Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Theoretische Informatik und Technische Informatik
Fachgebiet Prozessinformatik

Studienarbeit

Mustersuche in UML-Modellen mit Together 6.0.1

Véronique Spranger
30434

Februar 2004

Betreut durch Dipl. Inf Detlef Streitferdt

Inhaltsverzeichnis

1 Einleitung

Bei dieser Arbeit geht es um die automatische Suche von Entwurfsmustern in UML-Diagrammen. Heutzutage werden die meisten Softwaresysteme nicht ganz neu entwickelt, sondern es finden auf der Basis schon existierender Softwaresystemen Änderungen, Weiterentwicklungen, Anpassungen statt [Balzert98]. Für eine sichere Durchführung dieser Änderungen ist das Verstehen des Systems unabdingbar.

Um ein Softwaresystem zu verstehen, sind die dazugehörigen Dokumentation, Entwurf und Spezifikationsmodelle notwendig. Der übliche Fall ist allerdings, dass diese Dokumente ganz oder teilweise fehlen und die Systemsentwickler nicht immer für die Erklärung der Funktionalitäten des Systems zur Verfügung stehen.

In einem solchen Fall kann man nur noch aus dem Quellcode versuchen, die Funktionalität eines Softwaresystems zu verstehen. Aber allein aus dem Quelltext eines Programms wirklich zu verstehen, wie ein Softwaresystem funktioniert und welche Ideen dahinterstecken, ist eine ermüdende und zeitraubende Angelegenheit, die oft nicht zur vollen Zufriedenheit gelöst werden kann – mit dem Resultat, dass nicht genau überprüfbar ist, ob vorgenommene Änderungen nicht doch irgendwelche unerwünschten Seiteneffekte mit sich bringen.[Naum 01]

Entwurfsmuster stellen Musterlösungen für Entwurfsprobleme der objektorientierten Softwareentwicklung zur Verfügung. Sie spielen also eine große Rolle beim Verstehen eines Softwaresystems. Weiß man, welche Muster verwendet wurden, um ein Softwaresystem zu entwerfen, so kann man besser verstehen, wie dieses funktioniert. Aber ohne Dokumentation, wo beschrieben sein könnte, welche Muster zum Ansatz kamen, kann man diese Information nur noch aus dem Quellcode bekommen. Es ist aber keine einfache Aufgabe für den Menschen , manuell, aus dem Quellcode Muster zu extrahieren, also wäre es ein großer Schritt und eine große Erleichterung für ihn, diese Aufgabe von einem Softwaresystem lösen zu lassen. Es existieren verschiedene Ansätze die dieses Ziel verfolgen.

Um eine automatische Findung von Entwurfsmuster zu ermöglichen ist es wichtig sie zu beschreiben bzw charakterisieren.

Diese Arbeit basiert auf dem Ansatz von Naumann [Naum 01]. Er hat im Rahmen seiner Diplomarbeit einen Ansatz beschrieben, der auf dem Auffinden von Strukturen in UML-Modellen basiert. Er hat an jedem Muster Merkmale festgestellt, die bei der Anwendung eines Musters in der Struktur auftreten müssen. Außerdem hat er, um die Zahlen der falsch erkannten Muster zu verringern, Merkmale definiert, die nicht in dem Muster vorkommen dürfen.

Damit aus dem Quellcode UML-Modelle entstehen, ist Reverse Engineering notwendig. Unter Reverse Engineering versteht man die Extraktion und Repräsentation von Informationen aus einem Softwaresystem (Spezifikation) in einer anderen Form oder auf einem höheren Abstraktionsniveau. [Mül97].

Also versucht Reverse Engineering durch Analyse des Quelltextes, UML-Modelle zu extrahieren.

Im Rahmen dieser Studienarbeit wird ein Werkzeug (Together) verwendet, welches die Reverse Engineering Aufgaben übernimmt.

Auf der Basis der aus [Naum 01] entstandenen Suchalgorithmen sollen hier eine Teil der 23 Muster aus [Gamma+96] implementiert werden. Die in dieser Arbeit betrachteten Muster Muster sind:

- Kompositum
- Besucher

Zu diesem Zweck soll das Modul mit Hilfe der Together Open API entwickelt und in Together integriert werden.

Diese Arbeit gliedert sich wie folgt:

Zunächst wird in Kapitel 2 auf Entwurfsmuster aus [Gamma+96] eingegangen, die in dieser Studienarbeit implementierten Muster beschrieben und charakterisiert.

Kapitel 3 gibt einen Überblick über die Funktionalitäten des benutzten Werkzeugs, insbesondere wird hier die Open API beschrieben, die eine Erweiterung des Werkzeugs mit Modulen ermöglicht.

In Kapitel 4 stellt das Modul zur Mustersuche vor, die implementierten Algorithmen sowie den Test der Algorithmen an Projekten

Schließlich erfolgt in Kapitel 5 eine Zusammenfassung der Arbeit.

2 Entwurfsmuster

In diesem Abschnitt werden Entwurfsmuster aus [Gamma+96] kurz beschrieben. Die Struktur von elf der dreiundzwanzig Muster kann mit Together automatisch erzeugt werden. Die Struktur der anderen ist in [Gamma+96] mit Hilfe der OMT-Notation dargestellt. Hier werden die in dieser Arbeit implementierten Muster vorgestellt. Jedes Muster wird anhand seines Zweckes und einer kurzen Erklärung beschrieben. Wenn vorhanden, wird die Struktur mit Hilfe von UML-Diagrammen aus Together 6.0.1 dargestellt und der Zweck stammt aus dem Buch [Gamma+96] .

2.1 Definition

Christopher Alexander beschreibt ein Muster als ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung dieses Problems, so dass sie bei jedem Auftreten des Problems stets wieder angewendet werden kann. Christophe Alexander war der erste, der in der Architekturdomäne auf Muster hinwies und seine Definition trifft auch für die Objektorientierung zu und seine Arbeit wurde auch in dem Entwurf objektorientierter Software benutzt. In [Gamma+96] werden Entwurfsmuster mit vier Elementen beschrieben:

- **Mustername**: Durch den Musternamen wird ein Entwurfsproblem, seine Lösungen und Auswirkungen benannt.
- **Problemabschnitt**: Er beschreibt, wann ein Muster anzuwenden ist, welches Entwurfsproblem adressiert wird und was sein Kontext ist.
- **Lösungsabschnitt**: Er beschreibt die Elemente, die den Entwurf bilden und deren Beziehungen, Zuständigkeiten und Interaktionen.
- **Konsequenzabschnitt**: Er beschreibt die Konsequenzen der Anwendung des Musters, also die Vor- und Nachteile des resultierenden Entwurfs.

Gamma, Helm, Johnson und Vlissides [Gamma+96] definieren Entwurfsmuster als *"Beschreibung zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen."*

2.2 Kategorisieren

In [Gamma+96] werden Muster klassifiziert. In Abhängigkeit von der Aufgabe gibt es **Erzeugungsmuster**, **Strukturmuster** und **Verhaltensmuster**. Je nach Gültigkeitsbereich gibt es **klassenbasierte** Muster und **objektbasierte** Muster.

Die Abbildung 1 zeigt die Kategorien

		Aufgabe		
		Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Gültigkeitsbereich	**Klassenbasiert**	Fabrikmethode	Adapter	Interpreter Schablonenmethode
	Objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Memento Strategie Vermittler Zustand Zuständigkeitskette

Abbildung 1: Einteilung von Entwurfsmustern

2.2.1 Strukturmuster

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Man unterscheidet zwischen objektbasierten und klassenbasierten Strukturmustern.
Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen oder Implementierungen zusammenzuführen.
Objektbasierte Strukturmuster beschreiben hingegen Mittel und Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen. Bei dieser Objektkomposition kann das Kompositionsgefüge zur Laufzeit geändert werden. Dies ist mit statischer Komposition nicht möglich und stellt demzufolge eine zusätzliche Flexibilität dar.

- **Das Muster Kompositum (Composite)**

Das Muster Kompositum ist ein Strukturmuster.

Zweck

Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.

Struktur

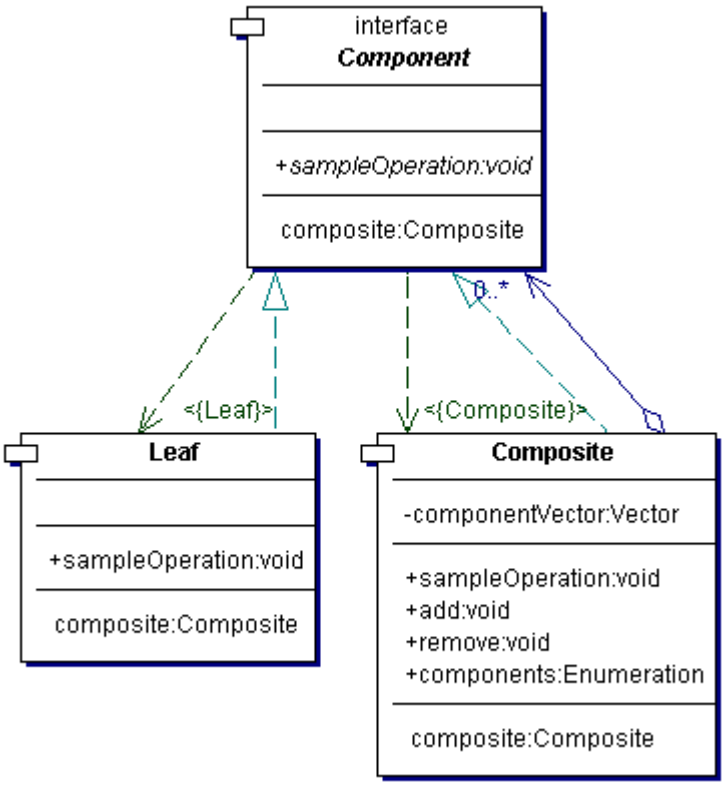

*Abbildung 2: Muster **Kompositum***

2.2.2 Verhaltensmuster

Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Sie beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch die Muster der Interaktion zwischen ihnen.

- **Besucher (Visitor)**

Zweck

Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Struktur

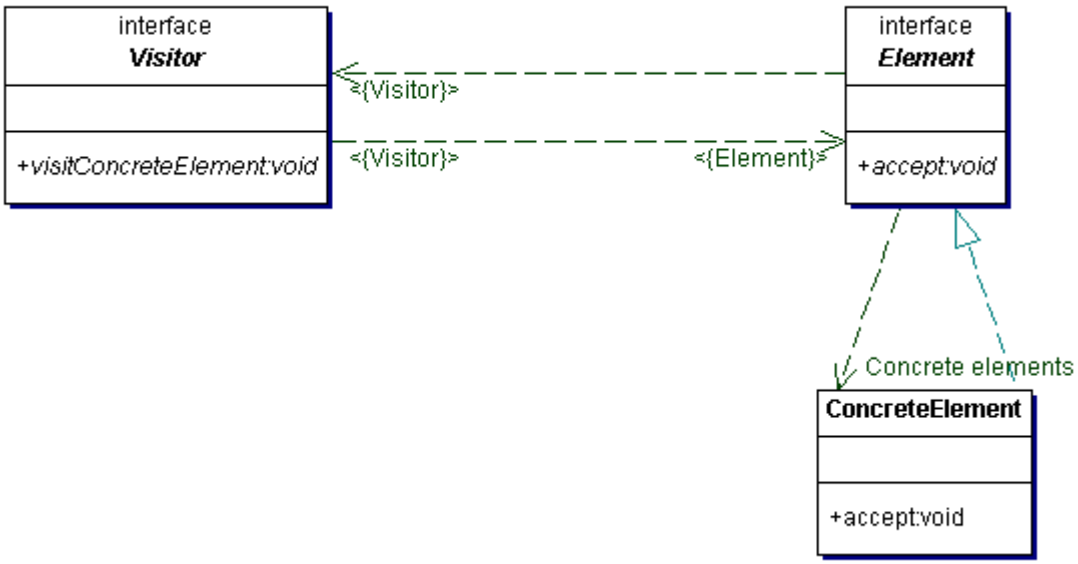

*Abbildung 3: Muster **Besucher***

3 Together 6.0.1

3.1 Allgemeine Funktionalitäten von Together

Together wurde von der Firma TogetherSoft entwickelt. Urspünglich wurde Together für die Modellierung gedacht, aber jetzt wird es viel mehr als eine komplette Entwicklungsplattform, die den kompletten Entwicklungsprozess unterstützt, von der Modellierung mit UML über die Implementierung in die gewählte Zielsprache bis hin zum automatisierten ***Deployment*** in die gängigsten Applikationsserver. Folgende Programmiersprachen werden unterstützt:

- Java
- C++
- C#
- Corba IDL
- Visual Basic 6
- Visual Basic.Net

Together versteht sich als eine Plattform in dem Sinne, dass es als gemeinsame Kommunikationsplattform verschiedener Werkzeuge fungiert. Die Modellierung kommuniziert mit einer integrierten Entwicklungsumgebung (IDE) und umgekehrt.
Das Ziel bei der Entwicklung von Together ist es, Redundanz zwischen Modell und endgültiger Implementierung zu vermeiden. In Together sollen jederzeit Sourcecode und Diagramme synchron sein. Diagramminformationen sollen zur Laufzeit in Code übersetzt werden und umgekehrt. Dies soll ein tatsächliches Reverse Engineering bestehender Software garantieren.

3.2 Together Open API

3.2.1 Kategorisierung

Durch die Open API kann Together erweitert werden. Sie basiert auf Java und ermöglicht es dem Nutzer eigene Module zu entwickeln.

3.2.2 Überblick

Die Open API besteht aus sieben Paketen:

- ***Baseexpert***
- ***Vfs***
- ***Rwi***
- ***Util***
- ***Sci***
- ***Ide***
- ***Vcs***

3.2.3 Relevante Funktionen für die Mustersuche

Für die Mustersuche haben folgende Pakete eine Bedeutung:

- Das Paket ***com.togethersoft.openapi.ide*** und seine Unterpakete.
- Das Paket ***com.togethersoft.openapi.rwi*** und seine Unterpakete.
- Das Paket ***com.togethersoft.openapi.sci*** und seine Unterpakete.

Im folgendem erfolgt eine kurze Beschreibung der Komponenten der API:

- **IDE**

Die **IDE** generiert gewünschte Ausgänge in Abhängigkeit von den Informationen, die in einem Modell enthalten sind. Man kann auf verschiedenste sichtbare Elementen zugreifen, man kann auf die Diagramme zugreifen und sich einen Zeiger auf sie holen. Es ist ein ***read only Interface***, also kann man Informationen aus dem Modell extrahieren, aber man kann das Modell nicht ändern. Ihre Funktionalitäten beziehen sich auf die Repräsentation des Modells in der Together **IDE** und die Interaktion mit dem Nutzer.

- **RWI**

Das ***Read Write Interface*** ermöglicht es dem Nutzer, tief in die Modellarchitektur hineinzugehen. Man kann sowohl Informationen aus dem Modell extrahieren als auch Informationen in das Modell schreiben. Also kann man auf Elemente, die in Diagrammen enthalten sind, zugreifen und sie verändern. Mit diesem ***Interface*** kann man die Fähigkeiten von Together erweitern. Man kann eine Aufzählung der Elemente eines Modells bekommen. Es werden gleichartige Objekte als eine Liste zurückgegeben und man kann dann diese Liste durchlaufen und die einzelnen Elemente ansprechen. Diese Elemente können z.B: Diagramme (Klassendiagramm, Sequenzdiagramm) oder Links sein.

- **SCI**

Durch das ***Source Code Interface*** kann man den ***Source Code*** ansprechen. Das heißt, man kann den kompletten Code eines geöffneten Projektes durchlaufen und jede Anweisung ansprechen und ändern. Ein SCI-Modell kann Objekte enthalten, die in unterschiedlichen Programmiersprachen geschrieben sind. Dieses Interface ist sehr mächtig und umfangreicher als die anderen.

3.2.4 Zwischenbewertung

Für einen Nutzer, der zum ersten Mal mit Together arbeitet, ist die in Together integrierte API-Dokumentation nicht detailliert genug. Um die Entwicklung des Moduls zur Mustersuche durchführen zu können, muss man die einzelnen ***Interfaces,*** die man benötigt, benutzen können. Diese Aufgabe, herauszufinden, wann und wie welches ***Interface*** benutzt werden kann, ist sehr zeitaufwendig.

3.3 Together-Erweiterung mit Modulen

Ein Modul in Together ist eine Java-Klasse, die das Interface ***IdeScript*** oder das Interface ***IdeStartup*** implementiert. ***Startup*** Module können nicht manuell ausgeführt werden. Sie werden automatisch von Together ausgeführt. Uns interessieren hier nur Module, die das Interface ***IdeScript*** implementieren. Sie können jederzeit ausgeführt werden indem die ***run*** Methode aufgerufen wird. Bei der Implementierung dieses Interface muss man u.A. die Methode ***run*** (***IdeContext***) implementieren. Bei der Ausführung wird der Methode ***run*** als Parameter eine Instanz von ***IdeContext*** übergeben. Diese Instanz enthält Informationen über alle markierten Elemente in dem Moment, wenn das Modul ausgeführt wird. Hier brauchen wir aber diese Objekte, nicht da alle Kassen systematisch nach Mustern durchsucht werden sollten.

Die Entwicklung von Modulen in Together kann zwei Formen haben:

- **Compilierte Module**: sie sind in Java geschrieben und mit Hilfe eines Java Compiliers compliliert. Sie benutzen die selbe Java virtuelle Maschine wie Together zur Laufzeit.
- **Module die in Tcl bzw Jpython geschrieben sind**: Sie werden von geeigneten Subsystemen von Together zur Laufzeit interpretiert.

Das Modul zur Mustersuche soll ein compiliertes Modul sein.

Je nach Sprache, Zeit des Aufrufs, und Art des ***Deployment*** kann man Module in vier Gruppen kategorisieren:

- **Nutzer-Module**: erscheinen im Modulsbaum und können dadurch ausgeführt werden, dass auf ***run*** geklickt wird.
- **Nutzer-Module mit Vorinitialisierung**: sie sind wie Nutzer-Module aber vor dem Modul wird die Initialisierungsmethode ausgeführt.
- ***Startup*** Module: Sie erscheinen nicht in dem Modulsbaum und werden automatisch mit Together gestartet.
- **Aktivierte bzw deaktivierte Module**: es ist eine Kombination von Nutzer und ***Startup*** Modulen. Also kann je nach Einstellung das Modul entweder mit Together gestartet werden oder vom Nutzer. Unser Modul soll vom Nutzer Typ sein.

Module werden in dem Verzeichnis **THG\modules\com\together\modules** abgelegt. Bei der Darstellung von Modulen in dem Modules ***Tab*** wird unterschieden zwischen der Source Code des Moduls, dem compilierten Modul und dem Tcl Script. Letzteres interessiert uns nicht.

Bei der Entwicklung von Modulen sind Namenkonventionen zu respektieren. Beispielsweise sollen Methodennamen aus zusammengesetzten Wörtern bestehen, wobei das erste Wort ein aktives Verb sein soll, das zweite ein Wort das mit einem großen Buchstabe anfängt, z.B. printCurrentDiagram() .

4 Das Modul zur automatischen Suche nach Entwurfsmustern

4.1 Kurze Beschreibung des Moduls

Das RWI-Interface, wie schon oben erwähnt, ermöglicht Zugriff und Verarbeitung von Diagrammen. Grob arbeitet das Modul zur Mustersuche wie folgt:

Es wird geprüft, ob ein offenes Projekt vorhanden ist, wenn nicht, kann das Modul nicht arbeiten und beendet sich.

Dann wird ein Rwi-modell Objekt geholt
RwiModel model = RwiModelAccess.getModel();

Sowie alle Packete
RwiPackageEnumeration roots=model.rootPackages(RwiProperty.MODEL);

Danach werden all Packete verarbeitet
while (roots.hasMoreElements()){

RwiPackage nextPackage = roots.nextRwiPackage();
packageProcessing(nextPackage);
}

Hier erfolgt die Verarbeitung dadurch, dass jedesmal ein Paket geholt und mit der Methode **packageProcessing()** verarbeitet wird.

Ein RwiPackage Element enthält Modelldaten wie zum Beispiel Knoten (Klassen oder Interface), Members, Diagramme und andere Pakete in einer Verzeichnisstruktur.

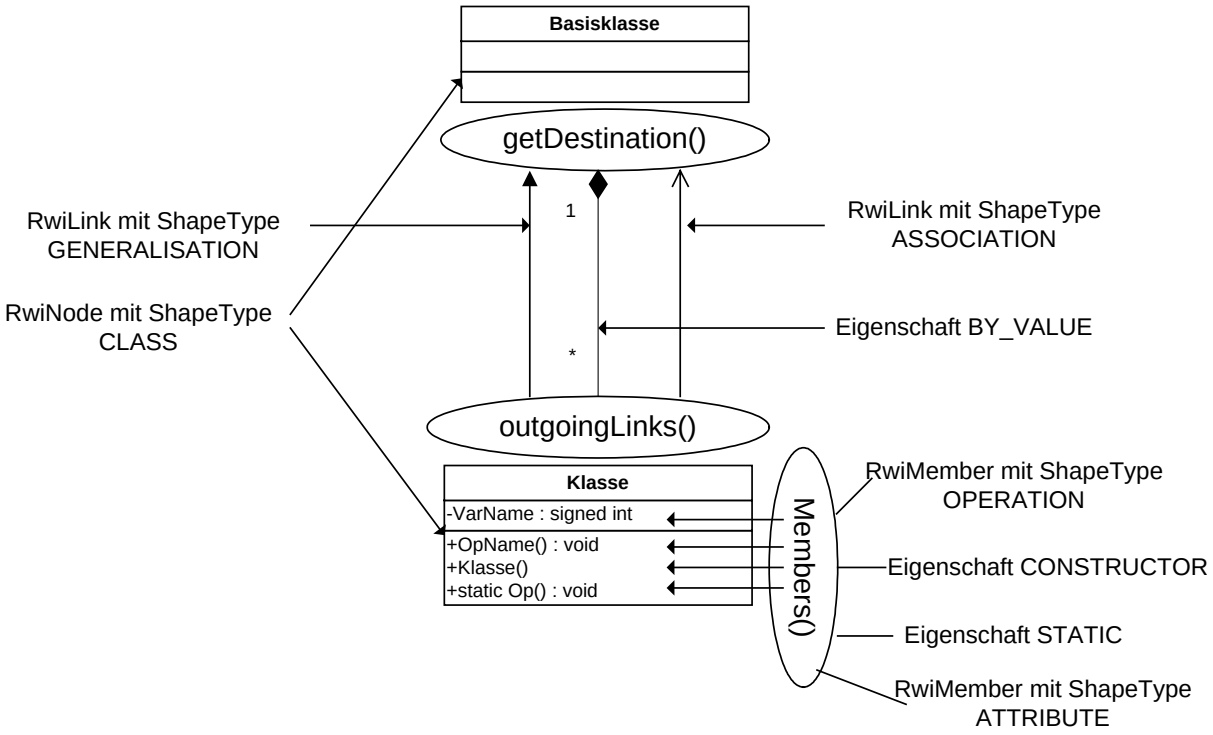

Abbildung 4: OpenApi Funktionen zum Durchsuchen einer Klasse

4.2 Algorithmen mit konkreter Umsetzung

In diesem Abschnitt werden die implementierten Algorithmen sowie die für die Umsetzung geschriebenen Funktionen beschrieben.

4.2.1 Beschreibung der Attribute der Klasse MusterVero

Variablenname	Typ	Bedeutung
_projectLanguageIsJava	Boolen	Enthält TRUE falls das offene Projekt ein Java-Projekt ist
_projectLanguageIsCPP	Boolen	Enthält TRUE falls das offene Projekt ein CPP-Projekt ist
_composites	Vector	Enthält die Namen aller Klassen die Kompositum sind

_visitors- Hashtable

Key (Klasse als RwiNode)	Value (Vector von RwiNodes der besuchten Klassen)
RwiNode von Klasse X	[RwiNode besuchte KlasseX1, RwiNode besuchte KlasseX2…]
RwiNode von Klasse Y	[RwiNode besuchte KlasseY1, RwiNode besuchte KlasseY2…]

_aggregatedClasses- Hashtable

Key (Klasse als RwiNode)	Value (Vector von RwiNodes die Ziel einer 1-zu-N-Aggregation sind)
RwiNode von Klasse X	[RwiNode KlasseX1, RwiNode KlasseX2…]
RwiNode von Klasse Y	[RwiNode KlasseY1, RwiNode KlasseY2…]

Neben diesen Attributen werden auch andere benutzt, die aus [Hel03] stammen. Es sind folgende:

Variablenname	Typ	Bedeutung
_classes	Vector	Instanzen aller Klassen
_attributes	Hashtable	Kombination von Klassenname(Key) und Vektor der Attribute
_operations	Hashtable	Kombination von Klassenname(Key) und Vektor der Operationen
_classNameToClass	Hashtable	Kombination von Klassenname(Key) und Klasseninstanz
_implementedClasses	Hashtable	Kombination von Klasseninstanz(Key) und Vektor der implementierten Klassen

.

_subClasses-Hashtable

Key (Klassenname als String)	Value (Vector von Strings der Subklassennamen)
KlasseX	[„SubklasseX1“, „SubklasseX2“, „SubklasseX3“]
KlasseY	[„SubklasseY1“, „SubklasseY2“, „SubklasseY3“]

_superClasses-Hashtable

Key (Klassenname als String)	Value (Vector von RwiNodes der Superklassen)
KlassennameX	[RwiNode SuperklasseX1, RwiNode SuperklasseX2…]
KlassennameY	[RwiNode SuperklasseY1, RwiNode SuperklasseY2…]

4.2.2 Allgemeine Methodenaufrufe des Moduls

Das Modul enthält eine Menge an Methoden, die unabhängig von dem gesuchten Muster aufgerufen werden. Diese Methoden dienen der allgemeinen Manipulierung von Modell- bzw. Codeelementen.

4.2.2.1 Die Methode Run(IdeContext context)

Die Methode ***Run*** ist die erste Methode, die beim Starten des Moduls aufgerufen wird. Sie wird mit einem Parameter vom Typ IdeContext aufgerufen. Hier werden alle Initialisierungen vorgenommen und die anderen Funktionen werden von dieser Funktion aus gestartet. Hier ist die Vorgehensweise ähnlich wie in [Hel03] und es wird zusätzlich die Programmiersprache des offenen Projektes eingestellt.
Zuerst wird geschaut, ob ein Projekt aktiv ist und falls ja, wird auf das Modell zugegriffen, alle Pakete aufgelistet und der ***MessagePane*** aktiviert für die Ausgabe. Danach werden die Pakete der Reihe nach durchgegangen, verschiedene Funktionen extrahiert und gespeichert. Danach werden alle Klassen weiß gefärbt, nochmals durchgelaufen und auf die Muster durchsucht.
Zum Schluss werden die gefundenen Muster ausgegeben.

4.2.2.2 Die Methode classContainsOperation ()

Rückgabetyp	Boolean
Parameter	String className, String operationName

Diese Methode testet, ob die übergebene Klasse **className** eine direkte Methode **operationName** enthält. Also eine Methode die nicht über Superklasse geerbt wurde.
Sie liefert true, falls die Klasse **className** eine Methode **operationName** hat, und false, falls die Klasse **className** eine Methode **operationName** NICHT hat

4.2.2.3 Die Methode operationHasClassAsParameter ()

Rückgabetyp	Vector
Parameter	RwiMember operation

Diese Methode liefert einen Vektor mit allen Klassen, die Parameter-Typ der Methode **operation** sind. Zu Beginn werden alle Parameter der Methode **operation** aufgelistet. Dann werden sie durchlaufen und für jeden Parameter die Eigenschaften NAME und TYPE extrahiert. Für C++ Projekte ist eine besondere Behandlung des Parametertyps, der aus dem Quellcode gelesen und in einem String gespeichert wurden, notwendig. Alle potentiellen const-Modifier vor dem Parameter-Typ müssen entfernt werden, genauso wie alle potentiellen Zeiger-* und Referenz-& vor dem Parameter-Typ. Der so gesäuberte Parameter wird dann mit dem Aufruf
RwiNode clas = this.getClassForName(paramType) benutzt, um eine Klasse mit dem gleichen Namen wie der Paramer-Typ zu finden.

4.2.2.4 Die Methode isCallForMethodInSuperClass()

Rückgabetyp	Boolean
Parameter	String subClassName, SciStatement statement

Die Methode isCallForMethodInSuperClass testet, ob das Statement **statement** einen Methodenaufruf auf eine Superklasse darstellt. Der Parameter **subClassName** ist der Name der SubKlasse und der Param **statement** die zu untersuchende Code-Zeile
Diese Methode liefert true, wenn es sich um einen Aufruf einer SuperKlassen-Methode handelt und **false,** wenn KEIN Aufruf einer SuperKlassen-Methode vorhanden ist.

4.2.2.5 Die Methode isCallForMethodInLocalClass()

Rückgabetyp	Boolean
Parameter	String subClassName, SciStatement statement

Die Methode isCallForMethodInLocalClass testet, ob das Statement **statement** einen lokalen Methodenaufruf darstellt. Der Parameter **subClassName** ist der Name der SubKlasse und der Parameter **statement** die zu untersuchende Code-Zeile.
Diese Methode liefert **true**, wenn es sich um einen Methodenufruf handelt und **false**, wenn KEIN lokaler Methodenaufruf vorhanden ist.

4.2.2.6 Die Methode getClassForName()

Rückgabetyp	RwiNode
Parameter	String className

Diese Methode liefert zu einem übergebenen Klassennamen **className** das dazugehörige Klassenobjekt. Dabei wird der Name **className** mit jedem Klassenobjekt der Hashtable **_classes** verglichen.

4.2.2.7 Die Methode getSubClasses()

Rückgabetyp	Vector
Parameter	RwiNode clas

Die Methode liefert alle Subklassen vom Klassenobjekt **clas**. Dabei wird zuerst der Name des übergebenen Klassenobjektes ermittelt. Dann werden aus der Hashtable _subClasses die Unterklassen, die zu diesem Klassennamen gehören, extrahiert.

4.2.2.8 Die Methode getSuperClassesFromModel()

Rückgabetyp	Vector
Parameter	RwiNode clas

Alle Superklassen zu des übergebenen Klassenobjektes **clas** werden aus dem Modell ermittelt und als Vector von RwiNodes zurückgeliefert.

4.2.2.9 Die Methode getSuperClasses()

Rückgabetyp	Vector
Parameter	RwiNode clas

Diese Methode liefert alle Superklassen vom übergebenen Klassenobjekt **clas**
Sie werden aus der Hashtable _**superClasses** geholt.

4.2.2.10 Die Methode getImplementedClassesFromModel()

Rückgabetyp	Vector
Parameter	RwiNode clas

Alle Klassen, die zu der übergebenen mit einem Implementierungslink verbunden sind werden durch diese Methode aus dem Modell ermittelt und als Vector zurückgeliefert.

4.2.2.11 Die Methode getImplementedClasses()

Rückgabetyp	Vector
Parameter	RwiNode clas

Die Methode liefert alle implementierten Interfaceklassen vom Klassenobjekt **clas**. Die Interfaceklassen werden aus der Hashtable _**implementedClasses** geholt und in einem Vektor zurückgeliefert.

4.2.2.12 Andere allgemeine Funktionsaufrufe

Die folgenden Funktionsaufrufe stammen aus [Hel03] und dort findet man eine nähere Beschreibung davon.

- Die Funktion **faerbeKlasseInDiagramm (String,String,String,String)** färbt die übergebene Klasse in der übergebenen RGB-Farbe

- Die Funktion **print(String text)** erzeugt Ausschrift in dem ***MessagePane***

- Die Funktion ***packageProcessing(RwiPackage rwiPackage):*** das übergebene Paket wird mit der Funktion ***doActionUponPackage(rwiPackage)*** verarbeitet, seine Unterpakete aufgelistet und rekursiv verarbeitet.

- In der Funktion **doActionUponPackage(RwiPackage rwiPackage)** werden die einzelnen Pakete verarbeitet. Es werden alle Knoten bestimmt und die werden in der Funktion **doActionUponNode** weiterverarbeitet

- Die Funktion **doActionUponNode(RwiNode node)** prüft, ob es sich bei dem Knoten um eine Klasse handelt. Wenn ja werden, sie in der Funktion **doActionUponClass** weiterverarbeitet.

- In der Funktion **doActionUponClass(RwiNode clas)** werden alle relevanten Klasseninformationen extrahiert und die Hashtables gefüllt.

- Die Funktion **addSubClass(String, String)** fügt einen String an den schon vorhandenen Vektor an und ändert dabei den Schlüssel nicht.

- Die Funktion **appendVector(Vector, Vector)** fügt den einen Vector an den anderen Vector an.

- Die Funktion **getNamesFromVector(Vector)** extrahiert alle Elementnamen aus dem übergebenen Array und liefert sie als Vector zurück

4.2.3 Spezifische Funktionen für das Muster Kompositum

Die Struktur des Musters Kompositum ist in ***Abbildung 2: Muster*** *Kompositum* zu sehen und der dazu gehörige Suchalgorithmus stammt aus [Naum 01]:

```
für jede Klasse i (Kompositum)
do
  besitzt Klasse i eine 1-zu-n-Aggregation zu einer Oberklasse (Komponente)? ja: weiter
  besitzt Klasse i Unterklassen?
     nein: Muster gefunden
     ja: für jede Unterklasse j (spezialisiertes Kompositum) von Klasse i
    do
        für jede Methode k der Klasse j
        do
          ruft Methode k eine Methode der Klasse i auf, und folgt dem ein lokaler Methodenaufruf?
             nein: weiter
             ja: Abbruch
        od
    od
Muster gefunden
od
```

4.2.3.1 Die Methode showComposites ()

Hier wird der Vektor mit den Namen der Komposita durchlaufen und für jeden Namen wird die Methode getClassForName() ausgeführt und das dazugehörige Klassenobjekt zurückgeliefert. Nun wird die Kompositum-Klasse in Messagepane angezeigt und im Diagramm blau gefärbt. Dasselbe geschieht für alle Unterklassen des Kompositum, falls es welche gibt.
Das Aggregat wird auch angezeigt sowie seine Unterklassen.

4.2.3.2 Die Methode void evaluateComposite()

Rückgabetyp	
Parameter	RwiNode clas

Methode zur Überprüfung, ob Klasse clas ein Kompositum ist. Wenn ja, wird diese Klasse in den Vektor _composites aufgenommen. Dafür werden die Methoden compositeTestSuperClasses(RwiNode clas) und compositeTestSubClasses(RwiNode clas) aufgerufen.

4.2.3.3 Die Methode compositeTestSuperClasses()

Rückgabetyp	Boolean
Parameter	RwiNode clas

Die Methode testet, ob das Klassenobjekt **clas** eine 1-zu-N-Aggregation zu einer (Ober)Klasse (Komponente)besitzt
Sie liefert **true,** wenn von Klasse **clas** mind. eine 1-zu-N-Aggregation ausgeht.
Sie liefert **false,** wenn von Klasse **clas** KEINE 1-zu-N-Aggregation ausgeht.

4.2.3.4 Die Methode compositeTestSubClasses()

Rückgabetyp	Boolean
Parameter	RwiNode clas

Diese Methode testet folgende Regel für das Auffinden von Kompositum-Muster
ja: für jede Unterklasse j (spezialisiertes Kompositum) von Klasse
i do
für jede Methode k der Klasse j do
ruft Methode k eine Methode der Klasse j auf, und folgt dem
ein lokaler Methodenaufruf?
nein: weiter, ja: Abbruch
od
od
Muster gefunden

4.2.3.5 Die Methode getAggregation1toNFromModel()

Rückgabetyp	Vector
Parameter	RwiNode clas

Aus dem Modell werden Klassen ermittelt, die zu der übergebenen **clas** einen Aggregationslink mit der supplierCardinality "1..*" (1-zu-N) haben und liefert diese als Vector von RwiNodes zurück.

4.2.3.6 Die Methode recipientIsSuper ()

Rückgabetyp	Boolean
Parameter	String recipient, String subClassName

In dieser Methode wird der Unterschied der Methodenaufrufe der Super-Klasse in den Programmiersprachen JAVA und C++ dargestellt. In JAVA wird die Superklasse explizit als Empfänger der Methode angegeben.
recipientIsSuper testet, ob **recipient** eine Oberklasse von **subClassName** darstellt, d.h wenn in Java recipient = "super" ist oder in C++ eine Oberklasse explizit angegeben ist. Sie liefert **true**, wenn

recipient eine Oberklasse von **subClassName** und **false** wenn **recipient** KEINE Oberklasse von **subClassName** ist.

4.2.3.7 Die Methode recipientIsThis ()

Rückgabetyp	Boolean
Parameter	String recipient, String subClassName

recipientIsThis testet, ob **recipient** die eigene Klasse darstellt, d.h wenn in Java **recipient** = "this" ist oder in C++ **recipient** = "this" oder **recipient** = "*this" ist. Sie liefert **true**, wenn **recipient** die eigene Klasse darstellt und **false**, wenn **recipient** die eigene Klasse NICHT darstellt.

4.2.4 Spezifische Funktionen für das Muster Besucher

Die Struktur des Musters Besucher ist in ***Abbildung 3: Muster*** *Besucher* zu sehen und der dazu gehörige Algorithmus stammt aus [Nau01]

für jede Klasse i (Besucher)
do
 für jede Methode j der Klasse i
 do
  besitzt Methode j andere Klasse k (Element) als Parameter?
  ja: für jede Methode m der Klasse k
  do
   besitzt Methode m als Parameter die Klasse i?
   ja: gibt es in dieser Methode einen Aufruf der Methode j der Klasse i?
    ja: übergibt sich bei diesem Aufruf die Klasse k selbst?
      ja: Muster gefunden
  od
 od
od

4.2.4.1 Die Methode showVisitors ()

Bei der Ausgabe aller Visitors werden sämtliche Elemente der Variable _visitors durchlaufen. Aus den Klassenobjekte werden mit Hilfe von der Anweisung ***String visitedClassName = visitedClass.getProperty(RwiProperty.NAME*** die Klassennamen gewonnen. Diese werden im Messagepane angezeigt und im Diagramm blau gefärbt. Ähnlicherweise werden alle Unterklassen der besuchten Klasse angezeigt.

4.2.4.2 Die Methode void evaluateVisitor()

Rückgabetyp	
Parameter	RwiNode clasI

Diese Methode untersucht die übergebene Klasse auf Eigenschaften des Visitors.

Zunächst wird aus dem übergebenen Klassenobjekt den Klassennamen ermittelt. Dann werden alle Methoden, die zu dieser Klasse gehören und die in der Hashtabelle _**operations** gespeichert sind, geholt und in den Vektor **operations** abgelegt. Falls dieser Vektor nicht leer ist, werden die enthaltenen Methoden durchlaufen und mit der Anweisung ***Vector classesK = operationHasClassAsParameter (operationJ)*** werden in dem Vektor classesK alle Klassen gespeichert die die Methode J als Parameter hat. Nun werden aus diesen Klassenobjekten Klassennamen ermittelt und mit der Anweisung ***Vector classKOperations = (Vector)_operations.get(classKName*** alle Methoden der Klasse K aus der Hashtabelle _operations geholt und im Vektor ***classKOperations*** gespeichert. Dieser Vektor wird nun durchlaufen und für jede dieser Methoden werden die Parametertypen, welche eine Klasse aus dem Modell darstellen, ermittelt. Dann wird überprüft, ob die Klasse **i** des Algorithmus in diesem Vektor enthalten ist. Mit dem Aufruf ***if (visitorTest (operationM, operationJ))…*** werden die übrigen Bedingungen des Visitors überprüft, nämlich ob es in dieser Methode **M** einen Aufruf der Methode **j** der Klasse **i** gibt und falls ja, ob sich bei diesem Aufruf die Klasse **k** selbst übergibt.

4.2.4.3 Die Methode visitorTest ()

Rückgabetyp	Boolean
Parameter	RwiMember operationM, RwiMember operationJ

Methode visitorTest übernimmt folgenden Teil des Algorhitmus:

> ja: gibt es in dieser Methode (operationM) einen Aufruf der Methode j (operationJ) der Klasse i?
> ja: übergibt sich bei diesem Aufruf die Klasse k selbst?
> ja: Muster gefunden

Der Parameter **operationM** ist die Methode, die nach einem Aufruf auf Methode **operationJ** durchsucht wird und **operationJ** ist die Methode, nach den in Methode **operationM** gesucht wird
Die Methode visitorTest liefert true, wenn es einen Aufruf auf Methode **operationJ** in Methode **operationM** gibt, bei der "this" übergeben wird. Sie liefert false, es keinen Aufruf auf Methode **operationJ** in Methode **operationM** gibt, bei der "this" übergeben wird

4.3 Praktischer Einsatz

4.3.1 Eigenes Beispiel Projekt1

Diese Projekt wurde für Testzwecke angelegt und wurde in dem selben Packet wie das Modul gespeichert, um ein ständiges Wechseln zwischen den Projekten zu vermeiden, da in Together nicht gleichzeitig zwei Projekte geöffnet werden können. Die Abbildung zeigt das Klassendiagramm dieses Projektes.

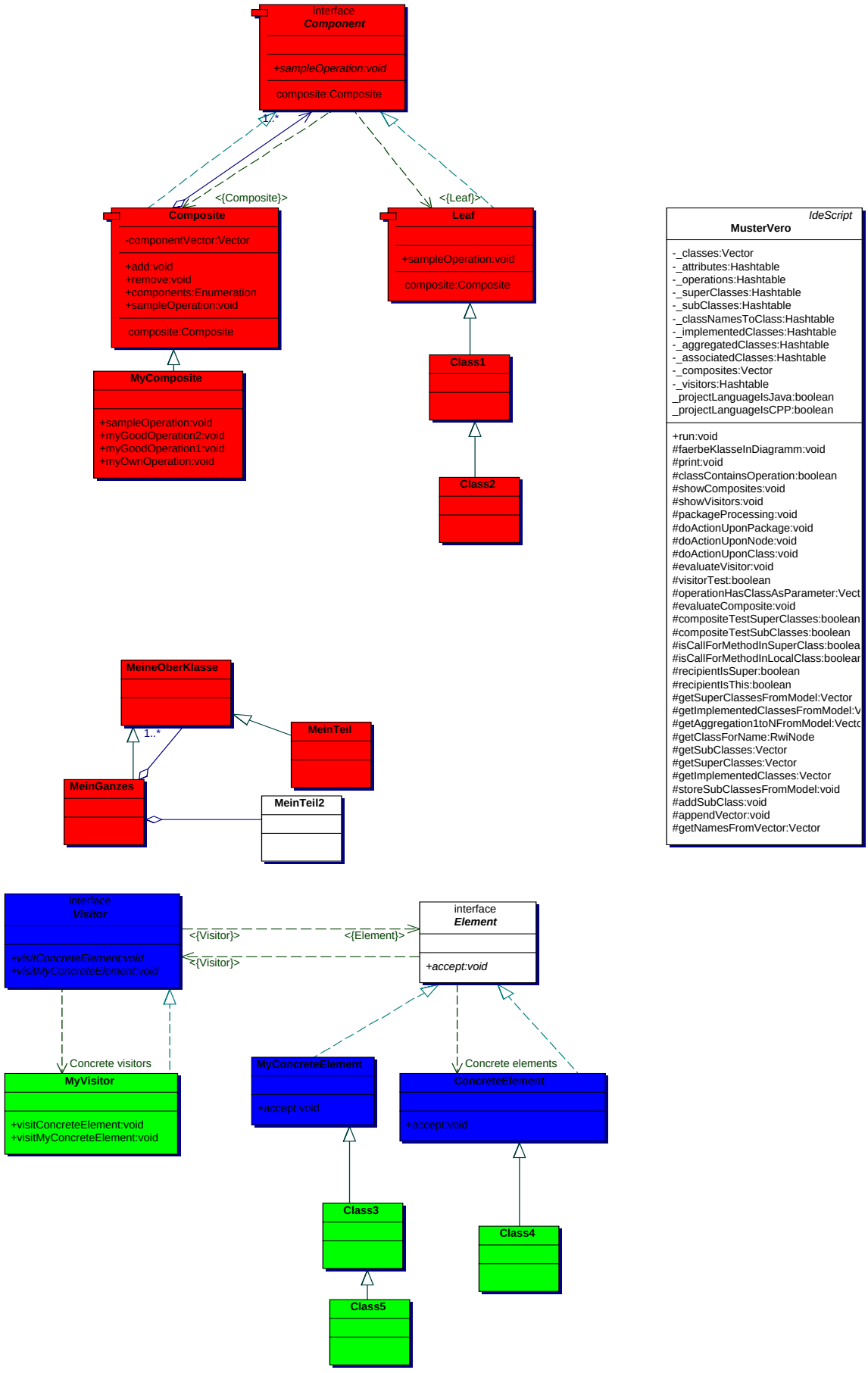

Abbildung 5: Klassendiagramm des Beispiel Projekts 1

4.3.2 Eigenes Beispiel Projekt2

Das Beispiel Projekt 2 ist ein in [Gamma+96] beschriebenes Beispiel das mit Together umgesetzt wurde. Es dient auch wie das Beispiel Projekt1 dem Testen des Moduls. Together erzeugte das Muster Kompositum erstaunlicherweise mit einer Aggregation der Kardinalität 0..* . Es war also in diesem Beisbiel Projekt notwendig die Kardinalität auf 1..* zu ändern. Danach wurden die Muster Komposite und Visitor in diesem richtig und schnell gefunden.

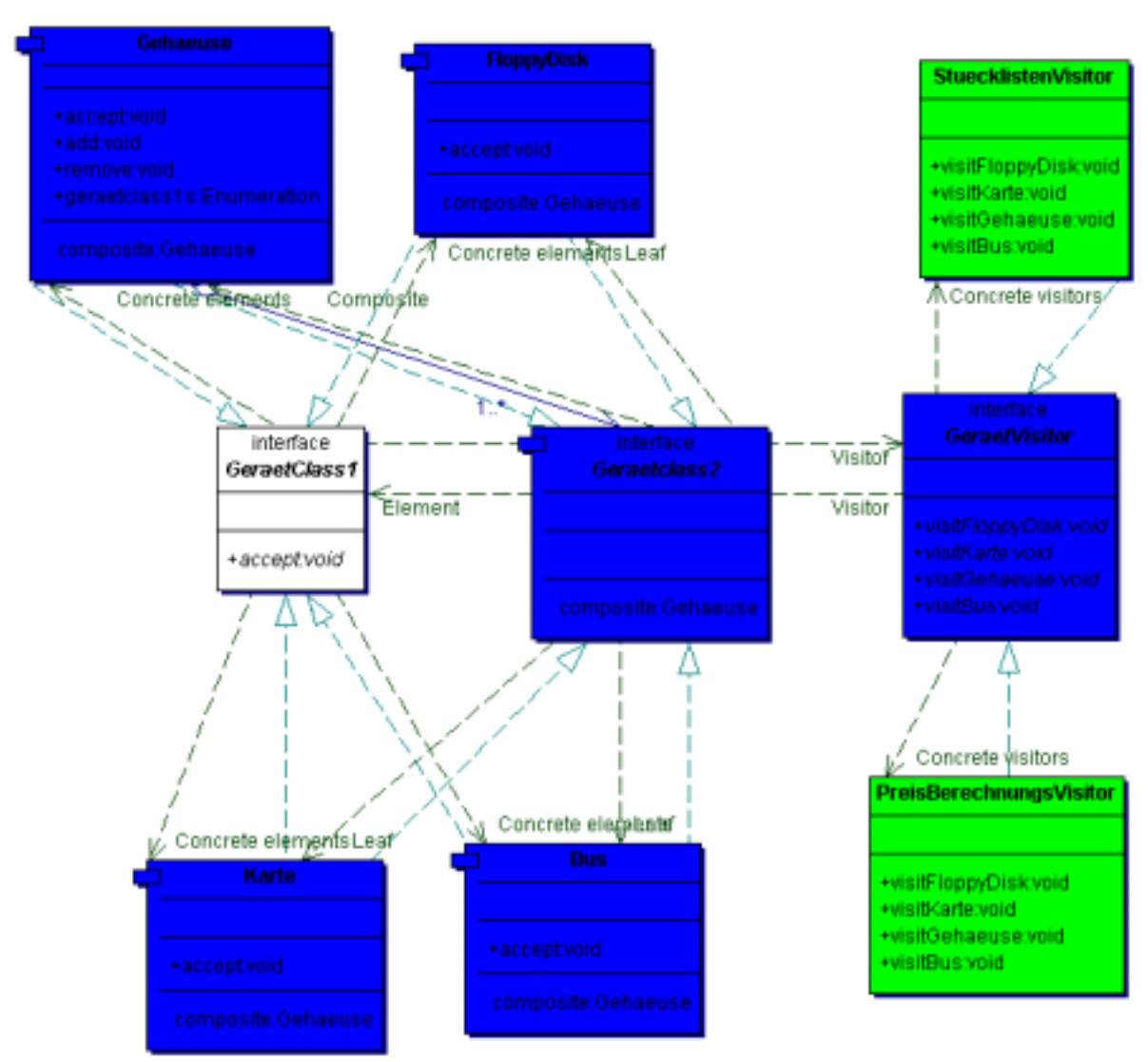

Abbildung 6: Klassendiagramm des Beispiel Projekts 2

4.3.3 Vorgegebene Projekte

Von den folgenden Projekte war der Source Code vorgegeben. Nach Reverse Engineering mit Together wurde das erstellte Modul darauf ausgeführt.

Projekt	Sprache	Gefundene Muster
Megamek	Java	3 Visitor- Muster gefunden
VDR	C++	Keine Muster gefunden
Texmacs	C++	Keine Muster gefunden durch Fehler bei der Ausführung
Art of Illusion	Java	Keine Muster gefunden
My SQL 4013	C++	Keine Muster gefunden
My SQL ++	C++	Keine Muster gefunden
CPP-Patterns	C++	1 Visitor- Muster gefunden

Nun einige Bemerkungen über die Ausführung des Moduls auf diesen Projekten

4.3.3.1 Megamek

In Megamek wurden folgende Visitors-Klassen und besuchte Klassen gefunden:

- Klasse: Mounted, besuchte Klasse: Mounted

Hier ist ein Programmabschnitt von Megamek, in dem das Muster gefunden wurde.

```
public class Mounted implements Serializable{   //Klasse i, k
...
   public void setLinked(Mounted linked) {   //Methode j, Klasse k = Mounted
      this.linked = linked;
      linked.setLinkedBy(this);    // Aufruf von Methode j der Klasse i mit k als Parameter
   }
...
}
```

Der Algorithmus zur Erkennung des Musters ***Visitor*** besagt nicht, dass Klasse i und k nicht dieselbe sein dürfen. Das ist aber in diesem Fall so. Deshalb hat die Mustererkennung die Klasse ***Mounted*** als ein Visitor-Pattern erkannt.

- Klasse: Player, besuchte Klasse: Game

```
public final class Player extends TurnOrdered  //Klasse i
{
...
   public void setGame(Game game) {  //Methode j, Klasse k = Game
      this.game = game;
   }
...
}
```

```
public class Game implements Serializable        // Klasse k
{
...
   public void addPlayer(int id, Player player) {   // Methode m
      player.setGame(this);        // Aufruf von Methode j der Klasse i mit k als Parameter
      players.addElement(player);
      playerIds.put(new Integer(id), player);
   }

   public void setPlayer(int id, Player player) {    // Methode m
      final Player oldPlayer = getPlayer(id);
      player.setGame(this);        // Aufruf von Methode j der Klasse i mit k als Parameter
      players.setElementAt(player, players.indexOf(oldPlayer));
      playerIds.put(new Integer(id), player);
   }
....
}
```

- Klasse: Entity

Unterklasse: BattleArmor
Unterklasse: BipedMech
Unterklasse: Infantry
Unterklasse: Mech
Unterklasse: QuadMech
Unterklasse: Tank
besuchte Klasse: Game

```
public abstract class Entity        //Klasse i
   implements Serializable, Transporter, Targetable
{
...
   public void setGame(Game game) {   //Methode j, Klasse k = Game
      this.game = game;
   }
...
}
```

```
public class Game implements Serializable        // Klasse k
{
...
   public void addEntity(int id, Entity entity) {       // Methode m
      entity.setGame(this);                    // Aufruf von Methode j der Klasse i mit k als Parameter
      entities.addElement(entity);
      entityIds.put(new Integer(id), entity);
   }

   public void setEntity(int id, Entity entity) {       // Methode m
      final Entity oldEntity = getEntity(id);
      entity.setGame(this);          // Aufruf von Methode j der Klasse i mit k als Parameter
      if (oldEntity == null) {
         entities.addElement(entity);
      } else {
         entities.setElementAt(entity, entities.indexOf(oldEntity));
      }
      entityIds.put(new Integer(id), entity);
   }

...
}
```

4.3.3.2 VDR

Bei dem Aufruf der Methode **evaluateVisitor** für die Klassen cTimer, cThread, cListObject wird die folgende Bedingung erfüllt:

besitzt Methode m als Parameter die Klasse i

Für die Klasse cLineGame scheitert die Visitor Bedingung dadurch, dass die Klassen K keine Methoden besitzen.

4.3.3.3 Texmacs

Reverse Engineering für dieses Projekt war sehr aufwendig da Together nur die oberste Stufe der Header-Dateien bindete. Alle Header-Dateien, die von Header-Dateien inkludiert sind, wurden nicht automatisch eingebunden. Alle Pfade mussten also manuell gesetzt werden. Das bedeutete ein ständiges Schalten zwischen dem Verzeichnis, das die Source-dateien enthält, und Together. Da diese Header-Dateien in Falle eines großen Projekts wie Texmacs nicht immer auf dem ersten Blick zu finden sind, musste jedes Mal für jede Datei eine Suche gestartet werden. Wodurch sich der Aufwand des Reverse Engineering zusätzlich erhöhte.
Wenn alle Header Dateien eingebunden wurden, musste auch dafür gesorgt werden, das alle CPP-Dateien in dem selben Verzeichnis liegen wie die dazugehörige Header-Datei.
Die Ausführung von Megamek lief nur teilweise gut und Rechner stürzte nach einer Weile immer ab.

4.3.3.4 Art of Illusion

Das Projekt Art of Illusion erfüllte laut Modul MusterVero keine der in den Algorithmen für das Muster Kompositum und Muster Besucher genannten Bedingungen.

4.3.3.5 My SQL 4013

Bei der Ausführung dieses Projektes erfüllte keine Klassen die Bedingungen für das Muster Besucher. Allerdings wurden bei der Überprüfung auf Kompositum- Eigenschaften viele Aggregationen gefunden, die aber die Kardinalität 1..* nicht hatten. Es ist aber nicht auszuschliessen dass der Code von My SQL 4013 Aggregationen mit Kardinalität 1..* enthält, die aber durch Reverse Engineering nicht erkannt wurden. Unten aufgelistet wird die Liste einiger betroffenen Klassen. Sie stammt aus der Ausgabe in ***Messagepane***.

evaluateComposite für Klasse st_io_cache_share
supplierCardinality nicht 1..*, ignoriere Aggr. zu pthread_cond_t
evaluateComposite für Klasse st_tree
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_mem_root
evaluateComposite für Klasse st_mi_check_param
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_io_cache
evaluateComposite für Klasse st_sort_info
supplierCardinality nicht 1..*, ignoriere Aggr. zu pthread_cond_t
evaluateComposite für Klasse st_mi_sort_param
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_io_cache
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_dynamic_array
evaluateComposite für Klasse st_myrg_info
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_list
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_queue
evaluateComposite für Klasse st_mysql_data
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_mem_root
evaluateComposite für Klasse st_mysql
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_net
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_mem_root
evaluateComposite für Klasse st_mysql_res
supplierCardinality nicht 1..*, ignoriere Aggr. zu st_mem_root
evaluateComposite für Klasse st_vio

4.3.3.6 MySQL++

Es wurden keine Muster Besucher oder Kompositum in diesem Projekt gefunden. Die Bedingungen der Suchalgorithmen wurden auch nicht teilweise ergefüllt.

4.3.3.7 CPP-Patterns

Hier wurden bei der Suche nach dem Kompositum-Muster auch einige Aggregationen gefunden:

evaluateComposite für Klasse BuilderOne

supplierCardinality nicht 1..*, ignoriere Aggr. zu OneEnded
evaluateComposite für Klasse BuilderTwo
supplierCardinality nicht 1..*, ignoriere Aggr. zu TwoEnded
…
evaluateComposite für Klasse Person
supplierCardinality nicht 1..*, ignoriere Aggr. zu Command
Die Suche nach dem Muster Besucher ergab folgende Ausgabe in ***Messagepane***:

Folgende Visitors wurden gefunden:

Klasse: Visitor
 Unterklasse: CallV
 Unterklasse: CountV
 besuchte Klasse: Red
 besuchte Klasse: Blu

Für die Klassen Person, Command, Red, Blu, ist immer jeweils die erste Bedingung für den Visitor-Muster erfüllt und die anderen nicht.

4.4 Bewertung der implementierten Algorithmen

Die implementierten Algorithmen sollen in diesem Abschnitt nach Präzision bzw. Recall- Werte bewertet werden. Wie oben schon erwähnt, erfolgt die Mustererkennung sehr unterschiedlich je nach Programmiersprache.

Muster	C++	Java
Kompositum	Probleme bei der Erkennung	
Besucher	Probleme bei der Erkennung	Wird problemlos erkannt

Aussagen über Recall- Werte können nicht getroffen werden, da es für die vorgegebenen Projekte keine Dokumentation gibt, welche die in den einzelnen Projekten implementierten Muster auflistet.

4.4.1 Kompositum

Bei genauer Betrachtung des Abschnitts

> für jede Methode k der Klasse j
> do
> ruft Methode k eine Methode der Klasse j auf, und folgt dem ein lokaler Methodenaufruf?

aus [Naum 01] kann man feststellen, dass die zweite Bedingung nicht korrekt ist. Es soll nach diesem Algorithmus überprüft werden, dass jede Methode des spezialisierten Kompositums eine Methode derselben Klasse (spezialisiertes Kompositum) aufruft. Wenn man sich die Abbildung 3.13 in [Naum 01] anschaut, insbesondere die Merkmale, die nicht in dem Muster vorhanden sein dürfen, entspricht diese Bedingung nicht dem, was in der Abbildung steht. Es ist durchaus möglich, dass es sich hier um einen Schreibfehler handelt. Mein Vorschlag wäre, Klasse j durch Klasse i zu ersetzen.

Aus dem so veränderten Algorithmus basiert meine Implementierung der Suche nach dem Kompositum-Muster.

Das Muster Kompositum wurde nur in Projekten gefunden, die in Together erstellt wurden. Der Grund dafür ist, dass im Modul eine Aggregation über das RwiProperty.AGGREGATION gefunden wird.
z.B.

nextLink.hasProperty(RwiProperty.AGGREGATION))

Dieses Property ergibt sich aus speziellen Javadoc-Eintragungen im Sourcecode wie z.B.

```
/**
 * @link aggregation
 * @supplierCardinality 1..*
 */
```

Wenn der Kommentar „@link aggregation“ nicht da ist, kann auch keine Aggregation gefunden werden. Normalerweise sind solche Kommentare in bestehendem Quellen, die zudem nicht mit Together erstellt wurden, nicht zu finden. Dasselbe gilt auch für die 1-N Beziehung die über den Eintrag „@supplierCardinality 1..*„ gewonnen wird.
Eine anderweitige Ermittlung der 1..*-Aggregation ist nur unter enormen zusätzlichen Aufwand denkbar. Dazu müsste in Java auf ein Attribut, welches eine Ableitung von der Collection-Klasse darstellt, getestet werden. In C++ wäre z.B. die Suche nach verketteten Listen oder Template-Typen notwendig.

4.4.2 Besucher

Bei der Implementierung des Algorithmus zur Suche nach dem Besucher-Muster muss man berücksichtigen, dass die Methode j mehrere Klassen als Parameter haben kann und dass ein Element nur eine von diesen Klassen ist. Dies ist nicht explizit in dem Algorithmus angegeben. Deshalb wurde der Algorithmus um eine Schleife erweitert, in der ALLE Parameter auf die Bedingung, dass der Parametertyp eine Klasse ist, überprüft.

```
für jede Klasse i (Besucher)
do
 für jede Methode j der Klasse i
 do
  besitzt Methode j anderen Klassen als Parameter?
  ja: für jede Klasse k der Klassenauflistung  (Element)
       do
      für jede Methode m der Klasse k
        do
   besitzt Methode m als Parameter die Klasse i?
   ja: gibt es in dieser Methode einen Aufruf der Methode j der Klasse i?
    ja: übergibt sich bei diesem Aufruf die Klasse k selbst?
       ja: Muster gefunden
        od
    od
 od
od
```

Das Muster Besucher wird in Java Projekten sehr schnell erkannt. Problematisch wird es jedoch in C++ Projekten, da das Klassendiagramm nur die Header-files abbildet. Um zu erkennen, ob sich einer Klasse bei dem Aufruf einer Methode übergibt, bzw. ob eine Klasse eine andere als Parameter enthält, ist es erforderlich, auf den Implementierung der Methoden (meistens *.cpp-Dateien) der gerade untersuchten Klasse zuzugreifen, und diesen zu analysieren.
Offen ist noch beim Besucher-Muster:
- ob und wie sich eine explizite Package-Information vor einer Klasse als Typ eines Methodenparameters auswirkt
- Wenn die besuchten Klassen bei der Erstellung des Quellcode des Projektes das auf Muster untersucht wird, importiert wurden, kann Together diese Klassen bei dem Reverse Engineering des Projektes nicht darstellen. In diesem Fall können die besuchten Klassen nicht im Modell selber als Klasse gefunden werden. (Als „Klassen“ gelten hier quasi nur diejenigen, die im Modell selber definiert sind).

5 Zusammenfassung

Die Suche nach Entwurfsmuster in UML-Modelle mit Together 6.0.1, die im Rahmen dieser Arbeit durchgeführt wurde , war mit vielen Problemen verbunden. Die in [Naum 019 beschrieben Algorithmen mussten ergänzt werden und für jedes Suchverfahren, das in diesen Algorithmen beschrieben ist, erfolgt eine spezielle Implementierung. Also gibt es kaum Methoden, die für zwei verschiedenen Muster aufgerufen werden können, und der Implementierungsaufwand kann leicht unterschätzt werden.

In Kapitel 2 wurden Entwurfsmuster kurz beschrieben, Kapitel 3 stellte die ***Open Api*** von Together dar, und die Möglichkeit ihrer Erweiterung mit Modulen. Kapitel 4 beschrieb die eigene Entwicklung des Moduls, die Implementierung der Suchalgorithmen, die Ausführung des Moduls auf eigenen und vorgegebenen Projekten. Die dabei aufgetretenen Probleme wurden auch dokumentiert.

6 Begriffe

API	Application Programming Interface
Recall	Verhältnis zwischen den die in dem bearbeitetem Modell gefundenen Muster und den tatsächlich in diesem Modell vorhandenen Mustern.
UML	Unified Modelling Language
Tcl, Jpython	Programmiersprachen
OMT	Object Modeling Technique

7 Literatur

[Balzert98] Helmut Balzert: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 1998.

[Gamma+96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Entwurfsmuster – Elemente wiederverwendbarer Software.* Addison-Wesley, 1996.

[Hel03] Jens Helbig, *Reverse Engineering von Entwurfsmustern*. Studienarbeit TU-Ilmenau, 2003.

[Mül97] Bernd Müller. *Reengineering. Eine Einführung.* B.G. Teubner Stuttgart 1997.

[Naum 01] Sebastian Neumann, *Reverse Engineering von Entwurfsmustern*. Diplomarbeit TU-Ilmenau, 2001.

[openAPI] Dokumentation der Open API in Together 6.0.1

8 Abbildungsverzeichnis

9 Anhang

9.1 Quellcode des Moduls

Link auf die Datei MusterVero.java

9.2 JavaDoc des Moduls

Link auf JavaDoc, generiert von Together