellten Matcher in dem Makro `EXPECT_THAT`.

¹⁰<https://github.com/google/googletest/tree/master/googletest/docs>

2.7 Zusammenfassung

In diesem Kapitel werden die Grundlagen der testgetriebenen Entwicklung erläutert. Dabei wird zunächst die besondere Rolle des Feedbacks während des Entwurfsprozesses hervorgehoben. Es wird gezeigt, wie das Feedback den Entwickler dabei unterstützt, die Umsetzung des Produktionscodes zu analysieren und mögliche Fehlkonzeptionen aufzudecken. Infolgedessen trägt es dazu bei, die Qualität der Software zu steigern. Als Werkzeug für das Bereitstellen von Feedback greift der testgetriebene Entwicklungsansatz auf die Implementierung von Tests zurück. Bei Tests handelt es sich um automatisierte Methoden, die jeweils kleine Bestandteile des Produktionscodes ausführen und deren Funktionalität überprüfen. Weiterhin wird erklärt, wie die testgetriebene Entwicklung im Gegensatz zu der konventionellen Softwareentwicklung durch die Kombination des *Test-First*-Ansatzes mit dem inkrementellen Ansatz schon während des Entwurfsprozesses kontinuierlich Feedback bereitstellt. Daraus resultieren kurze Entwurfszyklen, die es erlauben frühzeitig auf Unzulänglichkeiten des Systems zu reagieren. Die Vorgehensweise in jedem Entwurfszyklus entspricht dabei dem vorgestellten dreiphasigen Ablauf der testgetriebenen Entwicklung mit den Phasen Rot, Grün und *Refactoring*.

Die Implementierung von Tests kann auf unterschiedliche Weisen geschehen. Dadurch ergeben sich verschiedene Testarten, unter denen die *Unit Tests* als eigene Art bekannt sind. Da die testgetriebene Entwicklung im Rahmen dieser Arbeit auf *Unit Tests* basiert, wird diese Testart näher beschrieben. Zu diesem Zweck wird auf die Definition, den Aufbau sowie die Qualitätsmerkmale eines guten *Unit Tests* eingegangen. Ein besonderer Aspekt dieser Testart sind ihre konsistenten Ergebnisse, die sich bei mehrfacher Ausführung der Tests nicht verändern, solange der zugrundeliegende Produktionscode keine Modifikation erfährt. Diese Konsistenz ist das Ergebnis der isolierten Betrachtung des getesteten Systems, das dadurch von dem Verhalten externer Abhängigkeiten entkoppelt wird. Zur Isolation des getesteten Systems werden die anschließend vorgestellten *Test Doubles* verwendet, die in den Testmethoden die externen Abhängigkeiten ersetzen.

Nachfolgend werden die Effekte der testgetriebenen Entwicklung auf den Produktionscode, den Entwurfsprozess sowie den Entwickler verdeutlicht. Über den offensichtlichen Effekt der Verifizierung des Systemverhaltens hinaus wirkt sich die testgetriebene Vor-

2. Grundlagen der testgetriebenen Entwicklung

gehensweise im hohen Maße positiv auf das objektorientierte Design aus. Ein Grund dafür ist das inhärent testbare Design des Produktionscodes durch die Implementierung entlang zuvor definierter *Unit Tests*. Die isolierte Betrachtung unter Verwendung von *Test Doubles* erzwingt *Interfaces* zwischen dem getesteten System und den externen Abhängigkeiten, wodurch die Heuristiken Kohäsion und Kopplung sowie die Prinzipien des objektorientierten Designs eingehalten werden. Ein weiterer Effekt der testgetriebenen Entwicklung ist die verständliche Dokumentation des Systems durch die *Unit Tests* als kleine Anwendungsbeispiele der entworfenen Software. Dadurch kann die Funktionsweise des Produktionscodes leicht im Team geteilt und diskutiert werden.

KAPITEL 3

Konzeption des Entwurfs von CrossNative

Zur Interaktion mit dem Benutzer verfügen Applikationen häufig über grafische Oberflächen (engl.: *Graphical User Interface (GUI)*), die die Daten und Ergebnisse der Problemdomäne anzeigen sowie Schaltflächen für domänenspezifische Funktionalität bereitstellen. Neben der Fehlerfreiheit und der Performance eines Programmes ist unter anderem auch der Bedienkomfort über Erfolg oder Misserfolg des Projekts entscheidend. Da der einzige Kontakt des Anwenders mit der Applikation in der Regel über die GUI der Software geschieht, trägt ihre Gestaltung maßgeblich zu einer hohen Benutzerfreundlichkeit bei. Für einen leichten Umgang mit der Applikation sollte die GUI die domänenspezifischen Daten und Ergebnisse in möglichst übersichtlicher Form darstellen und eine intuitive Bedienung erlauben, die den Workflow des Benutzers unterstützt. Eine GUI, die diesen Anforderungen genügt, gelingt meistens nicht im ersten Versuch. Erfahrungsgemäß durchläuft der Entwurf einige Iterationszyklen, bis sich die optimale Gestaltung ergibt. Eine weitere Herausforderung besteht darin, der sich kontinuierlich ändernden Funktionalität der Software gerecht zu werden. Änderungen der Spezifikationen können Anpassungen in dem System der grafischen Oberfläche erzwingen. Somit

3. Konzeption des Entwurfs von CrossNative

rufen Modifikationen und Erweiterungen der Problemdomäne auch Modifikationen und Erweiterungen in der Gestaltung der grafischen Oberfläche hervor. Um einerseits eine hohe Benutzerfreundlichkeit zu erreichen und sich dennoch an die wandelnde Funktionalität anpassen zu können, sollte die GUI durch ein flexibles und wandlungsfähiges System repräsentiert werden, das den iterativen Entwurfsprozess erleichtert und schnell Änderungen in der Gestaltung der Oberfläche zulässt.

Für die Umsetzung von Benutzeroberflächen stehen GUI-Frameworks zur Verfügung, die allgemeingültige Implementierungen für grafische Elemente, wie z.B. Fenster oder Menüs, bereitstellen. Ein Nachteil dieser Frameworks ergibt sich aus ihren Abhängigkeiten zu bestimmten Betriebssystemen, durch die sich die Ausführbarkeit der Applikation einschränkt. Um die Zielgruppe möglicher Anwender zu vergrößern, ist die Plattformunabhängigkeit anzustreben. Zwar lassen sich mit *cross-platform* Frameworks wie z.B. Qt (siehe Abschnitt 3.1.1) Lösungen für mehrere Betriebssysteme und Plattformen zugleich erstellen, jedoch wird dabei nur das native *Look-And-Feel* imitiert. Um den Benutzern grafische Oberflächen zu bieten, deren Optik und Verhalten sich vollständig in die einzelnen Umgebungen integrieren, müssen die nativen Frameworks der Betriebssysteme verwendet werden. Damit Applikationen dennoch auf unterschiedlichen Plattformen ausgeführt werden können, ist ein flexibler Austausch des verwendeten GUI-Frameworks nötig.

In dieser Arbeit entsteht unter Anwendung der testgetriebenen Entwicklung nach Kapitel 2 ein Framework für den Entwurf von flexiblen und wandlungsfähigen grafischen Benutzerschnittstellen. Es unterstützt die leichte Austauschbarkeit der verwendeten GUI-Frameworks und ermöglicht dadurch plattformunabhängige Applikationen, die auf den plattformspezifischen Implementierungen basieren. Daraus leitet sich der Name CrossNative des Frameworks ab. Die Umsetzung erfolgt in der Sprache C++ unter MacOSX mit dem zugehörigen GUI-Framework Cocoa (siehe Abschnitt 3.1.2). Um die Austauschbarkeit der Frameworks zu überprüfen, wird Qt während der Entwicklung als zweites GUI-Framework herangezogen. Die Verwendung des *cross-platform* Frameworks Qt widerspricht zwar der Umsetzung nativer Oberflächen, erleichtert jedoch den Entwicklungsprozess, da die Ausführbarkeit der GUIs auch unter MacOSX möglich ist.

3.1 Allgemeiner Aufbau grafischer Oberflächen mit verschiedenen GUI-Frameworks

Das folgende Kapitel zeigt die Konzeption von CrossNative. Zunächst stellt es die verwendeten GUI-Frameworks Qt und Cocoa vor und erläutert allgemein wie sich die grafischen Oberflächen in dem jeweiligen System zusammensetzen. Anschließend wird erklärt, wie daraus Oberflächen erstellt werden können, die applikationsspezifisches Verhalten aufweisen. In einem nächsten Schritt werden die sich daraus ergebenden Schwierigkeiten für den Entwurf von CrossNative dargelegt und mögliche Strategien zu deren Bewältigung genannt. Im Zuge dessen wird ebenfalls die Idee sowie die gewünschte Funktionsweise des Frameworks erläutert. Der darauf folgende Abschnitt verdeutlicht die Rolle von Positionskriterien, durch die der Benutzer von CrossNative Einfluss auf die Position einzelner grafischer Bestandteile der Oberfläche erhält. Das Kapitel schließt mit der Diskussion über die Verwendung von `shared_ptr` ab, die für eine effiziente Speicherverwaltung des Frameworks sorgen.

Zu beachten gilt, dass es sich bei den im folgenden Kapitel geschilderten Ansätzen lediglich um eine Konzeption des Frameworks vor der Umsetzung handelt. Die tatsächliche Implementierung ergibt sich durch die Mechanismen der testgetriebenen Entwicklung und kann deshalb abweichen. Hier beschriebene Ansätze dienen nur zur Verdeutlichung möglicher Entwurfsschwierigkeiten und skizzieren die grobe Struktur der Software.

3.1 Allgemeiner Aufbau grafischer Oberflächen mit verschiedenen GUI-Frameworks

Grundsätzlich wird die GUI in der objektorientierten Programmierung auf Ebene des Quelltextes durch Klassen und Objekte repräsentiert. Für den Entwurf von grafischen Oberflächen kann auf die Funktionalität verschiedener Frameworks zurückgegriffen werden. Sie bieten bereits implementierte Klassen an, die für den Entwurf einer grafischen Oberfläche verwendet werden können. Benutzerschnittstellen besitzen in der Regel ein oder mehrere Hauptanwendungsfenster, die die optischen Grenzen der Applikation auf dem Bildschirm festlegen und den grafischen Inhalt der Applikation einrahmen. Die meisten Anwendungen verfügen zusätzlich über eine Menüleiste, die dem Benutzer die möglichen Menübefehle präsentiert. Dafür sind die einzelnen Befehle thematisch zusam-

mengehörend in Menüs eingeordnet. Die Steuerung der Applikation erfolgt entweder über Schaltflächen, die sich in den Hauptanwendungsfenstern befinden, oder über die Befehle in den Menüs der Applikation. Die folgenden Abschnitte zeigen, wie grafische Oberflächen mit den Frameworks Qt¹ und Cocoa² erstellt werden können, die über ein Hauptanwendungsfenster und eine Menüleiste verfügen.

3.1.1 Aufbau einer GUI unter Verwendung von Qt

Bei Qt handelt es sich um ein *cross-platform* Framework in der Sprache C++, das es erlaubt, grafische Oberflächen zu entwerfen, die ohne Veränderung auf unterschiedlichen Betriebssystemen ausführbar sind. Der Quelltext muss dafür lediglich durch entsprechende Compiler übersetzt werden. Die von diesem Framework zur Verfügung gestellten grafischen Elemente sind der nativer Frameworks detailgetreu nachempfunden, sodass auf den ersten Blick kein Unterschied zu den plattformspezifischen Frameworks auffällt. Die Basis der grafischen Darstellung von Qt liegt in der Klasse `QWidget`. Instanzen dieser Klasse können Events aus der Interaktion mit dem Benutzer erhalten und verarbeiten sowie ihre eigene Repräsentation auf dem Bildschirm zeichnen. Jedes grafische Element dieses Frameworks ist direkt oder indirekt von `QWidget` abgeleitet. Für häufig auftretende Darstellungsweisen wie z.B. Baumstrukturen und Tabellen bietet Qt bereits implementierte Klassen an (z.B. `QTreeView`, `QTableView`). Liegt keine vorgefertigte Klasse des Frameworks für die gewünschte Darstellung der Daten vor, können benutzerdefinierte `Widgets` abgeleitet werden. `Widgets` können sich auch aus anderen `Widgets` zusammensetzen und diese in einer bestimmten Anordnung anzeigen.

Der interne Aufbau von Qt-basierten Oberflächen gestaltet sich analog zu Abbildung 3.1. Hauptanwendungsfenster werden in Qt durch die Klasse `QMainWindow` repräsentiert. Jedes `QMainWindow` enthält ein zentrales `QWidget`, in dem sich der grafische Inhalt der Applikation befindet. Um die Anwendung mit einer Menüleiste auszustatten, enthält das `QMainWindow` die Methode `setMenuBar`, die eine Instanz der Klasse `QMenuBar` in das Hauptanwendungsfenster aufnimmt. Die Menüleisten in Qt basieren auf der Klasse

¹<https://doc.qt.io>

²<https://developer.apple.com/documentation>

3.1 Allgemeiner Aufbau grafischer Oberflächen mit verschiedenen GUI-Frameworks

QAction. Instanzen dieser Klasse spiegeln eine Handlung wider, die der Benutzer durch das Betätigen einer zugehörigen Schaltfläche auslösen kann. **QAction** leitet sich nicht von **QWidget** ab und besitzt daher auch keine Möglichkeit sich auf dem Bildschirm anzuzeigen. Das Zeichnen der Schaltflächen muss deswegen von der **QMenuBar** übernommen werden. Für jedes **QAction** fügt die **QMenuBar** eine Schaltfläche in die Menüleiste ein, die den Titel der **QAction** enthält. Die grafische Darstellung eines Menüs erfolgt durch die Klasse **QMenu**. Die **QActions** der Menüleiste werden über ihre **setMenu**-Methode mit einem solchen Menü versehen. Betätigt der Benutzer die Schaltflächen der Menüleiste, werden die **QMenus** angezeigt. Wie auch **QMenuBar** basiert **QMenu** auf der Klasse **QAction** und zeichnet entsprechende Schaltflächen. Für jede aufgenommene **QAction** entsteht deshalb ein neuer Eintrag in dem Menü. Werden die **QActions** in den **QMenus** wiederum mit Menüs ausgestattet, ergeben sich Untermenüs.

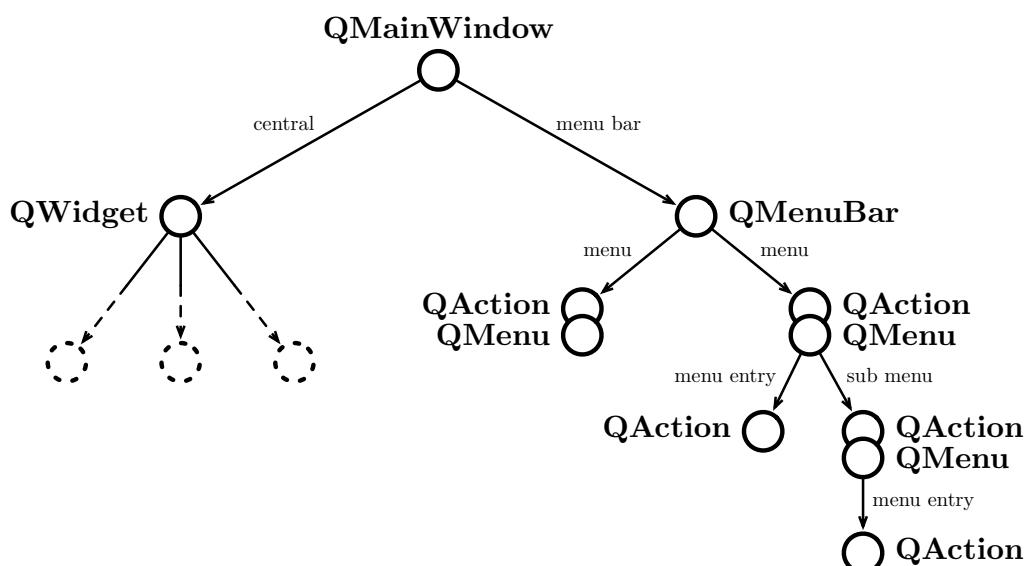


Abbildung 3.1: Objekthierarchie einer grafischen Oberfläche unter Verwendung des Frameworks Qt.

3.1.2 Aufbau der GUI unter Verwendung von Cocoa

Native Oberflächen für Apples Betriebssystem MacOSX werden mit dem Framework Cocoa implementiert, das eine API in der Programmiersprache *Objective-C* zur Verfügung

stellt. Für die Kompatibilität zu den in C++ umgesetzten Klassen des Frameworks CrossNative wird eine Zwischenschicht in *Objective-C++* benötigt.

Ähnlich wie **QWidget** in Qt existiert in Cocoa ebenfalls eine Basisklasse, die die Infrastruktur für das Zeichnen auf dem Bildschirm und für die Interaktion mit dem Benutzer liefert. Dabei handelt es sich um die Klasse **NSView**. Derivate dieser Schnittstelle stellen grafische Bestandteile der Oberfläche dar. Auch Cocoa besitzt vorgefertigte Implementierungen üblicher Darstellungsweisen von Daten. So repräsentieren zum Beispiel die Klasse **NSTableView** und **NSOutlineView** Tabellenansichten und Baumstrukturen auf Basis des Frameworks Cocoa. Analog zu Qt sind auch in Cocoa benutzerdefinierte Darstellungsweisen durch das Ableiten neuer Klassen von **NSView** möglich, die mehrere andere Subklassen von **NSView** enthalten können. Im Gegensatz zu Qt leiten sich in Cocoa nicht alle Klassen für grafische Bestandteile von **NSView** ab. Nur Klassen, die den Inhalt der Applikation in dem Hauptanwendungsfenster darstellen, entstammen dieser Vererbungshierarchie. Die Klassen **NSWindow** für Hauptanwendungsfenster sowie die Klassen **NSMenu** und **NSMenuItem** für das Zusammenstellen von Menüleisten und Menüs sind keine Derivate von **NSView**.

Die Verwendung von Cocoa lässt einen internen Aufbau der GUI nach Abbildung 3.2 entstehen. Die Klasse **NSWindow** stellt das Hauptanwendungsfenster der Applikation dar. Vergleichbar mit der Klasse **QMainWindow** enthalten Instanzen der Klasse **NSWindow** eine zentrale **NSView**, die den grafischen Inhalt der Applikation anzeigt. Die Menüs in der Benutzerschnittstelle werden von der Klasse **NSMenu** verwaltet. Cocoa unterscheidet dabei nicht zwischen Menüleiste und Menü. Bei MacOSX befindet sich die Menüleiste losgelöst von dem Hauptanwendungsfenster im oberen Bereich des Bildschirms. Anders als bei Qt wird sie deswegen nicht bei einer Instanz der Klasse **NSWindow** registriert, sondern einer **NSApplication** über die **setMainMenu**-Methode hinzugefügt. Die Klasse **NSApplication** verwaltet den *main event loop* und die von der Applikation verwendeten Ressourcen des Systems. Die Instanz der Klasse **NSMenu**, die in der **NSApplication** gesetzt ist, spiegelt die Menüleiste der GUI wieder. Der interne Aufbau der Menüleiste in Cocoa folgt dem gleichen Prinzip wie bei Qt. Dabei entspricht die Klasse **NSMenu** der Klasse **QMenu**, während **NSMenuItem** den Platz der Klasse **QAction** einnimmt.

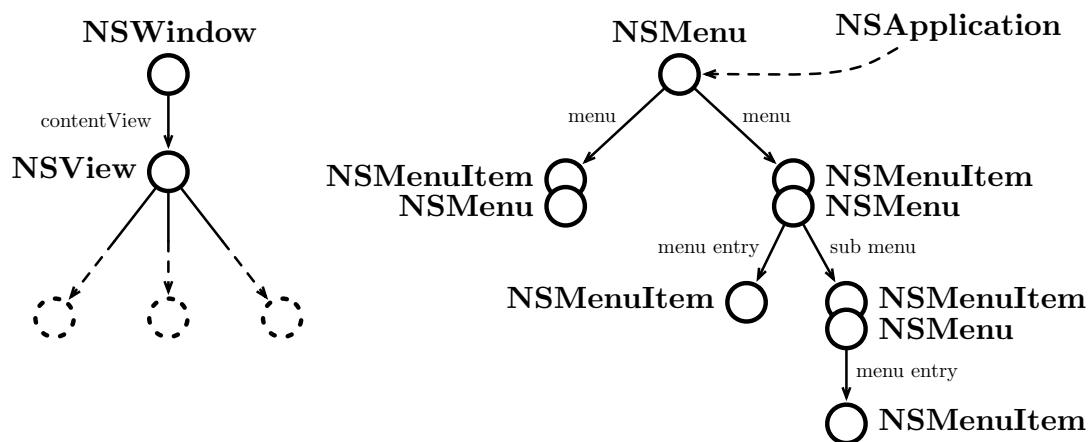


Abbildung 3.2: Objekthierarchie einer grafischen Oberfläche unter Verwendung des Frameworks Cocoa.

3.2 Aufbau applicationsspezifischer Oberflächen

Die vorhandenen Frameworks erlauben durch die Verwendung und Kombination ihrer bereitgestellten Klassen den einfachen Entwurf einer GUI. Da die einzelnen Implementierungen jedoch nur allgemeingültiges Verhalten besitzen, entsteht keine Oberfläche, die an die Daten und Ergebnisse der Applikation gekoppelt ist. Das allgemeingültige Verhalten der Implementierungen umfasst z.B. das Injizieren von Daten, die ein grafisches Element darstellen soll, das Verändern von Sichtbarkeiten, um einzelne Bereiche der GUI vorübergehend auszublenden bzw. anzuzeigen oder optische Einstellungen wie Hintergrundfarben oder Rahmen. Ein weiteres Beispiel ist das Blockieren von Schaltflächen, das den Benutzer in bestimmten Situationen vor fehlerhafter Bedienung schützen kann. Die Klassen der Frameworks haben keine Kenntnis über die Klassen der Problemdomäne und können somit auch nicht auf deren aktuellen Zustand reagieren. Damit die GUI dem Benutzer die Interaktion mit der Funktionalität der Problemdomäne ermöglicht und sich basierend auf dem aktuellen Zustand der Daten darstellt, muss die Oberfläche durch ein Klassensystem repräsentiert werden, das die Klassen der Frameworks um applicationspezifisches Verhalten erweitert. Dieses Problem wird durch das Erstellen von Klassen gelöst, die einerseits Schaltflächen mit dem Auslösen domänenspezifischer Funktionalität konfigurieren, andererseits die Darstellung der grafischen Elemente basierend auf den geänderten Daten aktualisieren.

Bei Applikationen mit geringem Funktionsumfang und Oberflächen mit moderater Komplexität reicht häufig eine einzige Klasse aus, die den Aufbau und die Funktionalität der gesamten GUI enthält. Je nach dem mit welchem Framework die GUI umgesetzt ist, könnte sie in diesem Fall die gesamte Objekthierarchie nach Abbildung 3.1 oder 3.2 umfassen und das Verarbeiten aller Handlungen des Benutzers sowie die Aktualisierung der grafischen Oberfläche übernehmen. Diese Klasse erhält üblicherweise den Namen `ApplicationShell` oder `Shell`. Mit wachsendem Funktionsumfang der Applikation steigt ebenfalls die Komplexität der GUI und damit auch die in der `Shell` enthaltene Logik. Um diese Komplexität zu organisieren, wird die Logik in mehrere Klassen aufgeteilt, die eine einzige oder mehrere Bestandteile der frameworkspezifischen Objekthierarchien enthalten. Dadurch werden mehrere grafische Bestandteile in Verhaltenseinheiten zusammengefasst. Jede dieser Klassen spiegelt eine grafische Ansicht bestimmter Daten oder Funktionalitäten der Problemdomäne wider und wird daher auch als View bezeichnet. Die GUI bzw. die `Shell` setzt sich dann aus Instanzen der einzelnen View-Klassen zusammen. Views sind für den Aufbau und das Verarbeiten von Benutzereingaben in ihren jeweiligen Ansichten zuständig. Bei zu hoher Komplexität in den einzelnen Bereichen der GUI können diese wiederum in mehrere View-Klassen zergliedert werden. Durch diese rekursive Dekomposition zur Beherrschung der Komplexität in der Benutzerschnittstelle entsteht eine applikationsspezifische Objekthierarchie, in der die frameworkspezifischen Strukturen nach Abbildung 3.1 bzw. 3.2 enthalten sind. Abbildung 3.3 zeigt beispielhaft, wie sich eine GUI aus der internen Sicht zusammensetzen könnte.

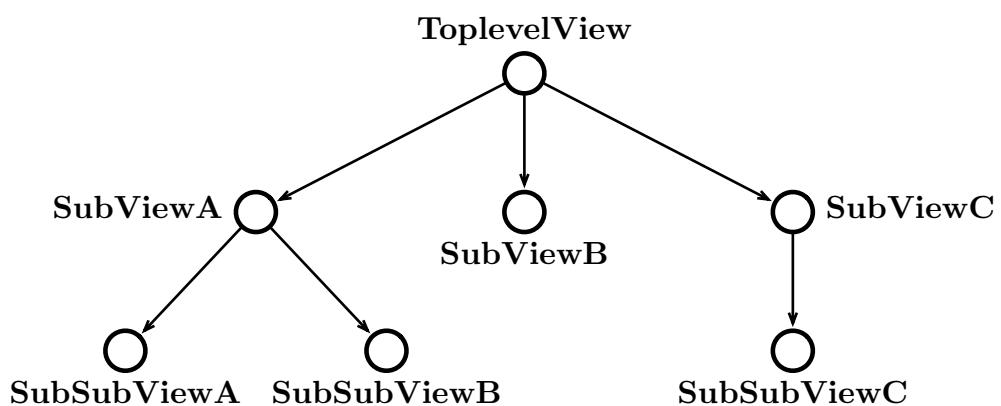


Abbildung 3.3: Die Objekthierarchie einer applikationsspezifischen GUI.

Abbildung 3.4 demonstriert eine mögliche Darstellung der View-Hierarchie aus Abbildung 3.3 auf dem Bildschirm. Jede View der Hierarchie spiegelt ein Rechteck auf dem Bildschirm wider. Views können sich aus einer beliebigen Anzahl an untergeordneten Views zusammensetzen und diese in ihrem eigenen Rechteck in einer bestimmten Anordnung anzeigen. Dadurch entsteht eine Verschachtelung einzelner Views. Der Darstellungsplatz und die Größe werden dabei von der übergeordneten View bestimmt. Bei Wurzelementen solcher Hierarchien (*ToplevelView*) handelt es sich meistens um eigenständige Fenster der Applikation. Ihre Größe und Position, und dadurch auch die Größe und Position der untergeordneten Views, wird durch die Interaktion mit dem Benutzer festgelegt. Bei MacOSX ist auch die Menüleiste eine *ToplevelView*, da sie sich nicht in den Hauptanwendungsfenstern der Applikationen befindet. Im Allgemeinen vergrößert sich das System der applikationsspezifischen Klassen der GUI sowie die Objekthierarchie der Views mit ansteigender Komplexität des Funktionsumfangs.

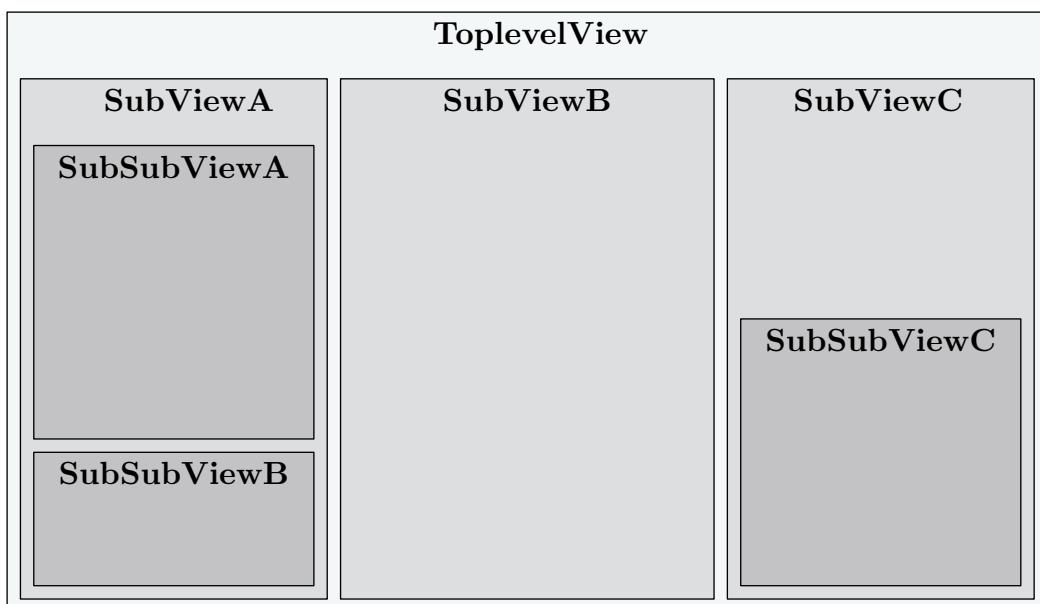


Abbildung 3.4: Mögliche grafische Darstellung der Objekthierarchie aus Abbildung 3.3.

3.3 Schwierigkeiten bei dem Entwurf von grafischen Oberflächen

Aus der zuvor beschriebenen Vorgehensweise beim Aufbau einer applikationsspezifischen GUI unter Verwendung der frameworkspezifischen Objekthierarchien ergeben sich einige Schwierigkeiten, die es in dem Entwurf von CrossNative zu berücksichtigen gilt. In dem nachfolgenden Abschnitt werden diese Problematiken näher erläutert und mögliche Strategien zu deren Bewältigung vorgestellt.

3.3.1 Komplexität der grafischen Oberfläche

Die Gestaltung der GUI nach Abschnitt 3.2 wird hauptsächlich durch zwei Faktoren beeinflusst: Die Art und Anzahl der in der GUI enthaltenen Views sowie deren Zusammensetzung in der applikationsspezifischen Objekthierarchie (siehe Abbildung 3.3). Der iterative Entwurfsprozess für eine GUI, die eine hohe Benutzerfreundlichkeit durch intuitive Bedienbarkeit und übersichtliche Darstellung der Daten aufweist, erfordert meistens ein flexibles Änderungspotential der Gestaltung, damit verschiedene Darstellungsarten und Kompositionen der Views ausprobiert werden können. Infolgedessen soll CrossNative den einfachen Austausch einzelner Views durch äquivalente Implementierungen ermöglichen und ohne großen Aufwand ein geändertes Arrangement in der Objekthierarchie der GUI zulassen.

Die Schwierigkeit darin besteht in der Komplexität der GUI, die stark von dem domänenspezifischen Funktionsumfang abhängt. Je mehr Funktionalität die Applikation zur Verfügung stellt, desto komplexer gestaltet sich auch das Verhalten der Oberfläche. Dadurch wächst einerseits das Klassensystem der GUI, andererseits die applikationspezifische Objekthierarchie der Views. Ein größer werdendes Klassensystem wiederum bringt eine stark ansteigende Anzahl an möglichen Abhängigkeiten mit sich, die die geforderte Flexibilität der Oberfläche verhindern können. Damit die flexible Gestaltungsmöglichkeit der GUI auch bei komplexen Applikationen gegeben ist, bestehen hohe Anforderungen an das objektorientierte Design in den Klassen der grafischen Oberfläche.

3.3.2 Unterschiede in den Funktionsweisen der Frameworks

Durch die einfache Austauschbarkeit des verwendeten Frameworks soll es CrossNative ermöglichen, Applikationen mit einer grafischen Oberfläche auf unterschiedlichen Plattformen auszuführen, ohne *cross-platform* Frameworks wie Qt zu verwenden. Dafür müssen die Views austauschbar gestaltet werden, sodass CrossNative eine GUI unabhängig von dem zugrundeliegenden Framework zusammensetzen kann. Aufgrund von verschiedenen Bedienkonzepten ergeben sich Differenzen in den Funktionsweisen der Frameworks, die sich in der internen Struktur der GUI bemerkbar machen (siehe Abbildung 3.1 und 3.2). Neben dem Austausch der View-Klassen muss daher auch die Komposition an die Funktionsweise des jeweiligen Frameworks angepasst werden. Damit der Entwickler das Arrangement entsprechend der Funktionsweise des verwendeten Frameworks gestalten kann, muss CrossNative ihm die Möglichkeit geben, Einfluss auf die Komposition der View-Objekte zu nehmen.

3.3.3 Die GUI – Der volatileste Bestandteil der Software

Üblicherweise folgt der Entwurf von Applikationen nach Buschmann et al. (1996) der Schichtenarchitektur. Dabei wird die Software in ihre Hauptbestandteile zergliedert und es entstehen auf oberster Abstraktionsebene mehrere Schichten (engl.: *Layer*), die jeweils eine Verantwortung des gesamten Systems übernehmen. Nach Brown et al. (2003) untergliedern sich Applikationen analog zu Abbildung 3.5 hauptsächlich in Datenschicht (*Data Source Layer*), Domänenschicht (*Domain Layer*) und Präsentationsschicht (*Presentation Layer*).

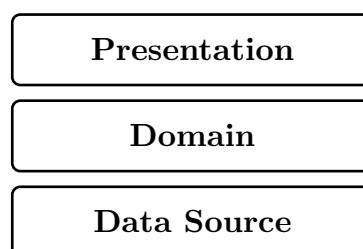


Abbildung 3.5: Schichtenarchitektur einer Applikation in Anlehnung an Brown et al. (2003).

3. Konzeption des Entwurfs von CrossNative

Die wichtigste der drei Schichten ist die Domänenschicht. Sie enthält die Business-Logik und charakterisiert somit das domänenspezifische Verhalten der Applikation. Die Objekte der Datenschicht stellen für die Domänenschicht einen allgemeingültigen Zugang zu spezifischen Datenquellen zur Verfügung. Dadurch wird ein Austausch der gewählten Datenbank möglich, ohne die Business-Logik zu gefährden. Bei der Präsentationsschicht handelt es sich um die Schnittstelle zwischen dem Anwender und der Applikation. Ihre Aufgabe umfasst das Bereitstellen der domänenspezifischen Funktionalität für den Anwender, das Verarbeiten von Benutzereingaben und die Darstellung des aktuellen Zustands sowie der Ergebnisse der Domänenschicht.

Grundsätzlich bilden niedrigere Schichten die Basis für höhere Ebenen der Architektur, indem sie ihre Funktionalität bereitstellen. Auf Klassenebene spiegelt sich das in Abhängigkeiten wieder. Findet eine Modifikation in Klassen einer niedrigeren Schicht statt, führt das zu einem Anpassungsbedarf in den abhängigen Klassen des nächst höheren Layers. Diese Anpassungen können wiederum Änderungen in noch höheren Ebenen erwzingen. In den Schichten der Applikation ergeben sich durch Modifikationen somit Änderungskaskaden in entgegengesetzter Richtung der Abhängigkeiten. Je höher daher eine Ebene in der Architektur des Systems angesiedelt ist, desto größer sollte auch ihr Änderungspotential sein. Orientiert sich der Entwurf der Software an der Schichtenarchitektur nach Brown et al. (2003), befinden sich die Klassen und Objekte der Präsentationsschicht in der höchsten Ebene. Da jede Änderung des Systems Anpassungen bis in die Präsentationsschicht auslösen kann, ist sie der volatilste Bestandteil von Software. Die Entwicklung von CrossNative befasst sich ausschließlich mit dem Entwurf eines Systems für die Gestaltung von grafischen Oberflächen und befindet sich somit vollständig in der Präsentationsschicht. Um jederzeit der Evolution tieferer Schichten gerecht zu werden, bedarf die mit CrossNative erstellte Oberfläche einer hohen Wartungsfreundlichkeit.

3.3.4 Adressieren der Entwurfsschwierigkeiten

Zusammengefasst soll CrossNative sowohl ein hohes Änderungspotential aufweisen, um Modifikationen der Problemdomäne kompensieren zu können, als auch die fle-

3.3 Schwierigkeiten bei dem Entwurf von grafischen Oberflächen

xible Gestaltung der Oberfläche durch die leichte Austauschbarkeit einzelner View-Implementierungen oder Variationen der Komposition zulassen. Ein üblicher Ansatz zur Umsetzung flexibler Software liegt in den Prinzipien und Metriken des objektorientierten Designs. Das Einhalten dieser Entwurfsregeln führt zu Systemen, die hinsichtlich ihrer Abhängigkeiten optimiert und somit leicht erweiterbar und wandelbar sind. Der Ansatz des objektorientierten Designs brachte in der Vergangenheit für ähnliche Entwurfsprobleme wiederkehrende Klassenstrukturen hervor, die sich für bestimmte Anwendungsfälle bewährten (Gamma et al., 2004). Auf diese Entwurfsmuster kann während der Implementierung der Software zurückgegriffen werden. Die Komposition von komplexen Objekthierarchien wird nach Gamma et al. (2004) durch das *Composite Pattern* analog zu Abbildung 3.6 adressiert. Dieses Entwurfsmuster sieht eine abstrakte Basisklasse vor (hier **View**), die die Infrastruktur für das Verwalten von Kindern bereitstellt. Aufgrund der rekursiven Komposition auf Basis der Schnittstelle **View** erfordert die Erweiterung der grafischen Oberfläche um neue **View**-Implementierungen nur die Erweiterung des

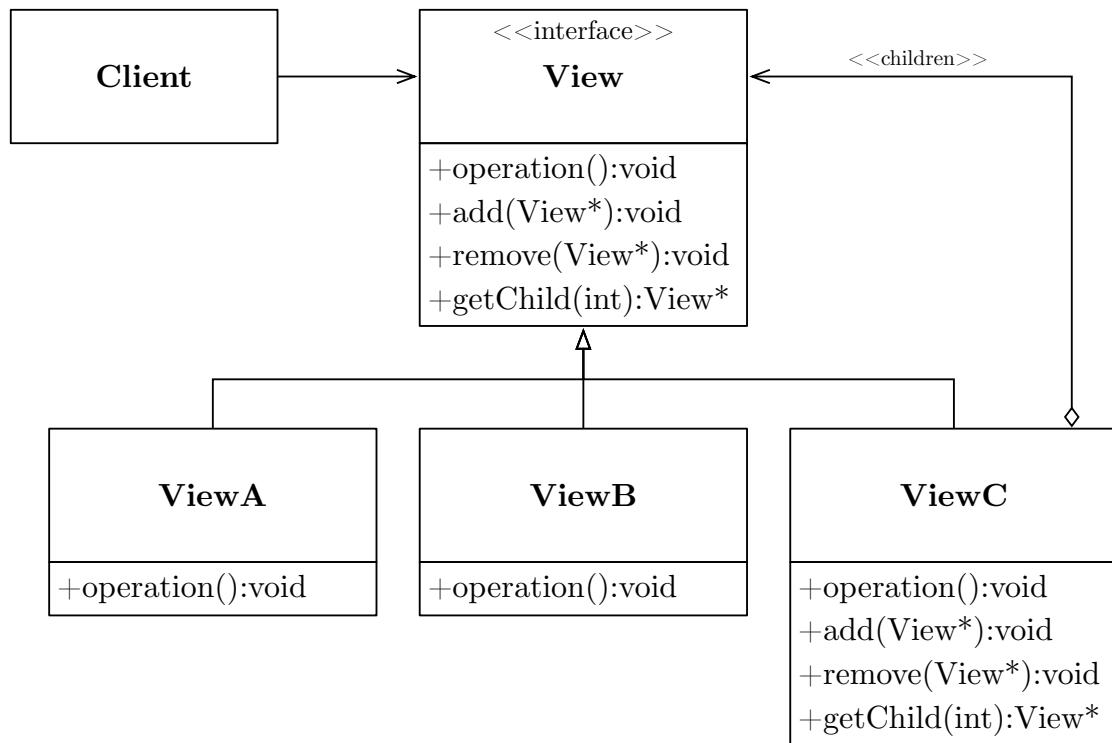


Abbildung 3.6: Composite Pattern in Anlehnung an Gamma et al. (2004).

3. Konzeption des Entwurfs von CrossNative

Klassensystems um neue Derivate der abstrakten Schnittstelle. Weder *Clients* von **View** noch andere Subklassen dieser Vererbungshierarchie müssen sich infolge neuer Derivate, ausgetauschter Implementierungen oder einer geänderten Zusammensetzung der Objekt-hierarchie anpassen.

Realisiert man die exemplarische Objekthierarchie aus Abbildung 3.3 mit Hilfe des *Composite Patterns*, erfolgt die Komposition der GUI analog zu Listing 3.1. Der Quelltext setzt voraus, dass die einzelnen Knoten der Hierarchie durch eigene Derivate der **View**-Schnittstelle repräsentiert werden. Die Bestandteile der GUI werden während der **main**-Methode der Applikation erzeugt und über die **add**-Methode der abstrakten Schnittstelle entsprechend verknüpft. Dadurch entstehen die Eltern-Kind-Beziehungen der Objekthierarchie. Die **add**-Methode wird dabei auf derjenigen **View** aufgerufen, die als Elternknoten in die Beziehung eingeht. Als Übergabeparameter erhält sie die **View**, die das Kind in der Beziehung darstellt.

```
int main(int argc, char** argv) {
    // ...
    View* subViewA = new SubViewA();
    subViewA->add(new SubSubViewA());
    subViewA->add(new SubSubViewB());

    View* subViewC = new SubViewC();
    subViewC->add(new SubSubViewC());

    View* TopLevelView = new TopLevelView();
    topLevelView->add(subViewA);
    topLevelView->add(new SubViewB());
    topLevelView->add(subViewC);
    // ...
}
```

Listing 3.1: Komposition der exemplarischen Objekthierarchie nach Abbildung 3.3 ohne Verwendung von CrossNative.

3.3 Schwierigkeiten bei dem Entwurf von grafischen Oberflächen

Um den internen Aufbau der GUI zu variieren, können die Adressaten der `add`-Methode und die übergebenen `Views` modifiziert werden. Besonders bei ansteigender Anzahl an beteiligten View-Objekten verliert der Quelltext bei einer Komposition auf diese Weise an Übersichtlichkeit. Aufgrund des sich ständig wiederholenden Musters mit ähnlichen Variablennamen und gleichen Methodenaufrufen sind fehlerhafte Eltern-Kind-Beziehungen leicht zu übersehen. Damit das Fehlerpotential minimiert wird, sollte das Zusammensetzen der Objekthierarchie von CrossNative übernommen werden. Entsprechend der unterschiedlichen Verantwortungen wird die Präsentationsschicht unter Anwendung von CrossNative, wie in Abbildung 3.7 gezeigt, in die zwei Module `Composition` und `Concrete Views` unterteilt. Dadurch wird die Logik der Komposition von den grafischen Repräsentationen der Oberfläche getrennt. Das Modul `Concrete Views` beinhaltet die in Abbildung 3.6 dargestellte Klassenhierarchie. Das Arrangement der `Views` in der Objekthierarchie hingegen wird von den Klassen des Moduls `Composition` bestimmt. So könnte der Benutzer des Frameworks die `View`-Objekte an das Kompositionsmodul übergeben, das eine entsprechende Objekthierarchie aufbaut. Die `View`-Implementierungen hängen nicht von den Klassen der Komposition ab. Daher können Änderungen des Kompositionsalgorithmus vorgenommen werden, die sich nicht auf das Modul der View-Klassen auswirken.

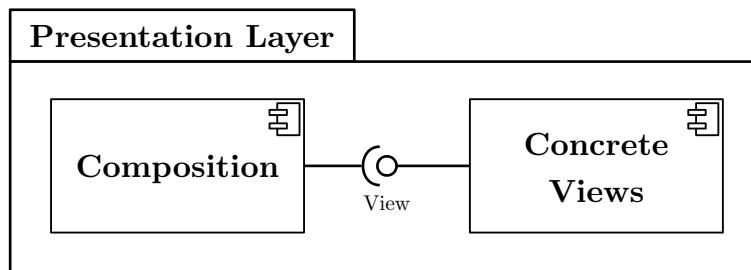


Abbildung 3.7: Entkopplung des Kompositionsalgorithmus von den View-Implementierungen.

Der geforderte Einfluss des Benutzers von CrossNative auf die Zusammensetzung der GUI könnte sich in der API des Kompositionsmoduls widerspiegeln. Wird ihm die Möglichkeit gegeben, zusätzlich zu einem `View`-Derivat Kriterien zu übergeben, die in den Kompositionsalgorithmus eingehen, kann er indirekt die Position des neuen Elements bestimmen. Durch die Variation der übergebenen Kriterien lässt sich

das Arrangement dann auf die Funktionsweisen der unterschiedlichen Frameworks anpassen. Listing 3.2 zeigt die mögliche Anwendung von CrossNative zur Komposition der exemplarischen Objekthierarchie nach Abbildung 3.3. Die Klasse `GuiHierarchy` ist Bestandteil des Kompositionsmoduls. Ihr werden `View`-Derivate übergeben, die sie entsprechend des Positions kriteriums in die Objekthierarchie einfügt.

```

1 int main( int argc , char** argv ) {
2     // ...
3     GuiHierarchy* hierarchy = new GuiHierarchy() ;
4
5     /*      View           , PositionCriterion      */
6     hierarchy->load( new TopLevelView() , topLevelCriterion ) ;
7     hierarchy->load( new SubViewA()       , childOfTopLevelCriterion ) ;
8     hierarchy->load( new SubViewB()       , childOfTopLevelCriterion ) ;
9     hierarchy->load( new SubViewC()       , childOfTopLevelCriterion ) ;
10    hierarchy->load( new SubSubViewA()    , childOfSubViewACriterion ) ;
11    hierarchy->load( new SubSubViewB()    , childOfSubViewACriterion ) ;
12    hierarchy->load( new SubSubViewC()    , childOfSubViewCCriterion ) ;
13    // ...
14 }
```

Listing 3.2: Komposition der exemplarischen Objekthierarchie nach Abbildung 3.3 mit CrossNative.

Des Weiteren unterstützt die Anwendung der testgetriebenen Entwicklung das Adressieren der Entwurfsschwierigkeiten. Durch die in Abschnitt 2.5 beschriebenen Effekte fördert sie die Umsetzung von Software mit hohem Änderungspotential, da ihre Wirkung unter anderem positiven Einfluss auf die Qualität des objektorientierten Designs nimmt. Die isolierte Betrachtung der einzelnen Testfälle führt zu hoher Kohäsion und loser Kopplung. Die resultierenden Systeme sind geprägt durch Abstraktionen, wodurch die Wartbarkeit und die Flexibilität erhöht wird.

3.4 Positionskriterien

Durch eine API des Kompositionsmoduls analog zu Listing 3.2 vereinfacht sich das Zusammensetzen der GUI, da das Aufnehmen von Bestandteilen in die Benutzerschnittstelle einheitlich und übersichtlich geschieht. Jede `View` wird bei dem gleichen Objekt der Klasse `GuiHierarchy` registriert, die es entsprechend des übergebenen Kriteriums in die Objekthierarchie aufnimmt. Die `GuiHierarchy` iteriert bei jedem Aufruf der `load`-Methode durch die zuvor entstandene Hierarchie und identifiziert anhand des Positionskriteriums den Elternknoten. Anschließend fügt sie dem ermittelten Elternknoten die neue `View` mit Hilfe der `add`-Methode hinzu.

Setzen sich Oberflächen, wie in Listing 3.2 gezeigt, nur aus einem Objekt zu jedem `View`-Derivat zusammen, können die Elternknoten anhand ihres konkreten Klassentyps identifiziert werden. Diese Art der Komposition liegt in der Hierarchie meistens in dem grafischen Inhalt der Applikation (siehe `contentView`, Abbildung 3.2) unterhalb des Hauptanwendungsfensters vor. Aufgrund der unterschiedlichen Funktionalitäten der einzelnen `Views` werden eigene applikationsspezifische Klassen erstellt, die üblicherweise nur ein Mal instanziert werden. Im Gegensatz dazu finden sich in der Hierarchie unterhalb der Menüleiste mehrere Instanzen gleicher Klassen wieder. Soll zum Beispiel in einer Cocoa-basierten GUI ein `NSMenuItem` in ein bestimmtes `NSMenu` aufgenommen werden, reicht es aufgrund der Vielzahl von verschiedenen Menüs nicht aus, den Elternknoten anhand seines Typs zu identifizieren. Damit die `GuiHierarchy` das Menü eindeutig erkennt, könnte sie stattdessen mit Hilfe des Kriteriums nach dem `NSMenu`-Objekt selbst oder nach einem eindeutigen Namen des Menüs suchen. Dafür muss die API des Kompositionsmoduls eine Struktur aufweisen, die den Austausch des Identifizierungsmechanismus durch unterschiedliche Typen von Kriterien erlaubt. Abbildung 3.8 zeigt eine mögliche Implementierung des Systems unter Verwendung von Polymorphie.

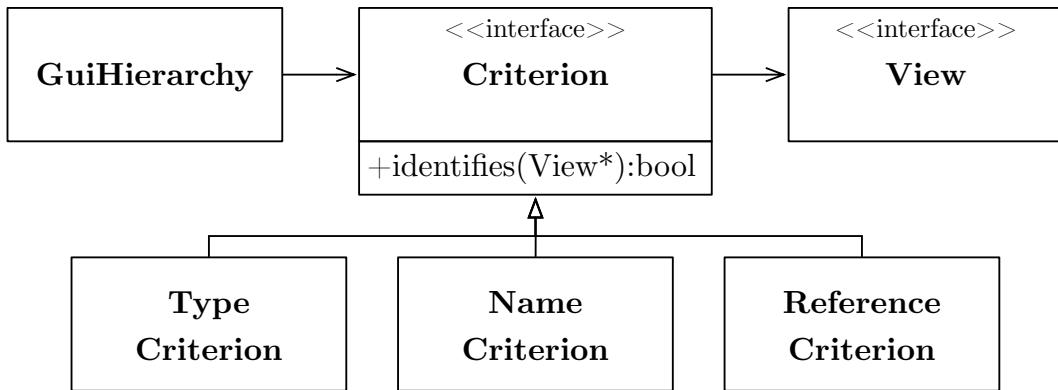


Abbildung 3.8: Austauschbarkeit des Identifizierungsmechanismus durch die Klassenhierarchie der Kriterien.

3.5 Speicherverwaltung durch intelligente Zeiger

Applikationen mit grafischen Oberflächen ermöglichen dem Benutzer die zugrundeliegenden Daten der Problemdomäne zur Laufzeit zu erzeugen. Die Anzahl der erstellten Objekte ergibt sich deswegen aus der Interaktion des Benutzers und ist zu Kompilierzeit unbekannt. Dafür benötigt man nach Kirch et al. (2015) Speicher, der dynamisch zur Laufzeit reserviert wird. In C++ wird das durch die Verwendung von Zeigern erreicht.

Verwendet man Zeiger in der rohen Form entsteht der Nachteil, dass der von ihnen referenzierte Speicher beim Verlassen ihres Gültigkeitsbereichs explizit freigegeben werden muss. Wird die Freigabe unterlassen, entsteht ein Speicherleck, da der referenzierte Speicher weiterhin belegt, jedoch nicht mehr erreichbar ist. Gleichzeitig führt auch eine zu frühe Freigabe des Speichers zu Problemen, da anschließende Aufrufe von Methoden den Absturz der Software hervorrufen. Die Verwendung von rohen Zeigern sollte daher nur in Systemen geschehen, in denen die Gültigkeitsbereiche eindeutig erkennbar sind.

Objektorientierte Systeme basieren darauf, dass sich unterschiedliche Objekte referenzieren und miteinander kollaborieren. Dabei ist es ebenfalls möglich, dass ein Objekt von mehreren anderen Objekten referenziert wird. Der Gültigkeitsbereich eines Objekts hängt von der Zeit ab, in der es von anderen Objekten verwendet wird. Erst wenn ein Objekt nicht mehr referenziert wird, endet sein Gültigkeitsbereich. Das letzte Objekt,

das die Beziehung zu dem referenzierten Objekt auflöst, müsste bei der Verwendung von Zeigern die Freigabe des Speichers vornehmen. Da kein Objekt evaluieren kann, ob es die letzte Referenz eines anderen Objekts hält, ist eine sichere Speicherfreigabe nicht möglich. Infolgedessen eignen sich rohe Zeiger nicht für die Umsetzung von objektorientierter Software.

Für die effiziente Verwaltung von dynamisch allokiertem Speicher bietet die C++-Standardbibliothek den intelligenten Zeiger `shared_ptr`³ an. Instanzen dieser Klasse verwalten den Speicher eines rohen Zeigers und geben diesen zum richtigen Zeitpunkt frei. Der Typ des verwalteten Zeigers wird dabei über ein *Template* bestimmt, das bei der Deklaration des `shared_ptr` festgelegt wird. Mehrere `shared_ptr` können denselben Zeiger verwalten. Erst wenn der letzte `shared_ptr` die Beziehung zu dem verwalteten Zeiger auflöst, wird der Speicher freigegeben. Im Rahmen dieser Arbeit erfolgt das Referenzieren von Objekten unter Verwendung des intelligenten Zeigers `shared_ptr`. Damit sichergestellt wird, dass jedes Objekt der Verwaltung eines intelligenten Zeigers unterliegt, ergibt sich ein Klassenschema nach Listing 3.3.

```
#include <memory>
2
3 class ExampleClass;
4 typedef std::shared_ptr<ExampleClass> ExampleClassPtr;
5
6 class ExampleClass {
7 public:
8     static ExampleClassPtr getInstance() {
9         return ExampleClassPtr(new ExampleClass());
10    }
11    virtual ~ExampleClass() {}
12 protected:
13     ExampleClass() {}
14};
```

Listing 3.3: Genereller Aufbau einer Klasse unter Verwendung von `shared_ptr`.

³http://de.cppreference.com/w/cpp/memory/shared_ptr

Ein privater oder geschützter Konstruktor unterbindet das Instanziieren eines rohen Zeigers. Die Erzeugung von Objekten geschieht über eine statische `getNewInstance`-Methode, die eine neue Instanz der Klasse erzeugt und sie in einen `shared_ptr` injiziert. Zur Verbesserung der Lesbarkeit werden *TypeDefs* verwendet, die die komplexen und *Template*-basierten Typen der intelligenten Zeiger hinter einfachen Ausdrücken verborgen. Diese Ausdrücke ergeben sich aus dem Namen der Klasse zusammen mit der Endung „`Ptr`“.

Die Umsetzung der Klassen entsprechend Listing 3.3 prägt ebenfalls die Signaturen aller Methoden. Da die Instanziierung der Klassen nur über die statische Methode `getNewInstance` möglich ist und dadurch nur Objekte vorliegen können, die von intelligenten Zeigern verwaltet werden, basieren auch die Übergabeparameter der Methoden auf `shared_ptr`. Aus Gründen der Übersichtlichkeit wird dieses Detail in den UML-Diagrammen (Fowler et al., 2000) der nachfolgenden Kapitel vernachlässigt. Zur Darstellung der Klassensysteme verwenden die UML-Diagramme lediglich die Typen Klassendefinitionen selbst.

3.6 Zusammenfassung

In diesem Kapitel erfolgt die Konzeption des Frameworks CrossNative. Es werden die funktionellen Anforderungen erarbeitet, die das zu implementierende Framework benötigt, um den flexiblen Entwurf von grafischen Oberflächen zu ermöglichen. Zunächst wird dafür der Aufbau von grafischen Oberflächen aus der internen Sicht im Hinblick auf verschiedene GUI-Frameworks analysiert. Im Fokus stehen hierbei die im Rahmen dieser Arbeit verwendeten Frameworks Qt und Cocoa. Die Analyse identifiziert grafische Oberflächen als Objekthierarchien aus den Implementierungsklassen der GUI-Frameworks, die aufgrund der unterschiedlichen Funktionsweisen der verschiedenen Frameworks in ihrem Arrangement variieren. Anschließend wird die Vorgehensweise beschrieben, wie aus diesen Objekthierarchien grafische Oberflächen mit applikationsspezifischem Verhalten erstellt werden können. Um die allgemeingültigen Implementierungen der GUI-Frameworks um applikationsspezifisches Verhalten zu erweitern, sind weitere Klassen

umzusetzen, die zusätzlich Kenntnis über die Problemdomäne besitzen. Dabei handelt es sich um die View-Klassen, die grafische Verhaltenseinheiten der Oberfläche widerspiegeln. Analog zu den verwendeten GUI-Frameworks ergeben sich dabei ebenfalls hierarchische Strukturen, in denen die frameworkspezifischen Objekthierarchien enthalten sind. Für die flexible Gestaltung komplexer GUIs ergibt sich deshalb als Kernaufgabe für CrossNative die Komposition von Objekthierarchien. Des Weiteren muss CrossNative den iterativen Entwurfsprozess grafischer Oberflächen unterstützen, indem es dem Benutzer erlaubt, das Arrangement der Hierarchie flexibel zu verändern, um neue View-Klassen zu erweitern oder Views gegen andere Views auszutauschen.

Im Anschluss werden die Entwurfsprobleme hervorgehoben, die es bei dem Entwurf von CrossNative zu berücksichtigen gilt. Es wird beschrieben, wie die Komplexität von grafischen Oberflächen mit hohem Funktionsumfang große und komplexe Klassensysteme innerhalb der Präsentationsschicht erzeugen, die aufgrund von Abhängigkeiten die geforderte Flexibilität des Arrangements gefährden. Um die Abhängigkeiten innerhalb dieser Systeme zu minimieren und somit die Flexibilität des Arrangements zu erreichen, bestehen hohe Anforderungen hinsichtlich des objektorientierten Designs in den Klassensystemen der Views. Auch wird dargelegt, dass der Austausch des verwendeten GUI-Frameworks Schwierigkeiten mit sich bringt. Aufgrund der verschiedenen Funktionsweisen der einzelnen Frameworks besitzen äquivalente grafische Oberflächen einen anderen internen Aufbau. Um diese Differenzen zu überwinden, muss CrossNative dem Benutzer die Möglichkeit bieten, während der Komposition Einfluss auf das Arrangement zu nehmen. Darüber hinaus wird beschrieben, dass die Präsentationsschicht aufgrund ihrer Abhängigkeiten zu tieferen Schichten der Applikation eine hohe Wartbarkeit erfordert.

Nachfolgend werden Strategien aufgezeigt, mit denen diese Entwurfsschwierigkeiten adressiert werden können. Es wird veranschaulicht, wie flexible und leicht erweiterbare Objekthierarchien mit Hilfe des *Composite Patterns* nach Gamma et al. (2004) umgesetzt werden können. Damit sich das Arrangement der Komposition nicht auf die einzelnen View-Klassen auswirkt, sollte des Weiteren auf eine strikte Trennung der Logik für das Zusammensetzen der Hierarchie von den Implementierungen der Views geachtet werden.

3. Konzeption des Entwurfs von CrossNative

Um dem Benutzer das Zusammensetzen der Hierarchie zu erleichtern, sollte CrossNative auch über einen Kompositionsalgorithmus verfügen, der das komplexe Zusammenfügen der grafischen Oberfläche hinter einer einfachen API verbirgt. Für den Einfluss des Benutzers auf das Arrangement der grafischen Bestandteile könnten dem Kompositionsalgorithmus zusätzlich zu einer einzufügenden View ein Positions kriterium übergeben werden, anhand dessen die Einfügeposition in Form des Elternknotens bestimmt wird.

KAPITEL 4

Testgetriebene Entwicklung des Kompositionsalgorithmus

Nach der Erarbeitung der groben Funktionsweise sowie des strukturellen Umrisses der Software in Kapitel 3 wird der Ansatz der testgetriebenen Entwicklung nach Kapitel 2 angewendet, um die vagen Spezifikationen des Frameworks CrossNative durch das Formulieren von Tests zu konkretisieren und dadurch die Entwicklung voranzutreiben. Die in Kapitel 3 beschriebene Struktur der Software dient dabei nur als Orientierung für den Entwurf des Frameworks. Die tatsächlichen Designstrukturen ergeben sich durch die Mechanismen der testgetriebenen Entwicklung.

Als Kernaufgabe befasst sich CrossNative mit der dynamischen Komposition von Objekthierarchien für den Entwurf von grafischen Benutzerschnittstellen. Im Mittelpunkt steht daher ein Kompositionsalgorithmus, der den dynamischen Aufbau der applikations-spezifischen View-Hierarchie ermöglicht. Das folgende Kapitel zeigt die testgetriebene Entwicklung dieses Algorithmus und verdeutlicht exemplarisch die Auswirkungen des Vorgehens auf die Umsetzung einer einzelnen *Unit of Work*. Dabei wird auch der Einfluss von TDD auf den Entwurfsprozess und das entstehende System diskutiert. Für die testgetriebene Entwicklung wird die Strategie der Triangulation nach Beck (2003)

gewählt, bei der sich der Algorithmus durch das sukzessive Einhalten mehrerer konkreter Tests der generischen Endlösung annähert. Mit jedem Durchlauf des testgetriebenen Entwicklungszyklus erfährt das System die Erweiterung um einen neuen Test, der die Funktionalität des Algorithmus in Richtung der generischen Lösung lenkt. Die Beschreibung der einzelnen Testzyklen erfolgt analog zu Abschnitt 2.2 in den Phasen Rot, Grün und *Refactoring*. Sind keine Modifikationen des Quelltextes zu Gunsten seiner Qualität möglich bzw. nötig, kann die Phase *Refactoring* entfallen.

4.1 Zyklus 1 – Erstmaliges Verknüpfen von Views

Der erste Test dieser *Unit of Work* befasst sich mit dem einfachsten Fall, der bei dem Erstellen einer Objekthierarchie auftreten kann. Besteht die applikationsspezifische View-Hierarchie aus nur zwei Objekten, muss das SUT lediglich den Kindknoten bei dem Elternknoten hinzufügen. Wird der Elternknoten zuerst an das SUT übergeben, kann das erreicht werden, indem die als zweites übergebene View der ersten View hinzugefügt wird. Abbildung 4.1 enthält eine grafische Darstellung des Tests, unterteilt in die Bestandteile *Arrange*, *Act* und *Assert* nach Abschnitt 2.3.2. FirstView spiegelt in dem Test den Elternknoten wider, SecondView den Kindknoten. Im *Arrange* des Tests wird zunächst die FirstView an das SUT übergeben. Anschließend wird während des *Acts* der Testmethode die SecondView geladen. Das erwartete Endergebnis im *Assert* beinhaltet, dass die SecondView der FirstView hinzufügt wird.

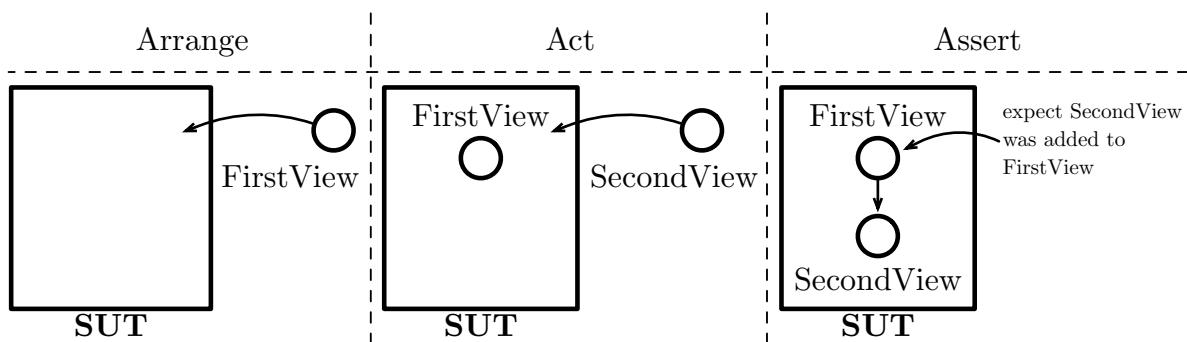


Abbildung 4.1: Grafische Darstellung des ersten Testfalls.

Rot

Wie in Abschnitt 2.2 erläutert, beginnt jeder Zyklus der testgetriebenen Entwicklung mit der Phase Rot, in der ein Test formuliert wird, der die zu erweiternde Funktionalität beschreibt. Da es sich um den ersten Testzyklus des Projekts handelt, liegt noch kein Produktionscode vor. Während des Entwurfs des Tests kann daher frei über Klassen- und Methodennamen, sowie die Funktionsweise des SUT bestimmt werden. Die Aufgabe des SUT beinhaltet es, View-Objekte dynamisch zu einer hierarchischen Struktur zusammenzusetzen. Aus diesem Grund wird das SUT durch die Klasse `CNDynamicViewHierarchy` repräsentiert. Der Präfix `CN` drückt die Zugehörigkeit zu dem Framework CrossNative aus. Dementsprechend ergibt sich für die View-Objekte der Typ `CNView`. Wird der `CNDynamicViewHierarchy` eine View übergeben, ermittelt diese zunächst die Einfügeposition in Form des Elternknotens. Anschließend fügt sie die View dem Elternknoten als Kindknoten hinzu. Dieser Prozess wird im Folgenden als das Laden einer View bezeichnet. Infolgedessen wird die öffentliche Methode der `CNDynamicHierarchy`, die die *Unit of Work* auslöst, mit dem Namen `load` versehen. Listing 4.1 zeigt die implementierte Testmethode zu Abbildung 4.1. Listing 4.2 zeigt das zugehörige *Test Fixture* `CNDynamicViewHierarchyTest`.

```

1 TEST_F(CNDynamicViewHierarchyTest ,
2     LoadedFirstView__LoadSecondView__SecondViewShouldBeAddedToFirstView) {
3     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
4     CNViewSpyPtr firstView = CNViewSpy::getInstance();
5     sut->load(firstView);
6
6     CNViewPtr secondView = CNViewDummy::getInstance();
7     sut->load(secondView);
8
8     CNViewPtr actual = firstView->getAddedView();
9     CNViewPtr expected = secondView;
10    EXPECT_EQ(actual, testing::Eq(expected)) << "SecondView should be
11        added to FirstView, but it was not!";
12 }

```

Listing 4.1: Zyklus 1 – Rot: Implementierung des ersten Testmethode.

```

1 #include <gmock/gmock.h>
2 #include <string>
3 #include "CNDynamicViewHierarchy.h"
4 #include "CNViewTestDoubles.h"
5
6 class CNDynamicViewHierarchyTest : public testing::Test {
7 protected:
8     virtual CNDynamicViewHierarchyPtr makeCNDynamicViewHierarchy() {
9         return CNDynamicViewHierarchy::getNewInstance();
10    }
11};

```

Listing 4.2: Zyklus 1 – Rot: Implementierung des *Test Fixtures* `CNDynamicViewHierarchyTest`.

Zur Konstruktion des SUT enthält das *Text Fixture* eine Fabrik-Methode¹, die die Instanziierung der Klasse `CNDynamicViewHierarchy` kapselt. Jeder Test des Kompositionsalgorithmus benötigt als SUT eine Instanz dieser Klasse. Die Erzeugung des SUT mit Hilfe der Fabrik-Methode vermeidet somit Redundanzen, wodurch sich die Wartbarkeit der Tests erhöht.

Zeile 2 - 4 des Listings 4.1 enthalten das *Arrange* des Tests. Mit Hilfe der Fabrik-Methode wird zunächst das SUT erstellt. Anschließend wird die View, die als Elternknoten in die Hierarchie eingeht, erzeugt und geladen. Das Endergebnis dieses Testfalls basiert auf der Interaktion mit dem Elternknoten und erzeugt durch das Hinzufügen der SecondView indirekten Output. Zur Verifizierung der korrekten Interaktion wird die FirstView als *Test Spy* ausgeführt. Im anschließenden *Act* erzeugt der Test die SecondView und übergibt sie der `load`-Methode der `CNDynamicViewHierarchy`. Da die *Unit of Work* keine Methoden der SecondView aufruft, wird hier ein *Dummy Object* verwendet. Das *Assert* des Tests in den Zeilen 9 - 12 des Listings 4.1 vergleicht das `CNView`-Objekt, das der FirstView tatsächlich hinzugefügt wurde (siehe `actual`), mit der erwarteten SecondView (siehe `expected`). Der nachfolgende `string` beinhaltet eine aussagekräftige Fehlernachricht, die ausgegeben wird, sobald der Test fehlschlägt.

¹Gamma et al., 2004

Um die Lesbarkeit des Tests zu erhöhen, wird das *Test Fixture* mit der in Listing 4.3 gezeigten Helper-Methode erweitert, die die Logik des *Asserts* kapselt. Infolgedessen vereinfacht sich das *Assert* in der Testmethode auf den in Listing 4.4 gezeigten Aufruf dieser Methode.

```

1 virtual void expectReceiverAddedView(CNViewSpyPtr receiver,
2     CNViewPtr view, std::string errorMessage) {
3     CNViewPtr expected = view;
4     CNViewPtr actual = receiver->getAddedView();
5     EXPECT_THAT(actual, testing::Eq(expected)) << errorMessage;
6 }
```

Listing 4.3: Zyklus 1 – Rot: Erweiterung des *Test Fixtures* um die Helper-Methode `expectReceiverAddedView`.

```

TEST_F(CNDynamicViewHierarchyTest,
1     LoadedFirstView__LoadSecondView__SecondViewShouldBeAddedToFirstView) {
2     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
3     CNViewSpyPtr firstView = CNViewSpy::getInstance();
4     sut->load(firstView);
5
6     CNViewPtr secondView = CNViewDummy::getInstance();
7     sut->load(secondView);
8
9     std::string errorMessage = "SecondView should be added to FirstView,
10    but it was not!";
11     expectReceiverAddedView(firstView, secondView, errorMessage);
12 }
```

Listing 4.4: Zyklus 1 – Rot: Verwendung der Helper-Methode aus Listing 4.3 in dem ersten Testfall.

Im letzten Schritt der Phase Rot erfolgt ein *Failure Run* (siehe Abschnitt 2.2) zur Überprüfung der Fehlerfreiheit des Tests. Da noch kein Produktionscode existiert, der die geforderte Funktionalität beinhaltet, sollte der Test bei korrekter Umsetzung auf die erwartete Weise fehlschlagen. Aufgrund der fehlenden Klassen und Metho-

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

den kann der Test jedoch nicht kompiliert werden. Die Umsetzung der in Listing 4.5 - 4.9 gezeigten Infrastruktur ermöglicht das erfolgreiche Bauen und Ausführen des Tests.

```
1 #include <memory>

3 class CNView;

5 class CNDynamicViewHierarchy;
typedef std::shared_ptr<CNDynamicViewHierarchy> CNDynamicViewHierarchyPtr;
7
9 class CNDynamicViewHierarchy {
public:
    static CNDynamicViewHierarchyPtr getInstance();
11    virtual ~CNDynamicViewHierarchy() {};
protected:
    CNDynamicViewHierarchy();
public:
    virtual void load(std::shared_ptr<CNView> view);
15};
}
```

Listing 4.5: Zyklus 1 – Rot: Implementierung der Klasse `CNDynamicHierarchy` (`CNDynamicViewHierarchy.h`).

```
#include "CNDynamicViewHierarchy.h"
2 #include "CNView.h"

4 CNDynamicViewHierarchyPtr CNDynamicViewHierarchy::getInstance() {
    return CNDynamicViewHierarchyPtr(new CNDynamicViewHierarchy());
6 }
CNDynamicViewHierarchy::~CNDynamicViewHierarchy() {}
8 CNDynamicViewHierarchy::CNDynamicViewHierarchy() {}

10 void CNDynamicViewHierarchy::load(CNViewPtr view) {}
```

Listing 4.6: Zyklus 1 – Rot: Implementierung der Klasse `CNDynamicHierarchy` (`CNDynamicViewHierarchy.cpp`).

```

1 #include <memory>
2
3 class CNView;
4 typedef std::shared_ptr<CNView> CNViewPtr;
5
6 class CNView {
7     public:
8         virtual ~CNView() {}
9     protected:
10        CNView() {}
11    };

```

Listing 4.7: Zyklus 1 – Rot: Implementierung des *Interfaces* CNView.

```

1 #include "CNView.h"
2
3 class CNViewSpy;
4 typedef std::shared_ptr<CNViewSpy> CNViewSpyPtr;
5
6 class CNViewSpy : public CNView {
7     public:
8         static CNViewSpyPtr newInstance() {
9             return CNViewSpyPtr(new CNViewSpy());
10        }
11        virtual ~CNViewSpy() {}
12    protected:
13        CNViewSpy() {}
14    public:
15        Virtual CNViewPtr getAddedView() {
16            return added;
17        }
18    private:
19        CNViewPtr added = nullptr;
20    };

```

Listing 4.8: Zyklus 1 – Rot: Implementierung des *Test Doubles* CNViewSpy.

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

```

1 #include "CNView.h"
2
3 class CNViewDummy;
4
5 typedef std::shared_ptr<CNViewDummy> CNViewDummyPtr;
6
7 class CNViewDummy : public CNView {
8     public:
9         static CNViewDummyPtr getNewInstance() {
10             return CNViewDummyPtr(new CNViewDummy());
11         }
12     protected:
13         CNViewDummy() {}
14     };

```

Listing 4.9: Zyklus 1 – Rot: Implementierung des *Test Doubles* `CNViewDummy`.

Betrachtet man das entstandene Klassensystem in Abbildung 4.2, verdeutlicht sich der Einfluss von TDD auf das objektorientierte Design. Für die Testbarkeit der `CNDynamicViewHierarchy` und die Isolation des Testfalls muss dem Test die Möglichkeit gegeben werden, *Test Doubles* anstatt der echten Abhängigkeit zu injizieren. Infolgedessen wird `CNView` als *Interface* ausgebildet. Dadurch werden das *Open/Closed Principle* und das *Dependency Inversion Principle* umgesetzt.

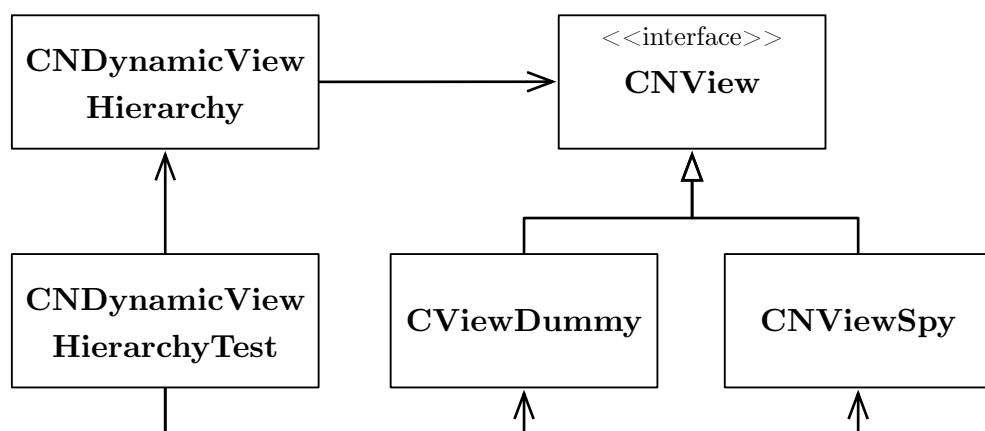


Abbildung 4.2: Das Klassensystem nach der Implementierung der ersten Testmethode.

Nicht nur das Klassensystem sondern auch die API des SUT wird von dem Test beeinflusst. Die Formulierung des Tests impliziert die Signaturen und die Rückgabetypen der öffentlichen Methoden des SUT. Nach Abschnitt 3.3.4 benötigt das Laden einer View ein zusätzliches Positionskriterium um die Einfügeposition zu ermitteln. Da in diesem einfachen Testfall die Position des Kindknotens bekannt ist, erübrigt sich die Verwendung von Kriterien. Um die Verständlichkeit des Tests zu wahren, werden sie hier vernachlässigt. Erst wenn ein Testfall tatsächlich auf die Verwendung von Positionskriterien angewiesen ist, wird die API angepasst. Dadurch ergibt sich zu jedem Zeitpunkt eine API des SUT, die auf die momentane Funktionalität zugeschnitten ist.

Grün

Sobald der Test im *Failure Run* auf die erwartete Weise fehlschlägt, erfolgt die Erweiterung des Produktionscodes, um die Anforderungen des Tests zu erfüllen. Nach Abschnitt 2.2 darf in dieser Phase nur so wenig Quelltext wie möglich umgesetzt werden, um den Test zum Bestehen zu bringen. Der einfachste Weg den in Phase Rot definierten Test bestehen zu lassen, liegt in der Erweiterung des SUT analog zu Listing 4.10 und 4.11.

```
1 private:
2     std::shared_ptr<CNView> firstView;
```

Listing 4.10: Zyklus 1 – Grün: Erweiterung der Klasse `CNDynamicViewHierarchy` um das Attribut `firstView`.

```
1 void CNDynamicViewHierarchy :: load (CNViewPtr view) {
2     if (!firstView) {
3         firstView = view;
4     }
5     firstView ->add (view);
6 }
```

Listing 4.11: Zyklus 1 – Grün: Implementierung der `load`-Methode entsprechend des ersten Tests.

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

Der Klasse `CNDynamicViewHierarchy` wird das private Attribut `firstView` vom Typ `CNView` hinzugefügt. Mit diesem Attribut kann das SUT die zuerst geladene View speichern. In der `load`-Methode wird zunächst überprüft, ob das Attribut `firstView` bereits einen Wert enthält. Ist das nicht der Fall, handelt es sich um den ersten Aufruf der Methode und die übergebene `CNView` wird in dem Attribut gespeichert. Bei einem erneuten Aufruf ist `firstView` bereits belegt. Das SUT fügt dann die übergebene View der zuerst geladenen View hinzu. Dafür erfolgt die Erweiterung des *Interfaces* `CNView` um die in Listing 4.12 dargestellte, rein virtuelle `add`-Methode.

```
#include <memory>

2   class CNView;
4   typedef std :: shared_ptr<CNView> CNViewPtr;

6   class CNView {
public:
8     virtual ~CNView() {}
protected:
10    CNView() {}
public:
12    virtual void add(CNViewPtr view) = 0;
};
```

Listing 4.12: Zyklus 1 – Grün: Erweiterung des *Interfaces* `CNView` um die `add`-Methode.

Aufgrund der Vererbung sind auch die *Test Doubles* `CNViewDummy` und `CNViewSpy` gezwungen sich anzupassen. Während die Methode in dem *Dummy Object* leer implementiert wird, merkt sich der *Test Spy* die übergebene View entsprechend Listing 4.13 in ihrem Attribut `added`, das dem Test zur Verifizierung des indirekten Outputs dient.

```

1 class CNViewSpy;
2     typedef std::shared_ptr<CNViewSpy> CNViewSpyPtr;
3
4 class CNViewSpy : public CNView {
5     public:
6         static CNViewSpyPtr newInstance() {
7             return CNViewSpyPtr(new CNViewSpy());
8         }
9         virtual ~CNViewSpy() {}
10    protected:
11        CNViewSpy() {}
12    public:
13        virtual void add(CNViewPtr view) override {
14            added = view;
15        }
16        Virtual CNViewPtr getAddedView() {
17            return added;
18        }
19    private:
20        CNViewPtr added = nullptr;
21 };

```

Listing 4.13: Zyklus 1 – Grün: Erweiterung des *CNViewSpys* um die Implementierung der `add`-Methode zur Verifizierung des indirekten Outputs.

4.2 Zyklus 2 – Unterlassen des Verknüpfens

Als Ergebnis des ersten Testzyklus ergibt sich ein Algorithmus, der beim mehrmali-
gem Aufruf die übergebenen `CNView`-Objekte der zuerst geladenen View als Kindknoten
hinzufügt. Dabei wird nicht überprüft, ob die erste View als Elternknoten für folgende
Views zulässig ist. Da es die `CNDynamicViewHierarchy` dem Benutzer des Frameworks
auch ermöglichen soll, mehrere Views zu laden, die nicht miteinander verknüpft werden
können, ergibt sich der zweite Testfall analog zu Abbildung 4.3.

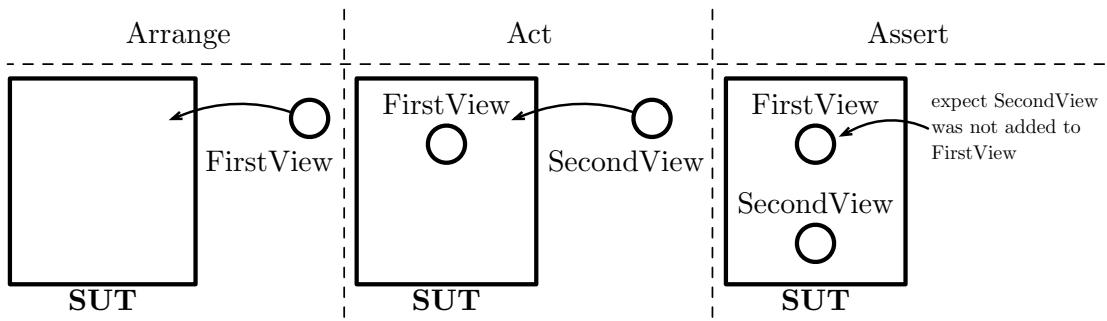


Abbildung 4.3: Grafische Darstellung des zweiten Testfalls.

Wie auch im ersten Test werden im *Arrange* und im *Act* der Testmethode jeweils eine View geladen. Zu berücksichtigen gilt dabei, dass die zuerst geladene *FirstView* hier nicht als Elternknoten der *SecondView* dient. Der Kompositionsalgorithmus in der `load`-Methode muss die unzulässige Verknüpfung erkennen und den Aufruf der `add`-Methode unterlassen. In der gegenwärtigen Form des Systems liegen dem Algorithmus zu wenig Informationen vor, um eine mögliche Eltern-Kind-Beziehung zu identifizieren. Durch Veränderungen in den APIs der beteiligten Klassen und Schnittstellen des Systems können diese Informationen in den Algorithmus eingetragen werden. Da die `load`-Methode bisher nur eine einzige Einfügeposition in die View-Hierarchie ermöglicht, reicht ein boolscher Ausdruck aus, um über die Verknüpfung der Views zu entscheiden.

Rot

Um dem Algorithmus den benötigten boolschen Ausdruck zur Verfügung zu stellen, sind mehrere Modifikationen des bisherigen Systems denkbar. Jede dieser Varianten entspricht einer bestimmten Designentscheidung, die durch den Entwickler getroffen werden muss. Die unterschiedlichen Designentscheidungen spiegeln sich in der Teststruktur wieder. Eine iterative Implementierung der Testmethode unterstützt den Entwickler dabei, sich aktiv für ein Design des Systems zu entscheiden, da ihn die Formulierung der Tests auf mögliche Vor- und Nachteile der einzelnen Entscheidungen hinweist. In diesem Abschnitt werden unterschiedliche Varianten diskutiert, um den Einfluss des Feedbacks zu verdeutlichen, das während der Implementierung einer Testmethode entsteht. Aus Gründen der Übersichtlichkeit werden hier nur zwei Varianten betrachtet.

In der ersten Variante erfolgt keine Modifikation in der API der Klasse `CNDynamicViewHierarchy`. Daher muss dem SUT der geforderte boolsche Ausdruck durch das *Interface CNView* geliefert werden. Es entsteht indirekter Input, der in den Kompositionsalgorithmus eingeht. Zu diesem Zweck könnte `CNView` beispielsweise um die Methode `matches` erweitert werden, die Aufschluss darüber gibt, ob sie der zuerst geladenen View hinzugefügt werden kann. Um den ausgeführten Pfad der *Unit of Work* zu kontrollieren, wird die `SecondView` dann als *Test Stub* umgesetzt, den der Test über eine setter-Methode mit dem boolschen Ausdruck konfigurieren kann. Daraus ergibt sich eine Implementierung der Testmethode nach Listing 4.14.

```

1 TEST_F(CNDynamicViewHierarchyTest, LoadedFirstView
2     _LoadSecondViewNotMatching__SecondViewShouldNotBeAddedToFirstView) {
3     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
4     CNViewSpyPtr firstView = CNViewSpy::getNewInstance();
5     sut->load(firstView);
6
7     CNViewStubPtr secondView = CNViewStub::getNewInstance();
8     secondView->setMatches(false);
9     sut->load(secondView);
10
11    std::string errorMessage = "Subview should not be added to View, but
12        it was!";
13    expectReceiverDidNotAddView(view, subView, errorMessage);
14 }
```

Listing 4.14: Zyklus 2 – Rot: Test 2, Variante 1 – Indirekter Input durch das *Interface CNView*.

Wie in dem ersten Test wird im *Arrange* ein `CNViewSpy` instanziert und geladen. An der Stelle des `CNViewDummy`-Objekts für die zweite View erzeugt der zweite Test jedoch ein `CNViewStub`. Über die `setMatches`-Methode injiziert er dem *Test Double* den gewünschten indirekten Input. Da die `SecondView` der `FirstView` nicht als Kindknoten hinzugefügt werden soll, übergibt der Test hier den Wert `false`. Als Endergebnis wird erwartet, dass die `add`-Methode der `FirstView` nicht mit der `SecondView` als Parameter

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

aufgerufen wird. Zugunsten der Lesbarkeit findet auch hier eine Erweiterung des *Test Fixtures* um eine Helper-Methode statt, die die Logik des *Asserts* kapselt. Listing 4.15 zeigt die in der Testmethode verwendete `expectReceiverDidNotAddView`-Methode.

```
1 virtual void expectReceiverDidNotAddView(CNViewSpyPtr receiver, CNViewPtr
2   view, std::string errorMessage) {
3   CNViewPtr unexpected = view;
4   CNViewPtr actual = receiver->getAddedView();
5   EXPECT_THAT(actual, testing::NotEqual(unexpected)) << errorMessage;
}
```

Listing 4.15: Zyklus 2 – Rot: Erweiterung des *Test Fixtures* um die Helper-Methode
`expectReceiverDidNotAddView`

Die zweite Variante beinhaltet eine veränderte Signatur der `load`-Methode. Neben dem `CNView`-Objekt wird ihr zugleich die boolsche Variable übergeben, die über das Verknüpfen der Views entscheidet. Der indirekte Input, den die *Test Doubles* in der ersten Variante des Tests leisten, wird somit zu direktem Input der Testmethode. Die Implementierung erfolgt dann analog zu Listing 4.16.

```
1 TEST_F(CNDynamicViewHierarchyTest, LoadedFirstView
2   __LoadSecondViewNotMatching__SecondViewShouldNotBeAddedToFirstView) {
3   CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
4   CNViewSpyPtr firstView = CNViewSpy::getInstance();
5   sut->load(firstView, false);
6
7   CNViewPtr secondView = CNViewDummy::getInstance();
8   sut->load(secondView, false);
9
10  std::string errorMessage = "SecondView should not be added to
11    FirstView, but it was!";
12  expectReceiverDidNotAddView(firstView, secondView, errorMessage);
13 }
```

Listing 4.16: Zyklus 2 – Rot: Test 2, Variante 2 – `load`-Methode mit direktem Input durch die Testmethode.

Betrachtet man beide Varianten im Vergleich, ist ersichtlich, dass die unterschiedlichen Implementierungen der Testmethode die Verantwortung über den boolschen Ausdruck an verschiedene Komponenten des Systems verteilen. Um sich auf eine der beiden Implementierungen festzulegen, muss die Bedeutung des boolschen Ausdrucks für den Kompositionsalgorithmus verdeutlicht werden. Anhand des Werts des Parameters entscheidet der Algorithmus über die Verknüpfung der FirstView und der SecondView. Nur wenn der Parameter den Wert `true` enthält, ist die FirstView als Elternknoten für die SecondView zulässig. Er dient somit zur Identifizierung möglicher Eltern-Kind-Beziehungen. Da die Einfügeposition in Form der FirstView als Elternknoten von dem bisherigen Algorithmus vorgegeben ist, reicht ein boolscher Ausdruck aus, um über die Verknüpfung zu entscheiden. Aus der Konzeption des Frameworks in Kapitel 3 geht hervor, dass die Einfügeposition einer View aufgrund der geforderten Flexibilität der grafischen Oberfläche und den unterschiedlichen Funktionsweisen verschiedener GUI-Frameworks variierbar sein muss (siehe Abschnitt 3.3.1 und 3.3.2). Damit die `load`-Methode den Elternknoten einer View dynamisch ermitteln kann, benötigt sie im weiteren Verlauf der Entwicklung nicht nur einen boolschen Parameter, der über das Einfügen an einer festgelegten Position entscheidet, sondern einen komplexen Mechanismus, der diesen boolschen Wert entsprechend eines betrachteten Knotens liefert. Da der Wert des Parameters später als Ergebnis dieses Mechanismus entsteht, ist die Entscheidung für eine der beiden Implementierungen der Testmethode wegweisend dafür, welche Komponente des Systems zukünftig über den verwendeten Identifizierungsmechanismus entscheidet.

Die erste Variante verwendet die Schnittstelle `CNView`, um den boolschen Parameter als indirekten Input in den Kompositionsalgorithmus einzutragen. Demnach obliegt die Verantwortung über diesen Parameter den `CNView`-Derivaten. Für den Benutzer von Cross-Native bedeutet diese Variante, dass er nur indirekten Einfluss auf das Arrangement der Komposition nehmen kann, indem er den verwendeten Identifizierungsmechanismus in den konkreten Views variiert. Da jede View während dieses Mechanismus ihren Elternknoten erkennen und dem Kompositionsalgorithmus entsprechend einen boolschen Wert übermitteln muss, entsteht darüber hinaus eine Kopplung verschiedener Views. Die Modifikation bzw. der Austausch einer View-Implementierung in der Komposition erfordert daher gegebenenfalls Anpassungen in allen vorhandenen Kindknoten der

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

View-Hierarchie. Des Weiteren verstößt diese Variante gegen das *Single Responsibility Principle*, da die Views neben der Darstellung viewspezifischer Inhalte auch für ihre Position in der grafischen Oberfläche verantwortlich sind.

Die Übergabe des Parameters als direkter Input in der zweiten Implementierung der Testmethode hingegen delegiert die Verantwortung über die Position einer View an den Benutzer des SUT bzw. der Klasse `CNDynamicViewHierarchy`. Dadurch wird dem Benutzer erlaubt, direkten Einfluss auf das Arrangement der Hierarchie zu nehmen, ohne die Views zu modifizieren. Daraus ergibt sich die Entkopplung der View-Derivate untereinander, die Einhaltung des *Single Responsibility Principles* sowie die einfache Austauschbarkeit einzelner Views in der Komposition der grafischen Oberfläche. Zugleich ruft die Implementierung der Testmethode entsprechend der zweiten Variante die in Abbildung 3.7 (siehe Seite 45) gezeigte Trennung der View-Implementierungen von der Kompositionslogik hervor und trifft damit die Intention des Frameworks CrossNative. Aus den zuvor genannten Gründen wird für die weitere Entwicklung die zweite Variante der Testmethode gewählt. An dieser Stelle verdeutlicht sich die Wirkung der iterativen Vorgehensweise während der Formulierung von *Unit Tests*. Anstatt das Design des Systems vor der Implementierung der Testmethode festzulegen und den Test im Anschluss entsprechend zu implementieren, wird das Formulieren der Testmethode verwendet, um das beste Design herauszufinden. Da der Test das SUT, wie in Abschnitt 2.5.2 beschrieben, aus Sicht des Benutzers betrachtet, steht dabei die API der getesteten Klasse im Mittelpunkt. Die unterschiedlichen Varianten der Testmethode spiegeln demnach verschiedene Anwendungsweisen der getesteten Klasse wider. Nach jeder Implementierung der Testmethode wird die durch den Test implizierte Funktionsweise mit der Funktionsweise verglichen, die den Vorstellungen des Entwicklers entspricht. Erst wenn die implizierte API den Entwickler zufriedenstellt, erfolgt die nächste Phase des Entwicklungszyklus. Da sich das Design des Systems aus der Absicht der Anwendung ergibt, handelt es sich hierbei um das in Abschnitt 2.5.4 erwähnte *Intentional Programming*.

Bevor nach den Regeln der testgetriebenen Entwicklung zur nächsten Phase übergegangen werden darf, ist erneut ein *Failure Run* notwendig. Aufgrund des zusätzlichen boolschen Übergabeparameters während der Aufrufe der `load`-Methode in

dem zweiten Test scheitert das Kompilieren des Quelltextes. Die Ausführbarkeit ergibt sich durch eine modifizierte API der Klasse `CNDynamicViewHierarchy` nach Listing 4.17 und 4.18. Die geänderte API des SUT erfordert zugleich die in Listing 4.19 dargestellte Anpassung der ersten Testmethode.

```
1 public:
2     virtual void load( std::shared_ptr<CNView> view , bool matches );
3 }
```

Listing 4.17: Zyklus 2 – Rot: Angepasste API der Klasse `CNDynamicViewHierarchy` zur Ausführung des Tests (`CNDynamicViewHierarchy.h`).

```
1 void CNDynamicViewHierarchy :: load( CNViewPtr view , bool matches ) {
2     if ( ! firstView ) {
3         firstView = view;
4     }
5     firstView ->add( view );
6 }
```

Listing 4.18: Zyklus 2 – Rot: Angepasste API der Klasse `CNDynamicViewHierarchy` zur Ausführung des Tests (`CNDynamicViewHierarchy.cpp`).

```
TEST_F( CNDynamicViewHierarchyTest , LoadedFirstView __LoadSecondView
      __SecondViewShouldBeAddedToFirstView ) {
    CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
    CNViewSpyPtr firstView = CNViewSpy :: getInstance();
    sut ->load( firstView , false );

    CNViewPtr secondView = CNViewDummy :: getInstance();
    sut ->load( secondView , true );

    std :: string errorMessage = "SecondView should be added to FirstView ,
      but it was not!";
    expectReceiverAddedView( firstView , secondView , errorMessage );
}
```

Listing 4.19: Zyklus 2 – Rot: Anpassen des ersten Tests an die veränderte API der Klasse `CNDynamicViewHierarchy`

Grün

Die Erweiterung des Kompositionsalgorithmus in der `load`-Methode zur Umsetzung der von dem zweiten Test definierten Funktionalität beschränkt sich auf die Abfrage der boolschen Variable `matches` in Zeile 4 des Listings 4.20. Wird der `load`-Methode bei dem zweiten Aufruf der Wert `true` übergeben, ruft das SUT die `add`-Methode der `firstView` auf und fügt somit die als zweites geladene View hinzu. Enthält `matches` den Wert `false`, unterlässt die `CNDynamicViewHierarchy` das Verknüpfen der Views.

```

1 #include "CNDynamicViewHierarchy.h"
2 #include "CNView.h"

4 CNDynamicViewHierarchyPtr CNDynamicViewHierarchy::getNewInstance() {
    return CNDynamicViewHierarchyPtr(new CNDynamicViewHierarchy());
6 }
CNDynamicViewHierarchy::~CNDynamicViewHierarchy() {}
8 CNDynamicViewHierarchy::CNDynamicViewHierarchy() {}

10 void CNDynamicViewHierarchy::load(CNViewPtr view, bool matches) {
    if (!firstView) {
12     firstView = view;
    }
14     if (matches) {
        firstView->add(view);
16     }
}

```

Listing 4.20: Zyklus 2 – Grün: Erweiterung der `load`-Methode um den Einfluss der boolschen Variablen `matches` (`CNDynamicViewHierarchy.cpp`).

Refactoring

Betrachtet man beide Testfälle, fallen Redundanzen in der Instanziierung der *Test Doubles* auf, die aus Gründen der Wartbarkeit zu minimieren sind. Dafür erfolgt

eine Erweiterung des *Test Fixtures* um die Fabrik-Methoden nach Listing 4.21, die anschließend von beiden Tests zur Erzeugung der *Test Doubles* verwendet wird.

```

1 virtual CNViewSpyPtr makeCNViewSpy() {
2     return CNViewSpy::getInstance();
3 }
4 virtual CNViewPtr makeCNViewDummy() {
5     return CNViewDummy::getInstance();
6 }
```

Listing 4.21: Zyklus 2 – Refactoring: Redundanzvermeidung durch Fabrik-Methoden für die Instanziierung der *Test Doubles* `CNViewDummy` und `CNViewSpy`

4.3 Zyklus 3 – Die dritte Ebene der Objekthierarchie

In den ersten beiden Entwicklungszyklen entsteht ein Algorithmus, der eine Objekthierarchie mit zwei Ebenen zusammensetzt. Dieser Zyklus erweitert die `load`-Methode, sodass die Objekthierarchie auch Views auf der dritten Ebene zulässt. Es werden nacheinander drei Views geladen, die in der Hierarchie eine Reihe bilden. Dabei spiegelt `FirstView` in dem Test den Knoten auf oberster Ebene wider. Bei `SecondView` und `ThirdView` handelt es sich um die Knoten auf zweiter und dritter Ebene der Hierarchie.

Rot

Listing 4.22 zeigt die umgesetzte Testmethode. Im *Arrange* erfolgt zunächst das Laden der `FirstView`. Anschließend lädt der Test die `SecondView` in die `CNDynamicViewHierarchy`, sodass sie der `FirstView` als Kindknoten hinzugefügt wird. Während des *Acts* lädt der Test die `ThirdView` zusammen mit dem boolschen Wert `true` in die Hierarchie. Als Endergebnis wird erwartet, dass die *Unit of Work* die `ThirdView` bei der `SecondView` als Kindknoten hinzufügt. Aufgrund der unveränderten APIs der beteiligten Klassen kann direkt der *Failure Run* erfolgen.

```

TEST_F(CNDynamicViewHierarchyTest ,
1   LoadedFirstView_LoadedSecondViewMatching_LoadThirdViewMatching
2     __ThirdViewShouldBeAddedTosecondView) {
3     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
4     CNViewPtr firstView = makeCNViewDummy();
5     sut->load(firstView, false);
6     CNViewSpyPtr secondView = makeCNViewSpy();
7     sut->load(secondView, true);
8
9     CNViewPtr thirdView = makeCNViewDummy();
10    sut->load(thirdView, true);
11
12    std::string errorMessage = "ThirdView should be added to SecondView,
13      but it was not!";
14    expectReceiverAddedView(secondView, thirdView, errorMessage);
15 }
```

Listing 4.22: Zyklus 3 – Rot: Implementierung des dritten Testfalls.

Grün

Ähnlich wie im ersten Zyklus wird zur Erfüllung der Anforderungen des Tests die Klasse `CNDynamicViewHierarchy` nach Listing 4.23 und 4.24 um ein Attribut erweitert in dem es die als zweites geladene View abspeichert. Bei dem dritten Aufruf der `load`-Methode zeigen sowohl `firstView`, als auch `secondView` auf ein Objekt. Daher fügt es die zuletzt geladene View der `secondView` hinzu.

```

private:
1   std::shared_ptr<CNView> firstView;
2   std::shared_ptr<CNView> secondView;
```

Listing 4.23: Zyklus 3 – Grün: Erweiterung der Klasse `CNDynamicViewHierarchy` um das Attribut `secondView`.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , bool matches) {
2     if (!firstView) {
3         firstView = view;
4     } else if (!secondView) {
5         secondView = view;
6         if (matches) {
7             firstView->add (view);
8         }
9     } else {
10        secondView->add (view);
11    }
12 }
```

Listing 4.24: Zyklus 3 – Grün: Erweiterung der `load`-Methode entsprechend des dritten Testfalls.

Refactoring

Nach der erfolgreichen Erweiterung des SUT sorgt ein weiterer Schritt des *Refactorings* für die Verbesserung der Wartbarkeit der Tests. Sowohl in dem ersten als auch in dem dritten Test werden Fehlernachrichten erzeugt, die sich bis auf die Namen der Views gleichen. Das Extrahieren der Helfer-Methode nach Listing 4.25 in das *Test Fixture* vermeidet die redundante Formulierung der Fehlernachricht.

```

virtual std :: string getReceiverAddedViewErrorMessage (std :: string
2 receiverName , std :: string viewName) {
    return viewName + " should be added to " + receiverName + ", but it
    was not!";
}
```

Listing 4.25: Zyklus 3 – Refactoring: Erweiterung des *Test Fixtures* um die Helfer-Methode `getReceiverAddedViewErrorMessage`.

4.4 Zyklus 4 – Unterlassen des Verknüpfens auf dritter Ebene

Auch auf der dritten Ebene der Hierarchie soll der Einfluss der boolschen Variablen berücksichtigt werden. Erfolgt der dritte Aufruf der `load-Methode` mit dem boolschen Wert `false`, darf die `CNDynamicViewHierarchy` keine Verknüpfung zwischen der SecondView und der ThirdView erstellen.

Rot

Wie in dem vorherigen Zyklus werden in diesem Test nach Listing 4.26 zunächst FirstView und SecondView geladen. Anschließend übergibt der Test die ThirdView an das SUT. Im Gegensatz zum dritten Test enthält hier jedoch der boolsche Ausdruck den Wert `false`. Für die Fehlernachricht im *Assert* des Tests wird das *Test Fixture* analog zu Listing 4.25 um eine weitere Helfer-Methode erweitert. Auch dieser Test kann sofort kompiliert und ausgeführt werden.

```

1 TEST_F(CNDynamicViewHierarchyTest ,
2     LoadedFirstView_LoadedSecondViewMatching__LoadThirdViewNotMatching
3     __ThirdViewShouldNotBeAddedToSecondView) {
4     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
5     CNViewPtr firstView = makeCNViewDummy();
6     sut->load(firstView, false);
7     CNViewSpyPtr secondView = makeCNViewSpy();
8     sut->load(secondView, true);
9
10    CNViewPtr thirdView = makeCNViewDummy();
11    sut->load(thirdView, false);
12
13    std::string errorMessage = getReceiverDidNotAddViewErrorMessage("SecondView", "ThirdView");
14    expectReceiverDidNotAddView(secondView, thirdView, errorMessage);
15 }
```

Listing 4.26: Zyklus 4 – Rot: Implementierung des dritten Testfalls.

Grün

Die Modifikation der `load`-Methode in Zeile 10 nach Listing 4.27 führt zu dem Bestehen des Tests. Wie auch im zweiten Testfall wird hier nur eine Abfrage der Variablen `matches` hinzugefügt, die über den Aufruf der `add`-Methode entscheidet.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , bool matches) {
2     if (! firstView) {
3         firstView = view;
4     } else if (! secondView) {
5         secondView = view;
6         if (matches) {
7             firstView->add (view);
8         }
9     } else {
10        if (matches) {
11            secondView->add (view);
12        }
13    }
14 }
```

Listing 4.27: Zyklus 4 – Grün: Erweiterung der `load`-Methode um den Einfluss der boolschen Variable `matches` auf zweiter Ebene der Hierarchie.

4.5 Zyklus 5 – FirstView als Elternknoten von ThirdView

Der bisherige Algorithmus erlaubt nur die Verknüpfung von unmittelbar nacheinander geladenen Views. Dadurch schränkt CrossNative den Benutzer in der Komposition der View-Hierarchie ein. Da auch das Verknüpfen von Views, die nicht direkt nacheinander an die `CNDynamicViewHierarchy` übergeben werden, einen realistischen Anwendungsfall darstellt, behandelt der fünfte Test die Eltern-Kind-Beziehung zwischen FirstView und ThirdView. Dafür werden FirstView, SecondView und ThirdView nacheinander geladen.

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

Als Ergebnis wird der Aufruf der `add`-Methode der `FirstView` mit der `ThirdView` als Parameter erwartet.

Rot

Listing 4.28 zeigt die Implementierung der fünften Testmethode. Wie im dritten und vierten Test werden zunächst `FirstView` und `SecondView` instanziert und geladen. Die Übergabe an die `load`-Methode erfolgt zusammen mit boolschen Werten, die eine Eltern-Kind-Beziehung zwischen `FirstView` und `SecondView` entstehen lassen. Das *Act* des Tests beinhaltet die Erzeugung und das Laden der `ThirdView`. Da diese View in die Hierarchie aufgenommen werden soll, enthält der übergebene boolsche Ausdruck den Wert `true`. Als Endergebnis überprüft der Test die Interaktion des SUT mit der zuerst geladenen View. Daher wird die `FirstView` hier als *Test Spy* ausgebildet.

```
TEST_F(CNDynamicViewHierarchyTest ,  
    LoadedFirstView_LoadedSecondViewMatching_LoadThirdViewMatching  
2     __ThirdViewShouldAddedToFirstView) {  
    CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();  
4     CNViewSpyPtr firstView = makeCNViewSpy();  
    sut->load(firstView, false);  
6     CNViewPtr secondView = makeCNViewDummy();  
    sut->load(secondView, true);  
8  
10    CNViewPtr thirdView = makeCNViewDummy();  
     sut->load(thirdView, true);  
  
12    std::string errorMessage = getReceiverAddedViewErrorMessage("FirstView"  
     " , "ThirdView");  
    expectReceiverAddedView(firstView, thirdView, errorMessage);  
14}
```

Listing 4.28: Zyklus 5 – Rot: Implementierung des fünften Testfalls.

Grün

Listings 4.29 zeigt die geringste Änderung des Kompositionsalgorithmus zur Umsetzung der von dem Test geforderten Funktionalität. Das Bestehen des Test wird durch die Erweiterung des `else`-Blocks um Zeile 13 erreicht.

```

void CNDynamicViewHierarchy :: load (CNViewPtr view , bool matches) {
    2   if (! firstView) {
        firstView = view;
    } else if (! secondView) {
        secondView = view;
    } 6   if (matches) {
        firstView->add (view);
    } 8
    } 10  else {
        if (matches) {
            secondView->add (view);
        } 12
        firstView->add (view);
    } 14
}

```

Listing 4.29: Zyklus 5 – Grün: Erweiterung der `load`-Methode um die Verknüpfung zwischen der ersten und der dritten View.

4.6 Zyklus 6 – Unterlassen des Verknüpfens von *FirstView* und *ThirdView*

Die Modifikation der `load`-Methode im vorherigen Testzyklus vernachlässigt den Einfluss der Variablen `matches`. Sie führt daher zu einem Algorithmus, der unabhängig von dem übergebenen boolschen Wert eine Eltern-Kind-Beziehung zwischen der *FirstView* und der *ThirdView* erstellt. Übergibt der Benutzer dem Aufruf der `load`-Methode jedoch den Wert `false`, soll die *ThirdView* nicht als Kindknoten bei der *FirstView* hinzugefügt werden.

Rot

Nach dem identischen *Arrange* des sechsten Tests mit der zuvor implementierten Testmethode erfolgt das Laden der ThirdView im *Act* entsprechend Listing 4.30 mit dem boolschen Wert `false`. Das *Assert* des Tests erwartet, dass bei diesem Aufruf die Verknüpfung von FirstView und ThirdView ausbleibt.

```
1 TEST_F(CNDynamicViewHierarchyTest ,  
2     LoadedFirstView_LoadedSecondViewMatching  
3     __LoadThirdViewNotMatching__ThirdViewShouldNotBeAddedToFirstView) {  
4         CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();  
5         CNViewSpyPtr firstView = makeCNViewSpy();  
6         sut->load(firstView, false);  
7         CNViewPtr secondView = makeCNViewDummy();  
8         sut->load(secondView, true);  
9  
10        CNViewPtr thirdView = makeCNViewDummy();  
11        sut->load(thirdView, false);  
12  
13        std::string errorMessage = getReceiverDidNotAddViewErrorMessage("FirstView", "ThirdView");  
14        expectReceiverDidNotAddView(firstView, thirdView, errorMessage);  
15    }
```

Listing 4.30: Zyklus 6 – Rot: Implementierung des sechsten Testfalls.

Grün

Für den Einfluss des boolschen Übergabeparameters auf den Aufruf der `load`-Methode der FirstView ergibt sich die Änderung des Algorithmus analog zu Listing 4.31. Die Zeilen 12 und 13 der `load`-Methode werden vertauscht, sodass sich der Aufruf der `add`-Methode nun innerhalb des letzten `if`-Blocks befindet.

```

void CNDynamicViewHierarchy :: load (CNViewPtr view , bool matches) {
    2   if (!firstView) {
        firstView = view;
    4   } else if (!secondView) {
        secondView = view;
    6     if (matches) {
            firstView->add (view);
        }
    8   } else {
        10    if (matches) {
            secondView->add (view);
            firstView->add (view);
        }
    12  }
    14}
}

```

Listing 4.31: Zyklus 6 – Grün: Modifizierung der `load`-Methode, um die Verknüpfung zwischen der ersten und der dritten View zu verhindern.

4.7 Zyklus 7 – Komplexes Verknüpfen

Betrachtet man den Algorithmus aus Listing 4.31, wird ersichtlich, dass eine View bei dem dritten Aufruf der `load`-Methode entweder beiden zuvor geladenen Views als Kindknoten hinzugefügt wird oder keiner. `CNDynamicViewHierarchy` lässt es nicht zu, den Elternknoten der neuen View präzise zu bestimmen. Durch den boolschen Übergabeparameter liegt dem Kompositionsalgorithmus lediglich vor, ob die View mit einer anderen View in der Hierarchie verknüpft werden soll. Es stehen jedoch keine Informationen über den Elternknoten zur Verfügung. Die `load`-Methode benötigt deshalb einen weiteren Übergabeparameter, der die fehlenden Informationen liefert. An dieser Stelle muss das System um den im zweiten Zyklus erwähnten Identifizierungsmechanismus erweitert werden. Dafür wird die Klasse `CNMatcher` eingeführt, die Verhalten für das Identifizieren einzelner `CNView`-Objekte bereitstellt und indirekten Input für den Kompositiosalgorithmus erzeugt. Das SUT kann dem Matcher Views übergeben und erhält

durch einen boolschen Rückgabewert die Informationen darüber, ob diese als Elternknoten dienen. Der nachfolgende Test übergibt während des Ladens der ThirdView einen Matcher, der die FirstView als Elternknoten erkennt. Als Ergebnis wird erwartet, dass die Verknüpfung zwischen der SecondView und der ThirdView ausbleibt. Die angestrebte Isolation der Tests durch die Verwendung von *Test Doubles* erzwingt die Ausführung dieser Klasse als *Interface*. Derivate von **CNMatcher** entsprechen den in Abschnitt 3.4 erläuterten Positions kriterien.

Rot

Die Implementierung der Testmethode erfolgt analog zu Listing 4.32. Der Aufruf der `load`-Methode im *Act* des Tests zum Laden der ThirdView bedarf einer Anpassung in der API der Klasse **CNDynamicViewHierarchy**, die auch die ersten beiden Methodenaufrufe beeinflusst. Daher müssen auch beim Laden der FirstView und der SecondView Matcher übergeben werden. Da der bisherige Algorithmus noch auf kein Verhalten der Matcher zugreift, ist die Verwendung von *Dummy Objects* zulässig. Für das Laden der ThirdView hingegen wird ein Matcher benötigt, der nur für die FirstView indirekten Input mit dem boolschen Wert `true` leistet. Eine Konfiguration durch den Test mit einem festen Rückgabewert ist daher nicht möglich. Deswegen handelt es sich bei diesem *Test Double* um ein *Fake Object*, das den indirekten Input basierend auf einer komplexen Entscheidung erzeugt. Zur Identifizierung des Elternknotens wird dem *Fake Objekt* die FirstView injiziert. Nach der Umsetzung der Testmethode sorgen die Modifikation der API nach Listing 4.33, die Implementierung des degenerierten *Interfaces* **CNMatcher** mit den zugehörigen *Test Doubles* sowie die Anpassung der übrigen Testmethoden durch die Verwendung von **CNMatcherDummy** für einen erfolgreichen *Failure Run*.

```
1 TEST_F(CNDynamicViewHierarchyTest ,  
2     LoadedFirstView_LoadedSecondViewMatching  
3         __LoadThirdViewMatchingFirstView  
4             __ThirdViewShouldNotBeAddedToSecondView) {  
5     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();  
6     CNViewPtr firstView = makeCNViewDummy();
```

```

7     sut->load(firstView, false, makeCNMatcherDummy());
8     CNViewSpyPtr secondView = makeCNViewSpy();
9     sut->load(secondView, true, makeCNMatcherDummy());
10
11    CNViewPtr thirdView = makeCNViewDummy();
12    sut->load(thirdView, true, makeCNFakeMatcher(firstView));
13
14    std::string errorMessage = getReceiverDidNotAddViewErrorMessage("SecondView", "ThirdView");
15    expectReceiverDidNotAddView(secondView, thirdView, errorMessage);
}

```

Listing 4.32: Zyklus 7 – Rot: Implementierung des siebten Testfalls.

```

1 public:
2     virtual void load(std::shared_ptr<CNView> view, bool matches,
3                         std::shared_ptr<CNMatcher> matcher);

```

Listing 4.33: Zyklus 7 – Rot: Modifikation der Signatur der `load`-Methode (`CNDynamicHierarchyView.h`).

Grün

Die von dem Test erwartete Funktionalität wird durch die Modifikation der `load`-Methode nach Listing 4.34 erreicht. Im letzten `else`-Block des Algorithmus ruft das SUT zunächst die `add`-Methode der `FirstView` auf, um die übergebene View als Kindknoten hinzuzufügen. Die anschließende Verknüpfung mit der `SecondView` geschieht basierend auf dem indirekten Input des Matchers. Dafür wird das *Interface* `CNMatcher` um die in Listing 4.35 dargestellte Methode `matches` erweitert. Aufgrund der Vererbung ergeben sich dadurch auch Anpassungen in den *Test Doubles*. Listing 4.36 zeigt die Implementierung der Klasse `CNFakeMatcher`. Während der `matches`-Methode vergleicht das *Fake Object* die übergebene View mit seinem Attribut `matchingView`, das durch den Test injiziert wird. Identifiziert der Matcher auf diese Weise die `FirstView` als Elternknoten, gibt er den Wert `true` zurück und das SUT erzeugt keine Verknüpfung zu der `SecondView`.

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

```
1 void CNDynamicViewHierarchy::load(CNViewPtr view, bool matches,
2                                     CNMatcherPtr matcher) {
3     if (!firstView) {
4         firstView = view;
5     } else if (!secondView) {
6         secondView = view;
7         if (matches) {
8             firstView->add(view);
9         }
10    } else {
11        if (matches) {
12            firstView->add(view);
13            if (!matcher->matches(firstView)) {
14                secondView->add(view);
15            }
16        }
17    }
18}
```

Listing 4.34: Zyklus 7 – Grün: Erweiterung der `load`-Methode um den Einfluss des `CNMatchers` in dem dritten Methodenaufruf.

```
1 class CNView;
2 class CNMatcher;
3 typedef std::shared_ptr<CNMatcher> CNMatcherPtr;
4
5 class CNMatcher {
6 public:
7     virtual ~CNMatcher() {};
8 protected:
9     CNMatcher() {};
10 public:
11     virtual bool matches(std::shared_ptr<CNView> view) = 0;
12};
```

Listing 4.35: Zyklus 7 – Grün: Erweiterung des `Interfaces CNMatcher` um die rein virtuelle Methode `matches`.

```

1 #include "CNMatcher.h"
2
3 class CNFakeMatcher;
4 typedef std::shared_ptr<CNFakeMatcher> CNFakeMatcherPtr;
5
6 class CNFakeMatcher : public CNMatcher {
7     public:
8         static CNFakeMatcherPtr newInstance(std::shared_ptr<CNView>
9             matchingView) {
10            return CNFakeMatcherPtr(new CNFakeMatcher(matchingView));
11        }
12        virtual ~CNFakeMatcher() {}
13
14     protected:
15         CNFakeMatcher(std::shared_ptr<CNView> matchingView) : matchingView(
16             matchingView) {};
17
18     public:
19         virtual bool matches(std::shared_ptr<CNView> view) override {
20             return matchingView == view;
21         };
22
23     private:
24         std::shared_ptr<CNView> matchingView;
25     };
26

```

Listing 4.36: Zyklus 7 – Grün: Erweiterung des `CNFakeMatchers` um die Methode `matches`.

Refactoring

Durch die Erweiterung der `load`-Methode um den Übergabeparameter vom Typ `CNMatcher` stehen dem Kompositionsalgorithmus zwei Informationsquellen für die Verknüpfung von Views zur Verfügung. Während der boolsche Wert der Variablen `matches` darüber entscheidet, ob die View in die Hierarchie geladen wird, gibt der Matcher anhand seiner `matches`-Methode Aufschluss über den Elternknoten. Beide Informationen können ebenfalls allein durch die Verwendung der Matcher generiert wer-

den, da das Identifizieren eines Elternknotens auch ein mögliches Hinzufügen der View als Kindknoten impliziert. Das Umstellen des gesamten Algorithmus auf die Verwendung der Matcher erlaubt es daher, den boolschen Übergabeparameter zu eliminieren. Zunächst erfolgt dafür die Anpassung der Tests in den Aufrufen der `load`-Methoden. Da der Algorithmus die `matches`-Methode der übergebenen Matcher aufruft, ist die Verwendung von *Dummy Objects* nicht zulässig. Stattdessen erfordert eine zuverlässige Ausführung der Tests die Übergabe von *Test Doubles*, die indirekten Input entsprechend des boolschen Übergabeparameters erzeugen. Der Aufruf der `load`-Methode mit dem Wert `false` bedeutet, dass der zugehörige Matcher keinen Elternknoten erkennen darf. Infolgedessen wird an diesen Stellen ein *Test Stub* übergeben, den der Test mit dem Wert `false` konfiguriert. Zur Instanziierung und Konfiguration des *Test Doubles* wird das *Test Fixture* um die in Listing 4.37 dargestellte Fabrik-Methode erweitert.

```

1   virtual CNMatcherStubPtr makeNotMatchingMatcher() {
2       CNMatcherStubPtr matcher = CNMatcherStub::getInstance();
3       matcher->setIsMatching(false);
4       return matcher;
5   }

```

Listing 4.37: Zyklus 7 – Refactoring: Erweiterung des *Test Fixtures* um eine Fabrik-Methode zur Instanziierung und Konfiguration der Klasse `CNMatcherStub`.

Die Übergabe des boolschen Werts `true` veranlasst den Algorithmus, die View einem Elternknoten hinzuzufügen. Der Elternknoten wird dabei durch die `load`-Methode bestimmt. Um den Algorithmus in die korrekte Funktionsweise unter Verwendung der Matcher zu überführen, müssen in diesen Methodenaufrufen `CNFakeMatcher` übergeben werden, die den von dem Test erwarteten Elternknoten identifizieren. Listing 4.38 zeigt exemplarisch die Anpassung der ersten Testmethode. Die `load`-Methode kann anschließend entsprechend Listing 4.39 modifiziert werden. Da der Kompositionsalgorithmus nach der Anpassung der übrigen Tests die Variable `matches` nicht mehr verwendet, kann sie in einem nächsten Schritt aus der Methodensignatur entfernt werden.

```

1 TEST_F(CNDynamicViewHierarchyTest, LoadedFirstViewNotMatching
2     _LoadSecondViewMatchingFirstView
3     _SecondViewShouldBeAddedToFirstView) {
4     CNDynamicViewHierarchyPtr sut = makeCNDynamicViewHierarchy();
5     CNViewSpyPtr firstView = makeCNViewSpy();
6     sut->load(firstView, false, makeNotMatchingMatcher());
7
8     CNViewPtr secondView = makeCNViewDummy();
9     sut->load(secondView, true, makeCNFakeMatcher(firstView));
10
11    std::string errorMessage = getReceiverAddedViewErrorMessage("FirstView",
12        "SecondView");
13    expectReceiverAddedView(firstView, secondView, errorMessage);
14 }
```

Listing 4.38: Zyklus 7 – Refactoring: Anpassen der Tests an die CNMatcher.

```

1 void CNDynamicViewHierarchy::load(CNViewPtr view, bool matches,
2                                     CNMatcherPtr matcher) {
3     if (!firstView) {
4         firstView = view;
5     } else if (!secondView) {
6         secondView = view;
7         if (matcher->matches(firstView)) {
8             firstView->add(view);
9         }
10    } else {
11        if (matcher->matches(firstView)) {
12            firstView->add(view);
13        }
14        if (matcher->matches(secondView)) {
15            secondView->add(view);
16        }
17    }
18 }
```

Listing 4.39: Zyklus 7 – Refactoring: Umstellung des Algorithmus auf die Verwendung der CNMatcher.

Im weiteren Verlauf dieser Phase erfolgt die Vereinfachung des Kompositionsalgorithmus durch die Umstrukturierung des Quelltextes. Als erstes werden dafür die Attribute `firstView` und `secondView` der Klasse `CNDynamicViewHierarchy` durch das Attribut `views` ersetzt. Dabei handelt es sich um einen `vector`, der `shared_ptr` des Typs `CNView` listenartig speichert. Dadurch ergeben sich Änderungen der `load`-Methode nach Listing 4.40. Anstatt sich die übergebenen Views in den Attributen `firstView` und `secondView` zu merken, fügt die `load`-Methode sie durch den Aufruf der `push_back`-Methode dem `vector` hinzu. Infolgedessen entspricht die `firstView` dem ersten und die `secondView` dem zweiten Eintrag der Liste. Für die Überprüfung der `firstView` und der `secondView` in den `if`-Anweisungen verwendet die `CNDynamicViewHierarchy` die `size`-Methode des `vectors`, die die Anzahl der gespeicherten Views zurückgibt.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , CNMatcherPtr matcher) {
2     if (views . size () == 0) {
3         views . push_back (view);
4     } else if (views . size () == 1) {
5         views . push_back (view);
6         if (matcher->matches (views [0])) {
7             views [0]->add (view);
8         }
9     } else {
10        if (matcher->matches (views [0])) {
11            views [0]->add (view);
12        }
13        if (matcher->matches (views [1])) {
14            views [1]->add (view);
15        }
16    }
17 }
```

Listing 4.40: Zyklus 7 – Refactoring: Verwendung eines `vectors` zum Speichern von Views (`CNDynamicViewHierarchy.cpp`).

Des Weiteren fällt auf, dass sowohl im ersten als auch im zweiten Block der `if-else`-Anweisung ein Aufruf der `push_back`-Methode erfolgt. Im letzten `else`-Block bleibt dieser Aufruf aus. Die schrittweise Erweiterung der Funktionalität zeigt jedoch die Tendenz auf, dass die `CNDynamicViewHierarchy` mit jeder neuen Ebene der Hierarchie eine weitere View speichern muss. Die Erweiterung der Funktionalität um eine vierte Hierarchieebene würde daher auch den letzten `else`-Block zu einem Aufruf der `push_back`-Methode zwingen. Da dann jeder Pfad das Abspeichern der übergebenen View vornimmt, kann die `load`-Methode den Aufruf der `push_back`-Methode analog zu Listing 4.41 auch nach der Verzweigung ausführen. Es entsteht ein leerer Anweisungsblock des ersten Pfads, der somit eliminiert werden kann.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , CNMatcherPtr matcher) {
2     if (views . size () == 1) {
3         if (matcher->matches (views [0])) {
4             views[0]->add (view);
5         }
6     } else {
7         if (matcher->matches (views [0])) {
8             views[0]->add (view);
9         }
10        if (matcher->matches (views [1])) {
11            views[1]->add (view);
12        }
13    }
14
15    views . push_back (view);
}

```

Listing 4.41: Zyklus 7 – *Refactoring*: Vorziehen des Aufrufs der `push_back`-Methode der Klasse `vector`.

Zur weiteren Vereinfachung des Algorithmus findet die Strategie *Reconcile Differences* nach Beck (2003) Anwendung, in der unterschiedliche Ausschnitte von Quelltext durch geschicktes *Refactoring* einander angeglichen und anschließend zusammengeführt werden. Im Zuge dieser Strategie wird versucht, die Verzweigung der `load`-Methode durch

4. Testgetriebene Entwicklung des Kompositionsalgorithmus

einen allgemeingültigen Pfad zu ersetzen. Dabei darf die Verzweigung des Algorithmus nur eliminiert werden, wenn alle Pfade die gleiche Logik enthalten. Eine erste Annäherung geschieht durch die Verwendung von `for`-Schleifen in beiden Pfaden der Verzweigung. Aufgrund dieser Umstrukturierung des Algorithmus unterscheiden sich die Pfade nur noch in der oberen Grenze der Laufvariablen `i`.

```
void CNDynamicViewHierarchy::load(CNViewPtr view, CNMatcherPtr matcher) {
    if (views.size() == 1) {
        for (int i = 0; i < 1; i++) {
            if (matcher->matches(views[i])) {
                views[i]->add(view);
            }
        }
    } else {
        for (int i = 0; i < 2; i++) {
            if (matcher->matches(views[i])) {
                views[i]->add(view);
            }
        }
    }

    views.push_back(view);
}
```

Listing 4.42: Zyklus 7 – Refactoring: Umschließen der Aufrufe der `load`-Methode durch `for`-Schleifen mit statischen Zählvariablen.

Die Aufrufe der Methoden `matches` und `add` im oberen Pfad der Verzweigung erfolgen nur für den ersten Eintrag des `vectors`. Daher ergibt sich hier eine Grenze mit dem Wert 1. Die Schleife des unteren Pfads bricht bei einer Grenze von 2 ab. Die Bedingung der `if`-Anweisung stellt sicher, dass sich bei der Ausführung des oberen Pfads nur eine View in der Liste befindet. Infolgedessen kann die Grenze der `for`-Schleife wie in Listing 4.43 durch den Aufruf der `size`-Methode ersetzt werden. Da die Testfälle bisher nur drei aufeinanderfolgende Aufrufe der `load`-Methode ausführen, ist ebenfalls sichergestellt, dass sich in der Liste bei der Ausführung des `else`-Blocks zwei Views

befinden. Daher ist auch im unteren Pfad der Verzweigung eine Grenze zulässig, die der Größe des `vectors` entspricht. Die nun identische Logik in beiden Pfaden erlaubt das Entfernen der Verzweigung. Es entsteht ein Algorithmus nach Listing 4.44.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , CNMatcherPtr matcher) {
2     if (views . size () == 1) {
3         for (int i = 0; i < views . size (); i++) {
4             if (matcher->matches (views [ i ])) {
5                 views [ i ]->add (view);
6             }
7         }
8     } else {
9         for (int i = 0; i < views . size (); i++) {
10            if (matcher->matches (views [ i ])) {
11                views [ i ]->add (view);
12            }
13        }
14    }
15
16    views . push_back (view);
17 }
```

Listing 4.43: Zyklus 7 – Refactoring: Ersetzen der statischen Zählvariablen durch die Größe des `vectors`.

```

1 void CNDynamicViewHierarchy :: load (CNViewPtr view , CNMatcherPtr matcher) {
2     for (int i = 0; i < views . size (); i++)
3         if (matcher->matches (views [ i ]))
4             views [ i ]->add (view);
5
6     views . push_back (view);
7 }
```

Listing 4.44: Zyklus 7 – Refactoring: Eliminieren des `if-else`-Blocks.

4.8 Zusammenfassung

Dieses Kapitel veranschaulicht die Anwendung der testgetriebenen Entwicklung am Beispiel der Entstehung des Kompositionsalgorithmus, bei der die in Kapitel 3 erarbeiteten Vorstellungen über die Funktionsweise und Strukturen von CrossNative als Orientierung dienen. Es werden sieben Durchläufe des dreiphasigen Entwurfszyklus gezeigt, die nach der Strategie der Triangulation (Beck, 2003) sukzessiv eine fehlerfreie und schlanke Implementierung des Kompositionsalgorithmus hervorbringen. Dabei beschreiben kleine Beispiele und Codeausschnitte die Wirkungsweise dieser Entwurfsmethodik auf die Klassenbeziehungen des SUT sowie die Implementierung der *Unit of Work*. Besonders in den ersten beiden Entwurfszyklen wird das aktive Designen des Entwicklers durch das Feedback infolge der vorgezogenen und iterativen Implementierung von Testmethoden in der Phase Rot hervorgehoben. Es wird gezeigt, wie das Feedback im Rahmen des *Intentional Programmings* dabei unterstützt die konkrete Funktionsweise des umzusetzenden Mechanismus zu erarbeiten, ohne sich auf die Implementierungsdetails, das objektorientierte Design oder die fehlerfreie Umsetzung zu konzentrieren. Dadurch entstehen beteiligte Klassen und Schnittstellen sowie deren APIs, die leicht verständlich und einfach anwendbar sind. Ein Beispiel für diese Wirkungsweise liefert die Implementierung der ersten Testmethode, durch die der Benutzer die Namen der beteiligten Klassen, Schnittstellen sowie die Signaturen und Rückgabetypen der aufgerufenen Methoden festlegt.

Auch wird gezeigt, wie die inkrementelle Vorgehensweise der testgetriebenen Entwicklung die Entscheidungen während des Entwurfsprozesses entzerrt und den Entwickler entlastet. Diesen Vorgang verdeutlicht die Evolution des boolschen Übergabeparameters, der im zweiten Entwurfszyklus eingeführt wird und sich entsprechend den Anforderungen der nachfolgenden Tests bis zu einem komplexen Objekt anpasst. Schon in der Konzeption in Kapitel 3 wird für die Beeinflussung des Elternknotens die Notwendigkeit eines komplexen Identifizierungsmechanismus erkannt (siehe Positionskriterien, Abschnitt 3.4). Dennoch beginnt die testgetriebene Entwicklung mit der Übergabe eines boolschen Parameters, da dieser zunächst zur Erfüllung der momentan geforderten Funktionalität des zweiten Zyklus ausreicht. Erst im siebten Zyklus benötigt das System durch einen

komplexeren Test den Identifizierungsmechanismus, für den das *Interface CNMatcher* entworfen wird. Diese Vorgehensweise erlaubt es die Entscheidungen über die Verantwortung (Zyklus 2) von der Funktionsweise (Zyklus 7) des Identifizierungsmechanismus zu trennen.

Darüber hinaus schafft die Diskussion über das Feedback der verschiedenen Umsetzungsweisen des Tests im zweiten Zyklus der Entwicklung Bewusstsein für die Rolle des Formulierens der Testmethoden. Unter anderem unterstützt das dabei generierte Feedback den Entwickler dabei die Semantik verschiedener Umsetzungsweisen zu deuten. Auch wird ersichtlich, wie die isolierte Betrachtung der *Unit Tests* durch die Verwendung von *Test Doubles* das SUT mit Hilfe der *Interfaces CNView* und *CNMatcher* entkoppelt. Infolgedessen werden das *Dependency Inversion Principle* und das *Open/Closed Principle* eingehalten und es entsteht ein lose gekoppeltes System. Weiterhin zeigt dieses Kapitel wie durch die Eliminierung von Redundanzen der Testmethoden durch Helper-Methoden in den *Test Fixtures Unit Tests* implementiert werden können, die aufgrund ihrer Wartbarkeit und Lesbarkeit eine hohe Qualität aufweisen.

KAPITEL 5

Entstehung des Framework CrossNative

Kapitel 4 beschreibt den Einfluss der testgetriebenen Entwicklung auf die Gestaltung einer einzelnen *Unit of Work*. Die Mechanismen dieser Entwicklungsmethodik wirken sich jedoch ebenfalls auf das Design der gesamten Software aus. Um die Einflüsse von TDD auf ein komplexes Klassensystem zu veranschaulichen, wird die Betrachtung in dem folgenden Kapitel von einem einzelnen SUT auf das gesamte Frameworks CrossNative ausgeweitet. Dabei wird weniger auf die Implementierung einzelner Testmethoden eingegangen, sondern vielmehr auf die daraus resultierenden Evolutionsschritte des Softwaresystems. Zur besseren Verständlichkeit wird dafür zunächst ein Anwendungsbeispiel für die Komposition einer grafischen Oberfläche gegeben, an dem sich der Entwurf von CrossNative orientiert.

In der Darstellung des Quelltextes wird auf die übliche Aufteilung von C++ in *Header-* und Implementierungsdateien verzichtet, um die Anzahl an Listings zu minimieren und die Verständlichkeit zu steigern. Der in diesem Kapitel gezeigte Quelltext dient nur zur Erläuterung der Funktionsweise bestimmter Mechanismen von CrossNative und entspricht nicht der tatsächlichen Umsetzung, für die auf den digital beigefügten Quelltext verwiesen wird.

5.1 Anwendungsbeispiel

Zur Erläuterung der Entstehung von CrossNative wird das nachfolgend beschriebene Anwendungsbeispiel herangezogen. Es soll eine GUI erstellt werden, die eine Menüleiste sowie ein Menü mit zwei Einträgen enthält. Unter Verwendung von Cocoa ergibt sich dafür aus der internen Sicht, wie bereits in Abschnitt 3.1.2 beschrieben, eine framework-spezifische Objekthierarchie nach Abbildung 5.1. Für die Menüleiste wird eine Instanz der Klasse `NSMenu` verwendet, die Objekte vom Typ `NSMenuItem` aufnehmen kann. Versieht man das `NSMenuItem` wiederum mit einem `NSMenu`, entsteht ein Untermenü, das in der Menüleiste angezeigt wird. Das Hinzufügen von Instanzen der Klasse `NSMenuItem` zu einem Untermenü führt zu den geforderten Menüeinträgen.

Aus der frameworkspezifischen Objekthierarchie leitet sich eine applikationspezifische View-Hierarchie nach Abbildung 5.2 ab. Im Kontext der Applikation übernimmt die Klasse `CocoaMenubar` die korrekte Anzeige der Menüleiste. Die Klasse `CocoaMenu` repräsentiert ein applikationsspezifisches Untermenü, das bei einem anderen Menü oder der Menüleiste registriert werden kann. Sie übernimmt sowohl die Verantwortung über das `NSMenuItem` als auch über das `NSMenu`, das als Untermenü registriert ist. Die Klasse `CocoaMenuItem` spiegelt die View des Menüeintrags wider.

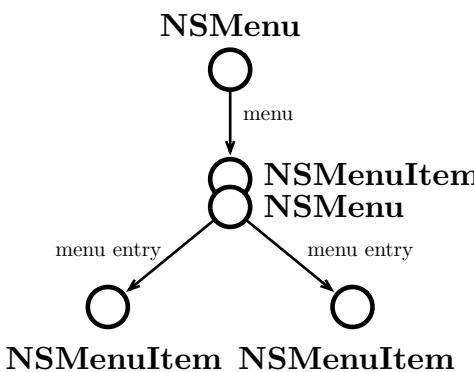


Abbildung 5.1: Frameworkspezifische Objekthierarchie des Anwendungsbeispiels.

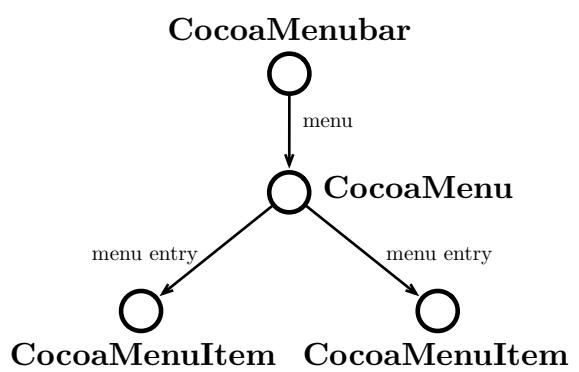


Abbildung 5.2: Applikationsspezifische Objekthierarchie des Anwendungsbeispiels.

Um den applikationsspezifischen View-Klassen Kontrolle über die GUI-Elemente des Frameworks zu ermöglichen, sind grundsätzlich zwei Ansätze denkbar. Zum einen kann

die Vererbung verwendet werden, um den Views Zugriff auf das Verhalten der GUI-Klassen zu erlauben. Dieser Ansatz führt zu einer starken Abhängigkeit zu dem verwendeten GUI-Framework, da neben dem öffentlichen auch auf das geschützte Verhalten zugegriffen werden kann. Die Abhängigkeit beschränkt sich daher nicht auf die öffentliche API des Frameworks, sondern beinhaltet auch Details dessen Implementierung. Des Weiteren sind die View-Klassen thematisch in dem Framework CrossNative angesiedelt. Vererbungsbeziehung zu abstrakten Klassen und *Interfaces* von CrossNative treten daher mit hoher Wahrscheinlichkeit auf. Zusätzliche Vererbungsbeziehungen der Views zu den Implementierungsklassen des GUI-Frameworks führen daher zu echter Mehrfachvererbung, die ein starres Klassensystem und somit unflexible Software hervorruft.

Ein weiterer Ansatz findet sich in der Komposition. Anstatt die applikationsspezifischen Views von den Klassen des Frameworks abzuleiten, setzen sie sich aus diesen zusammen. Die View-Klassen halten Attribute vom Typ der grafischen Repräsentationen des verwendeten GUI-Frameworks und hängen dadurch nur von deren öffentlichen API ab. Um einerseits die Abhängigkeit zu dem externen Quelltext zu verringern und andererseits die möglicherweise auftretende Mehrfachvererbung zu verhindern, wird für den Entwurf mit CrossNative die Komposition bevorzugt. Darüber hinaus gilt zu beachten, dass die Umsetzung von Cocoa mit der Sprache *Objective-C* die Verwendung der Vererbung in diesem Anwendungsbeispiel unterbindet, da eine gleichzeitige Vererbungsbeziehung zu einer C++-basierten Klasse nicht möglich ist. Bei dem Entwurf einer Oberfläche mit Cocoa übernehmen die View-Klassen zusätzlich die Aufgabe der in Abschnitt 3.1.2 erwähnten Zwischenschicht in *Objective-C++*, die es ermöglicht, die Funktionalität der in *Objective-C* implementierten Objekte aus C++ aufzurufen.

Für das Erstellen der applikationsspezifischen Objekthierarchie nach Abbildung 5.2 ist unter anderem die Verknüpfung von **CocoaMenubar** und **CocoaMenu** notwendig. Zu diesem Zweck erhält die Klasse **CocoaMenubar** analog zu Abbildung 5.3 eine **add**-Methode, die Instanzen der Klasse **CocoaMenu** aufnimmt und somit die Verknüpfung herstellt. Auch **CocoaMenu** wird um eine **add**-Methode erweitert, durch die eine Verknüpfung zwischen den Menüs und den Menüeinträgen erreicht wird. Zu jeder **add**-Methode existiert eine korrespondierende **remove**-Methode, die eine zuvor erstellte Verknüpfung aufhebt.

Neben den View-Objekten müssen für eine fehlerfreie grafische Benutzeroberfläche auch die frameworkspezifischen GUI-Objekte nach Abbildung 5.1 verknüpft werden. Infolgedessen extrahiert die **CocoaMenubar** während des Aufrufs der **add**-Methode die grafische Repräsentation des **CocoaMenus** und registriert diese analog zu Abbildung 5.3 bei ihrem eigenen **NSMenu**.

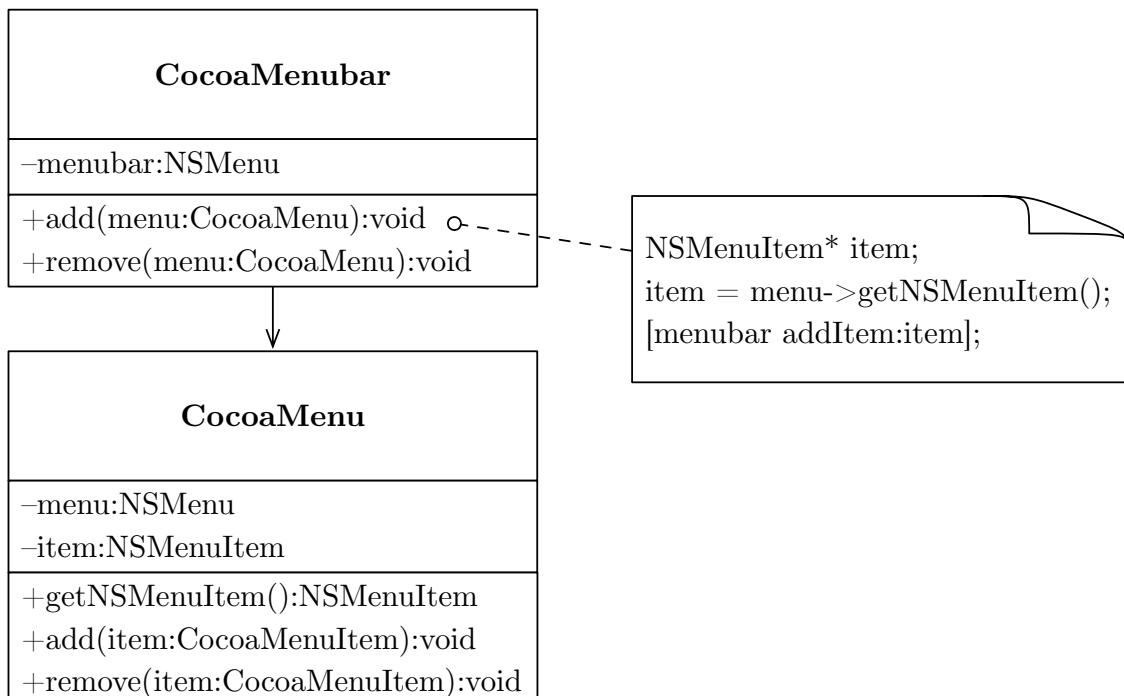


Abbildung 5.3: Verknüpfen der Menüleiste mit dem Menü.

Die direkte Abhängigkeit von **CocoaMenubar** zu **CocoaMenu** verstößt gegen das *Open/Closed Principle* und das *Dependency Inversion Principle* und unterbindet die in Abschnitt 3.3.4 geforderte einfache Austauschbarkeit der View-Implementierung. Zur Entkopplung der **CocoaMenubar** und zur Erfüllung der zuvor genannten Prinzipien wird nach Abbildung 5.4 die Schnittstelle **NSMenuItemBased** zwischen den beiden Klassen eingeführt, die die Abhängigkeit invertiert. Dadurch basiert die Klasse **CocoaMenubar** nur noch auf der Methode `getNSMenuItem`. Auch zwischen **CocoaMenu** und **CocoaMenuItem** wird nach dem selben Prinzip eine Schnittstelle verwendet, die den einfachen Austausch der Implementierung des Menüeintrags ermöglicht. Da auch die Menüeinträge durch die GUI-Klasse **NSMenuItem** repräsentiert werden, dient hier ebenfalls das Interface **NSMenuItemBased** zur Entkopplung der Klasse **CocoaMenu**.

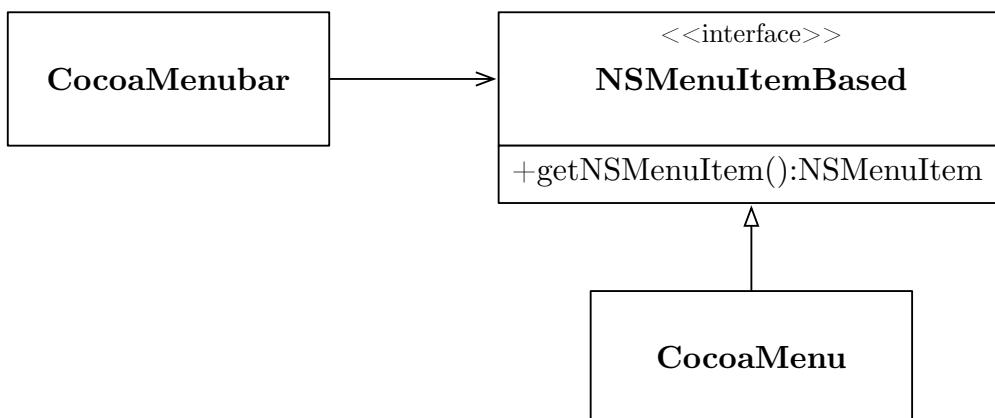


Abbildung 5.4: Umsetzung des *Dependency Inversion Principles* und des *Open/Closed Principles* im Klassensystem der Views.

5.2 Basis der Objekthierarchie

Im Anschluss an die in Kapitel 4 vorgeführte testgetriebene Entwicklung des Kompositonsalgorithmus erfolgt eine Erweiterung der Klasse `CNDynamicViewHierarchy` um eine `unload`-Methode, die eine zuvor geladene `CNView` aus der Objekthierarchie entfernt. Unter anderem erfordert diese Erweiterung gleichzeitig die Modifikation der Schnittstelle `CNView`. Neben den Methoden `add` und `remove` erhält das *Interface* weitere Methoden zur Verwaltung von Kindknoten. Da sich das System durch diese Veränderung dem *Composite Pattern* nach Gamma et al. (2004) annähert, wird das *Interface* `CNView` in Anlehnung an dieses Entwurfsmuster analog zu Abbildung 5.5 in `CNComponent` umbenannt. Gleichzeitig ändert sich ebenfalls der Name der dynamischen Hierarchie in `CNDynamicHierarchy`.

Unter Verwendung von CrossNative soll aus den View-Klassen des Anwendungsbeispiels mit Hilfe der `CNDynamicHierarchy` die gewünschte Oberfläche zusammengesetzt werden. Die einfachste Lösung dafür bietet eine Vererbungsbeziehung zu dem zuvor gezeigten *Interface* `CNComponent`, das im Kontext von CrossNative als Basis für Objekthierarchien dient. Die einheitliche Behandlung der einzelnen Knoten in der Hierarchie durch den Typ `CNComponent` erlaubt es durch die Verwendung von Polymorphie einerseits das System leicht um neue Knotentypen zu erweitern, andererseits die Zusammensetzung der Knoten in der Hierarchie dynamisch zu variieren.

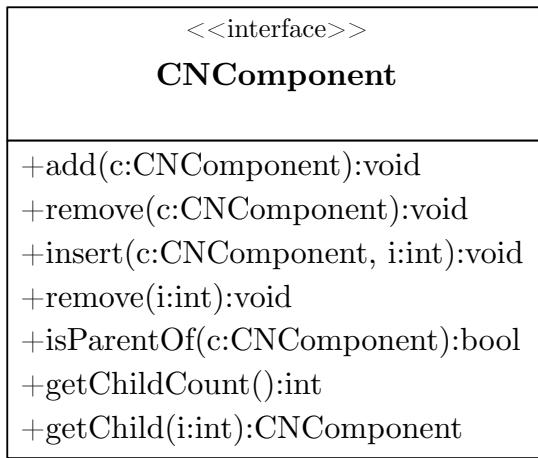


Abbildung 5.5: **CNComponent** als Schnittstelle des *Composite Patterns* in Anlehnung an Gamma et al. (2004).

Infolge der Vererbung der View-Klassen zu **CNComponent** müssen ihre Methoden `add` und `remove` entsprechend der API des *Interfaces* angepasst werden. Anstatt eines Objekts vom Typ **NSMenuItemBased** erhält beispielsweise die `add`-Methode der **CocoaMenubar** dann Instanzen vom Typ **CNComponent**. Damit die Menüleiste die grafischen Repräsentationen dennoch korrekt verknüpfen kann, muss eine Typumwandlung zu **NSMenuItemBased** stattfinden. An dieser Stelle verdeutlicht sich der Nachteil des *Composite Patterns*. Durch die generalisierten Übergabeparameter der Methoden für das Verwalten der Kinder verliert das System an Typsicherheit und erlaubt es ebenfalls der Menüleiste Derivate zu übergeben, die nicht zugleich auch von **NSMenuItemBased** abgeleitet sind. Die Verwendung des *Composite Patterns* ermöglicht daher keine typbasierten Einschränkungen bei dem Verknüpfen von Knoten. Ist die Funktionalität eines Derivats an den konkreten Typ eines anderen Derivats gebunden, müssen die Subtypen beim Verknüpfen der Knoten zur Laufzeit überprüft werden. Handelt es sich beispielsweise bei der hinzugefügten **CNComponent** tatsächlich auch um ein Objekt des Typs **NSMenuItemBased**, verknüpft die Menüleiste die grafischen Repräsentationen. In allen anderen Fällen wird dem Benutzer ein Ausnahmefehler ausgegeben, der Aufschluss über die fehlerhafte Benutzung des Systems gibt.

5.3 Zusammensetzbare Knoten der Hierarchie

Neben der Verknüpfung der grafischen Repräsentationen müssen auch die Verbindungen zwischen den `CNComponents` hergestellt werden. Zu diesem Zweck enthalten die View-Klassen, wie in Listing 5.1 am Beispiel der `CocoaMenubar` veranschaulicht, einen `vector`, in dem sie Referenzen auf die hinzugefügten Kindknoten speichern.

```

1 class CocoaMenubar : public CNComponent {
2 // ...
3 public:
4     virtual void add(CNComponentPtr c) override {
5         NSMenuItemBasedPtr item;
6         item = std::dynamic_pointer_cast<NSMenuItemBased>(c);
7         if (!item) throw std::exception();
8         [menubar addItem:item->getNSMenuItem()];
9
10        children.push_back(c);
11    }
12
13 private:
14     NSMenu* menubar;
15     std::vector<CNComponentPtr> children;
16 }
```

Listing 5.1: Implementierung der Klasse `CocoaMenubar` durch Ableiten von `CNComponent`.

Zunächst erfolgt in der `add`-Methode die dynamische Umwandlung der übergebenen `CNComponent` zu dem Typ `NSMenuItemBased`. Schlägt diese fehl, ist die Menüleiste nicht in der Lage, die grafischen Repräsentationen zu verknüpfen und löst eine *Exception* aus. Eine erfolgreiche Umwandlung des hinzugefügten Kindknotens hingegen führt zu der Verknüpfung der GUI-Objekte und dem anschließenden Speichern der `CNComponent` in dem `vector`.

5. Entstehung des Framework CrossNative

Die Umsetzung der Klasse `CocoaMenu` nach Listing 5.2 ergibt eine nahezu identische Implementierung. Auch hier wird nach erfolgreicher Typumwandlung die Verknüpfung der grafischen Repräsentationen vollzogen und anschließend eine Referenz des Kinds gespeichert.

```
1 class CocoaMenu : public CNComponent {
2 // ...
3 public:
4     virtual void add(CNComponentPtr c) override {
5         NSMenuItemBasedPtr* item;
6         item = std::dynamic_pointer_cast<NSMenuItemBased>(c);
7         if (!item) throw std::exception();
8         [menu addItem:item->getNSMenuItem()];
9
10        children.push_back(c);
11    }
12
13 private:
14     NSMenuItem* menuItem;
15     NSMenu* menu;
16     std::vector<CNComponentPtr> children;
17 }
```

Listing 5.2: Implementierung der Klasse `CocoaMenu` durch Ableiten von `CNComponent`.

Da jede View-Klasse neben den viewspezifischen Aufgaben zusätzlich die Logik für das Verwalten der Kinder enthält, verstößt diese Lösung gegen das *Single Responsibility Principle*. Des Weiteren minimiert sich infolge der entstehenden Redundanz die Wartbarkeit des Quelltextes. Darüber hinaus ergibt sich jedoch als entscheidender Nachteil dieser Lösung die mangelnde Testbarkeit der View-Klassen, die das Testen der Logik für das Verwalten von Kindknoten verhindert. Die Verwendung von Implementierungsklassen des *third-party* Systems Cocoa ermöglicht kein Austausch der externen Abhängigkeiten durch ein Framework-unabhängiges *Test Double*. Das Testen der View-Klassen müsste daher unter der Verwendung der echten Abhängigkeiten geschehen, die nach Abschnitt 2.3 sowohl die Konsistenz der Ergebnisse, als auch die Ausführungsgeschwindigkeit der

Tests beeinflussen könnten. Die Formulierung zuverlässiger Tests wird dadurch unterbunden. Leiten sich die View-Klassen von der Schnittstelle **CNComponent** ab, vermengt sich die eigentlich testbare GUI-unabhängige Logik mit der GUI-abhängigen Logik der View-Klassen und verliert ebenfalls ihre Testbarkeit.

Die Anwendung der Komposition trennt die beiden Verantwortungen in unterschiedliche Klassen auf, wodurch einerseits der Verstoß gegen das zuvor genannte Prinzip, andererseits die fehlende Testbarkeit der Logik für das Verwalten von Kindknoten adressiert wird. Dafür erfährt das System eine Modifikation nach Abbildung 5.6.

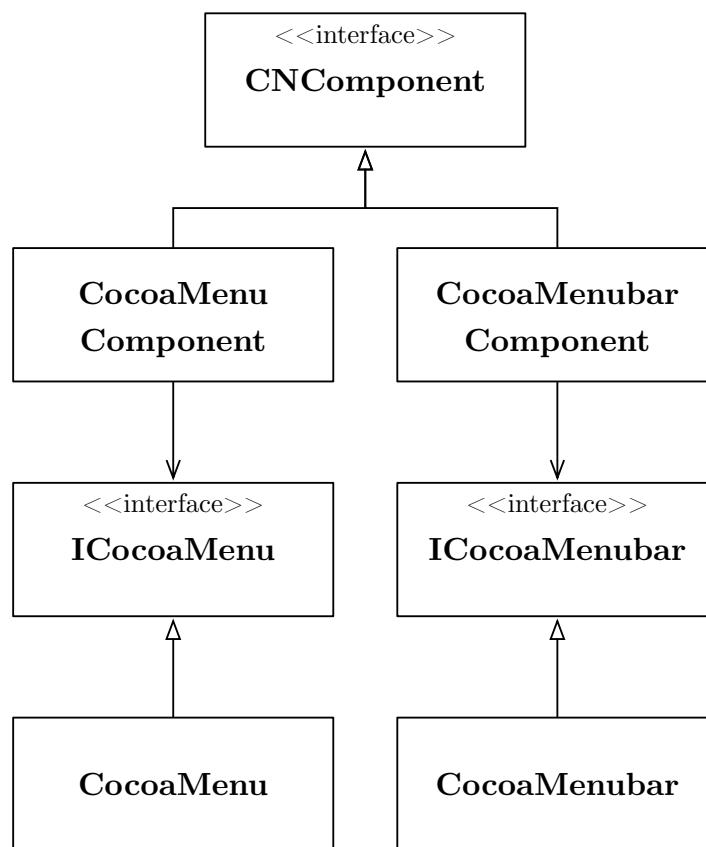


Abbildung 5.6: Verwendung der Komposition für die Umsetzung verschiedener View-Knoten.

Für jede View-Klasse entsteht ein eigenes **CNComponent**-Derivat, das die Verwaltung der Kinder entsprechend des *Composite Patterns* übernimmt. Da sich die Views nun nicht mehr selbst von der Schnittstelle ableiten, sind sie nicht an die API von **CNComponent**

gebunden. Aus diesem Grund erhalten sie ihre Typsicherheit in den Methoden `add` und `remove` nach Abschnitt 5.1 zurück. Die *Interfaces* der View-Klassen ergeben sich aufgrund der angestrebten Isolation der Tests der `CNComponent`-Derivate durch die Verwendung von *Test Doubles* nach Abschnitt 2.4. Sie stellen den `CNComponent`-Derivaten zur Verknüpfung der Views die typspezifischen Methoden `add` und `remove` zur Verfügung.

Da eine erfolgreiche Typumwandlung des übergebenen Knotens während der `add`-Methode der `CNComponent`-Derivate wegen der Verschachtelung der Objekte nicht mehr möglich ist, erfährt auch der Verknüpfungsmechanismus eine Modifikation. Anstatt den konkreten Typ des übergebenen `CNComponents` zu überprüfen, muss der neue Mechanismus in der Lage sein, den Typ der gekapselten View-Klasse zu identifizieren. Eine objektorientierte Lösung für die Rückgewinnung von Typinformationen bietet das *Ayclic Visitor Pattern* (azyklisches Besuchermuster) nach Martin et al. (2008). Abbildung 5.7 veranschaulicht die Verwendung dieses Entwurfsmusters im Rahmen des Frameworks CrossNative. Zu den Hauptbestandteilen des azyklischen Besuchermusters gehören zum einen das degenerierte *Interface* `CNVisitor` sowie die typbasierten Besucherschnittstellen, die durch die einzelnen konkreten View-Klassen vorgegeben werden. Dabei enthält jede Besucherschnittstelle eine `visit`-Methode, die die entsprechende View-Klasse als Übergabeparamter entgegennimmt. Durch die Vererbungsbeziehung zu dem *Interface* `CNVisitable` sind die View-Klassen gezwungen, eine Implementierung für die `accept`-Methode bereitzustellen. Während dieser Methode erfolgt eine dynamische Typumwandlung des übergebenen `CNVisitors` zu der jeweiligen Besucherschnittstelle. Gelingt die Umwandlung, ruft die View-Klasse die `visit`-Methode des Besuchers auf und übergibt sich selbst als Parameter. Durch den Aufruf der typspezifischen `visit`-Methode identifiziert der Besucher den konkreten Typ der View. Schlägt die Typumwandlung fehl, löst die View eine `CNVisitableVisitorMismatchException` aus, die zum Ausdruck bringt, dass der übergebene `CNVisitor` die View nicht identifiziert. Diese Sequenz an Methodenaufrufen wird im Zusammenhang mit dem Besuchermuster als *double-dispatch*-Mechanismus bezeichnet.

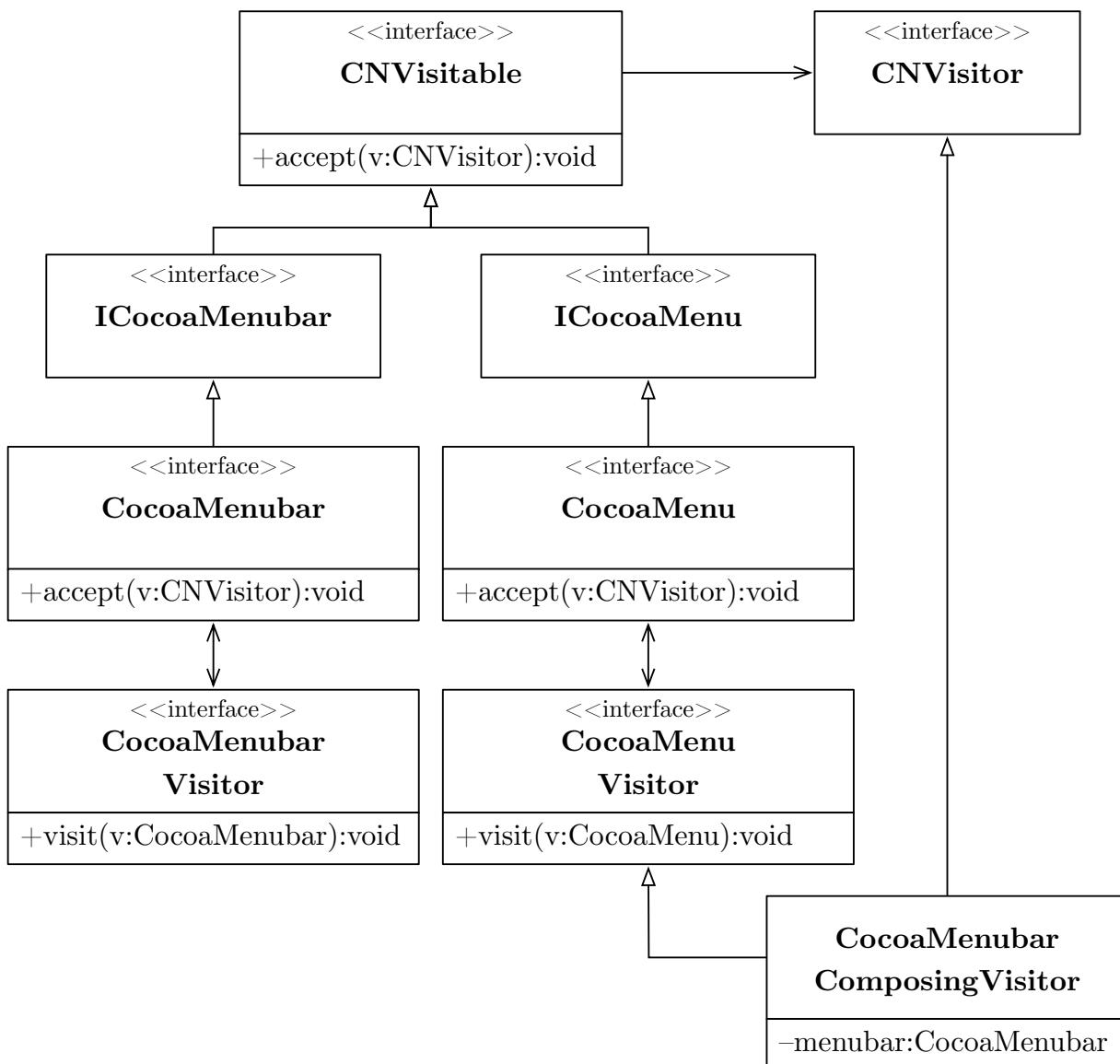


Abbildung 5.7: Verwendung der *Acyclic Visitor Patterns* in Anlehnung an Martin et al. (2008) zur Verknüpfung der Views.

Listing 5.3 zeigt exemplarisch die Implementierung der `accept`-Methode des **CocoaMenus**. Zu beachten gilt dabei, dass der `visit`-Methode aufgrund der Verwendung von `shared_ptr` nach Abschnitt 3.5 nicht `this` übergeben werden kann. Statt dessen wird ein `shared_ptr` benötigt, der `this` enthält. Zu diesem Zweck verwendet das **CocoaMenu** die `shared_from_this`-Methode, die der Vererbung zu der Klasse `enable_shared_from_this` der C++-Standardbibliothek entstammt.

```

1 class CocoaMenu : public ICocoaMenu,
2                         public std::enable_shared_from_this<CocoaMenu> {
3 // ...
4 public:
5     virtual void accept(CNVisitorPtr v) override {
6         CocoaMenuVisitorPtr menuVisitor;
7         menuVisitor = std::dynamic_pointer_cast<CocoaMenuVisitor>(v);
8
9         if (!menuVisitor)
10            throw CNVisitableVisitorMismatchException();
11
12         menuVisitor->visit(this->shared_from_this());
13     }
14 // ...
15 }
```

Listing 5.3: Implementierung der `accept`-Methode der Klasse `CocoaMenu`.

Die Rückgewinnung der Typinformationen durch den *double-dispatch*-Mechanismus ermöglicht konkreten Besucherklassen, die typbasierten Verknüpfungen der Views bzw. der grafischen Repräsentationen vorzunehmen. Zur Identifizierung aller möglichen Kindtypen muss dafür die Besucherklasse von den entsprechenden Besucherschnittstellen abgeleitet werden. Neben den typspezifischen *Interfaces* muss der Visitor zusätzlich eine Vererbung zu `CNVisitor` aufweisen, um als Parameter in die `accept`-Methode einzugehen. Die in Abbildung 5.7 dargestellte Klasse `CocoaMenubarComposingVisitor` übernimmt die Verantwortung über die Zusammensetzung der Menüleiste. Da nur Instanzen der Klasse `CocoaMenu` als Kindknoten auftreten, leitet sich diese Besucherklasse lediglich von der Schnittstelle `CocoaMenuVisitor` ab. Ein Aufruf der `visit`-Methode während des *double-dispatch*-Mechanismus bedeutet die Identifizierung eines Untermenüs, das in die Menüleiste eingefügt werden soll. Dementsprechend verknüpft der `CocoaMenubarComposingVisitor` analog zu Listing 5.4 in seiner `visit`-Methode das übergebene `CocoaMenu` mit der `CocoaMenubar`, die ihm während der Instanzierung injiziert und in dem Attribut `menubar` gespeichert wird.

```

1 class CocoaMenubarComposingVisitor
2   : public CNVisitor,
3     public CocoaMenuVisitor {
4   // ...
5   public:
6     virtual void visit(CocoaMenuPtr menu) override {
7       menubar->add(menu);
8     }
9   private:
  CocoaMenubarPtr menubar;

```

Listing 5.4: Implementierung der Klasse `CocoaMenubarComposingVisitor`.

Zu jedem Besucher, der für das Zusammensetzen einer View zuständig ist, entsteht eine weitere Besucherklasse, die die zuvor erstellten Verknüpfungen auflöst. So existiert neben dem `CocoaMenubarComposingVisitor` auch der `CocoaMenubarDecomposingVisitor`, der in seiner `visit`-Methode die `remove`-Methode der `CocoaMenubar` aufruft und die Verknüpfung zu dem besuchten `CocoaMenu` aufhebt.

Damit der Verknüpfungsmechanismus in der `add`-Methode der `CNComponent`-Derivate den *double-dispatch*-Mechanismus des Besuchermusters auslösen kann, muss auch die Schnittstelle `CNComponent` eine `accept`-Methode aufweisen. Zu diesem Zweck leitet sie sich ebenfalls von `CNVisitable` ab. Da es sich bei den `CNComponent`-Derivaten nicht um die zu identifizierenden Klassen handelt, geben sie den erhaltenen Besucher, wie in Listing 5.5 exemplarisch gezeigt, an die `accept`-Methode der gekapselten View weiter.

```

class CocoaMenuComponent : public CNComponent {
// ...
public:
  virtual void accept(CNVisitorPtr v) override {
    menu->accept(v);
  }
private:
  ICocoaMenuPtr menu;

```

Listing 5.5: Implementierung der `accept`-Methode der Klasse `CocoaMenuComponent`.

Sowohl der Verknüpfungsmechanismus als auch die Implementierung der `accept`-Methode in den `CNComponent`-Derivaten hängen dadurch nur von dem öffentlichen Verhalten des *Interfaces* `CNVisitable` ab. Es reicht daher in beiden Fällen aus, die Views als Objekt vom Typ `CNVisitable` zu referenzieren. Infolgedessen können auch die Schnittstellen `ICocoaMenu` und `ICocoaMenubar` aus der Vererbungsbeziehung entfernt werden, indem sich die View-Klassen direkt von `CNVisitable` ableiten.

Für die Verknüpfung der Views erhalten die beiden `CNComponent`-Derivate ein Composing-Decomposing-Visitor-Paar. Vor dem Speichern des hinzugefügten Kindknotens während der `add`-Methode lösen sie den *double-dispatch*-Mechanismus aus, indem sie die `accept`-Methode des übergebenen Knotens mit dem Composing-Visitor aufrufen. Listing 5.6 stellt beispielhaft die Implementierung der `add`-Methode der `CocoaMenuComponent`-Klasse dar.

```
1 class CocoaMenuComponent : public CNComponent {  
2 // ...  
3 public:  
4     virtual void add(CNComponentPtr c) override {  
5         c->accept(composing);  
6         children.push_back(c);  
7     }  
8  
9 private:  
10    std::vector<CNComponentPtr> children;  
11    CNVisitablePtr visitable;  
12    CNVisitorPtr composing;  
13    CNVisitorPtr decomposing;
```

Listing 5.6: Implementierung der `add`-Methode der Klasse `CocoaMenuComponent`.

Nach gleichen Prinzip übergeben die `CNComponent`-Derivate in der `remove`-Methode den Decomposing-Visitor an den zu entfernenden Kindknoten um die Verknüpfung der Views aufzuheben. Auch in den Index-basierten Methoden `insert` und `remove` der `CNComponents` wird der Verknüpfungsmechanismus vor dem Speichern bzw. Entfernen des Kindknotens durch den Aufruf der `accept`-Methode ausgelöst.

Aufgrund dieser Transformation ergibt sich eine identische Implementierung beider `CNComponent`-Derivate, die deshalb durch eine einheitliche Klasse ersetzt werden können. Infolgedessen wird auch die zuvor erwähnte Redundanz in der Logik für das Verwalten von Kindknoten der unterschiedlichen View-Klassen gelöst. Da diese Klasse einen Hierarchieknoten repräsentiert, der dynamisch in Abhängigkeit zu der gekapselten View und den verwendeten Besuchern zusammengesetzt wird, erhält diese Klasse den Namen `CNComposable`.

5.4 Vermeidung von Redundanzen durch den Acceptor

Die durch das Besuchermuster vorgegebenen Vererbungsbeziehungen der View-Klassen zu `CNVisitable` rufen weitere Redundanzen hervor, da die Implementierung jeder `accept`-Methode dem Schema analog zu Listing 5.3 folgt. Zur Eliminierung dieser Redundanzen findet im Rahmen eines *Refactorings* erneut die Komposition Anwendung. Es entsteht die Klasse `CNAcceptor`, die die Logik der `accept`-Methode kapselt. Anstatt die Typumwandlung und den etwaigen Aufruf der `visit`-Methode selbst zu implementieren, können dann die `CNVisitable`-Derivate eine Instanz der Klasse `CNAcceptor` verwenden und das Akzeptieren des Besuchers delegieren. Dafür übergeben sie sich selbst und den erhaltenen Besucher in ihrer eigenen `accept`-Methode entsprechend Listing 5.7 an die `accept`-Methode des `CNAcceptors`.

```
1 virtual void accept(CNVisitorPtr v) override {
    acceptor->accept(v, this->shared_from_this())
3 }
```

Listing 5.7: Umsetzung der `accept`-Methode unter Verwendung eines `CNAcceptors`.

Die Schwierigkeit bei der Implementierung des `CNAcceptors` liegt in den unterschiedlichen Zieltypen der Typumwandlung. Während zum Beispiel die Klasse `CocoaMenu` den übergebenen Besucher in ein Objekt vom Typ `CocoaMenuVisitor` konvertiert, benötigen Instanzen der Klasse `CocoaMenubar` einen Besucher des Typs `CocoaMenubarVisitor`. Um diesen Anforderungen gerecht zu werden, enthält die Klasse `CNAcceptor`, wie in

Listing 5.8 zeigt, das *Template VisitorType*, durch das ihr Benutzer den Zieltyp der Besucherschnittstelle festlegen kann. Eine korrekte Konvertierung des übergebenen `CNVisitors` ergibt sich im Kontext des `CocoaMenus` demnach durch das Ersetzen von `VisitorType` durch den Typ `CocoaMenuVisitor`. Im Gegensatz dazu konfiguriert die `CocoaMenubar` den `CNAcceptor` mit dem Typ `CocoaMenubarVisitor`. Damit nach erfolgreicher Umwandlung die typspezifischen `visit`-Methoden der unterschiedlichen Besucherschnittstellen aufgerufen werden können, muss die `accept`-Methode des `CNAcceptors` `CNVisitable`-Derivate der entsprechenden Subtypen entgegennehmen. Hierzu dient das *Template VisitableType*, das den Subtyp des übergebenen Derivats in der Signatur der `accept`-Methode vorgibt.

```

1 template<typename VisitorType , typename VisitableType>
2 class CNAcceptor {
3 // ...
4 public:
5     virtual void accept(CNVisitorPtr visitor ,
6                         std :: shared_ptr<VisitableType> visitable) {
7         std :: shared_ptr<CNVisitorType> castedVisitor = cast(visitor);
8         if (!castedVisitor)
9             throw CNVisitableVisitorMismatchException();
10        castedVisitor->visit( visitable );
11    }
12 private:
13     virtual std :: shared_ptr<CNVisitorType> cast(CNVisitorPtr v) {
14         return std :: dynamic_pointer_cast<CNVisitorType>(v);
15     }
16 };

```

Listing 5.8: Implementierung der *Template*-basierten Klasse `CNAcceptor`.

Das *Template VisitableType* enthält dafür den Typ, den die in `VisitorType` angegebene Besucherschnittstelle benötigt. Üblicherweise konfigurieren die `CNVisitable`-Derivate, wie in Listing 5.9 am Beispiel des `CocoaMenus` gezeigt, ihre `CNAcceptors` mit dem Typ ihrer eigenen Klassendefinition.

```

1 class CocoaMenu : public CNVisitable,
2                               public std::enable_shared_from_this<CocoaMenu> {
3 // ...
4 private:
5     CocoaMenu() {
6         acceptor = CNAcceptor<CocoaMenuVisitor, CocoaMenu>::getNewInstance();
7     }
8
9 private:
10    CNAcceptorPtr<CocoaMenuVisitor, CocoaMenu> acceptor;
11};

```

Listing 5.9: Instanziierung des `CNAcceptors` in dem Konstruktor der Klasse `CocoaMenu`.

Der Nachteil der *Template*-basierten Lösung ist, dass die eingesetzten Typen als vollwertige Typen bekannt sein müssen. Daher sind *Includes* zu den *Header*-Dateien der Besucherschnittstellen vor dem Ersetzen von `VisitorType` notwendig. Geht man von der üblichen Aufteilung des Quelltextes aus, befindet sich die Deklaration des Attributs `acceptor` in der *Header*-Datei der Klasse. Infolgedessen müssen die *Header*-Dateien der `CNVisitable`-Derivate *Includes* zu ihren Besucherschnittstellen aufweisen. Dadurch hängen dritte Klassen, die den Quelltext einer View verwenden, zusätzlich von den typspezifischen `Visitor-Interfaces` ab. Auch wenn sie Modifikationen der Besucherschnittstellen nicht direkt betreffen, ist ein erneutes Kompilieren erforderlich. Um die *Clients* der View-Klassen von den zugehörigen Besucherschnittstellen zu entkoppeln, wird die Implementierung des `CNAcceptors` hinter einem *Template*-freien *Interface* verborgen. Anstatt das *Template VisitableType* in der Signatur der `accept`-Methode zu verwenden, werden Objekte des festen Typs `CNVisitable` übergeben. Das Attribut `acceptor` in den *Header*-Dateien der Views kann dadurch mit dem Typ des *Template*-freien *Interfaces* deklariert werden, ohne dass *Includes* zu den Besucherschnittstellen notwendig sind. Damit dieses *Interface* den Namen `CNAcceptor` erhalten kann, verschiebt sich die Implementierung der `accept`-Methode in die Klasse `CNAcceptorImp`, die sich von `CNAcceptor` ableitet. Es entsteht das System nach Abbildung 5.8.

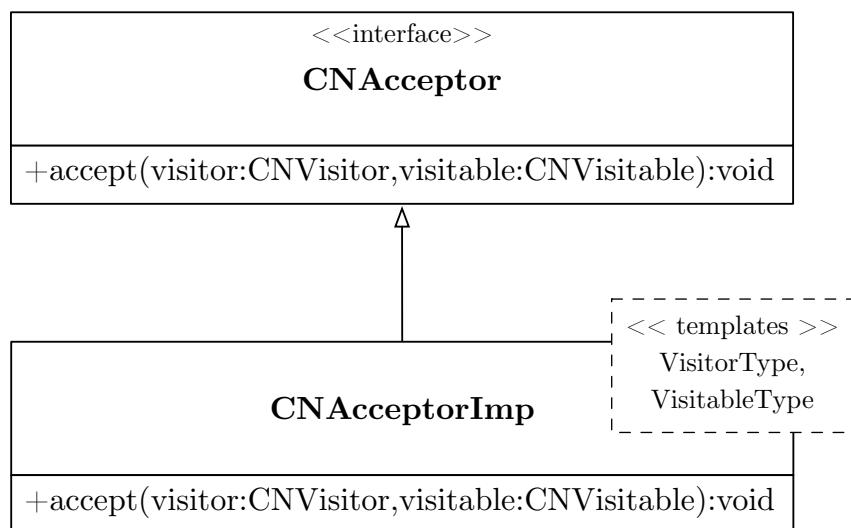


Abbildung 5.8: Verbergen der *Templates* mit Hilfe des *Interfaces* `CNAcceptor`.

Um die typspezifischen `visit`-Methoden trotz der Übergabe eines Objekts des Typs `CNVisitable` aufrufen zu können, verwendet die Klasse `CNAcceptorImp` das *Template* `VisitableType` für eine dynamische Typumwandlung des übergebenen `CNVisitable`-Derivats nach Listing 5.10. Schlägt diese Konvertierung fehl, löst die `accept`-Methode des `CNAcceptors` wie bei einer fehlerhaften Umwandlung des übergebenen `CNVisitors` eine `CNVisitableVisitorMismatchException` aus. Die fehlerfreie Delegation des Akzeptierens hängt somit maßgeblich von dem angegebenen *Template*-Paar ab.

```

1 template<typename CNVisitorType, typename CNVisitableType>
2   class CNAcceptorImp : public CNAcceptor {
3     // ...
4     public:
5       virtual void accept(CNVisitorPtr visitor, CNVisitablePtr visitable) {
6         std::shared_ptr<CNVisitorType> castedVisitor = cast(visitor);
7         std::shared_ptr<CNVisitableType> castedVisitable = cast(visitable);
8
8       if (!castedVisitor || !castedVisitable)
9         throw CNVisitableVisitorMismatchException();
10
11       castedVisitor->visit(castedVisitable);
12     }
13   }
  
```

```

15  private:
16    virtual std::shared_ptr<CNVisitableType> cast(CNVisitablePtr v) {
17      return std::dynamic_pointer_cast<CNVisitableType>(v);
18    }
19    virtual std::shared_ptr<CNVisitorType> cast(CNVisitorPtr v) {
20      return std::dynamic_pointer_cast<CNVisitorType>(v);
21    }
22 };

```

Listing 5.10: Implementierung der *Template*-basierten Klasse `CNAcceptorImp`.

5.5 Verknüpfen der Views durch den besuchenden Composer

Die Verantwortung des einheitlichen `CNComponent`-Derivats `CNComposable` beinhaltet die korrekte Zusammensetzung der Hierarchieknoten. Darunter fallen das Verwalten der Kinder im Kontext der Hierarchie sowie das Auslösen des Verknüpfungsmechanismus. Ruft das `CNComposable` dafür selbst die `accept`-Methode der gekapselten View mit dem Composing- bzw. Decomposing-Visitor auf, wird erneut gegen das *Single Responsibility Principle* verstößen. Sowohl die Modifikation der Verwaltung von Kindknoten als auch eine Veränderung in dem Verknüpfungsmechanismus rufen Anpassungen in `CNComposable` hervor. Dieser Verstoß spiegelt sich ebenfalls in den *Unit Tests* dieser Klasse wider. Das Injizieren des Composing-Decomposing-Visitor-Paars zusammen mit dem `CNVisitable`-Derivat nach Abschnitt 5.3 stört aufgrund der hohen Anzahl an Übergabeparametern während der Konstruktion des SUT neben der Wartbarkeit auch die Lesbarkeit der Tests. Aufgrund dieses Feedbacks wird der Entwickler auf eine möglicherweise unklare und unverständliche Struktur des SUT hingewiesen. Dabei fällt auf, dass beide `CNVisitor`-Derivate dem Verknüpfungsmechanismus angehören. Für eine verbesserte Teststruktur und eine höhere Semantik in dem System des SUT wird das Auslösen des *double-dispatch*-Mechanismus inklusive des Composing-Decomposing-Visitor-Paars in eine weitere Klasse ausgelagert. Durch Komposition erhält `CNComposable` Zugriff auf die öffentlichen

Methoden dieser Klasse und kann somit den Verknüpfungsmechanismus delegieren. Die Verwendung von *Test Doubles* nach Abschnitt 2.4 erzwingt eine Schnittstelle zwischen **CNComposable** und der verknüpfenden Klasse. Da die Derivate dieser Schnittstelle ausschließlich die Verknüpfung der Views bzw. der grafischen Repräsentationen übernehmen, ergibt sich der Name **CNComposer**. Es entsteht das Klassensystem nach Abbildung 5.9.

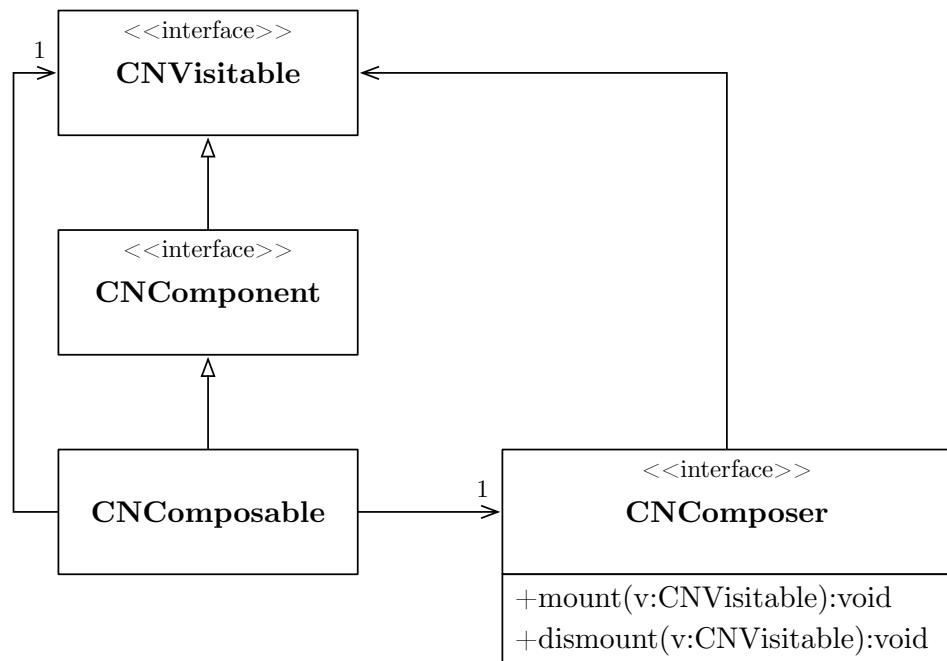


Abbildung 5.9: Verwendung des *Interfaces* **CNComposer** durch die Klasse **CNComposable**.

Durch diese Modifikation des Systems werden neben dem *Single Responsibility Principle* auch das *Dependency Inversion Principle* sowie das *Open/Closed Principle* eingehalten, was zu einer Entkopplung des **CNComposables** von der Implementierung des Verknüpfungsmechanismus führt. Objekte der Klasse **CNComposable** referenzieren jeweils einen **CNComposer**, der die Methoden `mount` und `dismount` zur Verfügung stellt. Dabei steht `mount` für das Erstellen und `dismount` für das Lösen von Verknüpfungen. Anstatt den *double-dispatch*-Mechanismus selbst in Gang zu setzen, ruft das **CNComposable**, wie in Listing 5.11 exemplarisch an der `add`-Methode verdeutlicht, vor dem Speichern bzw. Entfernen von Kindknoten die Methoden des **CNComposers** auf.

```

1 class CNComposable : public CNComponent {
2 // ...
3 public:
4     virtual void add(CNComponentPtr c) override {
5         composer->mount(c);
6         children.push_back(c);
7     }
8 private:
9     std::vector<CNComponentPtr> children;
10    CNVisitablePtr visitable;
11    CNComposerPtr composer;

```

Listing 5.11: Verwendung des **CNComposers** in der Klasse **CNComposable**.

Bei der zuvor extrahierten Klasse zur Verknüpfung von Views handelt es sich um das in Listing 5.12 dargestellte Composer-Derivat **CNVisitingComposer**. Dieser Klasse wird während der Instanziierung das Composing-Decomposing-Visitor-Paar injiziert, das es in den Attributen **composing** und **decomposing** referenziert. Der Aufruf der Methoden **mount** und **dismount** löst den *double-dispatch*-Mechanismus aus. Dafür ruft der **CNVisitingComposer** die **accept**-Methode des erhaltenen **CNVisitable**s mit einem der beiden **CNVisitor**-Derivaten auf.

```

1 class CNVisitingComposer : public CNComposer {
2 // ...
3 public:
4     virtual void mount(CNVisitablePtr v) override {
5         v->accept(composing);
6     }
7     virtual void dismount(CNVisitablePtr v) override {
8         v->accept(decomposing);
9     }
10 private:
11    CNVisitorPtr composing;
12    CNVisitorPtr decomposing;
13 }

```

Listing 5.12: Implementierung der Klasse **CNVisitingComposer**.

Das Erstellen eines Knotens, der im Kontext der Hierarchie eine entsprechend des Anwendungsbeispiels zusammensetzbare Menüleiste widerspiegelt, geschieht auf die in Listing 5.13 dargestellte Weise. Zunächst erfolgt die Instanziierung der Klasse `CocoaMenubar`, die in einem nächsten Schritt bei der Erzeugung des Composing-Decomposing-Visitor-Paars injiziert wird. Zur anschließenden Erzeugung des `CNVisitingComposers` erhält die statische `getNewInstance`-Methode das Composing-Decomposing-Visitor-Paar als Übergabeparameter. In einem letzten Schritt wird mit der Menüleiste und dem `CNVisitingComposer` eine Instanz der Klasse `CNComposable` erstellt.

```
1 CocoaMenubarPtr menubar = CocoaMenubar::getInstance();  
2  
3 CocoaMenubarComposingVisitorPtr composing;  
4 composing = CocoaMenubarComposingVisitor::getInstance(menubar);  
5  
6 CocoaMenubarDecomposingVisitorPtr decomposing;  
7 decomposing = CocoaMenubarDecomposingVisitor::getInstance(menubar);  
8  
9 CNVisitingComposerPtr composer;  
10 composer = CNVisitingComposer::getInstance(composing, decomposing);  
11  
12 CNComposablePtr menubarComponent;  
13 menubarComponent = CNComposable::getInstance(menubar, composer);
```

Listing 5.13: Erstellen eines Knotens zur Repräsentation einer zusammensetzbaren Menüleiste im Kontext der Hierarchie.

5.6 Identifizieren der Elternknoten durch den besuchenden Matcher

Aus den in Abschnitt 5.3 erläuterten `CNComposables` lassen sich mit Hilfe von Cross-Native Objekthierarchien erstellen, deren Komposition sich dynamisch zur Laufzeit er-

gibt. Zu diesem Zweck wird der Kompositionsalgorithmus verwendet, den die Klasse **CNDynamicHierarchy** in ihrer `load`-Methode kapselt. Um die **CNComponent**-Derivate als Knoten der Objekthierarchie zu registrieren, übergibt sie der Benutzer des Frameworks, wie in Kapitel 4 erläutert, zusammen mit einem Matcher an die `load`-Methode der **CNDynamicHierarchy**. Der Matcher dient dem Kompositionsalgorithmus dabei zur Identifizierung des Elternknotens. **CNDynamicHierarchy** iteriert während der `load`-Methode durch die bereits geladenen Knoten der Objekthierarchie und übergibt sie an die `matches`-Methode des Matchers. Anhand des boolschen Rückgabewerts dieser Methode erhält **CNDynamicHierarchy** die Information, ob es sich bei dem Knoten des aktuellen Iterationsschritts um den Elternknoten handelt. Die Identifizierung eines Knotens durch den Matcher veranlasst die **CNDynamicHierarchy** eine Eltern-Kind-Beziehung zu erstellen. Dafür ruft sie die `add`-Methode des Elternknotens auf und über gibt das zu ladene Component-Derivat als Parameter. Wie auch der Verknüpfungsmechanismus aus Abschnitt 5.5 basiert der Identifizierungsmechanismus der Matcher auf dem *double-dispatch*-Mechanismus des Besuchermusters. Daher besitzt die `matches`-Methode einen Übergabeparameter des Typs **CNVisitable**. Es entsteht das System analog zu Abbildung 5.10.

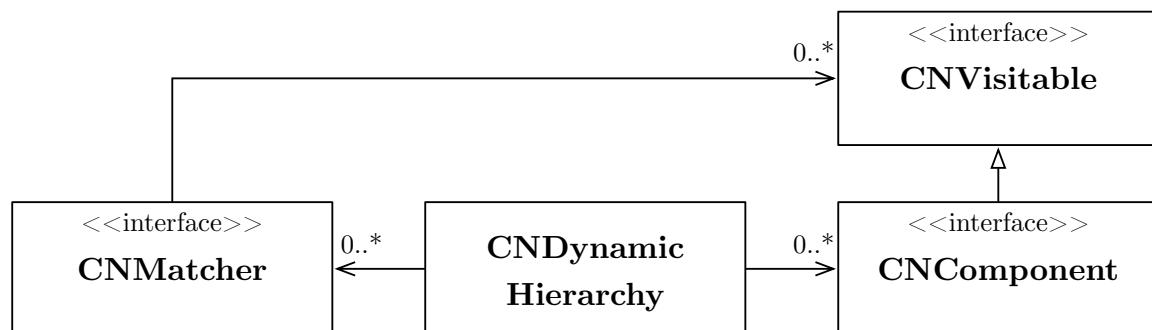


Abbildung 5.10: Klassenbeziehungen für das Laden eines **CNComponents** in die **CNDynamicHierarchy**.

Zur Komposition einer Objekthierarchie entsprechend des Anwendungsbeispiels nach Abschnitt 5.1 werden zunächst **CNComposables** erzeugt, die die geforderten Views enthalten. Durch den Aufruf der `load`-Methode werden sie in einem nächsten Schritt nacheinander in die Hierarchie geladen. Dabei müssen die übergebenen **CNMatcher** die gewünschten Elternknoten der unterschiedlichen **CNComponents** eindeutig identifizieren.

Da es sich bei der Menüleiste aufgrund der Umsetzung der GUI mit dem Framework Cocoa um eine *ToplevelView* (siehe Abschnitt 3.2) handelt, darf der zugehörige **CNMatcher** keine Knoten der Hierarchie erkennen. Die `matches`-Methode muss in diesem Fall immer den boolschen Wert `false` zurückgeben. Das Laden von *ToplevelViews* erfolgt mit Instanzen der Klasse **CNNullMatcher** nach Listing 5.14.

```
1 class CNNullMatcher : public CNMatcher {
2 //...
3 public:
4     virtual void matches(CNVisitablePtr v) override {
5         return false;
6     }
7 };
```

Listing 5.14: Implementierung der Klasse **CNNullMatcher**.

Damit sich eine applikationsspezifische Objekthierarchie nach Abbildung 5.2 einstellt, erfordert das korrekte Laden des Menüs bzw. der Menüeinträge **CNMatcher**, die die Menüleiste bzw. das Menü als Elternknoten identifizieren. Da die Objekthierarchie sowohl von **CocoaMenubar** als auch von **CocoaMenu** nur eine Instanz enthält, ist eine eindeutige Identifizierung der Objekte durch die Überprüfung des konkreten Subtyps möglich. Für diese Typüberprüfung wird das bereits implementierte azyklische Besuchermuster benutzt, da das Auftreten der **CNVisitableVisitorMismatchException** Rückschlüsse auf den konkreten Subtyp des **CNVisitable**s zulässt. Übergibt man z.B. ein **CNVisitor**-Derivat, das sich zugleich auch von **CocoaMenuVisitor** ableitet, rufen nur Instanzen der Klasse **CocoaMenu** die `visit`-Methode des Besuchers auf. In allen anderen Fällen löst die `accept`-Methode des **CNVisitable**-Derivats eine *Exception* aus.

Zur Identifizierung von Subtypen des *Interfaces* **CNVisitable** entsteht unter Anwendung der testgetriebenen Entwicklung zunächst die in Listing 5.15 gezeigte Klasse **CNTypematcher**. Ihr wird während der Instanziierung ein **CNVisitor**-Derivat injiziert, das sie in dem Attribut `identifier` speichert. In der `matches`-Methode löst der **CNTypematcher** den *double-dispatch*-Mechanismus aus, indem er die `accept`-Methode des übergebenen **CNVisitable**s mit dem `identifier` aufruft. Resultiert aus dem

Aufruf der `accept`-Methode eine *Exception*, passen Subtyp des `CNVisitable`s und Besucher nicht zusammen. Der `CNTypematcher` fängt den Ausnahmefehler ab und gibt den boolschen Wert `false` zurück. Tritt kein Ausnahmefehler auf entspricht das übergebene `CNVisitable` dem von dem `identifier` geforderten Typ. In diesem Fall liefert die `matches`-Methode den Wert `true`.

```

1 class CNTypematcher : public CNMatcher {
2 // ...
3 public:
4     virtual void matches(CNVisitablePtr v) override {
5         try {
6             v->accept(identifier);
7         } catch(CNVisitableVisitorMismatchException &e) {
8             return false;
9         }
10        return true;
11    }
12 private:
13     CNVisitorPtr identifier;
14 };

```

Listing 5.15: Implementierung der Klasse `CNTypematcher`.

Da die verwendeten Besucher-Derivate des `CNTypematchers` nur zur Identifizierung eines Subtyps dienen und keinen eigenen Zweck verfolgen, besitzen sie leere Implementierungen der `visit`-Methoden. Listing 5.16 zeigt exemplarisch die Implementierung des `CocoaMenuIdentifier`s zur Identifizierung von Instanzen der Klasse `CocoaMenu`.

```

1 class CocoaMenuIdentifier : public CNVisitor,
2                               public CocoaMenuVisitor {
3 // ...
4 public:
5     virtual void (CocoaMenuPtr v) override {}
6 };

```

Listing 5.16: Implementierung der Klasse `CocoaMenuIdentifier`.

Schwieriger gestaltet sich die Identifizierung von Knoten, wenn die Hierarchie mehrere Objekte desselben Typs enthält. Befinden sich beispielsweise mehrere Menüs in der Hierarchie, reicht es nicht mehr aus, nach dem Typ `CocoaMenu` zu suchen. Für eine eindeutige Unterscheidung müssen daher weitere Parameter in Betracht gezogen werden. Zu diesem Zweck erfährt die Klasse `CocoaMenu` die Erweiterung um ein *Tag*, das der Entwickler während der Instanziierung festlegen kann.

Ein mögliches Vorgehen zur Umsetzung der *Tag*-basierten Identifizierung wäre eine Modifikation der Klasse `CNTypematcher`, so dass sie sowohl Subtypen als auch *Tags* erkennen kann. Da die Modifikation die Funktionalität des zuvor implementierten Identifizierungsmechanismus sowie die zugehörigen Tests gefährdet, ist dieses Vorgehen in der testgetriebenen Entwicklung nicht zulässig. Stattdessen wird ein weiteres `CNMatcher`-Derivat erstellt, das den neuen Identifizierungsmechanismus implementiert. Erst wenn sich Redundanzen in beiden Derivaten zeigen, werden die Ansätze des objektorientierten Designs verwendet, um die Implementierung zu vereinheitlichen. Dieses Vorgehen stellt sicher, dass bereits umgesetzte Logik durch die Erweiterung des Systems nicht ihre Gültigkeit verliert. Gleichzeitig gewährleistet es, dass die Designstrukturen immer an die momentane Funktionalität der Software angepasst sind und verhindert unnötige Komplexität des Systems, in dem Funktionalitäten unbeachtet bleiben, die möglicherweise zukünftig folgen könnten.

Um ein Menü anhand seines *Tags* zu identifizieren, entsteht die Klasse `CocoaMenuTagMatcher`. Auch dieses Matcher-Derivat verwendet intern den *double-dispatch*-Mechanismus, indem es einen Besucher entsendet. Die Aufgabe des Besuchers beinhaltet in diesem Fall das *Tag* des Menüs in seiner `visit`-Methode mit einem erwarteten *Tag* zu vergleichen. Damit der `CocoaMenuTagMatcher` basierend darauf einen boolschen Wert zurückgeben kann, muss der Besucher das Ergebnis des Vergleichs bereitstellen. Anstatt des degenerierten Besucher-*Interfaces* verwendet der `CocoaMenuTagMatcher` die Klasse `CocoaMenuTagIdentifier`, die durch ihre `hasIdentified`-Methode das Ergebnis des Vergleichs liefert. Es entsteht das System nach Abbildung 5.11.

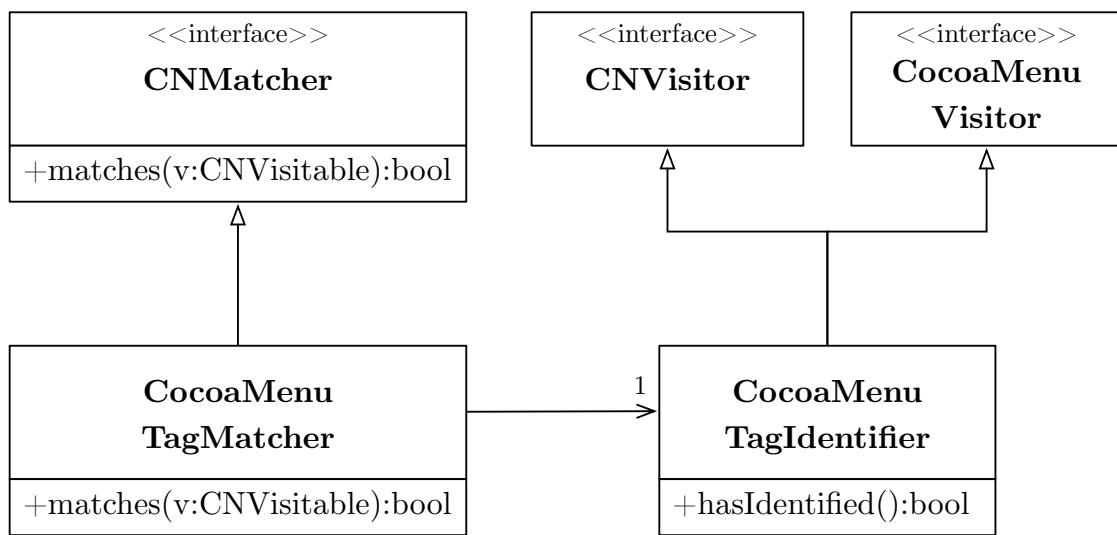


Abbildung 5.11: Klassensystem zur *Tag*-basierten Identifikation der Klasse `CocoaMenu`.

Die Implementierung des `CocoaMenuTagMatcher`s erfolgt nach Listing 5.17. In dem Konstruktur initialisiert das `CNMatcher`-Derivat das Attribut `identifier` mit einer neuen Instanz der Klasse `CocoaMenuTagIdentifier`, der das erwartete *Tag* injiziert wird. Das Aufrufen der `matches`-Methode übergibt den `identifier` an die `accept`-Methode des `CNVisitable`s und löst dadurch den *double-dispatch*-Mechanismus aus. Im Gegensatz zu `CNTypeMatcher` bedeutet das Ausbleiben der *Exception* hier nicht die Identifizierung des `CNVisitable`s, da neben dem Typ auch das *Tag* übereinstimmen muss. Dementsprechend gibt der `CocoaMenuTagMatcher` mit dem Aufruf der `hasIdentified`-Methode das Ergebnis des Vergleichs zurück.

```

1 class CocoaMenuTagMatcher : public CNMatcher {
2 // ...
3 private :
4     CocoaMenuTagMatcher( std :: string expectedTag ) {
5         identifier = CocoaMenuTagIdentifier :: getInstance( expectedTag );
6     }
7 public :
8     virtual bool matches( CNVisitablePtr v ) override {
9         try {
10             v->accept( identifier );
  
```

```

12     } catch( CNVisitableVisitorMismatchException &e ) {
13         return false;
14     }
15     return identifier->hasIdentified();
16 }
17 private:
18     CocoaMenuTagIdentifier identifier;
19 };

```

Listing 5.17: Implementierung der Klasse `CocoaMenuTagMatcher`.

Nach der Umsetzung beider `CNMatcher`-Derivate fällt eine hohe Ähnlichkeit in den Implementierungen auf. Überwindet man die Differenzen beider Klassen durch Modifikationen des Systems, können sie nun zu einem einheitlichen Identifizierungsmechanismus zusammengefasst werden. Der größte Unterschied in der Implementierung der `matches`-Methoden liegt in dem Rückgabewert bei Ausbleiben der `CNVisitableVisitorMismatchException`. Um den Quelltext beider Methoden einander anzunähern, wird auch der `CocoaMenuTypeIdentifier` um eine `hasIdentified`-Methode erweitert. Da der Aufruf dieser Methode in dem `CNTypematcher` den konstanten Wert `true` ersetzt, gibt auch sie den Wert `true` zurück. Die nun identische API der Klassen `CocoaMenuTagIdentifier` und `CocoaMenuItemIdentifier` können durch das *Interface* `CNIdentifier` repräsentiert werden. Die Verwendung dieser Schnittstelle in beiden `CNMatcher`-Derivaten führt zu einer identischen Implementierung der `matches`-Methode. Beide Klassen können deshalb durch den in Abbildung 5.12 gezeigten `CNVisitingMatcher` ersetzt werden. Es entsteht ein allgemeiner Identifizierungsmechanismus, der durch die Art des injizierten `CNIdentifier`-Derivats beeinflusst werden kann. Die Implementierung des `CNVisitingMatchers` gestaltet sich entsprechend Listing 5.18.

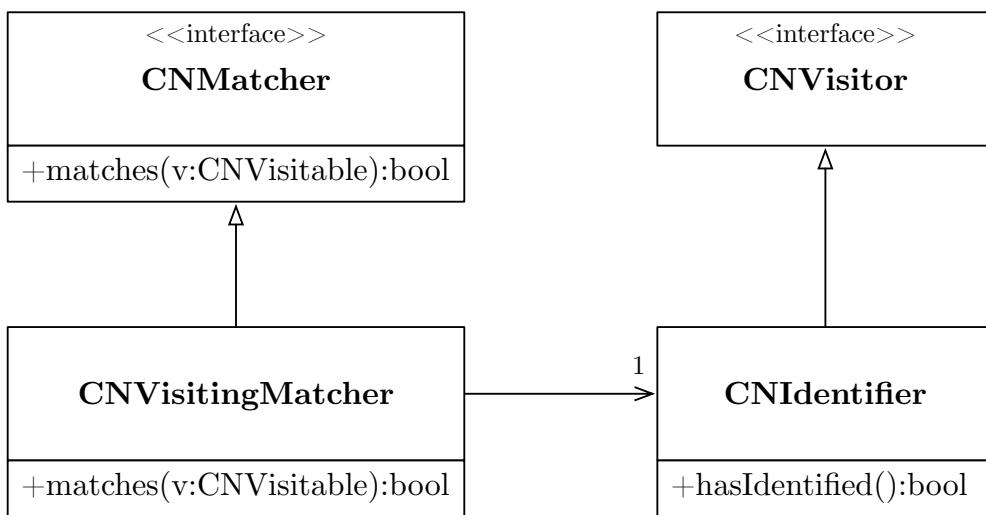


Abbildung 5.12: Klassenbeziehungen des CNVisitingMatchers.

```

1 class CNVisitingMatcher : public CNMatcher {
2 // ...
3 public:
4     virtual void matches(CNVisitablePtr v) override {
5         try {
6             v->accept(identifier);
7         } catch(CNVisitableVisitorMismatchException &e) {
8             return false;
9         }
10
11         return identifier->hasIdentified();
12     }
13
14 private:
15     CNIdentifierPtr identifier;
16 };
  
```

Listing 5.18: Implementierung der Klasse CNVisitingMatcher.

5.7 Verallgemeinerung der Tags

Der zuvor erläuterte `CocoaMenuTagIdentifier` dient allein zur Identifizierung von Menüs anhand eines eindeutigen *Tags*. Neben den Menüs einer Applikation sind jedoch durchaus weitere View-Klassen denkbar, von denen sich mehr als nur eine Instanz in der Komposition der GUI befindet. Um Objekte dieser Klassen in der Hierarchie eindeutig zu erkennen, werden ebenfalls *Tags* verwendet. Es entsteht das `CNIIdentifier`-Derivat `CocoaTagIdentifier`, das jede dieser Klassen anhand ihres *Tags* identifizieren kann. Dafür leitet sie sich, wie in Abbildung 5.13 dargestellt, von den typspezifischen Besucherschnittstellen aller *Tag*-basierten View-Klassen ab.

Listing 5.19 zeigt die Implementierung des `CocoaTagIdentifier`s. In den `visit`-Methoden vergleicht das `CNIIdentifier`-Derivat das *Tag* der übergebenen View mit einem erwarteten *Tag* und speichert das Ergebnis in der boolschen Variable `identified`, das *Clients* durch die `hasIdentified`-Methode zur Verfügung gestellt wird.

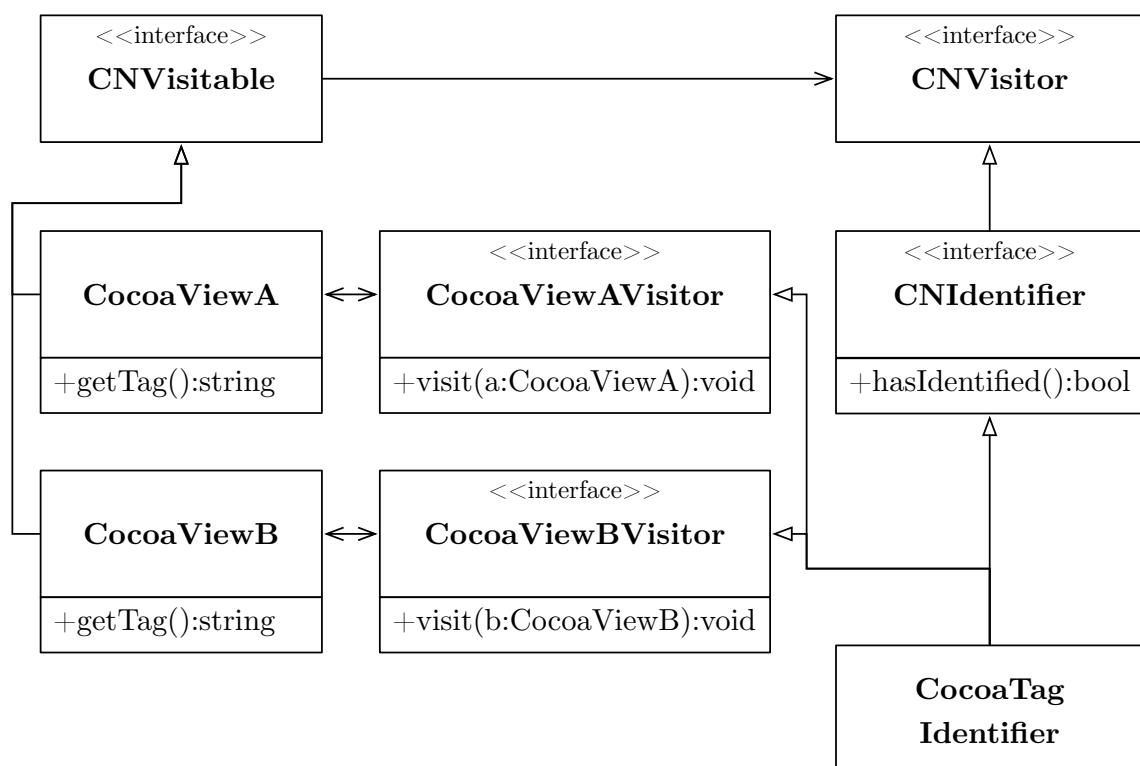


Abbildung 5.13: Klassenbeziehungen zur *Tag*-basierten Identifizierung der Views.

```

1 class CocoaTagIdentifier : public CNIdentifier {
2 // ...
3 public:
4     virtual void visit(CocoaViewAPtr v) override {
5         identified = (expectedTag == v->getTag());
6     }
7     virtual void visit(CocoaViewBPtr v) override {
8         identified = (expectedTag == v->getTag());
9     }
10    virtual bool hasIdentified() {
11        return identified;
12    }
13 private:
14    std::string expectedTag;
15    bool identified;
16 };

```

Listing 5.19: Implementierung der Klasse `CocoaTagIdentifier`.

Durch die Abhängigkeit des `CocoaTagIdentifier`s zu den konkreten View-Klassen verstößt diese Lösung gegen das *Dependency Inversion Principle*. Als Nachteil ergibt sich daraus, dass die Erweiterung des Systems um eine *Tag*-basierte View-Klasse auch die Modifikation des `CocoaTagIdentifier`s erzwingt. Diese Modifikation entfällt, wenn der `CocoaTagIdentifier` in seinen `visit`-Methoden anstatt der konkreten Typen der View-Klassen das einheitliche *Interface CNTagged* entgegennimmt, das die `getTag`-Methode bereitstellt. Die Vererbungsbeziehung einer neuen View-Klasse zu diesem *Interface* ermöglicht dann die *Tag*-basierte Identifizierung, ohne den `CocoaTagIdentifier` anpassen zu müssen. Der `CocoaTagIdentifier` ist dadurch unabhängig von den zu identifizierenden Subklassen und kann nicht nur in dem Kontext einer bestimmten Applikation verwendet werden. Er geht daher in den Bestand des Frameworks CrossNative über und wird dementsprechend in `CNTagIdentifier` umbenannt. Da die Signatur der `visit`-Methoden von den typspezifischen Besucherschnittstellen vorgegeben wird, erfordert die Umsetzung des *Dependency Inversion Principles* eine typspezifische Besucherschnittstelle für das *Interface CNTagged*. Es ergibt sich das System nach Abbildung 5.14.

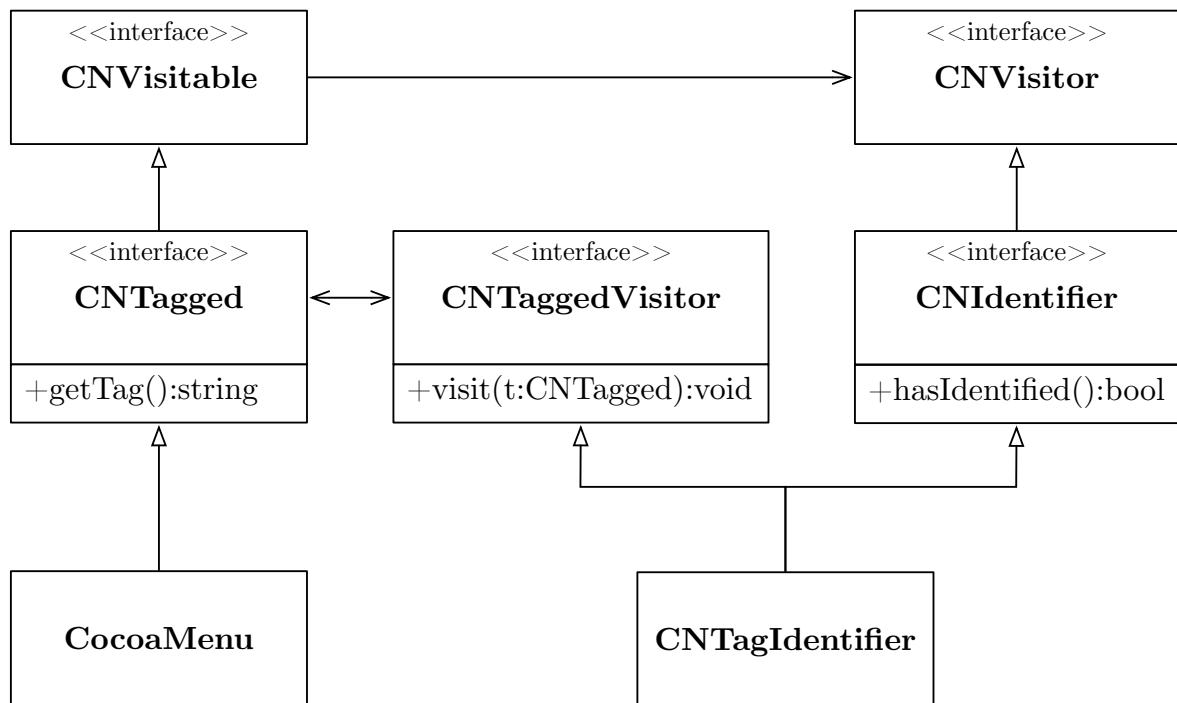


Abbildung 5.14: Verwenden des einheitlichen *Interfaces* `CNTagged` zur Identifizierung von Views.

Damit die Subklassen von `CNTagged` in ihren `accept`-Methoden neben den typspezifischen `visit`-Methoden ihrer eigenen Besucherschnittstellen auch die `visit`-Methode des `CNTagIdentifier`s aufrufen, sind Modifikationen der View-Klassen notwendig. Listing 5.20 zeigt die geforderte Implementierung am Beispiel der Klasse `CocoaMenu`. Neben dem Attribut `acceptor` weisen die *Tag*-basierten View-Klassen ein zusätzliches Attribut vom Typ `CNAcceptor` auf. Dabei handelt es sich um den `tagAcceptor`, der für das Akzeptieren von `CNTaggedVisitor`-Derivaten zuständig ist. Die `accept`-Methode der *Tag*-basierten Views delegiert wie zuvor das Akzeptieren des Besuchers an den `acceptor`. Löst sich dabei eine `CNVisitableVisitorMismatchException` aus, wird das Akzeptieren an den `tagAcceptor` weitergeleitet.

```

1 class CocoaMenu : public CNVisitable, public CNTagged,
2                               public std::enable_shared_from_this<CocoaMenu> {
3     // ...
4     private:
5         CocoaMenu() {
6             acceptor = CNAcceptorImp<CocoaMenuVisitor, CocoaMenu>::getNewInstance();
7             tagAcceptor = CNAcceptorImp<CNTaggedVisitor, CNTagged>::getNewInstance();
8         }
9     public:
10        virtual void accept(CNVisitorPtr v) {
11            try {
12                acceptor->accept(v, this->shared_from_this());
13            } catch (CNVisitableVisitorMismatchException &e) {
14                tagAcceptor->accept(v, this->shared_from_this());
15            }
16        }
17     private:
18        CNAcceptorPtr acceptor;
19        CNAcceptorPtr tagAcceptor;
20    };

```

Listing 5.20: Verwendung mehrerer **CNAcceptors** in der Klasse **CocoaMenu**.

5.8 Zusammenfassung

Dieses Kapitel erläutert die Einflüsse der testgetriebenen Entwicklung auf die Entstehung komplexer Klassensysteme am Beispiel des Frameworks CrossNative, das die dynamische Komposition von Objekthierarchien für den Entwurf von grafischen Benutzeroberflächen ermöglicht. Zur besseren Verständlichkeit wird dafür zu Beginn ein Anwendungsbeispiel gegeben, das die auftretenden Entwurfsprobleme während der Implementierung sowie die daraus resultierenden Designentscheidungen verdeutlicht. Anschließend werden die einzelnen Entwurfsschritte vorgeführt, die das in Abbildung 5.15 dargestellte Framework als Ergebnis hervorbringen.

5. Entstehung des Framework CrossNative

Betrachtet man CrossNative in seiner Gesamtheit ist ersichtlich, dass seine Struktur im Wesentlichen auf abstrakten Schnittstellen basiert, die nur von einigen konkreten Klassen implementiert werden. Grund dafür ist das Bestreben des testgetriebenen Entwicklungsansatzes die öffentlichen Methoden bzw. die unterschiedlichen *Units of Work* in Isolation zu testen. Daraus entstehen Klassen, die nur auf *Interfaces* basieren und eine lose Koppelung aufweisen.

Des Weiteren führt der dreiphasige Zyklus der testgetriebenen Entwicklung durch häufiges *Refactoring* zu Redundanzfreiheit. Dabei wird nicht versucht, die Duplikationen des Quelltextes wie in der konventionellen Softwareentwicklung durch eine ausgiebige Planungsphase zu verhindern. Die Anwendung von TDD lässt diese Redundanzen gezielt entstehen, um die Notwendigkeit von Designstrukturen hervorzuheben. Dieser Vorgang wird zum Beispiel an der in Abschnitt 5.6 veranschaulichten Entstehung des **CNVisitingMatchers** deutlich. Anstatt diese Klasse vorausschauend so zu entwerfen, dass jede erdenkliche Identifizierung möglich ist, entstehen zunächst die Klassen **CNTypematcher** und **CocoaMenuTagMatcher**, die jeweils einen Identifizierungsmechanismus repräsentieren. Eine testgetriebene Erweiterung der Software um neue Funktionalität erfolgt daher auf dem einfachsten Weg im Sinne der Komplexität des Systems. Die dabei auftretende Redundanz dient als Indikator für unzulängliches Design. Während eines *Refactorings* wird die Redundanz durch die geschickte Modifikation des Systems eliminiert. Dadurch gewährleistet die Anwendung von TDD, dass die Komplexität des objektorientierten Designs zu keinem Zeitpunkt die notwendige Komplexität überschreitet.

Darüber hinaus zeigt die Entstehung der Schnittstelle **CNComposer** den Einfluss der Teststruktur auf die Umsetzung des Designs. Die Injektion des Composing-Decomposing-Visitor-Paars in die Klasse **CNComposable** führt aufgrund der komplexen Konstruktion des SUT zu unklaren und unverständlichen *Unit Tests*. Das Ersetzen der **CNVisitor**-Derivate durch einen Übergabeparameter vom Typ **CNComposer** und erhöht die Qualität der Tests. Zugleich steigt damit die Semantik in der öffentlichen API des **CNComposables**, da durch die Übergabe eines **CNComposers** leichter auf die Verwendung des Verknüpfungsmechanismus geschlossen werden kann. Infolgedessen erhöht sich die Verständlichkeit des System.

Unter anderem führen die zuvor genannten Einflüsse der testgetriebenen Entwicklung zu einem Framework, das aufgrund seiner losen Kopplung und den eingehaltenen Prinzipien hohe Qualität des objektorientierten Designs aufweist. Daraus resultiert ein flexibles Softwaresystem, das aufgrund der hohen Anzahl an abstrakten Schnittstellen viele Möglichkeiten der Erweiterung bietet. Zum Beispiel erfordert der Austausch des Verknüpfungsmechanismus lediglich die Umsetzung eines weiteren `CNComposer`-Derivats, das die alternative Implementierung erhält. Das restliche System bleibt davon unbeeinflusst. Auch der Identifizierungsmechanismus des `CNMatchers` ist auf diese Weise leicht variierbar. Neben der Möglichkeit, den gesamten Mechanismus durch ein weiteres `CNMatcher`-Derivat auszutauschen, nimmt auch die verwendete Repräsentation der Schnittstelle `CNIdentifier` Einfluss auf das Identifizieren eines Knotens. Auf die in Abschnitt 5.7 gezeigte Weise kann der Identifizierungsmechanismus neben dem *Tag* um viele weitere Identifizierungsparameter ergänzt werden, ohne Anpassungen in der Klasse `CNVisitingMatcher` zu bedingen.

Die testgetriebene Entwicklung erzwingt aufgrund der angestrebten Isolation des SUT die öffentliche Injektion externer Abhängigkeiten. Zwar erhöht sich dadurch die Verständlichkeit Systems, allerdings erschwert das auch die Instanziierung von Objekten. Ein Beispiel dafür liefert die komplexe Konstruktion der zusammensetzbaren Menüleiste nach Listing 5.13. Um diesem Nachteil entgegenzuwirken kann auf die Erzeugungsmuster, wie z.B. die *Factory Method*, nach Gamma et al. (2004) zurückgegriffen werden.

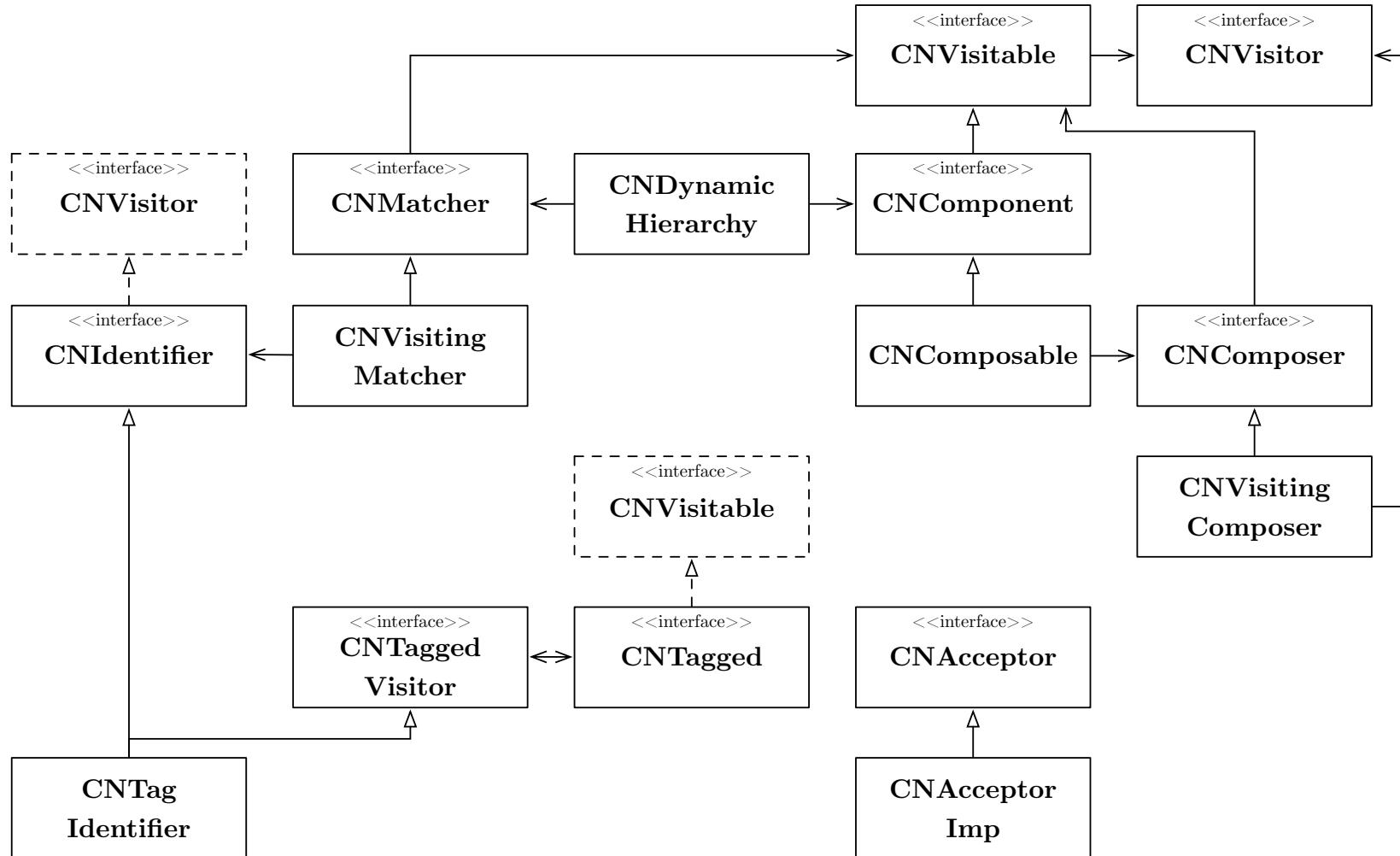


Abbildung 5.15: Das Framework CrossNative im Überblick.

KAPITEL 6

Entwurf des Gittergenerators – Anwendung von CrossNative

In vielen Ingenieursdisziplinen bilden Strömungssimulationen einen festen Bestandteil in Wissenschaft und Forschung. So werden sie beispielsweise im Bauwesen dafür verwendet, die wirkenden Kräfte eines umströmenden Fluids auf einen festen Körper zu ermitteln. Als Berechnungsgrundlage für die Stömungssimulationen dienen diskrete Gitter. In dem Paper “Effiziente Implementierung eines Gittergenerators auf GPGPUs” stellt Peters (2016) die Implementierung eines Gittergenerators vor, um diskrete Berechnungsgitter aus dreidimensionalen Geometrien zu erzeugen.

Zur Demonstration des entstandenen Frameworks umfasst dieses Kapitel den Entwurf einer grafischen Oberfläche zur Visualisierung der Prozesse und Ergebnisse des Gittergenerators nach Peters (2016). Damit auch die Kompatibilität zu unterschiedlichen GUI-Frameworks gezeigt werden kann, basiert die grafische Oberfläche des Gittergenerators im Gegensatz zu Kapitel 5 auf dem Framework Qt. Die Visualisierung der Ergebnisse erfolgt mit Hilfe des Frameworks VTK.

6.1 Konzept der grafischen Oberfläche

Die Basis für die Gittergenerierung nach Peters (2016) bilden Dreiecke, die im Zusammenschluss die Oberfläche dreidimensionaler Geometrien darstellen. Als Quelle für die Daten der Dreiecke dienen im Rahmen dieser Arbeit Dateien im Format der Stereolithography-Schnittstelle (i.F. STL-Dateien). Zur Steuerung der Gittergenerierung mit Hilfe der grafischen Oberfläche muss die Applikation das Einlesen von STL-Dateien ermöglichen. Des Weiteren muss sie dem Anwender erlauben, den Generierungsprozess anzustoßen sowie anschließend das Ergebnis zu visualisieren. Da der Vorgang der Gittererzeugung von den eingelesenen Daten abhängt, ergibt sich aus der internen Sicht eine Modellstruktur nach Abbildung 6.1.

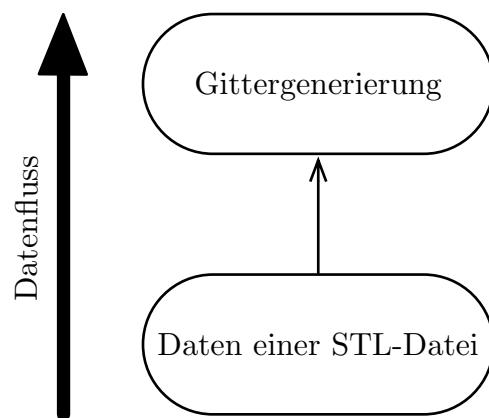


Abbildung 6.1: Interne Modellstruktur zur Generierung eines Gitters.

Zur Modellierung dieser Struktur enthält die GUI der Applikation einen *Solution Explorer*, der eine hierarchische Struktur in Form eines Baums darstellt. Über die Schaltflächen „Add *STLFile*“ und „Add *GridGenerator*“ des „Add“-Menüs in der Menüleiste kann der Benutzer Knoten erstellen und dem *Solution Explorer* hinzufügen. Dabei existieren zur Repräsentation der in Abbildung 6.1 gezeigten Prozesse und Daten die Knotentypen *GridGenerator* und *STLFile*. Während der Knotentyp *GridGenerator* den Prozess der Gittererzeugung darstellt, repräsentieren Knoten des Typs *STLFile* die eingelesenen Daten der STL-Dateien. Der Datenfluss wird über Eltern-Kind-Beziehungen realisiert, indem Elternknoten auf die Daten der Kindknoten zugreifen. Für die Erzeugung eines

Gitters muss daher ein Knoten des Typs *GridGenerator* erstellt werden, dem Knoten des Typs *STLFile* als Kindknoten zugeordnet sind. In die Erzeugung eines Gitters können durchaus auch Daten mehrerer Geometrien einfließen. Daher sind mehrere *STLFiles* als Kindknoten möglich. Abbildung 6.1 zeigt eine exemplarische Modellstruktur zur Erzeugung eines Gitters, in die die Daten der beiden STL-Dateien *STLFile_1* und *STLFile_2* eingehen. Um eine Eltern-Kind-Beziehung zwischen einem *GridGenerator* und einem *STLFile* zu erreichen, muss während des Hinzufügens des *STLFiles* der gewünschte Elternknoten in der Baumstruktur ausgewählt sein. Für das Entfernen eines ausgewählten Knotens mitsamt seiner Kindknoten steht die Schaltfläche „*Delete*“ des „*Edit*“-Menüs zur Verfügung.

Solution Explorer	
Name	Typ
▼ GridGenerator_1	GridGenerator
STLFile_1	STLFile
STLFile_2	STLFile

Abbildung 6.2: Exemplarische Darstellung einer Modellstruktur zur Generierung eines Gitters.

Neben der Baumstruktur weist die grafische Oberfläche des Gittergenerators eine tabellarische Ansicht zur Darstellung der Knoteneigenschaften auf. Dabei handelt es sich um den *Properties Explorer*, dessen Inhalt von dem ausgewählten Knoten des *Solution Explorers* abhängt. Ändert sich der ausgewählte Knoten in der Baumstruktur, so aktualisieren sich ebenfalls die angezeigten Eigenschaften. Abbildung 6.3 stellt beispielhaft die Eigenschaftstabelle des Knotens *STLFile_1* aus Abbildung 6.2 in dem *Properties Explorer* dar.

Nach der Modellierung der hierarchischen Struktur kann der Benutzer die einzelnen Knoten über die Schaltfläche „*Visualize*“ des „*Evaluate*“-Menüs auswerten und visualisieren. Das Betätigen dieses Menüeintrags lässt die grafische Darstellung des ausgewählten Knotens in der Oberfläche erscheinen. Die Art der Visualisierung hängt dabei von dem Knotentyp ab. Während bei einem *GridGenerator* der Prozess der Generierung ausgelöst und anschließend das entstandene Gitter angezeigt wird, enthält die Visualisierung eines *STLFiles* die grafische Darstellung der Geometrie.

Properties Explorer	
Property	Value
Type	STLFile
Name	STLFile_1
Path	Users/STLFiles/box.stl
Number of Triangles	12

Abbildung 6.3: Exemplarische Darstellung einer Eigenschaften der STL-Datei *STLFile_1* aus Abbildung 6.1 in dem *Properties Explorer*.

Eingerahmt werden *Solution Explorer*, *Properties Explorer* sowie die Visualisierungen der einzelnen Knoten von dem Hauptanwendungsfenster der Applikation. Abbildung 6.4 zeigt eine mögliche Aufteilung der Ansichten. Die Darstellung der unterschiedlichen Visualisierungen können dabei im rechten Bereich des Hauptanwendungsfensters zum Beispiel in Form von Reitern organisiert werden.

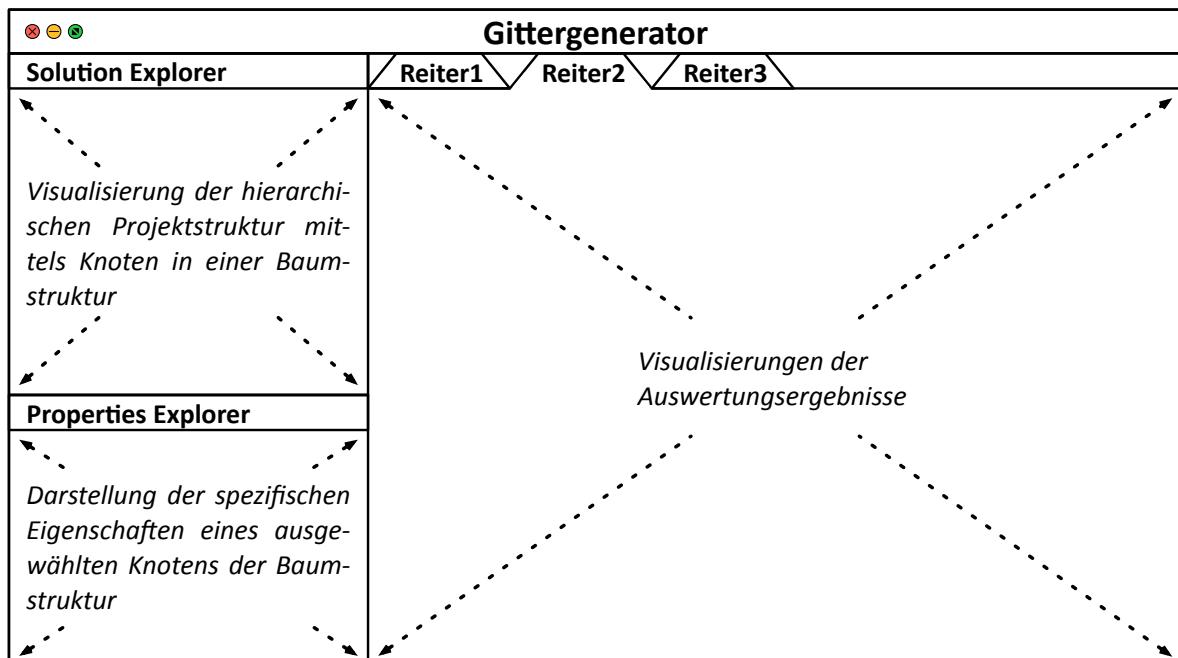


Abbildung 6.4: Mögliche Aufteilung der Ansichten in dem Hauptanwendungsfenster.

6.2 CrossNative als hierarchisches Modell

Die Knotentypen `GridGenerator` und `STLFile` werden durch die in Abbildung 6.5 gezeigten gleichnamigen Klassen repräsentiert. Die Schnittstelle `Geometry` dient zur Entkopplung der Klasse `GridGenerator` von der Implementierung des `STLFiles`. Die Verknüpfung des `GridGenerators` mit Derivaten der `Geometry`-Schnittstelle erfolgt mit den Methoden `addGeometry` und `removeGeometry`. Der Aufruf der `getGrid`-Methode veranlasst den `GridGenerator` das Gitter zu erzeugen und als Objekt des Typs `Grid` zurückzugeben. Die Daten der verknüpften Geometrien werden dafür mit Hilfe der `getTriangles`-Methode ermittelt.

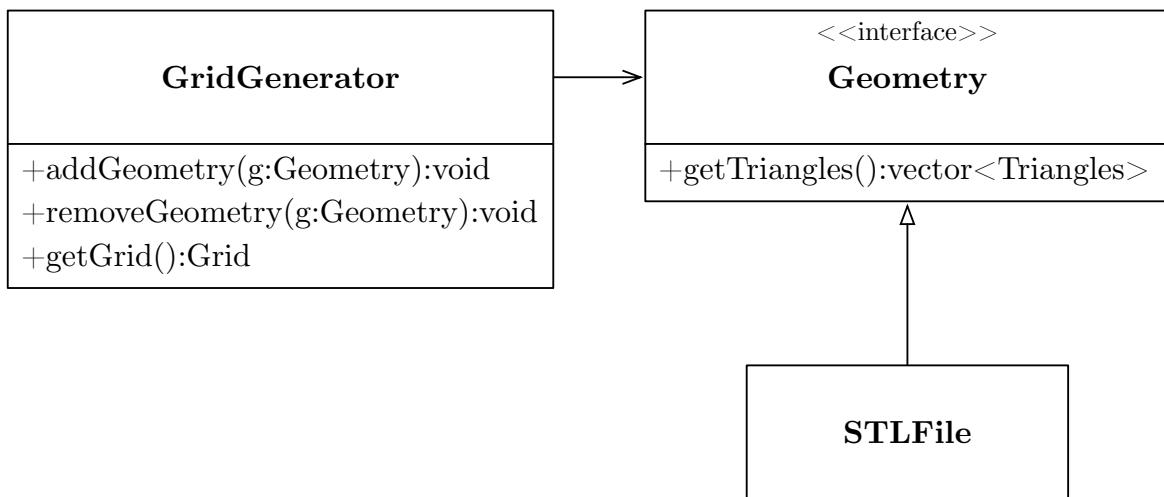


Abbildung 6.5: Klassenbeziehungen des `GridGenerators`.

Die Entkopplung des `GridGenerators` durch das *Interface* `Geometry` ermöglicht einen leichten Austausch der Datenquelle. So können anstatt von STL-Dateien beispielsweise ebenfalls Dateien in anderweitigen Formaten eingelesen werden. Darüber hinaus sind Erweiterungen der Software um neuartige Knotentypen denkbar. Zum Beispiel könnte das erzeugte Gitter des `GridGenerators`, wie in Abbildung 6.6 veranschaulicht, als Eingangsparameter für die Simulation von Strömungen dienen. In der hierarchischen Struktur des *Solution Explorers* würde sich das durch einen `GridGenerator` als Kindknoten einer `Simulation` widerspiegeln.

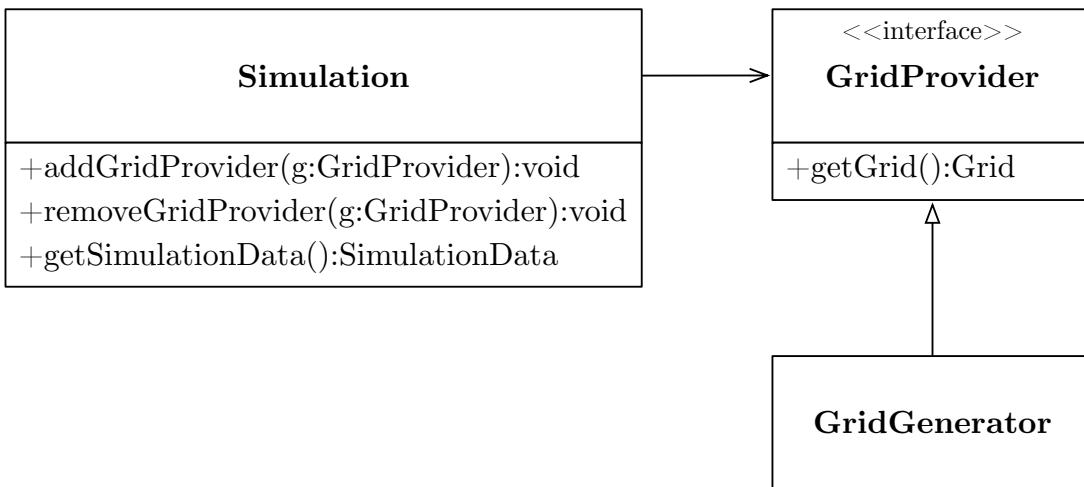


Abbildung 6.6: Klassenbeziehungen der *Simulation*.

Für die nachhaltige Umsetzung der Modellstruktur bedarf es daher einer einheitlichen Abstraktion, die die flexible Organisation der Knoten in hierarchischer Form erlaubt und gleichzeitig die leichte Erweiterbarkeit der Software um neuartige Knoten unterstützt. Aufgrund der Typ-basierten Komposition der unterschiedlichen Knotentypen benötigt das Klassensystem des Modells des Weiteren einen Mechanismus, der sicherstellt, dass nur gültige Eltern-Kind-Beziehungen erstellt werden.

Die zuvor genannten Anforderungen verdeutlichen die Ähnlichkeit zur Komposition von View-Hierarchien. Zur Organisation der hierarchischen Modellstruktur kann daher das in Kapitel 5 erläuterte System des Frameworks CrossNative verwendet werden. Wie die Komposition der Views basiert die Modellhierarchie auf dem *Composite Pattern* und verwendet das *Interface CNComponent* nach Abbildung 5.5 (siehe Seite 98). Die Typ-basierten Verknüpfungen übernimmt die bereits implementierte Klasse *CNComposable* in Kombination mit dem *CNVisitingComposer*. Infolgedessen erfordert ein zusammensetzbarer **GridGenerator**-Knoten im Kontext des Frameworks CrossNative ein Klassensystem nach Abbildung 6.7. Damit Objekte der Typen **GridGenerator** und **Geometry** in das System des **CNComposables** injiziert werden können, bedarf es einer Vererbungsbeziehung zu der Schnittstelle **CNVisitable** sowie der Implementierung der **accept**-Methode. Für die korrekte Zusammensetzung durch einen **CNVisitingComposer** muss zusätzlich ein entsprechendes Composing-Decomposing-Visitor-Paar implementiert wer-

den. Um während des *double-dispatch*-Mechanismus Verknüpfungen zu Geometrien herstellen bzw. aufheben zu können, leiten sich die beiden Besucher von der typspezifischen Besucherschnittstelle **GeometryVisitor** ab. Für das Auslösen der **visit**-Methoden delegieren die **Geometry**-Derivate, wie zum Beispiel die Klasse **STLFile**, das Akzeptieren der Besucher an **CNAcceptors**, die den Typ **GeometryVisitor** erkennen (siehe Abschnitt 5.4).

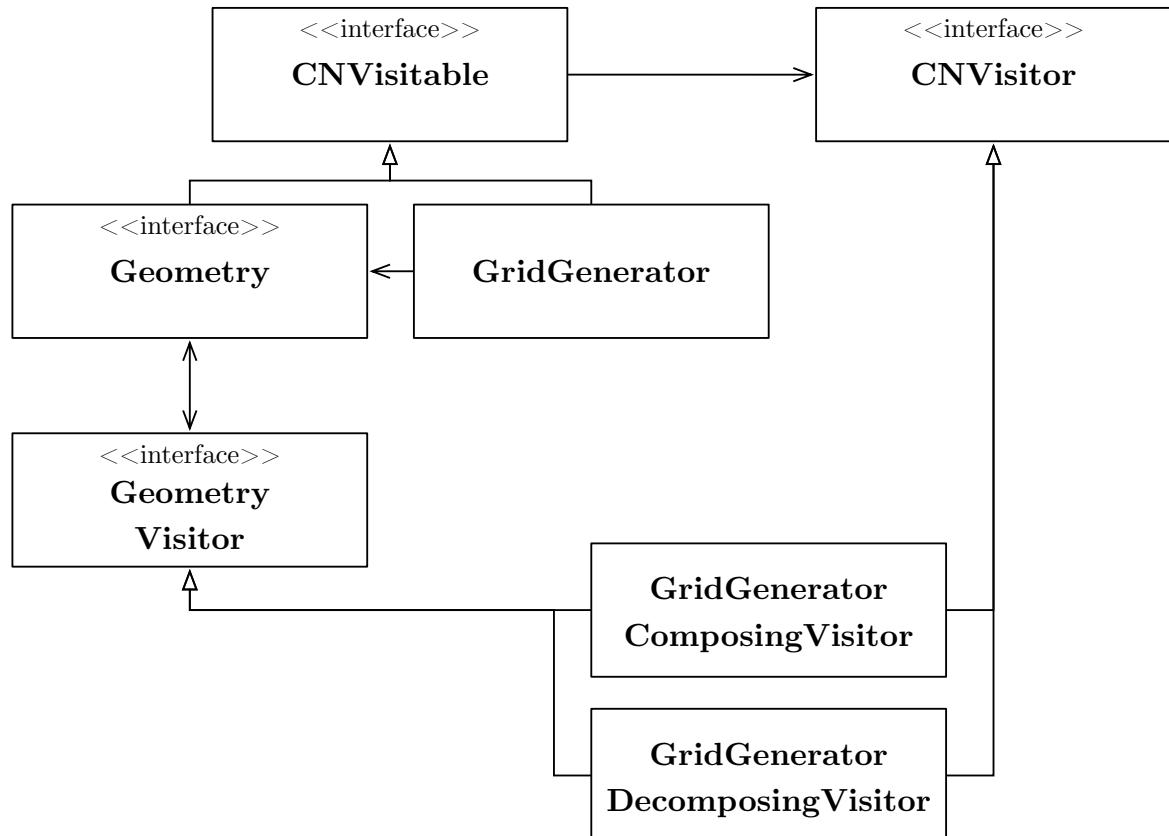


Abbildung 6.7: Zusammensetzen des *GridGenerators* unter Verwendung des azyklischen Besuchermusters.

Wie bei der Komposition von Views können ebenfalls im Modell nicht alle Knoten miteinander verknüpft werden, da die Typen der gekapselten **CNVisitable**-Derivate nicht immer zueinander passen. Im Anwendungsbeispiel des Gittergenerators ist es beispielsweise nur möglich, eine Eltern-Kind-Beziehung zwischen **GridGenerators** und **STLFiles** zu erstellen. In jedem anderen Fall würde der *double-dispatch*-Mechanismus während der Verknüpfung scheitern und die **CNVisitableVisitorMismatchException** auslösen. Um

das Erstellen fehlerhafter Verknüpfungen zu vermeiden, kann auf das **CNMatcher**-Derivat **CNVisitingMatcher** zurückgegriffen werden, das **CNVisitable**-Derivate in Abhängigkeit des verwendeten **CNIdentifiers** identifiziert.

Soll einem Knoten der Hierarchie zum Beispiel ein **CNComposable** hinzugefügt werden, das ein **STLFile** als **CNVisitable**-Derivat kapselt, wird zunächst mit Hilfe eines **CNVisitingMatchers** überprüft, ob es sich bei dem Knoten der Hierarchie um einen möglichen Elternknoten handelt. Dafür wird der **CNVisitingMatcher** in diesem Fall mit einem **GridGeneratorIdentifier** ausgestattet, der den Typ **GridGenerator** erkennt. Nur wenn das **CNMatcher**-Derivat den Knoten der Hierarchie als möglichen Elternknoten identifiziert, wird der Verknüpfungsmechanismus ausgelöst.

Im Gegensatz zu der Komposition der View-Hierarchien, bei der die Einfügeposition eines neuen Knotens dynamisch anhand eines mitgelieferten **CNMatcher**-Derivats ermittelt wird (siehe Kapitel 4), ergibt sich die Einfügeposition der Modellhierarchie aus der Interaktion mit dem *Solution Explorer*. Bei dem gewünschten Elternknoten der Einfügeoperation handelt es sich hierbei um den ausgewählten Eintrag der Baumstruktur. Die getroffene Auswahl in dem *Solution Explorer* liefert einen **CNIndex**, durch den der Knoten in der Modellhierarchie eindeutig identifiziert werden kann. Der **CNIndex** beschreibt dabei den Pfad eines Knotens relativ zum Wurzelement der Hierarchie. Da die Einfügeposition somit bekannt ist, kann auch die Einfügeoperation anhand des **CNIndex** erfolgen. Zur Organisation der Modellhierarchie existiert die in Abbildung 6.8 dargestellte Klasse **CNHierarchy**.

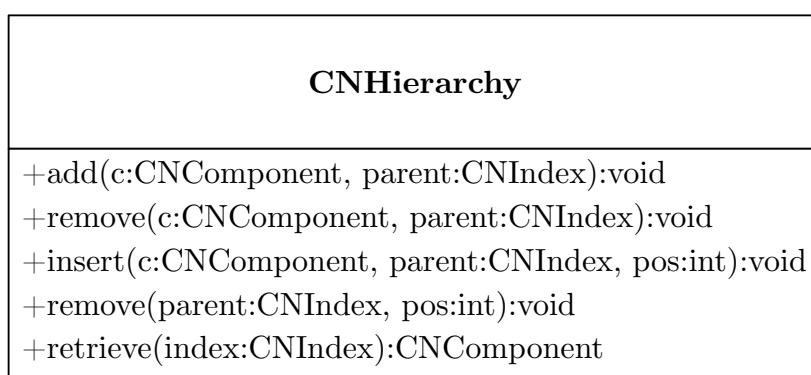


Abbildung 6.8: Die API der Klasse **CNHierarchy**.

6.3 Implementierung des Solution Explorers

Die Visualisierung der Modellhierarchie erfolgt in der Ansicht des *Solution Explorers*, die von der gleichnamigen View-Klasse `SolutionExplorer` repräsentiert wird. Für die Darstellung der Baumstruktur verwendet sie intern die Klasse `QTreeView` des GUI-Frameworks Qt. Die Datengrundlage der hierarchischen Ansicht bildet das `TreeModel`, das sich von der Qt-basierten Schnittstelle `QAbstractItemModel` ableitet. `QTreeView` und `TreeModel` sind dabei durch eine Form des Beobachtermusters (Gamma et al., 2004) miteinander gekoppelt, sodass sich die Ansicht der Baumstruktur aktualisiert, wenn sich die Daten des `TreeModels` verändern. Die Daten des `TreeModels` werden durch `TreeItems` repräsentiert, die einzelnen Knoten in der Baumstruktur entsprechen. Um die hierarchische Struktur der Modelldaten widerzuspiegeln, kann auch aus `TreeItems` eine komplexe Hierarchie aufgebaut werden. Erneut findet dafür das *Composite Pattern* Anwendung. Jedes `TreeItem` in dem `TreeModel` steht für einen Knoten der Modellhierarchie, dessen Namen und Typ der Ansicht zur Verfügung gestellt wird (siehe 6.2, Seite 131). Für die korrekte Darstellung der Modelldaten in der Baumstruktur muss der `SolutionExplorer` die Konsistenz zwischen der Modellhierarchie und dem `TreeModel` wahren. Wird etwa ein Knoten in die Modellhierarchie eingefügt, muss der `SolutionExplorer` ebenfalls ein `TreeItem` mit den Daten des neuen Knotens an die entsprechende Position des `TreeModels` setzen. Die Erzeugung des `TreeItems` delegiert der `SolutionExplorer` nach dem *Abstract Factory Pattern* (Gamma et al., 2004) an ein Derivat der Schnittstelle `TreeItemFactory`. Abbildung 6.9 veranschaulicht die zuvor beschriebenen Klassenbeziehungen des `SolutionExplorers`.

Ebenso wie bei dem Hinzufügen erfordert auch das Entfernen eines Knotens der Modellhierarchie die Anpassung des `TreeModels`. In beiden Fällen muss der `SolutionExplorer` über die Manipulation der Daten informiert werden. Zu diesem Zweck stellt die `CNHierarchy` das Interface `CNHierarchyListener` zu Verfügung, die das Beobachten der Modellhierarchie ermöglicht. Erfolgt eine Manipulation durch Hinzufügen oder Entfernen eines Knotens, ruft die Modellhierarchie anschließend die Methoden `onRemove` bzw. `onInsert` der registrierten `CNHierarchyListener` auf und informiert sie über die ausgeführte Operation. Damit der `SolutionExplorer` die Konsistenz der beiden hier-

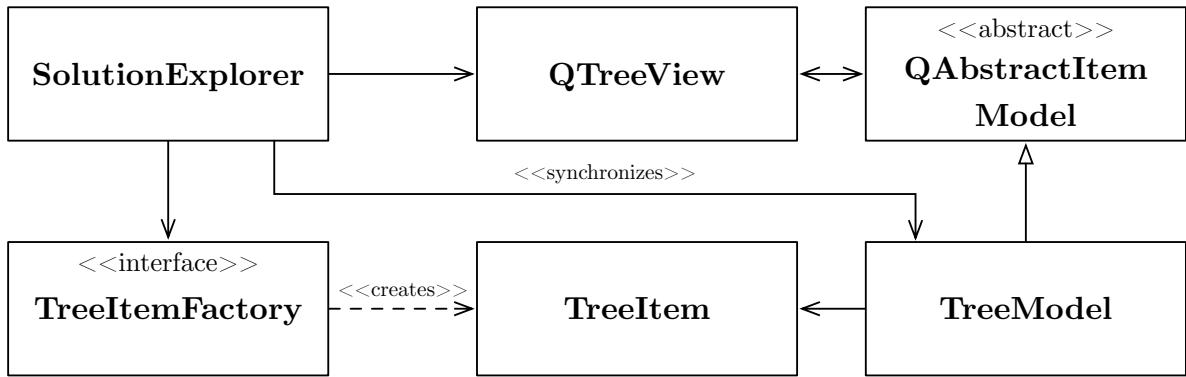


Abbildung 6.9: Klassenbeziehungen des **SolutionExplorer**.

archischen Strukturen gewährleisten kann, leitet er sich analog zu Abbildung 6.10 von dem *Interface CNHierarchyListener* ab und wird bei der Modellhierarchie registriert. Der Aufruf der `onRemove`-Methode veranlasst den **SolutionExplorer**, das entsprechende **TreeItem** aus dem **TreeModel** zu entfernen. Im Gegensatz dazu delegiert er in der `onInsert`-Methode mit der erhaltenen **CNComponent** die Erzeugung eines **TreeItems** an die **TreeItemFactory**. Das erstellte Objekt fügt er anschließend an der entsprechenden Position im **TreeModel** ein.

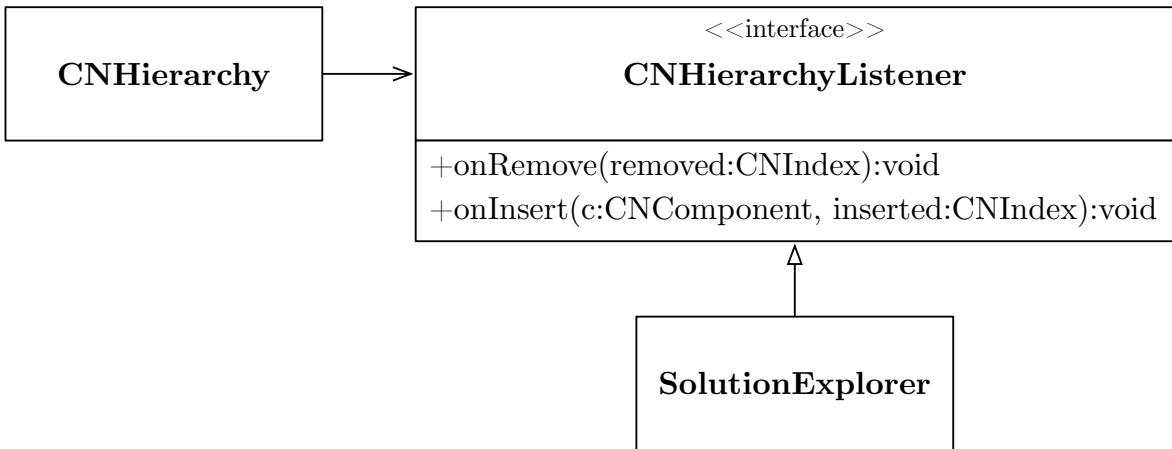


Abbildung 6.10: Beobachten der Modellhierarchie durch den **SolutionExplorer**.

Eine weitere Aufgabe des **SolutionExplorers** beinhaltet das Propagieren des ausgewählten Index der Baumstruktur. Damit sich andere Ansichten der GUI basierend auf der Auswahl des **QTreeViews** aktualisieren können, ohne in Kontakt mit dem **SolutionExplorer** zu stehen, wird der ausgewählte Index in Derivaten der Schnitt-

stelle `SelectionModel` gespeichert. Ändert sich die Auswahl in der Ansicht löst der `QTreeView` mit Hilfe des *Signal-Slot*-Mechanismus¹ von Qt ein *Signal* aus, das die neue Auswahl beinhaltet. Um auf dieses *Signal* entsprechend zu reagieren, implementiert der `SolutionExplorer` den *Slot* `onSelectionChanged`, währenddessen die neue Auswahl an die in Abbildung 6.11 gezeigte Methode `setSelectedIndex` des `SelectionModels` übergeben wird.

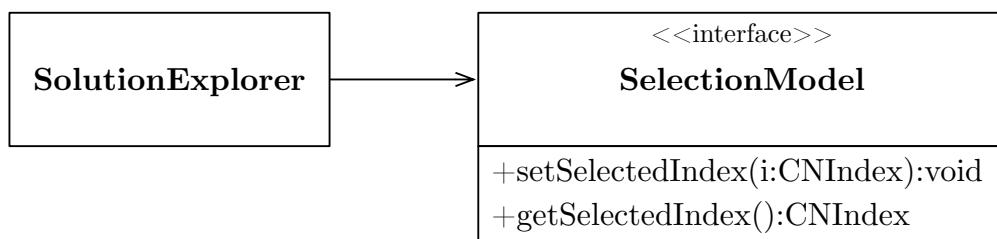


Abbildung 6.11: Propagieren des ausgewählten `CNIndex` durch den `SolutionExplorer`

6.4 Implementierung des Properties Explorers

Wie in Abschnitt 6.1 erläutert, enthält der *Properties Explorer* eine tabellarische Ansicht, die die Eigenschaften des ausgewählten Knotens der Baumstruktur anzeigt. Für die Repräsentation dieser View existiert die Klasse `PropertiesExplorer`. Zur Aktualisierung der tabellarischen Ansicht nach einer Änderung der Auswahl in dem `SolutionExplorer` verwendet auch der `PropertiesExplorer` das Interface `SelectionModel`. Der Aufruf der `getSelectedIndex`-Methode liefert den ausgewählten `CNIndex` der Baumstruktur. Mit diesem Index greift der `PropertiesExplorer` anschließend durch das *Interface* `CNHierarchyAccess` auf die Modellhierarchie zu und erhält dadurch Zugriff auf den ausgewählten Knoten. Basierend auf diesem Modellknoten aktualisiert der `PropertiesExplorer` dann die Eigenschaftstabelle.

Für die Aktualisierung der Tabelle muss der `PropertiesExplorer` über die Manipulation der Auswahl informiert werden. Zu diesem Zweck wird das *Observer Pattern* nach Gamma et al. (2004) umgesetzt. Der `PropertiesExplorer` leitet sich dazu von

¹<http://doc.qt.io/qt-5.9/signalsandslots.html>

der Schnittstelle `Observer` ab, die es erlaubt, ihn bei dem `ObservableSelectionModel` als Beobachter zu registrieren. Nach dem Speichern des neuen Index während der `setSelectedIndex`-Methode ruft dieses Derivat des `SelectionModels` die `update`-Methoden der registrierten Beobachter auf und informiert sie somit über die Änderung des Zustands. Im Falle des `PropertiesExplorers` wird dadurch die Aktualisierung der Eigenschaftstabelle ausgelöst. Abbildung 6.12 zeigt die zuvor beschriebenen Klassenbeziehungen.

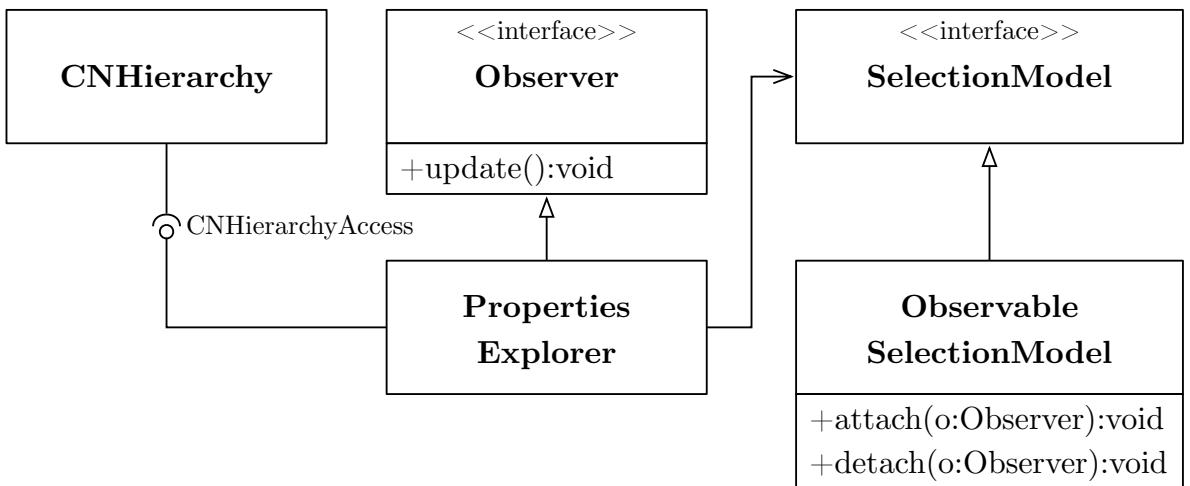


Abbildung 6.12: Beobachten der aktuellen Auswahl durch den `PropertiesExplorer`.

Als grafische Repräsentation verwendet der `PropertiesExplorer` die Implementierungsklasse `QTableView` des GUI-Frameworks Qt, die Daten in tabellarischer Form darstellt. Wie die Klasse `QTreeView` basiert sie auf einem Derivat der Schnittstelle `QAbstractItemModel`, das die Daten für die Darstellung enthält. In diesem Fall wird dafür die Klasse `TableModel` verwendet. Das `TableModel` setzt sich aus `TableItems` zusammen, die jeweils eine Zeile der Tabelle widerspiegeln und den Namen sowie den Wert einer Eigenschaft für die Darstellung zur Verfügung stellen (siehe Abbildung 6.3, Seite 132). Im Gegensatz zu dem `SolutionExplorer`, der zur Aktualisierung der Ansicht nur einzelne `TreeItems` hinzufügt oder entfernt, ersetzt der `PropertiesExplorer` die gesamte Datengrundlage. Für jeden ausgewählten Knoten wird ein neues `TableModel` erstellt, das als neue Datengrundlage dient. Die Erzeugung des `TableModels` inklusive der `TableItems` delegiert der `PropertiesExplorer` nach dem *Abstract Factory Pattern*

an ein Derivat der Schnittstelle `TableModelFactory`. Abbildung 6.13 zeigt die Klassenbeziehungen für die Aktualisierung der tabellarischen Ansicht.

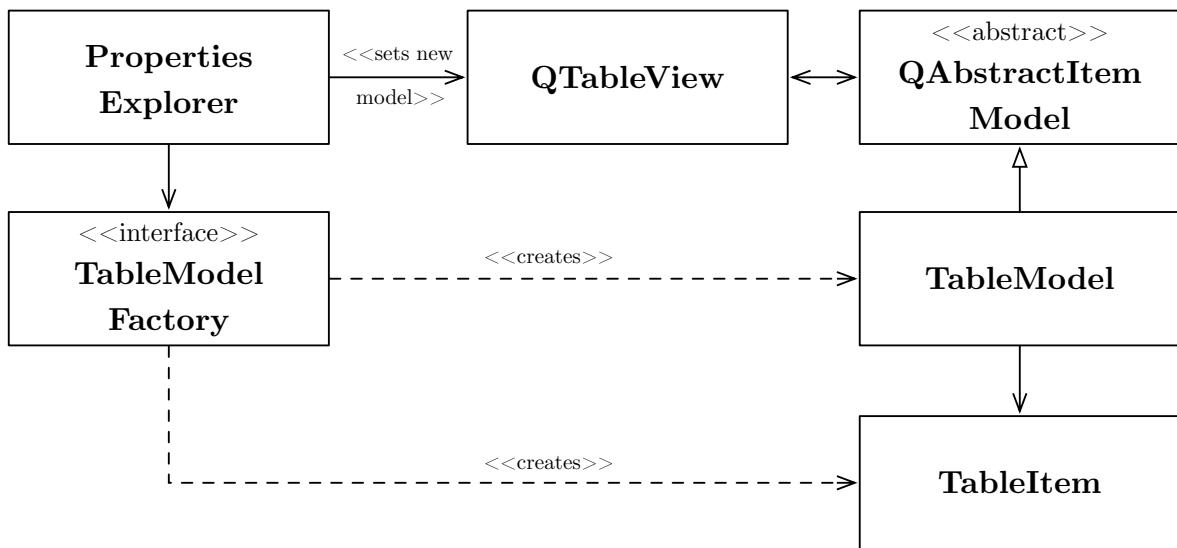


Abbildung 6.13: Aktualisierung der tabellarischen Ansicht des `PropertiesExplorers`.

6.5 Implementierung der Knotenvisualisierungen

Die Visualisierung der Knoten wird von weiteren View-Klassen übernommen. Es existiert für jeden Knotentyp eine eigene View-Klasse, die auf die Daten der Knoten zugreift, um sie anschließend grafisch darzustellen. Die Visualisierung der Gitter und der STL-Dateien erfolgt unter Verwendung des Frameworks VTK², das für die Kombination mit dem GUI-Framework Qt die Klasse `QVTKWidget`³ zur Verfügung stellt. Die Vererbung zu `QWidget` erlaubt es Instanzen dieser Klasse als Bestandteil in frameworkspezifischen Objekthierarchien entsprechend Abschnitt 3.1.1 einzugehen. Abbildung 6.14 zeigt exemplarisch das Klassensystem zur Visualisierung der generierten Gitter. Über die `getGrid`-Methode des `GridGenerators` erhält die Klasse `GridView` Zugriff auf die Daten des erzeugten Gitters, die es anschließend zur Darstellung an die grafische Repräsentation `QVTKWidget` übermittelt.

² „The Visualization Toolkit“ – <https://www.vtk.org/>

³<https://www.vtk.org/doc/nightly/html/classQVTKWidget.html>

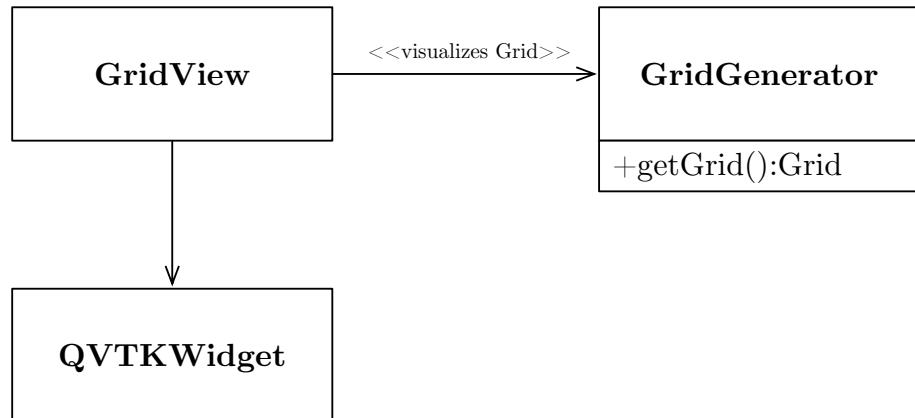


Abbildung 6.14: Visualisierung eines Gitters mit Hilfe des `QVTKWidgets`.

Nach dem gleichen System werden ebenfalls die STL-Dateien visualisiert. Es existiert eine Klasse `STLFileView`, die als grafische Repräsentation eine Instanz der Klasse `QVTKWidget` und als Datenquelle ein `STLFile` beinhaltet.

6.6 Implementierung der Menüsteuerung

Die Menüsteuerung einer grafischen Oberfläche stellt Schaltflächen zur Verfügung, über die der Benutzer applikationsspezifische Daten manipulieren oder Prozesse wie z.B. Berechnungen oder Visualisierungen starten kann. Bei diesen Schaltflächen handelt es sich um Menüeinträge, die sich zur Übersichtlichkeit der GUI thematisch zusammengehörend in Menüs gruppieren lassen. Die Menüs wiederum sind entweder als Untermenü Bestandteil anderer Menüs oder befinden sich direkt in der Menüleiste. Wie in Abschnitt 6.1 bereits erwähnt, enthält die Menüsteuerung des Gittergenerators die vier Menüeinträge „*Delete*“, „*Add STLfile*“, „*Add GridGenerator*“ und „*Visualize*“, die sich auf die drei Menüs „*Edit*“, „*Add*“ und „*Evaluate*“ aufteilen. Der nachfolgende Abschnitt erläutert die Klassensysteme der View-Klassen `MenuEntry`, `Menu` und `MenuBar` zur Erzeugung der geforderten Menüsteuerung.

6.6.1 Implementierung der Menüeinträge

Grundsätzlich werden Menüeinträge, wie Abschnitt 3.1.1 verdeutlicht, im Kontext des GUI-Frameworks Qt mit Hilfe der Klasse `QAction` umgesetzt. Daher verwendet die View-Klasse `MenuEntry` `QAction` als ihre grafische Repräsentation.

Das Betätigen der Schaltfläche des Menüintrags löst das Qt-basierte *Signal triggered* der `QAction` aus. Um entsprechend zu reagieren, implementiert die View-Klasse `MenuEntry` den *Slot onTriggered*, der das *Signal* empfängt und die gewünschte Handlung ausführt. Während zum Beispiel das Betätigen der Schaltfläche „*Add GridGenerator*“ des „*Add*“-Menüs einen *GridGenerator*-Knoten erstellt und diesen anschließend der `CNHierarchy` hinzufügt, entfernt der Menüeintrag „*Delete*“ des „*Edit*“-Menüs den aktuell ausgewählten Knoten der Modellhierarchie. Die Implementierung des *Slots* ist dabei die einzige Differenz in den unterschiedlichen Menüeinträgen. Die Umsetzung einer View-Klasse für jede der verschiedenen Schaltflächen würde daher zu erheblichen Redundanzen führen. Zur Vermeidung dieser Redundanzen wird das *Transaction Pattern*⁴ nach Gamma et al. (2004) angewendet. Anstatt die auszuführende Operation selbst zu implementierten, delegiert sie die Klasse `MenuEntry` durch den Aufruf der `execute`-Methode an ein Derivat der Schnittstelle `TransAction`, das während der Konstruktion injiziert wird. Durch die Delegation der auszuführenden Handlung an die Schnittstelle `TransAction` erfüllt die View-Klasse das *Dependency Inversion Principle* und das *Open/Closed Principle*. Infolgedessen erhöht sich ihre Wiederverwendbarkeit. Für neue Menüeinträge muss lediglich eine entsprechendes `TransAction`-Derivat implementiert und in die Klasse `MenuEntry` injiziert werden. Zur Umsetzung der Menüsteuerung des Gittergenerators existieren die `TransAction`-Derivate `AddSTLFileTransAction`, `AddGridGeneratorTransAction`, `DeleteTransAction` sowie die Klasse `AddVisualizationTransAction`. Abbildung 6.15 veranschaulicht die Klassenbeziehungen der View-Klasse `MenuEntry`.

Neben einem Derivat der `TransAction`-Schnittstelle wird dem `MenuEntry` während der Konstruktion ebenfalls ein Titel in Form einer Zeichenkette (`string`) übergeben, den die Schaltfläche in der Menüsteuerung anzeigt.

⁴auch bekannt als *Command Pattern* (Gamma et al., 2004)

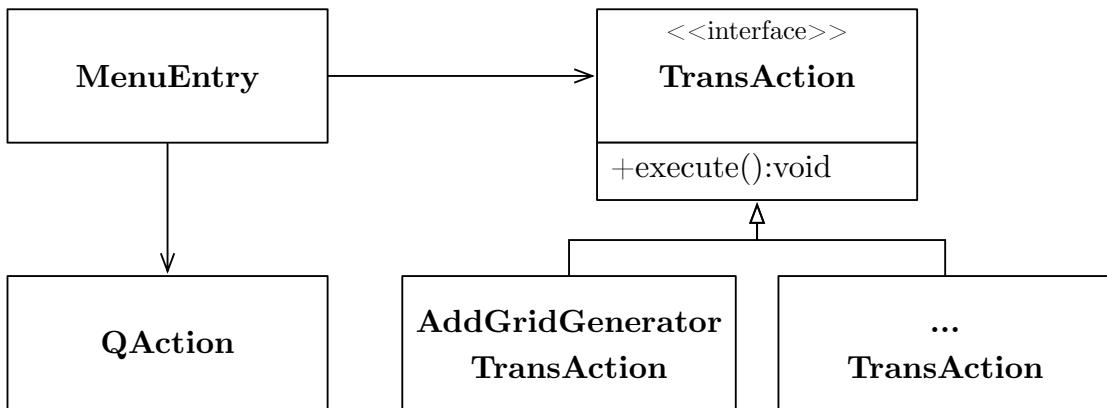


Abbildung 6.15: Klassenbeziehungen der View-Klasse **MenuEntry**.

6.6.2 Implementierung der Menüs

Die Umsetzung von Menüs erfolgt mit der View-Klasse **Menu**, die nach Abschnitt 3.1.1 ebenfalls die Klasse **QAction** als grafische Repräsentation verwendet. Im Gegensatz zu den Menüeinträgen beinhaltet das **Menu** jedoch zusätzlich eine Instanz der Klasse **QMenu**, die der **QAction** über die `setMenu`-Methode als Menü gesetzt wird (siehe Abbildung 3.1, Seite 35) und in der Oberfläche das Menü widerspiegelt. Für das Hinzufügen und das Entfernen von Schaltflächen besitzt das **QMenu** die Methoden `addAction` und `removeAction`, die Objekte des Typs **QAction** entgegennehmen.

Eine korrekte Verknüpfung der Views **MenuEntry** und **Menu** erfordert es, die von dem **MenuEntry** verwaltete **QAction** in das **QMenu** aufzunehmen. Dafür stellen die Menüeinträge die Methode `getQAction` bereit. Somit kann das **Menu** die grafische Repräsentation der Menüeinträge extrahieren und über die `addAction`-Methode in sein **QMenu** aufnehmen. Nach gleichem Prinzip werden die gekapselten **QAction**-Objekte der Menüeinträge über die `removeAction`-Methode aus dem **QMenu** entfernt. Auch die Verknüpfung zu anderen Views ist möglich, solange sie auf **QAction** basieren und die `getQAction`-Methode besitzen. Um die Klasse **Menu** von der konkreten Implementierung anderer Views zu entkoppeln, existiert für die Verknüpfung analog zu Abschnitt 5.1 das Interface **QActionBased** (vgl. **NSMenuItemBased**, Abbildung 5.4, Seite 97). Für das Erstellen bzw. Aufheben von Verknüpfungen zu **QAction**-basierten Views weist die Klasse **Menu** die Methoden `add` und `remove` nach Abbildung 6.16 auf. Da sich auch

Menu selbst von der Schnittstelle `QActionBased` ableitet, ist ebenfalls eine Verknüpfung zwischen Menüs möglich. Ebenso wie bei der Erzeugung von Menüeinträgen wird auch hier bei der Konstruktion ein Titel injiziert, den das Menü in der grafischen Oberfläche erhält.

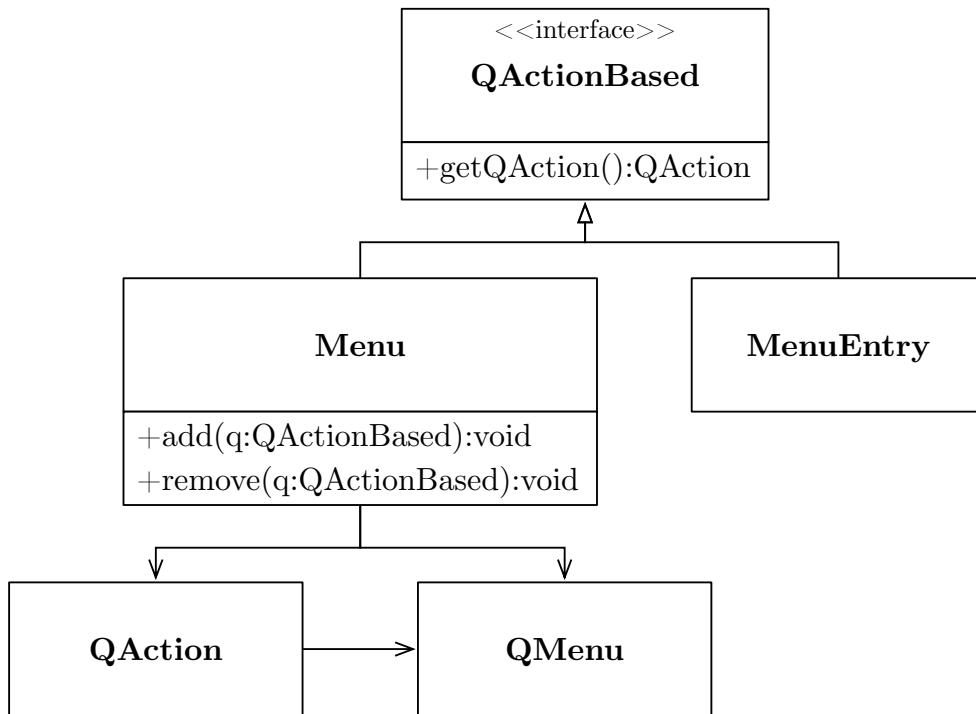


Abbildung 6.16: Klassensystem der Klasse `Menu`.

6.6.3 Implementierung der Menüleiste

Die Menüleiste der Oberfläche entspricht der Klasse `MenuBar`, die auf der Implementierungsklasse `QMenuBar` basiert. Wie `QMenu` besitzt auch `QMenuBar` die Methoden `addAction` und `removeAction` für das Hinzufügen und Entfernen von Schaltflächen. Die Verknüpfung von Views geschieht daher nach dem gleichen Prinzip wie bei der in Abschnitt 6.6.2 gezeigten View-Klasse `Menu`. Zur Verknüpfung der `MenuBar` mit allen `QAction`-basierten Views besitzt sie die Methoden `add` und `remove`, die Objekte des Typs `QActionBased` entgegennehmen. Abbildung 6.17 zeigt die Klassenbeziehungen der View-Klasse `MenuBar`.

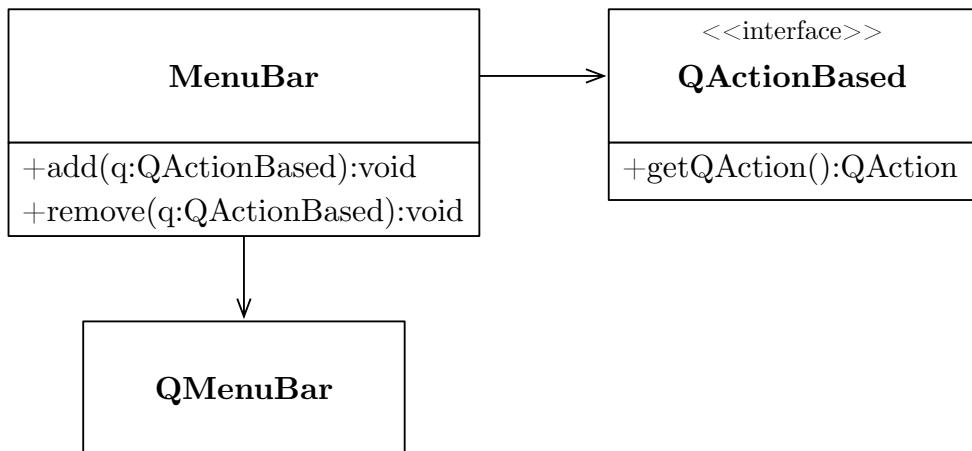


Abbildung 6.17: Klassenbeziehungen der Klasse `MenuBar`.

6.7 Implementierung des Hauptanwendungsfensters

Das Hauptanwendungsfenster einer Applikation wirkt wie ein Container, der die einzelnen Views zu der gesamten grafischen Oberfläche zusammenfügt. In diesem Fall teilt sich das Layout des Hauptanwendungsfensters in drei Bereiche ein, in denen Views platziert werden können. Dabei handelt es sich entsprechend Abbildung 6.18 um die Bereiche der linken und rechten Seitenleiste sowie um einen mittleren Bereich dazwischen. Für eine Komposition der GUI nach Abbildung 6.4 werden die Views *Solution Explorer* und *Properties Explorer* in der linken Seitenleiste angesiedelt. Der mittlere Bereich dient zur Visualisierung der Knotentypen und umfasst daher die Views vom Typ `GridView` und `STLFileDialog`. Neben dieser möglichen Zusammensetzung der GUI ist ebenfalls eine Komposition denkbar, in der beide *Explorer*-Ansichten in der rechten Seitenleiste oder getrennt voneinander in unterschiedlichen Bereichen angesiedelt sind. Enthält eine Seitenleiste keine Views, wird sie wie die rechte Seitenleiste in Abbildung 6.4 minimiert, sodass sie in der Oberfläche nicht zu sehen ist.

Beide Seitenleisten reihen ihre jeweilig enthaltenen Views in vertikaler Ausrichtung auf und stellen sie untereinander dar (siehe linke Seitenleiste, Abbildung 6.4). Die Größeneinteilung der Views kann dabei durch den Benutzer verändert werden, indem er die horizontale Grenze zwischen den unterschiedlichen Ansichten der Seitenleisten mit Hilfe der Maus verschiebt. Für diese Art der Darstellung bietet das Framework

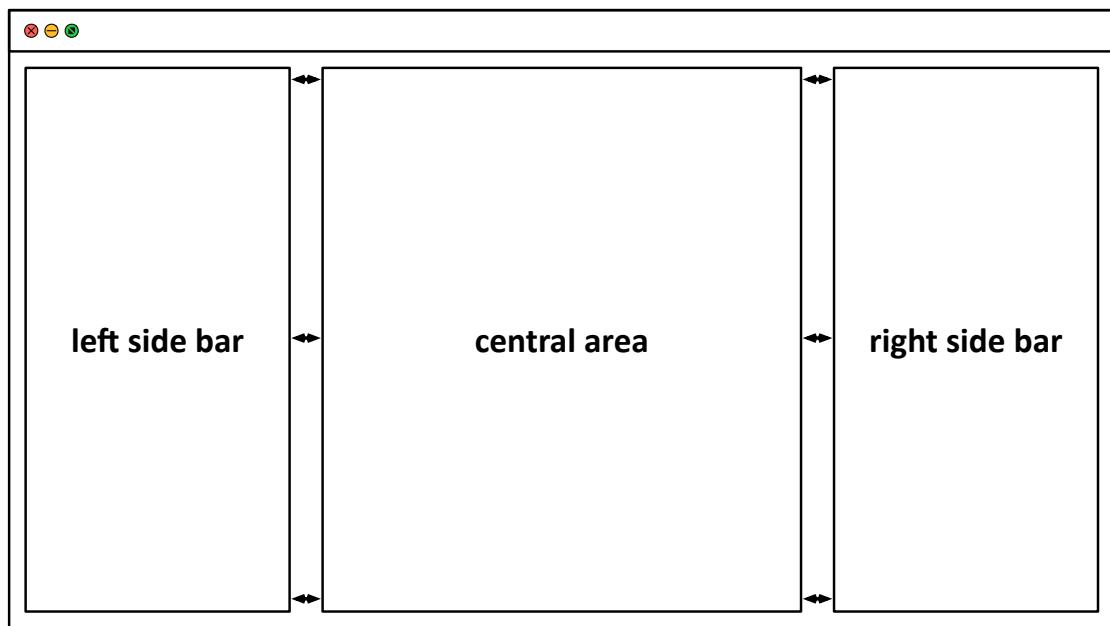


Abbildung 6.18: Konzeptionelle Darstellung der *Shell*.

Qt die Container-Klasse `QSplitter` an, der View-Objekte des Typs `QWidget` über die `addWidget`-Methode hinzugefügt werden können. Im Gegensatz zu den Seitenleisten, die alle ihre Views gleichzeitig anzeigen, stellt der Visualisierungsbereich in der Mitte des Hauptanwendungsfensters zu jedem Zeitpunkt immer nur eine seiner Views über die gesamte Größe des Bereichs dar. Zugleich enthält er zu jeder hinzugefügten View einen Reiter, über den der Benutzer die momentan angezeigte View variieren kann. Diese Funktionalität der GUI wird über die Klasse `QTabWidget` erreicht, die das Hauptanwendungsfenster für die Darstellung des mittleren Bereichs verwendet. Das Hinzufügen einer Ansicht zu einem `QTabWidget` basiert ebenfalls auf dem Typ `QWidget`. Auch die Bereiche des Hauptanwendungsfensters sollen in ihrer Größeneinteilung durch den Benutzer verändert werden können. Aus diesem Grund befinden sie sich ebenfalls in einem `QSplitter`, der zur Darstellung innerhalb eines Anwendungsfensters als zentrales `QWidget` (siehe Abschnitt 3.1.1) bei einem `QMainWindow` gesetzt wird. Da dieser `QSplitter` die Bereiche gegensätzlich zur den Views in den Seitenleisten nebeneinander anzeigen soll, muss seine Ausrichtung über die `setOrientation`-Methode entsprechend konfiguriert werden. Es entsteht eine frameworkspezifische Objekthierarchie nach Abbildung 6.19.

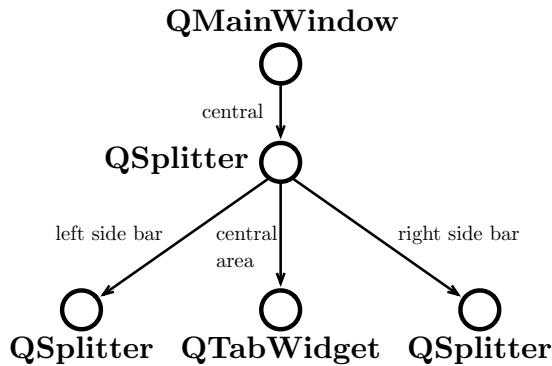


Abbildung 6.19: Frameworkspezifische Objekthierarchie der View-Klasse `Shell`.

Die Repräsentation des Hauptanwendungsfensters im Kontext der Applikation geschieht über die View-Klasse `Shell`, die die Verwaltung der gesamten frameworkspezifischen Objekthierarchie nach Abbildung 6.19 übernimmt. Im einzelnen beinhaltet die `Shell` die Logik für das Zusammensetzen der Hierarchie, das Aufnehmen und Entfernen anderer `QWidget`-basierter View-Klassen in die gewünschten Bereiche sowie die Minimierung der Seitenleisten, wenn diese keine Views enthalten. Für das Hinzufügen und Entfernen von `QWidget`-basierten View-Klassen besitzt die `Shell` nach Abbildung 6.20 `add`- und `remove`-Methoden für die jeweiligen Bereiche, die Objekte des Typs `QWidgetBased` als Übergabeparameter erhalten. Während dieser Methoden extrahiert die `Shell` die grafische Repräsentation der übergebenen View mittels der `getQWidget`-Methode und übergibt sie an die entsprechende Operationen des jeweiligen Bereichs. Damit die zuvor erläuterten View-Klassen `SolutionExplorer`, `PropertiesExplorer` sowie die Views zur Visualisierung der Knotentypen in die Bereiche der `Shell` aufgenommen werden können, müssen sie sich, wie am Beispiel des `SolutionExplorers` gezeigt, von der Schnittstelle `QWidgetBased` ableiten und ihre grafische Repräsentation als `QWidget` herausgeben. Da es sich bei den verwendeten Implementierungsklassen `QTreeView`, `QTableView` und `QVTKWidget` um Subklassen von `QWidget` handelt, ist das ohne Weiteres möglich.

Wie Abbildung 3.1 zeigt (siehe Seite 35), befindet sich die Menüleiste des Typs `QMenuBar` ebenfalls in dem Hauptanwendungsfenster. Daher fällt auch das Einfügen der Menüleiste in den Aufgabenbereich der `Shell`. Dafür existieren die Methoden `addMenuBar` bzw. `removeMenuBar`. Diese Methoden nehmen die Schnittstelle `QMenuBarBased` entgegen, von der sich die zuvor erläuterte View-Klasse `MenuBar` ableitet.

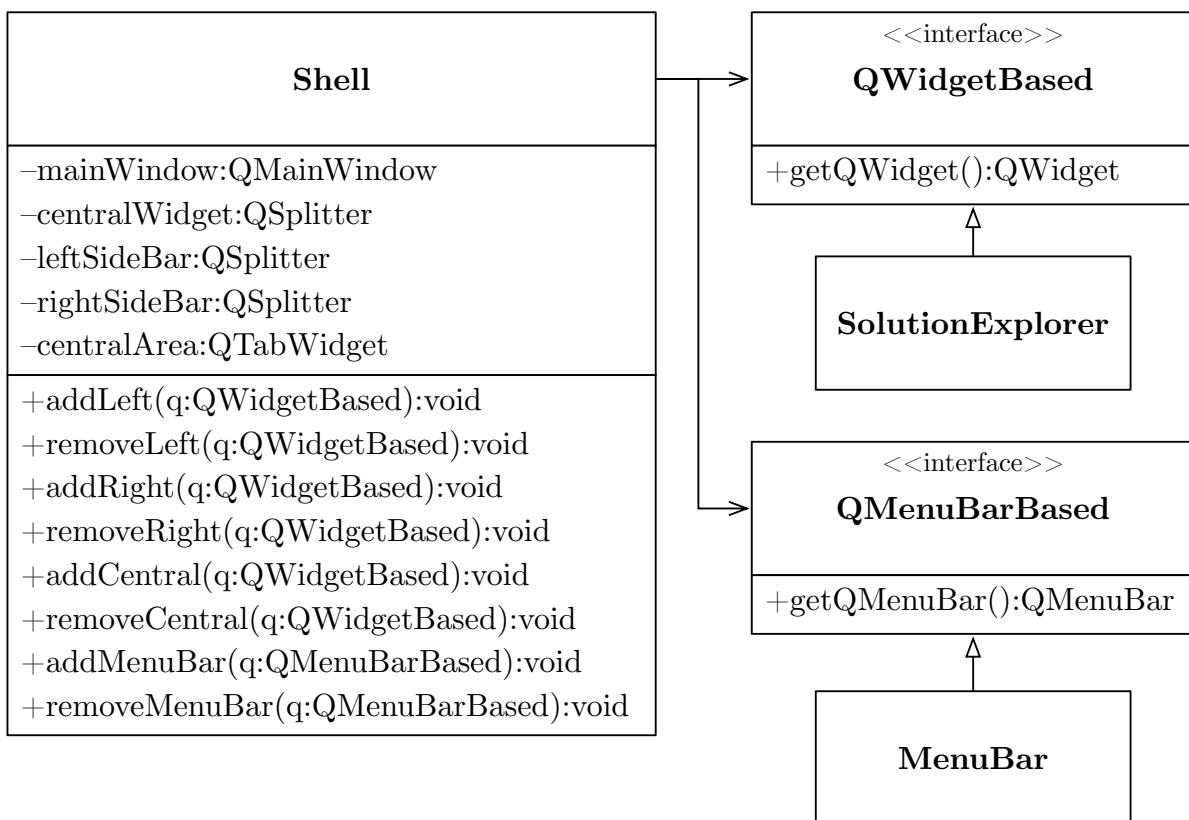


Abbildung 6.20: Klassensystem der View-Klasse **Shell**.

6.8 Komposition der GUI unter Verwendung von CrossNative

Für das Zusammensetzen der Oberfläche mit der Klasse **CNDynamicHierarchy** des Frameworks CrossNative bedarf das zuvor beschriebene System der View-Klassen Anpassungen. Aufgrund der Schlüsselrolle des azyklischen Besuchermusters wird das System zunächst für jede der View-Klassen um eine typspezifische Besucherschnittstelle ergänzt. Damit die Views im Kontext des **CNComposables** verwendet werden können, werden sie anschließend von der Schnittstelle **CNVisitable** abgeleitet. Zur Implementierung der **accept**-Methode greifen sie auf die Klasse **CNAcceptorImp** (siehe Abschnitt 5.4) zurück, der sie als möglichen Besuchertyp den Typ ihrer eigenen Besucherschnittstelle mitgeben. Der Verknüpfungsmechanismus erfolgt unter Verwendung des **CNComposer**-Derivats **CNVisitingComposer** (siehe Abschnitt 5.5). Daher benötigt das System die

Erweiterung um je ein Composing-Decomposing-Visitor-Paar für die zusammensetzbaren View-Klassen **Shell**, **MenuBar** und **Menu**. Abbildung 6.21 zeigt exemplarisch das Klassensystem für die Komposition der View-Klasse **Shell**.

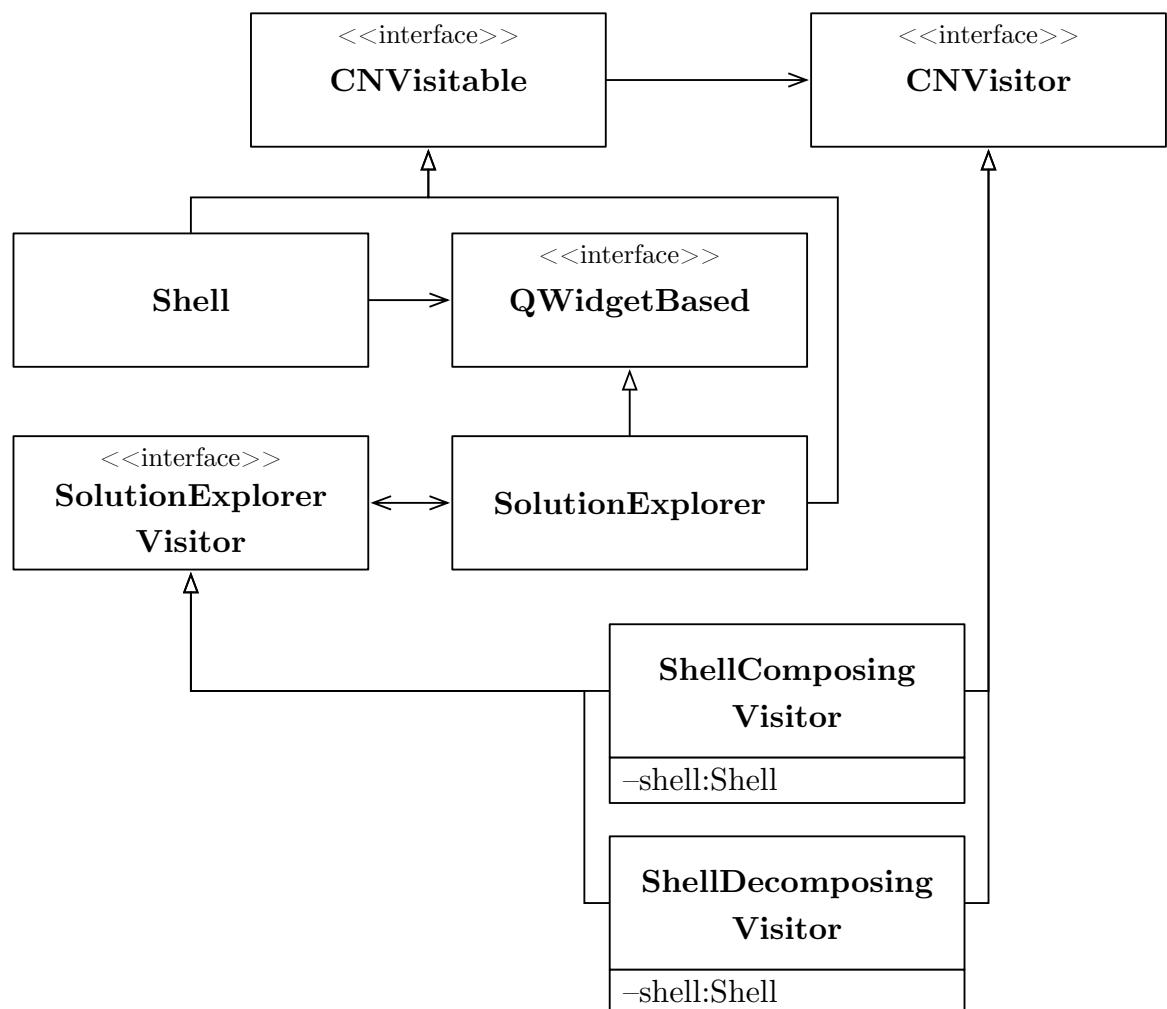


Abbildung 6.21: Klassensystem zur Komposition der **Shell** unter Verwendung von CrossNative.

Damit das Besucher-Paar der **Shell** während des *double-dispatch*-Mechanismus Instanzen der Klassen **SolutionExplorer**, **PropertiesExplorer**, **GridView** und **STLFileDialog** identifiziert und dementsprechend Verknüpfungen erstellt oder aufhebt, müssen sich sowohl der **ShellComposingVisitor** als auch der **ShellDecomposingVisitor** von den typspezifischen Besucherschnittstellen dieser View-Klassen ableiten. Verdeutlicht wird

das durch die in Abbildung 6.21 dargestellte Vererbungsbeziehung zu dem *Interface SolutionExplorerVisitor*. Die Implementierung der Besucher erfolgt analog zu Listing 6.1.

```

1  class ShellComposingVisitor
2      : public CNVisitor,
3          public SolutionExplorerVisitor,
4          public PropertiesExplorerVisitor,
5          public GridView,
6          public STLFileDialog {
7
8  public:
9      virtual void visit(SolutionExplorerPtr view) override {
10         shell->addLeft(view);
11     }
12     virtual void visit(PropertiesExplorerPtr view) override {
13         shell->addLeft(view);
14     }
15     virtual void visit(GridViewPtr view) override {
16         shell->addCentral(view);
17     }
18     virtual void visit(STLFileDialogPtr view) override {
19         shell->addCentral(view);
20     }
21
22 private:
23     ShellPtr shell;
24 };

```

Listing 6.1: Implementierung der Klasse `ShellComposingVisitor`.

Die Vererbungsbeziehung zu jeder Besucherschnittstelle bringt die Implementierung einer `visit`-Methode mit sich, in der der Besucher auf Basis des identifizierten Typs eine Verknüpfung erstellt oder aufhebt. Die Implementierung der `visit`-Methoden nach Listing 6.1 führt zu einer Komposition des Hauptanwendungsfensters nach Abbildung 6.4 (siehe Seite 132).

Nach gleichem Prinzip erfolgt ebenfalls die Implementierung der Besucher-Paare für die View-Klassen `MenuBar` und `Menu`. Während sich die Besucher der `MenuBar` zur Verknüpfung mit Menüs nur von der Schnittstelle `MenuVisitor` ableiten, besitzen die Besucher des Menüs Vererbungsbeziehungen zu `MenuVisitor` und `MenuEntryVisitor`. Für nicht zusammensetzbare Views wird als `CNComposer`-Derivat die Klasse `CNNullComposer` verwendet, die leere Implementierungen für die Methoden `mount` und `dismount` enthält.

Des Weiteren benötigt die dynamische Komposition der GUI unter Verwendung der Klasse `CNDynamicHierarchy` `CNMatcher`-Derivate, anhand derer die Elternknoten der zu ladenden Views eindeutig identifiziert werden. Hierfür bietet CrossNative den `CNVisitingMatcher` an (siehe Abschnitt 5.6), der `CNVisitable`-Derivate in Abhängigkeit des injizierten `CNIdentifier`-Derivats erkennt. Zur Typ-basierten Identifikation der `Shell` und der `MenuBar` mit Hilfe des `CNVisitingMatchers` werden die Klassen `ShellTypeIdentifier` und `MenuBarTypeIdentifier` analog zu Listing 6.2 umgesetzt.

```

1 class ShellTypeIdentifier : public CNIdentifier, public ShellVisitor {
2 //...
3 public:
4     virtual void visit(ShellPtr view) override {}
5     virtual bool hasIdentified() override {
6         return true;
7     }
8 };

```

Listing 6.2: Implementierung der Klasse `ShellTypeIdentifier`.

Im Gegensatz zu den Klassen `Shell` und `MenuBar` befinden sich von der Klasse `Menu` mehrere Instanzen in der applikationsspezifischen View-Hierarchie. Infolgedessen wird zur Identifikation der Menüs auf den *Tag*-basierten Identifikationsmechanismus zurückgegriffen. Dafür leitet sich die Klasse `Menu` von dem Interface `CNTagged` ab. Gleichzeitig erfolgt eine Anpassung des Konstruktor, sodass neben dem Titel des Menüs ein eindeutiges *Tag* bei der Erzeugung injiziert werden kann. Zusätzlich erfordert die *Tag*-basierte Identifikation der Menüs eine Modifikation der `accept`-Methode nach Listing 5.20 (siehe Seite 125).

Nach den zuvor beschriebenen Anpassungen der View-Klassen geschieht die Komposition der grafischen Oberfläche entsprechend Listing 6.3. Zunächst werden dafür die einzelnen Knoten der View-Hierarchie nach dem Schema in Listing 5.13 erzeugt (siehe Seite 114). Anschließend werden sie zusammen mit einem **CNMatcher**-Derivat an die **load**-Methode der **CNDynamicHierarchy** übergeben. Um die komplexe Konstruktion der **CNMatcher**-Derivate zu kapseln, existiert eine **MatcherFactory**, die je nach aufgerufener Fabrik-Methode ein **CNMatcher**-Derivat erstellt und mit einem **CNIdentifier**-Derivat konfiguriert. Die Methode **makeTopLevelMatcher** gibt einen **CNNullMatcher** zurück, der veranlasst, dass die mit ihm geladene **CNComponent** auf oberster Ebene der View-Hierarchie angesiedelt wird.

```

1 // instantiation of view components...
2
3 MatcherFactoryPtr matcherFactory = MatcherFactory::getNewInstance();
4 CNDynamicHierarchyPtr gui = CNDynamicHierarchy::getNewInstance();
5
6 gui->load(shellComponent, matcherFactory->makeTopLevelMatcher());
7 gui->load(menuBarComponent, matcherFactory->makeShellTypeMatcher());
8 gui->load(editMenuComponent, matcherFactory->makeMenuBarTypeMatcher());
9 gui->load(addMenuComponent, matcherFactory->makeMenuBarTypeMatcher());
10 gui->load(deleteEntryComponent, matcherFactory->makeTagMatcher("edit"));
11 gui->load(stlFileEntryComponent, matcherFactory->makeTagMatcher("add"));
12 gui->load(gridGenEntryComponent, matcherFactory->makeTagMatcher("add"));
13
14 //load solutionExplorerComponent, propertiesExplorerComponent ...

```

Listing 6.3: Implementierung der Klasse **ShellTypeIdentifier**.

6.9 Flexibilität der GUI durch CrossNative

Der Entwurf einer grafischen Oberfläche mit CrossNative beläuft sich auf die Gestaltung einzelner Views, zu deren Komposition das in Kapitel 5 beschriebene Klassensystem des Frameworks benutzt werden kann. Die einzelnen View-Klassen müssen sich dafür von der Schnittstelle `CNVisitable` ableiten und eine eigene typspezifische Besucherschnittstelle anbieten. Für Views, die sich aus untergeordneten Views zusammensetzen, bedarf es des Weiteren je ein Composing-Decomposing-Visitor-Paar, das über das Erstellen bzw. das Aufheben von Verknüpfungen zwischen den View-Klassen entscheidet. Darüber hinaus sind für alle View-Klassen, die während des Kompositionsalgorithmus anhand ihres Typs als Elternknoten identifiziert werden sollen, typspezifische `CNIIdentifier`-Derivate notwendig. Aus den unterschiedlichen Verantwortungen der zu implementierenden Klassen ergibt sich eine Modularisierung der Präsentationsschicht nach Abbildung 6.22. Während die in dem Modul `Views` enthalten Klassen die Funktionalität und das Verhalten der grafischen Bestandteile der GUI enthalten, entscheiden die Module `TypeIdentifiers` und `Composing-Decomposing-Visitors` über deren Komposition.

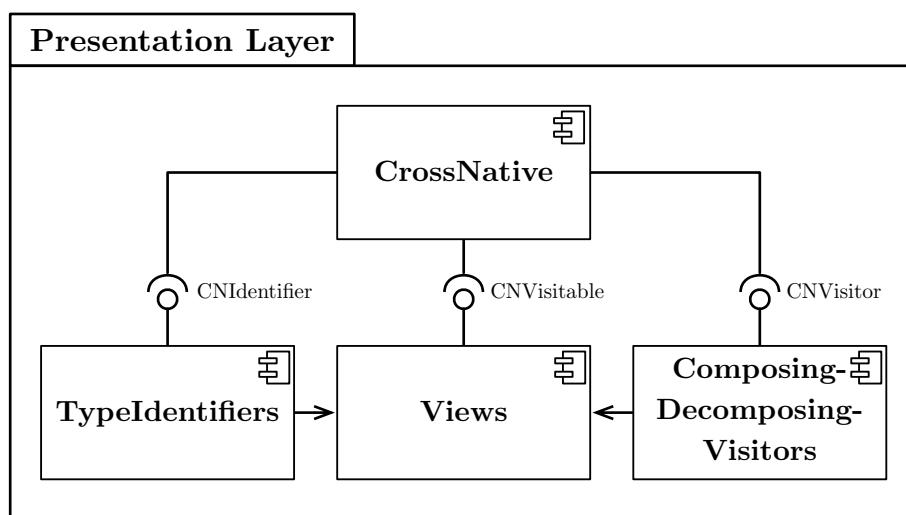


Abbildung 6.22: Die modulare Unterteilung des *Presentation Layers*.

In Abschnitt 3.3.4 (vgl. Abbildung 3.7, Seite 45) entsteht die Idee die Implementierung der View-Klassen von deren Komposition zu trennen, um beide Bereiche unabhängig voneinander variieren zu können. Durch die hervorgerufene Einteilung der

Präsentationsschicht unterstützt CrossNative diese Idee. Es nimmt sogar eine weitere Verfeinerung der Komposition vor, in dem es die Verantwortung für das Identifizieren des Elternknotens (siehe `TypeIdentifiers`) und das Erstellen bzw. das Aufheben von Verknüpfungen (siehe `Composing-Decomposing-Visitors`) separiert. Infolgedessen erlaubt CrossNative das Arrangement der grafischen Oberfläche durch kleinste Modifikationen im Quelltext zu variieren. So können beispielsweise die Positionen des *Solution Explorers* und des *Properties Explorers* innerhalb des Hauptanwendungsfensters verändert werden, in dem das Composing-Decomposing-Visitor-Paar (vgl. Listing 6.1) statt der Methoden für die linke Seitenleiste die Methoden für einen anderen Bereich des Hauptanwendungsfensters aufruft. Weitere Modifikationen des Quelltexts sind dafür nicht notwendig.

Durch die Variation des von dem `CNVisitingMatcher` verwendeten `CNIdentifier`-Derivats hingegen verändert der Benutzer von CrossNative nicht nur die Art der Verknüpfung an der selben Einfügeposition der View-Hierarchie, sondern erzwingt einen anderen Elternknoten. Dafür muss jedoch das Besucher-Paar des neuen Elternknotens in der Lage sein, eine Verknüpfung mit der zu ladenden View zu erstellen. Da sowohl die Menüleiste als auch das Menü in dem Anwendungsbeispiel des Gittergenerators Verknüpfungen zu anderen Menüs erlauben, kann die Einfügeposition eines Menüs durch die Variation des verwendeten Matchers modifiziert werden. Lädt der Benutzer beispielsweise die `editMenuComponent` aus Listing 6.3 mit einem Matcher, der die `addMenuComponent` als Elternknoten identifiziert, wird die `editMenuComponent` dort als Kind eingefügt. Da der Kompositionsalgorithmus die Einfügepositionen dynamisch ermittelt, hat die veränderte Position des „Edit“-Menüs keinen Einfluss auf das Laden seiner Kindknoten (vgl. `deleteEntryComponent`, Listing 6.3). Somit können durch minimale Änderungen des Quelltexts ganze Teilbäume der Hierarchie verschoben werden.

Aufgrund der Entkopplung der Views durch Schnittstellen wie z.B. `QWidgetBased` oder `QMenuBarBased` (siehe Abbildung 6.20) bedingen sich die konkreten Implementierungen der verschiedenen Views nicht gegenseitig. Zusammensetzbare Views geben durch diese Schnittstellen vor, welche Implementierungsklassen des GUI-Frameworks in die eigene grafische Repräsentation aufgenommen werden können. Solange sich neue Views von diesen Schnittstellen ableiten und auf der geforderten Implementierungsklasse des

verwendeten GUI-Frameworks basieren, rufen Erweiterungen der Oberfläche keine Modifikationen in übergeordneten Views hervor. Da jedoch die Identifizierung der neuen View-Klasse während des Verknüpfungsmechanismus auf Basis ihres konkreten Typs geschieht, muss das Composing-Decomposing-Visitor-Paar der übergeordneten View entsprechend angepasst werden. Diese Designschwäche lässt sich durch die Anwendung des *Dependency Inversion Principles* überwinden. Anstatt die Komposition basierend auf dem konkreten Typ vorzunehmen, kann eine kategorische Identifizierung erfolgen. Abbildung 6.23 verdeutlicht die Funktionsweise der kategorischen Identifizierung am Beispiel des zusammensetzbaren Hauptanwendungsfensters.

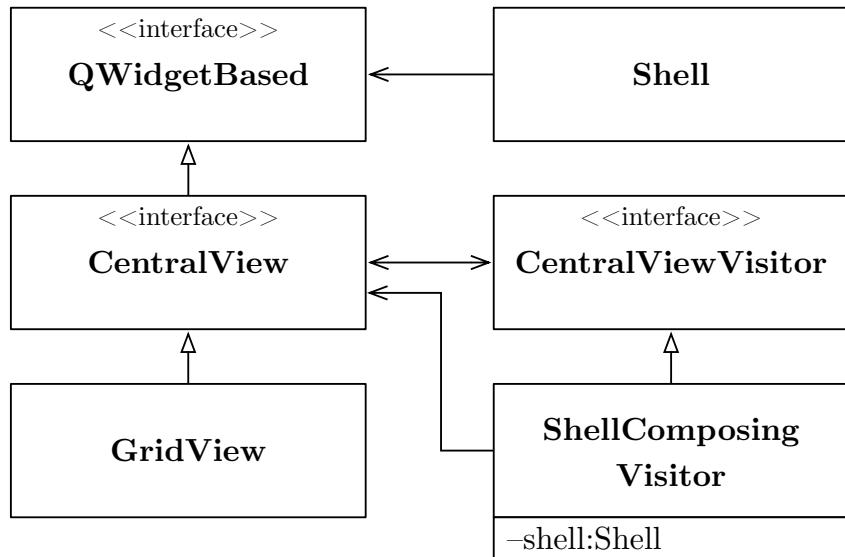


Abbildung 6.23: Kategorische Identifizierung des ShellComposingVisitors.

Für jeden Bereich des Hauptanwendungsfensters definiert das zugehörige Besucher-Paar einen eigenen abstrakten Typ (vgl. **CentralView**), der wiederum eine typspezifische Besucherschnittstelle (vgl. **CentralViewVisitor**) mit sich bringt. Die Vererbungsbeziehung zu dieser Schnittstelle ermöglicht dem Besucher den von ihm definierten Typ und damit auch den Bereich des Hauptanwendungsfensters zu identifizieren. Nach gleichem Prinzip existieren bei der Verwendung der kategorischen Identifizierung ebenfalls die *Interfaces LeftView* und *RightView* sowie die entsprechenden Besucherschnittstellen.

6.10 Zusammenfassung

In diesem Kapitel wird gezeigt, wie das entstandene Framework CrossNative verwendet werden kann, um Applikationen mit grafischen Oberflächen zu gestalten. Als Beispiel dafür dient der von Peters (2016) entworfene Gittergenerator, der aus dreidimensionalen Geometrien diskrete Berechnungsgitter erzeugt. Hierbei wird Qt als GUI-Framework herangezogen. Die Oberfläche soll dem Benutzer ermöglichen, die zur Gittergenerierung notwendigen Daten zu modellieren und zu visualisieren sowie den Generierungsprozess anzustoßen und das Resultat anzuzeigen. Als Datengrundlage für die Erzeugung der Gitter werden STL-Dateien verwendet, die dreidimensionale Oberflächen anhand von Dreiecken beschreiben. Die Modellierung dieses Prozesses erfolgt in einer hierarchischen Struktur, in der sich Knoten unterschiedlicher konkreter Typen befinden. Es existieren die Knotentypen *GridGenerator*, der den Prozess der Gittergenerierung repräsentiert, und *STLFile*, der die eingelesenen Daten einer STL-Datei kapselt. Es wird gezeigt wie CrossNative aufgrund der verwandten Entwurfsproblematik zu der Gestaltung von komplexen grafischen Oberflächen und der durch die testgetriebene Entwicklung entstandenen wiederverwendbaren Struktur auch für die Modellierung der hierarchischen Datenstruktur verwendet werden kann.

Neben der Menüleiste und ihren Menüs enthält die GUI zur Visualisierung der Daten und Prozesse verschiedene Views. So übernimmt der *Solution Explorer* die Darstellung der hierarchischen Datenstruktur innerhalb einer Baumstruktur. Der *Properties Explorer* verwendet eine tabellarische Ansicht um die Eigenschaften der unterschiedlichen Knoten anzuzeigen. Dabei enthält die Tabelle zu jedem Zeitpunkt die Eigenschaften des momentan ausgewählten Knotens der Baumstruktur. Die Visualisierungen der Knotentypen geschieht mit Hilfe des Frameworks VTK in den Klassen *GridView* und *STLFileView*. Ein gerahmt werden die unterschiedlichen Ansichten von dem Hauptanwendungsfenster, das in der dynamischen Hierarchie der View-Objekte den Elternknoten der zuvor genannten Views darstellt. Im Laufe des Kapitels wird zunächst erläutert, wie die einzelnen View-Klassen für sich implementiert werden können, ohne dabei die Verwendung von CrossNative zu berücksichtigen. Anschließend wird gezeigt, welche Modifikationen der View-Klassen notwendig sind, um eine Komposition durch CrossNative zu ermöglichen.

Letztlich wird die Flexibilität hervorgehoben, die durch die Komposition der GUI mit CrossNative erzeugt wird. Es wird gezeigt, dass die Implementierung einer grafischen Oberfläche mit CrossNative zu einer Präsentationsschicht führt, in der die Logik für das Identifizieren eines Elternknotens, das Zusammensetzen einer View und die Implementierung der Views selbst voneinander getrennt werden. Dadurch ergibt sich eine grafische Oberfläche, die den Anforderungen aus der Konzeption in Kapitel 3 gerecht wird.

KAPITEL 7

Zusammenfassung & Ausblick

Diese Arbeit veranschaulicht die Anwendung der testgetriebenen Entwicklungsmethode nach Beck (2003). Als Anwendungsbeispiel dient der Entwurf des Frameworks CrossNative, das dem Benutzer die Möglichkeit bietet grafische Oberflächen mit geringem Aufwand trotz hoher Komplexität und breitem Funktionsumfang flexibel und wandlungsfähig zu gestalten.

Zu Beginn dieser Arbeit werden die Grundlagen der testgetriebenen Entwicklung vorgestellt. Es wird gezeigt, wie die inkrementelle Vorgehensweise dieser Entwurfsmethodik sowie die vorgezogene Implementierung von *Unit Tests* kontinuierlich Feedback erzeugt, das den Benutzer bei der Umsetzung qualitativ hochwertiger Software unterstützt. Ein weiterer Vorteil ist das inhärent testbare Design der Software, durch das zugleich auch die Metriken und Prinzipien des objektorientierten Designs wie z.B. Kohäsion und Kopplung oder die SOLID-Prinzipien eingehalten werden. Darüber hinaus lässt die testgetriebene Entwicklung durch die *Unit Tests* eine leicht verständliche und anschauliche Dokumentation des Systems entstehen.

7. Zusammenfassung & Ausblick

Anschließend erfolgt die Konzeption des umzusetzenden Frameworks, in der die zu treffenden Anforderungen erarbeitet werden. Zunächst wird dafür der Aufbau von grafischen Oberflächen analysiert, wobei aus der internen Sicht eine hierarchische Objektstruktur identifiziert wird. Weiterhin werden einige Entwurfsschwierigkeiten bei der Gestaltung grafischer Oberflächen im Allgemeinen und bei dem Entwurf von komplexen hierarchischen Strukturen betrachtet, die die umzusetzende Funktionalität von CrossNative herausstellen. Als Schwerpunkt des Frameworks ergibt sich daraus die dynamische Modellierung von komplexen Objekthierarchien, die hohe Flexibilität in ihrem Arrangement aufweisen und den leichten Austausch einzelner Objekte ermöglichen.

Nachfolgend wird die testgetriebene Entwicklung eingesetzt, um CrossNative entsprechend der zuvor definierten Anforderungen umzusetzen und ihre Vorgehensweise sowie ihren Einfluss auf den Entwickler, den Entwurfsprozess und die resultierende Software zu veranschaulichen. Zunächst wird dafür die Implementierung und die Wirkungsweise dieser Entwurfsmethode auf eine einzelne zu testende Komponente gezeigt. Als Beispiel dafür dient der Kompositionsalgorithmus von CrossNative, der für das zusammensetzen von Objekten zu einer komplexen Hierarchie zuständig ist. Dabei wird demonstriert, wie verständliche und wartbare *Unit Tests* implementiert werden können, wie die Verwendung von *Test Doubles* zu lose gekoppelten und flexiblen Systemen führt und wie das erzeugte Feedback durch die iterative Implementierung der Testmethoden den Entwickler dabei unterstützt Designentscheidungen zu treffen.

In einem nächsten Schritt wird die Betrachtung auf das gesamte Framework ausgeweitet und es wird veranschaulicht, welche Auswirkungen die testgetriebene Entwicklung auf die Entstehung komplexer Klassensysteme besitzt. Durch die Anzahl an abstrakten Schnittstellen kann auch hier gezeigt werden, dass die Verwendung von *Test Doubles* zu lose gekoppelten Systemen führt, die über eine hohe Flexibilität und eine leichte Erweiterbarkeit verfügen. Auch wird erläutert, wie die entstehende Redundanz während der Entwicklung verwendet wird, um die Notwendigkeit für bestimmte Designstrukturen aufzudecken. Dadurch wird sichergestellt, dass das Design die für die Umsetzung der Problemdomäne benötigte Komplexität nicht überschreitet, wodurch sich die Wartbarkeit und die Verständlichkeit des Systems erhöht.

Abschließend wird CrossNative angewendet, um die leichte Anwendbarkeit, die Erweiterbarkeit und die Flexibilität der entstandenen Software zu demonstrieren. Als Beispiel dafür dient der von Peters (2016) entworfene Gittergenerator, der als Grundlage für Strömungssimulationen diskrete Gitter aus dreidimensionalen Geometrien erzeugt. Unter Verwendung von CrossNative wird eine grafische Oberfläche umgesetzt, die dem Benutzer ermöglicht, die für die Generierung benötigten Daten zu modellieren und die Ergebnisse der Problemdomäne zu visualisieren. Darüber hinaus wird durch die Verwendung von CrossNative im Modell der Applikation dargelegt, dass durch die testgetriebene Entwicklung Software entsteht, die zu hohem Maße wiederverwendbar ist. Demnach kann CrossNative nicht nur für die Gestaltung von grafischen Oberflächen verwendet werden.

Aus persönlicher Erfahrung lässt sich bestätigen, dass die testgetriebene Entwicklung unterstützend während des Entwurfsprozesses wirkt. Die Anwendung dieser Methode hilft dabei, sich auf die Umsetzung momentaner Funktionalität zu konzentrieren und zukünftige Eventualitäten zu vernachlässigen. Dadurch senkt sich die kognitive Auslastung des Entwicklers, was sich positiv auf die zu treffenden Designentscheidungen auswirkt. Infolgedessen steigt die Qualität des objektorientierten Designs der Software, wodurch sich die Wartbarkeit und die Flexibilität erhöhen und das System anpassungsfähig gegenüber neuen Anforderungen wird. Des Weiteren steigert die Möglichkeit, durch vorhandene *Unit Tests* jederzeit Feedback über die Fehlerfreiheit der Software erzeugen zu können, das Selbstvertrauen des Entwicklers Änderungen in dem Produktionscode vorzunehmen, ohne bereits implementierte Funktionalität zu gefährden. Als Folge daraus wird der Entwickler zu Modifikationen der Software zur Verbesserung ihrer Qualität ermutigt.

Zusammenfassend wird im Rahmen dieser Arbeit gezeigt, wie unter Anwendung der testgetriebenen Entwicklung fehlerfreie, leicht verständliche, einfach anwendbare, erweiterbare und wiederverwendbare Softwaresysteme umgesetzt werden können, die über ein qualitativ hochwertiges Design und eine vollständige Dokumentation verfügen. Dennoch ist die testgetriebene Entwicklung nicht als Garantie für hochwertige Software zu betrachten. Nur bei korrekter Anwendung stellen sich die positiven Effekte auf den Pro-

7. Zusammenfassung & Ausblick

duktionscode, den Entwurfsprozess und den Entwickler ein. Da das Design des Systems von den Testmethoden impliziert wird, handelt es sich bei der Qualität der implementierten *Unit Tests* um einen der wichtigsten Faktoren. Durch unzulägliches Design der Testmethoden kann die Anwendung der testgetriebenen Entwicklung sogar destruktiv auf den Entwurfsprozess sowie auf die Qualität des Produkts wirken.

Neben der testgetriebenen Entwicklung existieren weitere agile Praktiken, die sich aufgrund ihrer eigenen Art des Feedbacks positiv auf den Entwurf von Software auswirken. Um die Rolle der testgetriebenen Entwicklung in der agilen Softwareentwicklung einzurorden oder möglicherweise mit verschiedenen agilen Ansätzen zu kombinieren, könnten weitere Praktiken anhand von Anwendungsbeispielen veranschaulicht sowie ihre Wirkungsweisen diskutiert werden. Weitere Arbeiten könnten sich dafür unter anderem mit der Anwendung der *Continuous*-Ansätze wie z.B. *Continuous Integration* und *Continuous Deployment* oder mit der Container-Virtualisierung von Software befassen.

Literatur

- Beck, K. (2003). *Test-driven development: by example*. Boston: Addison-Wesley.
- Booch, G. (2007). *Object-oriented analysis and design with applications*. 3. ed., 1. printing. The Addison-Wesley object technology series. Upper Saddle River, NJ: Addison-Wesley. URL: <http://proquest.tech.safaribooksonline.de/9780201895513>.
- Brown, K. et al. (2003). *Enterprise Java Programming with IBM WebSphere*. Enterprise Java Programming with IBM WebSphere. Addison-Wesley. URL: <https://books.google.de/books?id=KBIXQPNGOmIC>.
- Buschmann, F. et al. (1996). *Pattern-Oriented Software Architecture: a system of patterns*. Bd. 1. John Wiley und Sons.
- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley.
- Fowler, M. et al. (2000). *UML Distilled (2Nd Ed.): A Brief Guide to the Standard Object Modeling Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Freeman, S. et al. (2009). *Growing Object-Oriented Software, Guided by Tests*. 1st. Addison-Wesley Professional.
- Gamma, E. et al. (2004). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl., [Neuaufl.] Programmer's choice. München und Boston [u.a.]: Addison-Wesley.

Literatur

- Kirch, U. et al. (2015). *C++ - Lernen und professionell anwenden*. Heidelberg: MITP-Verlags GmbH & Co. KG.
- Martin, R. C. (2009). *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall.
- Martin, R. C. et al. (2008). *Agile principles, patterns, and practices in C#*. 4. print. Robert C. Martin series. Upper Saddle River, NJ: Prentice-Hall.
- Meszaros, G. (2006). *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR.
- Meyer, B. (2009). *Object-oriented software construction*. 2. ed., 15. print. Upper Saddle River, NJ: Prentice Hall PTR.
- Osherove, R. (2014). *The Art of Unit Testing: With Examples in C#*. 2nd. Greenwich, CT, USA: Manning Publications Co.
- Peters, S. (2016). "Effiziente Implementierung eines Gittergenerators auf GPGPUs". In: *Forum Bauinformatik. Tagungsband*. Hrsg. von T. Berthold et al. 28. Aufl. Bd. 28, S. 165–176.
- Yourdon, E. et al. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ: Prentice-Hall.

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 13. Februar 2018

Janis Zisenis